数理解析研究所講究録
第 618 巻 1987 年 59-75

Inheritance Mechanisms in Distributed Object-Oriented Languages

Jean-Pierre BRIOT * Akinori YONEZAWA †

Abstract

This paper discusses some main issues of inheritance mechanisms for Object-Oriented Programming in the context of a distributed (and concurrent) language. It is based on the experience and the experiment of the ABCL Project (design, implementation and experimentation of an object-oriented concurrent language).

We review some of the main inheritance strategies: ***inheritance, delegation*** and ***recopy***. A fourth model is presented and all are compared and criticized. Techniques relying on a shared memory assumption are rejected. We then point out the conflict between distributing the knowledge among objects and a good synchronization of the access to data. A micro-interpreter of an object-oriented language is presented and used to illustrate this problem through a simple example.

Keywords:

object, actor, knowledge sharing, inheritance, class, subclassing, delegation, uniformity, message passing, variables, methods, proxy, dynamicity, synchronization, atomicity, serialization, granularity, code sharing, distributed system, recopy.

1 Introduction

Inheritance [1] is a mechanism intensively used in object-oriented programming. It allows classification and modularity plus a good code sharing. A basic idea is the reuse of models. After defining an object, we would often like to refine it as a more specialized one. Rather than defining a new object from scratch, we could define it

*Tokyo Institute of Technology, Dept. of Information Science, Ookayama, Meguro-ku, Tokyo 152, LITP, University Paris VI, 4 place Jussieu, 75005 Paris

†Tokyo Institute of Technology, Dept. of Information Science, Ookayama, Meguro-ku, Tokyo 152

[1]We should rather use the term *Knowledge sharing. Inheritance* is only one of the strategies to specify and implement knowledge sharing. *Delegation* [Lieberman 86b] is another strategy we will discuss as well. The preeminence of inheritance among the majority of object-oriented languages gave this term a generic meaning. Delegation is related to the *actor* model of computation [Hewitt 76] [Lieberman 81], one of the main *object-oriented programming* model.

by inheriting the specification of the previous one. This intuitive mechanism sounds already more general and powerful than a simple automatic recopy facility (such as duplicating a text fragment in a text editor). Changing the first object could also change the new one inheriting from it. In that sense, inheritance is a good means to share code efficiently, and as mentions [Touretzky 86], it is also supposed to make knowledge searching more efficient.

Inheritance is also used to give hierarchical (or almost hierarchical) structure to knowledge, while gaining modularity. For representation systems, a standard example is the hierarchy of animal species.

There are a lot of different description and implementation strategies for inheritance, but usually there is a compromise between full dynamicity and full recopy. Also many systems rely on strong assumptions. The most usual assumption is the existence of shared memory. This allows us to gain efficiency to follow the inheritance links, which are implemented by simple pointers. In contrast we want to explore the strategies of inheritance in a general memory model, i.e., a general distributed model.

As the background of our discussion, we will first review the two main techniques [2] used in object-oriented systems, namely ***inheritance*** and ***delegation***. We will compare them, following [Briot 84] and [Lieberman 86a]. Only the delegation scheme is flexible enough to be free from the shared memory assumption, so we will study and criticize it in a distributed memory context. We will then discuss an alternative to delegation that solves its problems, but unfortunately this second model also suffers from some weaknesses. Before concluding we will also discuss the tradeoff between flexibility and recopy, as well as some techniques to make the copy approach more flexible.

2 Inheritance

The ***inheritance*** model (also called ***subclassing***, cf §2.3) is proposed by Smalltalk-76. It allows knowledge sharing among ***classes***. A ***class*** [3] can be declared as ***subclass*** of another class, this subclass will inherit the definition of the previous one. This hierarchy is represented by a tree, called the ***inheritance tree***.

This model is followed by most object-oriented languages, even if some propose their own extensions such as the handling of ***multiple inheritance*** (cf §2.4).

[2]There are other models, among them ***frames*** and ***semantic networks***. The ***frame*** model (illustrated by the languages FRL [Roberts & Goldstein 77] and KRL [Bobrow & Winograd 77]) could be reduced to the delegation model of actors as shown in [Briot 84]. ***Semantic networks*** languages such as NETL [Fahlman 79] use another model based on links. [Touretzky 86] gives a good comparative analysis of frames and semantic networks, as well as a good study of redundancy and ambiguity in ***multiple inheritance*** systems (cf §2.4) such as NETL.

[3]For an introduction to ***object-oriented programming*** and the notion of ***class***, see [Stefik & Bobrow 86].

