Objects - A Fresh Look

KENNETH M. KAHN

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

Abstract

Actors and object-oriented programming have survived for about twenty years without significant changes in the basic conception of what an object or actor is. Looking at object-oriented programming from the concurrent logic programming framework leads one to question the very basic assumptions of objecthood. Need there be only a single port through which all messages to an object arrive? Or can there be several input ports corresponding to different capabilities or viewpoints? When a message is put on a port why can't there be more than one listener jointly reacting? What is the identity of an object and why do we think it persists over time? Would it be useful to be able to transfer communication ports? Perhaps a busy object should be able to clone itself and share the incoming requests and later rejoin.

All of these ways of stretching the notion of objecthood are expressible in a straight-forward and natural manner in concurrent logic programming languages. Concurrent logic programming techniques have been developed and are in common use which exploit these extra degrees of freedom. These techniques are presented along with a discussion of their importance and utility. The lesson which I hope object-oriented programmers will draw from this paper is that there is novel, relevant, and useful research going on the concurrent logic programming community.

1 Background

In 1983 Shapiro and Takeuchi published "Object-Oriented Programming in Concurrent Prolog" [ST83]. They presented concurrent logic programming techniques which enable one to define long-lived agents which exchange messages in a manner very similar to actors. They showed not only how one can build actors [Agh85] but also introduced an important new concept to object-oriented programming which they called "incomplete messages", i.e. messages which are only partially specified by the sender.

This work directly inspired work on high-level concurrent object-oriented programming languages such as Vulcan [KTMB87], Mandala [FAK*84], Polka [Dav88], and

A'UM [YC88]. All of these languages can be seen as different attempts to package the object-oriented programming techniques of concurrent logic programming into programming language abstractions. [Els88] is a similar attempt for a Prolog with freeze. What these languages gained over the underlying concurrent logic programming languages is the ability to concisely and directly define classes and methods. What they lost however was the simplicity and flexibility of Concurrent logic programming.

This paper explores some of the possibilities that the flexibility of concurrent logic programming enables that these languages cut off. Vulcan and A'UM, for example, automate many-to-one communication while losing the ability of the underlying framework to directly describe many-to-many communication and multiple input ports for a single agent.

An important theme in this paper is that by shifting one's conceptual vantage point, a new set of programming idioms become available. These idioms were technically possible in the old framework and probably quite useful but they are not used in common practice. Shifts in conceptual frameworks (e.g. from traditional object-oriented programming to concurrent logic programming) changes what is naturally motivated, coherent, and has a simple basis. Additionally language implementors tune implementations for expected use. For example, the kernels of concurrent logic programming implementations efficiently support multiple input ports. Were a programmer using an object-oriented language tuned for single unshared input ports to attempt to explore multiple shared input ports he would be forced to program in a clumsy and relatively inefficient manner. Most object-oriented languages, however, are well-tuned for many-to-1 communication which is clumsy and ackward in concurrent logic programming. This paper is not about which approach is better, nor the very interesting question of whether a hybrid could be designed which dominates either approach, but what the object-oriented programmers can learn from what has been discovered in the concurrent logic programming community.

2 Concurrent Logic Programming

Concurrent logic programming can be viewed as a model of computing based upon ephemeral asynchronous agents communicating by posting equality constraints upon shared variables. The variables range over a domain of trees. In this framework, an agent receives a communication, communicates with some other agents, generates new agents and terminates. The sending agent does not exsist long enough to receive a reply, instead the agents it generates receive any replies.

There are quite a number of languages which are based upon this model (for example, (Flat) Concurrent Prolog [Mie84], (Flat) Guarded Horn Clauses [Ued85], (Flat) Parlog [CG86], Strand [Lim89], the CC languages [Sar89], Oc [Hir84], P-Prolog [YA86], and Fleng [Nil88]). In order to present programs, a specific language needs to be used. Strand [Lim89] was chosen because its smallness and simplicity. Strand is the weakest language in this class and yet is capable of expressing all the programming techniques in this paper.

