Aldwych: A General Purpose Concurrent Language

Matthew Huntbach
Dept. of Computer Science, Queen Mary and Westfield College
Mile End Road, London, UK, E1 4NS
Email: mmh@dcs.qmw.ac.uk

Abstract

Aldwych is proposed as the foundation of a general purpose language for parallel applications. It has aspects variously of concurrent, functional, logic and object-oriented languages, and may also interface with existing systems like a coordination language, yet it forms an integrated whole. It is intended to be applicable both for small-scale parallel programming, and for large-scale open systems

Keywords: Concurrency, actors, multi-paradigm languages, coordination, open systems.

1. Introduction

The language Linda [Gele 85] was launched as the underdog, competing against sophisticated approaches to parallel programming [Ca & Ge 89], but has triumphed over those approaches it then saw as its competitors, to the point where it is now seen as the generic model for "co-ordination languages" [Ci & Ha 96]. As its authors note [Ge & Ca 92] one of the main reasons for this is that Linda provided a simple "glue" that could be used to combine systems written in languages with which ordinary programmers are familiar, whereas competing approaches expected programmers to master complete new languages.

As Gelernter and Carriero state [Ge & Ca 92], asynchronous ensembles are *the* dominating intellectual issue in the emerging era of computer systems research. A specialisation of this issue terms the components of such ensembles "agents", and the ensemble itself is thus a "multi-agent system" [Wern 91]. In this paper we introduce a language intended to be used for programming such systems, but mindful of the Linda experience, start by considering it as a coordination language. Our language is then built up by adding features to the simple coordination model in such a way that each new feature is translatable to the simpler form of the language without the feature at the expense of making the program more complex in appearance and hard to understand. The language eventually develops into one which incorporates features from several different paradigms, but does so in a way that makes it a coherent whole, rather than something which awkwardly jumps from one paradigm to another.

2. A Co-ordination Language

The basis of Linda is that a few simple primitives may be added to any language X to produce the parallel language Linda-X. One primitive, `eval`, sparks a new parallel process, and parallel processes communicate only through a global shared database, with the primitive `out` putting data-objects called "tuples" into this database, while `rd` and `in` read tuples from it (the former leaving it unchanged, the latter consuming the tuple read). An `in` or `rd` statement may contain variables which are matched against constants in the database, but if no matching tuple is found, the process containing the statement suspends until another process introduces one with an `out`. The system is multi-lingual since systems written in Linda-X may share a database with systems written in some other language Linda-Y.

An obvious criticism of this approach is that the global database offers no security. A tuple put `out` by one process intended for co-ordination with another may be removed by a third process using an `in` inadvertently due to a programming mistake, or by design if a hostile agent is attempting to "hack" into the system. Various ways of tackling this problem, generally involving multiple tuple spaces [Gele 89], [Mi & Le 94], have been proposed.

We propose that primitives of similar simplicity to Linda's be added to existing languages, but which rather than use a global data base communicate through shared single-assignment variables. Such variables have one writer and may have several readers, and access to them is granted only by the process initialising the parallelism (and recursively by any process to which access has been granted). Any reader which needs to access the value of a variable will suspend until it has been assigned one by its writer, but unbound variables may otherwise be passed as first-class values. The variables may be termed "logic variables" as variables in the logic languages have this non-rewriteable property, but we will use the term "futures", as used in parallel functional [Hals 85] and object-oriented [Lieb 87] languages. The use of single-assignment variables as a way of coping with parallelism has also been considered in imperative languages [Thor 95].

Any language which is to be used to build a component of a system co-ordinated in our approach needs a way in which a program in that language may be invoked with input and output futures, and a way of reading and writing futures. A future may be bound to a constant or to a tuple with a fixed number of arguments each of which are further futures, which may be bound then or at a later stage; some sort of error condition may be raised if there is an attempt to bind a future which has already been bound. Similarly, when a future is read, it is read as a name and a fixed-length list of further futures, which is empty if the future is bound to a constant and stores the arguments to a tuple otherwise. No guarantee can be given as to when or in which order these further futures may be bound to, but no "back-communication" (a feature of concurrent logic languages, for example see [Clar 88]) is assumed, that is a reader of a variable is always a reader of arguments of a tuple to which it becomes bound, and a writer is always a writer to the arguments of any tuple to which it binds a variable. Co-ordination is provided by giving primitives in languages which provide this reading and writing of tuples; we shall leave it to others to fit such primitives into existing languages, but they should be no more difficult to incorporate than Linda's `in`, `rd` and `out`. In fact, following criticisms that once we have multiple blackboards we have in fact just got a complicated form of shared variable [He & Li 84], our future approach may be considered as a simplification of the multiple tuple-space approach, in which a tuple space may only ever be written to once.

