Rule-Based Delegation for Prototypes

Jay Almarode

Oregon Graduate Center

ABSTRACT

Arguments have been given recently for providing the functionality of prototypes in object-oriented languages. Prototypes allow more flexible sharing of code and data by delegating messages to parent objects without the rigid structure of a class hierarchy. Prototypes can implement classes, and delegation can be used to model both single and multiple inheritance. However, one drawback with delegation is the difficulty in enforcing the semantics that delegation is used to model. This paper proposes a novel mechanism to control the delegation of messages with rules. In this system, the delegation of messages is governed by a set of rules possessed by each object. Rules can be used to implement classical single inheritance and can implement various solutions to multiple inheritance. In addition, rules can be created dynamically to model application-specific semantics. This paper describes how rule-based delegation works and illustrates various rules for rule-based delegation that have been implemented.

Introduction

Recent literature on prototypes [Borning 86], [Lieberman 86], and [Ungar Smith 87] has illustrated the flexibility such systems provide. Some systems, such as exemplar based Smalltalk [LaLonde Thomas Pugh 86], have combined both prototypes and classes. The key to the flexibility of prototypes is delegation. Delegation can model class-based inheritance in addition to other relationships between objects. However, one drawback of delegation is the difficulty in recognizing and enforcing the semantics associated with delegation. Delegation can be used to model various relationships, but the flexibility of delegation tends to obscure what semantics are being modelled. This paper proposes a novel mechanism to model the various semantics of delegation, especially in the context of multiple inheritance. This mechanism, called rule-based delegation, provides a concise, declarative way to specify delegation. Rule-based delegation can implement classical single and multiple inheritance, in addition to modelling other relationships and allowing for the dynamic creation of rules for delegation. This paper will first review the differences between class-based objects and prototypical objects and how prototypes can implement classes. Next various solutions for multiple inheritance will be discussed. Finally, rule-based delegation will be described and various rules for rule-based delegation will be illustrated.

Classes and Prototypes

In classical object-oriented languages such as Smalltalk [Goldberg Robson 83] and Flavors [Moon 86], each object is an instance of some class. The class specifies the operations, called methods, that can be invoked on instances of the class. A class defines the instance variables of each instance and may also define class variables that can be accessed by all instances of the class. New instances of a class are created by sending a message, such as *new*, to the class. Each class is defined as being a subclass of some other class(es), thus forming a chain of classes. When a class has only one immediate superclass, the class hierarchy is a tree and this form of inheritance is called single inheritance. When a class specifies more than one superclass, the chain of classes forms a directed acyclic graph and this form of inheritance is called multiple inheritance.

In delegation-based prototypical languages, an object is not an instance of some class, but rather it is simply a collection of named slots. A slot may contain another object or a block of code. When any message is sent to an object, the receiver determines if it has a slot corresponding to the message. If the slot contains a value, then it returns that value. If the slot contains a block of code, the block is executed and the result returned. If the object does not have a slot corresponding to the message, it delegates the message to some other object. Typically, the message is forwarded to the object contained in a designated parent slot of the receiver.

Since there are no classes in prototypical languages, an object does not inherit methods from its class. Instead, messages are forwarded to parent objects until a slot is found or no parent link is found. Because of delegation, the notion of *self* is different in delegation-based languages. The original receiver of a message is designated as self. This object remains as self even after delegating the message to another object, which is designated as the *client*. If the client does not recognize the message, it delegates the message to its parent, which becomes the new client. When a block of code is found in some client, it is executed and references to self are to the original receiver of the message, not the client.

Implementing Classes with Prototypes

Although there is no notion of classes in prototypical languages, prototypes can easily implement the functionality provided by classes. To implement classes with prototypes, all the common behavior of similar objects is combined into a single object that serves the same role as a class. A class-representing prototype has slots that contain code blocks corresponding to class and instance methods and slots that contain the values of class variables. Each instance object inherits what is stored in the class-representing object's slots by setting its parent link to the class-representing prototype. An instance object has slots corresponding to instance variables. Messages are delegated to the class-representing prototype through the parent link named "class".

Figure 1 illustrates the implementation of class Person, class Object, and class Class with prototypes. In this figure, aPerson is an instance of class Person, therefore its "class" slot points to the class-representing prototype for class Person. This object also has slots for name and birthdate that correspond to instance variables. The class Person, on the other hand, is an instance of a class, so its "class" slot points to the class-representing prototype for class Class. The prototype for class Person also has a slot containing a method to calculate the age of a person (called *calcAge*) and a slot that points to the superclass of Person, class Object. When a message is sent to aPerson, the receiver first determines if it has a slot corresponding to the message. If not, the message is delegated to the object contained in the "class" slot. If class Person does not have a slot corresponding to the message, it must delegate the message to its superclass, class Object. This is the object contained in the "superclass" slot for class Person. The class Object contains methods that are inherited by all objects (the *print* method, for example).

