THE DESIGN OF AN OBJECT ORIENTED ARCHITECTURE

Yutaka ISHIKAWA and Mario TOKORO

Department of E. E., Keio University
3-14-1 Hiyoshi, Yokohama 223
JAPAN

ABSTRACT

This paper proposes a new object model, called the **distributed object model**, wherein the model is unified as a protection unit, as a method of data abstraction, and as a computational unit, so as to realize reliable, maintainable, and secure systems. An object oriented architecture called **ZOOM** is designed based on this object model. A software simulator and cross assembler for this architecture have been implemented. The feasibility and performance of the architecture are discussed according to program sizes and estimated hardware size and execution speed.

1. INTRODUCTION

Object oriented computation in the broad sense is computation described as a sequence of requests to objects through a single access method such as message passing. Models of object oriented computation have been widely investigated in the field of operating systems as a powerful mechanism for controlling access to shared data [Linden76, Fabry74, Wulf75, Wulf81, Jones80, Intel81, Kahn81], in the field of programming languages as a mechanism for data abstraction and/or as a computational unit [Jones76, Liskov79a, DOD80, Wulf76, Goldberg83], and in the field of artificial intelligence as a mechanism for the modular representation of knowledge [Minsky75, Bobrow76, Bobrow82, Goldstein80, Hewitt73, Theriault82].

The unification of these object models is required in order to realize reliable, maintainable, and secure systems. One of the authors has proposed one such model for objects [Tokoro82, Tokoro83]. The model has made clear the meaning of a name and an object in programming languages and has proposed a way for managing names and objects in a non-distributed computing environment.

As an extension of this basic model, in this paper we propose a new object model, called the distributed object model, for a distributed computing environment. It unifies the model both as a passive protection unit and as an active computation unit into one general unit. We define an object as a self-contained protected active entity. Communication between objects is performed by means of message passing. Thus, an object is a concurrent computation unit in a distributed environment.

An object oriented architecture, called ZOOM, has been designed based on this object model. That is to say, this architecture has been designed to show its superiority especially when it is used for constructing a distributed system. A software simulator has been implemented which runs under the VAX/Unix† system. Several programs have been written and simulated by this simulator. A preliminary design of its hardware implementation has been performed in order to estimate the hardware size and execution speed.

2. RECENT OBJECT MODELS

In this section we describe two object models and discuss several important issues in extending these object models to represent computation in distributed systems.

2.1. The Model of Objects as Encapsulated Data

This is a model for data protection. Abstract data type languages such as CLU [Liskov79a], ALPHARD [Wulf76], and Ada [DOD80] can be explained well by this model. Also, Cm*/StarOS [Jones80] and iAPX432 [Intel81, Kahn81] can be envisaged as computing systems that are based on this model.

In this model, an object is a protected set of data, or, more specifically, a protected region of a memory. A procedure can manipulate the set of data if and only if the procedure is defined by the type definition that characterizes the object. Thus, the object can be seen only through the defined procedures of the type. An object is called **concrete** when the protected set of data is being seen. It is called **abstract** when only the defined procedures are being seen. Let us call the entity of the set of the procedures of a type **class**. By using the message passing form, relations between a class and an object in this model can be characterized as follows:

(1) A class is an active entity, such as a process or a monitor. It consists of the procedures that define the operations of that class.

(2) A class creates an object of that class. An object is a protected set of passive data.

† Unix is a trademark of the Bell Laboratories.

0194-7111/84/0000/0178$01.00©1984 IEEE

(3) At the request of an operation, the operation name and object names which are the parameters for the operation are sent to the class (but not to any of the parametric objects).

(4) Each procedure of the class can manipulate the internal data (or own) of any object which was created by that class. It cannot manipulate the internal data of an object which was not created by that class. Instead, it sends a request for an operation to the class which created the object.

Since the main purpose of this model is to encapsulate data, an object is passive data. On the other hand, a class is an active entity. That is to say, an object is abstract data outside its class and concrete data inside its class.

For example, let us assume that class *MATRIX2* defines a 2-dimensional matrix (Fig. 1). The variables *a*, *b*, and *c* are of type *MATRIX2*. If the following statement is executed,

$$a := b + c ;$$

a message *add: b, c* is sent to class *MATRIX2* (Fig. 2)*. When class *MATRIX2* receives this message, the objects denoted by *b* and *c* are converted from its abstract form to its concrete form. The class procedure *add* can then manipulate them because both objects belong to class *MATRIX2*. A new object, which is the result of adding the matrices, is created in the concrete form, and is then converted to the abstract form. Finally, its name is sent back to the caller as a result of the operation. CLU provides **cvt** to change from abstract to concrete and from concrete to abstract.

The architecture designed along this model employs the concepts of capability-based addressing [Dennis66, Fabry74] and seal/unseal [Morris73] mechanisms. The capability based addressing mechanism provides a method for identifying an object with an authorized operation. The seal/unseal mechanism provides for the conversion between abstract and concrete data.

2.2. The Model of an Object as a Computational Entity

This model is almost the same as that of a process which does not have shared data. An object is an active entity such as a process. A class has a template for creating instance objects. Each created instance object of a class is also an active entity. Programming languages and computation models such as Simula [Simula67], Smalltalk [Goldberg83], LOOPS [Bobrow82], and the ACTOR model [Hewitt73, Theriault82] belong to this object model. Relations between a class and an object in this model are characterized as follows:

(1) A class is an active entity. A class provides the function of creating instance objects. It also provides shared resources which are used by its instance objects (in the case of Smalltalk).