This leaves us with the need for the equivalent of Linda's `eval`. We may allow a process named `p` which is a reader of m futures and a writer of n to be set up by $\mathtt{eval}(p(x_1,\ldots,x_m)\rightarrow(y_1,\ldots,y_n))$ where each $x_1,\ldots,x_m$ is either a new future or one to which the originating process is a reader or writer, while each $y_1,\ldots,y_n$ is either a future to which the originating process is a writer or a new future. Any of $x_1,\ldots,x_m$ which is a new future becomes an output future in the calling process, while all $y_1,\ldots,y_n$ whether new or existing output futures become input futures. If n is 1, the brackets are omitted round the `y`s, if n is 0 the brackets and the arrow are omitted.

We will also allow a form of `eval` which sets up multiple processes, possibly communicating with each other through new futures. In this case, k concurrent processes are set up by:. $\mathtt{eval}(p_1(x_{11},\ldots,x_{1m_1})\rightarrow(y_{11},\ldots,y_{1n_1}),\ \ldots,\ p(x_{k1},\ldots,x_{km_k})\rightarrow(y_{k1},\ldots,y_{kn_k}))$. Here, any x_{ij} must be either an existing future to which the originating process is a reader or writer, or a new future, and similarly any y_{ij} must be either an existing future to which the originating process is a writer or a new future. No future may occur more than once in a y_{ij} position, a new future occurring in a y_{ij} position becomes an input future in the originating process, any new future occurring in an x_{ij} position but not a y_{ij} position becomes an output future in the originating process. An error condition occurs if the originating process terminates without binding all its output futures. This keeps the property that every future has exactly one writer and one or more readers.

3. A Language of Rewrite Methods

Having described the coordination interface, we move on to develop a language which is intended to work with this interface. We start by describing a simple rewrite language [Me & Wi 91] for manipulation of futures. A process of type `p` may be described by a header giving the number of input and output futures that it has and their names used internally in `p`, and a set of rewrite methods. A rewrite method has two sides, left and right, the left-hand side representing the conditions needed for a rewrite to take place, the right-hand side giving the rewrite. The left-hand side consists of a list of equality statements, `<Future>=<Tuple>`, where `<Future>` is any input future named in the header which does not occur in another equality statement in the same method, and `<Tuple>` a tuple or constant, whose arguments may be further tuples or constants or new futures. The right-hand side consists of a list of process creations in which the input futures are new futures or any future from the header or any future from tuples on the left-hand side, plus mandatory equality statements `<Future>=<Tuple>` for every output future from the header not used in a process creation. Any future which occurs only on the right-hand side must occur in exactly one output position and one or more input positions, any future from a tuple on the left-hand side must occur at least once in a process creation or a right-hand side equality statement. Right-hand side equalities are the mechanism by which futures actually become bound, and have an alternative form `<Future>⇐<Future>` which binds an output future to an input future. The restrictions ensure that all futures have exactly one writer.

As an example:

```
#p(u,v)→(x,y)
{
 u=f(w)       || r(u,v,w)→(y,x);
 u=g(w)       || q(v,w)→x, y=g(v);
 u=h(w), v=a  || x⇐w, y=b;
 v=e          || x=k(u,z), s(u)→(z,y)
}
```

is the syntax used to indicate that a `p` process waits on two futures `u` and `v`, and is a writer for two futures `x` and `y`. If `u` becomes bound to `f(w)`, where `w` is another future, it creates an `r` process taking `u`, `v` and `w` as inputs (`u` is at this point necessarily bound to `f(w)`, but may still be used as input), and writing to `x` and `y`. If `u` becomes bound to `g(w)` it creates a `q` process taking `v` and `w` as inputs and writing to `x`, while `y` is bound to the tuple `g(v)`. If `u` becomes bound to `h(w)` and `v` becomes bound to the constant `a`, the constant `b` is written to `y` while `x` becomes bound to `w` (if `w` is not itself bound at this point, any reader of `x` will become a reader for `w`). If `v` becomes bound to `e`, `x` is bound to the tuple `k(u,z)` (where `z` is a new future) while the process `s` with argument `u` writes to `z` and `y`. Note that it is possible to determine whether something is a future or a constant by context, thus `x=w` in the place of `x⇐w` would cause `x` to be bound to the constant `w`.

