Objects in Concurrent Logic Programming Languages

Kenneth Kahn, Eric Dean Tribble, Mark S. Miller, Daniel G. Bobrow

Knowledge Systems Area

Intelligent System Laboratory

Xerox Palo Alto Research Center

Abstract

Concurrent Prolog supports object-oriented programming with a clean semantics and additional programming constructs such as incomplete messages, unification, direct broadcasting, and concurrency synchronization [Shapiro 1983a]. While it provides excellent computational support, we claim it does not provide good notation for expressing the abstractions of object-oriented programming. We describe a preprocessor that remedies this problem. The resulting language, Vulcan, is then used as a vehicle for exploring new variants of object-oriented programming which become possible in this framework.

Introduction

The concurrent logic programming languages cleanly build objects with changeable state out of purely side-effect free foundations. As in physics, a causal chain of events with enough coherence over time can be viewed as an object with state.

The resulting system has all the fine-grained concurrency, synchronization, encapsulation, and open-systems abilities of actors. In addition, it provides unification, logic variables, partially instantiated messages and data, and the declarative semantics of first-order logic.

Abstract machines and corresponding concrete implementations support the computational model of these languages, providing cheap, light-weight processes, fast unification, and parallel architectures. The implementations provide the equivalent of "tail recursion optimization", so objects built on these foundations have the same complexity measure as objects implemented directly.

Since objects with state are not taken as a base concept, but are built out of finer-grained material, many variations on traditional notions of object-oriented programming are possible. These include object forking and merging, direct broadcasting, message stream peeking, prioritized message sending, and multiple message streams per object.

In exploring these issues, we found that the notation in which objects are expressed has some serious problems. It is significantly more verbose and awkward than that provided by traditional object-oriented programming languages. We remedy this by providing a preprocessor with syntax formalizing the cliches used for object-oriented programming in concurrent logic programming languages. We call the resulting language "Vulcan" because Vulcan is a fictional place characterized by a community of actors behaving logically.

Support for a programming paradigm

A programming paradigm is a programming style. Object-oriented programming, for example, is a programming style in which operations are grouped together with structured objects. Descriptions of operations and structure are collected together in classes which share operations and structural descriptions with their super-classes.

The support a programming system gives to a programming paradigm includes linguistic, semantic, execution, and environmental support.

Linguistic support.

The linguistic expression of a program should correspond well with the intentions of the programmer. The system should support clear and concise expression. Common programming cliches should be supported.

Clear semantics.

The basic constructs supporting a paradigm should have a simple clean semantics. The underlying semantics should be well-suited for both human understanding of programs and machine analysis and transformation of programs.

Execution support.

A system must be able to execute programs in the paradigm efficiently. A programmer should not be penalized unnecessarily for programming in a good style.

Environmental support.

The system should support the debugging of programs at the level of abstraction of the programming paradigm. If, for example, a system is supporting object-oriented programming then the tracing, editing and browsing of programs should be in terms of objects, methods, messages, classes, and instance variables and not the underlying implementation constructs.

In addition, if the system supports other programming paradigms these should be well-integrated.

Object-oriented programming in Concurrent Prolog

Shapiro and Takeuchi present Concurrent Prolog and a set of programming cliches for programming in an object-oriented style [Shapiro 1983a]. Here we summarize their findings and in the following sections evaluate and extend their work. We use Concurrent Prolog (CP) since it is a typical example of a concurrent logic programming language. Most of the ideas presented apply equally well to other concurrent logic programming languages.

Concurrent Prolog

As in Prolog, a program in Concurrent Prolog is a collection of Horn clauses. A Horn clause is a logical implication of the form

$\forall(x_1, \ldots x_k)\ A_0 \leftarrow A_1 \ \&\ A_2 \ \&\ A_n$

where the A's are *atomic formula*. n and k can be 0. There are no variables occurring in the A's other than x_1 through x_k.

In CP and Prolog, clauses have the syntax:

```
A0 :- A1, A2, ... An.
```

and variables are normally denoted by constants beginning with an upper case letter.

CP and Prolog programs can be read declaratively as sets of logical implications. A single clause reads as "for all x_1 through x_k A_0 is true if A_1 through A_n are all true". A clause in which n is zero is read as "for all x_1 through x_k A_0 is true". Clauses can also be read procedurally as "to solve a goal matching A_0, solve the subgoals A_1 through A_n".

Each of the A's is a *term* made from a *functor* and its *arguments*. The functor of the term, `plus(3, 4, X)`, is `plus`, and the arguments are `3, 4,` and `X`. The specification of a functor includes its arity, so `foo(X,Y)` and `foo(X,Y,Z)` are distinguished. Constants are 0-arity functors. Arguments are themselves terms. Sometimes the top-level terms (A's) are called *atomic literals* and the functors of the literals are called *predicate symbols*.

For Prolog, the list of clauses for the same predicate symbol (including arity) define a procedure. A procedure is interpreted as "to solve a goal, try the clauses sequentially until one is successful. If another solution is requsted, the procedure should try the remaining clauses.

CP extends the syntax of clauses by adding the commit operator "|". There is exactly one commit operator per clause and it either replaces one of the commas in the clause or is placed before A_1 or after A_n. A clause without "|" has the commit before A_1. One calls the consequent A_0 of a clause the *head*, the conjunction of atomic formulas before the commit operator the *guard* and the conjunction of formulas after the commit operator the *body*.

A CP clause:

```
H :- G1, ... Gm | B1, .. Bn
```

has a process or behavioral reading which says, "a process matching H, can be reduced to the system of processes B_1 through B_n if the guard processes G_1 through G_m successfully terminate". In CP atomic formula are treated as processes while in Prolog they are considered goals. A CP procedure is a collection of clauses for the same predicate. It is interpreted as "to reduce a process, commit, if possible, with one of the clauses and reduce the process to the processes described in the clause body". We say that a process can commit with a clause if the process unifies with the head of the clause and the guard successfully terminates. If there is more than one clause which can commit with a process then one is chosen non-deterministicly. If there are no clauses that can commit then it fails. Unlike Prolog, CP is not capable of backtracking or searching for all solutions. Correct CP implementations are sound but incomplete theorem provers (as are conventional Prologs).

An implementation of CP can be based upon fine-grained parallelism. The search for a clause which can commit can be performed in parallel. When a clause commits it stops the processing of the other clauses. This kind of parallelism is sometimes called limited or-parallelism. Even more important is the potential for and-parallelism in CP. And-parallelism is the concurrent execution of each CP process. It is called and-parallelism since the set of processes corresponds to a conjunction of atomic formula which must be true for the query to be true.

Unification in CP is extended to interpret read-only annotations on occurrences of variables (denoted by a "?" at the end of the variable name). An attempt to instantiate an uninstantiated variable with a read-only annotation causes the process to suspend. If no clause of a process can commit, but at least one clause suspended during the unification then the process is *suspended*. Efficient implementations of CP associate suspended processes with the variables which caused the suspension. No attempts are made to reduce a suspended process. A suspended process is activated only when the variable it is waiting for is instantiated. One can think about the read-only annotation as saying "don't use this occurrence of the variable until some other process running concurrently gives it a value". From the point of view of the declarative reading, the read-only annotations are ignored.

