Inheritance and Synchronization in Concurrent OOP

Jean-Pierre BRIOT* Akinori YONEZAWA

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

in Proceedings of the European Conference on Object-Oriented Programming (ECOOP'87), Lecture Notes in Computer Science, No 276, pages 32–40, Springer-Verlag, 1987.

Abstract

This paper discusses knowledge sharing (inheritance) mechanisms for Object-Oriented Programming (OOP) in the context of concurrent (distributed) languages. We review three different schemes: *inheritance*, *delegation* and *copy*. A fourth model, called *recipe-query*, is presented and all are compared and criticized. Techniques relying on the shared memory assumption are rejected. We point out the conflict between distributing knowledge among objects and the synchronization of concurrent objects.

Keywords:

objects, classes, actors, message passing, knowledge sharing, inheritance, subclassing, delegation, proxies, code sharing, copy, synchronization, atomicity, distributed systems.

1 Introduction

1.1 Knowledge Sharing for Modularity and Classification

Knowledge sharing (or inheritance) is a mechanism intensively used in OOP. Its basic idea is the reuse of object descriptions. After defining an object, we would often like to refine it into a more specialized one. Rather than defining a new object from scratch, we could define it by inheriting the specification of the previous one.

The first main advantage of knowledge sharing is the increase of *modularity* it brings to OOP rather than duplicating similar knowledge. The second advantage is the introduction of *classification* among objects. It is used to hierarchically (or almost hierarchically) structuring knowledge, and it makes knowledge searching more efficient [Touretzky 86].

From an implementation point of view, knowledge sharing is also a very good means to share code efficiently. However, *code sharing* is only one of the implementation model [Borning 86]. We will also present other strategies based on a copy approach.

1.2 Strategies, Assumptions and Compromises

The *Smalltalk inheritance* scheme is just one of the strategies to specify and implement knowledge sharing. But the preeminence of inheritance among the majority of OOP languages gave this term a generic meaning. *Delegation* [Lieberman 86b] is another strategy we discuss as well. Delegation is discussed in the framework of the *Actor* model of computation [Hewitt 76, Lieberman 81]. Other models presenting various knowledge sharing and classification mechanisms include *frames languages* (e.g., FRL [Roberts & Goldstein 77] and KRL [Bobrow & Winograd 77]) and *(parallel) semantic network languages* (e.g., NETL [Fahlman 79]).

We note that there are many different description and implementation strategies for knowledge sharing, but usually they are compromises between full dynamicity and full recopy. Also many systems rely on strong

*also LITP, University Paris VI, 4 place Jussieu, 75005 Paris, France

assumptions. The most usual assumption is the existence of shared memory. This allows to gain efficiency by following the inheritance links which are implemented by simple pointers. In contrast, we like to explore the strategies of inheritance in more general memory models, i.e., general distributed models, because we are concerned with concurrency and flexibility. (America's paper [America 87] on the same topic emphasizes reliability issues through the separation of inheritance from subtyping.)

1.3 Plan of the Paper

As the background of our discussion, we will first review the two main schemes used in object-oriented systems, namely inheritance and delegation, in Section 2 and Section 3 respectively. 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 examine and criticize it in the distributed memory context. We will show in section 4 that delegation can conflict with synchronization. Section 5 will then present an alternative to delegation based on a synchronous approach, but unfortunately this scheme (called the *recipe-query* scheme) also suffers from some deficiencies. The last scheme we will discuss in Section 6 is a *copy* approach. We will discuss the tradeoff between flexibility and efficiency in this approach and present some techniques to increase flexibility.

2 The Smalltalk Inheritance Scheme

The *Smalltalk inheritance* scheme (also often called *subclassing* and pioneered in Simula-67 [Dahl et al. 70]) was proposed in Smalltalk-76. This scheme has been followed by most OOP languages.

In this scheme, knowledge is shared among *classes*. A class can be declared as a *subclass* of another class, called the *superclass*. The subclass will then inherit the structure and properties of its superclass. This *subclass/superclass* hierarchy is usually represented as a tree, which is called the *inheritance tree*. (It could be a more general graph in case of *multiple inheritance*. We won't discuss it here.)

2.1 Variables and Methods

This inheritance scheme is strongly related to the *class* notion of Smalltalk and its implementation decisions. The designer of Smalltalk has separated the inheritance of the data-base of an object, called the *variable dictionary*, from that of its method-base, called the *method dictionary*. The first one is of the static nature whereas the second one of the dynamic nature.