2.1 Variables and Methods

The model of inheritance is strongly related to the *class* model of Smalltalk and its implementation decisions. As a consequence, the inheritance of the data-base of an object (called the *instance variable dictionary*) is opposed to the inheritance of its procedures-base (called the *method dictionary*). The first one is an inheritance of abstraction (specification), whereas the second one is an inheritance of behaviors (procedures code). The first one is static and the second one dynamic.

The instance variable dictionary of an object is split in two parts: the set of (instance) variables, specified by the class of the object, and the set of associate values, owned [4] by the object. These two sets must always be isomorphic. The set of values owned by an object is computed and assigned at its creation time (although each value could change later). For keeping the isomorphy, the set of variables of a class should be constant and assigned at creation time of the class before instantiating it. In consequence, the inheritance of the variables is *static* and performed only once. The set of inherited variables is merged [5] with the set of variables specified. Thus it is impossible to create a subclass of a not yet defined class.

On the other hand, inheritance of methods is dynamic. The lookup for a method begins in the method dictionary of the receiver's class. When none is found, the search resumes in the superclass. If there is no superclass, an error occurs.

2.2 Flexibility vs. Efficiency

This is one of the main tradeoffs in knowledge sharing. Inheritance is "half flexible and half efficient". Inheritance of variables is static thus efficient but changes won't propagate unless there is an automatic update mechanism (cf §7.2). The link between a class and its superclass is "hardwired," and uses pointers for efficiency (same for the link between an object and its class). Thus this model is not suitable for a distributed memory model.

2.3 From Concatenation to Inheritance

The *subclassing* mechanism adopted by Smalltalk-76 [Ingalls 78] and Smalltalk-80 [Goldberg & Robson 83] was first introduced in the language Simula-67 [Dahl et al. 70]. In this pioneering language, the notion of inheritance is really close to *concatenation* of text programs. The specifications of the two classes are merged, and their two program bodies concatenated (the keyword *inner* is used to specify where the inherited body is inserted). The flexibility was not a major goal in a compiled language built on an Algol foundation.

[4] The set of variables is owned by the class (thus shared by all instances of that class), while every instance of the class owns its private set of values. In contrast *class variables*, introduced in Smalltalk-72 [Goldberg & Kay 76], are variables whose values are also shared and owned by the class.

[5] These two sets are ordered and usually implemented with vectors. The simplest inheritance of variables becomes a simple concatenation (cf §2.3) of vectors, the inherited variables coming first.

2.4 Multiple Inheritance

A class could also have several superclasses. Such a system is called a *multiple inheritance* system. The strategy of exploration of the inheritance graph becomes more various and complex than for a simple tree. In fact every language proposes its own strategies. A sample of such strategies presentations could be found in [Moon & Weinreb 80] [Bobrow & Stefik 83] [Borning & Ingalls 82] [Touretzky 86]. This is not the subject of our paper.

3 Delegation

The *delegation* model proposed by Act-1 [Lieberman 81], as opposed to the above model, is fully dynamic. An object (called an *actor* in this computation model [Hewitt 76] where the *class* concept is absent) knows about another object called a *proxy*. The object will delegate to its proxy the messages that it doesn't recognize. This proxy could recursively delegate the message to its own proxy (which could in turn also delegate the message to its proxy). The proxy will handle and compute the message in place of the initial receiver. The intuitive formulation could be: "Could you please compute that request for me, because I don't know how to do this" [Lieberman 81].

3.1 Delegation vs. Inheritance

Compared to the Smalltalk-76 inheritance model, the inheritance of variables in the delegation scheme is dynamic. In fact accessing (read or write) a variable of the data-base is like performing an accessing method to this variable (cf next section). The second difference with the first model is the uniformity of the communication protocol: the delegation to the proxy of an object is performed by message passing and not through a system primitive and a physical link (pointer). Thus delegation can be fully designed at the language level and can be locally customized by the user easily. This model is independent of the assumption of shared memory and is perfectly suitable for a distributed model. It allows also complete dynamicity and modularity (there is no merging of the data-bases as in the Smalltalk-76 model). Two issues, efficiency and synchronization, must be addressed in this scheme. Efficiency can be gained by interpreter optimization such as caching [Deutsch & Schiffman 84]. But as we will see, synchronization conflicts with recursion in the delegation scheme, and the problem is not trivial.

3.2 Variables as Methods

In the delegation scheme, method activation as variable consultation should rely on message passing, because both of them could be delegated to another object. The strategy proposed by Henry Lieberman [Lieberman 86a] is to consider a variable as a method. Each variable is represented as a method (of the same name) which handles two messages: consulting the value, and assign a new value. We could view such a method as a closure which is able to return and assign its protected value.

