# Overview of a Parallel Object-Oriented Language CLIX

Jin H. Hur and Kilnam Chon

Department of Computer Science
Korea Advanced Institute of Science and Technology
P.O. Box 150 Chongryang, Seoul 131, Republic of Korea

UUCP: {mcvax,seismo}!kaist!jhhur
Internet: jhhur%sorak.kaist.ac.kr@relay.cs.net

## ABSTRACT

CLIX is a parallel object oriented language that embodies communicating process model of object oriented programming. It incorporates the notion of objects with communications in distributed systems. The objects in CLIX are encapsulation of information and communication handlers that operate on the information in response to messages. The underlying communication system is modeled as a mail system. In addition to presenting an overview of CLIX, it is reviewed in terms of relevant issues in parallel object oriented programming.

## 1. Introduction

Objects in object oriented programming can be viewed in many ways. In Smalltalk [12], objects are the information containers interrelated by the hierarchy of metaclass-class-instance and superclass-subclass. Objects in the Actor model [13] are active entities that communicate: Communicating Parallel Processes [7]. Objects in [20] is regarded as first class citizens of typed data abstraction augmented with type inheritance.

Objects can be viewed as self-contained active information processing agents that interact with their environment only through well-defined communication interfaces: the communicating process model [17]. A computation in the model is represented as a communication pattern over a network of objects [13]. Each object has the independent control thread of the execution; A network of objects has multiple, independent control threads. The objects thus may be units in concurrent computation. In sum, the object oriented system viewed as communicating processes renders itself for concurrent computation in distributed systems[1][2].

CLIX (Concurrent Language In EXperiment) is a parallel object oriented programming language that embodies the communicating process model of object oriented programming [14]. It is designed for programming in message passing architectures. CLIX incorporates the notion of objects with communications in distributed systems. In this sense, the objects in CLIX are equivalent to actors in the Actor model for the concurrent computation in distributed systems[1, 2]. CLIX also incorporates other issues of distributed computation like parallelism, and features of object oriented programming like class/instance relationship and inheritance mechanism by delegation[16].

In the subsequent section, an overview of the language CLIX is presented. In Section 3, the communication system, the central part of the model for CLIX, is described with an informal semantic description. Section 4 reviews CLIX in terms of relevant issues in parallel object oriented programming. Finally in Section 5 comes the current development status of CLIX.

## 2. Overview of CLIX

[1] The term 'distributed systems' in this context is to be interpreted in a broad sense including computer networks and message passing architecture such as multiprocessor systems without common store among processors.

### 2.1. Process Model for Objects in CLIX

Basic elements in CLIX are objects and communications. The objects in CLIX are similar to processes in distributed systems. The objects form a community executing asynchronously and communicating through message passing. In addition, the objects encapsulate information in themselves and provide well-defined communication interfaces to their environment. Encapsulation and communication handling are the two major functionalities of objects.

Objects in CLIX provide for the encapsulation of information in themselves with a communication interface invariant over their life time. The communication interface abstracts the detailed chores in accessing the internal data structure. Each object acts as a communication handler for the messages defined by the communication interface, and is activated only upon the receipt of a message.

CLIX basically provides for asynchronous communications in the underlying mail system; An object need not wait for the termination of the computation of the message receipient after message sending. In addition to the asynchronous communications, it provides for synchronous communications to deal with the case where the result is required for further processing. Thus, the programmer can have the full control over the communication system.

### 2.2. Objects

Objects in CLIX are encapsulation of local state information and communication handlers that operate on the local state. The definition of objects with common behavior is embodied in a class definition.

An object is composed of three parts: the object-id, the local state, and methods for acceptable incoming messages. The *object-id* is a system-wide unique mail address assigned to an object to which other objects - possibly itself - can send messages. The *object-id* is assigned to each object when the object is created. It may form a part of the local state of other objects, and may be passed around among objects. The *local state* is a collection of mail addresses to which messages can be sent, and is named by variables. The object may manipulate the local state by sending messages to the objects referenced by the local state variables. The *methods* are the actions the object takes in response to the corresponding incoming messages.

