NORTH-HOLLAND

An Object-Oriented Approach to Constructing Communication Protocols

JIUN-LIANG CHEN

FENG-JIAN WANG

and

YUNG-CHAO TING

Institute of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, R.O.C.

Communicated by Stephen S. Yau

ABSTRACT

The development of communication protocols for computer networks and distributed systems is an increasingly complex and cost-sensitive process. This paper presents an object-oriented concurrent (OOC) model for the development of communication protocols. This model consists of three kinds of entities: *data* entities, which represent the communicated data units, *state* entities, which describe their behavior, and *connection* entities, which are responsible for communication services. The state transitions in a protocol are modeled by using state entities, and a communication service is performed by a group of connection and state entities. A C++ library based on our OOC model is described that contains three class hierarchies, for data, state, and connection entities, respectively. In addition, an approach to constructing a protocol using this OOC model and the library is presented. An example in which a T.62 protocol is constructed shows that this approach provides a high level of modeling, concurrency, and reusability.

1. INTRODUCTION

Computers connected into a network interact according to a common set of rules known as a *protocol*, and different communication protocols may be needed by software applications using computer networks. One approach to developing a communication protocol is to use formal description techniques (FDT's), such as SDL [9], Estelle [7], and Lotos [6], to describe

INFORMATION SCIENCES **84**, 15–38 (1995)
© Elsevier Science Inc., 1995
655 Avenue of the Americas, New York, NY 10010

0020-0255/95/$9.50
SSDI 0020-0255(94)00072-J

specifications and to translate the specifications into programs. However, this approach has two shortcomings: it requires the construction of a complex translator to handle translation from the specification language to the programming language [4], and existing FDT descriptions are generally not reusable. Object-oriented approaches to software developmet offer a high level of modularity, extensibility, and reusability. In this paper, we present an object-oriented concurrent (OOC) model for communication protocols that has the virtues of object-orientation and concurrency.

An OOC model consists of *active* and *passive* objects. An *active* object serves as a thread of control, while a *passive* object is similar to a regular C++ [12] object. An OOC model is composed of the following three types of objets: (1) *data entities*, which describe the structure of the communicated data units in a protocol, (2) *state entities*, which work on state transitions and the corresponding services, and (3) *connection entities*, which are responsible for manipulating communication services. The first two types of entities are passive, while the third type is active. We have constructed a class library of three class hierarchies, one corresponding to each of the above types of entities. In addition to classes and inheritance relationships, the library uses programming techniques such as *double dispatching* and *delegation* to simplify extension. Double dispatching is a way of choosing a service method for a message based on the classes of the receiver and the first argument in the message. Delegation provides the power to inherit states as well as behavior dynamically [8]. In this paper, we describe the construcion of a library for C++ [12] using a lightweight process library [11]. Our library provides reusable components for implementing different protocols. In addition, a list of guidelines for developing a protocol with the OOC model and the library are also presented. The T.62 protocol [2] is used to illustrate reuse. This example shows that our library provides a high level of reusability, including modeling, extendibility, and concurrency.

The remainder of this paper is organized as follows. Section 2 presents the OOC model and double dispatching and delegation techniques. Section 3 explains how object-orientation and concurrency are integrated in our approach. Section 4 describes the class library, and Section 5 presents the development guidelines. Section 6 concludes the paper.

2. OOC MODEL FOR PROTOCOLS

2.1. AN OOC MODEL

In an OOC model, a communication protocol consists of data, state, and connection entities. The data entities represent the packed communication data. The state entities, including state and event objects, interpret the

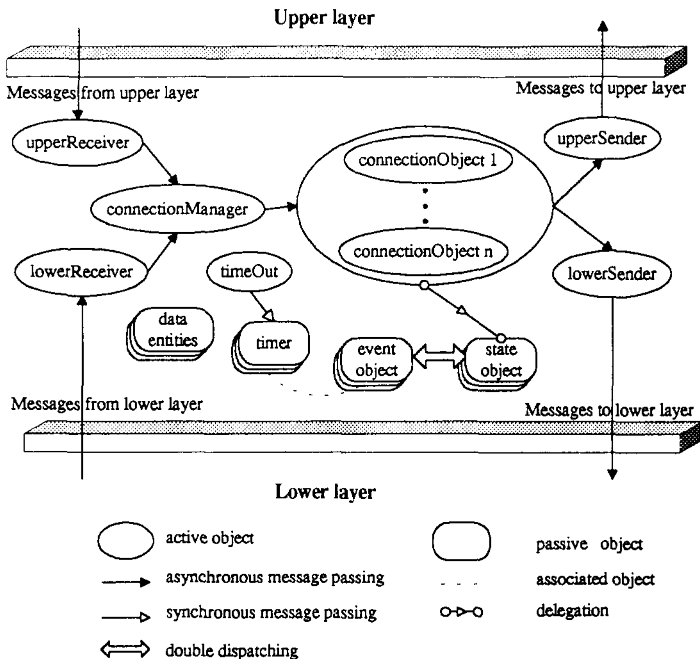


Fig. 1. OOC model for communication protocols.

mapped service primitives of state transitions. The connection entities are responsible for manipulating communication services. Each of the entities is specified as an object. Figure 1 shows the architecture of a protocol in which the connection entities, the active objects, are expressed by ellipses, while the data and state entities, the passive objects, are represented by rounded rectangles.

In a layered architecture, a layer of a communication protocol can be described with our model as follows. The *upperReceiver* and *lowerReceiver* objects receive input messages from the upper and lower layers, respectively, whereas the *upperSender* and *lowerSender* objects send messages to the upper and lower layers, respectively. The *connectionManager* object takes over all the communication channels, each of which is handled by a *connectionObject* object. A *connectionObject* object cooperates with state objects and event objects to perform communication services. The *timeOut* object monitors the timing constraints on the communication. The execution flows in a protocol are indicated by the arrows in Figure 1.

