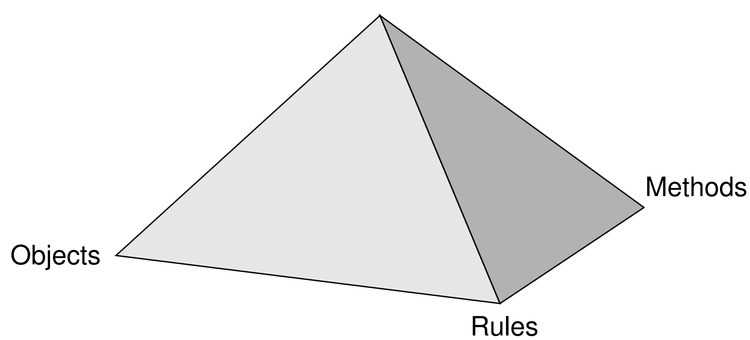
R++ User Manual



AT&T Bell Laboratories
480 Red Hill Road
Middletown, NJ 07748-3052
USA

March 1995

R++ User Manual

Contents

Example Applications 3-1

Rule Syntax and Semantics 4-1

Glossary G-1

Preface

Goals

The first goal of this manual is to illustrate the benefits of data-driven computation in object-oriented designs. "Data-driven computation" means that certain actions are triggered *automatically* in response to changes in monitored objects. This manual shows several examples of how data-driven computation can be applied in real-world problems and how it can make programs shorter, easier to understand, and more robust.

The second goal is to teach the syntax and semantics of R++ as an extension to C++. R++ is relatively easy to learn because it builds upon what programmers already know about the C++ class system. Specifically, R++ adds an IF–THEN rule construct as a new kind of class member. A rule is like a "automatic" member function whose IF part monitors specific data members and whose THEN part executes automatically when the condition is satisfied.

Audience

This manual is intended for C++ programmers who are familiar with object-oriented design. The benefits of using rules will be most apparent to designers and developers of applications involving complex inter-object relationships. Familiarity with *C++ Standard Components* is helpful in understanding some code examples, but is not essential. Familiarity with other rule-based languages such as OPS5 or C5 is *not* necessary.

Using this Manual

This manual is designed to help you get started quickly. The first three chapters provide enough information to enable you to begin using rules in otherwise ordinary C++ applications. The manual emphasizes *examples* as the way to learn about useful roles for rules and features of R++.

Later chapters define the rule construct in more detail and explain the finer points of how rules are integrated into the class mechanism. An extensive set of programming tips demonstrates recommended practices for using rules and avoiding common pitfalls. For the serious student, a complete and non-trivial application using R++ is described in an appendix.

R++ Information and Support

Information about R++ is available on the world-wide web, accessible from this universal resource locator:

`http://www.research.att.com/orgs/ssr/research/r++`

This is the preferred mechanism for obtaining information about R++, including news, how to obtain R++, release notes, planned enhancements, known bugs, and published technical papers. Alternatively, requests and comments can be sent via e-mail to:

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

Free Research Software License

To encourage further research in multi-paradigm programming languages, R++ is available free to research institutions upon execution of AT&T's research software licensing agreement. To obtain a copy of this agreement, contact R++ support as described above.

Commercial Availability

On October 21, 1994 AT&T filed a patent application with the U.S. Patent Office titled "Integrating Rules into Object-Oriented Programming Systems". AT&T plans to license the technology covered by this pending patent to commercial vendors of object-oriented languages such as C++ and Smalltalk. For information on the status of commercial licensing, contact R++ support as described above.

Acknowledgments

R++ is a result of a collaboration between research and development within AT&T, made possible by very supportive, forward-looking management. In particular, we thank Ron Brachman and Mike Wish of the Software and Systems Research Center for supporting the research effort behind R++, and we thank Raj Dube, Pramod Warty, and Dick Machol of the Operations Technology Center for supporting the development effort behind R++ and encouraging the release of R++ for academic research.

The thoughtful criticisms and suggestions of many friends and colleagues have helped to shape this description of R++. In particular, we thank (... to be filled in ...).

James Crawford
Daniel Dvorak
Diane Litman
Anil Mishra
Peter Patel-Schneider

Introduction

1

R++ is an extension of C++ that bridges the gap between object-oriented programming and rule-based programming. Data-driven rules are like demons that react to changes in objects.

Overview

R++ is an extension to C++ that bridges the gap between object-oriented programming and rule-based inference. C++ classes contain two kinds of members: data members and member functions. R++ extends the C++ class mechanism with a new kind of member, a rule, enabling object-oriented applications to employ data-driven computation.

Data-driven computation is an important capability because it enables program designs to be more declarative and less procedural, emphasizing *what* rather than *how* or *when*. With rules it is possible to design and program parts of an application at a higher level of abstraction, with less concern about flow of control.

Rules can be employed for a variety of purposes, such as enforcing invariants in a domain model, auditing complex data structures, monitoring the state of an equipment model as external events occur, or checking design constraints as a user interacts with a CAD design tool. The benefit, in all cases, is that rules are applied automatically, without procedural invocation.

As an extension to C++, R++ fits comfortably with C++ concepts and practices. Rules are relatively easy to learn since the syntax is similar and the behavior is much like a “reactive” member function. Rules can be introduced gradually into otherwise conventional C++ applications. Files that contains rules are translated into C++, which is then compiled using any C++ compiler.

Why Data-Driven Computation?

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 difficulty of maintaining model integrity has grown. For example, maintaining an invariant involving two related objects may require 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.

The addition of a data-driven programming construct simplifies the task of maintaining model integrity in two important ways. First, it enables invariants and constraints to be stated explicitly in a single place, rather than having several variants scattered in multiple places. This makes code easier to understand and modify. Second, because it is data-driven, the invariant/constraint is reevaluated automatically when relevant changes are made to object memory. This relieves the programmer of some of the burden of explicit procedural control. In particular, the application's procedural logic is no longer cluttered with code for maintaining model integrity.

More broadly, rules act as "monitors" of a model's state, observing changes to object memory and reacting when a condition is satisfied. For example, applications that monitor a physical system (such as a telephone switching system) can use rules to detect and react to critical states. Applications that are primarily reactive (such as an elevator controller) can use rules to express state transitions. Applications that apply business policies (such as in loan analysis) or engineering guidelines (such as in equipment configuration) can use rules to express policies/guidelines and apply them automatically when each situation arises. As a software engineering aid, rules can be used as a kind of non-procedural exception mechanism for detecting and handling illegal states. Data-driven rules are well-suited to such model-monitoring tasks because they can express situation–action directives without complicating an application's procedural logic.

Rules

A rule is a <u>new kind of class member</u>. In many respects a rule is like a member function: it is declared in a class, inherited like member functions, defined using largely C++ syntax, contains an implicit "`this`" variable, directly accesses class data members, and respects the public/private access controls. The key difference is that a rule is never "called"; it is triggered automatically when ordinary procedural code changes objects that the rule monitors. Thus, a rule is like an automaton or demon or monitor that reacts to changes in the state of object memory.

Rules in R++ are <u>data-driven</u> in that they are triggered automatically by changes to C++ class objects. Rule triggering is guaranteed provided that all changes to monitored data members are made through stereotyped access functions.

Rules in R++ are <u>path-based</u> in that a rule condition can follow a path of inter-object pointers connecting related objects. Rules thus provide a mechanism for reacting to states involving complex inter-object relationships.

Design Objectives

Owing to the dominant position of C++ within many of AT&T's software development organizations, R++ was planned from the outset as an extension to C++. The primary design goal was to integrate data-driven rules with C++ objects in a way that preserved the strengths of both forms of programming, i.e., the modularity, information-hiding, and locality-of-effect of object-oriented programming, plus the concise and declarative form of rules having clear semantics. Since the primary audience for R++ is C++ programmers, most of the design trade-offs have been made in favor of fitting naturally within the C++ model.

The resulting design supports these objectives in several ways: rules operate directly on C++ objects (without a separate working memory), are triggered through a natural mechanism (access functions), respect the access control mechanisms of the C++ class system, and utilize the inter-object relationships expressed in a domain model. Also, the syntax and semantics of R++ conform to many familiar concepts in C++: rule syntax has been made as close to C++ as possible, rules are like reactive member functions and are inherited and overridden in the same way as member functions, rules are triggered in specific-to-general order, and rules fire in a depth-first forward-chaining order. Finally, rule-containing files are translated into C++, allowing familiar software development tools to continue being used.

C++ owes much of its success to its evolutionary approach. As an extension of C, C++ simplified the transition to object-oriented programming for many programmers. In the same spirit, we hope that the evolutionary approach of R++ will simplify the adoption of data-driven computation among C++ programmers.

Comparison to OPS5 and C5

R++ differs from OPS-based languages such as OPS5 and C5 in several respects:

- R++ rules directly and natively access C++ objects. This makes it much easier to exploit rule-based reasoning in mainstream C++ applications. To do the same thing in C5 requires importing and exporting data between the C++ object memory and C5's working memory, with explicit hand-off of control between rules and procedural code.

- R++ rule syntax conforms to C++ syntax as close as possible, thus making rules more easily interpreted by C++ programmers, compared to the Lisp-like syntax of OPS5 and C5.
- Rule triggering is more tightly focused in R++ than in OPS5 and C5. R++ rules react only to changes in the monitored data members of an object, whereas OPS5/C5 rules react to any change in a working memory element (WME), even if the changed attribute is irrelevant to the rule.
- R++ rules allow inference on many set-valued and list-valued data members of a class. OPS5/C5 restricts WMEs to one vector attribute.
- There is no organizing principle for rules in OPS5/C5, whereas rules in R++ are organized by class. This makes it somewhat easier for a programmer to maintain an application containing many rules.
- OPS-style rules are more expressive in that they perform general pattern matching among unrelated objects, whereas R++ rules strictly follow access paths (pointers) among related objects. This is similar to the difference between a relational DBMS and an object-oriented DBMS; a RDBMS must perform "joins" while an OO DBMS follows pointers. Since pointer-following is simpler than "joins", R++'s inference algorithm is simpler than the RETE algorithm used in pattern-matching languages.

Motivations

The design of R++ was originally motivated by the need to diagnose equipment malfunctions in the principled way that an engineer employs — by tracing through a schematic diagram from the location of symptoms to components whose misbehavior could have caused the symptoms. This kind of "schematic reasoning" (sometimes called "model-based reasoning") applies a few principles of troubleshooting to a model of equipment design, rather than numerous special-case heuristics about symptoms and faults.

The emergence of object-oriented design—where the important physical objects, concepts, and relationships of the problem domain are explicitly modeled as objects and inter-object pointers—has provided a foundation of good design practice in which model-based reasoning and data-driven inference can be exploited. This manual shows where and how rules can be exploited, emphasizing that rules play specialized roles, and that they should work in concert with procedural code.

The decision to create R++ was also motivated by the difficulty in building hybrid C5–C++ applications, where programmers must know two very different languages and arrange to keep C++ object memory synchronized with C5 working memory. This practical difficulty has limited the use of rule-based programming in applications implemented primarily in C++.

What R++ is Not

R++ is <u>not</u> an “expert system shell”. Such shells generally promote rules as the primary method of representing “heuristic knowledge” within a proprietary language, and offer an API to a mainstream language such as C or C++ for interfacing with the rest of the world. In contrast, R++ is an *extension* of a mainstream object-oriented language in which rules provide a data-driven programming construct for enforcing invariants, checking constraints, monitoring for important states, etc.