An object upon creation and appropriate initialization is ready to process incoming messages. Upon the receipt of a message, it executes the corresponding method and waits for the next message. The incoming messages are processed one by one in sequential manner unlike the pipelined processing of messages in the Actor system[1, 2]. It is due to the nature of sequential execution behavior internal to an object pertaining to our model. A class is defined as follows in the EBNF form:

```
(define <id> [ (with <pattern>) ]
  [ (constant {<constant definition>} ) ]
  [ (state    {<local state variable declaration>}) ]
  [ (initially <action>) ]
  (method    {<method definition>} ))
```

### 2.3. Primitives

In response to the incoming messages, an object performs a sequence of three kinds of primitive actions in the method:

(1) Local state change: The object may change the value of the local state variables, hence incurring a side-effect in processing subsequent messages to the object. The action is expressed in **setq**-command.

```
(setq X [X + 10])
```

(2) Communications: The object may issue communications to objects - possibly itself - to transfer information. The communication may be a reply to the incoming message or may be requests to some objects to perform some actions. There are four primitives for handling the communications:

```
(send next add: aVal)
(ask aFactorial eval: 3)
(forward next tally: currentSum)
(reply [t1 * t2])
```

(3) Creation of a new object: The object may create a new object. The network of objects may grow as the computation proceeds by creating new objects in response to messages. The action is expressed in new-expression.

```
(new Window (with X: 0 Y: 0 width: 10 height: 10))
```

(4) Control structure commands: In addition to those primitives above, directives to control the sequence of the actions are provided. They are **conditional-command, loop-command, select-command,** and **let-command.** The **conditional-command** and the **loop-command** have the conventional meaning: for conditional branch and for iteration, respectively. The **select-command** introduces the choice nondeterminism in the form of the Dijkstra's guarded command. The expressions in the guard arms are evaluated concurrently, and one guard whose expression is evaluated as TRUE is selected nondeterministically. The **let-command** introduces a form of lexical scope for transient variables by binding values to identifiers to set up the execution environment of the body. The expressions whose values are to be bound to the variables are evaluated concurrently. The two commands above are the only source of the concurrent execution in one command.

An object has communication handlers called methods constructed out of the above primitives. A method is defined as follows:

```
('[' <pattern> ']' [ where <expression> ]
  [ (using <id list>) ]
  <command > {<command>} )
```

A method is identified by its message pattern that is composed of a message selector and variables that are to be bound to incoming arguments. The format of message patterns is similar to that of Smalltalk:a single keyword or a sequence of 'keyword argument' pairs. The method is invoked only when the incoming message of the given pattern satisfies the accompanying expression to be TRUE. Otherwise, the next message is tried, and the current message is tried again after the next message. If there is no next message, the object is blocked until there arrives another message. In addition to local state variables, there are temporary variables that constitute the execution environment of the method. They are distinguished from the local state variables in that they are not regarded as a part of the local state. For example, they are not stored when the object is stored into the secondary storage as a persistent object. The remaining part is the commands to execute.

A program in CLIX is a set of class definitions and a sequence of commands to act upon the class definitions.

```
(program <id>
  { <import directive> | <class definition> }
  [ (var <id list>) ]
  <command> { <command> } )
```

A class definition may be included directly in the program or may be imported from the previously defined classes. When a program is executed, a special object called *root object* is created. The root object behaves just like other objects except that it doesn't have any communication handler. It may create some objects, issue communications, and change its own local state. These actions are the same as the regular objects. Unlike regular objects, however, it terminates its execution when it reaches the end of its body.

### 3. Communication System

The computation model of CLIX largely depends upon the semantics of communication primitives. This section presents the model and the primitives for the communication system.

