Modeling Dynamic Collections of Interdependent Objects Using Path-Based Rules

Diane Litman
Peter F. Patel-Schneider
AT&T Labs—Research
180 Park Avenue
Florham Park, NJ 07932
{diane,pfps}@research.att.com

Anil Mishra
AT&T Network and
Computing Services
480 Red Hill Road
Middletown, NJ 07748
anilm@research.att.com

Abstract

Standard object-oriented languages do not provide language support for modeling changing collections of interdependent objects. We propose that R++, an integration of the rule and object-oriented paradigms, provides a mechanism for easily implementing such models. R++ extends C++ by adding a new programming construct called the *path-based* rule. Such data-driven rules are restricted to follow pointers between objects, and are like "automatic methods" that are triggered by changes to monitored objects. Path-based rules encourage a more abstract level of programming, and unlike previous rule integrations, are not at odds with the object-oriented paradigm and offer performance advantages for natural applications.

1 Introduction

Object-oriented languages have simplified the design and implementation of sophisticated applications. However, as application domains have become increasingly dynamic and complex, it has become necessary to model such domains using changing collections of interdependent objects. Such models are difficult to implement in standard object-oriented languages due to a lack of programming constructs at the right level of abstraction.

Consider the creation and maintenance of a model containing a multiple-object invariant, e.g., 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.[1] To implement such a model using standard object-oriented languages, a significant amount of low-level code must be written. Code to enforce the invariant must be added in multiple classes, each time slightly different. In other words, code must be written to simulate a multiple-object method. Code to trigger the code to enforce the invariant must also be added, in exactly all the appropriate places. Bookkeeping code such as maintaining backpointers between the interdependent objects must be provided.

Although the inherent difficulty of specifying and implementing models with changing inter-object dependencies is well known, and a number of proposals to solve this problem have recently emerged [19, 21], there is a notable lack of language support for such proposals in the context of commercial object-oriented languages. Thus, most object-oriented programmers must still model changing inter-object dependencies by designing and implementing their own low-level solutions using standard object-oriented techniques.

We propose that the problem of modeling dynamic collections of interdependent objects can be given a systemic solution, by providing explicit language support in terms of data-driven computation. Our solution, called R++ [9], extends the C++ language[2] [27] with a single new programming construct—*the rule*. Informally, a rule is a statement composed of a condition and an action specifying what to do when the condition becomes true. Whenever some program data changes, the rules whose conditions involve that data are examined, and if a rule's condition succeeds its action is executed. The action may of course modify data and therefore trigger other rules. Returning to the example above, our solution allows a C++ programmer to directly specify the multiple-object invariant using the R++ rule construct. The R++ translator then automatically generates low-level C++ code to implement the invariant (e.g., to do the bookkeeping, to associate the invariant with the multiple objects, and to trigger the code enforcing the invariant).

While there have been previous attempts to integrate standard (OPS5-style [5]) production rules into the object-oriented paradigm [20, 22, 1, 14, 18], R++ is notable in its use of a new kind of a rule—the *path-based* rule. Path-based rules allow R++ to be in conformance with, rather than at odds with, the object-oriented paradigm. Path-based rules are a new kind of class member, conceptually like an "automatic method". Path-based rules have syntax and semantics similar to C++ member functions. 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 the '`this`' object to related objects. Because they have no more access capabilities than member functions, path-based rules cannot bypass the relationships designed into an object-oriented domain model. In contrast, OPS5-style

[1] The usual statement of this example is that a manager must make more than his or her direct reports. When we were working at Bell Labs we found that this was not realistic, so we have modified the problem to fit reality more closely, as well as to be a better example.

[2] Little of R++ actually depends on C++, which was chosen as the base language for R++ largely because of its popularity.

rules can perform arbitrary joins between unrelated objects, thus violating the locality of reference designed into domain models. In addition, path-based rules yield a performance advantage over OPS5-style production rules, as the algorithm that processes path-based rules is simpler and typically faster than the algorithm that process production rules.

In this paper we discuss the design and implementation of R++ and analyze the costs and benefits of its use. Our experience suggests that path-based rules are an effective way to model changing collections of interdependent objects [8]. With respect to C++, the major cost of using R++ is less efficiency. R++ programs have longer compilation and execution times, and higher memory needs. The corresponding benefit, however, is increased programmer productivity. R++ programs are shorter, contain fewer similar-but-not-identical pieces of code, and are more modular. These conclusions should not be particularly surprising, as they are typical of higher-level programming paradigms. What is perhaps more surprising is that R++ has several benefits when compared to the standard high-level rule-based approach. Most notably, R++ programs typically run faster, and are more cleanly integrated with the object-oriented paradigm.

2 R++

2.1 Overview

R++ rules are *path-based* in that their conditions must start with some root object and can reach other objects only by following paths of pointers (*access paths*).[3] The conditions of path-based rules are evaluated in response to certain activities in the rest of the system. If the condition succeeds, then the action of the rule is executed. The action of a path-based rule has access to the object bindings from its condition.

R++ rules are associated with classes, just as data and functions are associated with classes, and not directly on the objects that are instances of the class. In effect, R++ views rules as "member rules" of classes. To further this resemblance, rules are given class-specific names, as are the data and functions associated with the class. Rules also follow the same access limitations as member functions.

The association of rules with classes provides a type for the root object of a rule, namely that class with which the rule is associated. Further, the association provides a specification for when objects are brought to the attention of rules, namely that the *creation* of an instance of a class brings the object to the attention of the member rules of the class. The only way to remove an object from consideration by the rules of its classes is to destroy it. Therefore a rule associated with a class is active on all objects that belong to the class.

Rules are "inherited" by sub-classes, just as member functions are. Rules can be overridden in sub-classes, just as member functions can. A sub-class can have a rule with the same name as a rule in a super-class. Such a rule will override the rule in the super-class for instances of the sub-class.

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

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

Figure 1: Simplified R++ Rule Syntax

The conditions of R++ rules monitor data in various objects. Changes to this data may cause the condition of the rule to succeed, or succeed in a different manner than before. Therefore, the conditions of rules are re-evaluated whenever the data they inspect changes or when a new object of the appropriate type is constructed. We call these changes *relevant changes*, and *relevant constructions* for the creation of relevant objects, when construction needs to be distinguished from other kinds of relevant change. There is no need for any other mechanism for causing the conditions of rules to be evaluated—changes to object data is the only mechanism required or allowed. The only priority mechanism for R++ rules is that rules on sub-classes are executed before rules on super-classes.

Because R++ rules are data-driven, they have to access the actual data members in objects, not just the values returned from member functions.[4] This may seem to violate the encapsulation principles of the object-oriented paradigm, except that R++ rules should be considered to form part of the implementation of collections of objects, where the underlying data is available. In fact, one major use of R++ rules is to maintain invariants on the data structures themselves.

2.2 Rule Syntax and Semantics

R++ rules look and act as much like the rest of C++ as possible. Externally, R++ rules are declared and defined in a manner similar to C++ class members, as shown in the simplified rule syntax given in Figure 1. An example R++ rule, illustrating the multiple-object invariant discussed in Section 1, is shown in Figure 2.

The condition (left-hand side) of an R++ rule is a sequence of C++ boolean expressions[5] interspersed with vari-

[3] Such *path-based rules* are not new, having appeared explicitly as "access-limited rules" in the Algernon implementation of access-limited logic [10, 7].