The variable dictionary of an object is split in two parts. The class owns the ordered set of variables (or the index to them), while every *instance* of the class owns its corresponding values. The class code is shared by all its instances, thus an instance needs a link to its class and the isomorphism between variables and values must be maintained. As a consequence, the set of variables owned by the class should not vary over time and is determined at the time of defining the class. The inheritance of variables of a class is *static* and performed only once when the class is created. (The list of variables of a class inheriting from its *superclass* is set by concatenating the list of variables of the superclass before the variables of the class.)

On the other hand, the inheritance of the method dictionary is dynamic. When one activates a method which is not explicitly specified in the definition of the class, the lookup process for the method begins in the method dictionary of the class the object belongs to. Then it propagates along the subclass/superclass hierarchy until the method is found.

2.2 Flexibility versus Efficiency

This is one of the main tradeoffs in knowledge sharing. The Smalltalk inheritance scheme is a good compromise. The inheritance of the variable dictionary is static thus efficient, but some flexibility is lost unless there is an automatic updating mechanism. (For instance in Smalltalk-80, when redefining a class, its subclasses may be recompiled if needed.) The link between a class and its superclass is "hard-wired," which is usually implemented as pointers for the efficiency purpose in the same way as the link between an object and its class. The inheritance of the method dictionary is dynamically performed at run time, which often slows down system performance. But this dynamicity gives us a greater degree of flexibility. Our main comment is that this inheritance scheme using hard-wired links is not suitable for distributed memory models.

3 Delegation

The *delegation* scheme has been proposed in the Act-1 language [Lieberman 81]. 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 (at runtime) delegate to its proxy the messages that it cannot recognize. This proxy can in turn delegate the message to its own proxy. The proxy will handle the message in place of the initial receiver of the message. The intuitive account might be: "I don't know how to respond to this message, can you respond for me?" [Lieberman 81].

3.1 Delegation versus Inheritance

In contrast to the Smalltalk inheritance scheme, in the delegation scheme the inheritance of the variable dictionary is dynamic. In fact, accessing (read or write) a variable in the variable dictionary is like invoking an accessing method associated with this variable. There is no merging of the data-bases as in the Smalltalk scheme. The delegation scheme enjoys the uniformity of the communication protocol: the delegation to the proxy of an object is performed by message passing, not by a system primitive through the hard-wired physical link (pointer). Thus delegation can be fully designed at the user-language level and it can be locally customized by the user easily. This scheme is independent of the assumption of shared memory and is perfectly suitable for distributed (memory and computation) models. It also allows full dynamicity and modularity.

In examining this scheme, two issues, namely efficiency and synchronization, must be addressed. Efficiency can be gained by interpreter optimization such as caching [Deutsch & Schiffman 84, Lieberman 86b]. But as we will see in Section 4, synchronization can conflict with recursion and the problem is not simple. There seems to be a tradeoff between distributing knowledge and synchronizing its access in concurrent models.

3.2 Access to Variables through Message Passing

In the delegation scheme, variable consultation as well as method activation should be performed through message passing because both can be delegated to another object (the proxy). Thus there is no direct variable reference, but an indirect access through message passing in order to be independent of variable location. Accessing to a variable in an object corresponds to a message sent to the object itself. The simplest and most natural way is to identify the name of the variable with the pattern (selector) of the message to access it.

3.3 Message Passing: From a Serial to a Parallel World

In order to present the message passing strategy in the delegation scheme, we will first restrict it to a functional and serial world. We will then move to a parallel world and point out the difficulty of synchronization which emerges. A simple example will help us to illustrate this point.

Thus we first identify message passing with function call as in most of OOP extensions of Lisp. Let `ask` denote the primitive to send a message and return the evaluation of the method activated. Then to consult the variable `x` is equivalent to perform a function call: `(ask self 'x 'get)`. To update the value of `x` is equivalent to: `(ask self 'x 'set` *new-value*`)`.

3.4 Implementation of Variables

We now need to explain how to implement variables, i.e., how the access messages are interpreted by the object owning the variables. We will present two strategies, with different levels of object *granularity*, i.e., size of objects.

3.4.1 Variables as Methods

The first strategy considers each variable as a method (with the same name as the variable) with an optional argument. This optional argument represents the new value in case of updating. Each *variable-method* is implemented by a closure to store and protect its value. The distinction between consulting and updating the variable can be made through the test of this optional argument. However we chose to explicitly specify the operation name (`get` or `set`) in the message for the readability purpose.