R++ rules are <u>not</u> statements of logic. R++ rules are *procedural* rules in that their semantics are defined procedurally. R++ rules live within the C++ environment where their triggering, evaluation, and firing are subject to a procedural flow.

Origins of R++

The kind of path-based rules used in R++ first appeared in 1990 as “access-limited” rules in the Algernon implementation of Access-Limited Logic in research conducted by James Crawford and Benjamin Kuipers at The University of Texas at Austin. Although this research focused on the trade-off between tractable inference and expressive power, path-based rules also provided an excellent model for integrating rules with objects.

The need to bridge the gap between object-oriented programming and rule-based reasoning was recognized at AT&T’s Network Services Division in mid 1992 based on experience in building intelligent applications for monitoring telephone switching equipment. A subsequent evaluation of existing “objects + rules” programming languages—conducted jointly at AT&T Bell Laboratories by the AI Principles Research Department and the Operations Technology Center—failed to find an existing language that integrated rules in a manner that was as type-safe as possible in C++, that respected the access control mechanisms of C++, and that supported inference using the inter-object relationships designed into a domain model.

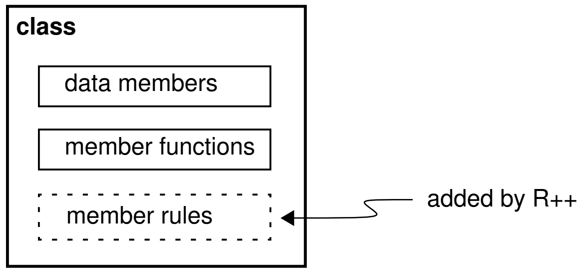
Following the decision in early 1994 to build a rule-based extension to C++, R++ took the basic ideas of path-based rules and rule attachment from Algernon (which had been implemented in Common Lisp) and applied the many changes required for seamless integration with C++. A significant departure from Algernon is seen in R++'s compiling method, where rules are translated into the host language (C++) with no run-time inference engine needed. This novel compiling method (patent applied for) yields a very efficient rule system.

Getting Started 2

• A minimal description of rule syntax and semantics.
• A small but complete R++ program.
• How to exploit rules in ordinary C++ programs.

Rule: A New Kind of Class Member

A class in C++ has data members and member functions; R++ extends the class concept to also include member rules.



Like other class members, rules are declared within a class definition and are inherited by derived classes.

```
class Rectangle {
private:
  monitored int height, width;
  int area;

public:
  Rectangle(int h, int w) : height(h), width(w) {};

  rule update_area;
```

rule declaration

Member rules add automatic (non-procedural) behavior to a class.

Access specifiers (public, private, protected) do *not* apply to rules since rules cannot be accessed, *i.e.*, they cannot be referenced like data members or called like member functions,

Rule: A Condition–Action Pair

A rule is like an if–then statement; if its condition is satisfied then its action is executed. The condition is a conjunction of tests and bindings; the action consists of ordinary C++ statements. An arrow “=>” separates condition from action.

class name   rule name

condition

action

```
rule Rectangle::update_area {
   height > 0  &&
   width  > 0
=>
   area = height * width;
}
```

“&&” separates the conjuncts

This rule enforces the invariant that a rectangle’s area = height × width. Like a member function, a rule has an implicit ‘`this`’ variable, *i.e.,* “`height`” is equivalent to “`this->height`”.

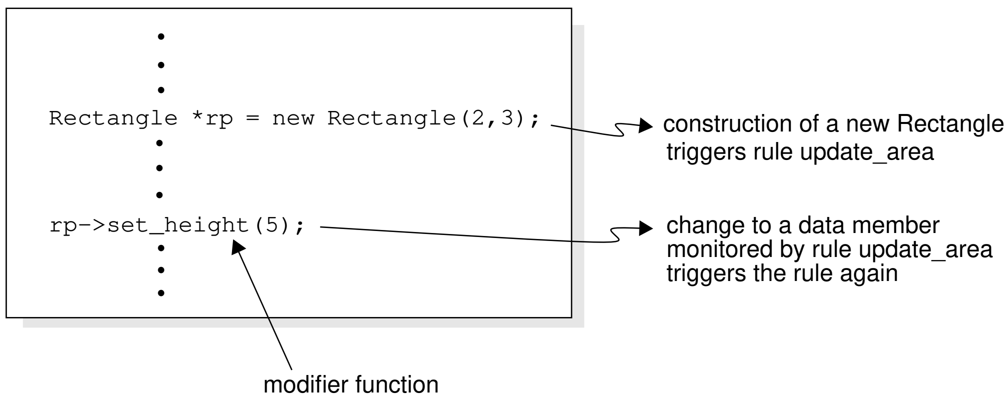
Rules React to Changes in Objects

A rule has no arguments and cannot be called. Instead, a rule monitors all objects of its class for changes in the values of data members appearing in its condition. Such data members should be declared as “`monitored`” to ensure that all changes are made through special “modifier functions”.

monitored data members

```
class Rectangle {
private:
  monitored int height, width;
   ...
};
```

When a monitored data member’s value is changed via a modifier function, the rule condition is reevaluated and, if satisfied, the rule’s action is executed.



Modifier Functions

To guarantee rule triggering, all changes to monitored data members must be made using R++-supplied modifier functions, *e.g.,* `set_height` and `set_width`. Modifier functions have stereotypic names:

— For a single-valued data member (*e.g.,* `int`, `float`) named *x*, the modifier function is named **`set_x`**.

— For a set of pointers named *x*, the modifier functions are named **`insert_x`** and **`remove_x`**.

— For a list of pointers named *x*, the modifier functions are named:

— **get_*x*** (remove and return element from head of list),

— **unget_*x*** (insert element onto head of list),

— **put_*x*** (append element to tail of list), and

— **unput_*x*** (remove and return element from tail of list).

R++ automatically defines these modifier functions for monitored data members, giving them the same public/protected/private access as their associated data members.

Inter-Object Rules

A rule condition can access not only '`this`' object but also neighboring objects, *i.e.,* objects accessible via inter-object pointers. A neighbor is accessed by binding a rule variable to the neighbor's address. Below, the first rule binds a rule variable to a single-valued data member (a "simple binding"); the second rule binds a rule variable to each element of a set of pointers (a "branch binding").

A simple binding succeeds only if the data member is non-null.

Simple Binding (accesses a single related object)

Branch Binding (rule branches on each member of the set)

A branch binding succeeds for each non-null element of a set or list of pointers.

```
class Person {
private:
  String name;
  monitored int age;
  monitored Person *spouse;
  monitored Set_of_p<Person> children;

  rule reflexive_spouse;
  rule child_age_check;
};

// If X's spouse is Y,
// then Y's spouse is X.
rule Person::reflexive_spouse {
  Person *s = spouse
=>
  s->set_spouse(this);
}

// Check for child older than parent.
rule Person::child_age_check {
  Person *child @ children  &&
  child->age > this->age
=>
  cout << "Error: " << child-name
       << " is older than parent "
       << this->name << endl;
}
```

The at-sign "`@`" in a branch binding can be read as "in", e.g., "for each child in children …".

Universal and Existential Quantifiers

A rule condition on a set or list of pointers can specify that all elements satisfy some test (universal quantification) or that at least one element satisfy some test (existential quantification).

universal quantifier

existential quantifier

```
// If all of a person's children are older than 17,
// then output an announcement.
rule Person::all_adult_children {
  all Person *c @ children { c->age >= 17 }
=>
  cout << name << " has all adult children\n";
}


// If there exists a child who is older than
// his/her parent, then output error message.
rule Person::age_check {
  exists Person *c @children { c->age > this->age }
=>
  cerr << name << " has an older child\n";
}
```

Function Calls in Conditions

A test in a rule condition may call a C++ function. The function should be a pure function (no side effects) whose return value depends solely on its arguments.

function call

```
// If the power dissipated by this resistor
// exceeds its limit, output a warning.
rule Resistor::overload {
  power(resistance,current) > power_limit
=>
  cout << "Overload: ...";
}
```

When a test consists of a function call only, a return value of zero is interpreted as false, otherwise true (as in C++).

A Complete R++ Program

The example below shows a complete R++ program consisting of three files: a class header file, a class source file, and a main source file. Rule-containing class files have the letter 'r' in their suffix (.rh or .rC) to indicate that the files must be translated into pure C++ by *rpp*, the R++-to-C++ translator.

Rectangle.rh

```
class Rectangle {
private:
   monitored int height, width;
   int area;
public:
   Rectangle(int h, int w) : height(h), width(w) {}
   add_to_height(int delta) { set_height(height + delta); }

   rule update_area;
   friend ostream& operator<<(ostream&, Rectangle * const);
};
```

Rectangle.rC

A ".rC" file must be translated into pure C++ by rpp.

```
#include <iostream.h>
$include "Rectangle.rh"

rule Rectangle::update_area {
   height > 0  &&
   width  > 0
=>
   area = height * width;
}

ostream& operator<< (ostream& s, Rectangle * const p) {
        return s << "Rectangle " << p->height << " * "
                                 << p->width  << " = "
                                 << p->area;
}
```

$include directs rpp to also translate the specified header file into C++

main.C

```
#include <iostream.h>
#include "Rectangle.r.h"

main() {
        Rectangle *rp = new Rectangle(2,3);
        cout << rp << endl;
        rp->add_to_height(3);
        cout << rp << endl;
        rp->add_to_height(-5);
        cout << rp << endl;
}
```

This includes the rpp-translated header file, which has a ".r.h" suffix.

The first step in compiling any program is to translate rule-containing classes into pure C++ using the R++ translator:

```
rpp Rectangle.rC
```

which generates pure C++ header and source files named "Rectangle.r.h" and "Rectangle.r.C". The C++ files are then compiled and linked in the normal way:

```
CC -c Rectangle.r.C
CC -c main.C
CC main.o Rectangle.r.o -l++
```

Execution of the `a.out` file outputs three lines demonstrating how the rule behaved:

Program

```
...
Rectangle *rp = new Rectangle(2,3);
cout << rp << endl;
rp->add_to_height(3);
cout << rp << endl;
rp->add_to_height(-5);
cout << rp << endl;
```

Output

```
Rectangle 2 * 3 = 6
Rectangle 5 * 3 = 15
Rectangle 0 * 3 = 15
```

Notice that the rule did not execute in reaction to `rp->set_height(-5)` since the rule's condition was not satisfied (because `height` was zero).

When to Use Rules

- Use rules to enforce *"things that should always be true"* (invariants) and detect *"things that should never be true"* (constraint violations). In particular, use rules to maintain the integrity of your domain model.
- Use rules as "demons" to monitor for and react to important states. Rules are good "watch dogs" because they are triggered by every relevant change.
- Use rules to express domain knowledge such as *business policies, engineering rules,* and *situation–action heuristics*. Such knowledge is often more clearly expressed in rules than in procedures.
- Use rules in place of member functions that would otherwise have to be called from many different places. Using rules as *"automatic member functions"* eliminates some of the burden of procedural control.
- Exploit the "data-driven" or "event-driven" nature of rules to propagate change through or update the status of an object model.

Example Applications

3

Rules can be used to enforce invariants, check constraints, monitor for important states, and express business policies and engineering rules.

