R++: Adding Path-Based Rules to C++

James M. Crawford
i2 Technologies

Daniel Dvorak
Jet Propulsion Laboratory

Diane J. Litman
AT&T Labs - Research

Anil Mishra
AT&T Network and Computing Services

Peter F. Patel-Schneider
AT&T Labs - Research

Abstract

Object-oriented languages and rule-based languages offer two distinct and useful programming abstractions. However, previous attempts to integrate data-driven rules into object-oriented languages have typically achieved an uneasy union at best. R++ is a new, closer integration of the rule-based and object-oriented paradigms that extends C++ with a single programming construct, the *path-based* rule, as a new kind of class member. Path-based rules—data-driven rules that are restricted to follow pointers between objects—are like "automatic methods" that are triggered by changes to the objects they monitor. Path-based rules provide a useful level of abstraction that encourage a more declarative style of programming, and are valuable in object-oriented designs as a means of modeling dynamic collections of interdependent objects. Unlike more traditional pattern-matching rules, path-based rules are not at odds with the object-oriented paradigm, and offer performance advantages for many natural applications.

Key Words: Object-Oriented Programming; Rule-Based Programming; C++

Contact: Diane J. Litman, room A-211
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932-0971

email: diane@research.att.com
phone: (973) 360-8314
fax: (973) 360-8970

Introduction

One of the purposes of a programming language is to provide the right level of abstraction, freeing programmers from irrelevant implementation details and allowing them to work at a more natural level for solving problems. Object-oriented programming with its rich modeling capability has taken a large step away from details, towards more direct problem-solving. In particular, the concepts of data abstraction and inheritance supported in object-oriented languages have simplified the design and implementation of sophisticated applications that model complex domains. However, as application complexity has increased, the need to model dynamic collections of interdependent objects has grown, and object-oriented programmers must still spend too much time on the low-level details of inter-object dependencies. For example, maintaining an invariant involving several related objects generally requires that similar-but-not-identical tests be inserted into a variety of places in the code. This can lead to errors of omission and logic by the programmer, but more importantly in the long run, the invariant is never explicitly stated, making code modification unnecessarily difficult and error-prone. We contend that certain tasks in an object-oriented design, e.g., those that maintain model integrity, should be performed *automatically*, without explicit procedural invocation. This suggests that object-oriented languages support another abstraction, namely data-driven computation.

Historically, data-driven computation has been supported in forward-chaining rule-based languages. In such languages, e.g., OPS5 [1], a rule is a statement composed of a condition and an action specifying what to do when some data satisfies the specified condition. Whenever a portion of the data changes, the rules whose conditions involve that data are examined, and if a rule's condition is satisfied on that data, its action is performed. The action may of course modify the data and therefore trigger other rules. Data-driven rule languages, however, have typically not supported data abstraction and other features of object-oriented languages, so their ability to model complex systems is limited.

The idea of integrating rules into object-oriented languages has recently been gaining currency [2], but there are a number of obvious, and not-so-obvious, impediments to such an integration. For example, production rule systems (such as most versions of OPS5) have historically been written in LISP rather than in a major object-oriented language. Further, many rule systems (such as ART-IM [3]) operate on their own data ("working memory elements") rather than the objects of their base language, thus requiring applications to move information between working memory and object storage. Some rule systems (such as CERS [4]) *do* operate directly on objects of the base language, but require special calls to bring objects to the attention of the rule system, thus requiring facilities *outside* of the object-oriented paradigm to control rules. Among object-oriented production-rule systems (including ART-IM and CERS as well as ILOG Rules [5], RAL/C++ [6], and RETE++ [7]), rules exist outside the object system, and thus are not organized in the same way as classes and objects. Rules are also often the main programming construct in these systems. Finally, even when rules operate directly on objects, they still do not act like the rest of the system. In particular, rules in all the above systems can form joins between unrelated objects, thus allowing programmers to ignore relationships designed into object-oriented domain models.

To provide data-driven computation in a form that can be readily applied by mainstream object-oriented programmers with little additional training, a seamless integration of rules, objects, and procedural code is needed. In particular, we contend that any rule construct should be designed as a natural extension of a commonly-used object-oriented language, with rules made subservient to the object-oriented paradigm. There should be little new syntax required in the rules, and the rule semantics should fit well within the object-oriented paradigm. Furthermore, this "small" addition of rules should provide "big" benefits in terms of new functionality.

Accordingly, we have designed and implemented R++, a small extension to C++ [8] that adds a single new programming construct: the rule. Rules in R++ are a new kind of a rule — a *path-based* rule—designed as a new kind of class member, conceptually like an "automatic method". Path-based rules in R++ have syntax and semantics similar to C++ member functions: they are declared inside classes, they operate directly on C++ objects, they respect the public/protected/private access controls of C++, they are inherited by derived classes, and they are activated with an implicit `this` argument. Unlike member functions, however, path-based rules are triggered automatically rather than explicitly called.

Path-based rules are so named because they strictly follow a path of inter-object pointers, from '`this`' object to related objects, giving them no more access capabilities than member functions, so they cannot bypass the relationships designed into a domain model. Because of their conformance to the C++ model, path-based rules are easy to learn, enabling C++ programmers to apply data-driven computation for many tasks, including enforcing invariants, checking constraints, expressing policies, and reacting to important states. The extra level of abstraction provided by rules results in increased C++ programmer productivity. R++ programs are shorter, contain fewer similar-but-not-identical pieces of code, and are more modular than C++ programs. In addition, the path-based rules of R++ are more cleanly integrated with the object-oriented paradigm than standard OPS5-style production rules. Finally, path-based rules yield a performance advantage over OPS5-style rules, as the algorithm that processes path-based rules is simpler and typically faster than the standard algorithm that processes production rules.

The Need for Rules

Many object-oriented applications depend on a domain model that is kept up-to-date and consistent as inputs/events are processed. Seemingly small tasks needed to maintain model integrity, such as enforcing invariants and detecting constraint violations, must be performed whenever updates are made to objects. The conventional way to do this is to add integrity-maintenance logic to the constructors and member functions that update the model, but this approach becomes error-prone for constraints and invariants involving two or more objects. In this section we concretely illustrate this point, by comparing two approaches to invariant maintenance. The first approach uses an object-oriented language (namely C++), while the second approach uses the path-based rules of R++.

Consider a simple model of employees with the multiple-object invariant that a manager must

```
class Manager;

class Employee {
friend Manager;
public:    virtual void set_salary(int);
           ~Employee();
protected: int salary;
           Set_of_p<Manager> staff_inverse;
           void maintainManagerSalary();
};

class Manager : public Employee {
friend Employee;
public:    virtual void set_salary(int);
           void insert_staff(Employee*);
           void remove_staff(Employee*);
           ~Manager();
protected: Set_of_p<Employee> staff;
private:   void maintainSalary();
};
```

Figure 1: Employee-Manager application in C++ (classes).

make as much as each employee reporting to him or her. Assume that this invariant is enforced by raising the manager's salary whenever necessary. Typical C++ code to support this type of reasoning is shown in Figures 1 and 2. Note the amount of code that has to be written to support this model. First, bookkeeping code is needed, such as code that manages backpointers from an employee to managers.[1] Obviously, code to enforce the invariant is also needed. However, due to the dependency between multiple objects, this code must be added in multiple classes, each time slightly different, yielding opportunities for programmer omissions and errors. Triggering the code to enforce the invariant in exactly all the appropriate places is also a problem. Too few calls results in incorrect behavior and too many calls results in inefficiency. In addition, Figures 1 and 2 show only the code to implement the dependency, not other code, such as constructors. This other code would also need to be modified to trigger dependency processing. Finally, since the invariant is never declaratively represented, it is unnecessarily difficult to understand and modify the code. In sum, while object-oriented languages facilitate the organization of classes, they do not provide

[1]It would be possible to hide such backpointers by encapsulating them in specially coded base classes with limited public interfaces. Even so, these internal classes would have to be carefully coded and the differences between them would make useful generic versions exceedingly difficult to provide.

```
void Employee::set_salary(int newSalary) {
  salary = newSalary;
  Employee::maintainManagerSalary();
}

void Employee::maintainManagerSalary() {
  Set_of_piter<Manager> iter(staff_inverse);
  for ( Manager * manager = iter.next(); manager; manager = iter.next() ) {
    if ( salary > manager->salary ) { manager->set_salary(salary); }
  }
}

void Manager::set_salary(int newSalary) {
  salary = newSalary;
  maintainSalary();
  maintainManagerSalary();
}

void Manager::maintainSalary() {
  Set_of_piter<Employee> iter(staff);
  for ( Employee* emp = iter.next(); emp; emp = iter.next() ) {
    if ( emp->salary > salary ) { set_salary(emp->salary); }
  }
}

void Manager::insert_staff(Employee* newEmployee) {
  staff.insert(newEmployee);
  newEmployee->staff_inverse.insert(this);
  maintainSalary();
}

void Manager::remove_staff(Employee* oldEmployee) {
  staff.remove(oldEmployee);
  oldEmployee->staff_inverse.remove(this);
}

Employee::~Employee() {
  Set_of_piter<Manager> iter(staff_inverse);
  for ( Manager* manager = iter.next(); manager; manager = iter.next() ) {
    manager->staff.remove(this);
  }
}

Manager::~Manager() {
  Set_of_piter<Employee> iter(staff);
  for ( Employee* emp = iter.next(); emp; emp = iter.next() ) {
    emp->staff_inverse.remove(this);
  }
}
```

Figure 2: Employee-Manager application in C++ (functions).

```
class Employee {
friend Manager;
public: Employee(int salary);
protected:  monitored int salary;
};

class Manager : public Employee {
rule maintainSalary;
public: Manager(int salary);
protected:  monitored Set_of_p<Employee> staff;
};

rule Manager::maintainSalary {
   Employee * emp @ staff   &&
   emp->salary > salary
=> set_salary(emp->salary);
}

Employee::Employee(int sal) : salary(sal) { }
Manager::Manager(int sal) : Employee(sal) { }
```

Figure 3: Employee-Manager application in R++.

support for expressing and maintaining certain kinds of dependencies between objects.