The rewriting is asynchronous and indeterminate. It is assumed that `p` will react to whichever left-hand side condition is met first, but it may make a choice if when it rewrites more than one is applicable, for example if `v` is bound to `e` and `u` to `f(w)`. Rewriting is a commitment, there is no Prolog-like backtracking to try alternatives, which clearly could not be combined with the coordination language role. No ordering is expected on the time at which a process learns of the binding of two independent futures: `u` may be bound to `f(w)` before `v` is bound to `e`, but `p` may learn of `v`'s binding first and rewrite according to its fourth method. If rewriting is dependent on more than one future value, as in the third method above, no commitment to any method is made before all necessary variables are bound. For example, if `v` becomes bound to `a`, `p` is not committed to wait for `u` to become bound to `h(w)`, it will rewrite according to its first method if `u` becomes bound to `f(w)`. Any number of rewrites may take place simultaneously if the system is implemented on a parallel architecture.

4. Rewrite Methods to Processes

At this point our processes hardly deserve the name. They are ephemeral, disappearing as soon as they rewrite. However, this is resolved by introducing a new sort of rewrite method, one in which there is an unwritten recursive call. Such a method is indicated by using a single bar rather than a double one to separate the left and right hand side. By default, the arguments to the unwritten recursive call are the same as those to the original process. However, an equality statement `<Future>=<Tuple>` on the right-hand side where `<Future>` is an input future in the heading causes that future to be replaced in its position by a future already bound to `<Tuple>` in the recursive call. In a recursive rewrite method there is no requirement that a future which is output in the header and not output in any new created processes is written to using an equality statement on the right-hand side, since the recursive call becomes its writer. If an output future is written to, either with an equality statement or by putting it as an output argument to a new process, any input use of that future is taken to refer to a new future which is the output in the equivalent position in the recursive call. Thus,

```
#p(u,v)→(x,y)
{
 u=f(w) | u←w, q(u,w,x)→x;
 u=g(w) | v=a, r(v,x,w)→u;
 v=e   || x←u, y=b
}
```

is a shorthand for:

```
#p(u,v)→(x,y)
{
 u=f(w) || q(u,w,x1)→x, p(w,v)→(x1,y);
 u=g(w) || v1=a, r(v,x,w)→u1, p(u1,v1)→(x,y);
 v=e    || x←u, y=b
}
```

In effect, the arguments to a process may be considered its state, with any changes in the recursive call representing a change of state. We now have something similar to the actors concept [Hewi 77], [Agha 90]. An Actor reacts on receiving a message by doing three actions: *creating* new actors, *sending* messages to other actors, and *becoming* a new actor (changing state). In our language, a process reacts on receiving a future binding by *creating* new processes, *binding* other futures, and *becoming* its recursive call (changing state). The difference between the two is that we separate out values (which may be constants, futures or tuples) from processes, processes not being first-class values, in fact processes have no way to refer to other processes, whereas with Actors, everything is an actor, there is not a separate concept of a value. We also have the ability to define a reaction which takes place only on receipt of multiple future bindings, whereas traditional actors can be specified as reacting only on receipt of a single message, however variants of Actors involving reaction on multiple message patterns have been suggested [Fr & Ag 94]

Our distinction between values and processes makes the foreign language interface, essential for coordination, simpler, since foreign language systems have only to read and write values to be linked into it. As we shall show, it leads to greater flexibility, and in fact process identities as values may be modelled. Another reason for not using the "everything is an actor" approach is given by Kahn and Miller [Ka & Mi 88], who point out the security problems of not knowing what lies behind a simple-looking actor, for example an actor representing a number is indistinguishable from one which behaves like a number but reports any messages it receives to a third party (and thus could be , say, a "Trojan horse" reporting back on the actions of some proprietary algorithm). In our language a number sent to a process is just a number.