The syntax of programs used in this paper is as a collection of Strand clauses which have the following form:

`<head>` ← `<guard>` | `<body>`.

The head of a clause has the form $\texttt{name}(\texttt{term}_1,\ldots,\texttt{term}_n)$ where $\texttt{term}_i$ is either a variable or an uninterpreted function of terms. The guards are simple tests such as arithmetic relations (e.g. $\leq$) and type tests. If there are no tests then "|" is omitted. The body consists of assignments (`<variable> := <term>`) and agent creations $\texttt{name}(\texttt{term}_1,\ldots,\texttt{term}_n)$. Lists have the syntax $[\texttt{term}_1,\ldots,\texttt{term}_n|\texttt{tail}]$. If `tail` is [] then it and the preceding "|" may be omitted.

The state of a Strand computation is a set of agents (sometimes called processes or goals). The behavior of each agent is defined by a set of clauses. A clause specifies a pattern to match against the arguments of the agent and a body. The body specifies assignments and agent creations that are to be performed if the body is executed. If an agent can be matched by multiple clauses then any one of them is chosen, the agent is removed, and the body of the clause executed. The assignment operation in Strand permits only a single assignment of a variable to a term. It also allows variables to be assigned to variables thereby unifying them.

It is instructive to paraphrase the computation model in object-oriented terminology. The name of the term in the head of a clause is the name of a class. Agents are very-shorted lived objects (they exist only for one message response). The arguments of the agent correspond to its state and its communication ports. All the clauses with the same name are the methods of that class. Variables are communication ports (or actor mailboxes). Only a single message may be placed upon a port (this is how to view "`Port := Message`"). Terms are messages. Two ports may be "fused" so that a message to either port is sent to the other port. (One can alternatively view this as creating a transparent forwarder from one port to another.) Ports are first-class entities in that they may be included in messages, shared, and so on. Unlike traditional object-oriented programming conditions upon any subset of the ports on an agent may specify the selection of "methods". Unlike object-oriented programming, there is no direct support for inheritance. This is discussed later.

3 Concurrent Logic Programming as Object-oriente Programming

We first consider how one can build a long-lived object out of ephemeral agents that only survive a single computation step. The basic idea is to define an agent so that upon receiving a request, it creates agents and does assignments to service the request, *and* it creates a new agent to deal with subsequent messages. The simplest structure to use for sequences of incoming requests is a list or stream. An incoming message typically is a pair consisting of a request and a port for subsequent requests. A paradigmatic program for defining objects servicing a single stream of incoming messages is presented in Program 1. We call recurrent agents which serve a stream of messages "objects". In real programs written in this style (e.g. Program 2) $\texttt{request}_i$ is a term, `State` and `NewState` is spread out, and $\texttt{service\_request}_i$ is coded inline.

In Program 2 simple bank accounts are defined which accept requests to deposit and

```
...
agent([request_i | In], State) ←
    service_request_i(State, NewState),
    agent(In, NewState).
...
```

Program 1: Paradigmatic Object-Oriented Program

withdraw money and query for the current balance.[1] When an account agent notices that the port corresponding to its first argument is a list whose head is a deposit request, it can be removed and a new account agent created which watches the tail of the list, which has a balance which is the sum of the original agent's balance and the amount deposited, and the other state variables (name, account number, and address) are the same as those of the original agent. Withdrawals are handled similarly, except for testing whether the account will be overdrawn. A balance query is expected to be an incomplete message, i.e. `Reply` is expected to be a port upon which a reply will be sent.