Objective

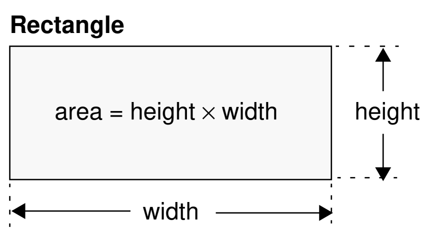
This chapter presents R++ through examples, partly to demonstrate different roles that rules can play and partly to describe rule programming more fully. Each example falls into one of four broad categories for rule use:

- rules as invariant enforcers
- rules as constraint checkers
- rules to express domain heuristics
- rules as automatic member functions

The examples illustrate that rules provide a new and simpler way to accomplish some tasks that have heretofore been programmed in a purely procedural or purely object-oriented manner.

Example 1: Enforcing an Invariant

We begin with the example from the preceding chapter, where a rule enforces the invariant that a rectangle's area is equal to its length times its width.



Rectangle.rh

rule-affecting data members should always be private.

```
class Rectangle {
private:
   monitored int height, width;
   int area;
public:
   Rectangle(int h, int w) : height(h), width(w) {}
   add_to_height(int h) { set_height(height + h); }

   rule update_area;
   friend ostream& operator<<
                  (ostream&, Rectangle * const);
};
```

This public member function calls the private R++-defined access function set_height.

Rules must always be declared.

Rectangle.rC

If height and width are positive ...

```
#include <iostream.h>
$include "Rectangle.rh"

rule Rectangle::update_area {
   height > 0  &&
   width  > 0
=>
   area = height * width;
}

ostream& operator<<(ostream& s, Rectangle* const p){
        return s << "Rectangle "
                 << p->height << " * "
                 << p->width  << " = "
                 << p->area;
}
```

... then the invariant will be enforced

main.C

```
#include <iostream.h>
#include "Rectangle.r.h"

main() {
        Rectangle *rp = new Rectangle(2,3);
        cout << rp << endl;
        rp->add_to_height(3);
        cout << rp << endl;
}
```

Benefits

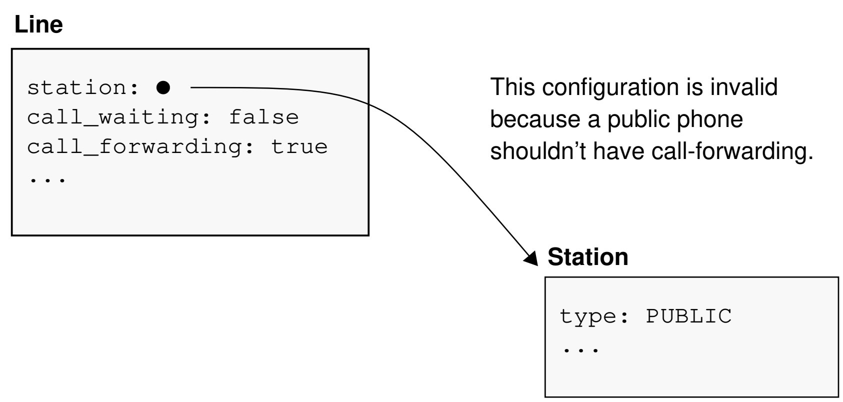
This example is very simple and could have been implemented without a rule by updating `area` from within user-defined member functions `set_height()` and `set_width()`. However, the rule approach holds two advantages:

— The invariant is specified in one place (the rule) rather than multiple places (the modifier functions). This eliminates two sources of programming error: error by omission, where the update is accidentally omitted from one modifier, and duplication error, where the update expression is not identical in all places.

— Programmers other than the original author can more quickly understand a program's purpose when the invariants are expressed in a declarative form, apart from the procedural code. In large software projects where a given module may be maintained and modified by several people during its life-cycle, improved readability and understandability of code pays large dividends.

Example 2: Consistency Checking

Object-oriented design defines the relationships that exist among objects, and path-based rules provide a simple mechanism for checking consistency among related objects. For example, when provisioning a telephone line that is connected to a public pay telephone, it would be a mistake to enable call-waiting or call-forwarding.



The rule below performs this sanity check, regardless of the order in which the line and the station are configured. (This example, and all following examples, show only the source lines that are essential to the discussion.)

Line.rh

As a member of class Line, the rule has access to its private data, but not to private data of other classes.

```
class Station;

class Line {
private:
  monitored Station *station;
  monitored Boolean call_waiting;
  monitored Boolean call_forwarding;
  ...
  rule public_phone_test;
};
```

Rules respect the access mechanisms of C++. As with member functions, a rule that accesses a private data member of another class must be declared a friend of that class (or, alternatively, the rule's class must be declared a friend of the other class, per Rule Declarations on page -3).

Station.rh

Rule is declared a friend of Station since it accesses a private data member of Station.

```
class Station {
private:
  monitored StationType type;
  ...
  friend rule Line::public_phone_test;
};
```

Line.rC

Must $include classes accessed in rule's condition.

Rule condition describes a path from a Line object to a Station object.

```
#include <iostream.h>
$include "Line.rh"
$include "Station.rh"

rule Line::public_phone_test {
  (call_waiting || call_forwarding) &&
  Station *s = station &&
  s->type == PUBLIC
=>
  cout << "Error: ...";
  set_call_waiting(false);
  set_call_forwarding(false);
}
```

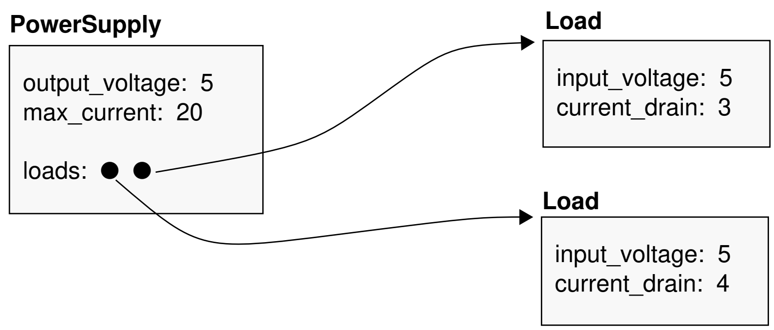
Benefits

This same consistency check can be done procedurally, but it requires that code be inserted in several places: the Line access functions for station, call_waiting, and call_forwarding, plus the Station access function for type. Further, the Station object would need a new data member pointing back to the associated Line object in order for a Station access function to check for consistency. In contrast, the rule expresses the consistency check in a single place and does not require the programmer to add back-pointers.

If the Line–Station provisioning process always has a distinct ending step, then the consistency check could have been implemented in a single place, as part of that ending step. However, a rule approach holds two advantages: the rule can alert the user to the error as soon as it happens (rather than waiting for some "final check"), and the rule will continue to work even if the provisioning process changes in a way that has several different possible ending steps.

Example 3: Constraint Checking

Many applications have natural constraints on the state of their objects. For example, suppose there is a computer-aided design (CAD) tool that allows circuit designers to connect electrical devices together visually on a screen. One natural constraint is that the input voltage of a "load" device must match the output voltage of the power supply to which it is connected; another constraint is that the total current drawn by all the load devices should not exceed the current capacity of the power supply.



This example shows how these two constraints can be expressed in rules.

PowerSupply.rh

Use Set_of_p for sets and List_of_p for lists.

This class has 2 rules.

```
#include <Set.h>

class Load;

class PowerSupply {
private:
  ...
  const float output_voltage;
  const float max_current;
  monitored Set_of_p<Load> loads;
  ...

  rule voltage_check;
  rule current_check;
};
```

This gives rules of class PowerSupply access to private members of Load. It's an alternative to making each rule a friend.

Load.rh

```
#include <Set.h>

class Load {
private:
  ...
  const float input_voltage;
  const float current_load;
  ...
  friend class PowerSupply;
  friend float sum_current(Set_of_p<Load>);
};
```

PowerSupply.rC

For each load in loads ...

... if load's voltage doesn't match power supply ...

... then output error message.

If total current load exceeds power supply's capacity ...

... then output error message.

This ordinary function is called in the preceding rule.

```
#include <Set.h>
$include "Load.rh"
$include "PowerSupply.rh"

rule PowerSupply::voltage_check {
  Load *lp @ loads &&
  lp->input_voltage != output_voltage
=>
  cout << "Error: voltage mismatch ...";
}

rule PowerSupply::current_check {
  sum_current(loads) > max_current
=>
  cout << "Error: overload ...";
}

float sum_current(Set_of_p<Load> loads) {
  Set_of_piter<Load> iter(loads);
  Load *lp;
  float sum = 0;
  while (lp = iter.next())
    sum += lp->current_load;
  return sum;
};
```

The rules above illustrate two different kinds of inter-object constraints. The first rule (voltage_check) uses a branch binding ("Load *lp @ loads") to individually check each Load's voltage against the power supply voltage. Each time a new

Load is added to a power supply, the rule checks the voltage of the new load ("the new branch") without rechecking previously-connected loads. In general, a rule can have multiple branch bindings applied to `Set_of_p` and `List_of_p` data members (see Branch Bindings on page 7).

The second rule (current_check) calls a function to compute total current load, which it then compares to the power supply's capacity. Each time a Load is added to (or removed from) a power supply, the rule is triggered and calls the function to recompute total current by iterating over all the Loads. Such iteration is more clearly expressed in a simple function like sum_current than in a group of rules. In general, a function may be called anywhere in a rule's condition, but should be free of side-effects and its return value should depend only on its arguments (see Function Calls on page 6).

Benefits

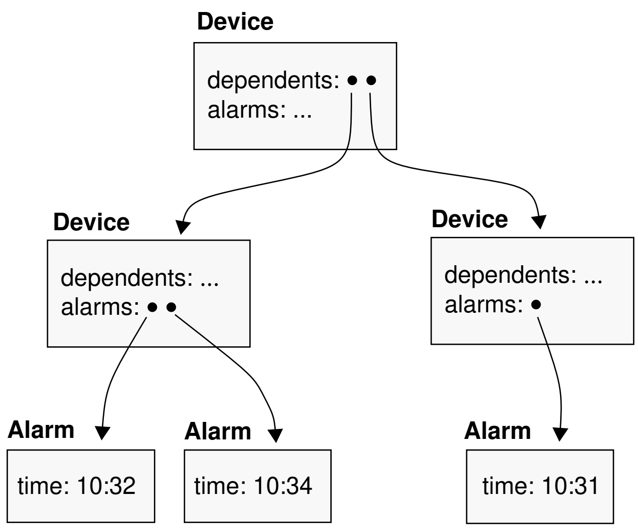
Rules offer a better way to check constraints than procedural code for two reasons. First, each constraint is clearly expressed in a single rule rather than being scattered in several pieces of procedural code, each of which may be different because they test the constraint from different perspectives. Second, software maintenance becomes easier since modifications to a constraint require changing only a single rule rather than several pieces of procedural code.

Another advantage of using a rule is precisely because it is *non*-procedural. Regardless of how many objects are involved in a constraint, regardless of the order in which objects are created and modified, regardless of how many places in the application where the constraint should be checked, a rule *guarantees* that the check will be reevaluated whenever a relevant change occurs[1]. Such guarantees are an important contributor to program robustness.

1. A rule is guaranteed to be triggered at appropriate times provided that all changes to rule-affecting data members are made through the stereotypic modifier functions.

Example 4: Event-Driven Alerting

Rules are well-suited for real-time monitoring tasks because they are event-driven, reacting to the arrival of messages and the occurrence of time-outs. Consider this equipment monitoring example where equipment is composed of many devices, some devices are dependent on other devices, and alarms from specific devices are reported as they occur.



The rule shown below expresses a common-sense rule of troubleshooting: a device is a suspect if <u>all</u> its dependents had a <u>recent</u> alarm.