[4] It would be possible to determine which data member functions depend on, but this would require analysis by the underlying programming language.

[5] C++ does not have boolean expressions per se. The intent here is to allow anything that can be interpreted as a boolean expression,

```
class Employee {
protected:
  monitored set<Employee *> directReports;
  monitored int salary;
}

class Manager : public Employee {
  Employee * delegate;

  rule maintainSalary;
}


rule Manager::maintainSalary {
  Employee* drctRpt @ this->directReports &&
  Employee* scndLvlRpt @
                 drctRpt->directReports &&
  this->salary < scndLvlRpt->salary
 =>
  this->set_salary(scndLvlRpt->salary);
}
```

Figure 2: An Example Use of R++

able bindings. The boolean expressions, and the expressions in the bindings can, of course, use the variables bound earlier, just as in C++.

The bindings in a condition look very much like C++ variable definitions. The first kind of binding, such as `Employee * dlg = this->delegate`, looks just like a C++ variable definition, and simply sets a variable to the value of an expression, succeeding if that value is non-null. The example binding declares a variable of type pointer to `Employee` and sets it to the value of `this->delegate`, succeeding only if the value is non-null. The other kinds of bindings, for example, `Employee * drctRpt @ this->directReports` in Figure 2, are similar to C++ variable definitions but use "`@`" instead of "`=`". These are branch bindings, where the "`@`" should be read as "at" or "in", and they bind a variable to elements of a set or list of values.[6] The example branch binding says to iterate over all values in the `directReports` data member, which is declared to be a set of pointers to employees, succeeding for each element of the set.

A condition is evaluated in the obvious way. It succeeds for those successful bindings that make its boolean expressions all evaluate to "true", or non-zero.

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

To make the connection between data members and rules explicit, R++ requires that the `monitored` keyword be given for any data member used in a rule. It is a syntax error for R++ rules to access non-monitored data members. R++ generates accessor functions for monitored data members, such as the function `set_salary` used in the rule in Figure 2.

such as pointers or integers.

[6] C++ does not have sets or lists. R++ thus requires some minimal set and list implementation, currently that from the Standard Template Library [24] or the Standard Components [28].

2.3 Rule Execution

There are three parts to rule execution:

1. triggering of rules by relevant change (including relevant construction),
2. subsequent evaluation of the rule condition, possibly delayed if other rules have been triggered by the same change, and
3. execution of the rule action for each way the rule condition succeeded.

The conditions of R++ rules are evaluated only in response to relevant change. However, the portion of the condition that is evaluated in response to a relevant change is unspecified, as is the order of evaluation. Because of this, any side-effect in a rule condition is not particularly useful, and thus rule conditions should be side-effect free. The actions of R++ rules are executed only when their conditions succeed, and only on the data that caused the condition to succeed.

There is no programmer control over the order of R++ rule execution. The order of rule execution is guaranteed only as follows:

- Rules respond to relevant change in a last-in, first-out fashion. That is, if the action of a rule makes a relevant change for some other rules, these other rules are triggered, evaluated, and potentially executed, before any other rules that may have been waiting to be evaluated.
- In response to a particular relevant change, rules on sub-classes are evaluated before rules on superclasses.

The order of rule evaluation is otherwise unspecified.

R++ rules do not execute on "old" data, nor execute more than once on the same data. For example, suppose a relevant change triggers two rules in a way that would cause both rule conditions to succeed, but the first rule to be evaluated and executed changes the original triggering data, thus retriggering both rules. Suppose that the first rule's condition does not succeed on this new data. Then there would be two triggerings of the second rule waiting to be evaluated. Only the most recent of these triggering can possibly cause the rule's action to be executed, as the older triggering is the result of data that has since been modified.

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

3 R++ Implementation

R++ does not have a "rule interpreter" per se. Instead R++ rule execution is handled by a collection of data members and member functions, most of which are placed on the root class of a rule, the class in which the rule is declared. The R++ translator inserts declarations of these data members and member functions into the appropriate classes and adds definitions of the member functions.

There are three aspects to executing R++ rules:

1. determining when and how to trigger rules;

2. determining when and how the condition of a rule succeeds; and
3. executing the action of a rule.

To handle the problem of stale data, R++ keeps track of its notion of the current time and timestamps all monitored data by means of extra data members in the class. Any change to a monitored data element or construction of an object in a class with rules increments R++'s current time. A change to a data element then sets the timestamp for that data element to the current time.

3.1 Triggering Rules

R++ has a strong notion of relevance, and only responds to relevant change (including relevant construction). To handle relevant construction, the constructors of classes that have rules are modified to trigger each rule defined in or inherited by the class on the new object, the root object, with current time set to when the object was created. Thus when a manager is created, the rule `Manager::maintainSalary` is triggered on the new object.

To handle other relevant change, the modifiers for monitored data members are constructed to trigger each rule that mentions that data member in its condition. This is done on each root object that could possibly result in the action of the rule being executed. The current time used for these triggerings is the time when the relevant change happened. These triggerings are also provided with the object that had its data member changed, the causative object, to further restrict the search for potential rule actions to execute.

The collection of root objects to use is determined by the root pointers for the rule and rule variable in the object being modified. For each rule and variable in the rule condition, R++ sets up a data member, called the root pointers, in the class of the variable.[7] Root pointers are maintained as rule conditions are evaluated, as detailed below. The basic idea is that every time the evaluation of a condition of a rule binds a new variable, a root pointer for that variable is added from the variable's value back to the current root object.

In the above example, a data member with the type `set<Manager *>` (and with a name constructed from the name of the rule `Manager::maintainSalary` and the variable `scndLvlRpt`) will be added to `Employee` to hold root pointers for the variable `scndLvlRpt` of rule `Manager::maintainSalary`. When a change is made to the salary of an employee, this is a relevant change for the `secondLevelReport` variable of the rule `Manager::maintainSalary` and the root pointers for this variable will be used to trigger the rule.[8] The employee where the change was made is also given to the rule execution process as the causative object.

Additions to data members used in branch bindings are handled in a special manner. In this case, both the object whose data member is being modified and the object being added are passed to the rule execution, so that only paths that involve the new link being added are used.

[7] For the implicit `this` variable the root pointer would just point back to the object itself. Rule triggering for relevant changes involving `this` are handled in an optimized fashion.

[8] Note that if the employee is also a manager this is also a relevant change for the `this` variable of the rule, and would trigger the rule using that variable also.

Because R++ controls the root pointers, R++ must arrange to remove the pointers when the root object is destroyed. R++ does this by keeping, in root objects, reverse root pointers for all root pointers pointing to the object.

When a root object is destroyed, these reverse root pointers are traversed to find and remove the root pointers to the object being destroyed. Similarly, when any object with root pointers is destroyed, its root pointers are traversed to find and remove the reverse root pointers to it.

R++ also has to be very careful to not follow pointers to objects that have been deleted. For this purpose R++ registers each of the pointers that it uses in the object being pointed to. These registered pointers are each set to null when the object is deleted, and R++ checks any pointer it uses to ensure that the pointer has not been set to null.

3.2 Evaluating Rule Conditions

Rule conditions are evaluated by means of member functions that are added to the root class of the rule. This means that the rule condition has exactly the access rights of a member function of the root class.

