Concurrent Object Oriented Programming in a Logic Variable Language

Matthew Huntbach
Department of Computer Science
Queen Mary and Westfield College
London E1 4NS, UK
mmh@dcs.qmw.ac.uk

Abstract This article introduces a concurrent object oriented language whose underlying operational semantics is based on the logic variable. The language is designed in reponse to Kahn's criticisms [Kahn 89] of previous attempts to build concurrent object-oriented languages on top of concurrent logic languages. We believe Aldwych is a language which removes the verbosity of concurrent logic language code, without removing the power for abstract concurrent programming.

Introduction

The Japanese Fifth Generation Project is often considered a failure. Having been launched with ambitious goals of producing intelligent systems, its spokespeople consider its most important product to be a programming language, KL1 [Sh & Wa 93]. In fact KL1 is just one example of a family of languages [Shap 89] which originated from attempts to introduce parallelism to the logic programming paradigm, but whose main characteristics stem from pragmatic decisions made in order to map a language with a Prolog-like syntax efficiently onto a parallel architecture.

The decision to abandon Prolog's backtracking as a feature heavily dependent on sequential access to a global stack, and to impose modes on predicate arguments in order to synchronise parallel execution, looked like seriously weakening the logic programming model. Unable to lose the "parallel Prolog" tag, these languages were viewed in terms of their weakness rather than their strength. However, their main contribution may be in their introduction of a simple concept of computation tailored to distributed computing, termed, in recognition of its origin, the "logic variable". The value of the logic variable to distributed computing is discussed in more detail by Haridi et al [Hari 99].

There is a growing realisation that computing needs to escape from the sequential von Neumann model. At the time of the Fifth Generation project the main reason for wishing to consider alternatives was one of efficiency. Declarative languages were considered more efficient to program with, particular for the complex problems of intelligent systems, but inefficient to run. The inefficiency of running them could be overcome by parallel architectures, for which declarative languages were considered particularly suited as their referential transparency property meant computations could easily be split up and distributed. More recently, however, the development of computers as things embedded in an environment rather than standalone devices performing single computations has been the dominant factor in pushing reconsideration of our underlying model of computation. The environment may be human users interacting with the computer, physical devices which the computer runs, or the network of computers of which the individual computer is just a component. Gelernter and Carriero [Ge & Ca 92], in arguing the case for languages oriented around coordinating computations rather than doing computations, state that this coordination is the dominating issue in modern computer systems research. Concurrency in artificial intelligence now is of interest not just for parallel speedups but also because multi-agent systems which are inherently concurrent are becoming an increasingly important aspect of the field.

Logic Variable Languages

While the concurrent logic languages did not achieve the success hoped for, they have been part of the background of several innovative programming languages. They may therefore be regarded as an evolutionary step. The languages which have evolved from them have generally dropped the claim to be part of the logic programming paradigm, but have kept to the simple underlying computational model first present in concurrent logic programming, one in which the logic variable accomplishes with ease many of the things accomplished only with great difficulty in languages where distribution is "bolted on" to the sequential von Neumann model. Additional concepts such as semaphores, critical sections or monitors are not required.

The first of these languages is Strand [Fo & Ta 89] which for a while was successfully marketed as a coordination language. In fact Strand was more a marketing trick than a new language: simply concurrent logic programming explained without using logic programming idioms. The authors of Strand developed the model further, into a notation they called PCN [Fost 96]: the "definitional variable" of PCN is a logic variable although PCN code does not resemble logic programming code. The emphasis in PCN is on composing existing components written in imperative languages to build systems which may be run on highly parallel computers. To aid this it adds to the model notations for mapping computations onto meshes of processing elemements.

The language ToonTalk [Kahn 96] describes the concurrent logic model in a cartoon metaphor, giving a graphic environment intended for children, where aspects of the language are represented by concrete objects in the cartoon world, and programming is done by manipulating these objects. Clearly, abandoning the text base of the language altogether is a particularly radical step in presenting the computational model of concurrent logic programming in a new form.

The most successful of the logic variable languages in terms of practical usage is Erlang [Arms 96]. Erlang was developed by Ericsson and has been used extensively by them in building telecommunications applications. Ericsson have noted the improvement in programmer productivity resulting from adopting a declarative language. Like PCN, Erlang adds mapping notations to the underlying model, but also mechanisms for dealing with node failure. Although Ericsson acknowledge the influence of concurrent logic languages in the development of Erlang, indeed an early version was implemented in Strand, they prefer to promote it as a functional language.

Challenging Erlang as a robust language with the sort of human and system support needed for a language to succeed, is Oz [Smol 95]. Oz originates from concurrent constraint programming [Sa & Ri 90], which adds arithmetic constraints to the partial binding constraints of the standard logic variable. While of all the languages described in this section, Oz is the one which makes the most claims about its links with logic programming, it is arguably the one which has moved the furthest distance from the original logic programming model. It has introduced mutable variables as a primitive, and while early versions of Oz kept with the concurrent logic language aim of exploiting parallelism implicitly, later versions [Ha & Fr 99] have abandoned this in favour of explicit control over concurrency by means of a thread creation construct.