R++ extends C++ with data-driven, path-based rules. Such rules allow programmers to directly and declaratively represent dependencies between related objects, without worrying about how to do the bookkeeping required, how to divide dependencies between various classes and objects, or how to trigger the processing required to implement dependencies. Figure 3 shows typical R++ code to handle the employee-manager example discussed above.[2] In particular, the multiple-object invariant is now expressed using the single R++ rule `maintainSalary`. This rule is named and declared within the class `Manager`, as a new kind of class member. The rule is defined somewhat like a member function, with an "if" part (the *condition*) and a "then" part (the *action*) separated by an arrow "=>". The rule's condition contains a "branch binding", whereby the rule branches on every member of the set `staff`. (Rule syntax and semantics will be detailed below.) Whenever the condition becomes true (any employee of the manager has a larger salary than the manager), the action is executed (the manager is assigned the employee's salary). Finally, the rule has no arguments and no return type since it cannot be called. Rather, the rule is triggered automatically,

[2]A complete R++ program is presented in the Appendix.

whenever a new instance of `Manager` is created, or whenever the value of a `monitored` rule-affecting data member (`salary` or `staff`) is changed. R++ automatically provides *modifier functions* (e.g., `set_salary`) for modifying these monitored data members, as will be detailed below. Since the rule is active on all objects that belong to `Manager` and responds to all changes that might cause its condition to be successfully evaluated, it enforces an invariant associated with the class.

Note that R++ takes care of the details of creating and maintaining bookkeeping backpointers, turning the rule into multiple C++ member functions that run under a variety of conditions, and triggering the rule in response to relevant occurrences. With more complicated rules R++ has to do even more—including keeping track of rules whose conditions are partially satisfied—tasks that are difficult to code correctly in C++. In addition, while the C++ code was incomplete, in Figure 3 no further code (such as constructors) is needed to trigger dependency processing. In sum, using data-driven rules for tasks such as maintaining model integrity helps eliminate programming errors and increase modifiability because: 1) a dependency can be expressed declaratively in *one* rule rather than in several functions that depend on perspective, and 2) rules are guaranteed to be evaluated *every* time a relevant change is made to the object model, without explicit procedural invocation.

Roles for Rules

Rules have many potential roles within the object-oriented paradigm. As illustrated in the previous section, rules can be used to enforce invariants (things that should always be true). Rules can also be used to express policies, detect constraint violations, model relations, react to events and states, propagate information, update dependent objects, and even remove dangling pointers. The benefit, in all cases, is that the conditions of rules are expressed declaratively and are applied automatically as monitored data changes, without any burden of explicit procedural control. This section further motivates the need for rules, as well as continues an informal introduction of R++

through examples, by concretely illustrating some of the roles that rules can play in object-oriented designs.

The first rule[3] in Figure 4 shows how R++ can be used to express a company's retirement policy, namely, if a person has reached age 65 and has not yet retired, then the retirement process should be initiated. It is the simplest kind of rule in R++, as it only monitors the state of a single object. The rule is evaluated whenever a new instance of `Employee` is created and whenever an existing `Employee`'s `age` or `status` is changed. Since rules are inherited by derived classes, this rule applies equally to instances of `Manager`. Thus the rule monitors all kinds of employees and reacts when its condition is satisfied.

R++ can also be used to detect constraint violations (things that should never be true). Detecting and reporting a constraint violation is a simple variation of enforcing an invariant (recall Figure 3), where instead of enforcing the invariant, the rule action reports the violation. The second rule in Figure 4 illustrates a constraint involving two objects (in general, a condition may involve an arbitrary number of objects), and detects when the delegate of an employee on vacation is also on vacation. Literally, the rule says that if '`this`' employee is on vacation and he/she has a delegate who is also on vacation, then output a message. The explicit use of '`this`' in the rule's condition, though not necessary, demonstrates that path-based rules, like member functions, are invoked with an implicit '`this`' pointer. Related objects are accessed by first binding a rule variable to the address of the object, as shown in `Employee *d = delegate`. Note that the related object is reachable from the first object (the implicit `this` object) using an access path (`this->delegate`). This rule detects violations of the vacation-coverage policy; it is triggered every time an employee's mode or delegate changes, or whenever a new instance of `Employee` is created. Although the action part of this rule is a simple output statement, any valid C++ statements can appear here.

In addition, R++ rules can be used to maintain relationships between objects in a model, such

[3] The integrity rules of some active database systems look similar to R++ rules. The important differences between R++ and work in databases will be discussed in the section on related work.

```
                                          class Employee {
                                          protected:
                                              char *name;
                                              monitored int age;
                                              monitored Status status;
                                              monitored Mode mode;
                                              monitored Manager *boss;
                                              monitored Employee *delegate;      rule
                                              ...                                names
rule declarations                             rule retirement_policy;
(rules are class members)                     rule vacation_coverage;
                                          };

A derived class inherits the              class Manager : public Employee {
rules of its base classes                 protected:
                                              monitored Set_of_p<Employee> staff;
                                              ...
                                              rule staff_inverse;
                                          };

start of rule definition                  rule Employee::retirement_policy {
                                              age >= 65 &&
rule condition                                status != retired
                                          =>
rule action                                   initiate_retirement();
                                          }

                                          rule Employee::vacation_coverage {
If this employee is on vacation...            this->mode == vacation &&
...and he/she has a delegate...               Employee * d = delegate &&
...who is also on vacation...                 d->mode == vacation
                                          =>
...then output a message.                     cout << "No coverage for " ...;
                                          }

                                          rule Manager::staff_inverse {
For each staff member of this manager...      Employee * emp @ staff
                                          =>
...'this' manager is his/her boss             emp->set_boss(this);
                                          }
```

Figure 4: Example R++ rules that express policies, detect constraint violations, and model relationships.

as inverses and transitive closures. The third rule in Figure 4 maintains an inverse relationship between a manager and the members of his/her staff. Specifically, for every staff member `emp` of manager `this`, `emp`'s boss is `this`. This rule's condition contains a "branch binding" whereby the rule branches on every member of a set or list[4], thus enabling rules to work with sets and lists of objects as easily as with single objects. The addition of a new member to a set or list creates a new branch for the rule to test.

More broadly, rules can act as "monitors" of a model's state, observing changes to object memory and reacting when a condition is satisfied. Data-driven rules are well-suited to all sorts of monitoring tasks because they can express situation/action directives without complicating an application's procedural logic. Applications that model complex relations and information flow can use rules to propagate information within a model. Applications that monitor a physical system can use rules to detect and react to critical states. Applications that apply domain knowledge can use rules to express business policies, engineering guidelines, and situation/action heuristics and apply them automatically when each situation arises. Applications that are primarily reactive can use rules to express state transitions. As a software engineering aid, rules can be used as a kind of non-procedural exception mechanism for detecting and handling illegal states. We have used rules in many of these roles in an alarm monitoring system for telephone switches [11, 12, 13], as will be discussed below and illustrated in the Appendix.

R++: Rules in an Object-Oriented Language

A key design goal of R++ was to add data-driven rules as a small, easy-to-understand extension of the object-oriented paradigm. The following sections discuss the key aspects of the design of R++, and illustrate how R++ rules were made to look and act as much like the rest of C++ as possible:

- R++ rule syntax is similar to C++ syntax.

[4] Rules in R++ deal with *pointers* to other objects, so a set or list is represented as a set of pointers or a list of pointers. The code examples use classes `Set_of_p` and `List_of_p` from *C++ Standard Components* [9]. R++ code can also use classes `set` and `list` from the *Standard Template Library* [10].

```
rule-declaration ::= rule identifier ; |
                     friend rule class-name::identifier ;
rule-definition ::= rule class-name::identifier { condition => action }
action ::= statement+
condition ::= conjunct |
              conjunct && condition
conjunct ::= boolean-expression |
             simple-binding |
             branch-binding |
             quantified-test
simple-binding ::= class-name * identifier = expression
branch-binding ::= class-name * identifier @ field |
                   class-name * identifier @ variable->field
quantified-test ::= all iterator-binding { condition } |
                    exists iterator-binding { condition }
iterator-binding ::= branch-binding
```

Figure 5: R++ rule syntax (an extension to C++ syntax).

- R++ rules are integrated into C++ classes as a new kind of class member.
- R++ rules are path-based: like members functions, path-based rules traverse only the inter-object paths designed into a domain model.
- R++ rule processing is triggered by calls to constructors and modifier functions.
- R++ rules are controlled by natural mechanisms of the object-oriented paradigm.

Rule Syntax and Semantics

The R++ rule syntax is defined in Figure 5. Rules are declared and defined in a manner similar to other C++ class members, with the `rule` keyword distinguishing member rules from data members and member functions. A rule's name is simply a label for the rule. The significance of the name is only that another rule with the same name in a derived class overrides the rule from the base class. (An example of rule overriding is given in the Appendix.) Note that rules can only be associated with non-primitive (class) types.

A rule is like an if–then statement consisting of a an “if” part (the *condition*) and a “then” part (the *action*). An action consists of ordinary C++ statements. A condition is a sequence of C++ boolean expressions[5] interspersed with R++ variable bindings and quantified tests. A variable binding declares and binds a pointer variable to the address of a related object, enabling a rule condition to reference data members of related objects (subject to C++ access controls). The boolean expressions and the expressions in the variable bindings can, of course, use variables bound earlier, just as in C++. Universally and existentially-quantified tests iterate over a set or list applying the same test to each element; the iterator variable is defined only within the scope of the quantified test.

There are two kinds of variable bindings. A *simple binding*, for example `Employee * d = delegate` in the second rule of Figure 4, looks just like a C++ variable definition, and simply binds a variable to the value of an expression, succeeding if that value is non-null. The example binding declares a variable of type `Employee *` and sets it to the value of `this->delegate`, succeeding only if the value is non-null.

A *branch binding*, for example `Employee * emp @ staff` in Figures 3 and 4, is similar to a C++ variable definition but uses “`@`” instead of “=” to bind a pointer variable to elements of a set or list of pointer values. The “`@`” should be read as “at” or “in”, e.g., “for each employee in staff do . . . ”. The binding succeeds for each element of the set or list, causing the rule to branch on each element.

There are two kinds of *quantified tests*.[6] Universally-quantified tests succeed if the test holds for all elements of the quantified set or list. Existentially-quantified tests succeed if there exists at least one element of the set or list for which the test holds. An example of a quantified test will be discussed below.

[5]What we mean here are both boolean expressions and expressions that can be treated as boolean, e.g., `int` or pointer.