Quicksort in Concurrent Prolog.

```
/* Sort clauses have the form:
        sort(UnsortedList,SortedList) */

sort([],[]).

sort([H|T],Sorted) :-
        partition(T?,H,L,G),
        sort(L?,LS),
        sort(G?,GS),
        concatenate(LS?,[H|GS?],Sorted).

/* Partition clauses have the form:
        partition(UnsortedList,PartitionKey,
                        LesserList,GreaterList) */

partition([],X,[],[]).

partition([H|T],X,[H|L],G) :-
        H <= X | partition(T?,X,L,G).

partition([H|T],X,L,[H|G]) :-
        H >= X | partition(T?,X,L,G).

/* Concatenate clauses have the form:
        concatenate(FirstList,SecondList,
                                ConcatenatedList) */

concatenate([],X,X).

concatenate([X|Xs],Y,[X|Zs]) :-
        concatenate(Xs?,Y,Zs).
```

The first clause of `sort` provides a base case for recursion: an empty list is already sorted. The second clause reads: use the first element to partition the remainder of the unsorted elements. This provides a list of elements that will precede the first element in the sorted list, and a list of elements to follow it. Compute sorted versions of these two lists, then construct a single list using `concatenate`. The elements preceding the first unsorted element get concatenated with a list made up of the first element and the elements greater than it.

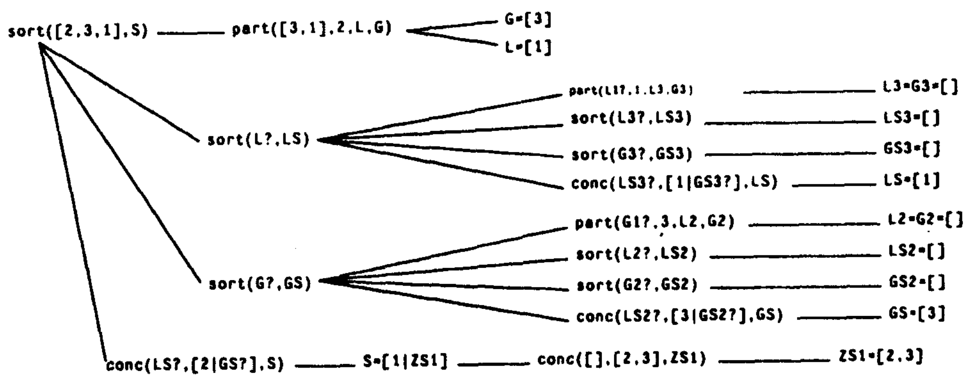
`Partition` has a similar base case. The second `partition` clause commits only if the first element of the `UnsortedList` is less than or equal to the `PartitionKey`. If so, it adds the element to the list of elements to precede the `PartitionKey`, then partitions the rest of the `UnsortedList`. The third clause operates similarly for elements greater than the `PartitionKey`. Notice that if the tested element equals the `PartitionKey`, one of the two work clauses is chosen non-deterministically.

`Concatenate` is a relation of three lists where the first two concatenate to form the third.

CP has been used successfully to do serious system programming and symbolic computing. An operating system and programming environment called Logix has been built upon a subset of CP called Flat Concurrent Prolog. [Silverman 85] The language has a simple and clean semantics. Implementations of CP provide very light-weight processes. Process synchronization is accomplished by use of the commit operator and the read-only annotations.

Objects in Concurrent Prolog

The ephemeral processes in the sorting example hardly posess the permanence and identity of objects. CP associates this identity with a communication channel carrying messages for consumption by an ephemeral process. When the process receives a message, it reduces to other processes. One of the processes has the same functor as the original and consumes the remainder of the message stream. CP represents state changes by incarnating that process with the new state instead of the old. This is so similar to tail-recursion that it is often referred to as that. Such continually reincarnated processes are called



This is a simplified diagram of the process spawn tree of the computation for quicksort. The nodes represent processes to be reduced, or bindings introduced by those processes.

perpetual processes. The communication channel passed along from process to process provides the identity of a perpetual process.

The communication channels of CP are simply shared logical variables. Since logical variables cannot be reset once they are set (instantiated), a communication variable must instantiate to a pair of the message transmitted and a new uninstantiated logical variable for further messages. A process waiting for a message suspends on the communication variable (via read-only annotations) until it becomes instantiated to such a pair, then executes the method appropriate to the first element of the pair. Once the pair contains a message, the new variable (second in the pair) must be used for further communication because the message part cannot be reset. This write-once property requires the recursive process creation scheme described above. Furthermore, multiple suppliers cannot put messages onto the same stream because one process would instantiate the message before the second, so the second process would fail. This problem will be dealt with later.

Since a communication channel - usually called a stream - identifies an object, the logic variable representing the channel is often treated and referred to as the pointer to the object. This engenders our variable naming convention: we name stream variables for the kind of process interpreting the message stream. We add the prefix 'New' to the name of the variable representing the future messages to an object (typically the variable used in the recursive call). Whenever examples include two different variables with the same name, numbers will be appended to their names to distinguish.

Consider the following example CP program. It implements a simple bank account defined to respond to the messages `deposit`, `withdraw`, and `balance`.

```
account([deposit(Amount)|NewAccount],
                        Balance,Name,...) :-
    /* Increase the balance by Amount */
    plus(Balance,Amount,NewBalance),
    account(NewAccount?,NewBalance?,Name,...).

account([balance(Balance)|NewAccount],
                        Balance,Name,...) :-
    account(NewAccount?,Balance,Name,...).

account([withdraw(Amount)|NewAccount],
                        Balance,Name,...) :-
    /* Subtract Amount from the balence unless the balance is
       too low. */
    Balance >= Amount |
    plus(amount,NewBalance,Balance),
    account(NewAccount?,NewBalance,Name,...).

account([withdraw(Amount)|NewAccount],
                        Balance,Name,...) :-
    /* Leave balance untouched and report attempt to
       overdraw */
    Balance < Amount |
    reportOverdrawn(Name,Balance,Amount,...)
    account(NewAccount?,Balance,Name,...).
```

The deposit clause increases the `Balance` by the given amount. The second clause simply instantiates the argument to the current `Balance`. The third clause can only commit if the account contains at least `Amount` in it. Otherwise the fourth clause reports the attempt to overdraw the account. The first two clauses each correspond to a method in an object-oriented programming system. The third and fourth clauses together correspond to an ordinary method. All the clauses have the same arguments. The first represents the input message stream. When the message stream of an account instantiates, it attempts unification with the first argument from each clause. Since all methods have the form

[Message|Variable], the actual Message determines which clause will successfully unify, so the message dispatches properly. Message can be any logical term and may include variables. Often those variables get used to return values, as in the balance clause.