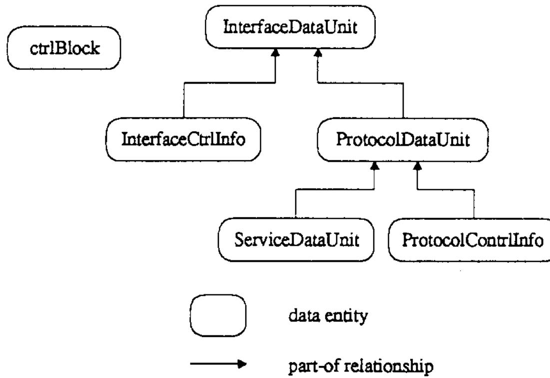


Fig. 2. Relationship between data entities.

2.2. *DATA AND STATE ENTITIES*

Data entities, such as *Service Data Units (SDU's), Protocol Control Information (PCI), Protocol Data Units (PDU's), Interface Control Information (ICI), Interface Data Units (IDU's)*, and *Control Blocks (CB's)*, represent the transferred and control information. State entities include state, event and timer objects. These describe the rules of state transitions and specify the timing constraints on an event. These two types of entities are passive objects.

Data entities are passive objects. An *SDU* object represents user data, and a *PCI* object represents a header attached to user data to identify the data to be transferred. A *PDU* object is composed of a *PCI* object and a *SDU* object. A *ICI* object records the interface information of the service primitive which is to be invoked in the next layer. An *IDU* object, which is composed of an *ICI* object and a *PDU* object, is an encapsulated message passed to the adjacent layer. On the other hand, a *CB* object contains information specifying the handling rules of each connection. The relationships between data entities are depicted in Figure 2.

A protocol machine can be treated as a finite state machine with a state transition diagram such as that shown in Figure 3. In the figure, there are three states, *ready, established*, and *close*, and four kinds of events, *request, confirm, indication*, and *response*. The finite state machine can be implemented by using a table-driven approach or a procedure-driven approach. A table-driven approach uses a two-dimensional array in which a row element represents a state and a column represents a distinct input event. A procedure-driven approach treats an event as a signal to trigger the current state. With these two approaches, it is difficult to extend a finite

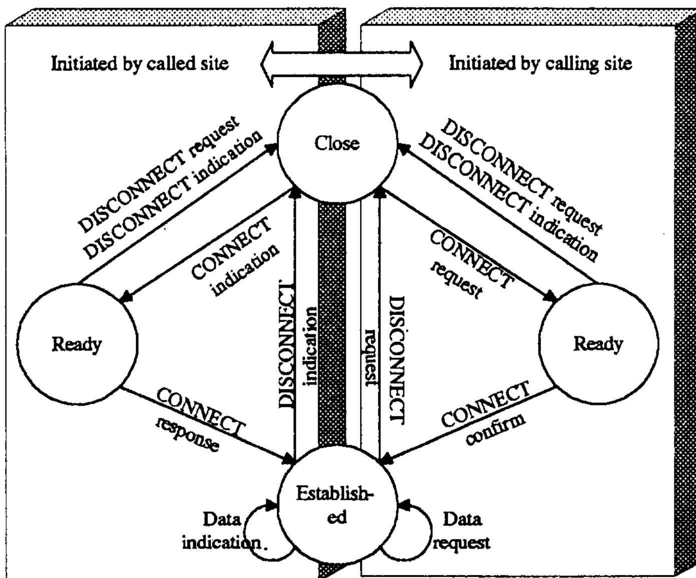


Fig. 3. State transition diagram of a protocol machine.

state machine for a protocol, because input events and changed states are tightly coupled.

We employ state and event objects to implement a finite state machine for a protocol. A state object represents one state of a protocol machine, and an event object represents the service that is requested. For timing constraints, an event object is associated with a timer object specifying a period of time. State objects, collaborating with event objects, perform state transitions by means of the double dispatching technique (described in a later subsection. The relationships between state entities are shown in Figure 4.

2.3. CONNECTION ETITIES

From the viewpoint of protocols, each communication service is manipulated through the following three steps: 1) messages are received from the upper or lower layer, 2) messages are processed according to the rules of the current layer, and 3) messages are sent to the upper or lower layer. According to these steps, a protocol entity can generally be divided into three parts: receiving, sending, and message processing. These parts can execute concurrently. In order to increase concurrency and modularity, these parts

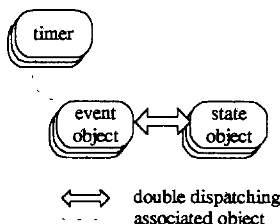


Fig. 4. Relationship between state entities.

can be decomposed into finer-grain entities, each of which is specified as an active object.

The receiving and the sending parts are refined individually to be *upperReceiver, lowerReceiver, upperSender*, and *lowerSender* objects in order to make message exchanging between layers more efficient. The message processing part establishes a connection to process incoming messages. A number of connections are allowed to exist at the same time, and each can be served by a distinct *connectionObject* object. On the other hand, a *connectionObject* object serves connections one by one: Each time it is in charge of one connection only. The object manipulates all the messages in the connection until the connection ends. All *connectionObject* objects are managed by a *connectionManager* object. The connection entities, including *upperReceiver, upperSender, connectionManager, connectionObject, lowerReceiver*, and *lowerSender* objects, are all active objects. In other words, each of them has its own thread of control.

An *upperReceiver* object and a *lowerReceiver* object are responsible for receiving messages from the upper layer and lower layer, respectively, and for checking the legality of the messages. These objects deliver incoming messages to a *connectionManager* object, if the messages are legal, In addition, an *upperReceiver* object handles all requesting and responding events, and a *lowerReceiver* object handles all indicating and confirming events.