[6]Quantified tests are not implemented in R++ Version 1.1.

A condition is evaluated in the obvious way, succeeding for those successful bindings that make the boolean expressions and quantified tests evaluate to “true”. When a rule’s condition evaluates to “true”, its action is executed. The action consists of a sequence of C++ statements, and may use any variables bound in the condition. It is important to realize that rules in R++ are *procedural rules*, not statements of logic. Thus, although a rule’s action is executed when its condition evaluates to true, the action is not “retracted” when the condition ceases to be true.

Rules as Class Members

One natural way to fit rules into the existing constructs of an object-oriented language is to associate rules with classes, just as data and functions are associated with classes, and have the rules work directly on objects that are instances of the class. In effect, we view a rule as a “member rule” of a class, declared within the class with a class-specific name. Just as data members define the *representation* of a class and member functions define the *interface* of a class, rules define the *automatic behavior* of a class.

Rules share several characteristics with member functions. First, since rules are class members and thus are defined by class *designers* (not class *users*), they are given access to the visible members of their immediate class, whether public, protected, or private. Second, rules respect the access controls of C++, so a rule in one class cannot access private members of another class unless the rule (or its class) is declared a friend of the other class. Third, rules of a base class are inherited in the sense that they are applied (automatically) to instances of derived classes. Finally, a rule in a derived class overrides a rule of the same name from a base class. Rules differ from member functions in that they cannot be called, so they have no argument list, no return type, and no public/protected/private access specification.

R++ rules view an object as a collection of data-members; rules directly monitor specified data members and are triggered by changes to their values. This view is distinct from the view that an object should be defined only by its external behavior (i.e., its response to messages). However R++ rules can still be valuable in applications where modularity is critical and objects are defined by their behavior. In such applications, the class designer can encapsulate the data and define the object's behavior through a public interface while still using rules as part of the internal implementation. In other applications where rules may be written at the "user" level, such as in traditional expert systems, application programmers can define (or derive) their own classes with their own rules expressing inferences on the data.

The association of rules with classes provides a type for the root object (the '`this`' object) of a rule, namely the class of which the rule is a member. 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 that class and its base classes. The only way to remove an object from consideration by rules is to destroy the object. Thus, a rule is always active on all objects of its class (including objects of derived classes).

Path-Based Rules

Rules in R++ are termed *path-based* because their conditions strictly traverse inter-object paths starting from a root object (the '`this`' pointer variable).[7] Such rules have the same access capabilities as member functions. While this property may seem unsurprising and even *expected* by object-oriented programmers, it is not present (to our knowledge) in any other rule language save Algernon [14, 15], the forebear of R++. Other rule languages support pattern-matching rules that can perform arbitrary joins between unrelated objects. In our view this violates the locality of reference designed into a domain model, making program behavior harder to predict and understand.[8]

[7]Note that global variables and static data members can be used in rule conditions, but changes to those variables will *not* trigger rules. For example, changes to static data members do not trigger rules because there is no well-defined `this` value for the rule to use. A static data member appearing in a rule condition should thus serve as a shared constant or as a shared variable whose changes need not trigger any rules.

[8]Note, however, that the path-based restriction of R++ means that procedural code, rather than rules, might sometimes be needed for legacy schemas. This is because if a rule needs to compare two objects that are not already related in the schema, R++ would require that those object relations (and any corresponding rules) be added and the system recompiled. An approach that would somewhat alleviate this schema evolution problem would be to allow R++ rules to branch on general expressions, not just on data in objects. This approach is discussed further in the section on future work.

Compared to the "inference engines" of pattern-matching rule languages, the algorithm for processing path-based rules is simpler since pointer-following is much less complex than join operations—so much so that rules in R++ simply translate into C++ member functions that are executed whenever a relevant occurrence is made, without the need for a run-time interpreter. The R++ translator (see the implementation section) compiles rules into member functions, adds bookkeeping data members to rule-affecting classes, defines modifier functions for monitored data members, and arranges rule-triggering by placing calls to rule functions inside constructors and modifier functions. In particular, for each data member monitored by a rule, the R++ translator adds a rule-specific data member that, during execution, will contain back-pointers to root objects (with the exception that data members of root objects themselves do not have back-pointers). When a relevant change is made to a data member of a non-root object, each triggered rule will follow each of its back-pointers to a root object and then evaluate its condition on all paths that can be formed from that root and that pass through the changed object. Full details regarding the R++ algorithm for processing path-based rules are presented elsewhere [16].

The algorithm for processing path-based rules is often more efficient than the typical RETE-based inference engine of pattern-matching rule languages. Given n objects, r rules, and v variables in the largest rule condition, the worst-case heap-storage requirement for R++ is $O(rn^2)$ whereas the RETE algorithm [17] uses space $O(rn^v)$. Note, however, that worst-case analyses have not been particularly good indicators of actual performance for matching algorithms, as the worst case is rarely achieved in practice (e.g., [18]). Below we present empirical results (using both synthesized rules and real-world application programs) that further support the efficiency of path-based rules.

Although reduction in complexity was a principle objective of the research upon which path-based rules was founded [14, 15], an equally important aspect of path-based rules for R++ is that it allows rules to be viewed as class members, and thus integrated cleanly into the object-oriented framework.

Rule Processing

There are three phases to rule processing: triggering, evaluation, and execution. A rule is *triggered* when object memory is changed in a way that might satisfy the rule's condition. One form of triggering—called *relevant construction*—occurs when a new instance of class c is created, triggering all member rules of class c. The other form of triggering—called *relevant change*—occurs when the value of a "monitored" (rule-affecting) data member is changed, triggering the rules that monitor that data member. To ensure proper rule triggering, relevant changes must be made through specially-named *modifier functions*. For a single-valued data member named x the modifier function is named `set_x`; for a set of pointers the modifier functions are named `insert_x` and `remove_x`; and for a list of pointers the modifier functions are named `get_x` (get from head), `put_x` (put to tail), `unget_x` (put to head), and `unput_x` (get from tail). Modifier functions for rule-affecting data members are defined automatically by the R++ translator, unless already defined by the user. It is generally considered good C++ programming style to use such modifier functions.[9] Figure 7 illustrates the rule triggering that results from calls to constructors and modifier functions.

Following a triggering event a rule's condition is *evaluated*. If any conjunct of the condition evaluates to false, the rule abandons all paths following from the false conjunct. Since a rule may be triggered and evaluated many times before its condition is finally satisfied, rule conditions should be free of side-effects. When an event triggers more than one rule, a rule in a derived class is evaluated before a rule in a base class; otherwise, the order of evaluation is unspecified.

When a rule's condition evaluates to true, its action is *executed* on the data that satisfied the condition. Execution of the action—which consists of ordinary C++ statements—may trigger other rules, causing a "forward chaining" of rule executions.

As a change to a data member can cause more than one rule condition to evaluate to true, the

[9] We could have translated assignments to data members into calls to the modification functions, but instead chose a method that emphasizes data encapsulation. Our method, which does not require trapping of assignments, also allows us to avoid parsing all the C++ code.

action of a rule may be delayed from the time of the change that caused its condition to evaluate to true. R++ ensures that rules do not execute on “old” data nor execute more than once on changed data; it does this by saving a “time tag”—a value that varies monotonically with time—with each monitored data member and element each time a relevant change is made. For example, if a relevant change 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 rule was triggered twice in this scenario, it will execute at most once on the changed data because the deferred rule evaluation will see that the data changed again *after* it was triggered.

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. When no rules can run, control returns to the modifier function or constructor call that triggered rule processing.

Rule Control

Data-driven computation is valuable as a programming abstraction precisely because it is non-procedural, but most applications require some sequentiality. Consequently, rule-based programming languages typically offer various mechanisms to control the order in which rules are applied, mechanisms such as user-definable conflict-resolution procedures, rule priorities, rule sets, and context stacks [19]. None of these mechanisms are natural to object-oriented languages and they all complicate language semantics.

R++ does not require special rule-control mechanisms for two reasons. First, as an extension to C++, the language already contains familiar procedural constructs for performing sequential tasks. Since rules and procedures operate on the same objects, it is natural to express the sequential logic in procedural code and reserve the data-driven tasks for rules. Second, where sequencing among rules is desired, the object-oriented paradigm provides natural mechanisms for controlling path-based rules. Specifically, path-based rules are class members, so rules are grouped by class with rules in a derived class considered *before* rules in a base class. This sequencing allows rules on more specific types to defeat rules on more general types. Within a class, sequencing can be controlled by the class designer by adding a private “control” data member whose value is tested in rule conditions and modified in rule actions. Since there is no complex priority scheme, there is no agenda that collects all rule activations[10] that are waiting to run. Rule activations are simply run in a depth-first fashion, as soon after they are triggered 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.

The integration of rules into the object-oriented paradigm obviates the need for other kinds of control mechanisms. Since 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. Since construction and destruction of objects brings objects into and out of consideration by rules, no enabling/disabling functions or object containers are needed. Since triggering is provided through modifier functions, no special “rule notification” mechanism is needed.

Additional Examples Illustrating R++

To further illustrate R++, Figure 6 extends the Employee–Manager example of Figure 4 to include a `Project` class, where each employee has a set of assigned projects. Rule `Manager::idle_employee` detects when any member of a manager’s staff has no active projects (or no projects at all). The rule’s universally-quantified test (`all ...`) returns true if all of an employee’s projects are “on hold”. Since this rule is a member of class `Manager` but requires access to a data member of class `Project`, it is declared as a `friend rule` of class `Project`.

Figure 7 further extends Figure 6 to illustrate how rules are triggered by construction of

[10] A rule activation is a rule plus a set of bindings of the variables in the rule’s condition to objects in memory.

```
class Employee {
protected:
    monitored Set_of_p<Project> projects;
    ...
};

class Manager : public Employee {
    ...
    rule idle_employee;
};

class Project {
private:
    char *name;
    monitored State state;
    ...
    friend rule Manager::idle_employee;
};

rule Manager::idle_employee {
    Employee *emp @ staff &&
    all Project *proj @ emp->projects {
      proj->state == on_hold }
=>
    ...
}
```

As “protected”, this data member is accessible to rules of derived classes, such as Manager rules.

As a friend, this rule gains access to private members of Project

For each staffer of this manager. . .

. . . if all his/her projects. . .