The simplest case of evaluating a rule condition is for relevant construction. The member function responsible for relevant construction evaluation for the rule is called with the root object and a current time. Each element of the condition is evaluated in turn until one fails or they all succeed. If one fails, then there is nothing left to do. If they all succeed, then the action of the rule is executed.

Evaluating a boolean expression in the condition is simply a matter of evaluating the expression and checking to see whether a value of "true" (non-zero) results. If so, the expression succeeds and evaluation continues; if not, the expression fails and this (branch of the) evaluation is abandoned. However, before the expression is evaluated, there is a check to see if the data it uses is still current. For each data member used in the expression, the timestamp for that data member is checked. If this timestamp is more recent than the time passed into the rule evaluation, the expression is considered to have failed, because the data it uses has been modified after the rule was triggered.

Evaluating a branch binding starts by checking the timestamp on the data member, just as for boolean expressions. Then, the rule variable for this binding is bound to one object in the set, if there is one, and rule evaluation continues in a branch evaluation, otherwise the binding fails. When this branch fails or succeeds, the rule variable for this binding is bound to another object in the set and a new branch is started, until all objects in the set have been considered, and the binding fails. In this way a branch binding iterates through all object in the set, potentially causing the execution of the rule's action on each of them.

Just after the rule variable is bound to a value, a pointer from the object back to the current root object is added to the root pointers for this variable, recording this path so that relevant changes to this object correctly trigger rules. Thus, evaluation of the above example rule will add root pointers for `drctRpt` from all `directReports` of all instances of `Manager` back to their supervisor and root pointers for `scndLvlRpt` from all `directReports` of `directReports` of instances of `Manager` back to their second-level supervisor.

Evaluating a non-branch binding is just like evaluating a branch binding with one element in the set, if the value of the expression is non-null, or a branch binding with an empty set, if the value of the expression is null.

As an example, when a new manager, `Bill`, with salary 90000 and no `directReports`, is created, R++ will execute the rule in Figure 2, starting from `Bill`. The evaluation of the rule's condition will not get very far, as `Bill` will not have any `directReports`, and this branch binding will immediately fail.

The evaluation of rule conditions in response to a relevant change is slightly more complex, but follows the same general path. The extra complexity is required to ensure that the rule is evaluated only on the object being changed, and not on other objects reachable from the root object. When the binding for the variable whose root pointer was used to trigger the rule is reached, any value bound to the variable is checked against the causative object passed into the evaluation. Any value different from the causative object results in the evaluation branch failing. Further, if no branch reaches the causative object, then there is no valid path from the root object back to the causative object. In this case the root pointer is no longer valid and is removed from the root pointers.

In the above example, rule evaluations resulting from changes to the salary of an employee passing through root pointers for `scndLvlRpt` will be passed a pointer to an employee as a causative object. When a value is bound to `scndLvlRpt`, it will be checked against the causative object, and values different from the causative pointer will cause that branch to fail, and the next element of the set to be checked.[9] If the causative object is never reached in some evaluation, the root object for this evaluation is removed from the root pointers for `scndLvlRpt` on the causative object.

In cases involving additions to data members used in branch bindings, both variables involved must match the values passed in to the rule execution. In the above example, when a new pointer is added to the `directReports` of some employee, the rule is triggered on the values of the root pointer for the `drctRpt` variable with both the employee who received a new direct report and the reportee as parameters.[10] Only branches that result in the causative pointer as a value for `drctRpt` and the new reportee as a value for `scndLvlRpt` will succeed for *these* triggerings.

For example, if a manager, say `Dan`, who has no `directReports`, is added to the `directReports` of `Bill`, the rule in Figure 2 will be triggered on `Bill`. This triggering only allows `Dan` as a value for `drctRpt`, and puts a root pointer for `drctRpt` from `Dan` back to `Bill`. As `Dan` has no `directReports`, the evaluation will not progress past the second assignment. If an employee, `Harry`, with salary 100000, is then added to the `directReports` of `Dan`, the rule of Figure 2 will be again triggered, starting with root object `Bill`, because `Bill` is in the root objects of `Dan` for variable `drctRpt` on the rule. This triggering will proceed to `Dan` (and to no other of the `directReports` of `Bill`), and thence to `Harry`, finally succeeding and resulting in execution of the action of the rule.

A further complication is that if some change is relevant for another, later variable of the rule, bindings for this later variable have to be different from the causative object. This complication is required so that rules will not be executed more than once for the same collection of variable bindings. (The same collection *will* be considered during triggering on the root objects for the later variable!) If the data member is used in a branch binding later branch bindings must be checked to ensure that they do not follow the same link. It is not sufficient to just check just the value of the branch variable or just the value of the variable of the data member in this case, both must be checked.

For example, when the `salary` of an `Employee`, say `Bill`, is changed, `Manager::maintainSalary` is triggered in two different manners. First, `Bill` is considered the root of the rule. In this triggering, `Bill` cannot be bound to the variable `scndLvlRpt`. Second, `Bill` is considered as a binding for `scndLvlRpt` and the root pointers for this variable are used to find root objects to trigger from. If the exclusion on binding `Bill` was not in the first triggering, and `Bill` was his own `scndLvlRpt`, then the rule's condition would be evaluated twice on the same data, possibly causing the rule's action to be executed twice using the same data.

3.3 Executing Rule Actions

The action of a rule is executed in the same member function as the evaluation of its condition. Therefore the code in a rule action has exactly the same access rights as a member function of the root class of the rule. The evaluation of the condition results in the correct variable bindings for the action, so the action is simply spliced into this member function. In the example just above, the execution of the rule would set the `salary` of `Bill` to 100000.

However, to ensure that the data examined by the rule are still current, the timestamps for all the data are checked just before the action is executed, and if any are newer than the time passed into the evaluation, the action is not executed.

4 Analysis

As with any new tool, R++ has both benefits that accrue from its use, and costs that it imposes, compared with C++. Most of the costs of R++ have to do with increased resource consumption. Other than these costs, the main cost with R++ is the time required to learn it, which is partially alleviated by its close integration with C++.

R++ adds a number of data members to classes. These extra data members increase the size of affected classes by a constant amount. For each variable of each rule, a data member is added to the class of the variable (the root pointers) and to the root class of the rule (the reverse root pointers). For each data member mentioned in a rule, a data member is added to the class (the timestamp).

As rule conditions are evaluated, elements are added to root and reverse root pointer sets. Each path investigated by a rule adds elements to two of these sets, provided that this element was not already there. These elements are deleted when they are discovered to be no longer valid.

When an instance of a root class is created or a relevant change is performed, R++ triggers rules. The resulting evaluation can take an arbitrary amount of time, so these operations cannot be considered to take negligible time.

[9] Actually, the set is not iterated over in this case. Instead a optimized version is used where a check is made to see if the causative pointer is in the set. If so, it is bound to the variable. If not, the binding fails.

[10] There is also triggering using the `this` root pointer.

Rule evaluation after relevant change proceeds by going back to the root object and following paths attempting to get back to the object where the change occurred. If R++ kept more data, then this rule evaluation would be faster. In the current implementation of R++, such evaluation can process a number of useless paths, but storage costs are reduced from an amount potentially exponential in the number of objects to an amount potentially quadratic in the number of objects.

One other cost that R++ imposes is incurred not at execution time, but at compile time. The methods the R++ translator inserts into user code increase compilation time, sometimes dramatically. One reason for this is that much of this additional code shows up in "`.H`" files, which are compiled multiple times in one recompilation, and involve templates, which are expensive to compile.