Figure 1 also points out a difficulty when delegation is used to implement class inheritance. If a message is initially sent to class Person instead of delegated to it, then delegation should be handled differently. For example, if the *new* message is sent to class Person, it should delegate the message to class Class, which contains the default method for instantiating new objects. This is because class Person is an instance of class Class, therefore its "class" slot points to class Class. In this case, instead of delegating a message to the object in its "superclass" slot, as was done when the message was originally sent to aPerson, class Person should delegate the message to the object in its "class" slot. In class-based systems, this distinction is hard-coded in the method search algorithm. When a message is sent to an object, the search first targets the class of the object, then follows the chain of superclasses. In delegation-based prototypical systems, however, delegation occurs outside the context that the message is sent. The delegation mechanism does not know if an object is the original receiver of the message or if the object is a client for a delegated message.

Multiple Inheritance

Many different approaches have been devised to handle multiple inheritance in class-based systems. The main issue involved with multiple inheritance is what to do when name conflicts exist between attributes and between methods of the multiple superclasses. Solutions have ranged from regarding name conflicts as errors which must be disambiguated, as in Smalltalk [Borning Ingalls 82], to allowing the programmer full control over how multiple inherited methods are invoked and their results com-

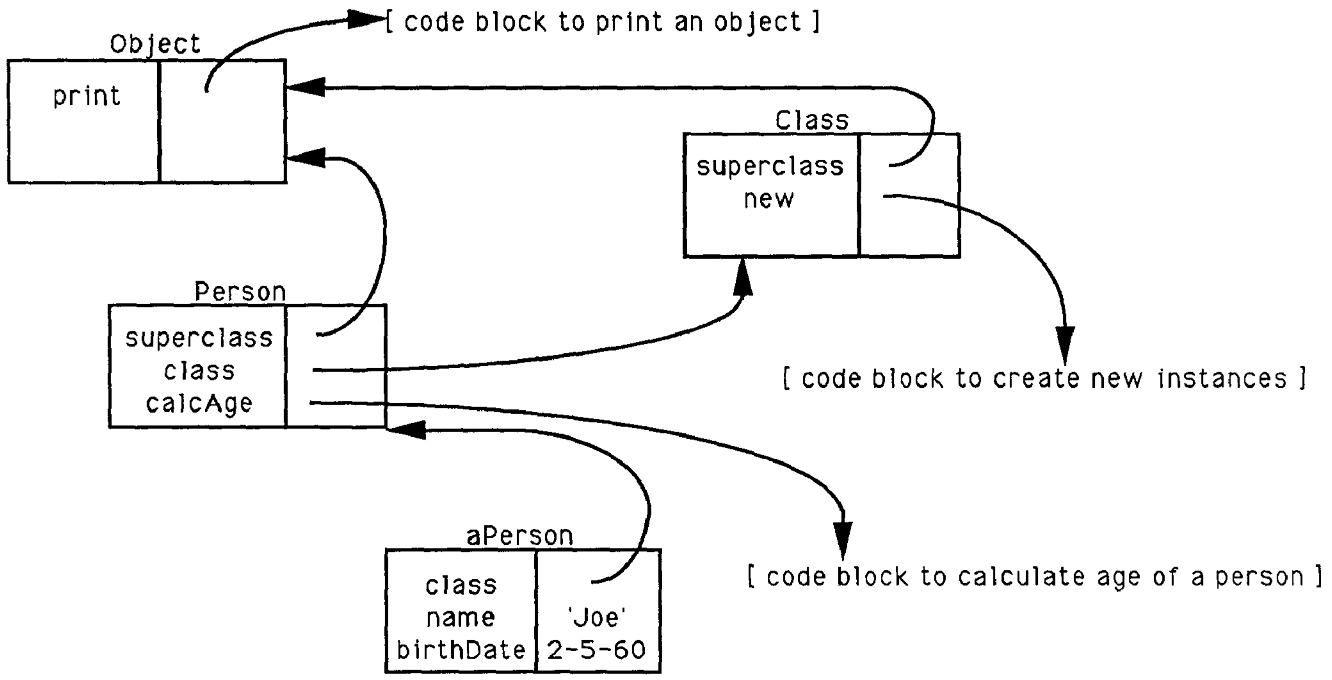

[code block to print an object]
Object
print
Class
superclass
new
Person
superclass
class
calcAge
[code block to create new instances]
aPerson
class
name
birthDate
'Joe'
2-5-60
[code block to calculate age of a person]

Figure 1. Implementing Classes With Prototypes

bined, as in Flavors [Flavors 86]. Both these languages provide extensions to specify which method(s) to invoke. Smalltalk allows compound selectors to exactly specify which methods to invoke, and Flavors provides a rich set of method-combination types to invoke more than one method and combine the results. However, multiple inheritance can become unwieldy when inheritance is used to model disjoint conceptual hierarchies. Classes impose a strict inheritance hierarchy and require that all instances have the same structure that is determined at instance creation.