. . . are “on hold”. . .

. . . then do something.

Figure 6: Further examples of R++ illustrating friend rules and quantified tests.

R++-defined Modifier Functions:

```
set_age, set_status, set_state
set_mode, set_delegate, set_boss
insert_staff, remove_staff
insert_projects, remove_projects
```

Rule Triggering:
retirement_policy

Rule-affecting ("monitored") data members must be modified only via specially-named modifier functions

Rule Triggering:
staff_inverse and idle_employee

Rule Triggering:
all Employee rules
all Manager and Employee rules

```
class Employee {
protected:
    int salary;
    ...
public:
    void increment_age(void);
};

void Employee::increment_age(void) {
    set_age(age + 1);
}

class Manager : public Employee {
private:
    int budget;
    ...
public:
    void add_staff(Employee *);
};

void Manager::add_staff(Employee *e) {
    budget += e->salary;
    insert_staff(e);
}

main() {
    Employee *e1 = new Employee(...);
    Manager *m1 = new Manager(...);
    ...
    e1->increment_age();
    ...
    m1->add_staff(e1);
    ...
}
```

Figure 7: Rules in R++ are triggered by constructors and by calls to R++-defined modifier functions.

and change to objects. As shown in `main`, construction of an object immediately triggers all rules of its class, including rules of base classes, so construction of a `Manager` object triggers not only `Manager` rules but also `Employee` rules. Likewise, changes to the values of rule-affecting data members, made via R++'s specially-named modifier functions, trigger the rules that monitor those data members. (Recall that R++ automatically defines modifier functions for all `monitored` data members.[11]) For example, when the main program calls the user-defined public member function `increment_age`, it in turn calls the R++-defined protected modifier function `set_age` which triggers all rules that monitor the `age` data member (in this case, just rule `Employee::retirement_policy`). Similarly, when the main program calls the user-defined public member function `add_staff`, it in turn calls the R++-defined protected modifier function `insert_staff` which triggers all rules that monitor the `staff` data member (in this case, rule `Manager::staff_inverse` and rule `Manager::idle_employee`).

Each R++-defined modifier function (e.g., `set_age`) has the same access specification as its corresponding data member. This ensures that the introduction of rules into a class has no effect on information-hiding. The only requirement on the class designer is that other member functions that modify the values of rule-affecting data members must use the specially-named modifier functions to make such modifications. Clients of a class continue to use the class through its public interface, whether or not the class has rules.

Implementation Issues

R++ is implemented by means of an R++ to C++ translator, itself written using C++, along with versions of *lex* and *yacc*.[12] C++ is not an ideal language to extend. We would have preferred to make R++ largely a library of classes, thus requiring fewer extensions to the base language,

[11] Note that users are not *required* to monitor non rule-affecting data-members (e.g. `salary` and `budget`, given the rules in Figures 4, 6, and 7). However, users may choose to optionally monitor such data members, for example, if they expect to add rules that will make the data-members rule-affecting.

[12] The approach we have taken is similar to the approach used in Ode [20], an object-oriented database management system that incorporates O++ [21], an extension to C++.

but C++ does not support the kind of control over object creation and 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, which would significantly complicate the translator.

In order to render the notification problem tractable and avoid parsing all the C++ code in R++ programs, we have placed several restrictions on proper code. First, 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. Second, changes to data members monitored by rules must be made via the specially-named modifier functions, as discussed above. Furthermore, functions should not be used in the condition of rules, as R++ cannot, in general, analyze the data used inside a function. This is because relevant change is defined as change to the data mentioned directly in the condition of a rule, including arguments to functions, but not including change to data used in functions called in the rule condition. 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++.

Another limitation of R++ concerns the use of legacy classes. There are two aspects to adding rules to C++ legacy classes—changing behavior by specialization and adding rules. R++ does not work well in either of these cases. Even in C++, changing the behavior of a legacy class by specialization assumes that the base class was designed to be extended in the desired direction. Thus, any designer of classes which have rules needs to similarly take into account that classes can have their behavior specialized by rules in subclasses. Obviously, legacy classes have not been designed with this in mind. As for adding rules to legacy classes, R++ brings with it a programming paradigm, namely data-driven computation. In order to do a good design to get the most benefits out of R++, C++ classes should be thought out from a data-driven computation perspective. Most importantly however, R++ requires exclusive control of the data members being monitored. This is not possible to do with existing legacy classes in either of the above cases without parsing all the code, and without having the ability to modify the legacy class code. We felt that the benefits of being able to add rules on existing C++ classes did not justify these costs, especially since procedural code could always be used.

A final point regarding our approach is that it has resulted in increased compilation time. 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 large and uses several template classes; template instantiation is notoriously slow in some C++ compilers.

Empirical Comparisons with RETE Pattern-Matching and C++

In this section we present two types of evaluations of the performance of R++. First, we compare R++ to C5, a C-based superset of OPS5 that uses the RETE algorithm [17], by comparing rule-firing rates on a set of artificially constructed trees of increasing depth and breadth.[13] Second, we compare the resource needs of R++, C5, and C++, by coding two well-known examples (a benchmark expert system program and a standard active database example) in all three languages.

Rule Firing Rates on Trees of Varying Depth and Breadth

The performance of path-based rules relative to pattern-matching rules on trees with varying properties is shown in Table 1. Performance rates are given in rule firings per second on a Sun SPARCstation 10. In each performance measurement a tree 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

[13]RETE is the most commonly used incremental match algorithm in production systems [22]. Although other match algorithms have appeared in the literature [23, 18, 22], we could not find any public domain versions of OPS5 that used these algorithms. Thus, we have been forced to limit our comparisons to RETE.

path length	*branching factor*	*# of leaves*	R++ *rate for sets*	R++ *rate for lists*	*C5 rate*
0	NA	1	120000	120000	2083
1	1	1	25575	10591	1428
1	2	2	23831	10082	1428
1	4	4	23301	9198	1333
1	8	8	22310	7653	1391
1	16	16	21399	5793	1391
1	32	32	20164	3883	1411
1	64	64	19418	2340	1383
1	128	128	18078	1307	1383
2	1	1	13797	7710	1363
2	2	4	8065	5577	1333
2	4	16	4854	3196	1317
2	8	64	2849	1424	1280
2	16	256	1524	496	1024
2	32	1024	682	141	640
2	64	4096	309	38	244
2	128	16384	138	10	42
3	1	1	9362	5825	1200
3	2	8	3196	2788	1142
3	4	64	1024	728	1200
3	8	512	264	160	682
3	16	4096	57	27	107
3	32	32768	13	4	13

Table 1: Rule-firing rates for R++ and C5 on trees of varying depth (path length) and breadth (branching factor).

in the path. The tree is monitored by a single rule that follows a path from the root object to a leaf object.

Since a tree can be implemented in C++ with either lists or sets, the results for both are shown. The R++ results for sets are most comparable to the C5 results, since trees in C5 are inherently unordered (they arise from joins among working memory elements). If order *is* important, however, it's easy to obtain in R++ simply by using lists instead of sets, though there is a performance penalty because time-tag processing (see page 18) incurs more overhead with lists. In C5, and in other rule systems, it is much more difficult and time-consuming to process a tree in an ordered way.

To test the worst-case for R++, all triggering changes were made in leaf objects. As can be seen, R++ generally 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. As path-length and branching-factor increase, R++'s rule firing rate declines considerably because its current algorithm trades time for space. In practice, the number of such slower-performing rules can be reduced by tuning a model, for example, by connecting distant objects with shorter paths.

Resource Measurements for Two Application Programs

A Line-Labeling Benchmark

The performance of path-based rules relative to pattern-matching rules and to C++ on a well-known line-labeling application program is shown in Table 2. This benchmark program labels line drawings using Waltz's constraint propagation algorithm [24], and has previously been used to evaluate matching algorithms for OPS5-style systems [23, 18, 25, 22].

The C5 version of the benchmark is a slightly-modified version of a simple OPS5 program obtained from Daniel P. Miranker.[14] Because OPS5 and R++ have different rule languages, the benchmark had to be recoded from OPS5 to R++. The R++ version was derived from the OPS5 program by first creating C++ data structures for the OPS5 working memory elements, modifying them as appropriate to fit within the C++ object model. Then, R++ rules were written to mimic the OPS5 rules that naturally fit within the R++ model. The other OPS5 rules were implemented in C++ code. This resulted in R++ code that was a close match to the original OPS5 code, and not in particularly efficient R++ rules.

The initial OPS5 program had 33 rules, 18 of which were directly related to the core of the line labeling algorithm, 7 of which were control rules, and 8 of which were printing and setup rules. The 18 labeling rules had a total of 92 conditions and 34 actions. They were turned into

[14] The benchmark is available from ftp://ftp.cs.utexas.edu/ftp/ftp/pub/ops5-benchmark-suite, and also from the CMU Artificial Intelligence Repository.

9 R++ rules with a total of 42 conditions and 22 action lines of code. Usually, OPS5 rules that translate into R++ rules translate on a one-for-one basis, but because OPS5 has no sets, the OPS5 code represents sets as sequences, and rules that act on any member (or subset) of the set have to be duplicated in OPS5 but not in R++. Thus, there are twice as many OPS5 rules as R++ rules. The reason that there are slightly more lines of action code per rule in the R++ example is that iterating over a set takes about three lines of code in C++. In contrast, because the OPS5 rules are replicated, they can directly access the appropriate element. Finally, there are slightly less average conditions per rule in R++.

The C++ implementation is a recoding of the R++ implementation. The R++ rules were turned into C++ methods, and the classes were augmented to incorporate extra data member pointers to allow these methods to run as appropriate. The C++ implementation incorporates quite a number of optimizations over the R++ implementation, based on an analysis of the entire program. For example, since all rules have the same triggering conditions, only one version of each rule needs to be written, all rules can be triggered from a single virtual method, and only one set of pointers to root objects need be created. Since all rules can safely be run on stale data, data time tags are not necessary; since all rules can be run more than once on the same data, rule time tags are also not necessary. Since neither objects nor pointers between objects are removed, there is no need for any pointer management.

The R++ program has 490 lines of code, while the (optimized) C++ program has 622 lines of code. (The C++ program output by the R++ translator has 4804 lines of code.) Most of this code is essentially the same between the two programs; the portion that defines the rules is 75 lines of code in the R++ program and 201 lines in the C++ program. The interesting portions of the C5, R++ and C++ implementations of the line-labeling benchmark can be found in Reference [16].