The Logic Variable

A logic variable conceptually has a fixed value from the moment of its creation, but the value is unknown until the variable is bound. The communication needed to bind a logic variable is part of the variable and not part of the program manipulating the variable. This means the variable can be passed around at will. The computations passing the logic variable need not know its value. In the concurrent logic model the only restriction on concurrency is that a computation will suspend if it needs to know

the value of a logic variable to make a choice. In a distributed environment, the value of a logic variable will eventually be communicated to every computation that is suspended waiting for it, but no guarantee is offered over communication time between one computation binding the variable and others learning of the binding, including no guarantee that one will learn of it before another if more than one is suspended on it. A variable may be bound to a value which is a compound term with components which may themselves be logic variables, the component logic variables may be bound separately at a later time. Variables may not be reassigned once bound, and concurrent logic programming's abandoning of Prolog's backtracking means there is no problem with a logic variable being tentatively bound to some value only to have that value rescinded through backtracking [Hu & Ri 95], this problem being one of the difficulties in creating a true parallel Prolog.

As described above, a logic variable is just like a future in concurrent functional [Hals 85] or object-oriented [Lieb 87] languages. Where it differs is over flexibility in readers and writers. Crucially, if a logic variable is bound to a compound term containing a logic variable, the binding of the inner logic variable need not be done by the same computation, it could be done by a computation which received the binding of the outer logic variable. This is known as "back communication".

Logic programming as originally conceived [Kowa 74] had no concept of variables having readers and writers, making no distinction between testing whether a variable had a particular value and assigning that value to the variable. In reality, however, the nature of Prolog's search order and primitives mean that most programs in the language make an actual but not syntactic distinction between input and output variables. Programmers have to know the intended role of arguments to predicates. Concurrent logic languages keep the lack of syntactic distinction between readers and writers (although some have a mode declaration [Cl & Gr 86]), but it rarely makes sense to have multiple unsynchronised writers to a single assignment variable, and mechanisms suggested to handle the possibility (such as "test-and-set" operations [Yard 90]) are awkward. The fact that the concurrent logic languages break from Prolog in making a strong distinction between setting and testing the value of a logic variable call into question the wisdom of keeping to the modeless syntax.

In fact the attempts of the concurrent logic languages to hold onto a syntax based on attempts to build "computational logic" may have been a prime reason for their lack of success. The cumbersome nature of programs in these languages has been a major motivation in the development of languages like Oz and Erlang which have logic variables but not a logic syntax. At the extreme, languages which closely resemble existing conventional imperative languages, but have logic variables rather than mutable variables have been suggested [Thor 95].

A Concurrent Object Oriented Language Implemented with Logic Variables

We propose building a concurrent object oriented languages which uses logic variables. In fact we shall insist that our language maintains a direct translation into standard concurrent logic language code. We call our language "Aldwych" as a pun on the concurrent logic language Strand (the London street Aldwych turns into the London street Strand).

This approach is not new. Following a description of a method of writing concurrent logic programs which viewed them in object terms rather than logic terms [Sh & Ta 83], a number of attempts were made to build object-oriented languages which compiled into concurrent logic languages [Davi 92]. The main advantage of maintaining the direct translation into a concurrent logic language is that it provides us with an operational semantics, in much the same way lambda-calculus provides

pure functional programming with an operational semantics. The existence of efficient compilers for concurrent logic languages means the translation provides us with an implementation which piggy-backs on this existing work done to implement logic variables on distributed systems [Roku 96]. This is not to say that Aldwych must always be implemented this way, however. Unlike some of the previous attempts to build concurrent object-oriented languages on top of concurrent logic languages, our aim is that Aldwych should be a language which stands on its own, rather than a tool for assisting logic programmers. The user of Aldwych should not need to know the concurrent logic translation, and should never have to resort to writing sections of code in a concurrent logic language. The clean nature of the underlying concurrent logic language should however guard against the tendency of languages to accumulate "features" which break the original spirit of the language turning it into a "ball of mud" (compare Lisp with functional languages which have maintained their pure functional nature). Also, the translation into a concurrent logic language may be further processed by using partial evaluation [Hunt 89], a process which works best when applied to a language lacking in large numbers of features whose interactions with each other have to be kept into account.

Additionally, we have designed Aldwych in the light of the criticisms made of "concurrent object on top of concurrent logic" languages by one of those involved with the development of one of these languages [Kahn 89]. The main criticism was that the languages, in attempting to closely model existing concurrent object-oriented languages, have lost the flexibility of their underlying concurrent logic model. Aldwych contains a number of features with easy representation in the concurrent logic model, but not found in previous concurrent object-oriented languages.

Concurrent Objects and Agents

The foundations of concurrent object-oriented programming were laid by Hewitt [Hewi 77], who recognised that computations could be described in terms of simple agents he termed "actors" exchanging messages rather than as a single thread of control through a program manipulating the mutable store of the von Neumann computer. There was a sharp distinction between the real concurrency envisaged by Hewitt, and the simulated concurrency of Simula [Birt 73], the language which laid the foundation for sequential object-oriented programming. Hewitt's actors have been a minor theme in the development of the object-oriented paradigm (stronger in Smalltalk, but of little influence in C++ and Java), but it is the theme of most interest to those who wish to develop object-oriented programming into multi-agent systems.