Consider the following transactions. The extra arguments to the above definitions were extra information attached to an account. For simplicity, only the Balance will be shown now.

```
account(A?,100),
A = [deposit(100), balance(B), withdraw(200)
      | NewAccount].
```

The account process immediately suspends if it tries to run. The second process runs, instantiating A to [deposit(100), balance(B), withdraw(200) | NewAccount]. Now the first process proceeds, unifying [deposit(100), ...] with the first clause (none of the others can succeed). It reduces, leaving the following processes:

```
plus(Balance,Amount,NewBalance1),
account([balance(B), withdraw(200)
            | NewAccount],NewBalance1?).
```

Taking the most pathological route, assume that account keeps being reduced. The balance message drops out immediately. It succeeds, and the argument will be instantiated when the plus process completes. If any concurrrent operation attempts to use the value of that variable, it will suspend because the variable is read-only. Such is the case with the withdraw message. It unifies with the correct clause, but the guard tries to compare an uninstantiated variable with a number, so the comparison suspends. The account process that looks at NewAccount also suspends. Therefore the plus process will run since it is the only remaining active process. It computes the new balance. The process doing the withdrawal compares that balance to discover that the withdrawal is legal, and so finally withdraws the money.

This sequence of reductions demonstrated the pipelining ability, and the power of the simple read-only annotations.

The declarative semantics of a perpetual process is peculiar. Each clause can be read declaratively as stating constraints on possible histories of messages and the corresponding state. The first clause of account, for example, can be read as "a history of messages beginning with one matching deposit(Amount) is a valid balance history if the remainder of the history is valid in the state where the Balance is the sum of the previous balance and the Amount".

This peculiar declarative semantics can be a useful basis of a debugger which keeps a pointer to the history of messages and provides a convenient way to browse the message history.

Special features of CP Objects

Shapiro and Takeuchi claim that CP realizes objects in the sense of Hewitt's Actor model. [Clinger 1981] CP objects have internal state, can be operated upon only by message sending, and can exchange arbitrary messages. Any number of instances can be created from a definition of an object. Unlike Smalltalk, Flavors, Loops and the like, CP objects are not based upon call/return message passing. CP objects have the full control generality of actors. Call/return is just a particular pattern of message passing in CP [Hewitt 1977].

In addition to the full generality of actors, CP objects have special features not found in other object-oriented programming systems. The streams and processes of CP are side-effect free. Perpetual processes seem to change because a process terminates and causes a similar but different one to be created. This lack of side-effects leads to a simpler, clearer declarative semantics. The explicit manipulation of streams is more flexible than the system-supported message queues of actor systems [Clinger 1981]. Several objects can share a stream of messages, enabling direct broadcasting. A stream can be held onto to provide a message history to inspect. A stream of incoming messages is available for peeking ahead. A perpetual process can consume multiple streams, differing in cabability or priority.

Concurrency

The underlying model and motivation for CP is based upon large-scale concurrency and simple, yet adequate, synchronization primitives. CP defaults to concurrency. It achieves sequencing and synchronization with the commit operator and read-only annotations. Shapiro has shown CP to be adequate for solving classical synchronization problems like multiple readers and writers and addressing issues like starvation, fairness, and deadlock. [Shapiro 1983b] CP programmers have found the commit operator and the read-only annotations easy to use and reason about. Unification and commit are thought of as atomic transactions.

Unlike Smalltalk, Flavors, Loops and the like, CP objects are *active*. They can execute continually, though by convention they normally suspend when there are no messages to process. Like serialized actors, CP objects process messages one at a time, though unlike serialized actors the typical CP object processes messages in a pipeline fashion. The body of the clauses for an object by convention creates a new process to receive subsequent messages. This process is normally spawned concurrently with some of the method computation. Consider the first clause of `account`.

```
account([deposit(Amount)|NewAccount],
                          Balance,Name,...) :-
      plus(Balance,Amount,NewBalance),
      account(NewAccount?,NewBalance?,Name,...).
```

When a `deposit` message is in the front of the stream of incoming messages this clause commits and spawns a process to add `Amount` to `Balance` and a process to receive more messages. If the new `account` process receives a message before the `plus` process has terminated then the `Balance` will be the uninstantiated variable `NewBalance`. Since `NewBalance` is read-only, it can only be instantiated by the `plus` operation that runs concurrently with the recursive call. When another process actually tries to use `NewBalance` it will suspend until `plus` finishes.

Logic Variables

The usefulness of the logic variable has been repeatedly demonstrated in both Prolog and CP programming. Unbound variables in logic programming are "first class objects" in that they can be passed around and embedded in structures. In object-oriented programming in CP, this is frequently exploited by a technique called *incomplete messages*. Messages are sent which contain variables and typically the recipient binds those variables. This is the most common way of sending a message and getting a reply. The `second` clause of `account` is an example of this.

```
account([balance(Balance)|NewAccount],
                          Balance,Name,...) :-
      account(NewAccount?,Balance,Name,...).
```

The sender of a `balance` message requests the current `Balance` of an object. Unlike call/return message sending, the sender proceeds immediately. Frequently the users of the response from an incomplete message have read-only annotations to prevent those parts of the computation from using the uninstantiated variables of the response before the recipient fills them in.

Another use of logic variables in CP objects is to leave some state variables uninstantiated. We saw how this could occur naturally in the pipeline style of message processing. It is also possible to create objects with parts left uninstantiated. The following processes creates an `account` with an unknown `Balance` and use a `balance` message to initialize it to 300.

```
account(A?,X,joe,...), A=[balance(300) | A1].
```

This technique may be useful in creating objects which need complex initialization.

Verbosity.

The major shortcoming of object-oriented programming in Concurrent Prolog is its verbosity. Each method must at the very least repeat the names of the state variables in both the head of the method and in the tail recursive call. Each method must explicitly fetch the next method from the stream and then recur on the stream of remaining messages. Such tedious repetition easily results in subtle mistakes. The tail-recursive call requires a read-only annotation on the stream of remaining messages, for example. Without it, the process does not suspend, so unification non-deterministically applies a clause (anything unifies with a variable), setting the first element of the stream to the message for that clause, then executing its body. Logically, we queried, "What sequence of messages could objects of this class have, starting from the current state, and what would be the new state?" Later message sends to the object would likely fail because they would try to unify different message terms. All this from leaving out one question mark!

[Shapiro 1983a] addresses the inconvenient necessity of repeating the state variables by packaging up the entire state into a single term and using special predicates to access or create modified versions. Consider the differences between the following two ways of writing a `move` method for a window.

```
/* Version with multiple state variables */
window([move(NewX,NewY)|NewWindow],
         X,Y,Width,Height,Contents) :-
     eraseRegion(X,Y,Width,Height) |
     window([show|NewWindow],
         NewX,NewY,Width,Height,Contents).

/* Version with one state variable */
window([move(NewX,NewY)|NewWindow],State) :-
     eraseRegion(State),

setWindowState(xy,NewX,NewY,State,NewState)|
     window([show|NewWindow],NewState).

/* the relevant part of the definition of setWindowState */
setWindowState(xy,NewX,NewY,
     windowState(X,Y,Width,Height,Contents),
     windowState(NewX,NewY,Width,
                              Height,Contents)).
```

Without sophisticated compile-time optimizations, packaging requies significantly more time and space. Also, `setWindowState` and `getWindowState` clauses must be defined for each collection of state variables.