Multiple inheritance has been proposed in prototypical languages as well. In the Self language [Ungar Smith 87], a prototype may have multiple parents which to delegate the message. Name conflicts among the slots of multiple parents are resolved by only searching on the path from the receiver of the message (self) to the object that has a slot with a code block from which the message was sent. Exemplar based Smalltalk [LaLonde Thomas Pugh 86] allows two different kinds of multiple inheritance: AND inheritance and OR inheritance. With AND inheritance, an exemplar inherits the union of the attributes and methods of its parents, while with OR inheritance it only inherits from one of its multiple parents.

Rule-Based Delegation

It is a basic knowledge representation dilemma whether to represent concepts as abstract sets (classes) or as concrete prototypes. Classes embody the common functionality that is believed to be true of all members of a set. A prototype, on the other hand, represents an actual instance of the concept, one believed to be a typical member of the set. Other members of the set refer to the prototype to share the characteristic functionality of all members. Classes are advantageous because they guarantee that all instances have the same external interface, and allow for specialization through subclassing. Prototypes are good for representing default knowledge and exceptional objects. A system that supports both prototypes and classes can gain the best of both worlds. Such a system can aid initial software development and encourage exploratory programming. The key to such a system is the flexibility provided by delegation. However, one drawback of delegation is the difficulty in recognizing and enforcing the semantics associated with delegation. Delegation can be used to model various semantics, including single and multiple inheritance, but the flexibility of delegation tends to obscure what semantics are being modelled.

The purpose of rule-based delegation is to provide the programmer with a mechanism to model the various semantics of delegation, especially in the context of multiple inheritance. In this system, the delegation mechanism is governed by the evaluation of a set of rules possessed by each object. In this way, the implementation of delegation is declarative and centralized in a single location, instead of embedded in the methods of many objects. With rule-based delegation, the delegation mechanism can be dynamic instead of fixed in a method search algorithm. Since each object possesses its own rule set (which can be shared), delegation can be customized per object. Rule-based delegation allows the relationship between the receiver and other objects to determine to which object to delegate the message.

Rule-based delegation works in the following way. When an object receives a message, each rule in the object's rule set is evaluated until a rule is found in which all the conditions of the rule are true. If such a rule is found, it is fired, i.e. its action code block is executed. The action block of a rule is responsible for returning a value for the message. If the result of the firing of a rule is nil (actually a "special" nil value, since ordinary nil may be an appropriate result), then the next rule with true conditions is fired. The rules are prioritized by a weight value assigned to each rule. If no rules are eligible to fire, then the message was not understood and an error message is output.

A rule is composed of conditional parts and an action code block. A rule can have many conditional parts and each one must evaluate to true or false. If all conditional parts evaluate to true, then the rule can fire. The action block of a rule returns a value for the message. The action block can access the receiver directly, forward the message to other objects, delegate a new message to other objects, alter the receiver (by adding new slots to self, for example), or alter future delegations (by adding new rules to the delegation rule set). To achieve rule-based delegation, statements inside the conditional parts and the action block of a rule can refer to self, the client, the message and its arguments, or any previous messages and the objects to which they were sent.

Rules for Rule-Based Delegation

The following section discusses individual rules that have been implemented for rule-based delegation. The discussion will focus on the rationale for the rules, with examples given where the rules prove useful. Psuedo code will illustrate the conditional parts and action block of each rule.

Rules When the Receiver Has a Corresponding Slot

When an object is sent a message for which it has a corresponding slot, the object typically returns the object at that slot. This object can be a stored value or it can be computed. Since rules determine what action to take when a message is sent, there must exist some rules that retrieve the value at the slot and return it. Obviously, these rules are necessary to end the delegation of the message, so they are given a higher priority than other rules. In fact, two rules are necessary to retrieve the object at a slot. One rule (the blockSlotRule) takes effect when the value of the slot is a block, since the block must first be executed. The other rule (the slotRule) takes effect when the value of the slot is anything other than a block, in which case the object is returned directly.

blockSlotRule
```
IF [ client has a slot corresponding to the message ]
AND [ the value of the slot is a block of code ]
THEN [ execute the block and return the result ]
```

slotRule
```
IF [ client has a slot corresponding to the message ]
THEN [ return the value at the slot ]
```

Is-A Rules

The Is-A relationship as described in [Brachman 83], can model a number of different semantics for inheritance. The following "Is-A" rules delegate the message according to the meanings associated with specific slot names that correspond to the different meanings of the Is-A relationship. Class-based systems model the Is-A relationship in two ways: one is the relationship between an instance and its class, and the other is the relationship between a class and its superclass. To allow both single and multiple inheritance with these relationships, four rules are used: the instanceRule, the instanceRuleForMultipleInheritance, the subclassRule, and the subclassRuleForMultipleInheritance.