4 Example of Delegation

The delegation model is simple, uniform, and flexible. Unfortunately, it has some deficiencies to handle safely synchronization in a concurrent environment. A simple example will illustrate this fact.

4.1 The Counter Example

The Counter object is a very simple example, but it gives a good illustration of objects with changing states. Let's describe it in an object-oriented language whose micro-interpreter will be presented in the next section.

```
; creation of a counter
(setq counter (make-object))

; the single variable
(put-variable counter 'contents 0)

; the three methods
(put-method counter 'reset '(lambda () (w 'contents 0)))

(put-method counter 'incr '(lambda () (w 'contents (1+ (r 'contents)))))

(put-method counter 'show '(lambda () (r 'contents)))
```

A method is represented by an anonymous function (lambda) whose parameters are the parameters of the method.
"(r *variable*)" means an access (read) to *variable* of the current object.
"(w *variable value*)" means the assignment (write) of *value* to *variable*.

A message is a list whose first element is the name of the method to be selected (thus called the *selector*). Other elements are the actual arguments of the message. The function *send* is used to send a message to an object (called the *target*).

Now here is a sample of interaction with the counter:

```
? (send counter '(reset))
= 0
? (send counter '(incr))
= 1
? (send counter '(incr))
= 2
? (send counter '(show))
= 2
```

4.2 1st Interpreter (no Delegation)

We now define a little interpreter of an object-oriented language in order to exhibit the delegation model. This is the first version of this interpreter, the simplest one that we will modify later to discuss several strategies. In this first interpreter, message passing is implemented as a simple functional call. Variables are accessed through primitive operations and not through message passing. As a consequence, this interpreter is not suitable for delegation.

The definition of this tiny interpreter (and the next ones) is given in Common-Lisp [Steele 84]:

```
; structure of an object
(defstruct object
  (variables (make-hash-table))
  (methods (make-hash-table))
  (proxy nil))

; message-passing form
(defvar self)

(defun send (self message)
  (apply (get-method self (car message)) (cdr message)))

; to define methods
(defun put-method (object name behavior)
  (setf (gethash name (object-methods object)) behavior))

(defun get-method (object name)
  (gethash name (object-methods object)))

; to define variables
(defun put-variable (object name value)
  (setf (gethash name (object-variables object)) value))

(defun get-variable (object name)
  (gethash name (object-variables object)))

; access to a variable
(defun r (variable)
  (get-variable self variable))

(defun w (variable value)
  (put-variable self variable value))
```

Every object owns two dictionaries (implemented by two distinct hash-code tables). The first one is the *(instance) variable dictionary*, containing the variables, and the second one the *method dictionary*, containing the methods. The third slot represents the *proxy* of an object and will be used later.

Message passing is implemented with a simple function application, applying the method associated to the selector (the head of the message list) to the arguments (the tail of the message list).

The variable *self* represents the current active object. It is declared as a special variable ([Steele 84]), which is dynamically bound at every activation.

Note that there is no error handler in this interpreter. It can be easily incorporated by adding some tests like checking if the receiver is an object, and testing the existence of a method (cf §4.4).

The access to variables as well as methods is a simple lookup in the corresponding hash-code table. There is no way to support (distributed) delegation with this first interpreter model because the access to variables doesn't rely on message passing.

4.3 2nd Interpreter (with Delegation)

We will now slightly modify our first interpreter in order to unify variables and methods. A variable is now represented by a method with 0 or 1 argument. Each *variable-method* is implemented by a closure to store and protect its value (the variable *value*). No argument means reading (and returning the value) of the variable. One argument means assigning that argument (which represents the new value) to the variable. The local variable *set?* is declared as predicate by the CommonLisp keyword *&optional* to test the existence of an argument.

Note that the *variables* slot of the *object* structure can be removed because it is now no longer used. We just show the changes from the first definition.

```
; to define variables
(defun put-variable (object name value)
  (put-method object name (make-variable-method value)))

(defun make-variable-method (value)
  #'(lambda (&optional (new-value nil set?))
       (if set? (setq value new-value) value)))

; access to a variable
(defun r (variable)
  (send self '(,variable)))

(defun w (variable value)
  (send self '(,variable ,value)))
```

The functions *r* and *w* are redefined as sending a message (whose selector is the variable) to the current receiver (*self*). The list messages are constructed with the backquote macrocharacter ('). A comma (,) indicates an element to be evaluated.