Shoham [Shoh 93] describes agent-oriented programming as specialising concurrent object-oriented programming by limiting the states which objects may have, the sort of messages they may exchange, and the way they may interpret those messages. Hewitt developed the actor model by investigating open systems [He & In 91], where an open system is a collection of interlinked services operating concurrently and asychronously, without central control. Actors and concurrent logic languages have been proposed as the two computational models most suited for open systems [Ka & Mi 88]. An important aspect of open systems is that the components must be able to encapsulate control of hardware alongside the software encapsulation of standard objects. Erlang may be considered as answering some of the criticisms of those working in open systems on concurrent logic languages lack of hardware control by adding explicit control of hardware, including mechanisms for dealing with sytems failure. The most explicit encapsulation of hardware resources in an object-oriented style is found in Joule [Agor 95], another language which acknowledges the influence of concurrent logic languages.

Consideration has been given to the direct use of concurrent logic languages for programming agents [Hunt 95]. A concurrent logic language augmented with

annotations (which may be compiled away giving a pure concurrent logic program) has been used to control a system involving multiple robots, with manipulators and vision sensors, which interact by negotiation [Nish 98]. However, the direct use of concurrent logic languages tends to lead to verbose programs in which there is a lot of obvious redundancy. One of the motivating factors of Aldwych is to identify and remove this redundancy by observing common patterns in concurrent logic programs designed in an object-oriented style and providing language features which represent these patterns in a more convenient form.

Readers and Writers

A call to an Aldwych computation consists of a process type name, followed by a list of variables which are those the computation is a reader for, and a further list of those the computation is a writer for. For example

```
comp(u,v)->(x,y)
```

represents a call to a computation of type `comp` which is a reader of `u` and `v` and a writer to `x` and `y`. The second pair of brackets may be omitted if a computation is a writer to only one variable. A collection of computations must have the property that every variable has exactly one writer and one or more readers. For example:

```
comp1(u,v)->(x,y), comp2(x)->u, comp3(y)->v, printer(v)
```

where `v` has one writer, a `comp3` computation, but two readers: a `printer` and a `comp1` computation. Aldwych has no primitive input/output commands, but processess could be linked to hardware devices as suggested by the term `printer`.

One cause of verbosity in concurrent logic languages is the lack of embedded calls, necessitating much invention of names to pass values. Aldwych has embedded calls, for example

```
comp1(comp2(x),v)->(x,y), comp3(y)->v, printer(v)
```

is defined as equivalent to the above. This gives the language more of a functional feel. Circular data flow, as with the variable `x` is permitted.

A process description in Aldwych consists of a header naming the input and output variables, and a set of reaction rules. A reaction rule has a left-hand side (lhs) giving the variable bindings that must occur on the input variables for the procedure call to rewrite to the procedure collection on the right-hand side (rhs) of the rule. On the lhs, `x=a` means "suspend until variable `x` is equal to constant `a`", on the rhs it means "bind variable `x` to constant `a`". On the rhs `x<-y` means "assign input variable `y` to output variable `x`. So the process description

```
#comp1(u,v)->(x,y)
{
 u=a || x=b, y<-v;
 v=c || z<-comp2(u), y<-z, x<-comp3(z);
}
```

means `comp1(u,v)->(x,y)` is suspended until either `u` gets assigned constant `a` or `v` gets assigned constant `c`. Other computations operating concurrently will be doing the assigning. The first rule says that when `comp1` learns that `u` has become `a` it assigns constant `b` to `x`, and variable `v` to `y`. The latter assignment means any reader of `y` becomes a reader of `v`. The second rule shows a more complex example. It sets up two computations and introduces a local variable `z`. Here `z<-comp2(u)` is just another way of writing `comp2(u)->z`. Local variables like `z` must have exactly one writer and one or more readers. Process output variables, like `x` and `y` must have exactly one writer in every rule. Process input variables like `u` and `v` need not have readers in the rules. Failure to have the correct number of readers or writers results in a compiler error.

Aldwych is indeterminate. A `comp1` computation will react to whichever of its input variables is bound first if it has immediate access to a processor. However if no attempt is made to evaluate it until both input variables are bound it reacts indeterminately according to either rule.

Long Lived Processes

The computation described above are ephemeral. The way of making long-lived processes in Aldwych follows the technique for programming in an object-oriented style in concurrent logic languages described by Shapiro and Takeuchi [Sh & Ta 83]. Note firstly that variables may be assigned tuple values as well as constants. `a=f(b,c)` on the lhs means "suspend until variable `a` has been assigned to a tuple with name `f` and two arguments given local variable names `b` and `c`", and on the rhs means "assign the tuple with name `f` and arguments variables `b` and `c`". If `a=f(b,c)` occurs on the rhs, `b` and `c` must have each have a writer there. However, no synchronisation is required between `a` being given value `f(b,c)` and `b` and `c` being given their values. If `a=f(b,c)` occurs on the lhs, `b` and `c` can have no writers on the rhs but can have any number of readers. Thus tuples as described here may not be used for back communication. The reason for this is that as `a` may have several readers, allowing `b` or `c` to be used for back communication would enable each of these readers to be a writer for `b` or `c`, thus breaking the rules that ensure every variable has exactly one writer. The match `a=f(b,c)` on the lhs has no dependence on whether `b` and `c` have been given values by the time it is made.