### 3.1. Characterization

The communication system is characterized as follows:

- Point-to-point
  An object, S, can send a message to another object, R, only when S knows of the mail address of R. S gets to know R by either creating R or receiving the address of R in incoming messages. The communication pattern hence is point-to-point between objects, and does not imply direct broadcasting nor stream.
- Guarantee of delivery
  Once a message sending request is issued, the message is guaranteed to be delivered to the destination object. It's the underlying communication system's responsibility to make sure of the delivery.
- Preservation of message ordering
  When an object, S, sends two or more messages to an object, T, the temporal ordering of the message receipt by T is the same as the temporal ordering of the message sending by S. This characteristics gives yet simpler semantics for the computation than the Actor system [2, 21].

The communication system of CLIX is modeled as a mail system where the message sending is asynchronous.

A message is composed of two parts: mail header and message body. The mail header carries the relevant information concerning the communication itself. The mail header consists of four parts: to, from, reply-to, and message-id. *To* identifies the message receipient, and *from* identifies the sender. *Reply-to* designates the continuation point to which a reply, if any, is sent to. *Message-id* is a system-wide unique tag that identifies a transaction requested by an object. It differs from the tag in the Actor system[2] in that the former uniquely identifies a higher level sequence of actions in a transaction while the latter identifies the message itself uniquely. We can preserve the continuation point in a sequence of communications by means of *reply-to* and *message-id*; they in combimation can realize the inheritance mechanism by delegation[16, 22]. In executing a method for a message, the object can access the values of the message header throught a set of leterals like '@from'. They are not a part of the local state, but form a part of the execution environment for the method the object executing at the time.

The message body is the information transferred between objects. It consists of a message selector and arguments for an appropriate method. The format of the message body is similar to that of Smalltalk: a single keyword or a sequence of pairs of keyword and argument. The keywords form the message selector for the corresponding method.

### 3.2. Primitives

There are four primitives in our communication system.

(**send** <destination> <message pattern>)
(**ask** <destination> <message pattern>)[2]
(**forward** <destination> <message pattern>)
(**reply** <expression>)

When an object issues *send* command, the sender resumes its execution right after the message is sent regardless of the response of the receiver. The message is appended to the message queue of the message receipient to be handled later. This form of asynchronous communication is the canonical form of requesting an object an action.

When the sender object requires the result of the method execution by the receipient object returned back, the *ask* primitive is issued. The *ask* primitive is used as a part of expressions in CLIX programs. When the *ask* is issued, a system-wide unique tag is generated and is included in the request message as *message-id*. The sender then waits for a reply message with the same *message-id*.

When an object receives an *ask* request, it may not be able to reply by itself, but need to delegate the job to another object. The *forward* primitive is provided for that purpose. It has the same semantics as *send* except that the values of *reply-to* and *message-id* field of the mail header are preserved in the

[2] (ask <destination> <message pattern>) can be abbreviated as [<destination> <message pattern>] as syntactic sugaring.

forwarded message so that the message receiver can reply to the original sender directly. The object issuing the *forward* primitive can then proceed for its own work.

The primitive, *reply*, is provided to express the reply action to the *ask* request, and returns the object representing the result of the method execution. The *ask-reply* pair can express the conventional call/return style method call[2].

### 3.3. Typical Scenario

The followings are an informal description of the semantics of typical communications.

(1) When an object, S, issues a communication request to an object, R, it packs an appropriate message according to the request type. Variations according to the request type of either send, ask, or forward come from the handling of the mail header. See Appendix for how the mail header is set up.

(2) If the message is acceptable by R, namely there is a method with the same message pattern, it is appended to the message queue of R[3]. Otherwise, an error state occurs. The sender enters the *wait* state for the reply in case it has issued an *ask* request, or proceeds its own computation in other cases.

(3) When R is ready to process another message, it picks up the first message in the queue that satisfies the expression of the corresponding method as TRUE, and dequeues the message.

(4) R sets up the execution environment, and executes the corresponding method. The execution environment includes the initialized temporary variables and the mail header values in addition to the local state variables. When a *reply* command is executed in the method, R retrieves the value of mail headers *reply-to* and *message-id* from its execution environment, packs a message with those values, and sends the message to S. S then resumes the execution from *wait* state. In this case, the queued messages at S, if any, are preempted by the reply message.