A Preprocessor Solution

A straightforward solution to the verbosity of object-oriented programming in CP is to build a preprocessor. User programs declare the state variables once and methods are written in a concise notation and expanded into ordinary CP. Vulcan is such a preprocessor. It formalizes the cliches used for object-oriented programming in Concurrent Prolog, and in doing so, reduces their verbosity.

Vulcan operates on clauses for classes and methods. Class clauses declare the names (and perhaps properties) of state variables for all the instances of the named class. Method clauses use the class declarations to expand into operations invoked by particular messages to the object.

```
class(window,[X,Y,Width,Height,Contents]).
```

declares that all window methods have a message stream, and all the named instance variables. It also generates a `make` clause for creating instances.

```
make(window,[X,Y,Width,Height,Contents],
                                    Window) :-
        /* This makes a process consuming the stream Window */
        window(Window?,X,Y,Width,Height,Contents).
```

A method for getting `the position of` a `window` can be defined as

```
method(window,position(X,Y)).
```

Vulcan recognizes references to the variables declared in the class, and generates code appropriately. The obvious expansion which is shown below directly unifies the arguments of the message with the appropriate state variables, just as the Vulcan code indicated. Later discussion shows that this only works if `X` and `Y` are terms and not streams to other objects.

```
window([position(X,Y) | NewWindow],
            X,Y,Width,Height,Contents) :-
        window(NewWindow?,
                  X,Y,Width,Height,Contents).
```

Since `position` has no body to expand, the clause just contains the head and the tail-recursive call. The preprocessor would actually generate unique names for variables rather than names like `NewWindow` that might conflict with programmer defined variables. These examples use the simpler names for readability.

The Vulcan expression for changing an object's state is `become((Var1,value1),(Var2,value2),...)`. This can be read as a parallel assignment statement. Thus, a `window`'s position could be moved by:

```
method(window,moveBy(DeltaX,DeltaY)) :-
        plus(X,DeltaX,NewX),
        plus(Y,DeltaY,NewY),
        become((X,NewX),(Y,NewY)).
```

which expands to

```
window([moveBy(DeltaX,DeltaY) | NewWindow],
            X,Y,Width,Height,Contents) :-
        plus(X,DeltaX,NewX),
        plus(Y,DeltaY,NewY),
        window(NewWindow?,
            NewX?,NewY?,Width,Height,Contents).
```

Message Sending

In CP one simply unfies a stream variable with a pair of a message and a new variable to send a message. The Vulcan equivalent is `send(ReceivingObject,Message)`. The pseudo-variable `Self` refers to the receiver of a message. Methods can treat `Self` as any other message stream.

Since any method for an object can refer to or change the object's state, `becomes` and messages to `Self` must be serialized. Vulcan assumes that those operations function on successive states of the object. All expressions lexically after a `become` or a send to `Self` refer to the new state. Vulcan achieves serialization by using message queues. The usual CP manner of sending messages to self is to "preset" the message stream on the recursive call. Vulcan packages any expressions lexically after a send to `Self` into a "continuation" method whose selector is preset onto the message stream after the earlier messages.

The above definition of `moveBy` doesn't erase or redisplay the `window`. A more realistic version is:

```
method(window,moveBy(DeltaX,DeltaY)) :-
        send(Self,erase),
        plus(X,DeltaX,NewX),
        plus(Y,DeltaY,NewY),
        become((X,NewX),(Y,NewY)),
        send(Self,show).
```

This translates to:

```
window([moveBy(DeltaX,DeltaY) | NewWindow1],
            X,Y,Width,Height,Contents) :-
        window([erase,privateMoveBy(DeltaX,DeltaY)
                    | NewWindow1?],
               X,Y,Width,Height,Contents).
```

```
window([privateMoveBy(DeltaX,DeltaY)
                | NewWindow2],
          X,Y,Width,Height,Contents) :-
      plus(X,DeltaX,NewX),
      plus(Y,DeltaY,NewY),
      window([show | NewWindow2?],
          NewX?,NewY?,Width,Height,Contents).
```

Messages sent after the become (show) are interpreted as occuring after the state change, and so are sent from the continuation method.

Immutable streams vs. pointers to mutable objects.

For object-oriented programming, CP fundamentally differs from languages like Smalltalk in that CP does not have pointers to objects. The message stream maintains the identity of CP or Vulcan objects. If two processes send different messages on the same stream, then the first process sends its message normally, but the second process usually fails because its message will typically not unify with the already-instantiated message variable. Two processes can only send messages to the same object by merging the streams of messages that they produce into the one stream consumed by the object. This is also called splitting the reference to the object. Passing out the contents of a state variable, or even a reference to Self, requires the splitting of the stream to the object. In CP, the programmer must split message streams explicitly to achieve object sharing. Neglecting to split a stream can lead to non-obvious bugs when multiple processes try to unify different messages with the front of the stream.

In general, a state variable can "point to another object" (i.e. contain a stream being consumed by a perpetual process) so the expansion must replace references to state variables in the method clause with one branch of the split of that state variable. The other branch becomes the new state variable. This gives the following definition of merge:

```
merge([Message|MoreXs],Ys,[Message|MoreZs] :-
    merge(MoreXs?,Ys,MoreZs).

merge(Xs,[Message|MoreYs],[Message|MoreZs] :-
    merge(Xs,MoreYs?,MoreZs).
```

These two clauses define the third argument as a non-deterministic merge of the two streams. The CP implementation must ensure that this merge is fair. Messages sent to one of the first two arguments will be sent to the third eventually.

A state variable doesn't necesarily contain a stream being consumed by a process. It may simply contain a term as a value. In order to deal with this uniformly, we extend merge to simply share the term itself by unification.

```
merge(Term,Term,Term) :- otherwise | true.
```

This relies upon the special guard literal otherwise which succeeds if all lexically preceding clauses fail. Declaratively, otherwise operates as the conjunction of negative literals corresponding to guards and unifications of the preceding clauses. This clause will succeed if none of the earlier clauses applies.

The position method above:

```
method(window,position(X,Y))
```

expands using merge to:

```
window([position(X1,Y1) | NewWindow],
          X,Y,Width,Height,Contents) :-
      merge(X1,X2,X),
      merge(Y1,Y2,Y),
      window(NewWindow?,
               X2,Y2,Width,Height,Contents).
```

In Prolog and CP, [Message|MoreYs] is just syntactic sugar for the term dot(Message,MoreYs) - the standard pairing function. Similarly, [Message1,Message2] translates to dot(Message1,dot(Message2,[])). Since lists are actually terms, the above definition of merge cannot distinguish between lists meant as message channels and lists meant as terms. The preprocessor avoids this problem by using a unique functor (like \stream) in place of the dot functor. Similarly, [] gets replaced by some unique constant like \endOfStream. Further examples continue to use the list notation, though, for readability.

Class Inheritance.

Vulcan can implement inheritance at least two ways. The first is the description copying semantics corresponding to subclassing. The second is inheritance by delegation to parts.

