ATOMIC DELEGATION: OBJECT-ORIENTED TRANSACTIONS

When you modify a type, atomic delegation updates the functionalities of the types that delegate to the modified type. This mechanism permits dynamic binding and code reuse in atomic actions.

MEHMET AKSIT
JAN WILLEM DIJKSTRA
University of Twente
ANAND TRIPATHI
University of Minnesota

Atomic delegation, a new object-oriented linguistic mechanism, lets you create dynamically defined classes of atomic actions. The technique is also uniformly integrated with other object-oriented techniques such as inheritance and dynamic binding.

Atomic delegation is more powerful than statically defined transactions because it supports the construction of open-ended systems and systematic reuse of software involving atomic operations.

Delegation¹ allows an object receiving a request message to forward it to some other designated objects for processing. Of course, forwarding a request message is not new and is common in distributed

computing for load sharing or finding network resources and services. But in object-oriented programming, the concept of delegation is a generalization of inheritance.² An object's behavior is implemented by the objects to which it delegates the request messages. Using delegation, an object can change its behavior. Thus, delegation is useful in building extensible and open systems.

In concurrent and distributed programming, atomic actions³ are important to maintain consistent states of shared resources. An atomic action is a sequence of operations that executes as a single, indivisible operation; an aborted execution of such a sequence is never permanent or vis-

ible to other actions. Atomic actions are common in database systems as transactions. More recently, Barbara Liskov integrated them into object-based programming paradigms in Argus,⁴ where every interface procedure of an object is implemented as an atomic sequence of operations.

The conventional model of an atomic action as a statically defined sequence of operations is not compatible with such object-oriented techniques as inheritance and dynamic binding, which are needed to support extensible system behavior. Consider a simple office system modeled with object-oriented techniques: An object representing an administrative secretary — an object whose interface methods correspond to financial transactions — executes all financial transactions requested by the office employees. Each transaction involves requester authentication followed by operations on a financial database. To add new functionalities to the object representing the financial database you must add new interface methods to the object representing the secretary or modify existing methods. For building extensible systems, this is undesirable. Adding new functionalities to a class should not require you to explicitly modify descendant classes in the inheritance hierarchy to incorporate the upgrades.

Atomic delegation lets an object delegate a sequence of request messages to one or more designated objects as an atomic action. By changing the delegation relationship between objects or by modifying the functionality of an object to which messages are delegated, you can dynamically change the set of atomic actions supported by an object.

We use the Sina language⁵ to illustrate the utility of atomic delegation, modeling a real-world problem using object-oriented techniques. We implemented Sina on a Sun-3 workstation.

DISTRIBUTED OFFICE

In this article we use a small example of a real-world system, an office with a number of departments.

Figure 1 is a functional view of the of-

fice system. Each department has a manager, a secretary, and employees. All share a common set of responsibilities: printing, reporting, sending and receiving mail, and managing a calendar. A department secretary coordinates some common responsibilities.

This office also has a director and a financial manager, who have the additional responsibilities of managing financial transactions. A manager can access financial information and make a deposit or withdrawal from the office budget. The financial manager carries out the monetary transactions requested by a department secretary on behalf of some manager.

To model such an office system using object-oriented design paradigms, you would use objects to represent the entities and resources in the office.

There are a number of well-known de-

sign problems in building such an office environment as a distributed computing system. The office staff concurrently shares the objects making up the office environment. Therefore, you need to control and synchronize access to shared objects.⁶ System failures may result in inconsistent data, so you must integrate such reliability mechanisms as atomic actions into the system design. Finally, you need information-protection mechanisms for critical and sensitive data.

The office organization may change: New positions and responsibilities may be introduced or existing ones may be modified or canceled. Also, an employee's expected functions may change. Therefore, you must provide mechanisms for on-line integration of new software tools or upgrades without disrupting ongoing office activities.

Figure 1. A distributed office system.

type personVerify interface is

begin

method verifyPerson(string as name, pwd, key) **returns** nil; /* interface methods */

method definePerson(string as name, pwd,) **returns** boolean;

messages { self.verifyPerson(*), self.definePerson(*) }; /* interface predicate */

end;

Figure 2. Type personVerify interface declaration.