5. Futures to Channels

Futures seem to provide once-only communication, rather than the extended communication across channels of notations like CSP [Hoar 78]. However, as Landin pointed out [Land 65], a value which is constructed piece-by-piece can be regarded as a stream of information rather than seen in its entirety. A future may be treated as a stream or channel communicating a value if it is bound to a tuple with two arguments, one the value, the second a further future representing the remainder of the stream or the continuation of the channel. For example, the following represents a simple process which has a single input channel and a single output channel, sending out a `b` when it receives an `a` and a `c` when it receives a `d`:

```
#p(in)→out
{
in=f(x,y), x=a | z=b, in←y, out=f(z,out);
in=f(x,y), x=c | z=d, in←y, out=f(z,out);
in=end        || out=end
}
```

The third method gives a termination condition: the end of the input stream is indicated by it being bound to the constant `end`, when this happens the output stream is similarly ended.

The syntax, however, is awkward, so we introduce further notation, where

```
#p(in)→out
{
in.a | out.b;
in.c | out.d;
in$ || out$

}
```

is the equivalent to the above, except that a standard cons operator is used in the place of the arbitary `f`, and a standard nil constant for the arbitary `end`. In general, `x.<Tuple>` on the left-hand side of a method, where `x` is any input future name, is a shorthand for `x=<Tuple>:x´` combined with a replacement of all occurrences of `x` on the right-hand side by the new name `x´`. The colon is an infix tuple name and `<Tuple>` is a tuple, as we can send any tuple not just constants as in the example above. On the right-hand side of a method, `x.<Tuple>` is equivalent to `x=<Tuple>:x`, where we already have any input use of `x` (which includes the `x` in `<Tuple>:x`) referring to the future in the `x` position in the recursive call. `x$` is equivalent to `x=$` on both left-hand and right-hand side, where `$` is treated as just another constant.

Note that since `in.a` inputs the constant `a`, we need another notation to input any future, and similarly to output one. We use `x?y` to mean `x=y:x,` where `y` is a new future, and `x^y` to output `y` on output channel `x`. So, for example, an indeterminate merger of two channels into one is given by:

```
#merge(in1,in2)→out
{
in1?m | out^m;
in2?m | out^m;
in1$ || out←in2;
in2$ || out←in1

}
```

Input and output of multiple messages is allowed, so `x?a.b` on the left-hand side matches against the input of any value (which may then be referred to as `a`) followed by constant `b` on channel `x`, while `y.b^c.d(a)` on the right-hand side indicates sending the constant `b`, followed by the future `c` followed by the tuple `d(a)` (where `a` is a future) on channel `y`. We also have a lookahead mechanism, enabling a value in a channel to be viewed but not consumed (like Linda's `rd`), so `x?a/.b` on the left-hand side of a method matches only when the constant `b` is the second value on channel `x`, but as it is not consumed the `b` becomes the first item on channel `x` in the recursive call.

6. Representation in a Concurrent Logic Language

At this point we can reveal that the language we have proposed is simply a variant syntax of a restricted form of a concurrent logic language [Shap 89]. Each "method" is equivalent to a clause, replacing the pattern matching of standard logic programming with named argument matching. So the equivalent of the first program we gave is:

```
p(f(W),V,X,Y) :- r(U,V,W,Y,X).

p(g(W),V,X,Y) :- q(V,W,X), y=g(V).

p(h(W),a,X,Y) :- X=W, Y=b.

p(U,e,X,Y) :- X=k(U,Z), s(U,Z,Y).
```

The restriction that a strict input/output moding of arguments is imposed has little effect, since in practice bimodal arguments are hardly ever used in concurrent logic programs. Program clarity is greatly aided by making a syntactical distinction. Our restrictions which force variables to have a single writer just formalises a practice which is generally kept to by programmers in the concurrent logic languages. Enforcing this restriction through syntax aids the production of bug-free programs. The lack of back-communication is however a serious problem since it is commonly used in concurrent logic programming, we shall address this later. The concurrent logic languages continue Prolog's convention of distinguishing between variables and constants by initial capital and small letters. We do not need to do this, since we make the distinction by context.