Performance figures in Table 2 are given in both seconds and MB, and were obtained by running each version of the benchmark on a Sun SPARCstation 20. The first set of R++ and C++ figures are for the Standard Components version of sets [9]. The second set of figures are for an already existing hash-set implementation optimized for good storage consumption with very small sets.

input size	*C5*		*R++*				*C++*			
			Standard Sets		*Hash Sets*		*Standard Sets*		*Hash Sets*	
	secs	MB	secs	MB	secs	MB	secs	MB	secs	MB
12	166.0	1.0	5.0	1.3	1.0	0.7	3.4	0.7	0.6	0.5
25	737.0	1.7	15.0	2.1	2.5	1.1	13.1	0.9	1.6	0.7
37	1839.0	2.3	30.0	2.8	4.2	1.4	27.4	1.2	3.1	0.8
50	3752.0	2.9	53.0	3.6	6.4	1.6	49.1	1.5	4.9	0.9

Table 2: Resource consumption for C5, R++ and C++ on Waltz line labeling.

The sets in the benchmark are almost all of size 2 or 3.

As can be seen, R++ is much faster than C5 on this application. This is largely due to the fact that the rules are small-branching-factor rules with a path length of 1. Further, as the input size increases, R++'s speed advantage becomes even bigger. As for storage, R++ consumes somewhat more memory than C5 when a set implementation with a large overhead (i.e., the Standard Components implementation) is used. With a better set implementation, R++ consumes somewhat less storage than C5.

The R++ implementation is slower and uses more storage than the C++ implementation, as expected. However, even though this example admits quite a number of optimizations over the rule control mechanism used by R++, the C++ implementation still consumes roughly 60% to 70% of the resources of the R++ implementation. Most of this difference has to do with the fewer pointers used in the C++ implementation; examples that need more of the R++ features, such as rules with differing data used in their conditions or management of pointers, would show a much smaller differential.

The above measurements are insufficient to confidently generalize. Nevertheless, our results provide evidence that R++ is much faster than C5 on certain kinds of applications, and that R++ rules are similar in size and number to OPS5-style rules. R++ is somewhat slower than C++ code for the same task, but the C++ code is considerably larger, even when global conditions are used to optimize the C++ code and reduce the amount of bookkeeping it needs to do.

input size	*R++ (Standard Sets)* secs	KB	*C5* secs	KB	*C++ (Standard Sets)* secs	KB
4369	1.1	1220	13.5	1288	0.8	1032
14425	4.2	3664	57.5	3000	2.8	3160
33825	10.0	8132	233.7	6896	6.9	7396
65641	18.9	14368	1347.7	14644	14.7	12308

Table 3: Resource consumption for C5, R++, and C++ on Employee-Manager example.

An Employee-Manager Application

Table 3 shows the same types of performance figures as in the previous section, this time for R++, C++, and C5 programs that create and maintain a model where a manager must always make as much as each employee reporting to a manager that reports to him or her, enforced by raising the manager's salary whenever necessary. The usual (simpler) statement of this multiple object invariant is that a manager must make more than his or her direct reports, as shown in Figure 3 and as discussed in the object-oriented database community [26]. We have increased the complexity of the example to make it more interesting. The three programs have equivalent functionality and were coded by one of the authors of this paper, who is proficient in all three programming languages. In this experiment, however, the C++ program was not highly optimized. The R++, C++, and C5 programs have 45, 130, and 25 lines of code, respectively, again suggesting that R++ programs are comparable in size to OPS5-style programs, and smaller than C++ programs.

As can be seen from Table 3, R++ is again much faster than C5 on this application, especially as the input size increases. (All measurements are again from a Sun SPARCstation 20.) While R++ is slower than C++, the differences do not explode with input size. As for storage, R++ consumes somewhat more storage than C++, and both somewhat less (this time even using the standard set implementation) and somewhat more storage than C5, depending on input size.

As with the line-labeling application, this experiment again provides evidence that the use of rules yields R++ programs that are shorter than equivalent C++ programs. The trade-off is that R++ programs require more memory and take longer to execute than C++ programs. While R++

and C5 programs are fairly similar in size, unlike C5, the use of path-based rules prevents the R++ execution times from exploding compared to C++.

Experiences Using R++

R++ has been successfully used in two AT&T operations support systems, by a set of developers who were not involved in the design and implementation of R++. ANSWER–4ESS [13, 12] automates the surveillance and maintenance of the the 4ESS telephone switching system. (The 4ESS handles most AT&T domestic long-distance calls.) ANSWER–4ESS is currently in field trial, the final phase of a system before deployment. ANSWER–SD automates the surveillance and maintenance of the segmentation directory subsystem. The development phase of ANSWER–SD has been completed, and the system will shortly be put into field trial. In this section we detail one of these experiences using R++. In particular, we first describe the ANSWER–4ESS system, then summarize the major benefits and costs of using R++ for its implementation.

ANSWER–4ESS performs real-time monitoring of alarm and other informational messages coming from the various components of the 4ESS; it runs 24 hours a day, seven days a week. ANSWER–4ESS correlates alarm messages, ignores transient alarms, recognizes chronic problems, runs diagnostics, restores the network element, automates the trouble ticket creation and signals problems requiring human attention. The system takes input in the form of messages from the 4ESS, and produces output in the form of warning and action messages to field technicians. The system operates by building and maintaining an explicit model of the network element, then reasons with respect to this model.[15]

The ANSWER–4ESS alarm monitor is implemented in R++ as follows. The domain model of the 4ESS is represented in C++, and includes both the type hierarchy of components in the switch and a set of diagnostically-important functional and structural relationships between components. R++ rules reason from this device model to dynamically create relevant portions of the 4ESS from

[15] The alarm monitoring approach adopted in ANSWER–4ESS is based on the use of "affective relations" for device representation and reasoning [11].

alarm messages, maintain the integrity of the device model by enforcing invariants and detecting constraint violations, propagate alarm information through the model, determine root causes of alarms, and monitor for other conditions that require technicians to be alerted. The application has 228 rules spread over 19 classes. Note that because R++ rules are tightly integrated with the 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. A detailed description of ANSWER-4ESS and its R++ implementation can be found in Reference [13]. An extremely simple (but complete) R++ program giving a feel for the ANSWER–4ESS alarm monitor is presented in the Appendix.

ANSWER–4ESS was written to replace an existing legacy alarm monitor, built using a purely rule-based paradigm and implemented using the C5 language. The existing alarm monitor was hard to maintain, so the goal of the ANSWER–4ESS project was to redesign and reimplement the previous system using an object-oriented paradigm. The use of R++ rather than C++ for the implementation of ANSWER–4ESS has yielded two major benefits.[16] First, R++ was a cost-effective approach, as it resulted in a reduction of the total headcount needed for software development. According to the technical manager of ANSWER–4ESS, it would have taken approximately twice as many staff-days to obtain the same level of functionality and reliability if the system had been programmed directly in C++ [27]. Several factors contribute to this doubling of productivity:

> "Clearly, the fact that R++ relieves the programmer from dealing with tedious technicalities (which typically accounts for 50% of the total amount of C++ code in ANSWER), will result in cost-savings via a number of ways, including:
>
> - R++ improves the development cycle by providing a mechanism to write requirements in a higher level language, R++ and thus reduce the gap between

[16] In addition to being written in an object-oriented manner, ANSWER–4ESS differs from the previous system in that it runs on a new platform and has added functionality.

requirements and implementation.

- R++ facilitates easier maintenance due to the clarity, robustness and compactness of the object-model.
- R++ promotes software reuse through code generation.
- R++ promotes team development through uniform style of programming, and providing a common object-model on which a system-engineer, domain-experts, software developer, and system testers can work." [27]

Second, R++ yielded an important qualitative improvement with respect to the end-to-end process. R++ enabled system engineers and software developers to spend more time solving the real problem, rather than dealing with low-level artifacts of C++:

> "R++ encourages the developer to clearly separate the model's behavior from the model's infra-structure by using R++ rules. R++ rules relieve the programmer from writing the tedious behind-the-scenes book-keeping code to deal with the vast amount of technicalities necessary to manage the (technical) relationships between the model's behavior and its infra-structure. Instead, by just writing R++ rules, the programmer is free to focus exclusively on the real problem, namely specifying the functional relationships between the model's behavior and its infra-structure." [27]

Using R++ to implement ANSWER–4ESS has also provided us with feedback regarding the most significant costs of using R++ rather than C++. The most frequent user complaint about R++ was that it increased compilation time.[17] For example, the R++ compilation times for the line-labeling and employee-manager applications from the previous section are 99.1 and 16.7 seconds, respectively, while the compilation times for the functionally equivalent C++ programs are 60.5 and 10.7 seconds. As discussed above, 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 second major complaint

[17] Apparently, the increased execution times and memory requirements of R++ were not nearly as bothersome. This is probably an artifact of the previous alarm monitor being written in C5, rather than in C++.

about R++ was that it lacked a good debugger. This is because current object-oriented development tools don't provide any assistance for the rule construct added to R++.

Finally, early indications from ANSWER–4ESS suggest that R++ will also yield a reduction of software maintenance costs. An analysis of R++ versus C++ using the quantitative software metric "lines of code," which has been shown to correlate with software life-cycle costs [28], lends support to this belief.[18] As demonstrated for the two application programs used to compare resource consumption, R++ programs have less lines of codes than functionally equivalent C++ programs, even when the C++ program does less work due to optimization. While we don't have a user-written C++ version of ANSWER-4ESS, we do have the C++ program that results from the R++ translation process. In ANSWER–4ESS, 2,897 lines of R++ code are translated into 36,674 lines of C++ code by the R++ translator. In addition, ANSWER–4ESS contains 7,797 lines of pure C++ code, which are not translated. In sum, the total size of the R++/C++ version of ANSWER–4ESS is 10,694 lines of code, while the purely C++ version (after the R++ translation process) is 44,471 lines of code. By using R++, the ANSWER programmers thus generated a C++ version of ANSWER–4ESS by writing slightly less than a quarter of the ultimate number of lines of code.