DATA-ABSTRACTION MECHANISMS

An abstract data-type definition in Sina consists of two parts: an *interface* definition and a *local* definition. An interface definition specifies a type's externally visible behavior, while a local definition contains the data structures and the code that implement the type. An abstract type represents a class of objects with similar behavior.

An abstract type's interface definition has three components: *method declarations*, *interface objects*, and an *interface predicate*. An interface definition declares the methods implemented by the type that other objects can invoke. It also declares the interface objects encapsulated by each instance of the type. The abstract behavior of such interface objects becomes visible as the behavior of the instances of the new type being defined.

Such interface objects are Sina's primary mechanism for supporting multiple inheritance. The interface predicate specifies the messages that objects of its abstract type can accept. It also specifies the precedence rules in searching for a requested method in the classes of its interface objects and among its own interface methods.

Figure 2 shows only the interface part of the type personVerify, a basic module in modeling the example office system. Objects of this type would be used to authenticate a person's identity. The method definePerson accepts two string argu-

ments, name and pwd, and stores them internally as an authorization identity.

Interface objects are Sina's primary mechanism for supporting multiple inheritance. The interface predicate specifies the messages its type can accept.

These two parameters must be encrypted according to the algorithm implemented by type Cipher. The method verifyPerson verifies its arguments, name and pwd, as an authorization identity.

The third argument, key, decodes these two arguments. If authorization fails, an exception condition terminates execution of the invoked method. Not shown here is the implementation of this type, which uses a local instance of type Cipher to decode the arguments of method verifyPerson.

Message expressions. An object can interact with another object only when it invokes its interface methods by executing message expressions. A message expression consists of a receiver object, a method selector, and a list of parameters for the method.

For example, a user of personId, which is an instance of type personVerify shown in Figure 2, would initialize the authorization identity of personId by executing the message expression

```
personId.definePerson
('Barbara', 'PBarbara');
```

This sends a request message to the receiver object personId. The method selector in the above expression is definePerson, and 'Barbara' and 'PBarbara' are the parameters.

An object can invoke an operation on

itself by using the pseudovariable *self* as the receiver object in a message expression. The receiver in a message expression can also be specified with a pointer to the object. For example, if objPointer is a pointer to personId, you can rewrite the above expression as

```
@objPointer.definePerson
('Barbara', 'PBarbara');
```

Atomic messages. You can define a message expression that sends to an object a sequence of related messages whose processing is performed as a single, indivisible, atomic action. Such a sequence has one or more messages, each with a method selector and appropriate parameters, grouped together within a pair of aggregate operators, <...>. The system processes such messages sequentially, treating the whole sequence as an atomic action. For example:

```
Barbara.<isFree(parameters),
arrange(parameters)>;
```

This atomic message sequence sends to Barbara an indivisible sequence of two messages with the method selectors isFree and arrange. The messages are processed as an atomic action.

During the processing of an atomic message, other atomic messages may be sent. An atomic message sent by a method that is itself processing an atomic message is called a nested atomic message. When a nested atomic message fails, the atomic message that invoked it also fails. An atomic message's failure does not change the system's state.

Interface predicates. A type's interface predicate defines the rules for searching for the method selectors and for accepting messages for processing.

The basic building block for an interface predicate, a *message proposition*, is similar to a message expression: It has a receiver object, a method selector, and parameters. (If the receiver object in a message proposition is *self*, you can omit *self* in the proposition.) A message satisfies a message proposition if the received message's selector and the arguments match the selector and the argument types, respectively, in the message proposition.

The following message proposition has a receiver object *idManager* of type *personVerify*:

```
idManager.verifyPerson(*)
```

It satisfies any message with selector *verifyPerson* that can be processed by the object *idManager*.

If a message argument has the character ***, the corresponding method declaration determines the argument's validity for a given message. If *** is a method selector, it is replaced by all the matching selectors as defined by the receiver object's interface predicate.

Such a specification may generate a set of propositions. These generated propositions form an ordered list defined by the receiver object's interface predicate. For example, the message expression *idManager.*(*)* is equivalent to the list

```
{idManager.verifyPerson(*),
 idManager.definePerson(*)}
```

of message propositions, which you obtain by substituting for the method selector *** every method selector that *idManager* accepts according to the interface predicate

of type *personVerify*.