A client of an account object make its requests by placing a message on a port which is the first argument of the account. Multiple uncoordinated clients cannot share a port since the first to send a message will have "used it up". (In Strand ports are really write-once variables. In other concurrent logic programming languages they can be written multiple times but only if the values written unify (i.e. are consistent).) There are many ways of attaining many-to-1 communication in the framework of concurrent logic programming. All of these methods are fundamentally awkward, especially when compared with actors or objects that support many-to-1 communication as a primitive notion. The simplest means of attaining many-to-1 communication is to use binary stream merging agents. Such agents are simple to define as in Program 3. Using `merge` one can merge the requests of multiple clients into a single stream to be serviced by an object. For example, Program 4 creates an account with two clients.

4 Beyond Traditional Objects

The programming technique illustrated in Program 1 together with the use of `merge` to share access to objects provides a functionality very similar to actors. As with actors, recurrent agents are encapsulated and concurrent. Instead of sending messages with continuation actors, incomplete messages (i.e. messages with continuation ports) are sent.

[1] The syntax of examples in this paper is often very verbose. There many syntactic extensions which can greatly increase the conciseness of these programs (e.g. [KS88]). Such a syntax is not used in this paper for clarity and simplicity. A pictorial syntax for these languages is being developed by the author which shows promise of alleviating this clumsiness.

```
account([deposit(Amount) | In], Balance, Name, Number, Address) ←
   NewBalance is Balance+Amount,
   account(In, NewBalance, Name, Number, Address).
account([balance(Reply) | In], Balance, Name, Number, Address) ←
   Reply : = Balance, account(In, Balance, Name, Number, Address).
account([withdraw(Amount, Reply) | In], Balance, Name, Number, Address) ←
   Balance ≥ Amount |
   Reply : = ok,
   NewBalance is Balance − Amount,
   account(In, NewBalance, Name, Number, Address).
account([withdraw(Amount, Reply) | In], Balance, Name, Number, Address) ←
   Balance < Amount |
   Reply : =
   overdraw_attempt(Amount, Balance),
   account(In, Balance, Name, Number, Address).
```

Program 2: Definition of Simple Bank Accounts

```
merge([X | Xs], Ys, Zs) ←
   Zs : = [X | ZsTail], merge(Xs, Ys, ZsTail).
merge(Xs, [Y | Ys], Zs) ←
   Zs : = [Y | ZsTail], merge(Xs, Ys, ZsTail).
merge([], Ys, Zs) ← Zs : = Ys.
merge(Xs, [], Zs) ← Zs : = Xs.
```

Program 3: Binary Merge

```
test_shared_access ←
   account(AccountIn, 100, 'John Doe', 0259, '12 Main St.'),
   merge(Access1, Access2, AccountIn),
   client1(Access1, ...),
   client2(Access2, ...).
```

Program 4: Creation of an Account with Two Clients

As with actors, there is a single private input port for account agents. Unlike actors, however, we are explicitly programming the communication mechanism and consequently can easily express variations which include multiple input ports, sharing and transferring input ports, explicit buffering, blocks of message blocks, and non-linear communication structures.

4.1 Multiple Input Ports

We have already seen an example of an object which has multiple input ports – merge. It has two input ports and one output port. All it does is copy requests on either input port to its output port. Clearly, more sophisticated mergers which filter or translate certain requests or which give priority to certain clients can be defined as variants of the basic merge program.

Multiple input ports can also be used to implement different client *capabilities*. For example, the account program can be extended to have an additional input port to which only system administrators have access. This is illustrated in Program 5.

```
account(In, [new_owner(NewName) | AdminIn], Balance, Name, Number, Address) ←
   account(In, AdminIn, Balance, NewName, Number, Address).
account(In, close_account, Balance, Name, Number, Address).
...
account([deposit(Amount) | In], AdminIn, Balance, Name, Number, Address) ←
   NewBalance is Balance+Amount,
   account(In, AdminIn, NewBalance, Name, Number, Address).
...
```

Program 5: A Bank Account with a System Administration Port

Another use of multiple input ports is to support specialized protocols with different clients. On one port messages may just be numbers interpreted by the receiver as the time, while on another port numbers may be interpreted as interest payments. Alternatively the different ports may correspond to different viewpoints of the recipient and it may respond differently to identical messages depending upon which port they arrive. Consider, for example, a balance sheet which has three ports corresponding to whether the view is for taxes, shareholders, or internal decision making. When queried for, say, total assets it might use purchase prices with different depreciation schedules for the first two ports and estimates of current market value for the third one. Note that each client of such a three-port balance sheet may have variables enabling it to send messages on one, two, or all three ports of the balance sheet.