For subclassing, class declarations include the superclasses as the third argument. The subclass is then created with source copies of all methods inherited from its superclasses. The superclasses field can be a single class or a list of classes for multiple inheritance.

```
class(labeledWindow,[Label],window).
```

makes the preprocessor expand all window methods with respect to labeledWindow. Below, the source from the moveBy method of window is copied to labeledWindow:

```
method(labeledWindow,moveBy(DeltaX,DeltaY)) :-
      send(Self,erase),
      plus(X,DeltaX,NewX),
      plus(Y,DeltaY,NewY),
      become((X,NewX),(Y,NewY)),
      send(Self,show).
```

expands to:

```
labeledWindow([moveBy(DeltaX,DeltaY)
                        | NewLabeledWindow],
          X,Y,Width,Height,Contents,Label) :-
      labeledWindow([erase,
                  privateMoveBy(DeltaX,DeltaY),
                  show | NewLabeledWindow?],
          NewX?,NewY?,Width,
          Height,Contents,Label).
```

Similarly, privateMoveBy is copied to labeledWindow.

If the programmer defines a new moveBy method for labeledWindow, it overrides the automatically generated one. As in Smalltalk, an overriding method can access the inherited version: sendSuper(Message) tells the preprocessor to expand inline the inherited method definition. (When the inherited method doesn't correspond to a single clause, then the relevent clauses can be copied with a unique predicate name, and a call to that predicate substituted inline) The show message from moveBy is a good example. It must show the label, then do whatever windows do to show their contents. The show definitions for the two classes:

```
method(window,show) :-
      send(Contents,displayAt(X,Y)).

method(labeledWindow,show) :-
      sendSuper(show),
      send(Self,displayLabel).
```

combine for labeledWindow to give:

```
method(labeledWindow,show) :-
      send(Contents,displayAt(X,Y)),
      send(Self,displayLabel).
```

which expands to:

```
labeledWindow([show | NewLabeledWindow],
          X,Y,Width,Height,Contents,Label) :-
      Contents = [displayAt(X,Y) | NewContents],
      labeledWindow([displayLabel
                        | NewLabeledWindow?],
          X,Y,Width,Height,NewContents,Label).
```

Inheritance by delegation to parts.

Shapiro and Takeuchi achieve the sharing of methods and structure common in systems with class inheritance by using delegation to parts. A labeledWindow contains a window as a part, rather than implicitly being a kind of window. The delegation code for class labeledWindow is:

```
class(labeledWindow,[Window,Label]).
```

A method not accepted explicitly by a labeledWindow is *delegated* to the contained window (referenced through a state variable) with a method of the form:

```
labeledWindow([Message|NewLabeledWindow],
              Window,Label) :-
      otherwise | /* no earlier methods were applicable */
      Window = [Message|NewWindow],
labeledWindow(NewLabeledWindow?,NewWindow?,
                                          Label).
```

Since the delegation does not pass along the labeledWindow with the message, the inherited method can only refer to the window. For example, a labeledWindow delegates moveBy messages to its window part. The show message to Self in the above implementation of moveBy gets sent directly to the window, so the label does not get displayed along with the rest of the window (as discussed in [Bobrow 1985]). This problem can be solved by including in the delegation message a stream on which the labeledWindow will interpret messages sent to itself [Lieberman 1986]. This stream is called Self because the class of the object it represents is unknown, and its only property is that it supplies the overall identity. The window is the proxy for the labeledWindow, since it performs operations for it. The labeledWindow is called Self because it represents the outer object being processed.

```
labeledWindow([Message|NewLabeledWindow],
              Window,Label) :-
      otherwise |
      Window =
         [handle(Message,Self,NewLabeledWindow)
                | NewWindow],
      labeledWindow(Self?,NewWindow?,Label).
```

This clause essentially sends a special message to its window part that asks it to execute the message normally, except to direct all sends to Self to the labeledWindow instead of the window. The labeledWindow recurs on Self rather than NewLabeledWindow so that it receives messages sent in the delegation before those sent externally (just as in send to Self). The handle method must eventually make NewWindow a tail of Self (Self = [Msg1,Msg2,... | NewWindow]).

Delegation can be explicitly used in places other than the automatic delegation method. The delegation version of the `show` method for `labeledWindow` is just:

```
method(labeledWindow,show) :-
      delegateTo(Window,show),
      send(Self,displayLabel).
```

This translates to

```
labeledWindow([show | NewLabeledWindow],
                        Window,Label) :-
     Window =
          [handle(show,Self,[displayLabel
                         | NewLabeledWindow])
                    | NewWindow],
     labeledWindow(Self?, NewWindow?, Label).
```

Properly demonstrating that the CP code described above actually supports reentrant, state changing delegation involves quite long and complicated examples, and so lies beyond the scope of this paper.

These two implementations of inheritance are compatible, allowing selection of an inheritance scheme suited to the problem.

Intricacies of Send

In most object-oriented systems, mutable and immutable objects are programmed in a uniform manner. To get the first element of a list one sends the list the same message independent of whether it is immutable or not. CP preserves this interchangeability and code-sharing between different implementations of an object when computing with perpetual processes.

Logical terms, however, are not interchangeable with streams to perpetual processes. Logical terms are created and accessed by unification. They are much simpler and more efficient than CP mutable objects. Logical terms can be incrementally filled in since they can contain logical variables to be instantiated later. Terms cannot, however, be the basis of object-oriented programming since they are "write-once". A term can represent a window of a particular size and location but cannot continue to represent that window as it moves and grows. Message sending must be extended to work with terms.

We have been sending messages to objects in CP by simply open-coding a unification of the message stream with the message and a new tail. In order to generalize message sending to apply to terms as well, we introduce a `send` predicate:

```
send([Message|NewObject],Message,NewObject).
```

A call to `send`

```
send(Window,move(3,42),NewWindow)
```

is equivalent to

```
Window = [move(3,42) | NewWindow].
```

The definition must be extended to handle terms, as was `merge`. The following clause is added.

```
send(X,Message,X) :-
      otherwise |
      sendToTerm(X,Message).
```

`sendToTerm` actually does the requisite operations, rather than just queuing messages, so it is named appropriately. Suppose we have both immutable and mutable line segments in a graphics package. Immutable line segments are terms of the form `line(StartPoint,EndPoint)` while mutable line segments are defined by:

```
class(line,[StartPoint,EndPoint]).
```

A method for displaying mutable lines can be specified as follows.

```
method(line,display) :-
 drawLine(StartPoint,EndPoint).
```

`send` can be extended for displaying immutable lines as follows.

```
sendToTerm(line(StartPoint,EndPoint),display) :-
      drawLine(StartPoint,EndPoint).
```

The above can be written as follows.

```
method(line(StartPoint,EndPoint),display) :-
      drawLine(StartPoint,EndPoint).
```

Thus, the same syntax defines methods for both mutable objects and immutable terms.

The correspondence should go both ways. Unfortunately, CP objects cannot be used where terms are used. Unification of CP objects is impossible since there are no pointers to the objects. Unification between streams to CP objects simply constrains the streams to have the same messages. Also, low-level primitives for arithmetic and the like deal only with terms.

Closing streams and reference counting.

A stream to a perpetual process can be closed by Vulcan. Stream closing requires that every class definition add a clause of the following form.

```
className([],StateVar1, ... StateVarN) :-
      closeStream(StateVar1),
      ...
      closeStream(StateVarN).
```

Where `closeStream` is defined as

```
closeStream([]).
closeStream(X) :- otherwise | true.
```

merge needs to be extended by the following clauses.

```
merge([],Stream,Stream).
merge(Stream,[],Stream).
```

The effect is that when one of the input streams of a merge process is closed off, the merge process simply unifies the other input stream to the output stream, and then goes away.