* In the figures in this paper, an arrow indicates that the variable denotes the object. A boldface arrow indicates that the object sends a message to an instance or a class. A dotted arrow indicates that the object is created by the class.

```
program main is                    class MATRIX2 is
    a, b, c : MATRIX2;               own [
        .                                m11, m12,
        .                                m21, m22  : real;
                                       ]
    a := b + c;                    procedure add(a, b : MATRIX2)
        .                                    returns MATRIX2;
        .

end main;                          end MATRIX2;
```

Fig. 1 2-dimensional matrix

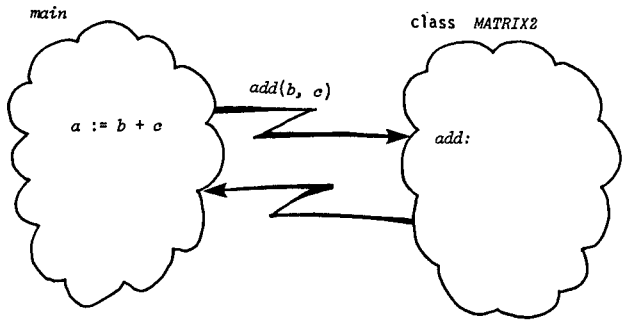

Fig. 2 An object as encapsulated data

(2) An instance object of a class is an active entity which is characterized by that class.

(3) At the request of an operation which is other than creation, the operation name and object names which are the parameters for the operation are sent to the instance object.

Let us consider the same example in order to compare this model with the one described above. As shown in Fig. 3, the main program sends a message *add: c* to the object denoted by *b*. When the object receives it, the object, in cooperation with the object denoted by *c*, creates a new object which is the result of adding matrices. Then the new object name is sent back to the caller.

In this model, an object is an active entity. Therefore the model has the potential for concurrent computation [Goldberg83, Theriault82].

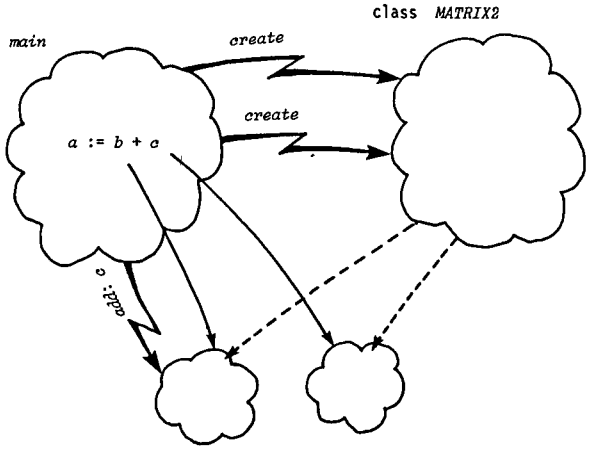

Fig. 3 An object as a computational entity

2.3. Issues

In order to extend these object models to describe distributed systems, issues including the following have to be considered:

(1) *Low parallelism*

In a distributed computation environment, it is desirable that a programmer be able to write a program as a natural mapping of the problem to the program and that the execution system automatically detects the parallelism of the problem and allocates portions of the program to its processors. In the model of objects as encapsulated data, instance objects cannot run concurrently, while classes can. This is because instances are protected passive data. In order to make an instance active for running concurrently with another, an additional concept for concurrent computation must be introduced. For example, Ada supports **task** which is dynamically created to run concurrently with other tasks, and the **guardian** constructor is introduced to manage processes and objects in CLU [Liskov79b].

In the model of an object as a computational unit, one object can run concurrently with another. In the ACTOR model, a synchronization mechanism can be constructed with a memory cell using the feature that a receiver takes messages in the order of their arrival [Greif75]. Yonezawa proposed specification and verification techniques for parallel computation [Yonezawa80]. Further discussion on synchronization and other implementation issues will be required to realize the architecture of this model.

(2) *Low protection capability*

In the model of an object as a protected data, the only access method to an object is specified by its type. In the model of an object as a computational unit, the acceptance of a request is determined by the content of the request message. In a distributed computation environment, the protection of an object should be controlled with respect to each accessing object by using the access list method rather than the capability list method.

(3) *Other issues*

Extending these models in order to describe distributed systems, we must consider the mechanism which decides where to create an object, the mechanism for inter-object communication beyond a processor, the mechanism for the migration of objects, the mechanism for controlling an object whose class is not on the same processor, and so forth.

3. PROPOSAL OF THE DISTRIBUTED OBJECT MODEL

In this chapter, we propose a new object model called the distributed object model as an extension of the basic object model [Tokoro82, Tokoro83]. This model provides the foundation for the design of distributed systems.

3.1. Objects

Objects are active entities which are **self-contained** and can execute concurrently with one another. Resources which are maintained by one object cannot be manipulated by other objects. When an object wants to operate on resources, the object has to send a request to the object that maintains those resources. Since each object consists of disjoint (non-shared) data and the procedures to manipulate the data, each object can run concurrently with others.

There are three kinds of objects: metaclass, class, and instance. The **metaclass** is the system defined object and is unique to the system. The metaclass provides for the creation of classes. Thus, a class is an instance of the metaclass.

A **class** is an object which consists of its class operations, class variables, and a template of instance variables and operations. The class operations include the creation of an instance. A class does not provide variables which are directly accessed by its instance objects. An **instance** is an object which consists of instance operations and its data.

3.2. Class and Instance Operation

The set of operations defined in a class consists of the creation of an object and the abstraction of relations between objects which belong to the class. A set of operations defined in an instance is the abstraction of the instance. This signifies the following: an operation which creates an instance of the class is defined as a class operation; an operation between two or more instances which belong to the same class can be defined as a class operation; and an operation which deals with one or more instances of different classes (or the same class) can be defined as an instance operation.

3.3. Inter-Object Communication

Communication between objects is achieved by sending/receiving a message. A **message** consists of the **name of the receiver**, a **request**, the **name of a sender**, and a **list of object names** as operands. A request and a list of object names in a message specify the request to be carried out in the receiving object.

3.4. Classes and Instances in a Distributed System

A class may reside in one or more processors. That is to say, the metaclass may create more than one **clone** of a class as different instances of the metaclass to run on different processors. Thus, a clone is a class object. A trivial example of such copies of a class is the class for integer with integer addition, subtraction, and so forth. User defined classes can also reside in more than one processor. It may be necessary for clones of the same class to communicate with one another.