In [KK89] multiple ports are use to attain secure communication. Agents had ports which were connected to trusted agents and external ports which were not. Money, third-party authentication, and third-party secrecy were implemented using agents with both secure and insecure ports.

In systems in which objects can have only one port, there are two ways to attain the functionality of multiple ports. In one scheme, there is a single-ported object which has all the state and behavior of multi-ported objects and there are message forwarding objects for each port. These message forwarders tag every incoming message with a token indicating their port type and passes it along to the object. Only these message forwarders have access to the "real" object, all the clients make requests via these forwarders. Another scheme, is to distributed the state and behavior of each port into separate objects and then build an internal protocol between these different objects to maintain consistency. It is worthwhile to note that concurrent logic programs frequently have multiple input ports and one rarely sees object-oriented programs emulate multiple ports using either of these two schemes.

4.2 Shared Input Ports

Another kind of flexibility inherent in concurrent logic programming is that a port can be shared by multiple agents. An agent frequently places a message upon a port upon which multiple agents are waiting. Sometimes the message is intended to be "broadcast" to a select group (e.g. the next simulation time, the change in location of the object carrying the objects in the group, etc.). Sometimes it is used to request an action of a group (e.g. move to the left, suspend computation, etc.). Sometimes the listeners on a port deal with different aspects of a request or different subsets of possible requests. For example, a `move` message may cause one listener to redisplay itself at a new location, another listener may move to keep a minimal distance to the first listener, while a third may log certain activities for playback and revision.

While shared input ports are commonly used in concurrent logic programs, there are difficulties placing incomplete messages on ports with multiple listeners. The problem is that there may be multiple responses to the same incomplete message. If the intent is to collect all the responses and the client knows the number of listeners then the different variables in the incomplete message can be the responsibility of different listeners. If the number of responses is dynamic, "short circuits" described later can be used.

Again it is possible to emulate this kind of behavior in an object-oriented program by creating a distributor object which upon receiving a message iteratively sends the message out to a list of recipients. This is sometimes done in object-oriented programs but it is clumsy and is not sufficiently abstract. There are many possible algorithms for distributing messages with different time/space/communication tradeoffs. By directly providing the abstraction of sharing ports the implementation is free to implement it as is appropriate for the situation. This same kind of argument argues for a primitive many-to-1 communication abstraction as is the case in object-oriented programming but not in concurrent logic programming.

4.3 Short Circuits

Short circuits (originally due to Takeuchi and discussed in length in [SWKS88]) is a widely-used concurrent logic programming technique that maintains a ring of agents where each agent has a port to its left and right neighbor. They can be used directionally to collect values by a simple protocol where each agent sends to the right its

contributions together with those it receives from the left. The rightmost agent will receive the contributions from every member of the ring. Any agent can splice in new agents into the ring as well as splice itself out. Program 6 presents a paradigmatic short-circuit program. An agent ready to dump its state waits for a "place holder" for the results from the left (called FromLeft) and fills in its place holder for a contribution (called ToRight) which it had previously sent to the right. It also generates a new place holder for the next phase (Right:=[_|_]). When requested to splice a new agent into the circuit a new port Middle is used as the right connection of one agent and the left connection of the other. When an agent leaves a short-circuit it simply connects its left and right ports.

```
agent(dump, [FromLeft | Left], [ToRight | Right], State) ←
    ToRight : = [State | FromLeft],
    Right : = [_ | _],
    agent(normal, Left, Right, State).
agent(splice, Left, Right, State) ←
    agent(normal, Left, Middle, State),
    agent(normal, Middle, Right, State),
    Middle : = [_ | _].
agent(short, [FromLeft | Left], [ToRight | Right], State) ←
    ToRight : = FromLeft,
    Left : = Right.
...
```