4.4 Implementation of Delegation

To implement a delegation mechanism, we have to redirect the message to another object (the *proxy*) in case of a method lookup failure. The send should now accept a third argument to correctly manage recursion as pointed out by Henry Lieberman [Lieberman 81] [Lieberman 86b]. We must send any recursive message to the initial (original) receiver and not to the current object. Intuitively, we could refine our first informal definition of delegation as: "Could you please compute that request for me, because I don't know how to do this. If you need any more information please send that request back to me".

The variable *self* is now no longer bound to the current receiver (the variable *target* is), but to the *initial* receiver.

The "external" (used in the programs) *send* function has only two arguments, the target and the message. It will call an "internal" *send* (the function *send1*) with three arguments, binding the variable *self* to the value of the variable *target*, thus identifying the initial receiver with the current receiver.

If no method associated to the selector is found, the message is delegated to the proxy. The message and the initial receiver (*self*) are communicated to the proxy through an "internal" call to *send1*. Thus recursion using the variable *self* will be handled correctly and always sent to the initial receiver of the *send*.

```
; message-passing handling the initial receiver
(defun send (target message)
  (send1 target message target))

(defun send1 (target message self)
  (let ((behavior (get-method target (car message))))
    (cond
      (behavior
       (apply behavior (cdr message)))
      ((object-proxy target)
       (send1 (object-proxy target) message self))
      (t
       (error "~S is a message unrecognized by ~S" (car message) self)) )))
```

Note that if there is no proxy, the delegation fails and an error is reported.

4.5 3rd Interpreter (Delegation and Concurrency)

We will now extend this second interpreter to accomodate concurrency. The "functional call" message passing will be changed in asynchronous message sending. A new slot *mailbox* is added to the *object* structure, and will buffer the messages sent to the object.

A new argument [6] to the send form is now the reply destination (*reply-to*) [Yonezawa et al. 86], allowing to specify the object an answer is sent to. We will use this to simulate bidirectional message call, because our message passing model is now unidirectional. The form "(reply *answer*)" is a simple way to send back a message to the reply destination if specified. For the same reason as *self*, the variable *reply-to* is declared as special (dynamic scoping).

To keep this interpreter small and simple, we will assume the existence of a *queue* data type module, and a *scheduler*. The function *make-queue* is the queue generator

[6]For the implementation of delegation, we should keep an internal *send1* function and the *self* argument as described above. We won't mention it here to avoid confusion.

(initially empty), ***enqueue*** the constructor, ***dequeue*** the operation to remove the first element, and ***queue-empty?*** the emptyness predicate.

```
; structure of an object
(defstruct object
  (variables (make-hash-table))
  (methods (make-hash-table))
  (proxy nil)
  (mailbox (make-queue)))

; message-passing form
(defun send (target message &optional reply-to)
  (enqueue (cons message reply-to) (object-mailbox target))
  (activate target))

(defvar reply-to)

(defun reply (value)
  (send reply-to '(reply ,value)))
```

The scheduler is assumed to include a set of active objects (also called ***ready***, i.e., eligible for the CPU service but not currently executing [Comer 84]). The function ***activate*** makes an object become active (if it is not already so) and inserts it in the set of active objects. The function ***disactivate*** removes an object from the set of active objects.

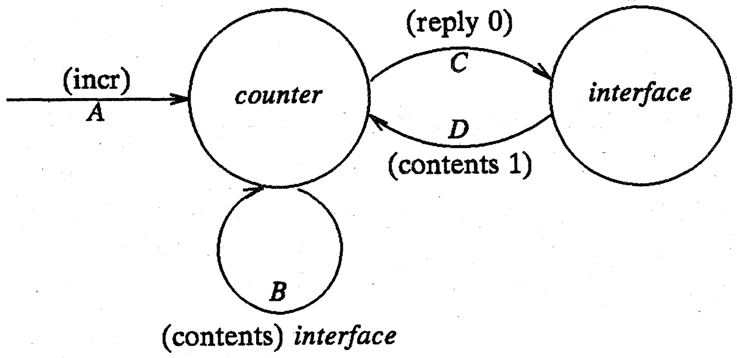
After selecting an object to become the ***current*** object to be executed, the scheduler will call the function ***execute***. For the sake of simplicity, we assume the computation of a message is atomic, i.e., it won't be interrupted. Thus there are only two cases. An object with an empty mailbox will be disactivated. Otherwise the first message will be removed from the mailbox queue, and it will be computed.

```
; activity of an object
(defun execute (self)
  (if (queue-empty? (object-mailbox self)) (disactivate self)
    (read-and-compute (dequeue (object-mailbox self)))))

(defun read-and-compute (envelope)
  (let ( (message (car envelope)) (reply-to (cdr envelope)) )
    (apply (get-method self (car message)) (cdr message))))
```