Related Work

Three different communities have added rules to object-oriented programming languages. The Artificial Intelligence community, as described in the introduction, has employed pattern-matching production rules, converted to work directly with native objects rather than with "working memory elements". CERS [4], ILOG Rules [5], RAL/C++ [6], RETE++ [7], and Venus [29] are recent examples of this approach. While pattern-matching rules are more expressive than path-based

[18] While several other metrics have been proposed in the literature [28], most have only been validated for standard (not rule-based or hybrid) languages. Thus, such metrics can not be meaningfully applied to R++. While the metric "conditions per rule" is applicable to R++, it is not applicable to C++. Recall that for the line-labeling benchmark discussed above, the R++ and C5 versions had a similar number of conditions per rule.

rules,[19] and allow more complex rule execution, such rules exist outside of the class system. In contrast, rules in R++ are class members and thus are organized within the inheritance hierarchy. Further, since the rules are path-based and not pattern-matching, the algorithms for processing rules are more efficient, and programmers cannot circumvent relationships defined in an object model,

A second community has approached rules in the context of active object-oriented databases. Many active databases (e.g., Sentinel [26], Ode [30, 31], ADAM [32], REACH [33], SAMOS [34]) follow an event-condition-action paradigm [35], where ECA-rules associate events with conditions and actions. (Ode uses a simpler event-action model, with conditions folded into events [36].) Events are monitored and trigger rules. Rule conditions are evaluated when rules are triggered, and rule actions are executed when rule conditions are satisfied. Much of the work in active databases thus focuses on the specification and detection of events. Although the details of particular systems vary, primitive event sets typically include method invocation, and sometimes also internal database events and time events. Primitive events can be generated by objects before or after they invoke methods, and in some systems also at any user-specified point in the method. All member functions can be assumed to be potential events, or users can specify which member functions generate events. Most systems allow complex events to be composed from primitive events using operators such as sequence, conjunction, disjunction, and negation. Different coupling modes between events, conditions, and actions address the composition of events relative to transaction boundaries, and specify when rules should be executed relative to their triggering.

R++, in contrast, reflects its roots in the condition-action paradigm of standard OPS5-style production systems, in that R++ rules do not explicitly specify events. However, while standard production rules are active all the time (in active database terms, every update is an event for every rule), R++ rules (like the rules of active databases) need to be triggered, i.e., events must be detected. As discussed above, R++ rules are implicitly triggered via relevant construction and relevant change. Events are implicitly specified in rule conditions, and event detection is implicitly

[19] The arbitrary joins of pattern-matching rules can, however, be simulated in R++ by adding new paths to the object model.

performed by C++ code that is automatically output by the R++ translator. Users only need to appropriately use the keyword `monitored` and the R++ modifier functions. Thus, compared to many active databases, R++ does more of the work involving event specification and detection. However, some active database systems also transparently detect primitive events, e.g., SAMOS modifies method bodies so that primitive events can be detected without user intervention. In addition, since R++ primitive events are at a finer level of granularity than typical database events, we believe that R++ helps guard against unexpected rule firings. On the other hand, R++ rules are less expressive than most database rules with respect to events. R++ event composition is restricted to conjunction, and R++ does not allow users to specify rule execution preferences via coupling modes. These are potential areas for future work, as discussed in the next section. The path-based restriction of R++ also makes its rules less expressive. As discussed above, we view this as a positive restriction, as global OPS5-style production rules are less object-oriented than R++ rules.

Independently of the rule paradigm, there are many other ways for systems to vary, based on how rules are incorporated into the object-oriented environment. For example, rules can have intra-object or inter-object scope, can be declared in classes or as first-class objects, and can be compiled into the host language and/or processed at runtime. In fact, many of the R++ design choices and implementation techniques have already been used in the active object-oriented database community. What is particularly novel about R++ is that it *simultaneously* allows inter-object rules to be declared in classes and preprocessed into the host language.

To elaborate, R++, inspired by the constraints and triggers of Ode, allows rules to be specified only at class definition time, which allows them to be pre-processed into host-language code. This approach has runtime efficiency advantages, and also naturally supports rule inheritance. (On an orthogonal note, R++, like Ode, does not allow rule execution orderings via rule priority levels or position in declarations.) REACH similarly requires rules to be specified at compile time, although in REACH rules are first class objects; a rule is mapped into a rule object and two C functions (for condition evaluation and action execution). SAMOS defines rules and events using a high-level specification language, then translates them into first-class objects. Note that users are thus not aware of the internal system representation of rules (and events) as objects. In all of these approaches, rule modification requires system recompilation. Although compiler-based approaches often work well when rule bases do not change with high frequency, in our major application, namely ANSWER-4ESS, the most frequent user complaint was in fact that R++ increased compilation time. Although not a concern for R++, the compilation approach is also problematic for interpretive object-oriented environments. Finally, the compilation approach limits the extensibility of the system when there are pre-existing objects in a database, a situation which is somewhat similar to when there are legacy classes in an object-oriented system. These concerns have led to runtime rule specifications in ADAM, and to both class definition and runtime rule specifications in Sentinel. Other reasons for supporting run-time rule specifications, as well as for treating rules and events as first class objects rather than as "member rules", are based on database concerns such as transaction semantics and persistence. Since R++ is not an active *database* system, concerns arising from the need to provide full database capabilities are not applicable to R++, and have led to different design choices.

Unlike many active database systems (e.g., Ode, ADAM, REACH), both R++ and Sentinel support rules that can be triggered by changes to multiple instances of objects, even from different classes. For example, while the triggers of Ode allow for the execution of C++ code when a condition is satisfied on an object, 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. In contrast, the rules of R++, like the rules of Sentinel, allow objects to react to their own changes as well as to the changes of other objects. R++ achieves this functionality while allowing rules to be declared only in classes. The designers of Sentinel mistakenly believed that rules must be treated as objects in order to allow rules to be shared across objects. They felt that if rules were declared only inside classes, then ". . . a rule that ensures an employer's salary is always less than his/her manager's salary needs to be declared twice—once within the employee class and once within the manager class." [26] As shown by the R++ coding of this example in Figure 3, rules can both be declared inside classes and shared. Note that the other disadvantages pointed out by the Sentinel designers with respect to rule declarations inside classes are still valid, however. In particular, changing rules involves modifying class definitions and thus system recompilation.

Other active database work is less similar to R++. The event-condition-action rules in POSTGRES [37] and Starburst [38] 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). VenusDB [29] extends a C++ traditional rule-based language to allow rule-based applications to execute against standard relational databases.

A third community has added rules to deductive databases, including object-oriented deductive databases such as Coral++ [39]. Rules in deductive databases are run to satisfy queries and so have a very different purpose than rules in R++.

Finally, our work (and much of the above related work) is similar to that of Lopez et al. [40], in that one of our motivations is a desire to declaratively express constraints among objects. However, we propose to express constraints using rules instead of a constraint language. Admittedly, rules still have a procedural component (they run in some order, their consequent consists of procedural code, etc), and so rules are in a sense less declarative than the constraint language proposed by Lopez et al. However, by using rules, we believe that we are able to achieve the effect of declarative representation of constraints at a much lower cost in terms of programmer learning time, semantic complexity, and implementation complexity. As we argued above, R++ rules fit naturally into the object-oriented framework and thus are a relatively small extension to C++.

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, based on the feedback from the developers of ANSWER-4ESS, the next version of R++ needs to have shorter compilation times. 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.

Second, 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.

Third, 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.

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 a general expression is possible, under the same caveat as given above for data used by functions, but keeping track of the branching is more difficult.

Fifth, some rule languages allow other syntactic constructs in their rule conditions. Although we have found the current expressive power of R++ sufficient for our purposes, it would be interesting to explore such possible extensions to R++. For example, we have not provided disjunction in rules, as it can be implemented using multiple rules, but it is a possible extension.

Finally, we believe that the same kind of integration that we achieved between rules and C++ objects in R++ can be realized in other object-oriented languages. We would thus like to incorporate R++ technology into other object-oriented languages such as Smalltalk and Java.

Summary

The key contribution of our work lies in the careful synthesis of two programming paradigms: object-oriented programming and data-driven computation. R++ adds path-based rules—rules that strictly follow inter-object paths in a domain model—as a new kind of class member, subservient to the object-oriented paradigm of C++. Member rules define the automatic behavior of a class, monitoring data members for changes that satisfy a rule's condition, and then executing the rule's action. A rule monitors all instances of its class and, by inheritance, all instances of derived classes. With syntax similar to C++ member functions and semantics like "automatic" member functions, R++ rules are relatively easy to learn and apply.

The main benefit of R++ compared to C++ is that the addition of rules provides a useful level of abstraction, allowing C++ programmers to avoid focusing on low-level implementation details when modeling dynamic collections of objects. 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 (for example, to react to important states, to describe state transitions, to apply business policies, and to detect exceptions). Rules can make programs clearer and more robust because a single rule to express a multi-object constraint or invariant or policy can replace several variants of the same logic scattered throughout the procedural code. Also, rules relieve the programmer of the burden of explicit procedural control since rules are triggered automatically by relevant construction and relevant change. By providing a useful level of abstraction, rules help bridge the gap between what a programmer wants to say and what he or she has to say.

The main benefit of R++ compared to previous integrations of rules and objects is that R++ uses path-based rather than OPS5-style pattern-matching rules. Because path-based rules must follow pointers already in a domain model, and because such rules have no more access capabilities than member functions, path-based rules are in conformance with the object-oriented paradigm. In addition, because path-based rules are simpler than OPS5-style rules, R++ admits a simpler, and more efficient, rule evaluation mechanism. While R++ programs take longer than C++ programs to execute, R++ programs typically take less time to execute than equivalent OPS5-style programs. Path-based rules thus offer performance advantages over pattern-matching rules for many natural applications.

By adding another paradigm to the programmer's repertoire, R++ expands the kinds and complexity of tractable applications. R++ has been used in the implementation of an alarm monitor for a telephone switching system where the object-oriented paradigm was used to model the structure of the switch and data-driven rules were used to monitor the state of the switch [11, 12]. Because R++ allowed programmers to focus more on the problem rather than on the low-level details, the use of R++ resulted in considerable savings of both code and expense.

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 at `http://www.research.att.com/sw/tools/r++`. Alternatively, send e-mail to `r++@orbit.hr.att.com`.