Program 6: A Paradigmatic Short Circuit Program

Short circuits can be used to repeatedly collect values from a group of objects. If there is a short circuit between all of the listeners to a shared port, then incomplete messages can be handled by the rightmost object and the short-circuit protocol illustrated in Program 6. Short circuits have also been used for detecting stable properties of concurrent computations [SWKS88].

Once again one can program short circuits in an object-oriented programming language. It is not, however, current practice. It is clumsier to remove a member of a circuit in the object-oriented framework. One must construct a forwarding object to remain in the ring. The concurrent logic programming version relies, instead, on the ability to unify the communication ports.

4.4 Explicit Manipulation of Ports

Short circuits are an example of the more general ability in concurrent logic programming to explicitly manipulate communication ports. The flexibility that results is commonly exploited as described below.

4.4.1 Logging, Transfer, Sharing, Merging, Etc.

One class of useful programming techniques transfer and stores communication structures. The simplest technique entails the retention of a port for logging, debugging, analysis or the like. It is a minor variant of the basic object-oriented programming technique in Program 1 and is illustrated in Program 7. To create a logging object an agent with n arguments is created and its only clause recreates it with n+1 arguments. The extra argument (`OriginalIn`) is the list of all the requests received by the object.

```
agent(In, State) ←
   agent(In, In, State).
...
agent([request_i | In], OriginalIn, State) ←
   service_request_i(State, NewState),
   agent(In, OriginalIn, NewState).
...
agent([history(Response) | In], OriginalIn, State) ←
   Response := OriginalIn,
   agent(In, OriginalIn, NewState).
```

Program 7: Paradigmatic Logging Program

In order to implement logging in object-oriented languages in which the behavior of objects is given as a collection of methods each method needs to be extended to store away the incoming message. In languages with forwarders, a forwarder can be defined which stores the message and forwards it. In the concurrent logic programming framework it is effortless to save a record of incoming messages since they are already explicitly in a data structure. Logging is just maintaining a pointer (`OriginalIn`) to that structure. In ordinary programs there are no pointers to the root of the structure so the storage is reclaimable by garbage collection. In a sense concurrent logic programming languages already pay the price for logging in that an explicit data structure is built for repeated communication regardless of whether logging is performed or not.

Ports can also be transferred between objects. An agent may terminate and transfer its input ports to another agent. For example, a technique for updating the program of a running object is illustrated in Program 8. The idea is that an object can be updated by sending it a `terminate` request. Then a new object with the state of the terminated object *and its input port* can be created. The ability to terminate objects is useful for evolving systems, but for modularity and security reasons should not be widely available. This can be remedied by making the ability to terminate objects a capability implemented by using a privileged input port. An example of this is illustrated in Program 9 where rectangles which were implicitly black are converted to rectangles with an explicit color parameter.

```
agent([terminate(StateReply, InReply) | In], State) ←
    StateReply : = State,
    InReply : = In.
```

Program 8: Clause to Dump and Terminate an Agent

```
rect(In, [terminate(InR, PrivInR, XR, YR, HR, WR) | PrivIn], X, Y, H, W)
    InR : = In,
    PrivInR : = PrivIn,
    XR : = X,
    YR : = Y,
    HR : = H,
    WR : = W.
...
colorize_rect(PrivInToRect, PrivInToColorRect) ←
    PrivInToRect : = [terminate(In, PrivIn, X, Y, H, W) | PrivInToColorRect],
    color_rect(In, PrivIn, black, X, Y, H, W).
```

Program 9: Updating a "Running" Object

4.4.2 Buffering of Messages

Takeuchi and Furukawa [TF87] explored programming techniques which give the programmer control over the degree of message buffering. Many object-oriented systems (and actor systems) rely upon unbounded buffering of messages. The programming techniques presented in this paper are also in this class since the sender of a message can always grow the list of messages. What is normally thought of as a buffer of messages is, in the concurrent logic programming framework, the difference between the pointer to the list held by the consuming process and the last tail written. Unlike conventional buffering these buffers can be shared (i.e. there can be multiple readers) since storage is not explicitly recycled and instead garbage collection is relied upon.