The access function to read a variable (the ***r*** function) has been slightly changed. This function accepts a second argument (***reply-to***), the reply destination the value will be sent to. There is no longer a function call, so the value won't be returned automatically. Therefore we must specify explicitly the reply destination.

```
; to define variables
(defun make-variable-method (value)
  #'(lambda (&optional (new-value nil set?))
        (if set? (setq value new-value) (reply value))))

; access to a variable
(defun r (variable reply-to)
  (send self '(,variable) reply-to))
```

As a consequence, the method ***incr*** of the counter becomes complicated. The starting point is the reading of the variable ***contents*** of the counter. This value should be sent back to some object. We will dynamically create a special (and local) object [7] dedicated to the incrementation. This object will receive the value, add 1 to it, and then send an assignment message [8] to the counter to assign ***contents*** with that new value. The next figure shows the ***counter*** object, this ***interface*** object, and their message exchanges.



Here are the four messages sent to increment the counter. The first one is the original one. They are time ordered from A to D by their causality. We assume the initial contents of the counter is 0.

```
(A)  (send counter '(incr))
(B)  (send counter '(contents) interface)
(C)  (send interface '(reply 0))
(D)  (send counter '(contents 1))
```

The definition of the ***incr*** method first creates this local object, bound to the local variable ***interface***. It will recognize only one method ***reply***, whose action is to send the assignment message "(contents (1+ value))" to the counter. This ***interface*** object created, ***counter*** will consult its variable ***contents*** and reply its value to the object ***interface***.

[7]Such an object is called a ***continuation*** in the ***actor*** model of computation [Hewitt et al. 74] [Lieberman 81]. A ***continuation*** is an actor which encodes all the behavior necessary to resume a computation after reception of an answer to a question.

[8]The message "(contents)" is used to read the variable ***contents*** of the counter. The message "(contents ***new-value***)" is used to assign it.

```
; redefinition of the method incr of the object counter
(put-method counter 'incr
  '(lambda ()
     (let ((interface (make-object)))
       (put-method interface 'reply
         '(lambda (value) (send counter `(contents ,(1+ value)))))
       (r 'contents interface))))
```

Note that the method *show* should also be redefined in the same way.

4.6 An Example of Bad Serialization

The model of the third interpreter has a big weakness as we can show now.

Suppose we send consecutively two messages *(incr)* to the counter. We cannot ensure that the contents will be the previous one plus 2. The *incr* method was "atomic" in our two previous interpreters, but is no longer now. Some messages can insert and interfer during the processing of the *incr* method, because this method now reduces to several message sendings. As a consequence, the two *(incr)* messages may overlap and the first side-effect could be shadowed by the second one. Another increment message will create four other messages passings (A' to D'). If the second message consulting the contents variable (B') happens to reach the counter before the first assignment message (D), the first incrementation will be shadowed by the (same) second one. In the other case, the double incrementation will work well as in the previous interpreter.

B' < D => contents : 1
D < B' => contents : 2

(M1 < M2 means that message M1 arrives before message M2 to the common target.)

5 Delegation and Synchronization

5.1 Atomicity and Synchronization

If we modify our third interpreter in an "undelegated" way, i.e., by coming back to a direct access to the variables like in the first interpreter, the example works fine again, because the *incr* method regains atomicity.

Our second interpreter uses also a delegation architecture, but the synchronization is ensured by the function call mechanism. Thus this interpreter also keeps variables access atomic.

5.2 Granularity of Objects

The distribution of objects increases modularity and concurrency in the system, but the serialization becomes hard to ensure. The *granularity* (i.e., the size of the objects) of an object-oriented system is an important tradeoff. ABCL is a language with a medium scaled granularity with no full uniformity, [9] different from the actor approach where actors are very small (and numerous).

5.3 Serialization Fairness and Recursion

The messages sent to an object O are ordered (in a queue) following the ordering of their arrivals to O. It is assumed that two messages cannot arrive at the same time. An undeterministic arbiter orders them. Thus in case of recursion (the object O sends a message to itself), the ordering of this recursive message and another message sent to O cannot be predicted. To ensure the atomicity of the incrementation of the counter, we need to lock the counter to avoid it to compute some other messages before the end of the computation of the *incr* message. But the problem is not simple because, on the other hand, the counter should accept the recursive messages (sent by itself) while temporary rejecting the other ones. It is possible to implement such a mechanism but the model becomes very complicated, which is opposed to its intuitive foundation. We need to use a *waiting mode* or/and a *synchronous* message passing type, such as the Now type of the ABCL model [Yonezawa et al. 86]. Then it is difficult to avoid the risk of deadlock.