A *connectionManager* object manages all *connectionObject* objects by creating and removing them. After receiving messages from an *upperReceiver* or a *lowerReceiver* object, a *connectionManager* object determines the connection to which the messages belong. If the connection does not exist, the *connectionManager* object will create a new *connectionObject* object to handle the messages. The messages are forwarded to their corresponding *connectionObject* objects by a *connectionManager* object, as shown in Figure 5.

An *upperSender* object and a *lowerSender* object are responsible for sending messages to the next upper layer and lower layer, respectively.

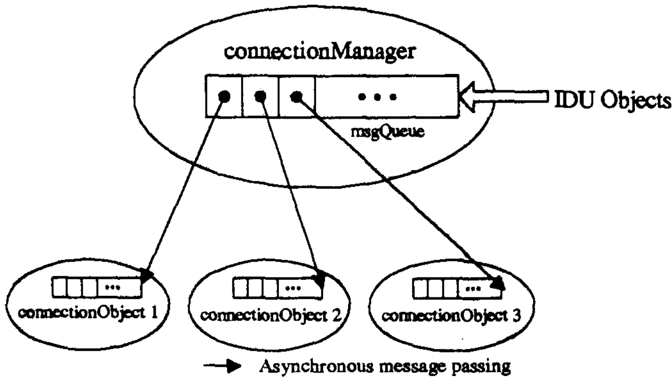


Fig. 5. The *connectionManager* and *connectionObject* objects.

These objects receive messages from a *connectionObject* object, encode and then pass the messages to the next upper or lower layer. An *upperSender* object is responsible for sending all indication and confirmation events, and a *lowerSender* object sends all request and response events.

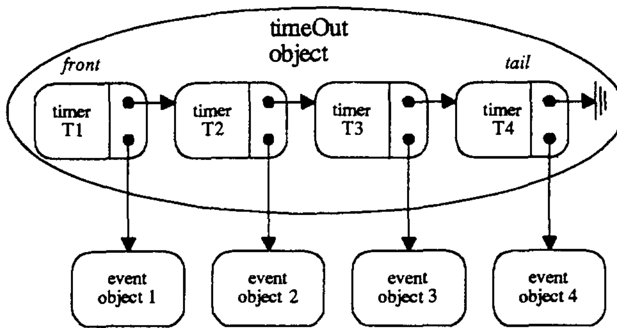
A *connectionObject* object performs the state transitions of a protocol machine by collaborating with state and event objects. A *connectionObject* object changes its behavior based on its current state and an input event. When a *connectionObjct* object is created, it is given its state according to an input event. After changing into another state, the object invokes the service function of the state object to process the messages. Because a *connectionObject* object and a state object are distinct entities without shared data, the delegation technique (which will be described later) is applied to these two objects.

2.4. HANDLING ABNORMAL CASES

Abnormal cases include sending expedited data, responding to internal errors, and handling time-outs. These abnormal cases can be dealt with by *msgQueue, timer*, and *timeOut* objects.

A FIFO queue is implemented in typical asynchronous message passing schemes, but the queue alone is unable to process messages in a special order. Instead, a *msgQueue* object, an object with a priority queue, is used to handle expedited data and internal errors to which a high priority is assigned. An active object with a *msgQueue* object can process high-priority data first.

A *timer* object records the time limit of an event object. To manage timing constraints efficiently, *timer* objects are organized as a linked list



Time-out errors occur at T1, T1+T2, T1+T2+T3, and T1+T2+T3+T4 time units in event object 1, event object 2, event object 3, and event object 4, respectively.

Fig. 6. List of timer objects and associated event objects.

sorted by the time limit (see Figure 6). A *timeOut* object is an independent thread of control responsible for detecting time-out situations. Because the time limit of a *timer* object is relative, a *timeOut* object only needs to decrease the amount in each *timer* object on the list. When the time limit of the first *timer* object reaches zero, the associated event object has timed out. A corresponding time-out handling routine is then invoked. The time-out handling routines may abort or reestablish a connection, depending on the specification of a protocol.

2.5. DOUBLE DISPATCHING IN STATE ENTITIES

The state transitions of a protocol machine are determined not only by the input event, but also by the current state. The double dispatching technique allows matching of the service method to be performed based on the classes of the receiver and the first argument object in the message. The double dispatching technique is applied in state entities to control the state transitions.

In an OOC model, state and event objects describe a protocol machine. A state object, such as a *ready, established,* or *close* object, represents the state of a protocol during communication. An event object, such as a *request, indication, response,* or *confirm* object, is the desired service decoded from an *ICI* object. A state object has to specify an action for each possible event. The relationships between state and event objects are shown in Figure 7. List 1 shows the implementation of state transitions in the traditional approach. In List 1, *Ready* is a state object with method *doIt( )* to specify the actions for events. This programming style reduces

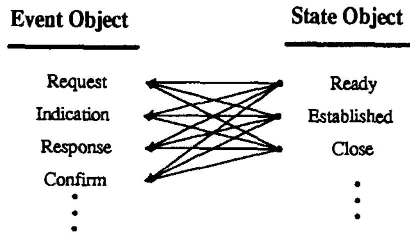


Fig. 7. Mapping between event and state objects.

```
Ready::doIt(eventType  *eventObj)  {
     if (eventObj->isRequest()){
          /* code for request event */
     };
     if (eventObj->isIndication()){
          /* code for indication event */
     };
     if (eventObj->isResponse()){
          /* code for response event */
     }; ...
}
```

List 1. Traditional approach to state and event objects.

the reusability of these codes, since the method *Ready::doIt( )* has to be redefined when an event object is changed.