Actors work by handling a message and specifying a new actor to handle the next message [Agha 90]. This is how concurrent object oriented programming is modelled in concurrent logic languages: a recursive call in the body of a clause is considered to be a continuation of the object which reacted according to that clause. A message is sent by binding a shared variable to a tuple, one of whose arguments is a variable intended to be used for similar communication of the next message. Here is a process which receives a stream of `add` messages, adding the arguments of the messages to a sum until the stream is ended, and the sum is then returned as the value of the computation:

```
#adder(count,mess)->sum
{
 mess=add(val,mess1) || sum<-adder(count+val,mess1);
 mess=end || sum<-count;
}
```

To avoid the verbosity here, Aldwych rules which have a single bar separator have an unwritten recursive call. The arguments to the recursive call are the same as the arguments to the call that made it unless they have been changed by an assignment. So an assignment `a<-e` on the rhs of a single bar rule where `a` is a process input variable and `e` is an expression means that `e` replaces `a` in the recursive call to the process. This means the above is more concisely written in Aldwych:

```
#adder(count,mess)->sum
{
 mess=add(val,mess1) | count<-count+val, mess<-mess1;
 mess=end || sum<-count;
}
```

It can be seen that though we are dealing with logic variables here, it makes sense to think of `count` as a mutable variable which forms part of the state of an `adder` process. Note that also an output process variable may have its value changed. If we start off with a version of the `adder` process which unlike the previous one is not tail recursive:

```
#adder(mess)->sum
{
 mess=add(val,mess1) || sum<-adder(mess1)+val;
 mess=end || sum<-0;
}
```
the version with implicit recursion is:
```
#adder(mess)->sum
{
 mess=add(val,mess1) | sum<-sum+val, mess<-mess1;
 mess=end || sum<-0;
}
```
This is less intuitive to those used to thinking in a sequential imperative style, since in `sum<-sum+val`, the second `sum` is a value which is to be calculated, not a previous value stored in variable `sum`. In general, if we have `a<-e` on the rhs of a single bar rule, where `a` is a process output variable and `e` an expression, any other reference to `a` (including any inside `e`) is taken to refer to the output variable of the implicit recursive call.

Aldwych goes further and uses a similar implicit recursion to handle channels used to pass messages. Firstly, let us go back to our original tail-recursive adder, and consider using `:` as an infix tuple operator:
```
#adder(count,channel)->sum
{
 channel=add(val):channel1 || sum<-adder(count+val,channel1);
 channel=end || sum<-count;
}
```
Here is the less verbose syntax allowed by Aldwych to mean the same as this:
```
#adder(count,channel)->sum
{
 channel.add(val) || sum<-adder(count+val,channel);
 channel$ || sum<-count;
}
```
So `c.mess` on the lhs is equivalent to `c=mess:c1` on the lhs with all occurrences of `c` replaced by `c1` on the rhs. Also `c$` indicates that a variable representing a channel has been set to a special value indicating “end of channel”. Combining this with the implicit recursion on process calls gives:
```
#adder(count,channel)->sum
{
 channel.add(val) | count<-count+val;
 channel$ || sum<-count;
}
```
Output channels can be expressed by a similar notation. `a.mess` on the rhs where `a` is a process output variable is equivalent to `a<-mess:a1,a1<-a` where the second `a` refers to the equivalent output variable from the implicit recursive call, other references to `a` in the rhs will also refer to this `a` from the recursive call. `a$` binds variable `a` to the “end of channel” value. So here is a version of the `adder` process which instead of returning a single sum, sends out a stream of `sum` messages which give the sums of the numbers received in `add` messages as they are received:
```
#adder(count,in)->out
{
 in.add(val) | count<-count+val, out.sum(count);
 in$ || out$;
}
```

Since `c.f(a,b)` on the rhs means "send the tuple `f(a,b)` on channel `c`", a separate notation is needed to compute and send a value: `c^f(a,b)` means "evaluate `f(a,b)` and send the result on channel `c`". On the lhs, `c?a` means the new local variable `a` is bound to whatever is the next value on channel `c`, compare with `c.a` which only reacts to channel `c` receiving the constant `a`.

Objects

The above gives us a notation which is similar to CSP [Hoar 85]. It could be used to program simple reactive agents. We could have input channels to a process connected to sensors, and output channels connected to effectors to program an agent which works in an environment. However, for more sophisticated agents we need better means of two way communication. In concurrent logic languages this is done with the logic variable's back communication, but the rules on readers and writers described above do not allow it. The rules above do allow one-to-many communication as a channel may have more than one reader, but do not allow the many-to-one communication of actor programming.