Both instance rules check to see if the receiver has a slot named *class, instanceOf,* or *isMemberOf.* If so, then the object is considered to be an instance of the class-representing prototype contained in the slot. These rules also check that the current binding of the pseudo-variable "self" is the same as the pseudo-variable "client". If this is the case, then the object is the original receiver of the message, and the message should be delegated to the class-representing prototype. This correctly handles the situation mentioned previously when a message is initially sent to a class (Person) and should be delegated to class Class.

```
instanceRule
IF [ client has a slot named "class", "instanceOf", or
     "isMemberOf" ]
AND [ client = self ]
THEN [ delegate the message to the object contained
         in the slot ]
```

```
instanceRuleForMultipleInheritance
IF [ client has a slot named "class", "instanceOf", or
     "isMemberOf" ]
AND [ the value at the slot is a collection ]
AND [ client = self ]
THEN [ delegate to each class according to the Smalltalk
         multiple inheritance algorithm ]
```

The two subclass rules check to see if the receiver has any slots named *IsA, superclass, subclassOf, subsetOf, aKindOf,* or *specializationOf.* If it does, then the receiver is considered to be a subclass of the class-representing prototype contained in the slot. Note that the implementation of multiple inheritance for the Is-A rules is the Smalltalk multiple inheritance algorithm. This implementation of multiple inheritance may be overridden by other rules described later.

```
subclassRule
IF [ client has a slot named "IsA", "superclass", "subclassOf",
     "subsetOf", "aKindOf", or "specializationOf" ]
THEN [ delegate the message to the object contained
         in the slot ]
```

```
subclassRuleForMultipleInheritance
IF [ client has a slot named "IsA", "superclass", "subclassOf",
     "subsetOf", "aKindOf", or "specializationOf" ]
AND [ the value at the slot is a collection ]
THEN [ delegate to each superclass according to the
         Smalltalk multiple inheritance algorithm ]
```

Figure 2 illustrates the use of the Is-A rules. In this example, myOffice is an instance of class OfficeAtHome. OfficeAtHome has multiple superclasses: class Home and class Office. Both class Home and class Office are subclasses of class Dwelling. When a message is sent to myOffice, the instanceRule delegates the message to the object contained in the slot "class". If class OfficeAtHome does not understand the message, the subclassRuleForMultipleInheritance is fired since this object has a slot named "superclass" and the object contained in this slot is a collection. This rule searches all superclasses for a slot corresponding to the message. If more than one superclass