A class (or a clone of a class) creates instances in its residing processor. An instance may **migrate** from one processor to another. In deciding the object's migration, processor failure, load balance, and the cost of communication are considered. The object's migration is realized as follows:

(1) A clone of the class of an object prepares the object in a special form, called the **frozen** object, which will be sent in a message.

(2) The clone sends the message to a clone of the class in the destination processor.

(3) The receiving clone of the class regenerates the object from the message.

3.5. Examples

Let us assume the following. As shown in Fig. 4, both class clones *C1* and *C2* reside in processor *P1*. Both class *C1* and *C3* reside in processor *P2*. Name *a* in an instance of class *C2* denotes instances of class *C1* and an instance of class *C3* in processor *P1*. Names *b*, *c* and *d* in an instance of class *C2* denote instance of class *C1* in processor *P2*. Assume the instance of class *C2* sends the following message to class *C1*.

request1: a, c.

Class *C1* resides in both processor *P1* and *P2*. Thus the message may be sent to the clone in either *P1* or *P2*. In this case we assume that the message is sent to the clone on processor *P1*. This message requests a class operation. Thus, the instances denoted by *a* and *c* are seen passively by the class. Since instance *c* does not exist on processor *P1*, a copy of the internal data of the instance denoted by *c* is sent by *C1* in *P2* to *C1* in *P1*. The instance may even migrate to *P1*. In any case, the internal data of the instances denoted by *a* and *c* are now visible by class *C1* in *P1* and can be manipulated by the class operation. The new instance which is the result of the class operation is created by *C1*. The name of the new object is sent back to the caller by a message.

Next, assume the following message is sent to instance *d* in processor *P2*.

request2: b.

In this case, instances *b* and *d* exist in processor *P2*. Thus, these instances do not migrate. The instance denoted by *d* operates using the instance denoted by *b*. A new instance (denoted by *e)* is created as the result of the operation. The instance denoted by *d* sends name *e* to the caller. The instance denoted by *e* can migrate, if necessary.

4. THE DESIGN OF AN OBJECT ORIENTED ARCHITECTURE

A new architecture, called ZOOM, has been designed based on the distributed object model.

4.1. Names and Objects

The relations between names and objects in the basic object model [Tokoro82, Tokoro83] are summarized as follows (see Fig. 5): A **name** consists of its **id, scope, property** and a pointer which denotes an object. **Scope** consists of a list of objects which are permitted to access the name according to the specified access methods. Scope defines the domain in which the object can be seen, i.e., the access list of the name. **Property** defines the property that must always be kept by the name, i.e., the invariant property of values denoted by the name. It includes such things as data type and assertion. **Object** consists of **attribute, representation,** and **bit sequence. Attribute** asserts the set of valid operations for the object. It thus represents the class of the object. **Representation** represents the interpretation method for the bit sequences. **Bit sequence** holds the internal state of the object.

The architecture follows this basic structure of names and objects and extends it as described in the following:

(1) For the objects of architecture pre-defined classes, such as integer, real, and character string, the basic structure is used. Specifically, such an object can be contained in the pointer part of a name if the size of the object is small enough. In such a case, the pointer part stores **attribute, representation,** and **bit sequence.**

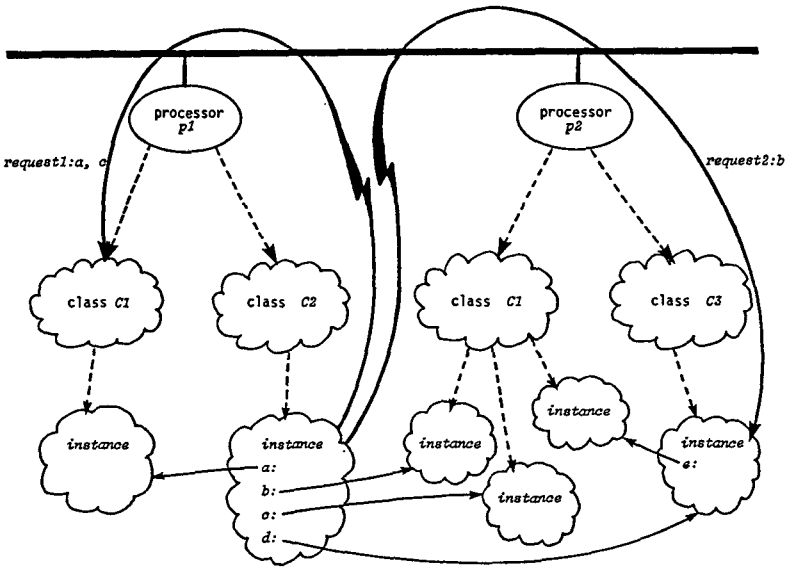

Fig. 4 The distributed object model

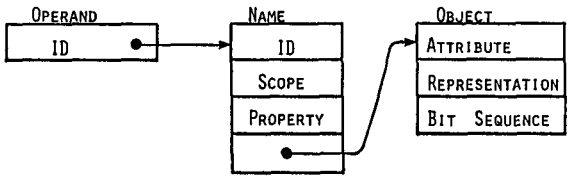

Fig. 5 The basic structure of names and objects

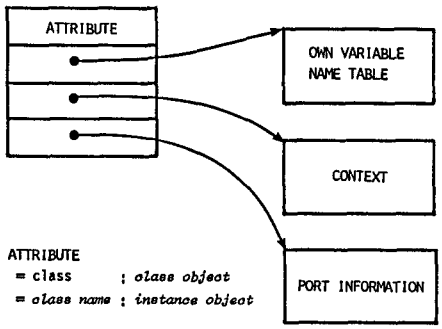

Fig. 6 The extended structure of objects

(2) For the objects other than architecture predefined classes, the extended structure shown in Fig. 6 is used. It consists of the **attribute,** pointer to **own variable name table,** pointer to **context chain,** and pointer to **port information.** There is no **representation** part in this structure since the structure of these objects is unique. The attribute of a class object is *CLASS.* The attribute of an instance object is the identifier of its class. **Own variable name table, contexts,** and **port information** together, which correspond to the **bit sequence** of the basic structure, compose the execution environment of the object. **Own variable name table** contains the own variables of the object. **Context** contains the status of execution, i.e., executing code segment number, instruction address, and the **local name table. Port information** contains information about the object's communication ports which will be described in detail in section 4.3.