The double dispatching mechanism, also called *multiple polymorphism* [5], can improve the reusability of the codes. Current object-oriented programming languages, such as C++, do not support this mechanism. However, we can specify a statement with dynamic binding and object invocation to perform the same function as this mechanism. An example of such a statement is *curEvent* $\rightarrow$ *doIt(curState)* in Figure 8, where *curEvent* and

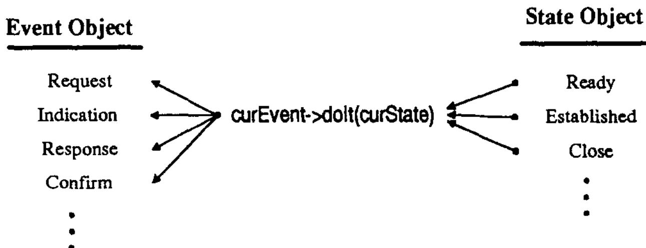


Fig. 8. Applying double dispatching to event and state objects.

```
inline void Request::doIt(stateMgr *curState)  {
  curState->Request(...); // relay function
};
inline void Indication::doIt(stateMgr *curState)  {
  curState->conIndication(...);  // relay function
};
inline void Response::doIt(stateMgr *curState)  {
  curState->conResponse(...);   // relay function
};
inline void Confirm::doIt(stateMgr *curState)  {
  curState->conConfirm(...);   // relay function
};
---------------------------------------------------------
void Ready::Request(...)  {
  // the action for Request
};
void Ready::Indication(...)  {
  // the action for conIndication
};
void Ready::Response(...)  {
  // the action for conResponse
};
void Ready::Confirm(...)  {
  // the action for conConfirm
};
```

List 2. Double dispatching using C++ code.

curState are dynamically bound to an event object and a state object, respectively, and the method *doIt( )* selects an action according to the bound state object. For example, *curState* is bound to a state object *Ready* and *curEvent* is bound to an event object *Request*. Both bindings occur before the statement *curEvent* → *doIt(curState)* is executed. When the event object *Request* is invoked, its method *doIt(curState)* selects an appropriate function according to the passed argument object *Ready*. In other words, the function *Ready::Request*( ) is selected for execution. As illustrated in List 2, state objects and event objects can be developed independently.

2.6. DELEGATION IN CONNECTION AND STATE ENTITIES

Delegation is often regarded as a language feature used to replace inheritance [10]; it may be regarded as a relationship between objects. With delegation, a delegator object can commission another object to perform a desired service for the delegator. We can separate the service definition from the connection entities via delegation.

In an OOC model, a communication service is performed by a *connectionObject* object cooperating with a state object. A state object is a service provider, and a *connectionObject* object manipulates all the messages in the connection. These two objects are different kinds of entities. On the other

```
    // class construction of connectionObject
    connectionObject {
        int data;
        ...
    // curEvent is bound to "Request" event object
    // curState is bound to "Ready" state object
    curEvent->doIt(curState, this);
    // delegating a message to the curState object
    /* connectionObject forwards a message to the current state
        object, and passes its object
        point as an  argument of the function doIt(...) */
};

// member function "doIt" of the class of an event object, Request.
inline void
    Request::doIt(stateMgr *curState, connectionObject *cObj)  {
        curState->eventAction(cObj,...);
    // forwarding a message to curState
}


// member function "Request" of the class of an state object,Ready.
void Ready::Request(connectionObject* cObj,...)  {
// accessing the data of connectionObject via cObj passed from the
// extra argument
    cObj->data; // access the data of the original object
    ...    // the code for the eventAction
}
```

List 3. Delegation in C++

hand, the delegation technique allows a *connectionObject* object to change its behavior according to its state object.

If a delegation mechanism is not provided in a language, as is the case with C++, then delegation can be implemented using a programming technique which appends an original receiver as an extra argument to each delegated message. List 3 shows such an implementation example. In this example, the *connectionObject* object passes its object reference, *this*, to an event object in the statement *curEvent→doIt(curState, this)*, which performs the double dispatching described in the previous subjection. In List 3, *curEvent* is bound to the *Request* event object and *curState* is bound to the *Ready* state object. Then the event object, *Request*, receives the reference of a *connectionObject* object, *cObj*, and passes it to the state object, *Ready*. Finally, the state object, *Ready*, receives the reference of the *connectionObject* object, and performs the service function, *Request*, on the data of the *connectionObject* object.

3. CONCURRENCY IN THE OOC MODEL

Most object-oriented languages are strong on reusability but weak on concurrency [14]. It is difficult to apply concurrency to object-oriented languages, because of the conflict between exclusive synchronization and inheritance [1]. Hence, we do not permit exclusive synchronization in the communication protocols using our approach.

An object is a well-defined entity that encapsulates data with a set of operations. When processing an input message, an object invokes its corresponding operation(s) to perform a service. With *classification* and *inheritance* (classes), an object is instantiated from a class, and a class may inherit or be inherited by another class. There are two kinds of objects in our OOC: active and passive objects. A active object encompasses its own thread of control, whereas a passive object does not. An object is either active or passive, neither both nor switchable. An active object is generally autonomous: It is associated with an (implicit) queue to buffer input messages. After being created, an active object has its own thread until the object reaches the end of the thread (or is killed). A passive object, on the other hand, executes its routine only when it receives a message, i.e., when a thread of control enters implicitly. It returns the service's result and the thread of control after completing the service.

An interaction between two objects occurs when one object requests the other's service by sending it a message. There are two kinds of message passings between objects: synchronous and synchronous. In synchronous message passings, a sender blocks until it gets a reply from the receiver. In asynchronous message passings, a sender continues its execution right after passing a message. In other words, the sender of a synchronous message passes its thread to the receiver implicitly, while the sender of an asynchronous message does not. Synchronous messages are sent to passive objects only, while asynchronous messages are sent to active objects only. Both active and passive objects are able to issue both synchronous and asynchronous messages.

Obviously, parallelism is integrated into the OOC model by means of active objects. Having these active objects execute simultaneously is analogous to having several processes run concurrently. The active objects in C++ language can be implemented with a task library.[1]

It is not necessary to consider the exclusive synchronization of active objects, since they have their own thread and receive asynchronous messages only. Data inconsistency may occur in a passive object when concurrent accesses to it occur.

[1]One such task library is the lightweight process library provided in *Objectkit\C++*, which is a product of ParcPlace Systems, Inc.