Device.rh

```
class Alarm;

class Device {
private:
  monitored Set_of_p<Device> dependents;
  monitored List_of_p<Alarm> alarms;
  ...
  rule dependent_alarms;
};
```

Alarm.rh

```
class Alarm {
private:
  const Time time;
  ...
  friend rule Device::dependent_alarms;
};
```

Device.rC

If a device has dependents
... and all dependents
... had at least 1 alarm
... within the last 5 minutes
... then output message.

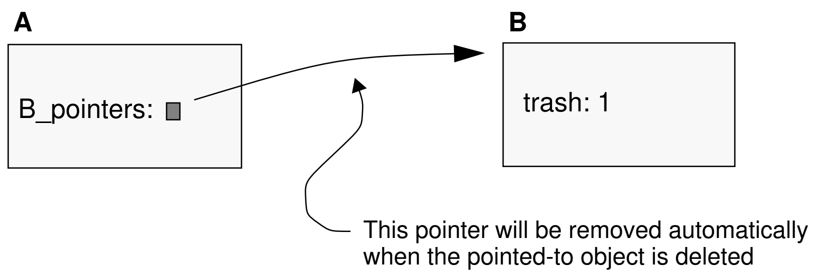
```
extern Time clocktime;

rule Device::dependent_alarms {
  dependents.size() > 0 &&
  all Device *d @ dependents {
    exists Alarm *a @ d->alarms {
      a->time > (clocktime - Duration::minutes(5))}}
=>
  cout << this << " is a suspect ...";
};
```

Benefits

The rule above illustrates how universal quantification (`all`) and existential quantification (`exists`) can express a non-trivial condition in a compact form (see Quantified Tests (all and exists) on page 8). Each time a new Alarm arrives, the rule checks the condition.

Example 5: Pointer Management

One of the most pernicious bugs in object-oriented programming is the “dangling pointer bug”, where a pointer to a deleted object is used. To avoid such bugs, a programmer must ensure that all pointers to object 'x' are removed or invalidated before 'x' is deleted, and the programmer must remember to do this every place in the code where an object is deleted. This kind of discipline can be hard to maintain during a system's life-cycle, with many different programmers making changes. The example below shows how a simple rule can eliminate a common form of dangling pointers.



B.rh and **B.rC**

When 'delete' is called, the trash flag is set, triggering the rule, before memory is freed.

```
class B {
private:
  ...
  monitored Boolean trash;
};

void B::operator delete(void *p, size_t) {
  ((B*)p)->set_trash(1);
  ::delete p;
}
```

A.rh and **A.rC**

If an 'A' object has a pointer to a 'B' object ...

... and that object was trashed

... then remove the pointer to it.

```
class A {
private:
  monitored Set_of_p<B> B_pointers;
  ...
  rule remove_Bptr;
};

rule A::remove_Bptr {
  B *bp @ B_pointers &&
  bp->trash
=>
  remove_B_pointers(bp);
}
```

main.C

This 'delete' triggers the rule, removing ap's pointer to bp before bp's memory is freed.

```
...
A *ap = new A;
B *bp = new B;
...
ap->insert_B_pointers(bp);
...
delete bp;
...
```

The rule above protects against dangling pointers from objects of class `A` to objects of class `B`. Specifically, since class `A` contains a set of pointers to objects of class `B`, it also has a rule to monitor those pointers and remove any that point to an object marked for deletion. When an object of class `B` is deleted, the '`delete`' operator for class `B` first sets the '`trash`' flag, immediately causing all rules sensitive to that flag to be reevaluated (in this case, only rule `A::remove_Bptr`). This technique ensures that monitored pointers are never left dangling.

Benefits

This example again illustrates the benefit of a non-procedural guarantee in software design. It doesn't matter how many different places in the code delete an object of class `B`; one rule serves all of them and doesn't clutter the procedural sequence of any of them.

Rule Syntax and Semantics

4

A rule, like a member function, is associated with a class and is inherited by derived classes. Rules are triggered automatically by changes in the data members that they monitor.

Overview

A rule is a new kind of class member that adds data-driven behavior to a class. A rule consists of a condition and an action. A rule monitors the data members referenced in its condition, and when their values change in a way that satisfies the condition, it executes the action.

Syntactically, a rule is similar to C++. The action part consists of ordinary C++ statements and the condition part consists of ordinary C++ boolean expressions plus special forms for binding rule variables to related objects and for expressing universally- and existentially-quantified tests.

Semantically, a rule is much like an "automatic" member function; it can access the same data and functions that a member function can, it has an implicit '`this`' variable, and it is inherited by derived classes. Rules cannot be called; they are triggered by construction of relevant objects and by changes made to such objects via stereotyped modifier functions. Rules fire in specific-to-general depth-first forward-chaining order.

This chapter explains, one-by-one, the R++ extensions to the C++ syntax, and the corresponding semantics. To see all of the syntax extensions in one place, see Appendix A.

Monitored Data Members

A rule monitors the data members appearing in its condition. Such data members should be declared with the "`monitored`" qualifier. This directs the compiler to generate the necessary modifier functions (see Modifier Functions on page 14) and warn about statements that would modify such data members without using a modifier function.

Data members that will be used in rule conditions, now or in the future, should be declared "monitored".

```
class Rectangle {
private:
   monitored int height, width;
   int area;
   ...
};
```

The 'monitored' qualifier can also be used on data members that are not presently used in rules but may be used later. Thus, code that modifies such a non-rule-affecting data member will use modifier functions from the beginning and will therefore not have to be changed when the data member is later used in a rule.

NOTE:

The "`monitored`" qualifier declares a data member to be "`const`". This enables the C++ compiler to detect statements that would modify the data member directly without using the supplied modifier function. The modifier function is able to modify the data member by casting away its `const`ness.

Rule Declarations

member-declaration:
    `...`
    rule-declaration `;`
    friend-rule-declaration `;`

rule-declaration:
    `rule` *rule-name*

friend-rule-declaration:
    `friend rule` *class-name* `::` *rule-name*

Every rule is a member of some class and therefore must be declared within the class definition. Access specifiers (`public/private/protected`) do not affect rule declarations because rules cannot be called.

```
class Rectangle {
private:
   monitored int height, width, area;

public:
   Rectangle(int h, int w) : height(h), width(w) {};

   rule update_area;
};
```

This class contains one member rule.

rule keyword

rule name

Rules respect the access mechanisms of C++. Therefore, a rule of one class that needs access to private data members of another class must be declared a *friend rule* of that class (or the first class must be a friend of the second).

The public_phone_test rule of class Line is declared a friend of class Phone to allow the rule access to private data members of class Phone.

```
class Station {
private:
  monitored StationType type;
  ...
  friend rule Line::public_phone_test;
};
```

Rule Definitions

rule-definition:
`rule` *class-name* `::` *rule-name* `{` *condition*$_{opt}$ `=>` *action* `}`

A rule is a condition–action pair, defined in a manner similar to member functions.

rule keyword   class name   rule name

condition

arrow

action

```
rule Rectangle::update_area {
   height > 0  &&
   width  > 0
=>
   area = height * width;
}
```

Although rules are like “automatic member functions”, rule definitions differ from member function definitions in two ways:

- a rule has no argument list and no return type because it cannot be explicitly called;
- a rule cannot be defined within a class body like an inline member.

Rule Conditions

condition:
  conjunct
  conjunct `&&` *condition*

conjunct:
  expression
  simple-binding
  branch-binding
  quantified-test

A rule condition is like the "if" part of an if–then statement: if the condition is satisfied then the action is executed. A rule condition identifies the classes and data members to which the rule is sensitive. A rule condition is a conjunction of tests and bindings, conjoined with "`&&`" denoting a logical–AND. Each conjunct must evaluate to non-zero in order for the rule action to be executed. As in C++, a non-zero value is considered "true" and a zero value is "false".

Null Condition

A rule with no condition is sensitive only to the construction of instances of its class. Such a rule will fire every time an object of its class is created, but will not fire in reaction to any subsequent changes to those objects because its condition does not monitor any data members.

This rule has no condition, so it fires only when a Person object is created.

```
rule Person::created {
=>
   cout << "Created: " << name << endl;
}
```

Expressions

A conjunct can be any valid C++ expression. If the expression evaluates to a non-zero value then the conjunct is considered “true”. A rule is reevaluated whenever there is a change in value of any data member appearing in its condition.

Any valid C++ expression is a valid conjunct, but should be functional and free of side-effects.

```
rule Triangle::red_right_triangle {
  color == RED  &&
  (a * a + b * b) == c * c
=>
  ...
}
```

WARNING:

Since a rule’s condition may be evaluated many times before it evaluates to true and the rule fires, all expressions in a rule’s condition, including function calls, should be functional and free of side-effects, i.e., the expression should not change any rule-affecting data members or variables, and should not perform any input/output.

Function Calls

A conjunct may contain function calls. Functions should be used to express tests or perform calculations that are difficult to express in rules, or that need to be used in more than one rule.

A function’s return value should depend solely on its arguments and constants.

```
// Theorem of Pythagoras
int rt_triangle(float a, float b, float c) {
  return a * a + b * b == c * c;
}

rule Triangle::red_right_triangle {
  color == RED  &&
  rt_triangle(a,b,c)
=>
  ...
}
```

CAUTION:

Since a rule is sensitive only to changes in the data members appearing in its condition, the return value from a function call should depend solely on its arguments and constants. If a function references data members that were not passed as arguments, then the rule will not be triggered by changes to such "hidden" data members

Simple Bindings

simple-binding:
    class-name * *ptr* = *expression*

In general, a "binding" enables a rule to access related objects in the rule's condition. A "simple binding" declares and binds a rule pointer variable to the address of a related object. The rule variable can then be used in subsequent conjuncts (as well as the action) to access data members of the related object.

```
// Print name of mother-in-law.
rule Person::mother_in_law {
  sex == male  &&                      // If this person is male ...
  Person *wife = spouse  &&            // ... and he has a wife ...   (simple binding)
  wife->sex == female  &&              // ... and she is female ...
  Person *mom = wife->mother           // ... and she has a mother ... (simple binding)
=>
  cout << "Mother-in-law"              // ... then print message.
       << mom->name << ...;            // Bound variables can be used in rule action.
}
```

A rule containing a simple binding will be reevaluated whenever there is a change in the value of any data member appearing in the initializing expression. As with any other conjunct, a simple binding is considered "false" if its initializing expression evaluates to zero, in which case rule evaluation is abandoned.

Branch Bindings

branch-binding:
    class-name * *ptr* @ *multi-data-member*

A “branch binding” is like a simple binding that branches on each address in a data member containing a set or list[1] of object addresses. This enables a rule to be applied to each element of a related set or list of objects. When a new element is added to the set or list, the rule is evaluated for the new element. The rule is *not* evaluated when an element is removed because their is no new value for the rule variable to bind to. Of course, the example rule below will also be evaluated when a Person’s age is changed.

Declares a set of pointers to Person objects.

For each child in children ...

...and child older than parent...

...then print message.

```
class Person
private:
  String name;
  monitored int age;
  monitored Set_of_p<Person> children;
  ...
  rule older_child;
};

// Alert if child older than parent.
rule Person::older_child {
  Person *child @ children  &&
  child->age > this->age
=>
  cout << "Error:" << child->name
       << "is older than parent"
       << this->name << endl;
}
```

branch binding

Bound variable can be used in rule action.

Branch bindings and simple bindings can be “chained” to describe arbitrarily long paths in a rule condition.