Unlike conventional object-oriented frameworks, concurrent logic programs explicitly describe bounded buffers when appropriate. The basic technique is illustrated in Program 10. The idea is that the sender of a message suspends until a cons cell is created by the receiver. The sender then places its message in the head of the cons. The size of the buffer can be initialized as in Program 11. The size can be changed dynamically. This scheme can be adopted for many-to-1 buffered communication by making the merge processes also wait for their output stream to become a cons.

```
receiver([request | In]) ← |
    In := [_ | _],
    service_request,
    receiver(In).

sender([Message | Out]) ←
    Message := request,
    sender(Out).
```

Program 10: Paradigmatic Bounded Buffer Program

```
start ←
    Buffer = [_, _, _ | _],
    receiver(Buffer),
    sender(Buffer).
```

Program 11: A 3-Long Message Buffer

4.4.3 Batches of Messages

The batch of messages technique provides two things: (1) the ability to send a batch of messages on a port to an object so that the messages are processed without any intervening messages and (2) the ability to lock and unlock a port to an object.[2] The technique is illustrated in Program 12. Program 13 uses this technique to implement an atomic transfer of funds between two accounts.[3]

```
agent([batch(Front, Tail, Acknowledgement) | In]) ←
    Acknowledgement : = ok,
    Tail : = In,
    agent(Front).
```

Program 12: Batching Messages Scheme

```
transfer(Account1, Account2, Amount, AccountTail1, AccountTail2, Reply) ←
    Account1 : = [batch(Front1, Tail1, Ack1) | AccountTail1],
    Account2 : = [batch(Front2, Tail2, Ack2) | AccountTail2],
    withdraw_then_deposit(Ack1, Ack2, Front1, Tail1, Front2, Tail2, Amount, Reply).

withdraw_then_deposit(ok, ok, Front1, Tail1, Front2, Tail2, Amount, Reply) ←
    Front1 : = [withdrawal(Amount, WithdrawalReply) | Tail1],
    deposit(WithdrawalReply, Front2, Tail2, Amount, Reply).

deposit(ok, Front2, Tail2, Amount, Reply) ←
    Front2 : = [deposit(Amount) | Tail2],
    Reply : = ok.

deposit(overdraw_attempt(Amount, Balance), Front2, Tail2, Reply) ←
    Front2 : = Tail2,
    Reply : = overdraw_attempt(Amount, Balance).
```

Program 13: Example Batching Messages

[2]This technique was discovered (or re-discovered) by Mark Miller, Eric Tribble, Jacob Levy and the author. It has been discussed on various open electronic mailing lists.

[3]A more sophisticated version of the program is necessary to avoid possible deadlocks.

In a concurrent framework this ability to specify a block of messages to be processed without intervening messages is quite useful. The ability to obtain and release locks on objects is equally important. As was the case with the programming techniques presented above, batching of messages and locking objects can be expressed in a traditional object or actor framework. Batching is not difficult to express if the language guarantees that messages sent to self are processed before any messages from the outside. The object could just send to itself each message in the batch. It is more difficult to handle a batch that is produced asynchronously as was the case in the example of transferring funds between accounts in Program 13. The ability to lock objects is tricky to implement in a traditional framework since the locked object is not processing external messages and yet it needs to process the message which releases the lock.