In our OOC model, the concurrent executing parts of a communication protocol are the connection entities, since they are modeled as active objects. The passive objects include data and state entities. The data entities are always treated as asynchronous messages passed during the execution flow of a protocol. Therefore, it is not necessary to apply exclusive synchronization to data entities, since they are processed in a single active object at one time. The state entities are shared by *connectionObject* objects, that is, a passive object may be concurrently accessed by several active objects. In order to avoid data inconsistency and eliminate additional synchronization protection, the state entities are defined so as to include no data shared between *connectionObject* objects: Each state entity performs operation(s) on the data of a *connectionObject* object via the delegation technique, but does not retain any information shared by *connectionObject* objects.

4. THE CLASS LIBRARY FOR PROTOCOLS

This section discusses the class hierarchies abstracted from the entities of the OOC model. A class definition can be divided into two parts: interface and body definitions. A class with an interface definition only is an *abstract* class. It is usually used as a base class. A class with both interface and body definitions is called a *concrete* class. The notation used in the rest of this paper, such as classes, objects, and inheritance and whole-part structure symbols, is adopted from [3].

4.1. CLASS HIERARCHY FOR DATA ENTITIES

The class hierarchy designed for data entities specifies the classes for *SDU, PCI, PDU, ICI, IDU*, and *CB* objects. The classes designed include `ServiceDataUnit`, `ProtocolCtrlInfo`, `ProtocolDataUnit`, `InterfaceCtrlInfo`, `InterfaceDataUnit`, and `CtrlBlock`. Common features are abstracted as the base class `InternalObj`. Classes `ServiceDataUnit`, `ProtocolCtrlInfo`, `InterfaceCtrlInfo`, and `CtrlBlock` can be specified as concrete classes when developing a new communication protocol. The class hierarchy is shown in Figure 9.

In the class hierarchy, class `ProtocolDataUnit` contains a service data unit and protocol control information, instances of classes `ServiceDataUnit` and `ProtocolCtrlInfo`, respectively. A `ServiceDataUnit` object expresses transmitted user data, containing a data buffer and the size. It also provides the methods retrieving/sending the user data from/to. A `protocolCtrlInfo` object represents a header or tailor, which is the information exchanged by peer layers at different sites of network. Class `Protocol-`

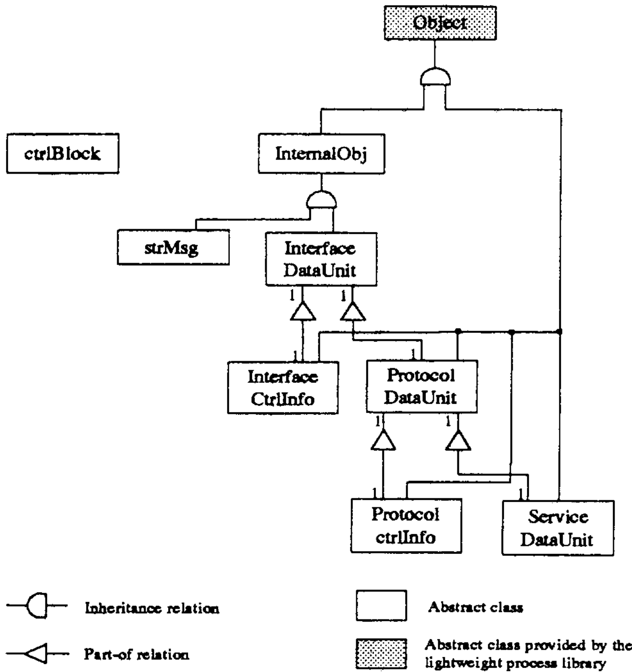


Fig. 9. The class hierarchy of data entities.

CtrlInfo, a base class for extension, includes coded data of protocol control information, a service type, a source address, and a destination address, and the oeprations on them.

Class InterfaceDataUnit contains an instance of class InterfaceCtrlInfo and an instance of class ProtocolDataUnit. It has priority by inheriting class InternalObj. A ProtocolDataUnit object represents the data transferred across layer boundaries. The methods encode () and decode () are responsible for data conversion, where encode () converts the object's internal data to the form of a string and decode () converts the data back into its original form.

Class InterfaceCtrlInfo (Interface Control Information) is a base class which contains the temporary data passed between adjacent layers to invoke a service function. It contains socked pointer, an index of connectionObjTable, and an event object corresponding to the service type in class ProtocolCtrlInfo. Class CtrlBlock is an empty base class to be extended when developing a specific protocol. An instance of its subclass represents all the control information about the state transitions of a connection.

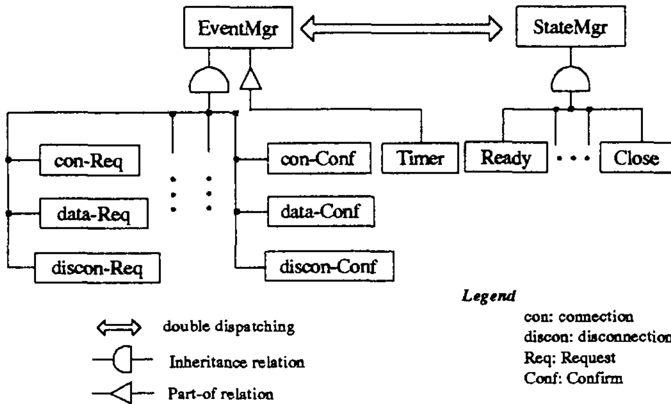


Fig. 10. The class hierarchy of state entities.