R++ provides a number of important benefits to weigh against its costs. These benefits have been determined by the use of R++, including its use in a major project to analyze and diagnose the 4ESS® telephone switching system, the major switch in AT&T's domestic long-distance telephone network [23].

In this project, the use of R++ allowed a fifty-percent reduction in the amount of code required to model and monitor the 4ESS switch over the amount required in a straight C++ implementation. Along with this reduction came a corresponding reduction in the amount of time and resources required to implement this portion of the system.

A related benefit is that the use of R++ eliminates the low-level, hard-to-understand portion of the code, resulting in a more-declarative implementation with clearer semantics. This makes for fewer errors in coding, a clearer and cleaner system, and easier modification and extension of the system. The rules in the 4ESS monitoring system were deemed to be much easier to understand than any C++ system developed to solve the same task. As well, the object-model was cleaner, at least in part because there was less need for the many low-level methods that would have been required to implement the rules in C++.

These benefits, while hard to quantify in many cases, are significant. In particular, the dominant software cost in many large software systems is not the initial development, but the subsequent modification and extension. Any tool that can ease these important tasks can be of great benefit.

R++ should be effective whenever there is a need for computation to be performed whenever a condition involving data of path-related objects becomes true, except in situations where time and space costs must be aggressively minimized.

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

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

5 Comparisons

5.1 Comparison with Rule Systems

There are other rule systems besides R++ that can be used to represent dynamic collections of interdependent objects. These systems generally belong to the OPS5 paradigm [5] of general pattern-matching rules. There are a number of modern systems that attempt to marry OPS5-style rules and object-oriented programming, including ART-IM [20], CERS [22], ILOG rules [1], RAL/C++ [14], and Rete++ [18].

The rules of OPS5 are more general[11] than the rules of R++, but, as they can form arbitrary "joins" between unrelated objects, violating the locality assumptions implicit in the object-oriented paradigm, there is a basic mismatch between this sort of rule and the object-oriented paradigm. OPS5-style rules also have a different semantics than R++ rules, involving a conflict resolution scheme with rule priorities, which is outside the object-oriented paradigm and which has to be kept in mind when programming with this sort of rules. The rules of R++, in contrast, exist completely within the object-oriented paradigm.

In fact, a number of the modern OPS5 systems use their own object system, called working memory, and not the object system of their base programming language. Applications using these rule systems thus need to move information between the working memory and object storage. Other systems (such as CERS [22]) require special calls to bring objects of the language to the attention of the rule system. Although this means that information does not need to be moved, it does require facilities outside of the object-oriented paradigm to control rules. In contrast, R++ directly uses the objects of its underlying object-oriented language, C++.

OPS5-style rules also sit outside the class hierarchy, unlike R++ rules, and thus are not organized in the same way as classes and objects, eliminating a beneficial interaction between R++ rules and the object-oriented paradigm.

5.2 Resource Consumption

Because R++ rules are simpler than OPS5-style rules, R++ admits a simpler, and more efficient, rule evaluation mechanism. First, R++ requires less data than a full algorithm for

[11] However, the arbitrary joins of OPS5-style rules can be simulated by adding new paths to the object model.

evaluating OPS5-style rules. Given n objects, r rules, and v variables in the largest rule condition, the worst-case heap-storage requirements for R++ are $O(rn^2)$, mostly for root pointers, whereas the RETE algorithm [13] and its variants use space $O(rn^v)$.

The speed comparison of R++ and OPS5-style rules is more complicated. Both R++ and the RETE algorithm and its variants use their stored data to reduce computation time. The speed of RETE is generally dominated by the recomputation of its stored data. In the worst case, it is possible to have to recompute all the stored data. The speed of R++ is generally dominated by finding the appropriate path from the root object back to where the change occured, which can take time $O(rn^v)$, the same as RETE.

Table 1 gives an empirical comparison between R++ and C5, a C-based superset of OPS5 using a RETE algorithm, on a single rule. Rates are given in rule firings per second on a Sun SPARCstation 10. In each performance measurement a tree[12] of objects of a specified depth (path length) and breadth (branching factor) was created; a path-length of n means that $n+1$ objects are in the path. The tree is monitored by a single rule that follows a path from the root object to a leaf object.

To test the worst-case for R++, all triggering changes were made in leaf objects. R++ outperforms C5, especially with small path lengths and low branching factors, where it is much faster.[13] This is largely due to the elimination of the bookkeeping that is required for the RETE algorithm.

Table 2 compares R++, C5, and C++ implementations of Waltz's line labeling algorithm [31], a standard benchmark for OPS5 performance. The interesting sections of the programs are contained in Appendix A. (All measurements are from a Sun SPARCstation 20.) The two sets of figures for R++ and C++ are for the Standard Components version of sets, and for an already-existing hash-set implementation optimized for good storage consumption with very small sets. The sets in this application are almost all of size 2 or 3.

Because C5 and R++ have different rule languages, the algorithm had to be recoded between C5 and R++. The C5 version is a slightly-modified version of an OPS program obtained from Daniel P. Miranker, used in a comparison study of various rule engines [3]. The R++ version was derived from the OPS program by first creating C++ data structures for the OPS working memory elements, modifying as appropriate to fit within the C++ object model. Then R++ rules were written to mimic the OPS rules that naturally fit within the R++ model. The other OPS rules were implemented in C++ code. This resulted in R++ code that was a close match to the original OPS code, and not in particularly efficient R++ rules.

The initial OPS 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 9 R++ rules with a total of 42 conditions and 22 action lines of code. Usually, OPS rules that translate into R++ rules translate on a one-for-one basis, but because OPS has no sets, the OPS code represents sets as sequences, and rules that act on any member (or subset) of the set have to be duplicated in OPS but not in R++. 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 R++ whereas the OPS rules, because they are duplicated, can directly access the appropriate element.

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, taking into account the following analysis of the entire program:

- All rules have the same triggering conditions, so
  - 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 "root pointers" need be created.
- All rules can safely be run on stale data, so data timestamps are not necessary.
- All rules can be run more than once on the same data, so rule timestamps are not necessary.
- Some variables are only set once, so there is no need for a check to see if the value actually changes.
- Neither objects nor pointers between objects are removed, so there is no need for any pointer management.

The C++ code for rule definitions and the definitions of functions that trigger the rules is over twice as long as the R++ rule definitions (224 lines to 111 lines), even though the C++ code does not perform many of the functions that the R++ code has to.

The R++ implementation is much faster than the C5 implementation on this example, largely due to the fact that the rules are small-branching-factor rules with a path length of 1. R++ consumes somewhat more storage than C5 when a set implementation with a large overhead is used; with a better set implementation R++ consumes somewhat less storage. Further, as the input size increases, R++'s speed advantage becomes even bigger.

The R++ implementation is slower 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, the results here provide evidence that R++ is much faster on certain kinds of applications than OPS-style rules and that R++ rules are similar in size and

[12] Trees can be implemented in C++ with either lists or sets, depending on whether order is important or not. Table 1 shows only the R++ rate for sets because trees in C5 are inherently unordered since they arise from joins among working memory elements. If order *is* important, it is easy to obtain in R++ simply by using lists instead of sets, though there is a performance penalty to be paid. In C5 and other OPS5 derivatives it is much more difficult and time-consuming to process a tree in an ordered way.

[13] Where R++ is slow, performance can be improved by tuning a model, for example, by connecting distant objects with shorter paths.