Acknowledgements

We would to acknowledge Ron Brachman, Raj Dube, Henry Kautz, Dick Machol, Brian Minnihan, Jennifer Thien, and Pramod Warty for their support of the research and development effort behind R++. We would like to thank Amal Ahmed for adding STL functionality to R++, Steve Correl, Johannes Ros, Walid Saba, Anoop Singhal and Gary Weiss for being some of the first users of R++, and Daniel Miranker for providing us with a set of OPS5 benchmarks. Finally, we would like to note that some portions of this paper were originally published in [41].

Appendix: A Complete R++ Program

This section presents an example of a complete R++ program, using a simple alarm monitoring domain. The program uses knowledge about a device's design in order to process alarms, and is motivated by our alarm monitor application. The program consists of three files: a class header file (Figure 8), a class source file (Figure 9), and a main source file (Figure 10). Rule-containing files have the letter "r" in their suffix to indicate that the files must be translated into C++ by the R++ translator. By R++ convention, rule-affecting classes are defined in ".rh" files, while member functions and R++ "member rules" are defined in ".rC" files. R++ translates these files into ".r.h" and ".r.C" files, respectively.

Figure 8 shows the class definitions that are used to represent the domain model. The device to be monitored and its component devices are organized into a class hierarchy. Diagnostically important relationships between devices are represented as links between objects. The `part_of` and `components_of` links represent physical part-whole and whole-part relationships, respectively. `Standby` means that if one component fails its standby partner will take over automatically. Device objects also provide repositories for information acquired during the monitoring process, namely, current device state (`state`), as well as alarm histories generated by both a device (`self_alarm`) and its component devices (`component_alarms`).

Figure 9 shows the rule definitions for the rule declarations in Figure 8. The first three rules maintain the integrity of the device model. For any device that points to a larger device via `part_of`, the rule `add_component` creates a backpointer `components_of` from the larger device to the device. The rule `standby_inverse` ensures that the standby property is symmetric. For `ABC` devices, rule `has_standby_controllers` adds a `standby` pointer from a component `controller` device to another component `controller`, when the controllers satisfy certain properties.

The next three rules propagate alarms across devices. Whenever a device both has a new `self_alarm` and is `part_of` a larger device, the rule `new_component_alarm` adds the

```
#include <iostream.h>                           // Basic C++ stream operations.
#include <Set.h>
#include <List.h>
#include <String.h>

class Alarm;
enum DeviceState { IN_SVC, OUT_SVC };

class Device {
public:
    Device(Device* dpart_of, const String& dname, int dnum);

    monitored String            name;            // Device "type"
    monitored int               num;             // Device number
    monitored Device*           part_of;         // Pointer to containing device
    monitored Set_of_p<Device>  components_of;   // Pointers to component devices
    monitored Device*           standby;         // Standby device for this device
    monitored DeviceState       state;           // Current device state
    monitored Alarm*            self_alarm;      // Alarm for this device
    monitored List_of_p<Alarm>  component_alarms;// Alarms on component devices

    rule add_component;
    rule standby_inverse;
    rule new_component_alarm;
    rule propagate_component_alarm;
    rule go_out_of_service;
    rule alert_if_device_out;
    rule alert_if_device_in_standby_out;
    rule alert_if_device_out_standby_out;

    friend ostream& operator<<(ostream&, Device *);};

class Frame : public Device {
public:
    Frame(Device* dpart_of, const String& dname, int dnum) : Device(dpart_of, dname, dnum) { };

    rule alert_if_multiple_alarming_components;};

class ABC : public Frame {
public:
    ABC(Device* dpart_of, const String& dname, int dnum) : Frame(dpart_of, dname, dnum) { };

    rule has_standby_controllers;};

class XYZ : public Frame {
public:
    XYZ(Device* dpart_of, const String& dname, int dnum) : Frame(dpart_of, dname, dnum) { };

    rule propagate_component_alarm;};

class Cntl : public Device {
public:
    Cntl(Device* dpart_of, const String& dname, int dnum) : Device(dpart_of, dname, dnum) { };};

class Proc : public Device {
public:
    Proc(const String& dname) : Device((Device *)0, dname, 0) { };};

class Alarm {
public:
    Alarm(Device *);};
```

Figure 8: File “example.rh” (classes).

```
$include "example.rh"

Device::Device(Device *dpart_of, const String& dname, int dnum)
{   cout << "\nConstructed Device " << dname << dnum << endl;  name = dname;
    num = dnum;  part_of = dpart_of;  state = IN_SVC; standby = 0;  self_alarm = 0;}

ostream& operator<<(ostream &ostrm, Device *dvc)
{   if (dvc->part_of) return ostrm << dvc->part_of << dvc->name << dvc->num;
    else return ostrm << dvc->name << dvc->num;}

rule Device::add_component {
    Device * dvc = part_of
=>  cout << "Rule Fired....Device::add_component to " << dvc << endl;
    dvc->insert_components_of(this);}

rule Device::standby_inverse {
    Device * stby = standby && stby->standby != this
=>  cout << "Rule Fired....Device::standby_inverse for " << stby << " is " << this << endl;
    stby->set_standby(this);}

rule ABC::has_standby_controllers {
    Device * co0 @ components_of && co0->name == "Cntl" && co0->num == 0 &&
    Device * co1 @ components_of && co1->name == "Cntl" && co1->num == 1
=>  cout << "Rule Fired....ABC::has_standby_controllers " << this << endl;
    co0->set_standby(co1);}

rule Device::new_component_alarm {
    Alarm * alrm = self_alarm && Device * dvc = part_of
=>  cout << "Rule Fired....Device::new_component_alarm added to " << dvc << endl;
    dvc->put_component_alarms(alrm);}

rule Device::propagate_component_alarm {
    Alarm * alrm @ component_alarms && Device * dvc = part_of
=>  cout << "Rule Fired....Device::propagate_component_alarm to " << dvc << endl;
    dvc->put_component_alarms(alrm);}

rule XYZ::propagate_component_alarm {
    Alarm * alrm @ component_alarms && Device * dvc = part_of
=>  cout << "Rule Fired....XYZ::propagate_component_alarm does nothing " << endl;}

rule Device::go_out_of_service {
    self_alarm
=>  cout << "Rule Fired....Device::go_out_of_service " << this << endl;
    this->set_state(OUT_SVC);}

rule Device::alert_if_device_out {
    state == OUT_SVC
=>  cout << "\n*** ALERT *** Device " << this << " is out of service" << endl;}

rule Device::alert_if_device_in_standby_out {
    state == IN_SVC && Device * stby = standby && stby->state == OUT_SVC
=>  cout << "\n*** ALERT *** Stand-by " << standby << " for device " << this << " is out of service" << endl;}

rule Device::alert_if_device_out_standby_out {
    state == OUT_SVC && Device * stby = standby && stby->state == OUT_SVC
=>  cout << "\n*** ALERT *** Stand-by partners " << stby << " and " << this << " are out of service" << endl;}

rule Frame::alert_if_multiple_alarming_components {
    component_alarms.length() > 1
=>  cout << "\n*** ALERT *** Multiple alarming components for device " << this << endl;}

Alarm::Alarm(Device * dvc) {cout << "\n>> New alarm on device " << dvc << endl; dvc->set_self_alarm(this);}
```

Figure 9: File “example.rC” (functions/rules).

```
#include <iostream.h>
#include "example.r.h" // Output by the R++ translator
#include "rpp__main.h" // Global declarations

main()
{   Proc *Proc0 = new Proc("Proc");
    ABC *ABC6 = new ABC(Proc0, "ABC", 6);
    Cntl *ABC6Cntl0 = new Cntl(ABC6, "Cntl", 0);
    Cntl *ABC6Cntl1 = new Cntl(ABC6, "Cntl", 1);
    XYZ *XYZ0 = new XYZ(Proc0, "XYZ", 0);
    Cntl *XYZ0Cntl0 = new Cntl(XYZ0, "Cntl", 0);
    Cntl *XYZ0Cntl1 = new Cntl(XYZ0, "Cntl", 1);

    new Alarm(ABC6Cntl0);
    new Alarm(XYZ0Cntl0);
    new Alarm(ABC6Cntl1);
}
```

Figure 10: File “main.C”

new alarm to the list of `component_alarms` of the larger device. Whenever a device with a `component_alarm` is itself `part_of` another device, the rule `propagate_component_alarm` further propagates the alarm. However, the next rule overrides this propagation rule, replacing it, for devices of type `XYZ`, with a rule that does not propagate the alarm.

The next rule takes a device out of service based on the results of alarm monitoring. While in a real monitoring system alarms are typically subject to thresholding, in rule `go_out_of_service` we assume that a single alarm on a device will take it out of service.

The remaining rules monitor for conditions that require technicians to be alerted, and generate appropriate alert messages. The rule `alert_if_device_out` generates an alert whenever a device is taken out of service. The next two rules use knowledge about the device's design (namely, knowledge of its `standby` relationships) to warn about potential and actual backup failures when devices go out of service. The rule `alert_if_multiple_alarming_components` uses the part-whole structure of the device model to hypothesize a root cause of some alarms. In particular, if at least two components of a `Frame` device are generating alarms, then the frame is considered suspect (even if it is itself not alarming).

Figure 10 shows the main source file. C++ constructor functions are used to create seven

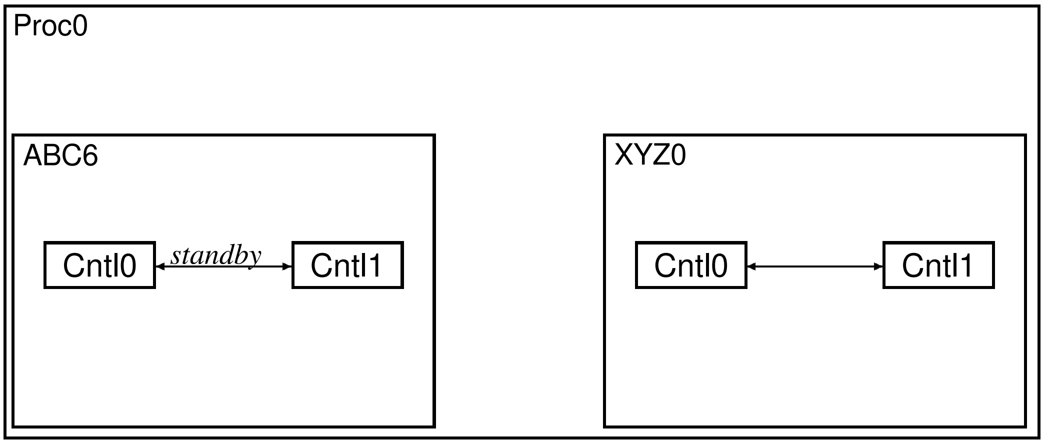

Figure 11: Partial pictorial representation of the device model.

devices and a set of `part_of` links between them. Constructor functions are also used to create three alarms and to associate them with devices using `self_alarm`.