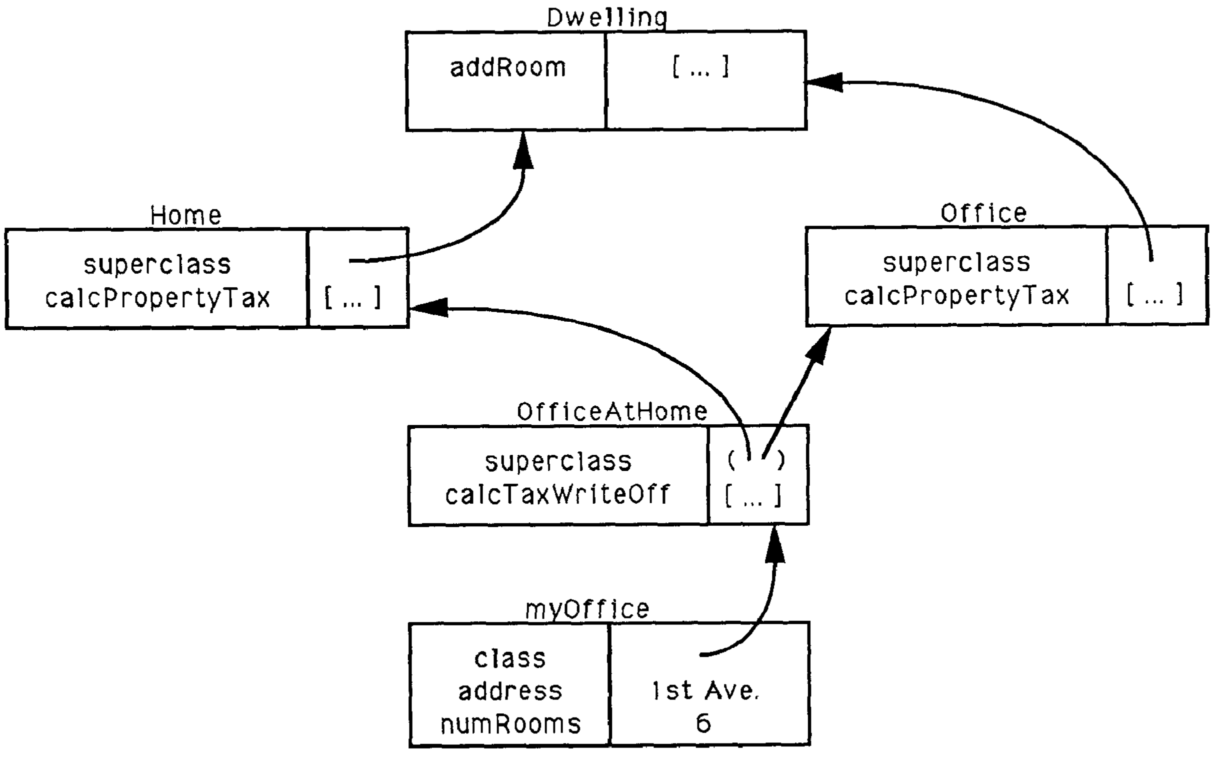


Figure 2. The Is-A Rules

contains a slot corresponding to the message, it is an error. For example, if the message was *calcPropertyTax*, a corresponding slot would be found in both class Home and class Office, so an error is signalled. It is not an error if the same slot is found via different paths. For example, if the message was *addRoom*, a corresponding slot is found when delegating the message to both class Home and class Office. However, the slot was found in the same object (class Dwelling), so it is not an error.

Other Rules for Multiple Inheritance

As mentioned earlier, the Self language constrains the search for a slot only on paths that contain the sender. The pathOfSenderRule checks to see if the object that just received the message (the client) has any slot values that are clients for a currently invoked method (i.e. a code block that is currently executing). If the receiver has any slot values that are also clients for a currently invoked method, then that object exists on the path of the sender. The pathOfSenderRule delegates the message to the object on the path.

Figure 3 illustrates how the pathOfSenderRule works when parent objects have slots with the same name. In this example, aVersatileAthlete is both a football player and a baseball player. The object aVersatileAthlete has two parents, the footballPlayerPart and the baseballPlayerPart, that both have a slot named *position*. The footballPlayerPart contains the slots for an individual instance of a football player and the class FootballPlayer contains slots that are shared by all football player instances. The baseballPlayerPart contains slots for an individual baseball player and is an instance of class BaseballPlayer.

Suppose the message *isRunningBack* is sent to aVersatileAthlete. This message returns true if the football player is a fullback (#FB) or halfback (#HB). The method corresponding to this message is found in class FootballPlayer as a result of being delegated to the footballPlayerPart because of the subpartsRule (discussed later) and then to the class FootballPlayer because of the instanceRule. As the method *isRunningBack* is executed, the message *position* is sent to self (aVersatileAthlete). At this point, the delegation path of the initial message (*isRunningBack*) is from aVersatileAthlete to the footballPlayerPart to the class FootballPlayer. The pathOfSenderRule overcomes the problem of deciding to which parent to delegate the message *position* by limiting the search path of the slot lookup. When the message *position* is sent to self (aVersatileAthlete), the message is delegated only to the footballPlayerPart because the footballPlayerPart is on the path of the sending method.

```
pathOfSenderRule
IF [ client has a slot that is on the path from self to the
     object containing the currently executing code block ]
THEN [ delegate the message to the object contained
       in the slot ]
```

Also mentioned earlier was the capability provided by Flavors to allow more than one method of an instance's multiple superclasses to be invoked and their results combined into a single value. The methodCombinationRule recognizes when a method-combination type is defined for the given message. Since method-combination types are defined on a class/message pair, it is necessary to determine the class of the object receiving the message and then determine if a method-combination type is defined for that class and the message. This requires that each class-representing prototype have a slot