5.4 Granularity of Delegation

Rather than a formal delegation model, Henry Lieberman proposed a new "philosophy" of inheritance. These ideas first introduced by Henry Lieberman in 1981 [Lieberman 81] have evolved [Lieberman 86a].

There is no formal definition of delegation, but the basic ideas of distributing [10] an object in several pieces to gain flexibility, and implementing variables as methods to allow a communication uniformity remain. Thus in any incarnation of the delegation in a concurrent world, the problem of serialization we pointed out remains and its resolution is not simple. On the other hand, delegation escapes these deficiencies when remaining in a sequential world, using function call to implement message passing.

5.5 The Replacement Actor Model

We noticed that the latest actor model [Agha 85] relies on *atomic* objects. In that recent model, there is no side effect. An actor must specify its *replacement actor* which could have different acquaintances, in order to allow changing states. As soon as this specification (using the primitive *become*) is encountered, the replacement actor begins to compute the next message in the mail queue, thus allowing a pipelining of messages. There is no delegation proposed in the actor model of [Agha 85], and it seems difficult to safely incorporate it as the next section shows.

[9] Other entities than objects (such as Lisp functions) are used in ABCL. In contrast Act1 [Lieberman 81] (and Smalltalk [Goldberg & Robson 83] as well) claims that everything is an object in its system.

[10] We presented a simplified model in §4.5. The model proposed in [Lieberman 86b] is more "distributed". Methods are themselves objects used with variables to build *extension* objects. An object can delegate a message to all its methods in parallel to give a chance to each method to handle it.

5.6 Serialization through Stream Manipulation

In the Vulcan language [Kahn et al. 86], the model used is close to the replacement actor model of Agha. A perpetual object is implemented in a Concurrent Logic Programming Language as a *perpetual process* (illusion), i.e., an *ephemeral* process that continually reincarnates itself in another process with the same functor and consuming the remainder of the message stream. Like in [Agha 85], a changeable state is represented by a new incarnation replacing the old one. The new process owns a new state instead of the old one. The stream (used as a communication channel) is passed along from process to process to provide a perpetual identity to the object. This strategy is analogous to the replacement actor scheme.

A delegation scheme is proposed [11] in Vulcan. The explicit manipulation of the communication stream avoids the serialization problem we pointed out above. Recursive messages (using *self*) are implemented by presetting the tail-recursive call with the messages to be interpreted before messages already on the stream. The same technique is used for delegation. Recursion will operate on *self* so that any messages sent to *self* during the delegation will arrive before those sent externally. This technique allows a good handling of delegation, because the communication channel is unique and explicitly declared. But the question of ordering preservation in case of stream merging is not addressed. In a *fair* merge of streams, any kind of insertion could occur, so the problems of overlapping we pointed out could reemerge.

6 A Reverse model to Delegation

We now present another model, we will call the "receipe query" or "reverse" model, because it is intuitively the reverse way compared to delegation.

6.1 Receipe Query Model

Rather than delegating the message, we will ask another object for the "receipe" (method), i.e., the way how to answer to the message. In that case, the initial receiver of the message remains the current one and keeps the control of the computation. It is waiting for the proxy to send it back the method associated to the message.

6.2 Assumptions and Limitations

This model relies on some assumptions. Every object should understand the message "What is the receipe for that message?". This pattern must be primitive and default to the set of methods of any object. We could also introduce a special type of message to get a method's body. In both cases this causes some problems of uniformity.

[11] In Vulcan, a class model is adopted. An inheritance mechanism is proposed, implemented by a recopy of the inherited code. A delegation mechanism is also proposed in order to define delegation from an object to its components (parts). This was formerly introduced in [Yonezawa 83] and [Shapiro & Takeuchi 83].

The second point is how to send back the inherited method to the initial receiver. Without shared memory assumption, we cannot just send the address of the code. A solution to this second problem is to copy the code body of the inherited method and to send it back. This operation is rather expensive in terms of memory allocation. Another solution assumes that methods are themselves objects. Then the address of the selected method can be sent back. The initial receiver will then send a request of self-computation to this method and wait for acknowledgment of its termination. Unfortunately, the method object may need to access to a variable by sending a message to the initial receiver which is already locked. One more time it becomes difficult to avoid deadlocks. This model, as delegation, is not fully suitable to our needs.

6.3 Object-Based Inheritance

We discovered recently the existence of a model very close to ours. Object-based inheritance [Hailpern 86] [Nguyen & Hailpern 86] owns some analogy with an "autoload" library concept.