Quantified Tests (`all` and `exists`)

quantified-test:
  `all` *iterator-binding* `{` *condition* `}`
  `exists` *iterator-binding* `{` *condition* `}`

iterator-binding:
  class-name `*` *ptr* `@` *multi-data-member*

1. Sets and lists in R++ release 1.0 are supported with classes Set_of_p and List_of_p from C++ Standard Components. Sets and lists from other class libraries will be supported in a future release.

Quantified tests iterate over a set or list to determine if some condition is satisfied. A universally-quantified test (**`all`**) evaluates to true if *all* elements of a set or list satisfy a condition. A rule containing a universally-quantified test is reevaluated whenever an element is inserted or removed in the set/list.

Prevents trivial success when no children exist.

Universally-quantified test

```
class Person
private:
  String name;
  monitored int age;
  monitored Set_of_p<Person> children;
  ...
  rule all_adult_children;
};

// Print persons who have all adult children.
rule Person::all_adult_children {
  children.size() > 0  &&
  all Person *c @ children { c->age >= 21 }
=>
  cout << name << "has all adult children.\n";
}
```

⚠ CAUTION:

An '`all`' test is logically equivalent to verifying that no elements violate the condition, i.e., $\forall x \{p(x)\} \equiv \neg\exists x \{\neg p(x)\}$*. Thus, the test trivially succeeds if the set or list is empty. In applications where this is not desired, an explicit test for non-empty must be included.*

An existentially-quantified test (**`exists`**) evaluates to true if at least one element of a set or list satisfies a condition. A rule containing an existentially-quantified test is reevaluated whenever an element is inserted or removed in the set/list.

Existentially-quantified test

```
// Print persons having a minor child.
rule Person::minor_child {
  exists Person *c @ children { c->age < 18 }
=>
  cout << name << "has a minor child.\n";
}
```

Quantified tests can be nested. (See "Example 4: Event-Driven Alerting" on page 10.) An iterator variable is defined only within the scope of its `all`/`exists` condition and therefore cannot be used in a rule action.

Global Variables

A global variable can be used in a rule condition, but the rule will *not* be triggered by changes to the value of the global variable. A rule can only monitor data members of objects reachable via the ‘`this`’ pointer variable and via global pointer variables.

global variable

This rule is not triggered by changes to fever_threshold. If fever_threshold is modified, its new value will be used when the rule is subsequently triggered by relevant change or relevant construction.

```
float fever_threshold = 99;

class Person {
  monitored float temperature;
  ...
  rule fever_alert;
};

//  Alert when person has fever.
rule Person::fever_alert {
  temperature > fever_threshold
=>
  cout << ...
}
```

Static Data Members (like globals)

A static data member is shared by all instances of a class, like a “global” variable for the class, and is therefore treated the same as a global variable when it appears in a rule condition. Specifically, changes to static data members do *not* trigger any rules, even when such changes are made through R++ modifier functions. The reason for this behavior is simple: a rule is like a non-static member function, so the ‘`this`’ variable must be defined when the rule is evaluated, but static data members can be changed without reference to any particular object.

A static data member appearing in a rule condition should serve as a shared constant or as a shared variable whose changes need not trigger any rules.

static data member

Changes to static data members do not trigger rules because there is no well-defined 'this' value for the rule to use.

```
class Person {
private:
  static float fever_threshold;
  monitored float temperature;
  ...
  rule fever_alert;
};

float Person::fever_threshold = 99;

//  Alert when person has fever.
rule Person::fever_alert {
  temperature > fever_threshold
=>
  cout << ...
}
```

Rule Actions

action:
    statement-list

The action part of a rule consists of ordinary C++ statements. These statements are like the body of a member function and can therefore access any data and functions that would be accessible to a real member function, including the implicit 'this' variable. In addition, a rule action can access:

— rule variables bound in simple bindings and branch bindings;

— data members and member functions of other classes that the rule is a friend of.

Inheritance of Rules

Rules in a base class are inherited by derived classes. Thus, a rule defined in a base class applies to all instances of the class, including instances of derived classes.

A derived class inherits the rules of its base class(es), so rule r1 applies to Primates as well as Mammals.

```
// Base class: Mammal
class Mammal {
  ...
  rule r1;
};

// Derived class: Primate
class Primate : public Mammal {
  ...
  rule r2;
};
```

Overriding of Rules

A rule declared in a derived class will override a rule of the same name in a base class, for instances of the derived class. Instances of the base class that are not instances of the derived class will be monitored by the base class rule.

Because they have identical rule names, rule Primate::r3 overrides rule Mammal::r3 for instances of Primate.

```
// Base class: Mammal
class Mammal {
  ...
  rule r3;
};

// Derived class: Primate
class Primate : public Mammal {
  ...
  rule r3; // overrides Mammal::r3
};
```

Rule Triggering

A rule monitors changes in objects of the class(es) appearing in its condition. When a triggering event occurs for a rule, the rule reevaluates its condition for the new/changed data and, if the condition is satisfied, the rule "fires" (its action is executed). There are two kinds of triggering events: *relevant construction* and *relevant change*.

Relevant Construction

Whenever a new object of some class is constructed, all member rules of that class are triggered, including rules inherited from base classes. Thus, if a new object is initialized with values that satisfy a rule, the rule will immediately fire.

Base class

Derived class

Construction of a new Primate object triggers rule r2 and, because it is inherited, rule r1.

```
class Mammal {
   ...
   rule r1;
};

class Primate : public Mammal {
   ...
   rule r2;
};

...

Primate *p = new Primate(...);
```

Relevant Change

A rule monitors changes in the data members appearing in its condition. When the value of a monitored data member is changed, all rules that monitor that data member are triggered. This is called "relevant change" because the affected data member must be *relevant* to the rule and the new value of the data member must be *changed* from its previous value. For sets and lists, "change" means that a new element is added or an existing element is removed. Rules are not triggered by non-change (e.g., assignment of the same value) simply because of a philosophy that rules should react to changed values.

NOTE:

To ensure rule triggering, relevant changes must be made through special access functions (see Modifier Functions on page 14).

This rule monitors changes in height and width of Rectangle objects.

```
rule Rectangle::update_area {
   height > 0  &&
   width  > 0
=>
   area = height * width;
}

...

main() {
  Rectangle *rp;
  ...
  rp->set_height(5);
  rp->set_height(5);
  ...
}
```

This relevant change triggers rule Rectangle::update_area.

This does not trigger the rule since there is no change.

This special access function is called a “modifier function”.

Modifier Functions

For every data member monitored by rules, R++ defines special access functions[1] (called *modifier functions*) through which all relevant changes must be made. A modifier function triggers all rules that monitor the changed data member, provided that the new value for the data member differs from its previous value.

A modifier function is a non-const member function having a stereotyped name of the form *operation_data-member-name* where *operation* depends on the type of the data member:

1. An “access function” is a member function that supports user access to otherwise private data. [Lippman, 1992. *C++ Primer, 2nd edition*, page 228]. We use the term more broadly to include access to public data members.

- — for a single-valued data member(e.g., int, float) named `x`, the sole modifier function is named **`set_x`**;
- — for a set of pointers (e.g., `Set_of_p<T>`) named `x`, the modifier functions are named **`insert_x`** and **`remove_x`**;
- — for a list of pointers (e.g., `List_of_p<T>`) named `x`, the modifier functions are named:
    - — **`get_x`** (remove and return head of list),
    - — **`unget_x`** (insert element onto head of list),
    - — **`put_x`** (append to tail of list), and
    - — **`unput_x`** (remove and return tail of list).

A modifier function has the same public/protected/private access specification as its associated data member. As the example below shows, R++ automatically defines the modifier functions for monitored data members.

```
class Person {
private:
  monitored int age;
  monitored
  Set_of_p<Person> children;
  ...
  rule older_child;
};

rule Person::older_child {
  Person *c @ children &&
  c->age > this->age
=>
  ...
}
```

R++-defined modifier functions

```
void set_age(int)

void insert_children(Person *)
void remove_children(Person *)
```

Users may define their own modifier functions, provided that they conform to the same naming convention and function signature described above. The R++ compiler will insert the appropriate rule-triggering code into such user-defined modifier functions.

Trigger ➠ Evaluate ➠ Fire ➠ Return

The procedural flow of a program is temporarily interrupted by rule processing every time a triggering event occurs (i.e., a relevant construction or relevant change). The sequence of events is Trigger ➠ Evaluate ➠ Fire ➠ Return. Specifically, a triggering event initiates rule processing. For each rule triggered by the event, its condition is evaluated on the new/changed data and, if the condition is satisfied, the rule is fired. After the last rule has been evaluated (and possibly fired), control is returned to the statement following the triggering event.

Forward Chaining

When a rule is fired, its action may perform a triggering event, causing another rule to be evaluated and fired, which may cause yet another rule to be evaluated and fired, etc. This is called *forward chaining*[1], where a single triggering event initiates a chain of rule firings.

NOTE:

In interpreted rule languages like OPS5 and C5, applications may perform "endless" forward chaining, as in continuous monitoring applications. Such applications are designed primarily as rule-based systems that call procedural code as needed. However, applications that use R++ should be designed primarily as object-oriented applications that trigger rules as needed. Endless forward chaining is an inappropriate control strategy in R++ because each forward-chaining step uses stack memory (which is freed when the triggered rule activity finishes).

Rule Evaluation Order

Rules in R++ are evaluated in a specific-to-general depth-first forward-chaining order:

1. "Forward chaining" is so named because rule activity moves forward from some current state to a goal state. In contrast, "backward chaining" rules move backward from a desired goal to a state that confirms the goal [Cooper & Wogrin, 1988. *Rule-based Programming in OPS5*]. R++ does not support backward chaining.

- When an event triggers rules in a base class as well as a derived class, the order of evaluation is *specific-to-general*, *i.e.*, the derived-class rules are evaluated before the base-class rules. If a derived-class rule overrides a base-class rule, then *only* the derived-class rule is evaluated.
- Rule triggering proceeds in depth-first order. Just as ordinary procedural code is temporarily interrupted when a triggering event initiates rule processing, so also is a rule's action temporarily interrupted when it performs a triggering event, causing other rule(s) to be evaluated and possibly fired. Thus, if a rule action contains two triggering events, the second one will be executed only after all rule activity triggered by the first event completes. Similarly, if a single event triggers two rules, the second rule will be evaluated only after all rule activity for the first rule has completed.
- Rules are "data-driven" or *"forward-chaining"* in the sense that a change to monitored data causes a rule to be triggered, and if it fires and changes some monitored data, then another rule(s) will be triggered, and so on.
- When a triggering event triggers two or more rules that are unrelated through inheritance, the order of evaluation is unspecified. Given the view of rules as demons, rules should really be designed as independent executable units. If it is necessary to enforce some firing order among a group of rules, the sequencing should be made explicit in the rule's condition by adding a control data member to the class whose value is compared in rule conditions and changed in rule actions. This will ensure that the group of rules continues to work correctly as the source code changes and as the R++ compiler changes.

Comparison to OPS5

In OPS-based languages such as OPS5 and C5, an inference engine continually loops through a 3-stage process of matching, selecting, and executing rules. When more than one rule instantiation is ready, the inference engine applies a "conflict resolution strategy" to select the next rule for execution [Brownston et al]. The conflict resolution strategy discards instantiations that have already fired and prioritizes the remaining instantiations based on recency of change and on number of tests in their rules.

In R++ there is no inference engine and no conflict resolution strategy. Instead, the tasks of triggering, evaluation, and firing are distributed through the C++ code emitted by the R++ translator. The specific-to-general depth-first firing order described above arises naturally from the way that rules are translated into C++ code.