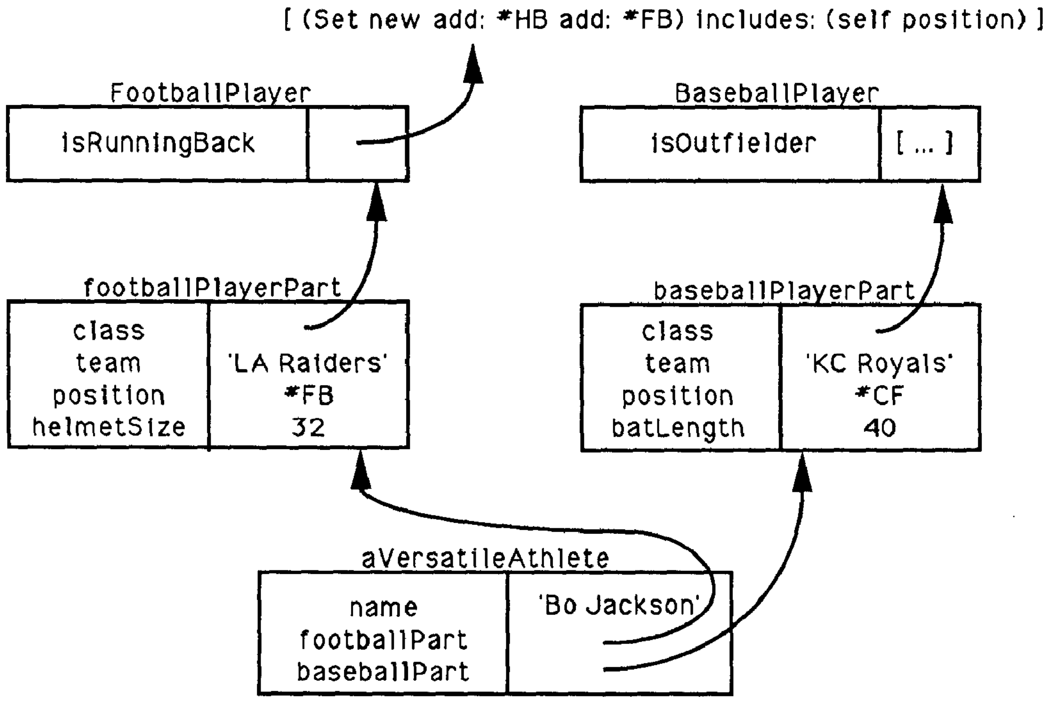


Figure 3. pathOfSenderRule

called "className" that contains the name of the class. The method-combination rule makes sure a method-combination type is defined for the given message, and if so, it executes the code block associated with that method-combination type. The arguments to the method-combination code block are an ordered collection of class-representing objects that corresponds to the component ordering of superclasses (i.e. the class hierarchy where the most specific classes are first and more general classes are last). These are the objects to which the message will be delegated and from which the return values will be combined.

```
methodCombinationRule
IF [ client has a slot named "class" ]
AND [ there is a method combination type corresponding
      to the class and the message ]
THEN [ execute the method combination code block ]
```

A-Part-Of Rules

Frequently, a message may be sent to an object that is composed of many subparts, one of which has a slot corresponding to the message. For example, the message *numberOfCylinders* may be sent to an object representing an automobile. The automobile may have slots containing objects that correspond to its subparts, such as engine, body, chassis, passenger compartment, etc. The engine subpart, in turn, may have slots that contain its subparts, such as engine block, crankshaft, electrical system, etc. The slot that contains the number of cylinders is contained in the engine block subpart of the engine. For the prototype representing the aggregate automobile to return a result for the message *numberOfCylinders*, it must delegate the message to its engine subpart and the engine subpart in turn must delegate the message to its engine block subpart. The subpartsRule delegates the message to each subpart of a composite object. If more than one subpart is found that has a corresponding slot, all the results are returned in an ordered collection. This rule can result in the message being delegated to many objects which do not have a corresponding slot before the appropriate object is found. Consequently, it is desirable that this rule be fired only when no other rules can be used, since it may result in a large number of unsuccessful delegations.

To overcome the proliferation of message delegations resulting from the subpartsRule, it is desirable that when the appropriate delegation path is found in response to a message, that the path is "remembered" for future delegations of the same message. Not only should the delegation path be remembered for the object that received the message, but also for all similar objects, i.e. all objects that are instances of the same class. The subpartsRule accomplishes this by creating a new rule for specifically handling the message and placing the new rule in the rule set of the class of the receiver. In this way, if any other instances of the same class receive the same message, the new rule will attempt to delegate the message along the same path that was previously found to be successful. Of course, if the correct path has changed or the receiver is an exceptional object, the new rule will fail and other rules may attempt to delegate the message.

```
subpartsRule
IF [ true ]
THEN [ delegate message to each subpart and group
       any non-nil results in a collection ]
```