Figure 11 pictorially represents part of this device model. Boxes represent devices, the nesting of boxes represent the `part_of` (and `components_of` relationships), and the labeled arrows represent the `standby` relationships (discussed below).

Figure 12 shows the output of the alarm monitor. R++ rules add the `components_of` and `standby` relationships to the device model. Note that since `has_standby_controllers` is defined for only `ABC` devices, `standby` relationships are added between the controllers of `ABC6` but not between the controllers of `XYZ0`.

R++ rules are also used to propagate alarms, to take alarming devices out of service, and to print alerts. For example, when the first alarm is generated (on `Cntl0` of `ABC6` of `Proc0`), the alarm is propagated to `ABC6`—and then propagated to `Proc0` in a forward chaining process. The initial alarm generation also causes `Cntl0` to go out of service, which causes the two alert messages to be generated.

When the second alarm is generated (on `Cntl0` of `XYZ0` of `Proc0`), the alarm is propagated to `XYZ0`. However, unlike the first alarm, the second alarm is not propagated further. This is because `XYZ`'s version of `propagate_component_alarm` overides `Devices`'s version. The

```
Constructed Device Proc0

Constructed Device ABC6
Rule Fired....Device::add_component to Proc0

Constructed Device Cntl0
Rule Fired....Device::add_component to Proc0ABC6

Constructed Device Cntl1
Rule Fired....Device::add_component to Proc0ABC6
Rule Fired....ABC::has_standby_controllers Proc0ABC6
Rule Fired....Device::standby_inverse for Proc0ABC6Cntl1 is Proc0ABC6Cntl0

Constructed Device XYZ0
Rule Fired....Device::add_component to Proc0

Constructed Device Cntl0
Rule Fired....Device::add_component to Proc0XYZ0

Constructed Device Cntl1
Rule Fired....Device::add_component to Proc0XYZ0

>> New alarm on device Proc0ABC6Cntl0
Rule Fired....Device::new_component_alarm added to Proc0ABC6
Rule Fired....Device::propagate_component_alarm to Proc0
Rule Fired....Device::go_out_of_service Proc0ABC6Cntl0

*** ALERT *** Device Proc0ABC6Cntl0 is out of service

*** ALERT *** Stand-by Proc0ABC6Cntl0 for device Proc0ABC6Cntl1 is out of service

>> New alarm on device Proc0XYZ0Cntl0
Rule Fired....Device::new_component_alarm added to Proc0XYZ0
Rule Fired....XYZ::propagate_component_alarm does nothing
Rule Fired....Device::go_out_of_service Proc0XYZ0Cntl0

*** ALERT *** Device Proc0XYZ0Cntl0 is out of service

>> New alarm on device Proc0ABC6Cntl1
Rule Fired....Device::new_component_alarm added to Proc0ABC6
Rule Fired....Device::propagate_component_alarm to Proc0

*** ALERT *** Multiple alarming components for device Proc0ABC6
Rule Fired....Device::go_out_of_service Proc0ABC6Cntl1

*** ALERT *** Device Proc0ABC6Cntl1 is out of service

*** ALERT *** Stand-by partners Proc0ABC6Cntl0 and Proc0ABC6Cntl1 are out of service

*** ALERT *** Stand-by partners Proc0ABC6Cntl1 and Proc0ABC6Cntl0 are out of service
```

Figure 12: Output.

second alarm generation also causes `Cntl10` to go out of service. This latter change to the device model causes the next alert message to be generated.

Finally, when the last alarm is generated (on `Cntl1` of `ABC6` of `Proc0`), the alarm is propagated to `ABC6`. This propagation causes the alarm to be further propagated to `Proc0` and the alert about `ABC6` to be generated. The alarm on `Cntl1` also causes `Cntl1` to go out of service, which causes the remaining alert messages to be generated.

References

[1] Thomas A. Cooper and Nancy Wogrin. *Rule-Based Programming with OPS5*. Morgan Kaufmann, San Mateo, California, 1988.

[2] F. Pachet, editor. *Proceedings of the OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems (EOOPS)*. 1994. Technical report LAFORIA 94/24, Institut Blaise Pascal.

[3] Inference Corporation. *Art Reference Manual*, 1987.

[4] Daniel P. Miranker, Frederic H. Burke, Jeri J. Steele, David R. Haug, and John Kolts. The C++ embeddable rule system. *International Journal on Artificial Intelligence Tools*, 2(1):33–46, 1993.

[5] Patrick Albert. ILog Rules, embedding rules in C++: Results and limits. In *Proceedings of the OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems (EOOPS)*, 1994. Technical report LAFORIA 94/24, Institut Blaise Pascal.

[6] Charles L. Forgy. RAL/C and RAL/C++: Rule-based extensions to C and C++. In F. Pachet, editor, *Proceedings of the OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems (EOOPS)*, 1994. Technical report LAFORIA 94/24, Institut Blaise Pascal.

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

[8] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, Massachusetts, second edition, 1991.

[9] USL. *C++ Standard Components Release 3.0 Documentation*, 1992.

[10] ObjectSpace. *STL Toolkit, Version 2.0*, 1996.

[11] 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, August 1995.

[12] Anil Mishra, Diane Litman, Peter Patel-Schneider, Johannes P. Ros, Anoop Singhal, Gary Weiss, James Crawford, and Daniel Dvorak. R++: Using rules in object-oriented designs. In *Experience Report, Addendum to the Proceedings of the OOPSLA'96 Conference on Object-Oriented Programming Systems, Languages, and Applications*, San Jose, CA, 1996.

[13] Anoop Singhal, Gary Weiss, and Johannes P. Ros. A model based reasoning approach to network monitoring. In *Proceedings of the Conference on Information and Knowledge Management (CIKM-96) Workshop on Databases: Active and Real-time*, pages 47–52, Rockville, MD, 1996. Available as CSE-TR-313-96 (N. Soparkar and K. Ramamritham eds.), University of Michigan.

[14] 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, July 1991.

[15] J. M. Crawford. *Access-Limited Logic—A language for knowledge representation*. PhD 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.

[16] Diane J. Litman, Anil K. Mishra, and Peter F. Patel-Schneider. Modeling dynamic collections of interdependent objects using path-based rules. In *Proceedings of the OOPSLA'97 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Atlanta, 1997.

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

[18] Daniel P. Miranker, David A. Brant, Bernie Lofaso, and David Gadbois. On the performance of lazy matching in production systems. In *Proceedings of AAAI-90*, pages 685–692, 1990.

[19] Production Systems Technology. *RAL Language Guide, Version 1*, 1991.

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

[21] S. Dar, R. Agrawal, and N. H. Gehani. The O++ database programming language: Implementation and experience. In *Proceedings of the 9th IEEE International Conference on Data Engineering*, Vienna, Austria, 1993.

[22] Daniel P. Miranker and Bernie J. Lofaso. The organization and performance of a treat-based production system compiler. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):3–9, 1991.

[23] Alexander J. Pasik. A source-to-source transformation for increasing rule-based system parallelism. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):336–343, 1992.

[24] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley Publishing Company, Reading, MA, 1984.

[25] David A. Brant, Timothy Grose, Bernie Lofaso, and Daniel P. Miranker. Effects of database size on rule system performance: Five case studies. In *Proceedings 17th International Conference on Very Large Databases*, 1991.

[26] E. Anwar, L. Maugis, and S. Chakravarthy. A new perspective on rule support for object-oriented databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 99–108, 1993.

[27] Jennifer Thien. R++. AT&T Internal Memorandum, June 1996.

[28] Lane Warshaw and Dan Miranker. A case study of Venus and a declarative basis for rule modules. In *Proceedings of the Fifth International Conference on Information and Knowledge Management (CIKM-96)*, Rockville, MD, 1996.

[29] Stephen Correl and Daniel P. Miranker. On isolation, concurrency, and the venus rule language. In *Proceedings of the 4th International Conference on Information and Knowledge Management (CIKM'95)*, 1995.

[30] Narain H. Gehani and H. V. Jagadish. Active database facilities in Ode. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 327–336, 1991.

[31] Narain H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In Widom and Ceri [42], pages 207–232.

[32] O. Diaz, N. Paton, and P. Gray. Rule management in object-oriented databases: A unified approach. In *Proceedings of VLDB*, 1991.

[33] A. P. Buchmann, J. Zimmermann, J. A. Blakeley, and D. L. Wells. Building an integrated active oodbms: Requirements, architecture, and design decisions. In *Proceedings of the 11th International Conference on Data Engineering*, 1995.

[34] Stella Gatziu and Klaus R. Dittrich. Events in an active object-oriented database system. In *Proceedings of the 1st International Workshop on Rules in Database Systems*, 1993.

[35] U. Dayal, B. Blaustein, A. Buchmann, S. Chakravarthy, D. Goldhirsch, M. Hsu, R. Ladin, D. McCarthy, and A. Rosenthal. The hipac project: Combining active databases and timing constraints. *ACM Sigmod Record*, 17(1), 1988.

[36] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *Proceedings of Sigmod*, 1992.

[37] Spyros Potamianos and Michael Stonebraker. The POSTGRES rule system. In Widom and Ceri [42], pages 43–61.

[38] Jennifer Widom. The Starburst rule system. In Widom and Ceri [42], pages 87–109.

[39] Divesh Srivastava, Raghu Ramakrishnan, S. Sudarshan, and Praveen Seshadri. Coral++: Adding object-orientation to a logic database language. In *Proceedings of the Nineteenth International Conference on Very Large Databases*, 1993.

[40] Bjorn Freeman-Benson Gus Lopez and Alan Borning. Implementing constraint imperative programming languages: The kaleidoscope'93 virtual machine. In *Proceedings of the OOPSLA'94 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 23–27, October 1994.

[41] 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.

[42] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, California, 1996.