(3) Two kinds of names, **local name** and **global name** are defined in this architecture. A **local name** is a name which is referenced within the context. A **local name** consists of the **local id** within the context, **property,** and a link to an id for **global name** to share an object with other contexts or a pointer to a locally defined object.

(4) A **global name** is a name which is shared among contexts. A **global name** consists of an id for **globalname, scope, property,** and a pointer to an object.

Fig. 7 shows the **Class Template Table** (CTT) in relation with the execution environment for instance *I1* and class *C1* on one processor. An entry of the **Class Template Table** designates the codes and local name tables of the class and instance operations, the class own variable name table, the templates for the instance own variable name table, the assertions, and the local instance pool.

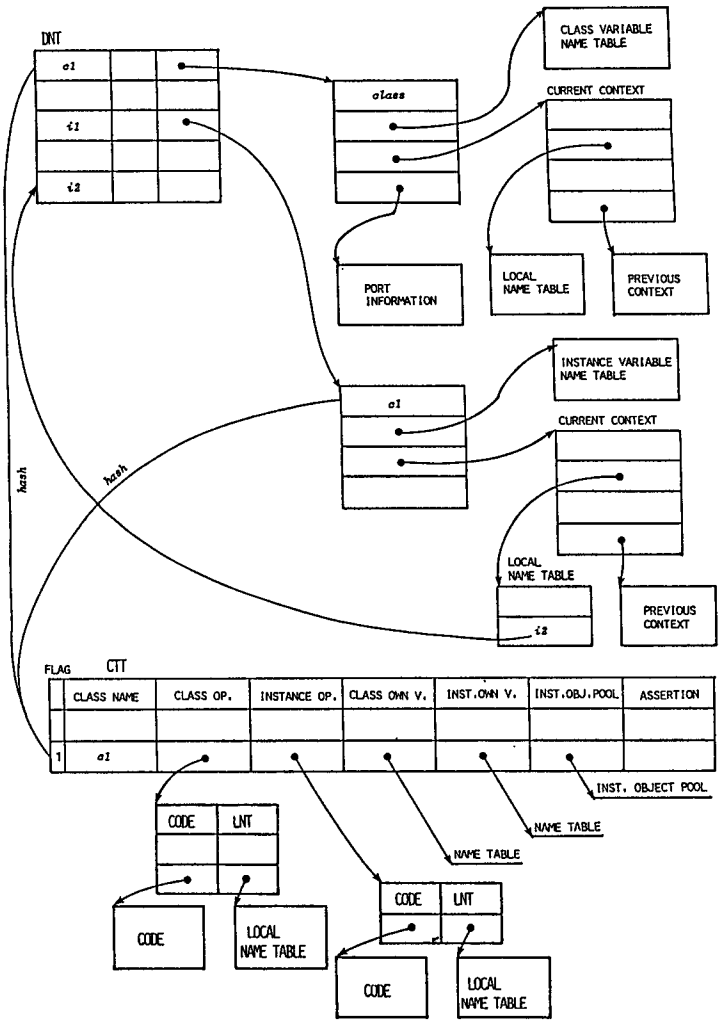

Fig. 7 The class template table and execution environment

4.2. Unique IDs

A global name has two identifiers. One is the unique ID within a processor called **domestic UID** and the other is the unique ID within connected networks called **global UID.** A global UID is the concatenation of the host id on which the object resides and the domestic uid of the object.

As shown in Fig. 8, all the objects, which can be accessed by any object within a processor, have their domestic UID's registered in the processor's **domestic name table** (DNT). Any accessible object which does not reside in the processor has its **domestic UID** with its **global UID** in the processor's **Unique ID Mapping Table** (UMT). Any object which is accessed by any object beyond the processor also has its **domestic UID** with its **global UID** in the processor's UMT.

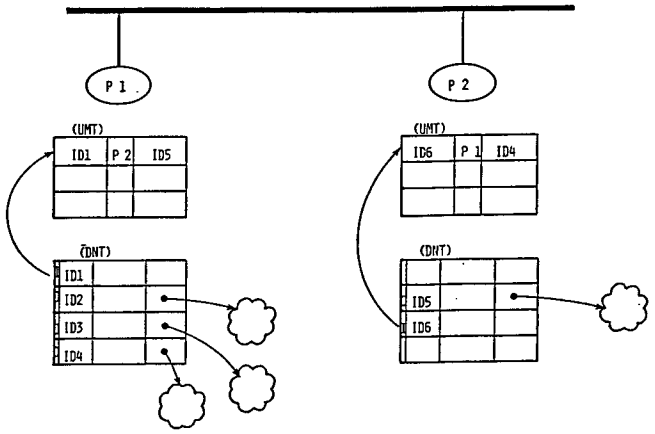

Fig. 8 Domestic and global UIDs for global names

4.3. The Realization of Inter-Object Communication

Our design goals for inter-object communication are as follows:

(1) Inter-object communication should be used for both remote invocation and inter-process communication.

(2) A peer object may be chosen dynamically at run time rather than statically at compile time.

(3) A powerful mechanism should be provided to protect an object from illegal and malicious access.

These goals are achieved as follows: Inter-object communication is realized by using **reliable datagram,** which provides reliable message packet transport, and **virtual circuit,** which provides reliable message stream transport. Reliable datagram is used for remote invocation and signal for exception handling. Virtual circuit is used for delivering an ordered sequence of messages between objects. Both of these provide blocking send/receive and nonblocking send/receive.

An object possesses **ports**. From the programmer's viewpoint, a port is a window which provides unidirectional message transport between objects. From the implementation viewpoint, a port is a protected message buffer. An object can send a message through one of its ports to the destination port of the destination object, and receive a message from a source port of a source object through one of its ports. Each input port of an object can correspond to a **request** to that object.