A major aim of our syntax is to remove the redundancy inherent in the attempt of the concurrent logic languages to hold on to the logic-like syntax of Prolog. This has led to the mistaken view that the concurrent logic languages are "parallel Prologs" and to criticisms based on this mistaken view [Hewi 85], [Ge & Ca 92]. In fact it was recognised early on [Sh & Ta 83] that an alternative view of concurrent logic languages as variants of the object-oriented paradigm is useful for writing practical programs in them, a view we have developed further ourselves [Hunt 95b]. The techniques we have used above of representing mutable encapsulated state by arguments to recursive predicate calls, and channels by tuples containing an argument for further messages are the key to this view. The redundancy in clauses representing objects comes from the need to give every argument in the head of every clause (even though only one is used for pattern matching) and then to repeat them again in the explicit recursive call, or to invent new names to represent changes. We consider the fact that programs in our language have an equivalence in concurrent logic useful though as it provides a semantics for the language. We shall insist that new features retain the convertability to a simple concurrent logic language. It also provides us with an easy method for implementing our language, since efficient implementations of concurrent logic languages now exist [RNC 96].

Our work has some aims similar to that of Foster and Taylor [Fo & Ta 89] who set out to promote a concurrent logic language (which they called "Strand" from "stream AND-parallelism") without placing any emphasis on its logic background. As we have, they emphasised its usefulness as a coordination language. An earlier version of our language [Hunt 95a] was called "Braid", with the idea that braids broke down into strands, though it turns out that name has already been used by others for a concurrent object-oriented language. So we call this latest language "Aldwych", since Aldwych also turns into Strand (on the London street-map). Strand was developed further into a notation called PCN [FOT 92], which like Aldwych makes use of the single-assignment logic variable but replaces pattern-matching in a logic-style clause by named argument matching. However, PCN kept Prolog's lack of syntactic distinction between input and output variables, which we regard as a major limiting factor on its development.

Simple conditional statements may be added to the left-hand side of Aldwych methods translating directly to the guards of flat concurrent logic languages. For example, the following:

```
#ordmerge(in1,in2)→out
{
in1?m1, in2/?m2, m2>=m1 | out^m1;
in1/?m1, in2?m2, m1>=m2 | out^m2;
in1$ || out←in2;
in2$ || out←in1
}
```

gives a an ordered merge of two ordered streams, having the equivalent in a concurrent logic language:

```
ordmerge([M1|In1],[M2|In2],Out) :- M2>=M1 |
      Out=[M1|Out1], ordmerge(In1,[M2|In2],Out1).

ordmerge([M1|In1],[M2|In2],Out) :- M1>=M2 |
      Out=[M2|Out1], ordmerge([M1|In1],M2,Out1).

ordmerge([],In2,Out) :- Out=In2.

ordmerge(In1,[],Out) :- Out=In1.
```

As a further removal of redundancy, we shall allow functional-like embedded calls on the right-hand side of methods. For example, `p(q(x))→y` will be interpreted as `q(x)→z,p(z)→y`. This aids clarity by reducing the name space (no need for new variable `z`) and the need to match output in one place with input in another through an extra variable. Any embedded call will be assumed to have just one output argument. Tuples may be given as arguments by using = as a quoting operator, thus `p(=q(x))→y` is concurrent logic `p(q(x),y)`. The quoting operator works only at one level, so `p(=q(r(x)))→y` is concurrent logic `r(x,z),p(q(z),y)`.

7. Introducing Objects

Actors and concurrent logic languages have been proposed as the two language paradigms most suitable as a foundation on which to design languages suitable for open systems [Ka & Mi 88], [He & Ag 90]. The advantage offered by actors of the clarity of direct representation of objects is offset by the greater flexibility of direct representation of channels and the possibility of multiple input ports in concurrent logic languages [Kahn 89]. Another balancing factor is that actors offer many-to-one communication directly but one-to-many communication is awkward and must be managed by the programmer, while with concurrent logic languages it is the other way round. We have overcome some of the problems of the awkward syntax of the concurrent logic languages, but we now go further and introduce syntax which brings us the advantages actors have without losing the advantages of concurrent logic languages. This is a different approach from several attempts to build actor languages on top of concurrent logic languages – as noted by Kahn [Kahn 89] these tended to lose the advantages of the underlying logic language by approaching actors too closely, in some cases necessitating a bilingual approach in which the programmer had to mix in concurrent logic language code.