You can combine a sequence of message propositions using the aggregate operator to form an aggregate proposition, which is syntactically similar to an atomic message. An atomic message sequence satisfies an aggregate proposition if there is a one-to-one correspondence between both their elements, and each element in the message sequence satisfies the corresponding proposition in the aggregate proposition.

For example, the aggregate proposition

```
<securityData.verifyId(*),
 financialManager.*(*)>
```

would be satisfied by an atomic message sequence of two messages. The first message in the sequence must have method selector *verifyId* and valid parameters, and the second message can have any method selector valid for the object *financialManager*. Failure of either message to process aborts processing of the entire atomic message sequence.

An interface predicate is an ordered list comprising message propositions or aggregate propositions. An instance of a type accepts a message (or an atomic message sequence) for processing only if the received message (or message sequence) satisfies a proposition in this list. Messages are matched with the propositions according to their order in the set defined by the interface predicate. The first proposition that satisfies the message is used to process the message.

Consider the interface predicate of type *securityBase*:

```
{idManager.*(*), self.verifyId(*),
 self.insertId(*), self.deleteId(*)}securityBase
```

Here a name in the superscript indicates the name of the type whose interface proposition is being referenced. According to this interface predicate, an instance of *securityBase* accepts all messages accepted by the object *idManager*, and it also accepts three other methods implemented by itself. This interface predicate is equivalent to an ordered list of propositions:

```
{{verifyPerson(*),
 definePerson(*)}personVerify, verifyId(*),
 insertId(*), deleteId(*)}securityBase
```

Inheritance and delegation. The interface-predicate mechanism treats conventional inheritance as a special case of delegation. Sina allows an object to delegate a request message that it receives either to some interface object or to some external object whose name appears in the interface predicate's propositions. Delegation of messages to interface objects corresponds to conventional single or multiple inheritance, where the abstract state of an interface object is a part of its encapsulating object's state. Precedence rules implied by the interface predicate in searching for method selectors solve the general problem of duplicate method names inherited from different interface objects.

An example illustrates how the type *securityBase* inherits from the type *personVerify*. Figure 3 shows the interface declaration of type *securityBase*. This new type implements an access-control list that stores some users' name-password pairs. A

type securityBase interface is

begin

objects personVerify as idManager;

method verifyID(string as name, pwd, key) **returns** nil;

method insertID(string as managersName, managersPwd, name, pwd, key) **returns** nil;

method deleteID(string as managersName, managersPwd, name, pwd, key) **returns** nil;

messages { idManager.*(*), self.verifyId(*), self.insertId(*), self.deleteId(*)};

end;

/ interface object */*

/ interface methods */*

/ interface predicate */*

Figure 3. Type *securityBase* interface declaration.

type secretary interface is

```

begin
  objects reporting as editing;
  method name() returns string;
  method setName( string as personsName ) returns nil;
  messages { name(*), setName(*), editing(*), <securityData.verifyId(*), financialManager.*(*)>;
end;

```

Figure 4. Type secretary interface declaration.

type financialManagement interface is

```

begin
  method status() returns integer;
  method deposit( integer as dollars ) returns nil;
  method transfer( integer as accountNo, dollars ) returns nil;
  messages { status(*), deposit(*), transfer(*), };
end;

```

Figure 5. Type financialManagement interface declaration.

Figure 6. Sina objects for implementing the distributed office.

system administrator, who is the only one allowed to make changes, controls the list. Therefore, the securityBase object also stores the system administrator's name-password pair. This functionality is inherited from type personVerify.

The securityBase object inherits the behavior of personVerify by creating an instance (called idManager) of this type in its interface. The securityBase object also introduces three new methods — verifyId, insertId, and deleteId — which the system administrator uses to update the access-control list.

The method verifyId validates its input parameters, name and pwd, using the internally stored access-control list, and raises an exception if it cannot find the input parameters as a name-password pair. The method insertId validates the first two parameters using the name-password pair for the system administrator. If they are validated, it adds a name-password pair, specified by the next two parameters, to the access-control list. Similarly, deleteId removes from the access-control list a name-password pair. These three methods use the last argument, key, to encode and decode the name-password pairs.