There are two reasons why it is of value to explictly close a stream. One reason is that it can be more efficient than waiting for the garbage collector to collect streams and associated suspended processes which are not accessible from the active processes. Depending upon the implementation of merge, many merge processes can terminate if streams are explicitly closed. The second reason for closing streams is to avoid the semantic difficulties with perpetual processes. What does it mean if all the active processes terminate but some perpetual processes are left suspended? If the expansions of Vulcan methods close streams then the objects can be realized as nearly perpetual processes. A nearly perpetual process is active until it explictly terminates.

Since streams are always split, no stream should ever be shared. Consequently it is safe for any object to close a stream to which it no longer intends to send messages. Typically this will remove one of the streams being merged before being consumed by the perpetual process. If the stream is being consumed directly, then the perpetual process actually receives [], and typically terminates.

A terminating perpetual process should close all streams contained in its state. Likewise, the update of state variables should close any newly unreferenced streams. Thus, Vulcan must insert calls to closeStream when expanding methods with become.

The analogy between stream splitting and incrementing a reference count and stream closing and decrementing a reference count is very strong. As with reference counting, stream closing is not a full substitute for garbage collection since there can be cycles of object references which are disconnected from all the active processes.

A very odd fact about Vulcan is that all processes will terminate only in programs for which an object reference count collector recovers all garbage. Programs which create cyclic object references and then drop all references to the cycle will have processes that never terminate.

A Send that does not rely on Merge

In CP and Vulcan sharing of streams is handled by calls to merge. Alternatively, send could search for the uninstantiated tail of the stream, instantiating it to the message/new-tail pair. This search can be easily implemented using the low-level CP primitive var which is true if its argument is currently unbound.

```
searchingSend(Stream,Message,NewStream) :-
      var(Stream) |
      Stream = [Message|NewStream].

searchingSend([OtherMessage|StreamRemainder],
              , Message,NewStream) :-
      otherwise |
      searchingSend(StreamRemainder,
                        Message,NewStream).

searchingSend(X,Message,X) :-
      otherwise |
      sendToTerm(X,Message).
```

The semantics of var is unclear and its implementation is problematic on a distributed system. The use of var can be avoided if a unique token is included in every message. The first clause of searchingSend could then be rewritten as follows.

```
searchingSend([Message|NewStream],
                        Message,NewStream).
```

SearchingSend has some unpleasant performance characteristics. The overhead of sending a message on a stream is proportional to the number of messages sent to the object since the last transmission from this reference to the stream. The average cost of sending a message on a shared stream is proportional to the number of references to the stream. If there is a reference to a stream that is rarely used it will prevent the garbage collector from collecting the history of old messages on that stream. It is possible for anyone sharing a stream to close it down by binding the variable in the tail to a constant.

It is for these reasons that Vulcan expands methods into clauses which call merge whenever a new reference to a stream is released. There is one case, however, in which the preprocessor cannot generate the appropriate calls to merge. If a state variable of a perpetual process contains a term which contains streams then the term can get passed out without the contained streams being split. Users of that term can then get multiple references to the contained streams. Vulcan alleviates this problem by using stream splitting when possible and falling back on searching send to recover gracefully from those situations in which streams were not split when needed. In the normal case, the stream splitting will prevent multiple references to a stream and there is no overhead to the searching send.

With a naive implementation of `merge` as given earlier, the overhead of a message transmission can, in the worst case, be linear in the number of references to a stream. The best implementation in CP of stream merging is logarithmic [Shapiro 1984]. [Shapiro 1986] and the Logix implementation of FCP [Silverman 1985] implements stream merging primitively so that it has a small constant cost. Vulcan can exploit these primitives to maintain a constant cost of a message transmission (on an implementation on a sequential machine).

Reliance on searching send is inconsistent with the above claims about reference counting. This is because it is only safe for Vulcan to generate stream closing code when it can tell from static analysis that the stream is unshared. Cleaning up this conflict is an area for future research.

More esoteric object-oriented programming features.

The streams consumed by perpetual processes are ordinary CP terms. As such it is possible for several perpetual processes to be consuming the same stream. When a message is put on the stream it is directly broadcast to all the objects sharing that stream. To be able to send to both individuals and groups the stream being consumed can be split into a shared broadcasting stream and a private stream. The following clause for making windows maintains the shared stream called `Ether`.

```
makeWindow(InitValues,Ether,PrivNewObject) :-
      make(window,InitValues,NewObject),
      merge(PrivNewObject?,Ether?,NewObject).
```

The caller of `makeWindow` gets a private channel to the newly created object which is consuming a stream which is the merger of the `Ether` and the private channel. `InitValues` contains initial values for the state variables.

One can broadcast on the `Ether` by sending messages just like any other message sending. For example, `send(Ether,redisplay,NewEther)` will cause all windows to redisplay themselves. If the message is incomplete, for example, `send(Ether,positions(Positions),NewEther)`, then one can arrange to get the bag of responses by using "open-ended" lists. Open-ended lists are lists whose last tail is always an uninstantiated variable. If one is interested in just the first answer (as in the Actor `race` construct), then one waits for just the head of the list. If one is interested all the elements, the short-circuit technique of [Takeuchi 1983] described in [Shapiro 1986] can be used to tell when they're all present.

Various object-oriented programming systems support features such as classes as objects, meta-classes, and method combination [Bobrow 1986]. These features can be incorporated into the Vulcan framework. Given some support for global naming of processes (e.g. the module feature of Logix [Silverman 1985]) it is possible to support classes as objects. If the Vulcan preprocessor itself was built in an object-oriented fashion then different meta-classes could process `class` and `method` specifications differently. Method combination is normally implemented as a define-time process. The Vulcan preprocessor could be extended to combine pieces of methods and expand the combinations into ordinary CP clauses.

There are some unique capabilities of objects in CP because the communication channels are explicit. A method can, for example, peek ahead in the message stream to see if some particular message is pending. The following method does nothing if there is an `undo` message of the same sort pending. The example is inspired by the time warp system for discrete simulations. [Jefferson 1982]

```
method(timeWarpObject,do(X)) :-
      pendingMessage(undo(X),Self) | true.

method(timeWarpObject,do(X)) :-
      otherwise | doHairyComputation(X,Self).
```

where `pendingMessage` is defined as

```
pendingMessage(Message,Stream) :-
      var(Stream) | false.

pendingMessage(Message,[Message|MoreMsgs]) :-
      otherwise | true.

pendingMessage(Message,
                [OtherMessage|MoreMsgs]) :-
      otherwise |
      pendingMessage(Message,MoreMsgs).
```

It is only by convention that messages to perpetual processes are put on streams. The structure being consumed can be any term. It might be useful to sometimes communicate via priority queues or binary trees. This interesting avenue of research has not been explored.