In a concurrent logic language, several objects may access one object if each has a channel to which it writes to, and these channels are merged to form a single channel which the object being accessed reads. Each writing object may regard its channel as a handle on the reading object. Thus:

```
p(pargs)→s1, q(qargs)→s2, r(rargs)→s3,
merge(s1,s2)→s4, merge(s4,s3)→s0, s(s0,sargs)
```

Here, the `p`, `q` and `r` objects have `s1`, `s2` and `s3` as their respective handles on the `s` object, with the `s` object receiving a single stream of messages from them in `s0`. To simplify this, we add object handles as a different type of value in our system, for simplicity distinguished from futures by names beginning with capital letters. References to an object handle are converted into channels with mergers added as appropriate for implementation. The above could be given as:

```
p(pargs)→S, q(qargs)→S, r(rargs)→S, s(S,sargs)
```

but this is counter-intuitive. `S` should be viewed as an argument and hence an input of the `p`, `q`, and `r` objects. The handle of an object is better seen as an output from its creation. So we write the system of a `p`, `q` and `r` object having common access to an `s` object as an acquaintance as:

```
p(S,pargs), q(S,qargs), r(S,rargs), s(sargs)→S
```

This is converted into the form with merged channels automatically for implementation. Syntactically, object handle variables are treated like future variables, for instance as above they must be in exactly one output position, but may be in any number of input positions; they may be assigned to each other (though object may not be assigned to futures and vice versa). A header argument input is an output to a method, so:

```
#p(u,v,S,T)→H
{
u=f(w) | q(w,S)→u, r(v,w)→T;
u=g(w) | r(v,w)→H, s(H,T)→(u,v);
u=h(R) | S←R, q(x,S)→u, s(R,T)→(x,v)
}
```

becomes on making the recursion explicit:

```
#p(u,v,S,T)→H
{
u=f(w) || q(w,S)→u1, r(v,w)→T1, p(u1,v,S,T1)→H;
u=g(w) || r(v,w)→H, s(H1,T)→(u1,v1), p(u1,v1,S,T)→H1;
u=h(R) || q(x,S)→u1, s(R,T)→(x,v), p(u1,v,R,T)→H

}
```

and then with the conversion of object handles to channels and the addition of the merges:

```
#p(u,v,h)→(s,t)
{
u=f(w) || q(w)→(s1,u1), r(v,w,t1), p(u1,v,h)→(s2,t1), merge(s1,s2)→s, t$;
u=g(w) || r(v,w,h), s()→(h1,t1,u1,v1), p(u1,v1,h1)→(s,t2), merge(t1,t2)→t;
u=h(r) || q(x,s)→u1, s()→(r1,t1,x,v), p(u1,v,h)→(r2,t2), merge(r1,r2)→r,
            merge(t1,t2)→t, s$

}
```

Note how an input object which is unused converts to an output channel which is closed by assigning the nil list to it.

The third method above causes us a problem. We wanted to deal with a case where an input tuple contained an object handle, but the reversal of input/output polarity in the conversion of object handles to channels would require back communication (the `r` in `h(r)` should be used for output) which we have ruled out. To prevent this we do not allow futures to be assigned tuples which contain object handles or to be used as channels to send messages containing object handles, so the third method is actually invalid Aldwych.

However, note how the rule that object handles, like futures, have only one writer means they convert to channels with only one reader. This means we *can* allow them to be used for back-communication. The objection to back-communication was that the one-to-many communication meant that a message intended for back-communication could become duplicated and thus the back-communication slot would have multiple writers. As this is not the case with channels derived from objects, we allow messages to be sent on object handles which do have reply slots and which have arguments, both input and/or reply, which are themselves object handles. This is perhaps the key result in our work – it enables us to provide a flexible language which encompasess objects as first-class citizens, but which retains the flexibility of the underlying concurrent logic language.