They consider two kinds of methods: *simple* methods and *inherited* methods. A *simple* method contains the actual code to be executed. An *inherited* method contains pointers to other objects which contain the code. Two types of messages are provided: usual type called a *request*, and *inheritance* used to access *inherited* methods. A copy of the body of the inherited method is sent back and evaluated in the name-space of the initial receiver. Message passing is synchronous, and messages are serialized to each object except for inheritances which could be processed in parallel because [12] memory is not affected. This model suffers from the problems pointed in the previous section.

7 The Copy Model

In order to avoid the problems met with the delegation scheme, we will briefly present another extreme alternative based on recopy. The advantages of this approach are the efficiency of execution and the atomicity of objects, thus avoiding the serialization problems encountered in the delegation model.

But two big deficiencies are to be considered. The code size is big because of all the recopies, specially in the case of very general objects to be inherited by a high percent of the objects population. The second deficiency is the staticity of inheritance, i.e., once copied, a change to the initial object won't affect the new one. It is possible to use some techniques to reduce these two weaknesses. We review them below.

[12]The methods are supposed to remain unchanged to avoid "observation" of side effects (see [Gifford & Lucassen 86]).

7.1 Memory Occupation

In languages with inheritance, there is usually a class called *Object*. This usual default superclass represents the most common behaviors and is the root of the inheritance tree. As a consequence, every object will recognize the methods specified by this *root* class, unless they are shadowed somewhere in the hierarchy.

We cannot use such a basic object in our copy model. If we do so, its code will be copied into every object. Thus basic (default) behaviors should be declared as global (or at least on each machine). Their invocation will rely on a default mechanism and not the usual inheritance scheme. As a consequence we loose the uniformity of inheritance. Even worst, the use of inheritance should be restricted to objects which have a small offspring (the objects that will inherit from them) otherwise the recopy scheme is too heavy. In other words, this inheritance strategy exists but cannot be used extensively. This is the main weakness of the copy scheme.

7.2 Updating Problem

An automatic updating mechanism can be easily added. Such a facility is used in the inheritance of the Flavors [Moon & Weinreb 80]. We assume that objects won't change too often, and again that their offspring is not too wide. (In the Flavors system, a change made to the *Vanilla* flavor, the root of the hierarchy, will start a recompilation of the entire system!)

8 Conclusion

In this paper we reviewed and discussed some of the main strategies proposed for inheritance in object-oriented concurrent programming. We remarked that the attempts to increase flexibility of inheritance, by widely distributing it among the objects, complicates the serialization issues. A tradeoff between distribution for flexibility, and atomicity for synchronization appears to be fundamental as usual in parallelism. An alternative based on a synchronous approach has some weak points. We now need to give a stronger formalization to this study and explore good compromises between this strategies. A next paper to appear will present our current research in that field.

Acknowledgments

We would like to thank Pierre Cointe, Brent Hailpern, Henry Lieberman, Mark Miller, Etsuya Shibayama and other people for various fruitful discussions.

References

[Agha 85] Agha, G., Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge MA, USA, December 85.

[Bobrow & Winograd 77] Bobrow, D. G., T. Winograd, An Overview of KRL, A Knowledge Representation Language, *Cognitive Science*, Vol. 1, No 1, pp. 3-46, January 77.

[Bobrow & Stefik 83] Bobrow, D., M. Stefik, The LOOPS Manual, Xerox PARC, Palo-Alto CA, USA, December 83.

[Borning & Ingalls 82] Borning A., D. H. Ingalls, Multiple Inheritance in Smalltalk-80, *Proceedings of the AAAI-82*, pp. 234-237, Pittsburgh PA, USA, August 82.

[Briot 84] Briot, J-P., Instanciation et Héritage dans les Langages Objets, *(thèse de 3ème cycle)*, *LITP Research Report*, No 85-21, LITP - Université Paris-VI, Paris, France, December 84.

[Comer 84] Comer, D., Operating System Design - the XINU Approach, Prentice-Hall, London, UK, 84.

[Dahl et al. 70] Dahl, O., B. Myhrhaug, K. Nygaard, Simula-67 Common Base Language, *SIMULA information*, S-22 Norvegian Computing Center, Oslo, Norway, October 70.

[Deutsch & Schiffman 84] Deutsch, L. P., A. M. Schiffman, Efficient Implementation of the Smalltalk-80 System, *11th ACM Symposium on POPL*, pp. 297-302, Salt Lake City UT, USA, Jan 84.