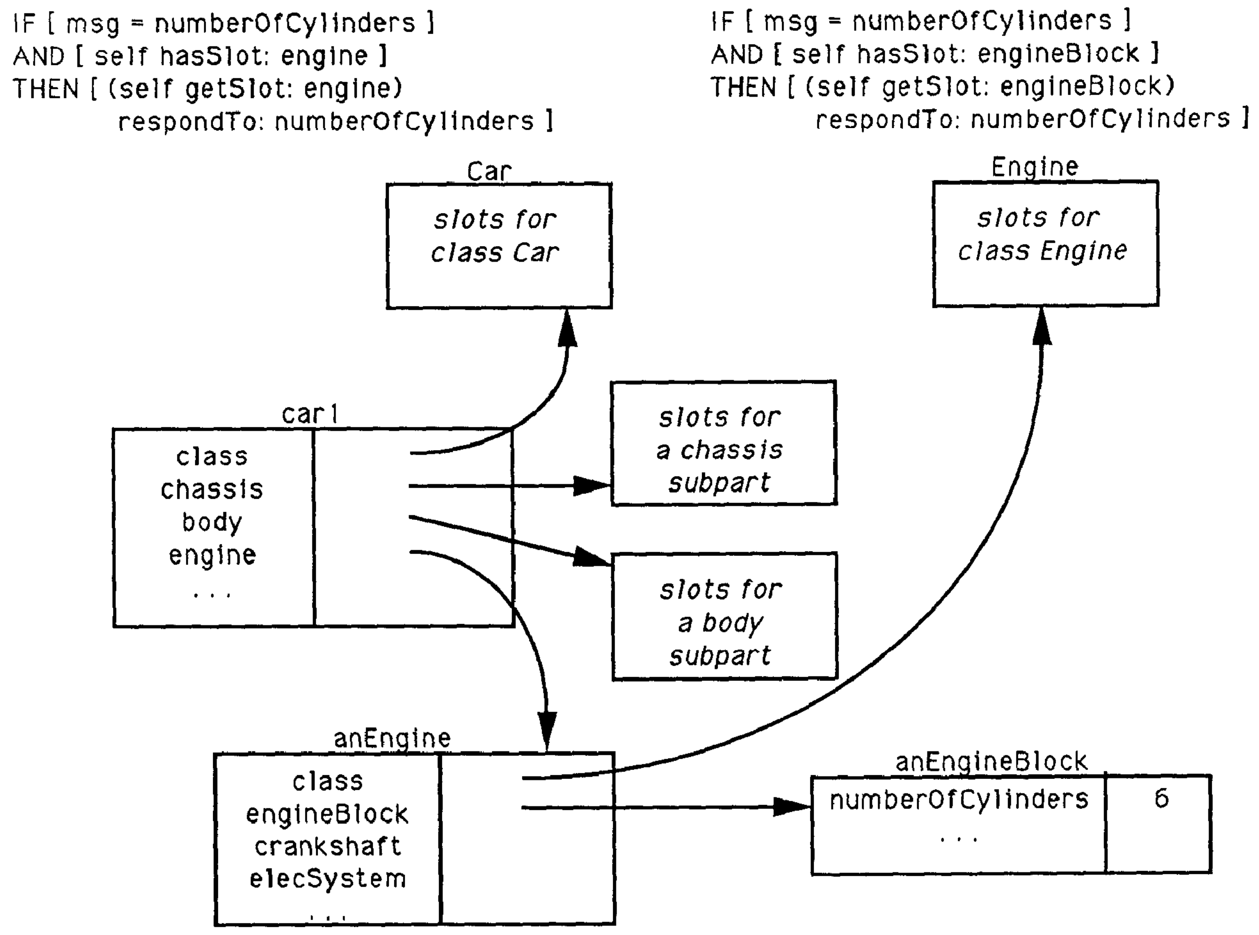
Figure 4 illustrates how the subpartsRule creates a new rule to handle future delegations of the same message. When the message *numberOfCylinders* is initially sent to a car object (car1), the object does not have a corresponding slot. All other rules fail to delegate the message successfully, so the subpartsRule tries delegating the message to each subpart of the car. The delegation of the message to the chassis subpart fails, as well as delegating the message to the body subpart. However, when the message is delegated to the engine subpart, the following happens. The engine object tries delegating the message with other rules that are unsuccessful, so the subpartsRule is called upon again to delegate the message to the subparts of the engine. When the message is delegated to anEngineBlock object, the corresponding slot is found and a value is returned as the result of the message. The subpartsRule that was invoked for the engine subpart recognizes that the delegation was successful, so it creates a new rule for this delegation. The new rule checks if the message is *numberOfCylinders* and if the receiver has a slot named *engineBlock*. If so, the rule will delegate the message to the object contained in the *engineBlock* slot. This new rule is placed in the rule set of class Engine so that all instances of Engine will benefit from this rule. A similar process occurs when the result of the message is returned to the subpartsRule that was invoked for the car1 object. In this case, another new rule is created that checks if the message is *numberOfCylinders* and if the receiver has a slot named *engine*. This new rule is placed in the rule set of class Car. Now if any instances of car are sent this message, the message will be delegated to the class by the instanceRule, and the class will have a specific rule to handle the message.