In CommonLisp, such variables could be defined with:

```
(defun make-variable-method (value)
   #'(lambda (operation &optional new-value)
         (if (eq operation 'set) (setq value new-value) value)))
```

In this implementation model, variables are directly invoked through method calls.

3.4.2 Variables as Objects

Another strategy, proposed by Henry Lieberman [Lieberman 86a], represents each variable by an object. Each such *variable-object* owns the two methods that handle two kinds of messages: one to consult the variable (`get`), and the other to update it (`set`). In contrast to the previous strategy, we need to explicitly redirect the access message to the variable-object. Now the *script* of any object (i.e., the behavior of the object when it receives a message) needs to distinguish a method invocation from a variable access. If no method is applicable, the script checks the existence of a variable with same name in order to redirect the accessing message to it, otherwise the unrecognized message is delegated to the proxy.

Table 1 shows in column 2 how to access a variable of an object through message passing to the object. This is common to both strategies. On the other hand, column 3 (message passing to the variable-object) is specific to this second strategy. All the discussion in the rest of the paper is independent of the strategy choice.

variable reference	*message to the object*	*message to the variable*
read `x`	`(ask self 'x 'get)`	`(ask x 'get)`
update `x` with *new-value*	`(ask self 'x 'set` *new-value*`)`	`(ask x 'set` *new-value*`)`

Table 1. Variable accessing through message passing.

(Remark that we omit here the description and the solution to the *self problem*, i.e., how to handle correct recursion, which is described in [Lieberman 86b].)

4 Delegation and Synchronization

4.1 Variable Access and Atomicity

The delegation scheme relies on message passing to access variables, as previously shown. The `ask` primitive using a function call model synchronizes such accesses. However in some concurrent models such as the Actor model, message passing is asynchronous and unidirectional. There is no implicit synchronization. As a consequence, variable access is no more atomic and could be mishandled.

To present this problem, we now consider the delegation scheme in a parallel world. The functional call (synchronous) `ask` primitive is replaced by the asynchronous `send` primitive. The optional `:reply-to` keyword allows the specification of a *continuation* or *customer*[1].

A simple example will illustrate a synchronization problem when accessing a variable. This example doesn't involve delegation but it shows that the requirement of the delegation scheme, namely accessing to variables through message passing, conflicts with an atomic access to variables.

[1] A *customer* is an object which encodes all the behavior necessary to resume a computation after reception of an answer to a question [Hewitt 76, Lieberman 81].

4.2 The Counter Example

Suppose that we define a simple object, named `counter`, which owns only one variable, named `contents`. We now explain how `counter` will increment its `contents`.

When `counter` is requested to increment, the first thing to do is the reading of its `contents` variable. Because message passing is unidirectional, we need to explicitly specify an object which will receive the value of `contents` and increment it. We dynamically create a *customer* object called `incrementor` dedicated to the incrementation. (Such a customer could also be automatically created by a compiler such as Scripter [Lieberman 83].) The `incrementor` object will receive the value of `contents`, add 1 to it, and then send a message to `counter` to update `contents` with that new value. Figure 1 shows the `counter` and `incrementor` objects, and their exchange of messages.

Figure 1. Incrementation of `counter`.

Table 2 shows the computation of the `incr` method. Four message passings are performed and they are ordered from (A) to (D) by their causality. (A) is the initial increment message (`incr`) sent to `counter`. We assume the value of `contents` is initially 0.

(A)	`(send counter 'incr)`	*from* top-level
(B)	`(send counter 'contents 'get :reply-to incrementor)`	*from* `counter`
(C)	`(send incrementor 'reply 0)`	*from* `counter`
(D)	`(send counter 'contents 'set 1)`	*from* `incrementor`

Table 2. Incrementation of `counter`: causality of messages.

4.3 Message Ordering Issue

Suppose that we send consecutively two `incr` messages to `counter`. We cannot ensure that the value of `contents` will be 2, because the `incr` method is not atomic. Messages can interfere during the processing of the `incr` method, because this method reduces to several message passings. As a consequence, the two `incr` computations may overlap and the first assignment may be shadowed by the second one. The second `incr` message (A') will create three new message transmissions (B') to (D'). If the second message (B') consulting the `contents` variable happens to reach `counter` before the first assignment message (D) caused by the first `incr` message, the first incrementation will be shadowed by the second one. Then the value of `contents` becomes 1. The problem is the ordering of messages which cannot be predicted.