Refraction (Time Tags)

R++ guarantees a property, called refraction[1], that a given rule will fire at most once for a given instantiation (an instantiation is a time-stamped state of data members satisfying a rule's condition). Although the mechanism for guaranteeing this property is neither visible to nor controllable by the user, it is explained here to complete the details of rule processing.

When a rule is triggered by a relevant change at time t, the rule evaluation is queued with the value t. When the evaluation is actually performed (which will be delayed if other rules were triggered at the same time), the evaluation will be abandoned if any element of the instantiation has a time tag greater than t. If it does, this would mean that the data was changed between triggering and evaluation, and because of depth-first forward chaining, this same rule would have already been evaluated and possibly fired on the changed data. Thus, the evaluation queued at time t is abandoned to prevent multiple firings on the same instantiation.

Consider the two rules below — one to sequentially activate units 0 through 9, and the other to report when the activations are completed. (This example exists only to explain refraction; such a procedural task would not ordinarily be implemented with rules.) When an instance of `X` is constructed (with `unit` = 0), both rules will be triggered. If rule `activate` is evaluated before rule `finished`, it will fire, changing `unit` to 1, which then triggers both rules again. Because of depth-first

1. The term refraction is by analogy with the refractory period of neurons. No amount of stimulation can trigger a neuron to generate a second impulse within a few milliseconds of firing [Brownston, p. 62].

forward chaining, rule `activate` will fire ten times before rule `finished` gets evaluated for the first time. In fact, there will be ten evaluations of rule `finished` queued for evaluation, but only one can fire because of "refraction".

This call triggers both rules. If rule *activate* is evaluated before rule *finished*, then *activate* will execute ten times before *finished* gets evaluated for the first of its ten triggering events.

```
class X {
  monitored int unit;
  ...
  rule activate;
  rule finished;
};

rule X::activate {
  unit < 10
=>
  activate_unit(unit);
  set_unit(unit + 1);
}

rule X::finished {
  unit >= 10
=>
  cout << "All activations finished\n";
}
```

Summary of Rule Semantics

Triggering:

— Modifier functions and constructor functions guarantee to trigger all relevant rules.

— Rules are never triggered by irrelevant changes or by changes that bypass the modifier functions.

Firing:

— A rule is fired when its condition evaluates to true, provided it has not already fired for this instantiation (refraction).

— When rule activity completes, all rules whose condition evaluates to true have been run on all the paths that allow their conditions to evaluate to true.

Firing-Order:

— From any triggering event, rules are fired in specific-to-general depth-first forward-chaining order.

— If an event simultaneously triggers two or more rules unrelated through inheritance, the order of evaluation is unspecified.

Limitations of R++

- Since a rule in R++ is a member of a class, rules can only be defined for classes, not for fundamental types such as `int` or derived types such as structures. Such data types must be wrapped in a class before rules can be defined.
- R++ rules can be defined only for top-level classes; a class defined within another class cannot have rules.
- Rules cannot be declared within template classes. This limitation is a consequence of the current implementation where the R++-to-C++ translator, which parses rules, runs before the C++ compiler, which parses templates.
- R++ does not support backward-chaining rules.
- R++ rules do not perform general pattern matching (as OPS5 and C5 rules do) because R++ strictly follows the access paths in an object model.

Tips

5

The nature of a language is not entirely revealed in its design. This chapter elaborates through tips about do's and don'ts, how to, cautions, common mistakes, efficiency, and style.

How to Think about Rules

Roles for Rules

Like any other programming construct, rules are good for some tasks and not good for others. As a data-driven mechanism, use rules to:

— enforce invariants;

— detect constraint violations;

— express business policies and engineering rules,

— monitor for important states and events;

— propagate information.

Don't expect to write an entire application in rules. A rule is simply another kind of programming construct, to be used where appropriate.

Rules as Demons

A rule is like a demon or automaton that constantly monitors its world and reacts when needed. A rule sits apart from the procedural code but is triggered automatically by relevant changes in objects it monitors. Rules relieve a programmer of the burden of explicit control since there is no need to “call” the rule in all the appropriate places.

Rules as Automatic Member Functions

As an extension to the C++ class mechanism, a rule is much like a member function in that it is a member of a class, is inherited by derived classes, respects access restrictions, and is invoked with an implicit '`this`' pointer. The key difference is that it is triggered automatically rather than called explicitly. Thus, a rule is like an <u>automatic</u> member function.

How to ...

Reacting to "Global" Data:

Although a rule condition can reference a global variable, the rule will not be triggered by changes to that global variable. To make a rule react to changes in "global" data, the data can be placed inside a class object that is pointed to by a global variable. In the calendar-alerting example below, time-of-day is the "global" data and the rule reacts to changes in its value.

Put "global" data in a monitored data member.

This global variable will point to the sole Globals object.

This rule will be triggered every time modifier function Globals::set_time is called.

```
class Globals {
  monitored Time time; // time-of-day
  ...
};

extern Globals *global;

class Calendar {
  monitored List_of_p<Appointment> apps;
  ...
  rule appointment_alert;
};

class Appointment {
  monitored Time time;
  ...
  friend rule appointment_alert;
};

rule Calendar::appointment_alert {
  Appointment *app @ apps  &&
  global->time == app->time
=>
  cout << ...
}
```

Taming Trigger-Happy Rules:

Rules sometimes fire before you're "ready" because they are triggered *immediately* by relevant changes. For example, in populating the data members of a new object using modifier functions, each such call will trigger rules, even though you're not done filling in initial values. One way to solve this problem is to add a boolean flag to the class, have it initialized to false by the class constructor, and test the flag in every rule that monitors the class. The code that populates new instances will set the flag true only after the last data member has been set. Thus, no rules will fire until the object is marked “ready”.

```
class Person {
  monitored int age;
  monitored int height;
  monitored int ready;
  ...
  Person() : ready(0) {};

  rule age_and_height;
};

rule Person::age_and_height {
  ready  &&
  height > 60  &&
  age < 10
=>
  ...
}

main() {
  Person *p = new Person();
  ...
  p->set_age(21);
  p->set_height(65);
  ...
  p->set_ready(1);
  ...
}
```

This initializes the “ready” flag to false.

This conjunct guards against premature firing.

These trigger the rule ...

... but it can’t fire until here.

"Fire-Once" Rules:

If you want a rule to fire only once for a given object, regardless of how that object changes afterward, then add a boolean flag to the object that records whether the rule has fired or not. Naturally, the flag should be initialized by the class constructor to its "false" state.

This conjunct prevents subsequent firings ...

... after the first firing.

```
rule Person::senior_citizen {
  senior == false  &&
  age >= 65
=>
  set_senior(true);
  ...
}
```

Finite State Machines:

Since rules are event-driven, they provide a good mechanism for advancing a finite-state machine in response to events. The example below shows a rule to advance a device's finite state machine upon arrival of a new message.

```
class Event {
  monitored int seq_no;
  monitored int event_id;
  ...
};

class Transition;
```

Finite state machine consists of State nodes and Transition links.

```
class State {
  String state_name;
  monitored Set_of_p<Transition> trans;
  ...
};

class Transition {
  const int event_id;
  const State *next_state;
  ...
};
```

Every device is in some state...

... which may change as new events occur.

```
class Device {
  monitored State *state;
  monitored List_of_p<Event> events;
  monitored int num_consumed;
  ...
  rule advance_state;
};
```

When an event arrives...
...that hasn't been seen...
...and there is a transition...
...that matches the event...
...then "consume" the event
...and update the state.

```
// Advance state when new event arrives.
rule Device::advance_state {
  Event *e @ events &&
  e->seq_num > num_consumed &&
  Transition *t @ state->trans &&
  e->event_id == t->event_id
=>
  set_num_consumed(e->seq_num);
  set_state(t->next_state);
}
```

This rule is carefully designed to avoid refiring repeatedly on the same event when it updates the device's state. Note that each incoming event is tagged with an ever-increasing sequence number whose value is saved with the device in `num_consumed` as a record of what events have been processed. Every time the rule processes an event, it updates two values — `num_consumed` and `state` — and the order is critical because each update immediately re-triggers the rule. The update of `num_consumed` immediately re-triggers the rule, but the evaluation will fail because of the new value in `num_consumed`. The update of `state` also re-triggers the rule, and the evaluation fails for the same reason (as desired).

NOTE:

If these two updates had been reversed, the update of `state` would re-trigger the rule and the evaluation would succeed, possibly leading to another firing that inappropriately advanced the state again (and again, and again, etc.).

Matching Unrelated Objects:

Suppose you want to write a rule that detects pairs of people who have the same first name. Since such pairs of people are not ordinarily related via inter-object pointers, there is no person-to-person path for an R++ rule to use. In cases like this where it is necessary to compare an object to all other objects of its class, the solution is to maintain a set of all the objects, typically in a static data member of the class. The class constructor and destructor must update the set.

```
class Person {
  monitored static Set_of_p<Person> allpersons;
  monitored String firstname;
  ...
};

// Constructor
Person::Person(String n, ...) :
  firstname(n), ... { allpersons.insert(this); }

// Destructor
Person::~Person() { allpersons.remove(this); }

// Detect people with same first name.
rule Person::same_first_name {
  Person *p @ allpersons &&
  this->firstname == p->firstname &&
  this != p
=>
  ...
}
```

The need for inference among unrelated objects does not arise very often in object-oriented designs because the most important relationships are already represented through inter-object pointers. Although the example above illustrates how to accomplish this in rules, realize that this same task could have been accomplished as procedural code inside the constructor.

Backward Chaining:

R++ does not provide backward-chaining rules, such as the backtracking rules of Prolog. Such rules are useful in goal-directed reasoning (as opposed to data-driven reasoning) where the control strategy reasons backward from a goal through inference rules to find if data exists to support the goal. For example, some forms of medical diagnosis reason backward from a hypothesis, building up a deductive chain from the hypothesis back to supporting evidence, gathering evidence as needed.

Goal-directed reasoning is like answering a query, and there are two ways to do that in a language like R++. The first and most obvious way is simply to write a function that queries the object space and returns the answer. For many applications this is probably the simplest way.

A second technique for performing goal-directed reasoning is to use a group of forward-chaining rules to mimic a backward-chaining rule. This technique is covered in other references [Brownston, Ch. 3], but is not generally recommended because the group of rules needed is moderately complicated.

Do's and Don'ts

- Always use modifier functions to change monitored data members (except inside constructor functions).
- Don't use modifier functions inside constructors (it will cause double-triggering of rules). Just assign or initialize data members in the normal C++ way.
- Don't write member functions that directly assign to monitored data members, even though they have access permission. Such changes will fail to trigger the rules.
- Don't use preprocessor macros in rule conditions. Such macros are not expanded before *rpp* runs, so it may cause *rpp* to report syntax errors or fail to recognize all the data members that a rule monitors. It's OK to use macros in rule actions since that code is not examined by *rpp*.
- Rule conditions should be functional and free of side-effects.
- Don't put print statements in a rule condition. A rule condition may be evaluated _many_ times before it fires, resulting in many output messages.
- When writing a function or macro to be used in a rule condition, do create arguments for every data member that it uses. Otherwise, the rule will not be triggered by every relevant change.
- Don't give two rules in the same inheritance hierarchy the same name unless you want a derived class rule to override a base class rule.

Reminders

Rule on a Legacy Class

To apply rules to a “legacy class” (a class whose definition cannot be changed because other applications use it), create a “wrapper class” around the legacy class and apply the rules to the wrapper class.

Monitoring a Set/List of Primitive Types

A branch binding in a rule condition is only valid for class objects, not primitive types. For example, if you want to branch on a list of integers, the integer must be within a class, i.e., it must be wrapped. In short, rule variables must always point to class objects.