When an instance of securityBase receives a request, then, according to its interface predicate, it first searches the idManager interface for the method selector in the request. If the message satisfies any message proposition in the interface predicate of type personVerify, it is delegated for processing to idManager. If the method selector does not match any personVerify interface methods or the parameters do not match with the expected parameter types, then the search continues in the remaining part of the ordered list of propositions in the securityBase interface predicate.

Atomic delegation and extensibility. An aggregate proposition in an object's interface predicate specifies atomic message sequences that it can accept for processing. Delegation within an aggregate proposi-

tion allows you to define a dynamic set of atomic actions.

Figure 4 illustrates this atomic delegation construct with type Secretary's interface declaration. Type Secretary provides a department secretary's basic tools and implements two methods, name and setName. Also, it inherits from type Reporting by using an instance of this type as an interface object, and delegates elements of atomic messages to securityData and financialManager, which are external objects of types securityBase and financialManagement. Type Reporting implements the basic reporting functions such as printing a document and sending electronic mail. The ordered list of interface propositions for type Secretary is

```
{name(*), setName(*), editing.(*),
  <securityData.verifyId(*),
  financialManager.(*)>}Secretary
```

The last proposition in the predicate of Secretary — that is, <securityData.verifyId(*), financialManager.(*)> — provides a set of atomic actions for financial transactions. This proposition is satisfied by an atomic message sequence whose first message verifies the sender's identity. Only managers and directors can perform financial transactions. Type Secretary delegates the first message to securityData for name-password validation.

The second message, which is executed only if validation does not fail, is the financial transaction. It is delegated to the financialManager. The second message can contain any method selector acceptable to the object financialManager.

Figure 5 shows the interface of the type financialManagement, which models the financial manager's functions. Its tasks are to give a financial status, deposit funds, and transfer funds. The interface predicate is quite simple:

```
{status(), deposit(*),
  transfer(*)}financialManagement
```

All propositions in the predicate refer to methods implemented by the type itself.

Any new methods added to financialManagement's interface become automatically available to the department managers as valid financial transactions through the global object called departmentSecretary. You do not need to modify type Secretary.

MODELING THE OFFICE SYSTEM

Figure 6 illustrates the relationships between various classes and objects that we use to model the example office system. The type securityBase inherits from the type personVerify. Type Secretary implements financial transactions by delegating to two external objects, financialManager and securityData.

An employee is represented by the type Person, which delegates invocation requests to an external object called departmentSecretary for such office-related functions as mail, document preparation, and financial transactions. Type Person also delegates to calendarBase, an object of type calendarManager for scheduling meetings. An object of type Manager acquires all attributes of type Person by delegating to a global object called departmentManager, which is of type Person. However, it redefines and reimplements the methods name and setName so each manager can be assigned a different name.

A secretary's tasks are to print the documents prepared by the employees, handle mail, and support communication between the department managers and the financial manager. Figure 4 showed the interface part of type Secretary. Instances of type Secretary can respond to the messages name and setName.

The third proposition in Secretary's interface predicate indicates that if the method selector in a message is not name or setName, Secretary can search for it in the interface predicate of editing. Object editing, an interface object for the type Secretary, is an instance of type Reporting with the interface predicate

```
{printDocument(*), sendMail(*)}reporting
```

Since Editing is an interface object of each instance of Secretary and Secretary's interface predicate contains the proposition

editing.(*), type Secretary inherits all Editing's methods. This means that the departmentSecretary, which is an instance of type Secretary, provides the methods printDocument and sendMail to its clients.

An employee is represented by an instance of type Person. Figure 7 shows the interface definition and use of the type Person as it would appear to a user of our system. The interface predicate of type Person is

```
{name(*), setName(*), receiveMail(*),
  readMail(), initializeCalendar(*),
  calendarBase[index].(*),
  departmentSecretary.(*(*)}person
```

The message proposition name(*) indicates that all instances of type Person recognize the message name, which returns a string containing the name of the message's recipient. The type Person can define its own name by invoking the method setName. The method receiveMail is used to receive the letters sent by electronic mail.