4.4 Fairness and Recursion

In an asynchronous message passing model, 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 arbiter orders them. This arbiter must be *fair* [Clinger 81]. As a consequence, 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. Meanwhile we would like to handle recursive messages at first to regain the atomicity for variable access, but then it would conflict with fairness.

We may lock `counter` to avoid it to process some other messages before the end of the `incr` method computation. But the problem is not simple because, on the other hand, `counter` needs to accept the recursive messages (sent by itself) and even messages sent by local customers such as `incrementor`, while temporarily rejecting other incoming ones. It is possible to implement such a mechanism but the scheme becomes very complicated, which is opposed to its intuitive foundation.

A way is 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]. In this model, the order of message transmission from one object to another object is preserved in the message arrival order. We may also simulate a waiting mode with *insensitive actors* [Agha 84] in the new Actor model. However, in any case we cannot rely on synchronous calls in case of recursion, because we get a deadlock. If we use a synchronous approach, the delegation model should be replaced by another scheme, the *recipe-query* scheme, we present in Section 5.

4.5 The Replacement Actor Scheme

We notice that in the latest Actor model [Agha 85] there is no assignment of variables. In order to allow changing states, an actor must specify its *replacement actor* which could have a different state. As soon as this replacement takes place (using the primitive *become*), the replacement actor may begin to process the next message in the mail queue, thus allowing a pipelining of incoming messages. There is no delegation proposed in the Actor model of [Agha 85]. It seems complicate to combine this replacement scheme with delegation scheme because the replacement action should be delegated to proxies, but the state changing of an object and that of its proxies must be synchronized (through transition to insensitive actors) in a proper manner. The next section will show another strategy to synchronize delegation messages by explicitly ordering the messages through a manipulation of communication channels (streams).

4.6 Synchronization through Stream Manipulation

In the Vulcan language [Kahn et al. 86], the basic computation model is similar to the replacement Actor model of Agha. A "perpetual" object is implemented in a Concurrent Logic Programming Language [Shapiro & Takeuchi 83] as the illusion of a *perpetual process*, i.e., an *ephemeral* process that continually reincarnates itself into another process with the same functor and consuming the remainder of the message stream. A state change is represented by a new incarnation of the object 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 in order to provide a perpetual identity to the object.

A delegation scheme is proposed[2] in Vulcan. The explicit manipulation of the communication stream avoids the serialization problem we pointed out above. Recursion will operate on *self* so that any message sent to *self* during the delegation will arrive before those sent externally. This technique allows a good handling of recursion and delegation, because the communication channel is unique and explicitly declared. But the issue of ordering preservation in case of stream merging is not addressed. In case of a *fair* merge of streams, any kind of insertion could occur, so the problems of overlapping we pointed out could reemerge.

The delegation scheme is conceptually natural and flexible. However, the synchronization issue must be carefully taken care of because accessing variables is performed through message passing.

5 An Alternative Scheme to Delegation

We now present another scheme, which we will call the *recipe-query* or "reverse" scheme, because it intuitively works in the reverse way of the delegation scheme.

5.1 Recipe-Query Scheme

Rather than delegating an unrecognizable message, the receiver object will ask another object (proxy) for the "recipe," i.e., the method of how to answer to the message. In that case, the receiver object waits for the

[2] In Vulcan, a class model is adopted. An inheritance mechanism is 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 previously introduced in [Shapiro & Takeuchi 83] and [Yonezawa 83].

proxy to send back the method associated to the message. Then using that method, the original message will be processed. Figure 2 compares *delegation* and *recipe-query* schemes.

Figure 2. Delegation vs. Recipe-Query.

This scheme relies on some assumptions. Every object should understand the message: “What is your recipe 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 it decreases the uniformity.

The second assumption is concerned with how to send back the recipe method to the initial receiver. Without the shared memory assumption, we cannot just send the address of the code. A solution to this problem is to copy the code body of the recipe method and to send it back. Then the body is executed. This operation might be rather expensive in terms of memory allocation. Another solution assumes that methods are themselves objects. Then the address (or name) of the selected recipe object can be sent back. The initial receiver object will then send an “eval” request to this method and wait for acknowledgment of its termination. Unfortunately, the method object may need to access a variable in the receiver object by sending a message to the receiver object which has been already locked. Once again it is difficult to avoid deadlock. This scheme, as delegation, does not fully satisfy our needs.