The classes described above are extracted directly from a protocol specification. For implementation, several additional classes, such as `StringMsg` *(String Message)*, `AddrTable` *(Address Table)*, and `connectionObjectTab` *(Connection Object Table)*, are needed. A `StringMsg` object is an encoded form of an `InterfaceDataUnit` object and is passed between adjacent layers. It has the priority by inheriting class `InternalObj`. An `AddressTable` object represents an address table based on caller addresses. Each entry points to a related index of a `connectionObjTable` object which stores each connection object. Each connection can be identified by the address of the caller. Under asynchronous message passing, this message passing scheme is not enough to express a message with priority. One way to resolve this problem is to use a `msgQueue`, a priority queue. The methods `dequeue ()` retrieving an element, `dequeue ()` appending an element, and, `putback ()` adding an element according to its priority are provided in class `msgQueue`.

4.2. CLASS HIERARCHY FOR STATE ENTITIES

This class hierarchy contains the classes of state entities. The classes `stateMgr`, `eventMgr`, and `timer` are defined for state, event, and timer objects. Becuase of the relationship of double dispatching between state objects and event object, classes `stateMgr` and `eventMgr` have to be specified as base classes of state and event classes, respectively. The class hierarchy for state entities is shown in Figure 10.

Class `StateMgr` is a base class for state classes. Each state class concerns one state of a protocol specification. The method of `stateMgr` delegates

its operation to the first argument, a connectionObject object. Class eventMgr is a base class for event classes. It contains a timer object, a method doIt (), a time-out handling method timeExpired (), and an error handling method errorHandler (). The timer object represents a specified time period during which this associated event must be performed. The method doIt () is the relay function of the double dispatching technique. Both time-out handling and error handling methods are virtual member functions which must be refined in each subclass. Class timer is the class of timer objects and contains three methods, stop (), reset (), and expired (). The method stop () is invoked when this event object has been completed successfully. The method reset () is invoked when an event object is re-transferred. The method expired () is invoked when a time period has elapsed.

4.3. CLASS HIERARCHY FOR CONNECTION ENTITIES

The class hierarchy for connection entities specfies classes such as upperReceiver, lowerReceiver, upperSender, lowerSender, connectionObject, connectionManager, and timeOut. These classes are defined for *upperReceiver, lowerReceiver, upperSender, lowerSender, connectionObject, connectionManager*, and *timeOut* objects, respectvely. All of these classes are subclasses of class task, from which they inherit the property of being active objects. In addition, several base classes, Protocol, Receiver, and Sender, are introduced to develop these classes. The class hierarchy is depicted in Figure 11, where the classes task (representing a lightweight process in this library) and Protocol are the root (base) classes.

Class Protocol is an abstract class used as the root class for active objects except for the class Scheduler, and contains three objects, instances of class AddrTab, class ConnectionObjectTab, and class Scheduler, and two methods, put2Scheduler () and Scheduling (). The AddrTab object is used for an address table and the ConnectionObjectTab object is for a connection object table. In addition, the metod put2Scheduler () adds an active object to the scheduling queue by its priority, and the method Scheduling () checks which active object can run next. A Scheduler object is used to handle the scheduling of connection entities.

In class Receiver, method eventMapping () converts input requests to the corresponding event objects, method checkIn () determines the entry of the associated connectionObjTable object according to an input message, and method decode () converts an input string into an associated InterfaceDataUnit object. Classes upperReceiver and lowerReceiver are subclasses of class Receiver. An upperReceiver object is responsible

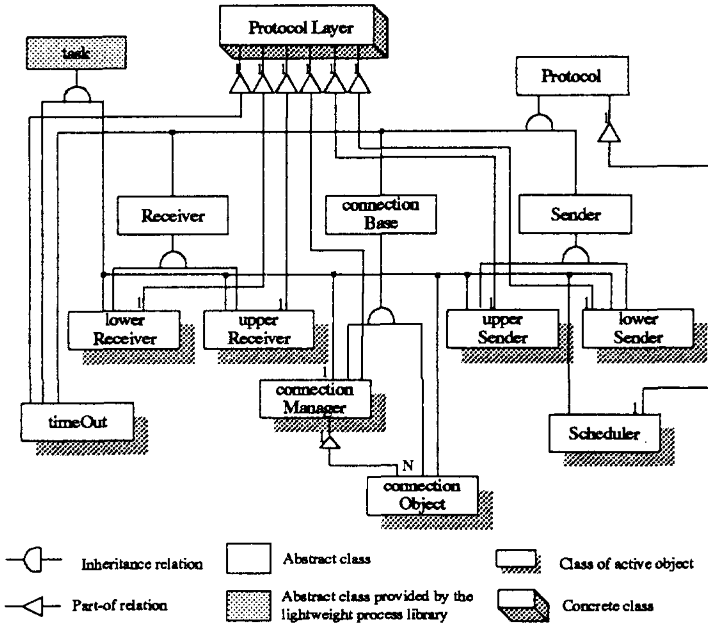


Fig. 11. The class hierarchy for connection entities.

for requesting and confirming events. A `lowerReceiver` object is responsible for indicating and responding events. Both of these send messages to a `connectionManager` object. In class `Sender`, method `encode ()` converts an `InterfaceDataUnit` object into the associated string. Classes `upperSender` and `lowerSender`, subclasses of `Sender`, deal with the messages from a `connectionObject` object and then deliver appropriate message(s) to the `lowerReceiver` and `upperReceiver` objects of the adjacent lower and upper layers.

A `connectionManager` object is a dispatcher for all connections; its method `getObjID ()` determines the `connectionObject` object to which an input message is sent. A `connectionObject` object contains a state record, a control block for handling rules, a data unit (*IDU* object), and timing constraints. Such an independent active object makes dynamic adjustments (through setting priority) possible. In addition, a `connectionObject` object delegates the message of a state transition, received from the `connectionManager`, to the current state object.

A `timeOut` object is responsible for monitoring a set of timed objects, which are `timer` objects associated with a specific event object. To make

the management efficient, `timer` objects are stored in a linked list in order of expiration time. The timing of a `timer` object in the list is relative to that of the previous object, while the timing of the first is relative to the current time. When the timing in the first `timer` object reaches zero, the associated event object times out, and the corresponding time-out handle method `timeExpired ()` is invoked.