In some cases, it may be desirable for the subpart of an object to inherit attributes from its aggregate object. For example, a car door object may be sent the message *color* to get the color of the door. In this case, the door is a part of the body of car, and the door should inherit the same color as the body of the car. The APartOfRule delegates the message to the object(s) contained in a slot named *APartOf* or *APO*. One may think of the subpartsRule as delegating the message "inward" for a composite object, while the APartOfRule delegates the message from the subparts of an object "outward" to the aggregate object. There may be more than one object contained in the *APartOf* slot for an object. In this case, the message is delegated to each object in the collection and the results are grouped in a collection and returned.

```
APartOfRule
IF [ client has a slot named "APartOf" or "APO" ]
THEN [ delegate the message to the object contained
       in the slot ]
```

The Order Rules Are Fired

If more than one rule can fire when a message is sent, the order the rules are tried is determined by the weight associated with each rule. As mentioned before, the rules with the highest priority are the ones for which the receiver has a slot corresponding to the message. The rules to be tried next are the ones that implement various solutions for multiple inheritance: the methodCombinationRule and the pathOfSenderRule. The methodCombinationRule is tried first because method-combination is a programmer specified mechanism that is intended to override any default mechanism. The pathOfSenderRule is tried next because it limits the search path for multiple



IF [msg = numberOfCylinders]
AND [self hasSlot: engine]
THEN [(self getSlot: engine)
respondTo: numberOfCylinders]
IF [msg = numberOfCylinders]
AND [self hasSlot: engineBlock]
THEN [(self getSlot: engineBlock)
respondTo: numberOfCylinders]
Car
slots for class Car
Engine
slots for class Engine
car1
class
chassis
body
engine
. . .
slots for a chassis subpart
slots for a body subpart
anEngine
class
engineBlock
crankshaft
elecSystem
. . .
anEngineBlock
numberOfCylinders
6
. . .

Figure 4. Creating a New Rule with the SubpartsRule

inheritance and therefore should be fired before the Is-A rules for multiple inheritance, which search the paths of all superclasses. The Is-A rules are tried next, before the rules that search composite objects (the APartOfRule and the subpartsRule) because the Is-A rules implement classical single and multiple inheritance. The subpartsRule is tried last because it can result in the delegation of the message to a large number of objects before the appropriate one is found. However, the subpartsRule may create new rules for specific messages and these new rules have weight values that are higher than the methodCombinationRule. Rules have weights which order them as follows:

1. blockSlotRule
2. slotRule
3. any new rules created by the subpartsRule
4. methodCombinationRule
5. pathOfSenderRule
6. instanceRuleForMultipleInheritance
7. instanceRule
8. subclassRuleForMultipleInheritance
9. subclassRule
10. APartOfRule
11. subpartsRule

Conclusions

Rule-based delegation is a flexible mechanism for controlling delegation in a system with classes and prototypes. With rule-based delegation, a number of multiple inheritance solutions can be integrated in a common framework. In this way, the programmer can choose the appropriate semantics of delegation for the application. The cost of the flexibility of rule-based delegation is performance. Every message sent can result in the evaluation of many rules. In many cases, either the receiver has a slot corresponding to the message or delegation follows the path of classical single inheritance. To increase performance, rules that implement such behavior could be hard-coded into the delegation mechanism at the expense of flexibility.

To explore rule-based delegation, the functionality of prototypes and rule-based delegation has been implemented in Smalltalk-80. All of the rules mentioned in this paper have been implemented and work. Future areas to be explored include other conflict resolution strategies for rules, interactive rules in which the conditional part of a rule prompts the user for more information, and rules to promote prototypes to classes [Stein 87].

The advent of systems such as exemplar based Smalltalk illustrates the trend toward including the functionality of prototypes in object-oriented languages. A law-based approach [Minsky Rozenshtein 87] is another similar way of controlling delegation. As languages add this functionality, the semantics associated with delegation will need to be clearly understood and made available to the programmer. This research has illustrated one means in which to specify the semantics of delegation in a simple framework.

Acknowledgements

I would like to extend a special thanks to Professor Jim Diederich and Professor Jack Milton of the University of California at Davis for providing stimulating discussion and valuable guidance in the writing of this paper.

References

Borning, A. "Classes versus Prototypes in Object-Oriented Languages", Proceedings of the ACM/IEEE Fall Joint Computer Conference, Dallas, TX, November, 1986.

Borning, A., D. Ingalls. "Multiple Inheritance in Smalltalk-80", Proceedings of the AAAI Conference, Pittsburgh, PA., 1982.

Brachman, R. "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks", Computer, October 1983.

Flavors, Chapter 17 of "Symbolics Common Lisp: Language Concepts", August 1986.

Goldberg, A., D. Robson. "Smalltalk-80: The Language and its Implementation", Addison-Wesley, 1983.

LaLonde, W., A. Thomas, J. Pugh. "An Exemplar Based Smalltalk", OOPSLA '86 Proceedings.

Lieberman, H. "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems", OOPSLA '86 Proceedings.

Minsky, N., D. Rozenshtein. "A Law-Based Approach to Object-Oriented Programming", OOPSLA '87 Proceedings.

Moon, D. "Object-Oriented Programming with Flavors", OOPSLA '86 Proceedings.

Stein, L. "Delegation Is Inheritance", OOPSLA '87 Proceedings.

Ungar, D., R. Smith. "Self: The Power of Simplicity", OOPSLA '87 Proceedings.