Many-to-one communication in concurrent logic programming is achieved by channel merging. A non-deterministic merge of two channels is easily defined in Aldwych:

```
#merge(c1,c2)->m
{
 c1?v  |  m^v;
 c2?v  |  m^v;
 c1$   ||  m<-c2;
 c2$   ||  m<-c1;
}
```

Messages received on the two input channels will be output on the output channel in the order received. Two channels which are merged may both be considered handles on a single process:

```
comp1(a1)->c1, comp2(a2)->c2, comp3(merge(c1,c2),a3)->out
```

So the `comp3` process will have a single channel containing messages from both the `comp1` and `comp2` process, `c1` and `c2` being their respective handles on it. However, for a closer resemblance to actors, the handles should be considered as inputs (or acquaintances) of the `comp1` and `comp2` processes. In Aldwych, a variable starting with a capital letter is taken to refer to an object. Object variables have the same rule as other variables, having exactly one writer and any number of readers. A translation is made so that an object variable in a read position is converted to a channel with a unique name in a write position, and all the write channels derived from a single object variable are merged to form a read channel. This read channel replaces the single write occurrence of the object variable. So the above can be written:

```
comp1(a1,C), comp2(a2,C), comp3(a3)->(out,C)
```

Here `comp3` produces both an output value and an object handle. This is permissible, but more usually a process will produce only object handles (in which case it can be regarded as an actor) or non-object values (in which case it can be regarded as a function). Although `comp1` and `comp2` are indicated as having no outputs, they interact by sending messages to the object `C` which they share.

Now, each reference to an object, such as `C` above, becomes a channel which has just one writer and one reader, the implicit `merge` process. The `merge` process writes to one channel which again has one reader, the process which had the one write occurrence of the object variable. Thus we are guaranteed that a message sent on one of the channels used to implement the object is not duplicated in a one-to-many communication, so we can use it for back communication. This is the key result which enables us to build actor-like concurrent object oriented programming into Aldwych.

A message is sent to an Aldwych object just like messages are sent to channels, but messages sent to channels may not have return variables nor may any of their arguments be object variables. No such restrictions apply to messages sent to objects. The syntax `Obj.mess(in)->out` refers to sending a message of type `mess` to object `Obj` with argument `in` and having a reply returned in variable `out`. This counts as a write occurrence of `out`. The object referred to by `Obj` is guaranteed to receive it and must eventually bind variable `out`, but there is no synchronisation except a process which needs the value of `out` to react will suspend until it receives it. A process which does not need to know the value of `out` may handle it as a future. We can implement the past, now and future types of message passing of the concurrent object-oriented language ABCL [Yone 90] without a special syntax.

Bank Account Objects

Bank account objects are often used as an example to illustrate object oriented programming, and we shall demonstrate Aldwych using this familiar example. Here is the basic bank account class:

```
#account(balance)->Handle
{
 Handle.deposit(sum) | balance<-balance+sum;
 Handle.balance->reply | reply<-balance;
 Handle.withdraw(sum)->reply, sum>balance | reply<-0;
 Handle.withdraw(sum)->reply, sum<=balance |
     reply<-sum, balance<-balance-sum;
 Handle$ ||;
}
```

The last rule causes an object to terminate if its handle becomes “end-of-channel”, which combined with setting a channel to an object to end-of-channel when it is no longer referenced gives us garbage collection on objects. The code is still verbose, though much less than if written directly in a concurrent logic language. Further shorthands are available in Aldwych to reduce verbosity and bring code closer to an object-oriented look. Firstly, if a class has a single output object handle, the output stream may be made anonymous (indicated by `~`). An unattached message on the lhs is taken to be attached to the anonymous handle, rendering the above (the garbage collecting final clause is unwritten as it is added as a default by the Aldwych system):

```
#account(balance)~
{
 deposit(sum) | balance<-balance+sum;
 balance->reply | reply<-balance;
 withdraw(sum)->reply, sum>balance | reply<-0;
 withdraw(sum)->reply, sum<=balance |
     reply<-sum, balance<-balance-sum;
}
```

Secondly, anonymous replies to messages are allowed, indicated by dropping the `>` in `->` on the lhs, and on the rhs `>value` means assign `value` to the anonymous return on the lhs. This gives us:

```
#account(balance)~
{
 deposit(sum) | balance<-balance+sum;
 balance-|>balance;
 withdraw(sum)-, sum>balance |>0;
 withdraw(sum)-, sum<=balance |>sum, balance<-balance-sum;
}
```

Finally, the syntax

```
a [ b | rule1; c | rule2 ]
```

is introduced as shorthand for

```
a, b | rule1;
a, c | rule2;
```

and we can even have C-style assignment shorthands, giving:

```
#account(balance)~
{
 deposit(sum)    |  balance+=sum;
 balance       -|> balance;
 withdraw(sum)-[
   sum>balance   |> 0;
   sum<=balance  |> sum, balance-=sum;
 ]
}
```

Here is the code which will set up a joint account with an initial balance of 100, shared by two objects:

```
husband(Account), wife(Account), account(100)->Account
```

The power of Aldwych comes from the way (answering the concerns of [Kahn 89]) in which although it can be used in a standard actors style, that is only a subset. As a simple example, an object in Aldwych can have more than one handle, allowing for different classes of access. A bank account, for example, may have privileged access for the bank:

```
#account(balance)->(Customer,Bank)
{
 deposit(sum)    |  balance+=sum;
 balance       -|> balance;
 Customer.withdraw(sum)-[
   sum>balance   |> 0;
   sum<=balance  |> sum, balance-=sum;
 ]
 Bank.withdraw(sum) | balance-=sum;
}
```

so the bank can withdraw fees from the account regardless of the balance. If a message has no handle, it is assumed to be applicable to all of them, so for example the first rule above is shorthand for:

```
 Customer.deposit(sum)   | balance+=sum;
 Bank.deposit(sum)       | balance+=sum;
```

A privileged access handle can be used to implement the express message passing feature of the concurrent object-oriented language ABCL [Yone 90].