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	1.0	5	1.3	1.0	0.7	3.4	0.7	0.6	0.5
25	737	1.7	15	2.1	2.5	1.1	13.1	0.9	1.6	0.7
37	1839	2.3	30	2.8	4.2	1.4	27.4	1.2	3.1	0.8
50	3752	2.9	53	3.6	6.4	1.6	49.1	1.5	4.9	0.9

Table 2: Waltz resource consumption for C5, R++, and C++

number to OPS-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.

5.3 Comparison with Database Rule Systems

The work on rules in active object-oriented databases has some similarities to R++. Many active databases (e.g., Sentinel [2], Ode [16, 17], ADAM [12], REACH [4], SAMOS [15]) have rules that follow an event-condition-action paradigm [11]. 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. Instead, as discussed above, R++ rules are implicitly triggered in response to relevant change, including relevant construction. Events are thus implicitly specified in rule conditions, and event detection is implicitly performed by C++ code that is automatically output by the R++ translator. 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. 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 pre-processed 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. 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.

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. Again, 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" [2]. As shown by the R++ coding of the example in Figure 2, rules can both be declared inside classes and shared.

Other active database work is less similar to R++. The event-condition-action rules in POSTGRES [25] and Starburst [29], 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 [6] extends a C++ traditional rule-based language to allow rule-based applications to execute against standard relational databases.

Rules have also been added to deductive databases, including object-oriented deductive databases such as `Coral ++` [26]. Rules in deductive databases are run to satisfy queries and so have a very different purpose than do the rules in R++.

6 Conclusion

The R++ integration of rules into the object-oriented programming paradigm provides a mechanism for easily implementing changes to interdependent collections of objects. R++ extends C++ with a single programming construct—the *path-based* rule—as a new kind of class member, subservient to the object-oriented paradigm. Path-based rules strictly follow the inter-object paths in a domain model. Such member rules define the automatic behavior of a class, by monitoring data members for changes that could cause a rule's condition to succeed, then executing the rule's action. A rule monitors all instances of its class and, by inheritance, all instances of derived classes. By adding the path-based rules to the object-oriented programmer's repertoire, R++ expands the kinds and complexity of tractable applications, and can result in considerable savings of both code and expense. Our discussion of R++ has highlighted both the design choices behind the notion of path-based rules, and the algorithmic details needed to implement such data-driven computation.

Our analysis and comparisons of R++ have highlighted both its costs and benefits. The main cost of R++ compared to C++ is that R++ programs require more memory and take longer to compile and execute than equivalent C++ programs. In addition, there is a cost to learning R++, although 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. Rules can make programs clearer and more robust because a single rule to express a "multiple-object method" can replace several variants of the same logic scattered throughout standard object-oriented code. Also, rules relieve the programmer of the burden of explicit procedural control since rules are triggered automatically by relevant construction and relevant change.

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 contrast, there are many ways in which OPS5-style rules can violate the locality of reference designed into object-oriented domain models. In addition, because path-based rules are simpler than OPS5-style rules, R++ admits a simpler, and more efficient, rule evaluation mechanism. R++ thus offers performance advantages for many natural applications.

The ideas and implementation underlying R++ do not depend on the details of C++. We believe that it would be quite easy to produce a path-based-rule extension to any class-based object-oriented language that supports method inheritance and overriding and a notion of sets. The only change required to R++ would be to change the syntax of R++ rules to conform to the syntax of the underlying object-oriented language. For most such languages, the R++ implementation would also need only minor changes. We have been contemplating a JAVA version of R++, and have done some of the design work for it.

R++ is available free to research institutions. For further information, including the *R++ User Manual*, see the R++ home page on the world-wide web at

`http://www.research.att.com/sw/tools/r++`

or send e-mail to

`r++@orbit.hr.att.com`.

Acknowledgments

Jimi Crawford and Dan Dvorak were part of the original team of developers for R++. Much of the work reported here derives from their efforts in the team. The support of Ron Brachman, Raj Dube, Henry Kautz, Dick Machol, Brian Minnihan, Jennifer Thien, and Pramod Warty was instrumental to the research and development effort behind R++. We would also like to thank Amal Ahmed, Steve Correl, Johannes Ros, Walid Saba, Anoop Singhal, and Gary Weiss, for being some of the first users of R++ and Amal Ahmed for adding Standard Template Library functionality to R++. Dan Miranker provided the Waltz line-labeling OPS implementation.

References

[1] Patrick Albert. ILog Rules, embedding rules in C++: Results and limits. In *Position Papers for the OOPSLA'94 Embedded Object-Oriented Production Systems Workshop (EOOPS)*, October 1994.

[2] 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 Mangement of Data*, pages 99–108, 1993.

[3] David A. Brant, Timothy Grose, Bernie Lofaso, and Daniel P. Miranker. Effects of database size on rule system performance: Five case studies. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 287–296, 1991.

[4] 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.

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

[6] 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.

[7] 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.

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

[9] 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*, Portland, Oregon, August 1996. American Association for Artificial Intelligence.

[10] 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. American Association for Artificial Intelligence, July 1991.

[11] 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.

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

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

[14] Charles L. Forgy. RAL/C and RAL/C++: Rule-based extensions to C and C++. In *Position Papers for the OOPSLA'94 Embedded Object-Oriented Production Systems Workshop (EOOPS)*, October 1994.

[15] 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.

[16] 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.

[17] Narain H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In Widom and Ceri [30], pages 207–232.

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

[19] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings OOPSLA'90*, 1990.

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

[21] Haim Kilov and V. J. Harvey, editors. *Fifth Workshop on Specification of Behavioral Semantics, OOPSLA'96*, San Jose, California, 1996.

[22] 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.

[23] Anil Mishra, Johannes P. Ros, Anoop Singhal, Gary Weiss, Diane Litman, Peter F. Patel-Schneider, Daniel Dvorak, and James Crawford. R++: Using rules in object-oriented designs. In *Addendum to Object Oriented Programming Systems, Languages, and Applications*. Association for Computing Machinery, 1996.

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

[25] Spyros Potamianos and Michael Stonebraker. The POSTGRES rule system. In Widom and Ceri [30], pages 43–61.

[26] 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.

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

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

[29] Jennifer Widom. The Starburst rule system. In Widom and Ceri [30], pages 87–109.

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

[31] Patrick Henry Winston, editor. *Artificial Intelligence*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1984.

A Waltz Line Labeling Example

The Waltz line labeling algorithm takes a collection of edges that represent a 3-dimensional solid, joined at vertices, and assigns labels to these edges showing whether the edge is a boundary (B in the C5 rules), a front edge (+, or a back edge (-). Full details of the algorithm are available in [31].