[Fahlman 79] Fahlman, S. E., NETL: A System for Representing and Using Real-World Knowledge, MIT Press, Cambridge MA, USA, 79.

[Gifford & Lucassen 86] Gifford, D. K., J. L. Lucassen, Integrating Functional and Imperative Programming, *1986 Conference on Lisp and Functional Programming*, pp. 28-38, Cambridge MA, USA, August 86.

[Goldberg & Kay 76] Goldberg, A., A. Kay, Smalltalk-72 Instruction Manual, *Research Report SSL 76-6*, Xerox PARC, Palo Alto CA, USA, March 1976.

[Goldberg & Robson 83] Goldberg, A., D. Robson, Smalltalk-80 The Language and its Implementation, Addison-Wesley, Reading MA, USA, 83.

[Hailpern 86] Hailpern, B., Object-based Inheritance, *(position paper)*, *OOPSLA'86*, Portland OR, USA, November 86.

[Hewitt et al. 74] Hewitt, C. E. et al., Behavioral Semantics of Nonrecursive Control Structures, *IFIP'74*, Paris, France, April 74.

[Hewitt 76] Hewitt, C. E., Viewing Control Structures as Patterns of Message Passing, *A.I. Memo*, No 410, MIT, Cambridge MA, USA, December 76.

[Ingalls 78] Ingalls, D. H., The Smalltalk-76 Programming System Design and Implementation, *5th ACM Symposium on POPL*, pp. 9-15, Tucson AR, USA, January 78.

[Kahn et al. 86] Kahn, K., E. Dean Tribble, M. S. Miller, D. G. Bobrow, Objects in Concurrent Logic Programming Languages, *OOPSLA '86*, Special Issue of SIGPLAN Notices, Vol. 21, No 11, pp. 242-257, Portland OR, USA, November 86.

[Lieberman 81] Lieberman, H., A Preview of Act1, *AI Memo*, No. 625, MIT, Cambridge MA, USA, June 81.

[Lieberman 86a] Lieberman, H., Delegation and Inheritance - Two Mechanisms for Sharing Knowledge in Object-Oriented Systems, *3rd AFCET Workshop on Object-Oriented Programming*, J. Bezivin and P. Cointe (ed.), Globule+Bigre, No 48, pp. 79-89, Paris, France, Janvier 86.

[Lieberman 86b] Lieberman, H., Using Prototypal Objects to Implement Shared Behavior in Object Oriented Systems, *OOPSLA '86*, Special Issue of SIGPLAN Notices, Vol. 21, No 11, pp. 214-223, Portland OR, USA, November 86.

[Moon & Weinreb 80] Moon, D. A., D. Weinreb, Flavors: Message Passing In The Lisp Machine, *AI Memo*, No. 602, MIT, Cambridge MA, USA, November 80.

[Nguyen & Hailpern 86] Nguyen, V., B. Hailpern, A Generalized Object Model, *Proceedings of the Object-Oriented Workshop*, SIGPLAN Notices, Vol. 21, No 10, pp. 78-87, Yorktown Heights NJ, USA, June 86.

[Roberts & Goldstein 77] Roberts, R. B., I. P. Goldstein, The FRL Manual, *AI Memo*, No. 409, MIT, Cambridge MA, USA, 77.

[Steele 84] Steele, G. L., Common Lisp - The Language, Digital Press (DEC), Burlington MA, USA, 84.

[Shapiro & Takeuchi 83] Shapiro, E., A. Takeuchi, Object Oriented Programming in Concurrent Prolog, *New Generation Computing*, Vol. 1, No 1, pp. 25-48, Ohmsha, Tokyo, Japan - Springer-Verlag, Berlin, West Germany, 83.

[Stefik & Bobrow 86] Stefik, M., D. Bobrow, Object-Oriented Programming: Themes and Variations, *AI Magazine*, Vol. 6, No 4, pp. 40-62, Winter 86.

[Touretzky 86] Touretzky, D. S., The Mathematics of Inheritance Systems, *Research Notes in Artificial Intelligence*, Pitman, London, UK - Morgan Kaufman, Los Altos CA, USA, 86.

[Yonezawa 83] Yonezawa, A., On Object-Oriented Programming, *Computer Software*, Vol. 1, No 1, Iwanami Publisher, Japan, April 83.

[Yonezawa et al. 86] Yonezawa, A., J-P. Briot, E. Shibayama, Object-Oriented Concurrent Programming in ABCL/1, *OOPSLA '86*, Special Issue of SIGPLAN Notices, Vol. 21, No 11, pp. 258-268, Portland OR, USA, November 86.