The objects of CP are really sequences of process reductions that we choose to view as an object with identity and permanence. By convention a process representing an object reduces to a collection of processes one of which has the same predicate and the same state variable values unless they have been explicitly changed. It is worthwhile exploring the usefulness of occasionally breaking this convention. An object can become an object of another type or become several objects which perhaps share some state or channels. If some useful cliches involving identity forking or changing are discovered they may be supportable in Vulcan as variants or extensions of the `become` statements.

Comparisons with other work

There have been several approaches to merging the logic programming and object programming paradigms. Components of the logic programming paradigm are: terms, clauses, goals, logic variables, unification, etc. Components of the object programming paradigm are: objects, classes, methods, messages, message sending, instance variables (slots, acquaintances), object references, continuations, inheritance, etc. The different approaches to merging these paradigms can be roughly categorized according to how they draw correspondences between their respective components.

A straightforward way to embed objects in Prolog is for an object to consist of a set of unit-clauses asserted in the database. Consider:

```
name(sym0003,bob).
species(sym0003,human).
age(sym0003,24).
mother(sym0003,sym0002).

parent(X,Y) :-
      mother(X,Y).

parent(X,Y) :-
      father(X,Y).
```

In object-oriented terminology, `sym0003` is an object reference to an object with instance variables `name`, `age`, etc., and that responds to the message `parent`. Each unit clause stating a property of such an identifier serves as the storage for an instance variable of that object. In this framework, the user makes no distinction between stored and computed values, i.e., between instance variables and methods. It is also natural in this scheme to have methods defined on individuals, or on classes whose membership is based on some computed quality (e.g. all `humans` with a parent named `fred`). The approach taken by [Gullichsen 1985] is essentially similar to this.

There are disadvantages with this approach. One is its heavy reliance on the non-logical assert and retract: they must be used to change the value of instance variables, and even to create new instances (something that is pure in most other systems). Because these objects exist as assertions in the database, garbage collecting them is also hard. Also there is no encapsulation so any program can inspect or update the state of an object without sending it a message.

LM-Prolog [Kahn 1984], CommonLog (a logic programming extension to CommonLoops [Bobrow 1986]), Tao [Okuno 1984], and Uniform [Kahn 1981] directly support employing user defined objects as terms. Unification of primitive term types like symbols and conses is handled primitively as in any Prolog. Unification of user defined objects causes unification messages to be sent, so that the objects can unify according to their abstract properties. This supports the abstraction power of object-oriented languages. In contrast, standard Prologs only support unification of syntactic representations. If these user-defined objects also have changeable state (as in CommonLog and Tao), then the declarative semantics of unification are lost. This approach doesn't merge the notions of changeable state and logic programming, it just allows them to co-exist uneasily.

Prolog-with-Equality [Kornfeld 1983] represents a similar approach in which regular Prolog syntactic terms are able to serve as user defined objects. When the Prolog interpreter fails to syntactically unify terms `A` and `B`, it instead attempts to satisfy the goal `equals(A,B)`. If this goal succeeds, then the two terms are considered to be unified, with the bindings introduced by the equality proof in effect. In this approach, the clauses defined for `equals` correspond to unify methods, and functors of terms typically correspond to classes. This approach also deals only with object-oriented abstraction, and not with changeable state.

A weakness of Vulcan in comparison to some of the above approaches is the inability to unify Vulcan objects. This could be dealt with by extending the underlying Prolog to send `equals` messages when attempting to unify objects. This would not work in the standard concurrent logic programming languages, since messages cannot be sent from the guard.

Several people have suggested the following approach for dealing with changeable state within the logic framework: A term corresponds to the state of an object at a given moment in time. An object consists of the list of such states, representing the object's history. An object reference consists of a pointer into this list. The current state of the object is the state immediately before the currently uninstantiated tail of this list. The searching-send technique is used to find this state. Any state changes to the object are represented by further instantiating the object's history list. From the logic point of view, the object's state isn't being changed, more of its history is being discovered. Searching send could avoid actual search if implemented primitively. The semantics of this approach are suspect since a memory cell is being simulated by using the meta-logical `var` predicate to reveal the otherwise hidden side-effect in unification. This approach also does not support object encapsulation.

Mandala is perhaps most similar in approach to Vulcan. Mandala objects are also perpetual processes consuming streams, built on a concurrent logic programming language. Unlike Vulcan, all object references are indirect by name through a name server object, which binds names to streams. The individual Mandala object contains its own set of clauses, which are processed meta-interpretively when the object processes a message. This meta-interpretive set of clauses is reminicent of Logix modules. There are currently unresolved efficiency issues with this approach: the name server is a central bottleneck in the system, and garbage collection is hard since all objects are refered to by name. It is hoped that the interpretive overhead can be removed through partial evaluation.

An Evaluation of Vulcan as an object-oriented programming language

As discussed in the introduction, a system supports object-oriented programming well if it provides good linguistic, execution, and environmental support and has a clear semantics. In addition, the paradigm should be supported in a way that coexists smoothly with other supported paradigms.

Linguistic support.

The primary advantage of Vulcan over CP is the clear, concise, and mnemonic manner of defining methods. One can program as if one had real pointers to objects. Class definitions provide a more declarative means of defining creation and initialization methods. Common cliches such as inheritance and method specialization are supported.

Execution performance.

The performance of Vulcan programs should be comparable with the equivalent CP programs since the Vulcan programs are translated into CP. The translated Vulcan may call `merge` more often than the hand-coded equivalent program. This can be alleviated by state variable declarations in the class. The support for interchangeability of terms and streams to perpetual processes does require a simple run-time test in `merge` and `send`. Again declarations can be used to alleviate this overhead.

Environmental support.

The environmental support of Vulcan has yet to be developed but no fundamental obstacles are expected. Normally it is difficult to debug the execution of translated programs (e.g. debugging compiled programs) since the program being executed is so different from its source. The declarative information in the class definitions is necessary for good browsing tools. Tracing can be accomplished by having Vulcan leave behind extra code in methods that saves or prints the relevant state. More relevant error messages such as "message not understood" can be produced by automatically generating for every class a final clause to catch the error. Declarative debugging systems [Shapiro 1982] should be modifiable to provide an object-oriented view.

Semantics.

The operational and declarative semantics of Vulcan programs is given in terms of their translation to CP. So long as this translation is straight-forward this is acceptable. The semantics of CP is on a sound footing [Shapiro 1983b]. The programs can be viewed as logical axioms and the execution viewed as controlled deduction. The semantics of perpetual processes is an active area of research and does not appear problematic. There are no side-effects in CP and yet via tail-recursion optimizations there are no performance penalties for avoiding side-effects. The CP clause which is the expansion of a Vulcan method definition is a Horn clause partially defining the permissible histories an object can have. [Shapiro 1983b]

Peaceful coexistence with other paradigms.

Vulcan can be the basis of a single paradigm system, in that it can be used as just an object-oriented language. It does coexist with CP which is based upon both a logic programming paradigm and a process reduction paradigm. A potential source of trouble in mixing Vulcan and CP in a program is that Vulcan is based upon very strong conventions of CP programming. The user writing CP may easily violate these conventions when, for example, manipulating a stream being consumed by an Vulcan process. A problem which is easy to fix is that Vulcan generates predicates whose names are the names of the classes. These should be made unique since they may conflict with user predicates and they are never called explicitly by user programs. These interactions of CP and Vulcan have not been explored in depth.