A.1 C5 rules

The following rules implement the Waltz line-labeling algorithm. There are other rules in the C5 implementation, but they are used to set up the data structures and print the results, not to do the basic algorithm. The C5 rules are a minor modification of an OPS implementation of the algorithm written by Dan Miranker. This implementation can be obtained at
ftp://ftp.cs.utexas.edu/pub/ops5-benchmark-suite/

```
(literalize stage value)
(literalize edge p1 p2 joined label plotted)
(literalize junction p1 p2 p3 base_point type)

(p label_L
  (stage ^value labeling)
  (junction ^type L ^base_point <p1>)
  (edge ^p1 <p1> ^p2 <p2> ^label << + - >>)
  (edge ^p1 <p1> ^p2 <> <p2> ^label nil)
--> (modify 4 ^label B))

(p label_tee_A
  (stage ^value labeling)
  (junction ^type tee ^base_point <bp>
        ^p1 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p1> ^label nil)
  (edge ^p1 <bp> ^p2 <p3>)
--> (modify 3 ^label B)
    (modify 4 ^label B))

(p label_tee_B
  (stage ^value labeling)
  (junction ^type tee ^base_point <bp>
        ^p1 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p1>)
  (edge ^p1 <bp> ^p2 <p3> ^label nil)
--> (modify 3 ^label B)
    (modify 4 ^label B))

(p label_fork-1
  (stage ^value labeling)
  (junction ^type fork ^base_point <bp>)
  (edge ^p1 <bp> ^p2 <p1> ^label +)
  (edge ^p1 <bp> ^p2 {<p2> <> <p1>} ^label nil)
  (edge ^p1 <bp> ^p2 {<> <p2> <> <p1>})
--> (modify 4 ^label +)
    (modify 5 ^label +))

(p label_fork-2
  (stage ^value labeling)
  (junction ^type fork ^base_point <bp>)
  (edge ^p1 <bp> ^p2 <p1> ^label B)
  (edge ^p1 <bp> ^p2 {<p2> <> <p1>} ^label -)
  (edge ^p1 <bp> ^p2 {<> <p2> <> <p1>} ^label nil)
--> (modify 5 ^label B))

(p label_fork-3
  (stage ^value labeling)
  (junction ^type fork ^base_point <bp>)
  (edge ^p1 <bp> ^p2 <p1> ^label B)
  (edge ^p1 <bp> ^p2 {<p2> <> <p1>} ^label B)
  (edge ^p1 <bp> ^p2 {<> <p2> <> <p1>} ^label nil)
--> (modify 5 ^label -))

(p label_fork-4
  (stage ^value labeling)
  (junction ^type fork ^base_point <bp>)
  (edge ^p1 <bp> ^p2 <p1> ^label -)
  (edge ^p1 <bp> ^p2 {<p2> <> <p1>} ^label -)
  (edge ^p1 <bp> ^p2 {<> <p2> <> <p1>} ^label nil)
--> (modify 5 ^label -))

(p label_arrow-1A
  (stage ^value labeling)
  (junction ^type arrow ^base_point <bp>
        ^p1 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p1>
              ^label {<label> << B - >>})
  (edge ^p1 <bp> ^p2 <p2> ^label nil)
  (edge ^p1 <bp> ^p2 <p3>)
--> (modify 4 ^label +)
    (modify 5 ^label <label>))

(p label_arrow-1B
  (stage ^value labeling)
  (junction ^type arrow ^base_point <bp>
        ^p1 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p1>
              ^label {<label> << B - >>})
  (edge ^p1 <bp> ^p2 <p2>)
  (edge ^p1 <bp> ^p2 <p3> ^label nil)
--> (modify 4 ^label +)
    (modify 5 ^label <label>))

(p label_arrow-2A
  (stage ^value labeling)
  (junction ^type arrow ^base_point <bp>
        ^p1 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p3>
              ^label {<label> << B - >>})
  (edge ^p1 <bp> ^p2 <p2> ^label nil)
  (edge ^p1 <bp> ^p2 <p1>)
--> (modify 4 ^label +)
    (modify 5 ^label <label>))

(p label_arrow-2B
  (stage ^value labeling)
  (junction ^type arrow ^base_point <bp>
        ^p1 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p3>
              ^label {<label> << B - >>})
  (edge ^p1 <bp> ^p2 <p2>)
  (edge ^p1 <bp> ^p2 <p1> ^label nil)
--> (modify 4 ^label +)
    (modify 5 ^label <label>))

(p label_arrow-3A
  (stage ^value labeling)
  (junction ^type arrow ^base_point <bp>
        ^p1 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p1> ^label +)
```

```
  (edge ^p1 <bp> ^p2 <p2> ^label nil)
  (edge ^p1 <bp> ^p2 <p3>)
--> (modify 4 ^label -)
    (modify 5 ^label +))

(p label_arrow-3B
  (stage ^value labeling)
  (junction ^type arrow ^base_point <bp>
        ^p1 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p1> ^label +)
  (edge ^p1 <bp> ^p2 <p2>)
  (edge ^p1 <bp> ^p2 <p3> ^label nil)
--> (modify 4 ^label -)
    (modify 5 ^label +))

(p label_arrow-4A
  (stage ^value labeling)
  (junction ^type arrow ^base_point <bp>
        ^p1 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p3> ^label +)
  (edge ^p1 <bp> ^p2 <p2> ^label nil)
  (edge ^p1 <bp> ^p2 <p1>)
--> (modify 4 ^label -)
    (modify 5 ^label +))

(p label_arrow-4B
  (stage ^value labeling)
  (junction ^type arrow ^base_point <bp>
        ^p1 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p3> ^label +)
  (edge ^p1 <bp> ^p2 <p2>)
  (edge ^p1 <bp> ^p2 <p1> ^label nil)
--> (modify 4 ^label -)
    (modify 5 ^label +))

(p label_arrow-5A
  (stage ^value labeling)
  (junction ^type arrow ^base_point <bp>
        ^p1 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p2> ^label -)
  (edge ^p1 <bp> ^p2 <p1>)
  (edge ^p1 <bp> ^p2 <p3> ^label nil)
--> (modify 4 ^label +)
    (modify 5 ^label +))


(p label_arrow-5B
  (stage ^value labeling)
  (junction ^type arrow ^base_point <bp>
        ^p1 <p1> ^p2 <p2> ^p3 <p3>)
  (edge ^p1 <bp> ^p2 <p2> ^label -)
  (edge ^p1 <bp> ^p2 <p1> ^label nil)
  (edge ^p1 <bp> ^p2 <p3>)
--> (modify 4 ^label +)
    (modify 5 ^label +))
```

A.2 Common Code for R++ and C++

The following file is common to both the R++ and C++ implementation. It defines the class `Point` that is used during the set-up for the line labeling algorithm.

A.2.1 waltz.H

```
#include <Set.h>

enum EdgeLabel { nil , plus, minus, boundary };

class Vertex;
class Edge;

class Point {
public:
  Point(int loc) : location(loc), edges(), v(0) {}
  int operator==(const Point &o)
        { return this->location==o.location; }
  operator int() const { return location; }
  void addEdge(Point * pt1,Point * pt2) const;
  Vertex * makeVertex();
private:
  int location;
  Set_of_p<Edge> edges;
  Vertex * v;
};
```

A.3 R++ Rule Code

The following two files implement the R++ rules for the algorithm. The R++ rules have been given names analogous to the names of the C5 rules that they mirror.

The file `waltz.rh` is processed by R++ and turned into a `.h` file that defines the classes that have rules defined on them, including the declarations of the methods used for the rules. The file `waltz.rC` is also processed by R++, but is turned into a `.C` file that has the definitions of the rule methods.