Batched messages can be thought of as a generalization of the linear stream of messages that is commonly used. The clients are able to express a hierarchical structure corresponding to the desired atomicity of response. There are other examples of useful communication structures which are non-linear. One way of achieving many-to-1 communication, for example, is to have each client extending their own private branch of the tree of requests to the object. If a client wants to share a port to such an object it just branches its own private branch, passing out one branch and continuing with the other.

5 Inheritance and Delegation

Concurrent logic programming points to many significant extensions and variations of the notion of object; it does not, however, provide any direct support for class inheritance. One remedy is what was done in Vulcan, A'UM, Polka, etc., which is to provide inheritance in a higher-level language. An alternative described below is to attain much of the same functionality by quite different means.

Actors typically use delegation rather than inheritance to share functionality. The metaphor is that an object may have a lawyer (or lawyers) to whom it can delegate requests. The lawyer is authorized to act on behalf of its client. Such a pattern of communication is illustrated in Program 14. When the delegating agent receives certain messages it delegates them by transferring its input port to its lawyer. The lawyer services the request and sends messages back to its client. The protocol ensures that the messages from the lawyer are serviced before any other requests. In general, the lawyer may need access to information or capabilities which the agent should not make widely available. In such situations the agent can pass its lawyer a privileged port. An alternative (due to Eric Dean Tribble) is to define the agent in such a way that it is prepared for private requests on its public port only during a delegation. This variant is used to implement secure delegation in Program 15.

Another approach to attaining the functionality of inheritance is through the use of port sharing. A bordered window with scroll bars can be implemented as three objects: a border, a plain window, and a scroll bar – each of which share an input port. When a request comes to, say, resize then each one responds accordingly.

```
agent(delegate, In, Lawyer) ←
   Lawyer : = [delegated(In, NewIn) | LawyerTail],
   agent(normal, NewIn, LawyerTail).

lawyer([delegated([request_i | ClientIn], NewClientIn) | In]) ←
   service_request_i,
   ClientIn : = [requests_to_client | NewClientIn],
   lawyer(In).
...
```

Program 14: Delegation Scheme

```
agent(delegate, In, Lawyer, PrivateMessagesExpected) ←
   Lawyer : = [delegated(In, NewIn) | LawyerTail],
   NewPrivateMessagesExpected is PrivateMessagesExpected+1,
   agent(normal, NewIn, Lawyer, NewPrivateMessagesExpected).

agent(normal, [private_request_i | In], Lawyer, PrivateMessagesExpected) ←
   PrivateMessagesExpected>0 |
   service_private_request_i,
   agent(normal, In, Lawyer, PrivateMessagesExpected).

agent(normal, [private_request_i | In], Lawyer, 0) ←
   refuse_request,
   agent(normal, In, Lawyer, 0).

agent([delegation_finished | In], Lawyer, PrivateMessagesExpected) ←
   NewPrivateMessagesExpected is PrivateMessagesExpected − 1,
   agent(In, Lawyer, NewPrivateMessagesExpected).

lawyer([delegated([request_i | ClientIn], NewClientIn) | In]) ←
   service_request_i,
   ClientIn : = [requests_to_client, delegation_finished | NewClientIn],
   lawyer(In).
```

Program 15: Secure Delegation Scheme

6 New Directions

Saraswat [Sar89] presents a family of concurrent constraint languages which include the concurrent logic languages discussed here. The framework is parameterized by the kinds of partial information which can be communicated between agents. This leads to two additional kinds of generalizations of objects. Messages may be incomplete in a much more flexible way. A message containing two variables, for example, may constrain one variable to be an integer which is greater than the other variable. The other kind of generalization is in new kinds of communication structures. [Sar89], for example, explores the use of bags (unordered collections) as communication structures. This seems especially promising for dealing with many-to-many communication, and provides communication functionality that subsumes that of Linda [Gel85] (since messages can be "incomplete").