The *become* operation and nested code

Now let us consider a further example, a bank account where the customer is allowed to make one withdrawal which makes the account balance negative, but this causes the account to be switched to a special overdrawn account (where the balancer is the amount overdrawn). The overdrawn account is switched back to a standard account when the overdraft is cleared. This has been used elsewhere [To & Sch 89] as an illustration of concurrent object-oriented programming. Here is a version of such an account which keeps the privileged bank access:

```
#account(balance)->(Customer,Bank)
{
 deposit(sum)    |  balance+=sum;
 balance        -|> balance;
 Customer.withdraw(sum)-[
   sum>balance  ||> sum,
                   overdrawn(sum-balance)->(Customer,Bank);
   sum<=balance |> sum, balance-=sum;
 ]
 Bank.withdraw(sum) | balance-=sum;
}

#overdrawn(balance)->(Customer,Bank)
{
 deposit(sum) [
  sum<=balance | balance-=sum;
  sum>balance || account(sum-balance)->(Customer,Bank);
 ]
 balance        -|> -balance;
 Customer.withdraw(sum)-|>0
 Bank.withdraw(sum) | balance+=sum;
}
```

This shows how the *become* operation of the actor model [Agha 90] is implemented by assigning output handles. An account object can become an overdrawn object and vice-versa, and an object sending it messages on its handle does not need to know which it is.

An alternative to this is to have the code for one sort of object nested inside another:

```
#account(balance)->(Customer,Bank)
{
 deposit(sum)    |  balance+=sum;
 balance        -|> balance;
 Customer.withdraw(sum)-[
   sum>balance  |> sum, balance<-sum-balance
        {
         deposit(sum) [
            sum<=balance | balance-=sum;
            sum>balance || balance<-sum-balance;
            ]
          balance        -|> -balance;
          Customer.withdraw(sum)-|>0
          Bank.withdraw(sum) | balance+=sum;
         }
   sum<=balance |> sum, balance-=sum;
 ]
 Bank.withdraw(sum) | balance-=sum;
}
```

In general, if we have

```
#behav1(in)->out
{
…
rhs | lhs { rules }
…
}
```

this is equivalent to:

```
#behav1(in)->out
{
…
rhs | lhs, behav2(in)->out;
…
}
```

```
#behav2(in)->out { rules' }
```

where `rules'` is obtained from `rules` by replacing any rule of the form

```
rhs || lhs
```

by

```
rhs || lhs, behav1(in)->out
```

In each case, `in` refers to the input state variables after `lhs` has altered them, and `out` to the output state variables before `lhs` has altered them. There are also triple-barred rules which cause termination from both the inner and outer behaviour, so `rules` may have rules of the form

```
rhs ||| lhs
```

which are equivalent to

```
rhs || lhs
```

in `rules'`. In fact nested code may go to any depth, with the number of bars indicating the number of nestings exited. There is further syntax, not described here, for allowing additional local variables in nested code.

The reason for introducing this nesting is that it avoids having to expand the name space of process type names. Without it, programs can end up a convoluted tangle of mutually recursive process types. Nesting gives a loop-within-a-loop effect, and combined with two further constructs not discussed here (roughly equivalent to a sequencing and an if-then-else operator) means that code can be written in a style which resembles conventional structured imperative programming.

Comparisons and Conclusion

We have introduced a programming language, Aldwych, whose underlying model is based on the logic variable. The aim is to keep the declarative simplicity and clear operational semantics of concurrent logic programming, while removing the verbosity of programs written in an object-oriented style directly in concurrent logic languages. The language is intended for modern models of computation where the emphasis is on interaction rather than algorithm. In this paper, it has only been possible to describe informally a subset of the features of Aldwych. The full language extends the idea presented here of a carefully designed notation to keep the flexibility of the underlying logic variable model, but to structure programs to fit in with other language paradigms. The two-handled bank account is just one example of where Aldwych provides a flexibility beyond existing concurrent object oriented languages. Further features exist enabling Aldwych programs to be written in styles which resemble both functional and imperative programming. An implementation of Aldwych exists, which translates the language into the concurrent logic language KL1, which may further be translated into efficient programs in C [Roku 96].

In the history of computers, many programming languages have been proposed but few have achieved significant use. It is therefore a necessity for those proposing new languages to say why existing languages are not suitable. Below we describe the disadvantages of other languages compared to Aldwych, but also features in them which could be used in future development of Aldwych.

Java – although often promoted as a language suitable for concurrent programming [Ma & Kr 99], Java is still very much oriented to sequential computation. Concurrency is achieved by threads which are awkward to use and "bolt on" to the underlying sequential language. One of the most useful lessons to be learnt from Java is the importance of a comprehensive standard library alongside the core language.