If however, we are allowed to send the environment of the object with the “eval” request, variables could be accessed directly, and no deadlock will occur. Thus the above problem will be resolved. But the number of message transmission will become very large (to get all the variables distributed along the proxies-chain), which might not be appetizing to those who are seriously concerned with the current state of the art in computer architecture. Note that the number of message transmission can be reduced if the inheritance of variables is disallowed.

5.2 Object-Based Inheritance Scheme

We recently discovered the existence of a scheme which is very close to ours. The object-based inheritance proposed by Hailpern and Nguyen [Hailpern 86, Nguyen & Hailpern 86] has some similarity 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 (hard-wired links) to other objects which would contain the code. Two types of messages are provided: the usual type is called *request*, and the other is called *inheritance* which is 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 memory is not affected. Memory allocation problem seems to be significant in this scheme.

6 The Copy-Everything Scheme

In order to avoid the difficulties met with either delegation or recipe-query schemes, we can adopt a very simple scheme in which both variable dictionaries and method dictionaries of inherited objects are all copied and installed into the inheriting object when it is created or compiled. The advantages of this approach are the efficiency of execution and the atomicity of processing a message, thus avoiding serialization problems. This approach is usually chosen in static (compiled) OOP languages.

The two deficiencies associated with this approach should be pointed out. First, the code size becomes very large because of all the recopies, specially in the case of very general objects being inherited by a large number of objects. The second deficiency is the staticity of inheritance, i.e., once copied, a change to the initial object won't affect the new ones. It is possible to use some techniques to reduce these two weaknesses. We review them below.

6.1 Memory Occupation

In languages with inheritance, there is usually a class called `Object`. This usual default superclass represents the most common (or default) behavior 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 scheme. If we do so, its code will be copied into *every* object. To alleviate this problem, basic (default) methods should be declared as global (or at least on each machine). Their activation will rely on a default mechanism, not on the usual inheritance scheme. Then the uniformity of inheritance will be lost. One way is to store default knowledge about a message with the message itself, as suggested in [Lieberman 86b]. Anyway the use of inheritance should be restricted to objects which have a small number of light offsprings otherwise the copy-everything scheme is too heavy.

6.2 Updating Problem

An automatic updating mechanism can be easily added in order to remedy the second shortcoming. Such a facility is used in the inheritance of the Flavors system [Moon & Weinreb 80]. We assume that objects won't change too often, and again that their offsprings are not too many. (In the Flavors system, a change made to the *Vanilla* flavor, the root of the hierarchy, will start a recompilation of the entire system!). [Borning 86] proposes a definition of inheritance by augmenting a copy mechanism with a *constraint* mechanism to support classification and updating. Unfortunately solving the updating problem doesn't solve the memory occupation problem.

7 Conclusion

In this paper, we have reviewed and discussed the main strategies proposed for inheritance (knowledge sharing) in object-oriented concurrent programming [Yonezawa & Tokoro 87]. We remarked that the attempts to increase the flexibility of inheritance by widely distributing functions and knowledge among objects complicates the synchronization issues. The tradeoff between the distribution for the sake of flexibility and the atomicity for the sake of synchronization appears to be fundamental in concurrent (distributed) computation. Alternatives based on synchronous communication tend to restrict the degree of concurrency that is the primary concern in concurrent computation and increase the deadlock risks. More static approaches based on the copy scheme solve these problems but inheritance should be restricted to avoid memory overloading. Delegation seems to be the most promising approach for knowledge sharing in distributed systems, but we must be very careful in combining the requirements of the delegation approach with the necessity of synchronization. We now need to find good compromises which are derived from both the characteristics of application domains and the underlying architectural supports for object-oriented concurrent programming.

Acknowledgments

We would like to thank Pierre Cointe, Brent Hailpern, Henry Lieberman, Etsuya Shibayama, as well as the reviewers for various fruitful discussions or comments.
Part of this research was carried out while the first author was visiting RIMS at Kyoto University.

References

[Agha 85] Agha, G., Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge MA, USA, December 1985.

[Agha 84] Agha, G., Semantic Considerations in the Actor Paradigm of Concurrent Computation, *Seminar on Concurrency*, LNCS, Springer-Verlag, No 197, pp. 151-179, Carnegie-Mellon, Pittsburgh PA, July 1984.