7 Conclusions

This paper has attempted to present a large number of concurrent logic programming techniques which we believe to be useful in object-oriented programming. One purpose is to encourage researchers in object-oriented programming to reconsider the fundamental notion of objecthood. Objects can usefully and coherently have multiple input ports, input ports can be shared among objects, communication structures need not be linear, and explicit manipulation and transfer of ports can be a powerful programming capability. The very notion of the *identity* and permanence of an object is brought into question since there are situations in which an object may transform itself completely or even fork into multiple copies.

Another purpose of this paper is to encourage members of the object-oriented programming community to become familiar with the research on concurrent logic programming and concurrent constraint programming – especially to learn how basic issues in object-oriented programming are being addressed from a very different perspective.

8 Acknowledgements

I am very grateful for the comments on earlier drafts of this paper from Vijay Saraswat, Geoff Phipps, and Danny Bobrow.

References

[Agh85] G. Agha. *Actors: a model of concurrent computation in distributed systems.* PhD thesis, University of Michigan, 1985.

[CG86] K. L. Clark and S. Gregory. Parlog: parallel programming in logic. *TOPLAS*, 8(1):1–49, January 1986.

[Dav88] A. Davison. *Polka: A Parlog Object Oriented Language.* Technical Report, Department of Computing, Imperial College, London, 1988.

[Els88] N.A. Elshiewy. Modular and communicating objects in sicstus prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 792–799, 1988.

[FAK*84] K. Furukawa, Takeuchi. A., S. Kunifuji, H. Yasukawa, M. Ohki, and K. Ueda. Mandala: a logic based knowledge programming system. In ICOT, editor, *Proc of the Internation Conference on Fifth Generation Computer Systems*, 1984.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[Hir84] M. Hirata. Self-description of oc and its applications. In *Proc. of the Second National Conference of Japan Society of Software Science and Technology*, pages 153 – 156, 1984.

[KK89] K. Kahn and W. Kornfeld. *Money as a Concurent Logic Program.* Technical Report, System Sciences Laboratory, Xerox PARC, 1989.

[KS88] K. Kahn and E. Shapiro. *Logic Programs with Implicit State.* Technical Report, Weizmann Institute of Science, Rehovot, Israel, 1988.

[KTMB87] K. Kahn, E. Tribble, M. Miller, and D. Bobrow. *Research Directions in Object-Oriented Programming*, chapter Vulcan: Logical Concurrent Objects, pages 75–112. The MIT Press, 1987.

[Lim89] Artificial Intelligence Limited. *Strand88 User Manual.* Technical Report, Artificial Intelligence Limited, Greycaine Road, Watford, Herts, WD2 4JP, England, 1989.

[Mie84] Colin Mierowsky. *Design and Implementation of Flat Concurrent Prolog.* Technical Report CS84-21, Weizmann Institute of Science, December 1984.

[Nil88] M. Nilsson. Massively parallel implementation of flat ghc on the connection machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1031–1040, 1988.

[Sar89] Vijay A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989.

[ST83] Ehud Shapiro and A. Takeuchi. Object oriented programming in Concurrent Prolog. *New Generation Computing*, 1:25–48, 1983.

[SWKS88] Vijay A. Saraswat, David Weinbaum, Ken Kahn, and Ehud Shapiro. Detecting stable properties of networks in concurrent logic programming languages. In *Proceedings of ACM Symposium on PODC 88, Toronto*, August 1988.

[TF87] A. Takeuchi and K. Furukawa. *Concurrent Prolog: Collected Papers*, chapter Bounded Buffer COmmunication in Concurrent Prolog, pages 464–476. Volume I, The MIT Press, 1987.

[Ued85] K Ueda. *Guarded Horn Clauses.* Technical Report TR-103,, ICOT Technical report, June 1985.

[YA86] Rong Yang and Hideao Aiso. P-prolog: a parallel logic language based on exclusive relation. In *Third International Conference on Logic Programming*, pages 255– 269, 1986.

[YC88] K. Yoshida and T. Chikayama. A'um - a stream-based concurent object-oriented language. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 638–649, 1988.