Linda [Gele 85] – widely used and has the advantage of extreme simplicity, just a few extra notations to be added to any programming language. The tuple database of Linda, however, is unsuited for agent programming, as it is both a bottleneck and insecure. From Linda we could learn that simple notations to be added to languages to give them an interface might enable the gradual adoption of a language like Aldwych as a co-ordination language.

Erlang – has achieved a niche use in building several real-life systems. Language developers should be grateful to the Ericsson company, within which Erlang was developed, for having the courage to persist with a non-standard language and showing it can be used for real systems and achieve programmer productivity gains. Erlang is unsuitable for multi-agent systems, however, because it is not object-oriented. What can be learnt from it is the importance, if practical use is to be achieved, of having notations which link the language to the architecture and enable it to deal with such things as hardware component failure.

Oz – has achieved very wide prominence in academic circles, due to having a large team of experienced academic researchers in programming languages working on its development. However, there is little evidence of it achieving much use outside academia. Its constraint mechanism may be a bottleneck for multi-agent systems programming. Recent versions of Oz have retreated from implicit concurrency to sequential evaluation with concurrency gained by explicit threads. Oz has also added features which take it away from the logic programming paradigm, claiming instead a semantics based on the γ-calculus [Smol 94], a form of process calculus like the π-calculus [Miln 92].

PICT [Pi & Tu 95] – is intended to be just a sugared version of the π-calculus. The lesson from Oz and PICT is the importance of a formal semantic background, and we believe it possible (and have done some preliminary work to this end) to give a formal description of Aldwych in a process calculus form. PICT itself, although building up from the π-calculus using various derived forms, still suffers from the lack of structure in the calculus. This is noted by Honda et al [Hond 98] who propose some additional primitives to a process claculus base for complex interactions among processes. We feel Aldwych manages such interactions more elegantly than these.

Joule – the only published evidence of this language is its manual [Agor 95] produced by a software house specialising in electronic commerce. Joule has similar problems to Oz in having a rather cumbersome syntax, and it lacks a clear underlying semantics. We mention it as the only language we are aware of that attempts to implement the ideas of hardware resource encapsulation first proposed by Miller and Drexler [Mi & Dr 88]. We feel that control of hardware resources is important in making agents autonomous. Our previous research [Hunt 91] shows that abstract concurrent languages cannot always assume hardware resources are unlimited. The mechanisms proposed by Miller and Drexler and implemented in Joule provide a more sophisticated and agent-oriented approach to rationing hardware resources than the simple priority mechanism we proposed.

The idea of a language which is concurrent by design remains a novel one. While declarative languages promised to liberate us from the von Neumann machine [Back 78] mapping them onto parallel machines and employing them in an interactive styles has often not been easy. In the ancestry of Aldwych lies the attempts of the

Japanese Fifth Generation project to parallelise the logic programming paradigm, but this is well hidden by the syntax. In fact the syntax of Aldwych is deliberately intended not to be a variation of some existing programming language paradigm. Our hope is that the simple model of computation set out here could be the foundation on which a new programming paradigm can be built.

References

[Agha 90] G.Agha. Concurrent object-oriented programming. *Comm.ACM* 33(9).

[Agor 95] Agorics Inc. Joule: *Distributed Application Foundations*. `http://www.agorics.com/joule.html`

[Arms 96] J.Armstrong, R.Virding, C.Wikström, M.Williams. Concurrent Programming in ERLANG. Prentice-Hall `http://www.erlang.org/`

[Back 78] J.Backus. Can programming be liberated from the von Neumann style? *Comm ACM* 21(8) pp.613-641.

[Birt 73] G.M.Birtwistle, O-J.Dahl, B.Myhrhaug and K.Nygaard. *Simula Begin*. Van Nostrand Reinhold.

[Cl & Gr 86] K.L.Clark and S.Gregory. Parlog: parallel programming in logic. *ACM Trans. on Programming Languages and Systems* 8(1) pp.1-49.

[Davi 92] A.Davison. A survey of logic programming based object oriented languages. In *Research Directions in Concurrent Object Oriented Programming*, MIT Press.

[Fost 96] I.Foster. Compositional parallel programming. *ACM Trans. on Programming Languages and Systems*. 18(4) pp.454-476.

[Fo & Ta 89] I.Foster and S.Taylor. *Strand: New Concepts in Parallel Programming*. Prentice Hall.

[Gele 85] D.Gelernter. Generative communication in Linda. *ACM Trans. on Programming Languages and Systems*. 7(1) pp.81-112.

[Ge & Ca 92] D.Gelernter and N.Carriero. Co-ordination languages and their significance. *Comm.ACM* 35(2) pp.97-107.

[Ha & Fr 99] S.Haridi and N.Franzén. *Tutorial of Oz*, Mozart documentations. `http://www.mozart-oz.org/documentation/tutorial/index.html`

[Hals 85] R.H.Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*. 7(4) pp.501-538.