Directions for Future Research

Although research groups in Japan, Israel, England, and other countries have been actively programming in an object-oriented style in CP and related languages, there is no experience yet with using Vulcan for a serious application. This experience is necessary to evaluate the system and to discover its strengths and weaknesses.

CP is a well-known instance of a growing class of concurrent logic programming languages. Very little in Vulcan depends upon the special features of CP. It should be the case that only small changes are necessary to change Vulcan to translate into Guarded Horn Clauses (GHC) [Ueda 1985], P-Prolog, or Parlog [Clark 1984]. CP and GHC have "flat" versions that restrict the guards to calls to simple predicates. Some rewriting of Vulcan is necessary to produce FCP or FGHC programs.

The Vulcan language provides at least as much basic functionality as most object-oriented programming languages do. Exploring new functionality such as message stream peeking, direct broadcasting, multiple streams per object (for capabilities and perspectives), and object forking looks very exciting. Also trying to absorb the ideas of multi-methods and meta-objects of CommonLoops [Bobrow 1986] into Vulcan may be worthwhile.

A promising avenue of research being pursued in the logic programming context is to provide language extensions by writing extended meta interpreters and use partial evaluation to remove the cost of the extra layer of interpretation. Vulcan implementation was not conceived of as an interpreter written in CP, but instead as a translator to CP. The meta-interpreter approach combined with partial evaluation provides a means of experimenting with variants of Vulcan without paying a performance penalty.

The expansion of Vulcan code, when read declaratively in the underlying logic programming language, is a description of the permissible history of the object's behavior. In this, the expansion of a Vulcan program resembles the actor semantics [Clinger 1981] of an actor program. This leads to the interesting possibility that we may have here a good representation language for reasoning about, and proving properties of, programs with side effects. The CP interpreter itself (being a sound, though incomplete, theorem prover) can serve as the proof engine.

There is the interesting question of how well languages like this can make effective use of highly parallel hardware. It seems quite plausible that the simple reduction process of an FCP interpreter is well adapted for execution on fine-grained SIMD machines like the Connection Machine. Techniques like broadcast streams may correspond directly to constant time operations on such hardware. Towards this end we have started exploring Vulcan programs that seem to express this style. In particular, some published Connection Machine programs [Hillis 1985] seem to be expressible quite naturally in Vulcan.

Summary

Concurrent Prolog is a suitable base for object-oriented programming. Unadorned, it suffers some syntactic problems of awkwardness and verbosity. We have presented a design for Vulcan, a syntactic sugaring for CP that remedies these problems. The result is a language in which logic programming and object-oriented programming are smoothly integrated. The language supports, in addition to the usual object-oriented constructs, concurrency, unification, and incomplete messages. This framework facilitates the exploration of variations on object-oriented programming including message stream peeking, direct broadcasting, and object forking.

Acknowledgements

The authors would like to thank Curtis Abbott, Peter Deutsch, Stan Lanning, and Mark Stefik for their comments on earlier drafts.

References.

[Bobrow 1985] Bobrow, D. G. "If Prolog is the Answer, What is the Question?, or What it Takes to Support AI Programming Paradigms" *IEEE Transactions on Software Engineering*, November 1985

[Bobrow 1986] Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. *CommonLoops: Merging Common Lisp and Object-oriented programming*. ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland Oregon, September 1986.

[Clark 1984] Clark K., Gregory S., "PARLOG: Parallel Programming in Logic", Research report DOC 84/4, Dept of Computing, Imperial College, London, 1984

[Clinger 1981] Clinger, W. "Foundations of Actor Semantics", MIT AI-TR-633, May 1981

[Gullichsen 1985] Gullichsen E., "BiggerTalk: Object-Oriented Prolog", STP-125-85, MCC-STP, Austin, TX, Nov 1985

[Hewitt 1977] Hewitt C., "Viewing Control Structures as Patterns of Passing Messages", Artificial Intelligence, Volume 8, pp. 323-363, 1977

[Hillis 1985] Hillis W. D., *The Connection Machine*, MIT Press, 1985

[Jefferson 1982] Jefferson D., Sowizral H., "Fast Concurrent Simulation using the Time Warp Mechanism, Part 1: Local Control", N-1906-AS Rand Corporation, December 1982

[Kahn 1981] Kahn, K., "Uniform -- A Language based upon Unification which unifies (much of) Lisp, Prolog, and Act 1", *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, PP. 933-939, Vancouver, Canada, 1981

[Kahn 1982] Kahn, K., "Intermission - Actors in PROLOG" in *Logic Programming*, eds K.L. Clark and S.-A.Tarnlund, Academic Press, 1982

[Kahn 1984] Kahn, K., Carlsson M., "How to implement Prolog on a LISP Machine", in *implementations of PROLOG*, ed J.A. Campbell, Ellis Horwood, 1984

[Kornfeld 1983] Kornfeld W., "Equality for Prolog", *Proceedings, Seventh International Joint Conference on Artificial Intelligence*, (1983), pp. 514-519

[Lieberman 1986] Lieberman H., "Delegation and Inheritance : Two Mechanisms for Sharing Knowledge in Object - Oriented Systems", in *Langages Orientes Object*, Jan. 1986, pp. 79-89

[Okuno 1984] Okuno H., Takeuchi I., Osato N., Hibino Y., Watanabe K., "TAO: A Fast Interpreter-Centered System on Lisp Machine ELIS",

Proceedings 1984 Lisp and Functional Programming Conference, 1984

[Shapiro 1982] Shapiro, E. *Algorithmic Program Debugging*, MIT Press, 1982

[Shapiro 1983a] Shapiro, E. and Takeuchi,, A. *Object Oriented Programming in Concurrent Prolog* New Generation Computing, Springer Verlag V. 1, No. 1 1983, pp. 25-48.

[Shapiro 1983b] Shapiro, E. "A Subset of Concurrent Prolog and Its Interpreter", ICOT Technical Report, *TR-003* (1983)

[Shapiro 1984] Shapiro E., Mierowsky C., "Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog", *Proceedings, International Symposium on Logic Programming*, Atlantic City, IEEE, pp. 83-91, 1984

[Shapiro 1986] Shapiro E., Safra S., "Multiway Merge with Constant Delay in Concurrent Prolog", New Generation Computing, Vol. 4 No. 2, pp 211-216, OHMSHA, LTD. and Springer-Verlag, 1986

[Silverman 1985] Silverman W., Hirsch M., "Logix: User Manual for Release 1.1", Weizmann Institute of Science, Dec. 1985.

[Takeuchi 1983] Takeuchi A., "How to solve it in Concurrent Prolog", *Unpublished note*, 1983

[Theriault 1981] Theriault D. "A Primer for the Act-1 Language", MIT AI Working Paper 221, June 1981

[Ueda 1985] Ueda K., "Guarded Horn Clauses", ICOT Technical Report, 103, June 1985

[Zaniolo 1984] Zaniolo C., "Object Oriented Programming in PROLOG", *Proceedings, International Symposium on Logic Programming*, Atlantic City, IEEE, pp. 265-270, 1984