[America 87] America, P., Inheritance and Subtyping in a Parallel Object-Oriented Language, *ECOOP'87*, P. Cointe and H. Lieberman (ed.), Springer Verlag, Paris, France, 15-17 June 1987.

[Bobrow & Winograd 77] Bobrow, D.G., Winograd, T., An Overview of KRL, A Knowledge Representation Language, *Cognitive Science*, Vol. 1, No 1, pp. 3-46, January 1977.

[Borning 86] Borning, A.H., Classes Versus Prototypes in Object-Oriented Languages, *Proceedings of the Fall 86 Joint Computer Conference*, pp. 36-39, November 1986.

[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 1984.

[Clinger 81] Clinger, W.D., Foundations of Actors Semantics, *Phd thesis*, *AI-TR-633*, MIT, Cambridge MA, USA, May 1981.

[Dahl et al. 70] Dahl, O-J., Myhrhaug, B., Nygaard, K., Simula-67 Common Base Language, *SIMULA information*, S-22 Norvegian Computing Center, Oslo, Norway, October 1970.

[Deutsch & Schiffman 84] Deutsch, L.P., Schiffman, A.M., Efficient Implementation of the Smalltalk-80 System, *11th ACM Symposium on POPL*, pp. 297-302, Salt Lake City UT, USA, Jan 1984.

[Fahlman 79] Fahlman, S.E., NETL: A System for Representing and Using Real-World Knowledge, MIT Press, Cambridge MA, USA, 1979.

[Hailpern 86] Hailpern, B., Object-based Inheritance, *(position paper)*, *OOPSLA'86*, Portland OR, USA, November 1986.

[Hewitt 76] Hewitt, C.E., Viewing Control Structures as Patterns of Message Passing, *AI Memo*, No 410, MIT, Cambridge MA, USA, December 1976.

[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 1978.

[Kahn et al. 86] Kahn, K., Dean Tribble, E., Miller, M.S., Bobrow, D.G., Objects in Concurrent Logic Programming Languages, *OOPSLA'86*, Special Issue of SIGPLAN Notices, Vol. 21, No 11, pp. 242-257, Portland OR, USA, November 1986.

[Lieberman 81] Lieberman, H., A Preview of Act1, *AI Memo*, No. 625, MIT, Cambridge MA, USA, June 1981.

[Lieberman 83] Lieberman, H., An Object-Oriented Simulator for the Apiary, *Proceedings of the AAAI-83*, pp. 104-108, Washington DC, USA, August 1983.

[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, January 1986.

[Lieberman 86b] Lieberman, H., Using Prototypical 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 1986.

[Moon & Weinreb 80] Moon, D.A., Weinreb, D., Flavors: Message Passing In The Lisp Machine, *AI Memo*, No. 602, MIT, Cambridge MA, USA, November 1980.

[Nguyen & Hailpern 86] Nguyen, V., Hailpern, B., A Generalized Object Model, *Proceedings of the Object-Oriented Workshop*, SIGPLAN Notices, Vol. 21, No 10, pp. 78-87, Yorktown Heights NY, USA, June 1986.

[Roberts & Goldstein 77] Roberts, R.B., Goldstein, I.P., The FRL Manual, *AI Memo*, No. 409, MIT, Cambridge MA, USA, 1977.

[Shapiro & Takeuchi 83] Shapiro, E., Takeuchi, A., Object Oriented Programming in Concurrent Prolog, *New Generation Computing*, Vol. 1, No 1, pp. 25-48, Ohmsha, Tokyo, Japan - Springer-Verlag, Berlin, West Germany, 1983.

[Touretzky 86] Touretzky, D.S., The Mathematics of Inheritance Systems, *Research Notes in Artificial Intelligence*, Pitman, London, UK - Morgan Kaufman, Los Altos CA, USA, 1986.

[Yonezawa 83] Yonezawa, A., On Object-Oriented Programming, *Computer Software*, Vol. 1, No 1, Iwanami Publisher, Japan, April 1983.

[Yonezawa et al. 86] Yonezawa, A., Briot, J-P., Shibayama, E., 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 1986.

[Yonezawa & Tokoro 87] Yonezawa, A., Tokoro, M., (ed.), Object-Oriented Concurrent Programming, MIT Press, Cambridge MA, USA, May 1987.