In addition to assigning **scope** to an object, we have decided to assign **scope** to each port of an object. The scope of a port consists of a list telling who is permitted to communicate with that port, i.e., the access control list of the port. More accurately speaking, the name of an object contains the names of the ports of the object, and each port name has its scope (Fig. 9). The scope of a port is defined when the port is opened (for reliable datagram) or connected (for virtual circuit) and is valid until the port is closed. Thus, access to an object is managed by the object itself, not by the accessing object. There are five methods for specifying the scope: **specific, instanceof, any, oneof,** and **except. Instanceof** permits communication with any instances of the specified class. **Any** permits communication with any object/port. **Oneof** permits communication with one of the object/port names in the list. **Except** permits communication with an object/port if it is not found in the list. Providing a scope list for each port realizes powerful access control for a port by architecture, and this reinforces the protection of objects.

A **message** is created when a send instruction is executed. A message consists of a sender object name, sender port name, receiver object name, receiver port name, the number of parametric objects, and a list of the names of the parametric objects.

4.4. The Instruction Set

Three basic operations, **assoc, link,** and **eval,** were defined for names and objects in [Tokoro82, Tokoro83]. **Assoc** associates an object to a name. **Link** associates one name with another name. **Eval** evaluates an object with one of the defined requests and operand object names. The instructions supported by this architecture can be classified into three categories: name instructions which correspond to **assoc** and **link,** send/receive instructions which correspond to **eval,** and other instructions. All the instructions with necessary/possible parameters are listed in Table 1.

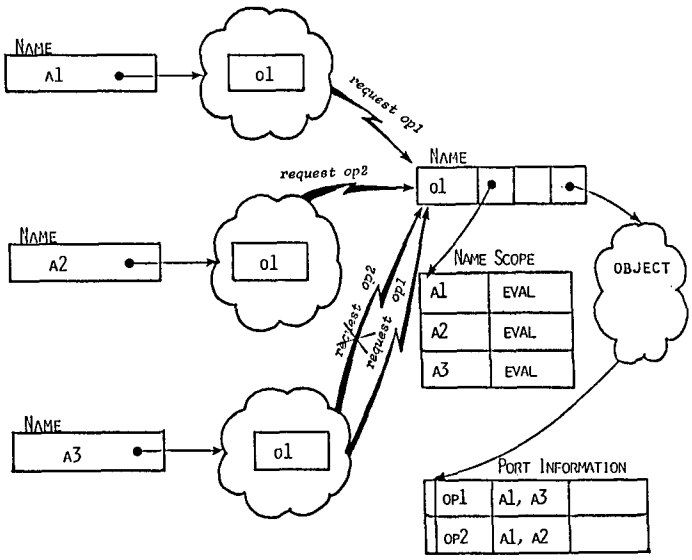

Fig. 9 The protection mechanism

4.4.1. Name Instructions

The name instructions deal with the relation between a name and object. There are ten instructions in this category: **assoc, assocg, link, linkg, unlink, unlinkg, reggnt, equal, setscope,** and **resetscope** instructions. **Assoc** lets the ultimate local name linked by NAME2 point to the first global name pointed to by local name NAME1. **Assocg** lets the ultimate global name linked by NAME2 point to the object denoted by local name NAME1. **Link** lets local name NAME1 point to local name NAME2. **Linkg** lets the first global name linked by local name NAME1 point to the first global name linked by local name NAME2. **Unlink** deletes the link from local name NAME. Thus, the local name NAME hereafter points to no name or

Table 1 The instruction set

OPERATOR	OPERAND
assoc	NAME1, NAME2
assocg	NAME1, NAME2
link	NAME1, NAME2
linkg	NAME1, NAME2
unlink	NAME
unlinkg	NAME
reggnt	NAME1, NAME2
equal	NAME1, NAME2, BOOLEAN
setscope	NAME, SCOPE
resetscope	NAME1, NAME2
openport	L-PORT, SCOPELIST, MODE, PORT
closeport	PORT
brdgsend	PEER-PORT, PEER-OBJECT, N-ARG, ARG-LIST, PORT
brdgrec	PORT, N-ARG, ARG-LIST
nbrdgsend	PEER-PORT, PEER-OBJECT, ENTRY-POINT, N-ARG, ARG-LIST, PORT
nbrdgrec	PORT, ENTRY-POINT, N-ARG, ARG-LIST
connectport	L-PORT, SCOPELIST, MODE, CONN
disconnectport	CONN
bvcsend	CONN, N-ARG, ARG-LIST
bvcrec	CONN, N-ARG, ARG-LIST
nbvcsend	CONN, ENTRY-POINT, N-ARG, ARG-LIST
nbvcrec	CONN, ENTRY-POINT, N-ARG, ARG-LIST
notify	N-ENT, PORT, PROC-NAME, ...
createinst	NAME, SCOPE
freeze	NAME1, NAME2
unfreeze	NAME1, NAME2
self	NAME
exit	

object and becomes undefined. **Unlinkg** deletes the link from the first global name linked by local name NAME. Thus, the global name hereafter points to no name or object and becomes undefined. **Reggnt** creates a global name, lets the global name denote the object denoted by local name NAME1, and lets the ultimate local name linked by NAME2 point to the global name. **Equal** checks whether NAME1 and NAME2 denote the same object or not. If they denote the same object, BOOLEAN is set to *true;* otherwise it is set to *false*. **Setscope** adds a scope for the name of an accessing object to the scope of the first global name linked by NAME. **Resetscope** deletes a scope for the name of an accessing object designated by NAME2 from the scope of the first global name linked by NAME1.