The method initializeCalendar assigns a unique calendar manager to an instance of Person. This method also has one integer parameter for initializing a local variable called index. This local variable is used in the message proposition calendarBase[index].(*). Object calendarBase is a global array of elements of type calendarManager. The proposition indicates that if a message does not match any of the previous propositions in the interface predicate, the message is then delegated to an object in the array calendarBase. The search now continues in the interface predicate of type calendarManager. The variable index is used to select this array element.

Because a different value for index initializes each instance of type Person, each person can share the same code but still delegate messages referring to its own calendar manager object. Delegation to an element of an array using an index, where

Delegation to an array element using an index that can be changed at runtime is called dynamic delegation. This mechanism can also implement dynamic inheritance.

Figure 7. Type person interface declaration and use.

Figure 8. Type calendarManager interface declaration and use.

the index can be changed at runtime, is called *dynamic delegation*. The same mechanism can implement dynamic inheritance if you declare the array as an interface object.

The last proposition in the interface predicate of type Person is departmentSecretary. If an instance of type Person receives a message that does not match any other propositions, it delegates the message to departmentSecretary. This way every instance of type Person acquires type Secretary's functionality to print and

prepare documents, send mail, and execute financial transactions if it has the privileges of a manager or the director.

Figure 8 shows the interface of the type calendarManager. A calendarManager's tasks are to arrange and cancel appointments and to list the participants scheduled for an appointment. The methods arrange, cancel, and list implement these basic tasks. This type inherits from type Calendar through an interface object of this type. The type calendarManager's predicate is defined as

```

{<calendarData.isFree(*), arrange(*)>,
calendarData.status(*),
cancel(*),(*)}calendarManager

```

The first proposition, <calendarData.isFree(*),arrange(*)>, means that the message sequence consisting of isFree followed by arrange can only be executed atomically. Without this requirement, two persons might find out at about the same time that someone is free at a given time, and then arrange an appointment with that person.

In the last two message propositions,

the method Cancel cancels a meeting at a given date, while List displays the participants scheduled for an appointment. Both methods have one parameter representing the date. The two messages are also atomic to avoid any inconsistencies if a person requests a meeting-participant list while some participants are canceling their appointments.

Figure 9 shows the implementation of type Manager. Besides performing an employee's normal functions — printing, sending and receiving electronic mail, and scheduling with a calendar — an object of type Manager can also perform financial transactions, as reflected by its predicate:

```
{name(*), setName(*), financialStatus(*),
 deposit(*), transfer(*),
 departmentManager.*(*)}manager
```

The first two propositions in the list represent the operations for associating a string name with an instance of type Manager. The next three propositions — financialStatus(*), deposit(*), and transfer(*) — take care of a manager's financial transactions. The last proposition, departmentManager.*(*), specifies delegation to a global object called depart-

ment-Manager, which is of type Person. This delegation gives all instances of type Manager the behavior of type Person.

Figure 9 also shows an implementation of the method Transfer, which validates the name-password pair. In the atomic message

```
self.<verifyId(myName, myPass, key),
 transfer(accountNo, dollars)>
```

the first message validates the invoker's identity. Funds are transferred only if the name-password pair is valid. If the validation fails, an error condition aborts processing of the entire message sequence. The third argument, key, encodes the parameters myName and myPass. The local method computeName encodes the manager's name-password pair.

This atomic message sequence is sent to self, so the message is first searched for in Manager's interface predicate. The message does not satisfy the first three propositions, so it is delegated to departmentManager, which is of type Person. Type Person in turn delegates the message to departmentSecretary, which recognizes this atomic message. Type Secretary delegates the first component to securityData,

which verifies identification, and delegates the second component to the financialManager, which executes the financial transaction.

This example illustrates delegation as a general mechanism for inheritance and code reusability. The implementations of types Secretary, Person, and Manager show how a set of atomic actions can be defined and inherited using delegation. Also, the interface predicate of type Person shows that the delegation relationship between objects can be changed dynamically.

IMPLEMENTATION ISSUES

For synchronization, our implementation of atomic delegation gives each object an object manager through which atomic messages lock the objects they access, according to the two-phase locking protocol.⁶ Thus a process executing an atomic message cannot release any objects if it still needs to access any other objects. All objects accessed by an atomic message stay locked until the atomic message sequence commits or aborts.