[Hari 99] S.Haridi, P van Roy, P.Brand, M.Mehl, R.Scheidhauer and G.Smolka. Efficient logic variables for distributed computing. To appear in *ACM Trans. on Programming Languages and Systems*. `http://www.mozart-oz.org/papers/abstract/TOPLAS99.html`

[Hewi 77] C.Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence* 8(3) pp.323-364.

[He & In 91] C.Hewitt and J.Inman. DAI betwixt and between: from "intelligent agents" to open systems science. *IEEE Trans. on Systems, Man and Cybernetics* 21(6) pp.1409-1419.

[Hoar 85] C.A.R.Hoare. *Communicating sequential processes*. Prentice Hall

[Hond 98] K.Honda, V.T.Vasconcelos and M.Kubo. Language primitives and type discipline for structured communication-based programming. ESOP'98 Springer LNCS 1381.

[Hunt 89] M.Huntbach. Meta-interpreters and partial evaluation in Parlog. *Formal Aspects of Computing* 1, pp.193-211.

[Hunt 91] M.Huntbach. Speculative computing and priorities in concurrent logic languages. *3rd UK Annual Conference on Logic Programming*. Springer.

[Hunt 95] M.Huntbach, N.R.Jennings and G.A.Ringwood. How agents do it in stream logic languages. *First Int. Conf on Multi-Agent Systems* MIT Press, pp.177-184.

[Hu & Ri 95] M.Huntbach and G.Ringwood. Programming in concurrent logic languages. *IEEE Software* 12(6) pp.71-82.

[Kahn 89] K.M.Kahn. Objects – a fresh look. In *Proc. 3rd European Conf. on Object-Oriented Programming (ECOOP 89)*. S.Cook (ed.), Cambridge University Press.

[Kahn 96] K.M.Kahn. Drawing on napkins, video game animation, and other ways to program computers. *Comm.ACM* 39(8) pp.49-59.

[Ka & Mi 88] K.M.Kahn and M.S.Miller. Language design and open systems. In *The Ecology of Computation* B.A.Huberman (ed.) Elsevier.

[Kowa 74] R.A.Kowalski. Predicate logic as a computational formalism. *Proc. IFIP*, North Holland, pp.569-574.

[Lieb 87] H.Lieberman. Concurrent object-oriented programming in Act1. In *Object-Oriented Concurrent Programming*, A.Yonezawa, M.Tokoro (eds) MIT Press.

[Ma & Kr 99] J.Magee and J.Kramer. *Concurrency: State Models and Java Programs*. Wiley.

[Mi & Dr 88] M.S.Miller and K.E.Drexler. Markets and computation: agoric open systems. In *The Ecology of Computation* B.Huberman (ed.), Elsevier.

[Miln 92] R.Milner, J.Parrow and D.Walker. A calculus of mobile processes (parts I and II). *Information and Computation* 100 pp.1-77.

[Nish 98] H.Nishima, H.Ohwada and F.Mizoguchi. A multiagents robot language for communication and concurrency control. *Third Int. Conf on Multi-Agent Systems* IEEE Computer Society, pp.206-213.

[Pi & Tu 95] B.C.Pierce and D.N.Turner. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming*, T.Ito and A.Yonezawa (eds). Springer LNCS 907 pp.187-215.

[Roku 96] K.Rokusawa, A.Nakase and T.Chikayama. Distributed memory implementation of KLIC. *New Generation Computing* 14, pp.261-280.

[Sa & Ri 90] V.Saraswat and M.Rinard. Concurrent constraint programming. *Proc. 7th ACM Symp. on Principles of Programming Languages*, pp.232-245.

[Shap 89] E.Shapiro. The family of concurrent logic programming languages. *ACM Comp. Surv.* 21 (3) pp.413-510.

[Sh & Ta 83] E.Shapiro and A.Takeuchi. Object-oriented programming in Concurrent Prolog. *New Generation Computing* 1, pp.25-48.

[Sh & Wa 93] E.Shapiro and D.H.D.Warren (eds). The Fifth Generation Project: personal perspectives. *Comm. ACM* 36(3) pp.46-101.

[Sho 93] Y.Shoham. Agent-oriented programming. *Artificial Intelligence* 60 pp.51-92.

[Smol 94] G.Smolka. A foundation for higher-order concurrent constraint programming. In *1st Int. Conf. on Constraints in Computational Logic*, J-P.Jouannaud (ed.), Springer LNCS 845 pp.50-72.

[Smol 95] G.Smolka. The Oz programming model. In *Computer Science Today*, J. van Leeuwen (ed.), Springer LNCS 1000, pp.324-343.

[Thor 95] J.Thornley. Declarative Ada: parallel dataflow programming in a familiar context. *Proc. 23rd ACM Computer Science Conference* pp.73-80.

[To & Sch 89] C.Tomlinson and M.Scheevel. Concurrent object-oriented programming languages, in *Object-Oriented Concepts, Databases, and Applications* W.Kim and F.H.Lochovsky (eds) Addison Wesley.

[Yard 90] E.Yardeni, S.Kliger and E.Shapiro. The languages FCP(:) and FCP(:,?), *New Generation Computing* 7 pp.89-107.

[Yone 90] A.Yonezawa (ed.). *ABCL: An Object-Oriented Concurrent System*. MIT Press.