4.4.2. Send/receive Instructions

There are thirteen instructions in this category. The first six instructions are for the reliable datagram communication. **Openport** opens the port designated by L-PORT with scope SCOPELIST. A **port incarnation number** is returned to PORT. The opened port can send/receive messages to/from a peer port of a peer object if and only if the peer port of the peer object is included in the scope of the opened port. **Closeport** closes the port designated by PORT. **Brdgsend** sends a message which consists of the number of arguments (N-ARG) and the list of arguments (ARG-LIST) from PORT to PEER-PORT of PEER-OBJECT. The execution is suspended until the message is received by PEER-OBJECT. **Brdgrec** receives a message from PORT and associates each of the arguments to formal parameters designated by ARG-LIST. The execution is suspended until a message arrives at PORT. **Nbrdgsend** sends a message in the same manner as **brdgsend.** The execution, however, proceeds, and when the message is received by PEER-OBJECT, the execution moves to the program designated by ENTRY-POINT. After the end of the execution of the program, the execution resumes. **Nbrdgrec** receives a message in the same manner as **brdgrec.** The execution proceeds, and when a message arrives at PORT, the execution moves to the program designated by ENTRY-POINT. After the end of the execution of the program, the execution resumes.

The next six instructions are for the virtual circuit communication. Two **connect** instructions executed in different objects establish a connection between their L-PORTs if and only if i) one is with MODE *send* and the other with MODE *receive* and ii) their SCOPELISTs contain the other's object/port. When it is established, a **connection incarnation number** is returned to CONN. While connected, these ports communicate only with one another. **Disconnect** disconnects CONN. After a port is disconnected, the port may be connected with another object/port. **Bvcsend** and **bvcrec** are the blocking send and receive instructions through connection CONN. **Nbvcsend** and **nbvcrec** are the non-blocking send and receive instructions through connection CONN.

The last instruction in this category is **notify,** which is used for both the datagram and virtual circuit communications. **Notify** takes a list of the pairs of either PORT or CONN and PROC-NAME. When a message arrives at PORT or CONN, its corresponding PROC-NAME is executed.

4.4.3. Other Instructions

There are five instructions in this category: **createinst, freeze, unfreeze, self,** and **exit. Createinst, freeze,** and **unfreeze** are instructions which are used exclusively in a class. **Createinst** creates the skeleton of an instance of the class and associates it with NAME. **Freeze** freezes the object denoted by NAME1 and associates the frozen object with NAME2. **Unfreeze** unfreezes the frozen object denoted by NAME1 and associates it with NAME2. **Self** makes a link from local name NAME to the global name of the object that execute this instruction. **Exit** exits from the current context.

4.5. Addressing

There are five addressing modes for each operand of an instruction: **global name** (gn), **local name** (ln), **own variable name** (on), **local instance pool** (ip), and **immediate** (im). **Global name** mode is used for system defined names which are globally accessible. **Local name** mode is used for accessing a local name table of the context. **Own variable name** mode is used for accessing the object's own variable name table. **Local instance pool** mode is used to access literal constants within the object. **Immediate** mode is used to interpret the operand as an immediate literal constant.

4.6. Architecture Predefined Objects

The architecture provides the following four facilities: the **METACLASS, basic classes, file system** and **user interface.** The **METACLASS** generates user defined classes. **Basic classes** are the integer, real, character, string, and logical classes. **File system** and **user interface** is being designed for this architecture.

5. EVALUATION

This section describes the evaluation of the architecture through program execution on a software simulator and the estimation of hardware cost and speed using the preliminary hardware design.

5.1. The Simulator and Program Execution

A software simulator and a cross assembler for the ZOOM architecture have been implemented, both of which run under the VAX/Unix operating system. The program size of the simulator is about 7000 lines, and the program size of the ZOOM assembler is about 3000 lines in the C programming language.

Various programs have been written in the ZOOM assembly language and simulated by the simulator. The purpose of program execution on the simulator is twofold. One is to check the validity and to measure the efficiency of the ZOOM architecture. The other is to produce application and system programs in advance of the expected VLSI implementation of the architecture.

As an example of the ZOOM assembly language

programs, Fig. 10 shows a part of class *MAILBOX* for a simple mail system (called the ZOOM mail system hereafter). An instance *mailbox* manages mail by the use of the list of instances of class *MAILLIST*. Each user has one instance of *MAILBOX* in order to receive mail. A *mail* is an instance of class *MAIL* and contains a mail text. Table 2 shows the class and instance operations of these classes and instances.

The following steps are taken to send mail from one user to another: A user sends message *CREATE* to class *MAIL* in order to create instance *mail*. The user sends the name of the instance to the RECEIVE port of instance *mailbox* of the receiving user. The receiving user sends message *TAKE* to his/her *mailbox*, and the *mail* at the head of *maillist* is returned. The receiving user then sends message *READ-MAIL* to *mail*, and the text of *mail* is returned.

In order to compare the descriptivity and efficiency of programs on the ZOOM architecture, a similar mail system has been programmed in the C programming language to run under Unix 4.1BSD

```
#
# This is mailbox class
#
MAILBOX:class
classown {
        createp, owner, answer: port
}
instanceown {
        receivep, countp,
        sendersp, takep,
        mailp, mlp, mlstp,
        bc, bp, up, answer,
        truep, falsep,
        whilet, whilef:         port
        owner:                  name
        head, tail, tlist:      MAILLIST
        mcount, tcount:         int
        temp:                   MAIL
        myself:                 self
        mail: link              MAIL
        maillist: link          MAILLIST
}
#
iop {
        zero:           int = 0
        one:            int = 1
}
#
# The following methods are class operations.
#
#
# (0) initialize & main loop.
init:   cproc
        openport        CREATE, scope(any, anyport),
                        mode(receive), createp[on]
        notify          from(createp[on]), then(create)
        closeport       createp[on]
        exit
        end
#
# (1) external port.
create: cproc
        int{
        sender, gnam:   name
        sendp:          pid
        inst:           own
        }

        brdgrec         from(createp[on]),
                        mesg(sender[ln], sendp[ln], owner[on])
        openport        RET, scope(specific, sender[ln],
                        sendp[ln]), mode(send), answer[on]

        createinst      inst[ln]
        assoc           zero[ip], inst.mcount[ln]
        reggnt          inst[ln]
        brdgsend        to(sendp[ln]), in(sender[ln]),
                        mesg(gnam[ln]), from(answer[on])
        closeport       answer[on]
        exit
        end
```

Fig. 10 Programs for a simple mail system

Table 2 Class and instance operations

	MAILBOX	MAILLIST	MAIL
class operation	CREATE	CREATE	CREATE
instance operation	RECEIVE! COUNT? SENDERS? TAKE?	SET_SUC! SET_PRE! DELETE! NEXT? GET_MAIL?	READ_MAIL? SUBJECT? SENDER?