Global Variables and Static Data Members

Remember that although global variables and static data members can be used in rule conditions, changes to those variables will <u>not</u> trigger rules.

Common Mistakes

Failure to Initialize

The rules of a class are triggered immediately when an instance of that class is created. Thus, *all* constructors of the class should initialize *every* monitored data member. Without this, rules may fire on garbage data.

"all" Succeeds When Empty

Remember that an `all` test trivially succeeds on an empty set/list. For example, the test "`all Person *p @ children { p->age < 10 }`" will succeed if '`children`' is an empty set, because no child violated the test "`p->age < 10`". To prevent the test from succeeding in this empty case, add "`children.length() > 0`" as a separate test.

Exclude Self-Matches

When selecting pairs of objects from the same list or set, remember to exclude the case of matching an object to itself. For example, the last test in the rule below is important:

Excludes
self-matches

```
rule Person::friends_with_same_name {
  Person *p1 @ friends &&
  Person *p2 @ friends &&
  p1->firstname == p2->firstname &&
  p1 != p2
=>
  cout << ... ;
}
```

Cautions

Avoid Stack Objects

Rule-affecting objects should be allocated from the free store using '`new`'. Avoid declarations that create temporary objects on the stack. Rules will run on such objects the moment they are created, and if any rules create pointers to such temporary objects, there may be dangling pointers when a temporary object goes out of scope.

Set and List Iterators

The sets and lists shown in this manual are from C++ Standard Components. A future release will allow other class libraries to be used for sets and lists of pointers. To work properly with R++, such classes must have a “stable iterator” having two properties: (1) if an element is added while an iterator is active, the iterator should *not* see the new element; (2) if an element is deleted while an iterator is active, the iterator should *not* see the deleted element.

Sets and Lists in a Condition

Be aware that when a set or list data member appears in a conjunct other than a binding, the rule will be triggered by *any* changes to the set or list, whether insertion or removal. Consider rules r1 and r2 below. They are almost identical except that r2 tests for a non-empty “children” set. The programmer of r2 might have intended the test as a guard against branching on an empty set, but that is unnecessary and will yield an unwanted behavior: the rule will be reevaluated whenever a person is *removed* from “children”. Rule r1, in contrast, will be reevaluated only upon insertion.

```
rule Person::r1 {
  Person *p @ children &&
=>
   ...
}
```

This rule is triggered only by insertion of a person into “children”.

```
rule Person::r2 {
  children.size() > 0  &&
  Person *p @ children &&
=>
   ...
}
```

This rule is triggered by insertion <u>or</u> removal of a person in “children”.

Main Loop in C++ Code

Applications designed for continuous operation must place the main loop in C++ code, <u>not</u> in R++ code. For example, an application that continuously monitors an equipment log file must put the file read loop in C++ code, and simply let rules react to the incoming data (and when no more rules can fire, control returns to the read loop). This is necessary because of the way stack memory is used during rule processing: the stack grows with each forward-chaining rule firing, then shrinks back to normal when no rules can fire. Thus, if an application implemented its main loop in rules, with no return to ordinary C++ code, the stack would grow until it ran out of memory. Placing the main loop in C++ code avoids this problem.

Destructors

Don't perform rule-affecting actions in destructors.

Signals

Signal handlers should return to the point of interrupt. (R++ must postpone signals around certain pointer setups).

Efficiency Tips

The main computational overhead that rules add to an application is in the repeated evaluations of rule conditions in response to triggers. This fact suggests three obvious efficiency tips:

— In a condition containing both tests and bindings, the tests should be placed as close to the beginning as possible.

— Tests that are more likely to fail should precede tests that are more likely to succeed, where possible.

— Tests that are more time consuming, such as function calls and "`all`" and "`exists`" tests, should be placed as close to the end as possible.

Rules in R++ are termed "path-based" because a condition can describe a path from one object to another to another, etc. The *length* of the path affects the amount of computation in some obvious ways:

— A rule having a long path introduces more overhead than a rule with a short path.

— The larger the branching factor in the object space described by the path, the greater the amount of work during evaluation.

— If a rule having a long path can reasonably be decomposed into two shorter rules, it will probably be more efficient.

Stylistic Conventions

- In a rule definition, the keyword **`rule`** should begin at the left margin followed by the rule's class name, scope operator, rule name, and left brace (`{`). The rule condition should begin on a new line, indented at least two spaces. When coding a rule condition, place each binding or test on a separate line. The forward-chaining arrow (**=>**) should be on a separate line without indentation. The rule action should begin on a new line, indented the same amount as the condition.
- Place **`rule`** declarations and **`friend rule`** declarations together at the end of the class definition. In this way it is easy for the class designer to see the names of all rules that monitor the class.

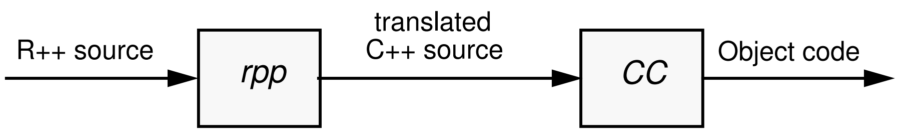
Compiling R++

6

R++ source code is C++ with added rules. R++ code is first translated to pure C++ by the R++ translator, then compiled with a C++ compiler.

Overview

Compilation of a program containing rules is a two-step process. First, classes that contain rules or participate in rules of other classes must be translated to pure C++ by *rpp*, the R++ translator. Second, all C++ source files, including those emitted by *rpp*, must be compiled by *CC*.



File Naming Convention

The convention in C++ is that for some class named “`Thing`”, the class is defined in a file named “`Thing.h`” and its member functions are defined in a file named “`Thing.C`”.

Thing.h

```
class Thing {
  •
  •
  •
};
```

Thing.C

```
#include "Thing.h"
  •
  •
  •
```

Since rule-affecting classes must be translated by *rpp*, the convention in R++ is to name such files with distinctive suffixes “`.rh`” and “`.rC`”. For example, if a rule is added to class Thing, the files must be renamed to “`Thing.rh`” and “`Thing.rC`”.

Thing.rh

```
class Thing {
   •
   •
   •
   rule r1;
};
```

Thing.rC

```
$include "Thing.rh"
   •
   •
   •
rule Thing::r1 {
   ...
}
```

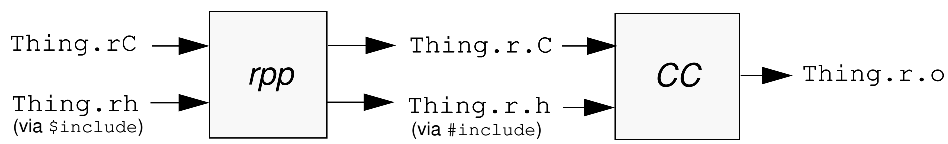
Notice that source file "Thing.rC" uses `$include` rather than `#include` to include header file "Thing.rh" in order to direct *rpp* to translate the associated header file.

Translating R++ source with *rpp*

Any class that participates in any rule must be translated by rpp. For example, class "Thing" is translated with the command line:

```
rpp Thing.rC
```

which generates two pure C++ files named "`Thing.r.h`" and "`Thing.r.C`", for subsequent compilation by *CC*. The "`.r.`" in each file name signifies that the file was *rpp*-generated (and therefore should <u>not</u> be edited).



Compiling a Family of Classes

A rule whose condition references objects of more than one class creates a "family" of classes that must be processed concurrently by *rpp*. For example, if a rule of class X references classes Y and Z, then all three classes must be translated together, as in:

```
rpp X.rC Y.rC Z.rC
```

This is necessary so that *rpp* can arrange for the rule to be triggered by modifier functions of all three classes. If any family members are missing when *rpp* translates an inter-class rule, *rpp* will report a syntax error. Disjoint families can be *rpp*'d separately.

C++ Preprocessor Directives

Since *rpp* precedes *CC* in the compilation of rule files, the C++ pre-processor (*cpp*) runs after *rpp*. That means that pre-processor directives such as `#include`, `#ifdef`, and `#define` have no effect on the source statements seen by *rpp*; they are simply passed through *rpp* for subsequent processing by *CC*. There are three consequences of running *cpp* after *rpp*:

— Rule-containing source files must use `$include` (not `#include`) to include rule-containing header files. For example, file `Thing.rC` must contain the statement `$include "Thing.rh"`.

— Ordinary C++ files that `#include` rule-containing header files must be sure to include the *rpp*'d version of the header file. For example, "`main.C`" must include "`Thing.r.h`" rather than "`Thing.rh`".

— Macros can be used in rule conditions only if they abide by the same requirements as for function calls, namely, that all rule-affecting data members appear as arguments of the macro.

Syntax Errors with `const` types

Data members that are declared "`monitored`" are translated by *rpp* into `const` declarations so that *CC* can detect and report any statements that do direct assignment to such data members. Such statements are errors because they bypass the modifier functions.

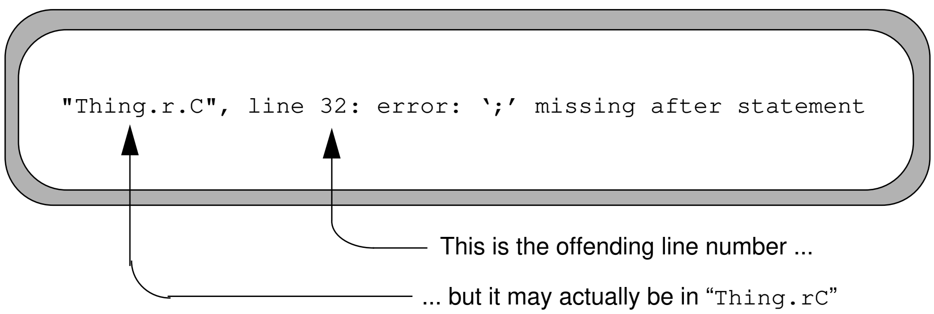
```
"Thing.r.C", line 22: error: assignment to const type
```

If you get this kind of error from CC, check for direct assignment to a monitored data member.

Syntax Errors: rpp vs. CC

When *rpp* processes rule files, it checks the syntax of rule declarations and rule definitions and reports any syntax errors present in the rule files. Since *rpp* does very little parsing of plain C++ code, including the action part of rules, code translated by *rpp* can still contain syntax errors that will be detected by *CC*. To help localize such errors to statements in one of the original "`.rh`" or "`.rC`" files (rather than in the intermediate "`.r.h`" or "`.r.C`" files), *rpp* inserts `#line` directives in the intermediate files. Thus, many of the syntax errors reported by *CC* will refer back to lines in the original source files.

Syntax errors reported by *CC* always identify a file name and a line number, as shown below. Although the line number will always be correct, the file name will be incorrect for some errors. This anomaly is due to *cfront*'s use of `#line` directives in template instantiation. Until this problem is corrected, understand that an error reported in "*file*`.r.C`" (or "*file*`.r.h`") may really exist in "*file*`.rC`" (or "*file*`.rh`") at the reported line number.



Compiler Options

rpp provides two option flags, `-r` and `-s`, as described in Chapter 7 on tracing and debugging. The '`-s`' option causes *rpp* to generate a static analysis of rule triggering on standard output. The '`-r`' option causes *rpp* to insert code for run-time tracing.

Sample Makefile

The sample makefile shown below is for an application consisting of three pairs of files for classes `X`, `Y`, and `Z`, plus a main program file. Classes `X` and `Y` form a "family" due to inter-object rules defined in class `X`, so they are *rpp*'d together. Class `Z` is not used in any rules, so it is compiled directly by *CC* without translation by *rpp*.