A.3.1 waltz.rh

```
#include "waltz.H"

class Edge {
public:
  Edge(Point * pt1, Point * pt2) : points(pt1,pt2)
              { set_label(nil); }
  const Point* otherPoint(const Point* point)const;
  EdgeLabel get_label() const { return label; }
  monitored EdgeLabel label;
private:
  Set_of_p<Point> points;
};

class Vertex {
protected:
  Vertex() { }
public:
  virtual void cornerLabel();
};

class Ell : public Vertex {
public:
  Ell(Set_of_p<Edge> edgs) {
    Set_of_piter<Edge> iter_edge =
        Set_of_piter<Edge>(edgs);
    for (Edge *edge1 = iter_edge.next();
         edge1; edge1 = iter_edge.next()) {
      insert_edges(edge1);
```

```
    }
  }
  virtual void cornerLabel();
private:
  monitored Set_of_p<Edge> edges;
rule label;
};

class Tee : public Vertex {
public:
  Tee(Edge * barb1, Edge * shaft, Edge * barb2)
  {
      set_upright(shaft);
      set_side1(barb1);
      set_side2(barb2);
  }
private:
  monitored Edge * upright;
  monitored Edge * side1;
  monitored Edge * side2;
rule label;
};

class Fork : public Vertex {
public:
  Fork(Edge * barb1, Edge * shaft, Edge * barb2) {
      insert_sides(shaft);
      insert_sides(barb1);
      insert_sides(barb2);
  }
private:
  monitored Set_of_p<Edge> sides;
rule label_1;
rule label_2;
rule label_3;
rule label_4;
};

class Arrow : public Vertex {
public:
  Arrow(Edge * barb1, Edge * shaft, Edge * barb2)
  {
      set_center(shaft);
      insert_sides(barb1);
      insert_sides(barb2);
  }
  virtual void cornerLabel();
private:
  monitored Edge * center;
  monitored Set_of_p<Edge> sides;
rule label_12;
rule label_34;
rule label_5;
};
```

A.3.2 waltz.rC

```
$include "waltz.rh"

Vertex * Point::makeVertex() {
  if ( this->v ) {
    return this->v;
  } else if ( this->edges.size()==3 ) {
    return this->v =
        make_3_junction(this,this->edges);
  } else {
    return this->v = new Ell(this->edges);
  }
}

rule Ell::label {
  Edge * edg1 @ this->edges &&
  ( edg1->label==plus || edg1->label==minus ) &&
  Edge * edg2 @ this->edges &&
  ( edg2->label==nil )
=> edg2->set_label(boundary);
}

rule Tee::label {
  Edge * sde1 = this->side1 &&
  Edge * sde2 = this->side2 &&
  ( sde1->label==nil  || sde2->label==nil )
=> sde1->set_label(boundary);
   sde2->set_label(boundary);
}

rule Fork::label_1 {
  Edge * sde1 @ this->sides &&
  sde1->label==plus
=> Set_of_piter<Edge> sdes = this->sides;
   for ( Edge * sde = sdes.next();
         sde; sde = sdes.next() ) {
     sde->set_label(plus);
   };
}

rule Fork::label_2 {
  Edge * sde1 @ this->sides &&
  sde1->label==boundary &&
  Edge * sde2 @ this->sides &&
  sde2 != sde1 &&
  sde2->label==minus
=> Set_of_piter<Edge> sdes = this->sides;
   for ( Edge * sde = sdes.next();
         sde; sde = sdes.next() ) {
     if (sde->label==nil) sde->set_label(boundary);
   }
}

rule Fork::label_3 {
  Edge * sde1 @ this->sides &&
  sde1->label==boundary &&
  Edge * sde2 @ this->sides &&
  sde2 != sde1 &&
  sde2->label==boundary
=> Set_of_piter<Edge> sdes = this->sides;
   for ( Edge * sde = sdes.next();
         sde; sde = sdes.next() ) {
     if (sde->label==nil) sde->set_label(minus);
   }
}

rule Fork::label_4 {
  Edge * sde1 @ this->sides &&
  sde1->label==minus &&
  Edge * sde2 @ this->sides &&
  sde2 != sde1 &&
  sde2->label==minus
=> Set_of_piter<Edge> sdes = this->sides;
```

```
   for ( Edge * sde = sdes.next();
         sde; sde = sdes.next() ) {
     if (sde->label==nil) sde->set_label(minus);
   }
}

rule Arrow::label_12 {
  Edge * sde @ this->sides &&
  (sde->label==boundary || sde->label==minus) &&
  Edge * osde @ this->sides &&
  osde != sde &&
  Edge * centr = this->center &&
  (osde->label==nil || centr->label==nil)
=> centr->set_label(plus);
   osde->set_label( sde->label );
}

rule Arrow::label_34 {
  Edge * sde @ this->sides &&
  (sde->label==plus) &&
  Edge * osde @ this->sides &&
  osde != sde &&
  Edge * centr = this->center &&
  (osde->label==nil || centr->label==nil)
=> centr->set_label(minus);
   osde->set_label(plus);
}

rule Arrow::label_5 {
  Edge * centr = this->center &&
  centr->label==minus &&
  Edge * sde @ this->sides &&
  (sde->label==nil )
=> Set_of_piter<Edge> sdes = this->sides;
   for ( Edge * sde = sdes.next();
         sde; sde = sdes.next() ) {
     sde->set_label(plus);
   }
}
```

A.4 C++ "Rule" Code

The following two files correspond closely to the two R++ files above.

A.4.1 waltzC.H

```
#include "waltz.H"

class Edge {
public:
  Edge(Point * pt1, Point * pt2)
              : points(pt1,pt2), vertices()
        { set_label(nil); }
  const Point* otherPoint(const Point* point)const;
  void addVertex(Vertex * v) {vertices.insert(v);}
  void set_label(EdgeLabel l);
  EdgeLabel get_label() const { return label; }
private:
  EdgeLabel label;
  Set_of_p<Point> points;
  Set_of_p<Vertex> vertices;
};

class Vertex {
protected:
  Vertex() { }
public:
  virtual void cornerLabel();
  virtual void labelEdges() = 0;
};

class Ell : public Vertex {
public:
  Ell(Set_of_p<Edge> edgs) : edges() {
    Set_of_piter<Edge> iter_edge =
                Set_of_piter<Edge>(edgs);
    for (Edge *edge1 = iter_edge.next();
         edge1; edge1 = iter_edge.next()) {
      insert_edges(edge1);
    }
  }
  virtual void cornerLabel();
  virtual void labelEdges();
  void insert_edges(Edge *);
private:
  Set_of_p<Edge> edges;
};

class Tee : public Vertex {
public:
  Tee(Edge * barb1, Edge * shaft, Edge * barb2) {
      upright = shaft;
      side1 = barb1;
      side2 = barb2;
      this->labelEdges();
  }
  virtual void labelEdges();
  void set_upright(Edge *);
  void set_side1(Edge *);
  void set_side2(Edge *);
private:
  Edge * upright;
  Edge * side1;
  Edge * side2;
};

class Fork : public Vertex {
public:
  Fork(Edge * barb1, Edge * shaft, Edge * barb2) {
      insert_sides(shaft);
      insert_sides(barb1);
      insert_sides(barb2);
  }
  virtual void labelEdges();
  void insert_sides(Edge *);
private:
  Set_of_p<Edge> sides;
};

class Arrow : public Vertex {
public:
  Arrow(Edge * barb1, Edge * shaft, Edge * barb2)
                                  : sides() {
    set_center(shaft);  // must be first
    insert_sides(barb1);
    insert_sides(barb2);
  }
  virtual void cornerLabel();
```

```
  virtual void labelEdges();
  void set_center(Edge *);
  void insert_sides(Edge *);
private:
  Edge * center;
  Set_of_p<Edge> sides;
};
```