Figure 9. Type manager interface declaration and use. The top window shows the interface declaration; the bottom window, implementations of methods transfer and computeName. The inset window shows an operation invocation on an object named Barbara, which is of type manager.

If the processing of an atomic message aborts, the objects modified by that message must return to their earlier states. The object manager creates a backup copy of each object before it is modified by an atomic message. Then, if an atomic message aborts, object states are restored with the saved copies before the locks are released.

You can also abort an atomic message by executing the message `self.error` ("An error message"). The objects changed by the atomic message are restored to their previous states. Thus, when you detect an error, you can abort an atomic message without side effects.

To detect deadlocks, a blocked process starts a probe computation.⁷ First the process sends a probe message to the object managers of the objects locked by the process. Each object manager propagates the probe message to the processes waiting to access its object. These waiting processes, in turn, propagate the probe to the object managers of the objects they have already locked. If the system is deadlocked, the probe message is eventually received by a process that has propagated the probe itself. This process then knows it is part of a deadlocked process cycle.

After a deadlock is detected, a search identifies a victim, the youngest process among the deadlocked set, which is aborted and restarted. A restarted process

retains its current age. The result is that the oldest process in the system is sure to commit successfully.

The integration of interface predicates, atomic message sequences, and delegation supports extensible and open systems involving atomic actions. The interface predicate construct is general so you can use it to construct the conventional object-oriented data abstraction and code-sharing strategies. You can also implement both inheritance and delegation.

Delegation in an atomic message sequence is key to dynamic binding and code reusability within atomic actions. When you modify a type, Sina's delegation mechanism automatically updates the functionalities of those types that delegate to the instances of the modified type. For example, if you upgrade type reporting or type financialManagement, then types Secretary, Person, and Manager automatically incorporate the new functionalities. The new functionalities also become visible as part of defined classes of atomic actions.

We implemented Sina using Smalltalk-80. At the University of Twente, we are implementing a new prototype using C++. We are also investigating integration of synchronization mechanisms into the interface predicate construct. ♦

Acknowledgments

We thank Albert Esterline for his many helpful comments during the revision of this article.

References

1. H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," *Proc. ACM OOPSLA Conf.*, ACM, New York, 1986, pp. 214-223.
2. P. Wegner, "Dimensions of Object-Based Language Design," *Proc. OOPSLA*, ACM, New York, 1987, pp. 168-182.
3. T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, Dec. 1983, pp. 287-318.
4. B. Liskov, "Distributed Programming in Argus," *Comm. ACM*, March 1988, pp. 300-312.
5. M. Aksit and A. Tripathi, "Data Abstraction Mechanisms in SINA/ST," *Proc. OOPSLA*, ACM, New York, 1988, pp. 267-275.
6. P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, June 1981, pp. 185-222.
7. D.P. Mitchell and M.J. Merritt, "A Distributed Algorithm for Deadlock Detection and Resolution," *Proc. ACM Symp. Principles of Distributed Computing*, ACM, New York, 1984, pp. 282-284.

Mehmet Aksit is an associate professor at the University of Twente, The Netherlands. His research interests include the application of object-oriented technology to distributed software-development environments and languages.

Aksit received an MS from the Technical University of Eindhoven and a PhD from the University of Twente. He is a member of the IEEE Computer Society and ACM.

Jan Willem Dijkstra works in research and development at Océ International in Venlo, The Netherlands. His research interests include concurrency in object-oriented systems, distributed object-oriented systems, and the persistence of objects.

Dijkstra received an MS in computer science from the University of Twente.

Anand R. Tripathi is an associate professor of computer science at the University of Minnesota. His research interests include distributed computing, fault-tolerant systems, parallel processing, and object-oriented programming.

Tripathi received a B.Tech in electrical engineering from the Indian Institute of Technology, Bombay, and an MS and a PhD in electrical engineering from the University of Texas at Austin. He is a member of the IEEE Computer Society and ACM.

Address questions about this article to Aksit at University of Twente, Computer Science Dept., P.O. Box 217, 7500 AE Enschede, The Netherlands.