5. DEVELOPING A PROTOCOL WITH THE OOC MODEL

5.1. A REUSE APPROACH WITH THE OOC MODEL

The specification of a protocol generally comprises two sets of documents: a service definition document and a protocol specification document. The service definition document contains a specification of services provided by the protocol to its upper layer. The protocol specification document contains a precise definition of protocol data units and a precise definition of the operations of the protocol. Furthermore, it is necessary to consider the temporal order ad run-time behavior (e.g., the order of a sequence of service requests and the maximum number of connections allowed at one time) during the implementation phase. In the OOC model, the definitions of protocol data units can be specified as data entities. Each of the services provided can be specified as an event object, and the oeprations can be described in state objects. The temporal order and run-time behavior can be defined in connection entities. After being interpreted in terms of data, state, and connection entities, a protocol specification can be developed by the approach presented in the following subsection.

The class hierarchies in Section 4 can be constructed easily. Because the OOC model is used, the components in the library can be reused directly to develop different protocols. Here we present an object-oriented approach to constructing a protocol with an OOC model and the library. In the analysis phase, the requirements of the protocol are specified with our model (and additional constraints). In the design phase, the specfied entities are mapped to the classes in the library. Specifically, the classes `ServiceDataUnit`, `ProtocolDataUnit`, `InterfaceCtrlnfo`, and `InterfaceDataUnit` provided by our library can be applied for *SDU*, *PDU*, *ICI*, and *IDU* entities directly. In addition, classes `connectionObject` and `ConnectionManager` can be reused directly. The rest of the classes employed during the design (and implementation) of a protocol can be created according to the following steps, where steps 1 and 2 are for data entities, steps 3 and 4 are for sate entities, and step 5 is for connection entities.

1. Idetify the *PCI* entity and construct a class for this object based on the class `ProtocolCtrlInfo`.

2. Identify the object that Handles the procedural rules for state transitions. Construct a class for this object based on the class `ctrlBlock`.
3. Identify all service primitives in the protocol. Construct a class for each service primitive (event), a subclass of class `eventMgr`, wih its own functionality.
4. Identify all states in the protocol. Construct a class for each state object, a subclass of class `stateMgr`, and define its responsibility by overriding or augmenting member functions.
5. Identify all possible input requests (events) for *lowerReceiver* and *UpperReceiver* entities, respectively. Construct two classes for each of these two entities, respectively, by (1) inheriting classes `upperReceiver` and `lowerReceiver` and by (2) overriding the virtual function `Receiver::eventMapping ()` with double dispatching.

5.2. AN EXAMPLE

We here use CCITT recommendation T.62 [2] as an example to illustrate the reuse of our library. This recommendation defines the end-to-end procedures to be used within the Teletex and Group 4 facsimile services and concerns the end-to-end control procedures that are network-independent. By applying the guidelines in the design phase, we can develop an appropriate protocol. The following is a brief description of our development of T.62 after modeling it.

Step 1: The class `t62PCI` is extended from the base class `ProtocolCtrlInfo` to describe the T.62 protocol control information with additional data members.

```
class t62PCI : public ProtocolCtrlInfo   {
public:
  unsigned int   transfer_time;     //reliable transfer mode II
  unsigned int   document_reference_information;
       // reliable transfer mode I
  unsigned int   synchronization_point;
  int checkpoint_mechanism;
       // map onto session using Mechanism 2
  unsigned int    token_priority;
       // D_TOKEN_PLEASE service parameters
  char   service_ID;
  char   document_reference_number;    // S_ACIVITY_START
  char   document_type_ID;
  char   checkpoint_reference_number; // S_ACIVITY_END
  unsigned int  checkpoint_serial_number;
  void    show ();     // overriding
```

```
  void   t62PCI::t62PCIDefault ();   //default setting
          t62PCI(int, char*, char*, char**=NULL);   //initialize
};
```

Step 2: The class `t62ctrlBlk` is derived from the base class `ctrlBlock` by adding the elements which are specified in the T.62 protocol.

```
class t62ctrlBlk :  public ctrlblk   {
public:
   t62ctrlBlk (); //constructor
 int S;// The next expected checkpoint reference number
 int R;// the next allowed expected checkpoint reference number
 int P;// the next expected checkpoint reference number to be
       // acknowledged
 int Q;// the next allowed expected checkpoint reference number
       // to be acknowledged
 int I;// an actual checkpoint reference number in CDPB or CDE
 int K;// an actual checkpoint reference number in RDPBP or RDEP
 int C;// a checkpoint reference number from which the source
       // will resume transmission
 int W;// acknowledgment window size
};
```

Step 3: These services in T.62 include *S-CONNECT, S-RELEASE, S-U-ABORT, S-DATA, S-TOKEN-PLEASE, S-SYNC-MINOR, S-ACTIVITY-START, S-CONTROL-GIVE*, and so on. Each communication service is performed through a sequence of service primitives such as request, indication, response, or confirm. Every service primitive is regarded as an event object, an instance of (newly defined) event classes inheriting class *eventMgr* but overriding the method. The abstract definitions of all event objects are alike. One such definition is shown below.

```
class tokenPlsResp:public eventMgr   {
public:
tokenPlsResp (int time =-1):  eventMgr(``tokenPleaseResponse'',
  time)
{}
inline void doIt (connectionObject*); //relay function
void timeExpired ();        //handling time out
void errorHandler();        //handling error
};
```