(5) S and R proceed their own execution in parallel.

(6) When R finishes the execution of the method, it gets ready for another message.

## 4. Issues in Parallel Object Oriented Programming

### 4.1. Object-oriented-ness

The computation model in CLIX adopts the uniform object oriented metaphor from small objects like integers to large objects like databases as in Smalltalk. Conceptually, all request even to integers are made in the form of communications like

(ask 1 + 1 )

It provides a programmer a uniform view on the computation solely in terms of communications. It allows the programmer to break down a problem into subproblems and map the subproblems into objects, and to break down each subproblem into subproblems further, and so on. It comes in line with the concept of object oriented programming where objects are the realization of real world problems [12].

### 4.2. Inheritance

Inheritance mechanisms in many object oriented languages are expressed as the structural relationship among classes, and assume copy semantics; viz, a part of the code of superclasses is copied into the body of objects in the subclass[4]. Usually, the inheritance hierarchy matches the type hierarchy where a class is regarded as a type [18].

There is no inheritance mechanism as structural relationship among classes in CLIX. Instead, CLIX incorporates the 'inheritance by delegation' mechanism[16] by means of programmer-controllable communication primitive - *forward*. It works as follows: When an object, O, issues an *ask* request to an object,

---

[3] We here ignore the finite message transmission time from S to R unlike Actor model[2]. Relying on the assumption of 'guarantee of message delivery' and 'preservation of message ordering,' the state where the message is queued at the destination is considered to be the next state in the transition of the system state. It has a drawback that the potential unbounded nondeterminism due to finite, but not bounded, message delay[10] may not be stated, but simplifies the execution model.

$C_1$, and $C_1$ cannot satisfy the request by itself, $C_1$ delegates the request to another object $C_2$ by the *forward* primitive. $C_2$ may also delegate the request to $C_3$, and so on, until an object, $C_n$, may be able to process the request and reply to O. Each object $C_i$ may be regarded as the prototype object [16] of the object $C_{i-1}$. The *forward* primitive keeps the thread of request in the sequence $O \rightarrow_{ask} C_1 \rightarrow_{forward} C_2 \rightarrow_{forward} \ldots \rightarrow_{forward} C_n \rightarrow_{reply} O$ with the aid of mail headers *reply-to* and *message-id*.

### 4.3. Parallelism

The execution mode inside an object in CLIX is sequential in nature, and there is no explicit constructs for parallel composition of commands except for the simultaneous evaluation of expressions in select-command and let-command. Owing to the asynchronous communication and the inherent multitude of processes in the communicating process model, CLIX provides for the coarse grain parallelism.

There are a number of objects running simultaneously.

Some parallel object oriented languages like POOL[3] provides for the object body in an object. It renders a possibility of more parallelism since each object has something to do even without communications. In CLIX, an object acts only as the handler of request messages; Objects in a system is more closely coupled with communications. Asynchronous communications allow two or more objects to run in parallel, and the delegation mechanism frees objects as soon as they have done all they can do, hence being ready for another job.

The following example is a non-recursive factorial object exemplified in [19]. One can evaluate n! by

```
(ask (new Factorial) eval: 3)
```

for example. The **Factorial** object then creates **IntMult** objects, and sends request messages to them. The number of **IntMult** object executing in parallel increases as the depth of the method invocation increases. The multitude of objects running in parallel like this is the major source of the parallelism in CLIX.

```
(define Factorial
  (method
    ([ eval: ?n ] do
      (if [n = 0]
       then (reply 1)
       else (let ((mid [[n / 2] ceil:]))
              (let ((t1 [(new IntMult) low: 1 high: mid])
                    (t2 [(new IntMult) low: [mid + 1] high: n]))
                (reply [t1 * t2])))))))

(define IntMult
  (method
    ([ low: ?low high: ?high] do
       (select
         ([low > high] (reply 1))
         ([low = high] (reply low))
         ([low < high] (let ((mid [[[low + high] / 2] ceil:]))
                         (let ((t1 [(new IntMult) low: low high: mid])
                               (t2 [(new IntMult) low: [mid + 1] high: high]))
                           (reply [t1 * t2])))))))))
```

Active data structure composed of several objects can be accessed concurrently.