A.4.2 waltzC.rules.C

This file implements the effects of the R++ rules in C++, under the following assumptions:

- All rules have the same triggering constraints, so rule triggering can be very simple.
- All rules are additive, so it doesn't matter if rules run on old data.
- All rules are idempotent, so it doesn't matter if rules run more than once on the same data.
- Some variables are only set once, so there is no need to check for actual changes.
- No links are deleted, so there is no need for maintainance of backpointers.
- No objects are deleted, so there is no need to check to see if objects still exist nor is there any need to remove backpointers.

```
#include "waltzC.H"

void Edge::set_label(EdgeLabel l) {
  if ( label != l ) {
    label = l;
    Set_of_piter<Vertex> iter = this->vertices;
    for ( Vertex * v = iter.next();
          v; v = iter.next() ) {
      v->labelEdges();
    }
  }
}

Vertex * Point::makeVertex() {
  if ( this->v ) {
    return this->v;
  } else if ( this->edges.size()==3 ) {
    Vertex * vertex =
        make_3_junction(this,this->edges);
    Set_of_piter<Edge> edgs = this->edges;
    for ( Edge * edge = edgs.next();
          edge; edge = edgs.next() ) {
      edge->addVertex(vertex);
    }
    return this->v = vertex;
  } else {
    Vertex * vertex = new Ell(this->edges);
    Set_of_piter<Edge> edgs = this->edges;
    for ( Edge * edge = edgs.next();
          edge; edge = edgs.next() ) {
      edge->addVertex(vertex);
    }
    return this->v = vertex;
  }
}

void Ell::insert_edges(Edge * e) {
  if ( ! this->edges.contains(e) ) {
    this->edges.insert(e);
    this->labelEdges();
  }
}

void Ell::labelEdges() {
  if ( this->edges.size()==2 ) {
    Set_of_piter<Edge> iter = this->edges;
    Edge * edg1 = iter.next();
    Edge * edg2 = iter.next();
    if ( ( edg1->get_label()==plus ||
           edg1->get_label()==minus ) &&
         ( edg2->get_label()==nil ) ) {
      edg2->set_label(boundary);
    }
    if ( ( edg2->get_label()==plus ||
           edg2->get_label()==minus ) &&
         ( edg1->get_label()==nil ) ) {
      edg1->set_label(boundary);
    }
  }
}

void Tee::set_upright(Edge * e) {
  this->upright = e;
  this->labelEdges();
}
void Tee::set_side1(Edge * e) {
  this->side1 = e;
  this->labelEdges();
}
void Tee::set_side2(Edge * e) {
  this->side2 = e;
  this->labelEdges();
}

void Tee::labelEdges() {
  if ( this->side1 && this->side2 &&
       ( this->side1->get_label()==nil ||
         this->side2->get_label()==nil ) ) {
    this->side1->set_label(boundary);
    this->side2->set_label(boundary);
  }
}

void Fork::insert_sides(Edge * e) {
  if ( ! this->sides.contains(e) ) {
    this->sides.insert(e);
    this->labelEdges();
  }
}

void Fork::labelEdges() {
  // label_1
  Set_of_piter<Edge> iter1 = this->sides;
  for ( Edge * sde1 = iter1.next();
        sde1; sde1 = iter1.next() ) {
    if ( sde1->get_label()==plus ) {
      Set_of_piter<Edge> sdes = this->sides;
      for ( Edge * sde = sdes.next();
            sde; sde = sdes.next() ) {
```

```
      sde->set_label(plus);
    }
  }
}
// label_2
Set_of_piter<Edge> iter2 = this->sides;
for ( Edge * sdeb = iter2.next();
      sdeb; sdeb = iter2.next() ) {
  if ( sdeb->get_label()==boundary ) {
    Set_of_piter<Edge> iter = this->sides;
    for ( Edge * sdeo = iter.next();
          sdeo; sdeo = iter.next() ) {
      if ( sdeo->get_label()==minus ) {
        Set_of_piter<Edge> sdes = this->sides;
        for ( Edge * sde = sdes.next();
              sde; sde = sdes.next() ) {
          if ( sde->get_label()==nil) {
            sde->set_label(boundary);
          }
        }
      }
    }
  }
}
// label_3
Set_of_piter<Edge> iter3 = this->sides;
for ( Edge * sdec = iter3.next();
      sdec; sdec = iter3.next() ) {
  if ( sdec->get_label()==boundary ) {
    Set_of_piter<Edge> iter = this->sides;
    for ( Edge * sdeo = iter.next();
          sdeo; sdeo = iter.next() ) {
      if ( sdeo != sdec &&
           sdeo->get_label()==minus ) {
        Set_of_piter<Edge> sdes = this->sides;
        for ( Edge * sde = sdes.next();
              sde; sde = sdes.next() ) {
          if ( sde->get_label()==nil) {
             sde->set_label(minus);
          }
        }
      }
    }
  }
}
// label 4
Set_of_piter<Edge> iter4 = this->sides;
for ( Edge * sde4 = iter4.next();
      sde4; sde4 = iter4.next() ) {
  if ( sde4->get_label()==minus ) {
    Set_of_piter<Edge> iter = this->sides;
    for ( Edge * sde5 = iter.next();
          sde5; sde5 = iter.next() ) {
      if ( sde5->get_label()==minus ) {
        Set_of_piter<Edge> sdes = this->sides;
        for ( Edge * sde = sdes.next();
              sde; sde = sdes.next() ) {
          if ( sde->get_label()==nil) {
            sde->set_label(minus);
          }
        }
      }
    }
  }
}
}

void Arrow::set_center(Edge * e) {
  this->center = e;
  this->labelEdges();
}
void Arrow::insert_sides(Edge * e) {
  if ( ! this->sides.contains(e) ) {
    this->sides.insert(e);
    this->labelEdges();
  }
}

void Arrow::labelEdges() {
  if ( this->sides.size()==2 && this->center) {
    Set_of_piter<Edge> iter = this->sides;
    Edge * side1 = iter.next();
    Edge * side2 = iter.next();
    // label_12
    if ( ( side1->get_label()==boundary ||
           side1->get_label()==minus ) &&
         ( side2->get_label()==nil ||
           this->center->get_label()==nil ) ) {
      this->center->set_label(plus);
      side2->set_label(side1->get_label());
    }
    if ( ( side2->get_label()==boundary ||
           side2->get_label()==minus ) &&
         ( side1->get_label()==nil ||
           this->center->get_label()==nil ) ) {
      this->center->set_label(plus);
      side1->set_label(side2->get_label());
    }
    // label_34
    if ( ( side1->get_label()==plus ) &&
         ( side2->get_label()==nil ||
           this->center->get_label()==nil ) ) {
      this->center->set_label(minus);
      side2->set_label(plus);
    }
    if ( ( side2->get_label()==plus ) &&
         ( side1->get_label()==nil ||
           this->center->get_label()==nil ) ) {
      this->center->set_label(minus);
      side1->set_label(plus);
    }
  }
  // label_5
  if ( this->center &&
       this->center->get_label()==minus ) {
    Set_of_piter<Edge> iter = this->sides;
    for ( Edge * side = iter.next();
        side; side = iter.next() ) {
      if ( side->get_label()==nil ) {
        side->set_label(plus);
      }
    }
  }
}
```