(called the C mail system hereafter). The C mail system is composed of two programs, *MAILBOX* and *MAIL*. A mail text exists as a file. Each user has one spool directory to hold their mail texts. The *MAIL* program receives mail from a user and places it in a file under the receiver's spool directory. The *MAILBOX* program manages files under the receiver's spool directory. The C mail system does not support the protection or the mail delivery beyond a processor supported by the ZOOM mail system.

Table 3 and 4 show the usage frequency of instructions and program sizes for the ZOOM mail system, respectively. Table 5 and 6 show those for the C mail system. The number of source lines of the ZOOM mail system is almost the same as that of the C mail system. The object program size of the ZOOM mail system is twice as large as that of the C mail system. The size of the executable program of the C mail system, however, is eight times larger than that of the ZOOM mail system. This is because the executable program of the C mail system includes various library programs. Such library programs correspond to the metaclass and system defined objects of the ZOOM architecture. In the ZOOM mail system, only the function of instance creation provided by the metaclass is used. In addition, the C mail system would have been much larger if it had supported protection and mail delivery functions similar to the ZOOM mail system. Thus, the object program size of the ZOOM mail system is considered reasonably small.

In previous programming languages and systems, an executable program is stored in a file and is copied into the main memory to be executed. When the execution terminates, the program is removed from the main memory. When they want to preserve the internal state of the execution afterward, the program must explicitly be written to create a file and store the internal state in the file, and the file name or the directory name of the files must be predefined for the programs which use that internal state. Moreover, in cases where multiple programs exchange data asynchronously, when programs are executing concurrently they communicate with each other by using inter-process communication primitives; otherwise they communicate through files. Thus, the programmer pays most attention to describing data exchange among programs and protection rather than describing the functions of the objective system.

On the other hand, in object oriented systems, once objects are created, they remain extant. They accept requests and respond in terms of messages. The information preserved in an object cannot be accessed directly. Thus, the method for communication among objects is unique and does not vary

according to the state of the receiving object. Thus, the distinction between an object and the objects that use the object becomes clear. Therefore, the programmer can concentrate on describing the functions of the objective system and the described programs become easy to maintain.

Table 3 Usage frequency of the instruction set for the ZOOM mail

	MAILBOX		MAILLIST		MAIL	
	class	instance	class	instance	class	instance
openport	2	26	2	8	7	9
closeport	2	26	2	8	7	9
brdgsend	1	28	1	4	11	4
brdgrec	1	24	1	5	8	5
notify	1	6	1	1	3	1
createinst	1	0	1	0	1	0
assoc	1	4	2	2	3	0
reggnt	1	0	1	0	1	0
exit	2	15	2	6	4	6
total	12	129	13	34	45	34

Table 4 Sizes of the objects for the ZOOM mail (Byte)

	MAILBOX		MAILLIST		MAIL	
	class	instance	class	instance	class	instance
object code	70	962	61	228	316	230
own variable name table	24	200	8	80	88	48
total of local name tables	32	80	48	72	56	72
local instance pool	12		0		97	
total size	1380		497		907	

5.2. The Preliminary Hardware Design

The following assumptions are made in the preliminary hardware design for the ZOOM architecture:

(1) the size of the unique id's are 48 bits;

(2) the width of the memories and busses is 32 bits;

(3) there are three interleaved memories: the instruction memory, global name table memory, and object memory;

(4) there is no cache mechanism in this design;

(5) input/output, memory management, and floating point arithmetic functions are excluded in this design; and

(6) there is no consideration of the load factors on the gates.

The result of the preliminary hardware design shows that the architecture requires about 6000 random logic gates, 30 registers (about 1500 bits total), and ROMs for microprograms which have not, at the moment of this writing, been estimated. Although such estimated hardware sizes may largely change with progress in the stages of the development, we consider the sizes shown to be well within state of the art of VLSI technology which can be implemented on a chip.

Table 7 shows the required memory cycles for some instructions. The memory cycles for **send, receive,** and **open** vary according to the number of arguments and the content of scope definition. **Assoc** requires memory cycles for executing an assertion procedure if one is specified. The memory cycles shown in the table are those when **send** and **receive** have three arguments, **open** has a scope with one entry, and **assoc** associates an object to a name without an assertion. **Add** is one of the architecture provided integer class operation. One context change requires 16 memory cycles.

Table 5 Usage frequency of the instruction set for the C mail

OPERATION	MAILBOX	MAIL
clrl	1	0
incl	1	0
subl2	1	0
subl3	24	6
tstb	6	0
tstl	7	1
cmpl	4	0
movl	4	1
movab	2	1
pushl	74	24
jbr	23	3
jeql	14	1
jneq	4	0
calls	38	17
ret	4	1
cvtbl	1	0
total	208	55

Table 6 Sizes of the programs for the C mail (Byte)

		MAILBOX	MAIL
relocatable code	text	920	300
	data	384	152
	bss	0	0
	total	1304	452
executable code	text	6144	5120
	data	1024	1024
	bss	1908	1564
	total	9076	7708

Table 7 Memory cycles for some instructions

INSTRUCTION	MEMORY CYCLES
send	18
receive	16
openport	11
closeport	8
reggnt	5
link	4
assoc	4
add	6

The result of the estimated execution speeds of the instructions suggests that this architecture executes programs very fast. For example, when we assume that one memory cycle is 500 nsec, **send** and **receive** instructions take less than 10 μsec. One context change takes just 8 μsec. By introducing caches and optimizing hardware especially for architecture provided operations, we expect this architecture to execute programs much faster.