Conventional data structures are represented as passive data that are operated by an imaginary control in a sequential manner. Data structures in CLIX are represented as a set of objects that operate on themselves cooperating through message passing. The following class definition shows a part of the implementation of the **Collection** class similar to that of Smalltalk.

```
(define Collection
  (state value next)
  (method
   [tally: sum] (if [next = NIL]
                    then (reply sum)
                    else (forward next tally: [sum + value]))
   [add: aVal]  (if [next = NIL]
                    then (setq value aVal)
                         (setq next (new Collection))
                    else (send next add: aVal))))
```

One may create a new empty collection by

```
(setq aList (new Collection))
```

and add a new value to the collection by

```
(send aList add: 3)
```

When one issues a request,

```
(ask aList tally: 0)
```

each element examines whether it is the last element in the list. If so, it replies the accumulated sum thus far to the requester. If not, it forwards the request to the *next* element, and gets ready to process another message. When the collection is shared by a number of users and is accessed concurrently by the users, it can process several requests while processing a request. Thus, it increases the degree of parallelism.

Rather extensive examples on tree and graph data structure are provided in [11] in his Concurrent Smalltalk.

## 5. Conclusion

CLIX is a parallel object oriented language targeted for message passing architecture. Based on a communication system modeled by a mail system, it can support for object oriented programming with fair degree of parallelism. We have reviewed CLIX in terms of relevant issues in parallel object oriented programming.

We develop CLIX as an experiment on the object oriented programming environment in the Bee machine project [8]. Bee machine project embodies experiments on various programming environments including the conventional message passing operating system kernel. CLIX is designed as a way to study feasibility of the object oriented programming in the context.

In its own model, CLIX may be regarded as a descendent of the Actor model of the computation. CLIX however includes some features like versatile communication primitives and sequential execution of a method. We selected to design our own language for our experiment after surveying a number of parallel object oriented languages. Notable of them are POOL-T[5], ABCL/1[21], XCPL[6], and Orient84/K[15].

We are now developing a prototype implementation on a single CPU machine. In parallel, we are designing the object placement scheme including optimal object dispatch and object migration. The basic principle of the object placement scheme is to place objects in runtime without managing the state information based on the novel Client-Solicitor interaction model [9]. We will play a part of the build-up in our own learning on the development of programming environments in the multiprocessor systems as the experiment proceeds.

## Acknowledgement

The authors would like to thank D. Lee, C. Chung, and H.J. Park in System Architecture Laboratory for the discussion which has made the idea in the paper what it is. Great thanks also are to P. America in Philips Research Laboratories, Eindhoven, The Netherlands whose comments pointed out what might have been missed otherwise in the design of the language.

## Appendix

The semantics of four communication primitives is explained in terms of the manipulation of tl mail header. Let's assume primitives *primitive-send* and *wait* available for asynchronous message sendir and waiting for a reply, respectively.

(**primitive-send** to: R from: SELF reply-to: SELF message-id: τ <pattern> )
(**wait** τ )

The mail header is expressed as a sequence of 'name: value' pairs. SELF is the mail address of tl object issuing the command, and τ is a new system-wide unique tag. The object can also access in the for '@name' the value of mail header of the message it is currently processing. Then, each communicatic primitive is translated as follows:

(**send** o <pattern>)
→ (**primitive-send** to: o from: SELF reply-to: *nil* message-id: τ <pattern>)

(**ask** o <pattern>)
→ (**primitive-send** to: o from: SELF reply-to: SELF message-id: τ <pattern>)
(**wait** τ )

(**forward** o <pattern>)
→ (**primitive-send** to: o from: SELF reply-to: @reply-to message-id: @message-id <pattern>)