Step 4: The state transition diagram in T.62 includes fourteen states *state0–14*. Each state is regarded as an instance of one of the state classes inheriting class `stateMgr` but overriding several related service primitives. Each extended state class can be also used as a base class if this state contains general properties of a set of substates (such as *state2*). Some state classes are shown below.

```
class state0_1:public stateMgr   {
public:
   state0_1() :  stateMgr (``sate0_1'');
 void conReq(connectionObject*);     //overriding service
primitive
 void conInd(connectionObject*);
 void conResp(connectionObject*);
 void conConf(connectionObject*);
 void dataReq(connectionObject*);
 void dataInd(connectionObject*);
 void discReq(connectionObject*);
 void discInd(connectionObject*);
};
class state1_1 :  public stateMgr   {
public:
 state1_1() :  stateMgr (``state1_1'');
 void conConf(connectionObject*);// overriding service primitive
 void conResp(connectionObject*);
};
class state2 :  public stateMgr   {
public:
 state2(char *name):stateMgr(name);
 void discInd(connectionObject*);// overriding service primitive
 void ctrlGiveInd(connectionObject*);
 void actIntReq(connectionObject*);
 void actDcadReq(connectionObject*);
};
class state2_1_1 :  public state2   {
public:
 state2_1_1() :  state2 (``state2_1_1'');
 void actBegContInd(conectionObject*);
       //overriding service primitive
 void actBegStrtInd(conectionObject*);
```

```
 void capabDataInd(conectionObject*);
 void actDcadReq(conectionObject*);
};
```

The defiitions of the service primitives depend on the protocol specification. The definition of one service primitive is shown as follows.

```
//primitive S-CON-REQ
void state0_1::conReq(connectionObject* cObj)    {
      //cObj stands for the related connectionObject, that is
      //the delegator of this message
cout << ''\nL5:  Session Connect request'' << endl;

      //start the timing constraint associated to this event
cObj→getCurEvent ()→getTimeObject ()→reset\
      (cObj→getCurEvent (), 1);
      //deal with something related to this service primitive
cObj→setCurrentState(&state81);    // change state
cObj→getLS()→put(cObj→getIDU()); // put data into lowerSender
cout << ''in state0_1 to state8_1'' << endl;
}
```

Step 5: The event types bound to an associated event object can be specified in the overriding `Receiver::eventMapping ()` method. The event types are the session commands and responses, such as command session start (CSS), response session start positive (RSSP), command session end (CSE), response session end positive (RSEP), response document re-synchronize positive (RDRP), and so on. The following definition is the method `lowerReceiver::eventMapping ()`. The `upprReceiver-::eventMapping ()` can be defined analogously by replacing the *Indication* with *Request* and *Confirm* with *Response.*

```
eventMgr* lowerReceiver::eventMappng(unsigned char type)    {
        eventMgr *event;
        switch(type)    {
          case CSS:
            event = &conIndication; break;
          case RSSP:
            event = &conConfirm; break;
          case CSE:
            event = &discIndication; break;
```

```
            case RSEP:
                event = &discConfirm; break;
            case RDRP:
                event = &actIntConfirm; break;
            ...
            default:
                event = NULL;
        };
        return event;
};
```

6. CONCLUSION

This paper presents an object-oriented concurrent model to facilitate the development of communication protocols. The model consists of active objects, which accomplish services provided by a protocol, and passive objects, which represent data units and state transition diagrams. Double dispatching and delegation are applied to increase the extensibility of the protocol software. We have constructed a reusable library including a number of general classes, the entities abstracted in our OOC model. Our experience shows that this model is well suited to developing communication protocols since 1) its OO modeling allows communication protocols to be modeled naturally and intuitively; 2) its OO library provides good extensibility and flexibility for resue; and 3) it facilitates implicit concurrency of communication protocols. Planned enhancements to the library include developing several specific classes for the needs of each specific layer and expanding the library to support verification and validation of communication protocols.

REFERENCES

1. C. Atkinson, *Object-Oriented Reuse, Concurrency and Distribution*, Addison-Wesley, Reading, MA, 1991.
2. CCITT Blue Book volume VII-Fascicle VII.3, Control procedures for the Teltex and Group 4 facsimile services. *CCITT Recommendation T.62*, 1988.
3. P. Coad and E. Yourdon, *Object-Oriented Analysis*, second ed., Prentice-Hall, Englewood Cliffs, NJ, 1991.
4. S.-J. Hsiao and F.-J. Wang, A framework and its class hierarchy for communication protocols, In *Proceedings of National Computer Symposium*, Vol. 1, Taiwan, R.O.C., 1991, pp. 294–299.
5. D. H. H. Ingalls, A simple technique for handling multiple polymorphism, *OOPSLA'86 Conference Proceedings*, ACM, 1986, pp. 347–349.

6. ISO-IEC/JTC1/CS21/WG1/FDT/C, Lotus, a formal description technique based on the temporal ordering of observation behavior, *ISO International Standard IS8807*, Feb. 1989.
7. ISO-IEC/JTC1/CS21/WG1/FDT/B, Estelle, a formal description technique based on extended state transition model, ISO International Standard IS8807, July 1989.
8. R. E. Johnson and J. M. Zweig, Delegation in C++, *J. Object-Oriented Programming*, pp. 31–34, Nov./Dec. 1991.
9. A. Rockstrom, *An Introduction to the CCITT SDL*, Televerket, Stockholm 1985.
10. L. A. Stein, Delegation is inheritance, *OOPSLA '87 Conference Proceedings*, ACM, 1987, pp. 138–146.
11. B. Stroustrup and J. E. Shopiro, A set of C++ classes for coroutine style programming, In *Proceedings of the USENIX C++ Workshop*, USENIX Association, 1987, pp. 417–439.
12. B. Stroustrup, *The C++ Programming Language*, second ed., Addison-Wesley, Reading, MA, 1991.
13. Y.-C. Ting, J.-L. Chen, and F.-J. Wang, An object-oriented concurrent model for communication protocols, in *Proceedings of International Conference on Telecommunications*, Dubai, Jan. 1994, pp. 155–158.
14. C. Tomlinson and M. Scheevel, Concurrent object-oriented programming language, in *Object-Oriented Concepts, Databases, and Applications*, ACM, 1989, pp. 79–124.

Received 1 December 1993; revised 1 April 1994