This architecture can compose a distributed system as a natural extension and the same programs can run in the distributed system. In such a distributed environment, the advantages of this architecture are more than just faster execution speed.

6. CONCLUSION

In this paper, we proposed a new object model, called the **distributed object model,** that is suitable for distributed computing. In this object model, an object is **active** and **self-contained.**

We then designed a new object oriented architecture called **ZOOM** based on this model. In the ZOOM architecture, **scope** provides a powerful protection mechanism which is more reliable and versatile than the capability-based mechanism. Inter-object communication and object migration mechanisms provide efficient and versatile functions for the realization of distributed systems.

We have implemented a software simulator and a cross assembler for this architecture. The results of evaluating the architecture through writing and executing ZOOM programs on the simulator and the preliminary hardware design show the feasibility of the architecture and the high potential for being one promising candidate for the next generation computing systems.

Thorough evaluation of the architecture is being performed to improve the architecture. The development of various application and system programs are also being performed by using the simulator. The detailed hardware design for the VLSI implementation of this architecture will commence sometime this year.

The design of an object oriented language called ORIENT is also being carried out. This language will be used to implement the operating system, programming environment, and various application systems which run on this architecture.

ACKNOWLEDGEMENT

The authors are grateful to the members of the Object Oriented Programming/Processing Systems (OOPS!) project for their valuable comments. This research has been supported in part by the Ministry of Education, Science, and Culture under Grant-in-Aid for Scientific Research No. 57550222.

REFERENCES

[Bobrow 76] Bobrow, D.G. and Winograd, T., "An Overview of KRL, a Knowledge Representation Language," CSL-76-4, Xerox PARC, July 1976.

[Bobrow 82] Bobrow, D.G., "The LOOPS Manual," Palo Alto Research Center Xerox PARC, KB-VLSI-81-13, 1982

[Dennis 66] Dennis, J.B. and Van Horn, E.C., "Programming Semantics for Multi-programmed Computations," Comm. ACM, Vol. 9, No. 3, March 1966.

[DOD 80] "Reference Manual for the Ada Programming Language," United States Department of Defense, 1980.

[Fabry 74] Fabry, R.S., "Capability-Based Addressing," Communications of ACM, Vol. 19, No. 7, 1974.

[Goldberg 83] Goldberg, A. and Robson, D., "Smalltalk-80: The Language and its Implementation," Addison wesley, 1983.

[Goldstein 80] Goldstein, I.P. and Bobrow, D.G., "Extending Object Oriented Programming in Smalltalk," Proc. of the Lisp conference, Stanford, Ca, August, 1980.

[Greif 75] Greif, I., Hewitt, C., "Actor Semantics of PLANNER-73," Proc. ACM SIGPLAN-SIGACT Conf. Palo Alto, CA., 1975.

[Hewitt 73] Hewitt, C., et al., "A Universal Modular Actor Formalism for Artificial Intelligence," Proc. of IJCAI, pp. 235-245, 1973.

[Intel 81] "Intel iAPX432 General Data Processor Architecture Reference Mannual," Intel, Aloha, Oregon, 1981.

[Jones 76] Jones, A., et al., "A Language Extension for Controling Access to Shared Data," IEEE Trans. on Software Eng., Vol. SE-2, No. 4, pp. 277-285, Dec. 1976.

[Jones 80] Jones, A., et al., "The Cm* Multiprocessor Project: A research Review," CMU-CS-80-131, Department of Computer Science, CMU, July, 190.

[Kahn 81] Kahn, K.C., et al., "iMAX: A Multiprocessor Operating System for an Object-Based Computer," Proc. of the Eighth Symp. on Principles of Operating Systems, Dec. 1981.

[Linden 76] Linden, T.A., "Operating System Structures to Support Security and Reliable Software," Computing Surveys 8(4), December, 1976.

[Liskov 79a] Liskov, B., et al., "CLU Reference Manual," TR-225, Laboratory for Computer Science, MIT, Oct. 1979.

[Liskov 79b] Liskov, B., "Primitives for Distributed Computing," Proc. of the 7th Symp. on Operating Systems Principes, pp. 33-42, 1979

[Minskey 75] Minsky, M., "A framework for representing knowledge.," In P. Winston(Ed.), The psychology of computer vision. NEW York, McGraw-Hill, 1973.

[Morris 73] Morris Jr.,J.H., "Protection in Programming Languages," Comm. of the ACM, Vol. 16, No. 1, pp. 15-21, 1973.

[Simula 67] "The SIMULA 67 Common Base Languages", Publication S-22, Norwegian Computing Center, Oslo, 1970.

[Theriault 82] Theriault, D., "A Primer for the Act-1 Language," A.I. Memo No.672, April, 1982.

[Tokoro 82] Tokoro, M and Takizuka, T., "On the Semantic Structure of Information --- A Proposal of the Abstract Storage Architecture," Proc. of the 9th Int'l Symp. on Computer Architecture, pp. 211-217, April 1982.

[Tokoro 83] Tokoro, M., "Toward the Design and Implementation of Object Oriented Architecture," RIMS Symp. on Software Science and Eng. in the series of Lecture Notes in Computer Science, No. 147, Springer-Verlag, 1983.

[Wulf 75] Wulf, W.A., et al., "Overview of the Hydra Operating System," Proc. of the 5th Symposium on Operating System Principles, pp. 122-131, Nov. 1975.

[Wulf 76] Wulf, W., et al., "An Introduction to the Construction and Verification of Alphard Programs," IEEE Trans. on Software Eng., Vol. SE-2, No.4, pp. 253-265, 1976.

[Wulf 81] Wulf, W.A., Levin, R., and Harbison, S.P., "HYDRA/C.mmp: An Experimental Computer System," McGraw-Hill, New York, 1981

[Yonezawa 80] Yonezawa, A., "Specifying Software Systems with High Internal Concurrency Based on Actor Formalism," Journal of Inf. Proc., Vol.2, No. 4, 1980.