(**reply** e)
→ (**primitive-send** to: @reply-to from: SELF reply-to: *nil* message-id: @message-id e )

## References


1. G. Agha, "An Overview of Actor Languages," *SIGPLAN Notices* 21(10) pp. 58-67 (Oct. 1986 Special Issue on Object Oriented Programming Workshop at IBM Yorktown Height, Jun. 1986
2. G.A. Agha, "Actors: A Model for Concurrent Computation in Distributed Systems," AI TR-84 MIT AI Laboratory (Jun. 1985).
3. P. America, "Definition of the Programming Language POOL-T," ESPRIT Doc.Nr. 0091, Philij Research Laboratories (Sep. 1985).
4. P. America, "Object-Oriented Programming: A Theoretician's Introduction," ESPRIT Doc.N 0159, Philips Research Laboratories (May 1986).
5. P. America, "POOL-T -- A Parallel Object-Oriented Language," in *Object Oriented Concurre Systems*, ed. A. Yonezawa, M. Tokoro,MIT Press (Sept. 1986).
6. W.C. Athas, "XCPL: An Experimental Concurrent Programming Language," TR:5196:85, Califo nia Institute of Technology (Dec. 1985).
7. H. Baker, "Actor Systems for Real-Time Computation," LCS TR-197, MIT Lab. for Computer Sc ence (Mar. 1978).
8. K. Chon, "Bee Machine - Initial Assessment," Technical Memorandum SA-TM-01, KAIST (Fel 1987).
9. C. Chung, K. Chon, J.H. Hur, and D. Lee, "A New Distributed Computing Paradigm Based o Asynchronous Communications: Client-Solicitor Model," *Submitted to IEEE Tr. on Softwar Engineering*, (Feb. 1987).
10. W.D. Clinger, "Foundations of Actor Semantics," AI TR-633, MIT AI Laboratory (May 1981).
11. W.J. Dally, "A VLSI Architecture for Concurrent Data Structures," 5209:TR:86, California Insti tute of Technology (1986).
12. A. Goldberg and D. Robson, *Smalltalk-80: The Language Definition and Its Implementatior* Addison-Wesley (1983).

13. C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages," *Artificial Intelligence* 8 pp. 323-363 (1977).

14. J.H. Hur, "Definition of Programming Language CLIX," Technical Memorandum SA-TM-02, KAIST (Feb. 1987).

15. Y. Ishikawa and M. Tokoro, "A Concurrent Object-Oriented Knowledge Representation Language Orient84/K: Its Features and Implementation," *Proc. of ACM Conf. on OOPSLA '86*, pp. 232-241 (Oct. 1986). Also available as ACM SIGPLAN 21(11), Nov. 1986

16. H. Liberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," *Proc. ACM Conf. on OOSPLA '86*, pp. 214-223 (Sep. 1986).

17. V. Nguyen and B. Hailpern, "A Generalized Object Model," *SIGPLAN Notices* **21**(10) pp. 78-87 (Oct. 1986). Special Issue on Object Oriented Programming Workshop at IBM Yorktown Height, Jun. 1986

18. A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages," *Proc. ACM Conf. on OOSPLA '86*, pp. 38-45 (Sep. 1986).

19. D.G. Theriault, "Issues in the Design and Implementation of Act2," AI TR-723, MIT AI Laboratory (Jun. 1983).

20. P. Wegner, "Classification in Object-Oriented Systems," *SIGPLAN Notices* **21**(10) pp. 78-87 (Oct. 1986). Special Issue on Object Oriented Programming Workshop at IBM Yorktown Height, Jun. 1986

21. A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda, "Modeling and Programming in an Object Oriented Concurrent Language ABCL/1," TR C-75, Tokyo Institute of Technology, Tokyo, Japan (Mar. 1986).

22. A. Yonezawa, J-P. Briot, and E. Shibayama, "Object-Oriented Concurrent Programming in ABCL/1," *Proc. ACM Conf. on OOSPLA '86*, pp. 258-268 (Sep. 1986).