Note the dependencies for target "`main.o`". File "`main.C`" must `#include` header files for all three classes, but in the case of classes `X` and `Y`, it must include the *rpp*'d versions of those header files (i.e., "`X.r.h`" and "`Y.r.h`").

Makefile (for make)

```
main:   X.r.o Y.r.o Z.o main.o
        CC -o main X.r.o Y.r.o Z.o main.o -l++

main.o: X.r.h Y.r.h Z.h main.C
        CC -c main.C

X.r.o:  X.r.h X.r.C
        CC -c X.r.C

Y.r.o:  Y.r.h Y.r.C
        CC -c Y.r.C

Z.o:    Z.h Z.C
        CC -c Z.C

X.r.h X.r.C Y.r.h Y.r.C: X.rh X.rC Y.rh Y.rC
        rpp X.rC Y.rC
```

Tracing & Debugging

7

To make rule behavior visible, R++ provides a static analysis of rule triggering, run-time tracing of rule activity, and post-mortem analysis of rule activity.

Static Analysis of Rule Triggering ('-s')

It's easy to look at a rule and see the condition under which it will fire, but it's much harder to look at a modifier function and identify all the rules that it will trigger. For this reason, *rpp* provides option '-s' to display a static analysis of rule triggering. This analysis is of the following form:

```
RULE TRIGGERING ANALYSIS:

For class X:
  upon construction: X::r1, Y::r2, ...

For X::dm
  upon modification: X::r3, Y::r5, ...
  upon insertion: ...
  upon deletion: ...
```

It's helpful to consult this static analysis when modifying a program in order to remain aware of all the rule activity that a single relevant change can initiate.

Run-Time Tracing ('-r')

The two most common questions in debugging a rule-based program are "why didn't this rule fire?" and "why did this rule fire so many times?" Both of these questions can be answered by examining the run-time trace generated by an R++ program that has been translated with option '-r'. The trace displays all relevant constructions, relevant changes, and rule reevaluations. As the sample trace below shows, each record identifies the affected object or data member or rule, the activity, and the associated serial number and depth of forward chaining. (A depth value of zero accompanies events that occur at the base level, i.e., in ordinary procedural code.)

```
depth rule/dm    time-tag   activity
   0     X          25     instance created at address p
                           [calls ostream operator<< if overloaded]

   0     X::dm1     26     value changed from x to y
                           (or value unchanged)

   0     X::dm2     27     pointer x inserted (or deleted) (or unchanged)
                           [calls ostream operator<< if overloaded]
```

```
0     X::r1      27     term i evaluated true [with binding v]
                        (or evaluated false,
                         or out-of-date with serial# n)

1     X::r1      27     start with bindings x=?, y=?, ...

1     X::r1      27     end
```

In the case of rule reevaluations (as in the fourth record), it's important to understand that two serial numbers are involved: one associated with the relevant change that triggered the reevaluation, and one associated with the value or binding to be used. The serial number displayed in the third column is due to the rule-triggering event; if the term being reevaluated has been changed since the rule-triggering event, then the trace will also show the term's higher serial number (which will abort the rule reevaluation). The last two records show the start and end of a rule firing. In general, there may be many other trace records between start/end pairs if the rule causes other relevant changes.

Rule tracing can be made more selective through the use of two preprocessor directives, "`#pragma trace_on`" and "`#pragma trace_off`", and by varying the trace level on the '`-r`' option. Every rule is effectively tagged by the preprocessor as either "default", "trace_on", or "trace_off", depending on the last pragma seen (if any) prior to the rule definition. Rules are then traced or not depending on their tags and the trace level 0–3 given with '`-r`' per the table below.

Rule Tracing

	`trace_off`	`default`	`trace_on`
-r0			
-r1 (default)			✔
-r2		✔	✔
-r3	✔	✔	✔

Post-Mortem Analysis (*rpm*)

A run-time rule trace contains additional information that is useful in debugging and tuning, but hard to see. For example, are there any rules that didn't fire? What rule fired the most? What rule was reevaluated the most? What was the maximum depth of forward chaining, and what change triggered it? What modifier function accounted for the most rule processing? All of these questions can be answered by analyzing a rule trace, and this task is automated by *rpm*, the R++ post-mortem trace analyzer.

To use *rpm*, the trace output (which goes to cerr) must first be redirected to a file. After the application terminates, run the command:

`rpm` ***filename***

The output of *rpm* summarizes events in the trace file, as described below:

- The "rule statistics" show, for each rule, how many times it fired, how many times its condition was evaluated, and how many of those evaluations were logically false and how many were expired per the time tag.
- The "relevant change statistics" show, for each monitored data member, how many times its modifier function was called with a change or no-change. The "avg. work" column estimates the average amount of work triggered by the changes by counting the number of lines of trace output attributable to the change. This number estimates the amount of rule processing overhead, but does not measure actual CPU time consumed.
- The "relevant construction statistics" show, for each monitored class, how many new instances of the class were created and how much work was triggered, on average.
- The "forward-chaining matrix" shows how many times one rule's firing caused another rule to fire. Specifically, the number in row i column j is the number of times that rule i caused rule j to fire.

```
Rule statistics:
 Rule    #firings      #evals        #false    #expired
 X::r1      100          400          180         120
 Y::r2        0           97           97           0
 ...

Relevant change statistics:
 Data mem    #changes     #no-change   avg. work
 X::dm1        277           23          14.6
 Y::dm2         58            0          98.2
 ...

Relevant construction statistics:
 Class     #instantiations   avg. work
   X            46              3.2
```

```
Max depth of forward chaining = 7, triggered by Y::set_dm2
Forward-chaining matrix:
         X::r1  Y::r2  Z::r3
outside   13      0      78

X::r1      0      0       9

Y::r2      0      0       0

Z::r3      5      0       0
```

Grammar Summary

A

R++ extends the grammar of C++ by adding a new kind of class member – a rule. This appendix shows the extensions beyond the C++ grammar.

Grammar Summary

This grammar summary shows only the extensions made to the C++ grammar. Section numbers correspond to those in *The Annotated C++ Reference Manual* [Ellis & Stroustrup].

This grammar is intended as an aid to comprehension; it is not an exact statement of the R++ extension to C++. In particular, this grammar accepts a superset of valid R++ statements.

17.1 Keywords

rule-name:

identifier

17.3 Declarations

declaration:

...
rule-definition

decl-specifier:

...
`monitored`

17.4 Declarators

rule-definition:

`rule` *class-name* `::` *rule-name* `{` $condition_{opt}$ `=>` *action* `}`

condition:

conjunct
conjunct `&&` *condition*

conjunct:

    expression
    simple-binding
    branch-binding
    quantified-test

simple-binding:

    class-name `*` *ptr* `=` *expression*

branch-binding:
iterator-binding:

    class-name `*` *ptr* `@` *multi-data-member*

quantified-test:

    `all` *iterator-binding* `{` *condition* `}`
    `exists` *iterator-binding* `{` *condition* `}`

multi-data-member:

    identifier
    ptr->identifier

action:

    statement-list

17.5 Class Declarations

member-declaration:

    ...
    rule-declaration `;`
    friend-rule-declaration `;`

rule-declaration:

    `rule` *rule-name*

friend-rule-declaration:

    `friend rule` *class-name* `::` *rule-name*

Glossary

G

Rule-based reasoning in general—and path-based rules in particular—introduces several new terms to the programmer's lexicon.

Glossary

Access path: A path from a root object to its neighbors (and their neighbors, etc.) via inter-object pointers. Each instantiation of a rule's condition defines an access path. A path of length 0 involves only a root object; a path of length 1 involves a root object and an immediate neighboring object.

Action: The "consequent" or "right-hand side" of a rule. An action consists of ordinary C++ statements that may use any variable bound in the condition (plus "`this`") and any member function of the class(es) in the rule's access path.

Condition: The "antecedent" or "left-hand side" of a rule. A condition consists of zero or more bindings and zero or more tests.

Binding: A form in a rule condition that binds a rule variable to a value. See "simple binding", "branch binding", and "iterator binding".

Branch binding: A binding of the form "*class-name* *p @ mvdm" is called a branch binding because the path described by the rule's condition effectively branches on every pointer value in the multi-valued data member "mvdm". The bound variable ("p") can be used in the rule's action.

Evaluation: A rule's condition is "evaluated" in response to being triggered. If the condition evaluates to true, the rule is fired.

Fire: A rule is said to "fire" when its action is executed. A rule fires only when its condition evaluates to true.

Forward Chaining: Rule activity that moves forward from some current state to a goal state is termed "forward chaining". Forward-chaining (data-driven) rules can cause a "chain" of rule firings as the action of one rule triggers another rule, and that rule fires and triggers another rule, etc.

Full Instantiation: A complete, consistent set of bindings for every variable in a rule's condition (values that concurrently satisfy all tests in the condition) [Brownston et al]. Equivalent to a full access path. Sometimes called "activation".

Instantiation: A time-stamped state of data members satisfying a rule's condition.

Iterator Binding: A binding associated with an "`all`" or "`exists`" form is called an iterator binding because it iterates over the associated set or list to determine if the "`all`" or "`exists`" form evaluates to true.

Modifier Function: An access function that modifies a monitored data member, either by assigning a new value or by inserting/removing an element in a list or set.

Monitored: A class is said to be "monitored" if a rule (of any class) binds a variable (in its condition) to an instance of that class. A data member is said to be "monitored" if it is referenced in a rule condition.

Neighbor: Object *y* is a neighbor of object *x* if a data member of *x* contains a pointer to *y*.

Object: An instance of a class.

Path-Based: Rules in R++ are termed "path-based" because rule conditions can specify an path from the root object to its neighbors and its neighbors' neighbors, etc.

Partial Instantiation: A consistent set of bindings for the first *N* variables in a rule's condition, where *N* is less than the number of variables in the condition.

Quantified Test: The "all" and "exists" tests that can appear in a rule condition are termed "quantified tests", as in the terminology of first-order logic, to indicate that the test's truth is determined by ranging over all the elements of a set or list.

Refraction: The property of rule semantics that guarantees that a given rule will fire at most once per instantiation.

Relevant change: A rule is triggered by "relevant change"; it is "relevant" only if the affected data member is referenced in the condition, and it is "change" only if the data member's new value differs from its old value.

Relevant construction: All rules defined for class C are triggered by construction of an instance of class C.

Root class: The class that a rule is declared in is called its "root class" because only an instance of that class can form the root of an access path.

Root object: The instance of the root class (or a derived class) used in a rule instantiation. A rule's condition can access only the root object and its neighbors.

Rule: A data-driven programming construct—associated with a class—consisting of a condition and action. A rule's action is executed ("fired") when its condition is satisfied by objects in memory. A rule whose condition references only the "this" object is termed an intra-object rule, otherwise an inter-object rule. Sometimes called a "production" or "production rule".

Simple binding: The condition form "*class-name* *p = *expression*" binds the variable "p" to the object address returned by *expression*. This is called a simple binding, in contrast to a branch binding. The bound variable ("p") can be used in the rule's action.

Time tag: Every relevant change is tagged with an integer value known as the time tag. Larger time tags indicate more recent changes. Although time tags cannot be manipulated by the user, it's helpful to understand their meaning since it affects rule semantics and since it appears in the run-time trace output (see chapter 3, section "Refraction").

Trigger: An event that triggers reevaluation of a rule's condition, specifically a relevant change or relevant construction.

R

S

T