The syntax for reading and writing message to and from object handles is the same as that for reading one from a channel, but messages are read from an output object handle. Messages take the form `<Name>(<Input arguments>)→<Output Arguments>`, with argument names beginning with capital letters if they are intended to be object handles, and the → omitted if there are no output arguments. All output arguments on the left-hand side of a method must be given a value on the right-hand side by occurring exactly once in an output position. Messages may be sent on both input and output object handles, the latter being equivalent to "sending a message to self" in standard object-oriented terminology, working by joining the message to the front of the incoming stream that forms the output handle. Thus:

```
#p(a,b,Q)→H
{
H.f(x)→(y,z)  | q(a,b)→y, r(x,Q)→z;
H.g(c,R)→S    | p(q(c,b),b,R)→S, b←c;
H.h(c)→d      | Q.m(c,t(a,Q))→d;
H.r(R)        | Q←R, s(b,Q,R);
H.s(c)→S      | H.g(c,Q)→S;
H$           || Q.bye(a)
}
```

defines a `p` process which has two input values and one input object and outputs one object handle. It reacts to five different sorts of messages received on the object handle it outputs:

`f`: has one input value and two output values; `p` computes these values using `q` and `r` processes.

`g`: has one input value, one input object and one output object; `p` sets up the output object by setting up another `p` object, and replaces its own `b` value by the value input in the message.

`h`: has one input value and one output value; `p` sends an `m` message to its own input object to obtain the output value, with an extra value produced by a new `t` process.

`r`: has just an input object; `p` changes its own input object to the new input object, and sets up a new `s` process (note though this process has no output it can affect the rest of the system by sending messages to its object inputs).

`s`: has an input value and an output object; `p` gives a value to the output object by sending itself a `g` message.

The final method may be considered as providing a destructor function. It tells the `p` process what to do when there are no references anywhere to its output handle, in this case send a final `bye` message containing its `a` argument to its `Q` input object argument.

Making the recursion explicit, converting the objects to tuples (including syntactically incorrect back-communication messages to channels, indicated by italics), and expanding the embedded calls gives the following as the equivalent to the above:

```
#p(a,b,h)→q
{
h=f(x,y,z):h1 || q(a,b)→y, r(x)→(q1,z), p(a,b,h1)→q2, merge(q1,q2)→q;
h=g(c,s,r):h1 || p(d,b,s)→r, p(a,c,h1)→q, q(c,b)→d;
h=h(c,d):h1   || q=m(c,e,d):q3, p(a,b,h1)→q1, t(a)→(q2,e), merge(q1,q2)→q3;
h=r(r):h1     || s(b)→(q,r1), p(a,b,h1)→r2, merge(r1,r2)→r;
h=s(c,s):h1   || h2=g(c,s,q1):h1, p(a,b,h2)→q2, merge(q1,q2)→q;
h=$           || q=bye(a):$
}
```

Messages may be received on object handles using `?`, and sent using `^`, with the rule that any message received using `?` must be referred to again exactly once in a message send using `^`, and no where else. This is because such a message is not a normal tuple as it may contain object arguments and back-communication arguments.

The coordination mechanism would need to be extended so that systems written in other languages can be treated as Aldwych objects. A simple syntax for receiving messages on an object handle would suffice.

8. Further Constructs

Some further constructions are useful in building up the language. In particular, we wish to move it to a direction where programs which are close in appearance to functional or object-oriented code are also valid Aldwych.

We allow arguments to tuples and messages and ← assignments on the right-hand side of methods to be full expressions built up using the standard arithmetic symbols as infix operators, and also ++ as infix list append and <> as infix list merge. Merging given by <> is indeterminate, in time order for dynamically produced streams, but we also have infix << for biased merge (see below). Values in expressions of the form `<Name>(<Args>)` are replaced by a new variable name, say `X`, with the call `<Name>(<Args>)→X` added to the statement, while `<Name>.<Name>(<Args>)` is replaced by `X` with the object message send `<Name>.<Name>(<Args>)→X` added. In both cases there is no need to be concerned whether `X` represents a value or an object as it has just one reader and one writer. The ^ notation used to send messages on channels is extended so that if its right-hand argument is an expression the value of that expression is sent on the channel, it may not however be used if the left-hand argument is an object name and the right-hand argument anything but a variable name representing a previously read object message.

Input arguments on the left-hand side and output arguments on the right-hand side of a method may be replaced by an underscore, giving an "anonymous variable" effect as in Prolog, so that values may be produced but not used (for example, if only one of several outputs is needed). The reverse is not permitted so as to ensure every variable used as input has a writer. Otherwise a variable which is produced as output or read as input, but not used, is an error.

Lookahead is permitted on object messages as well as channel messages, but as an object message with replies is not consumed, the replies being provided by its consumer, the replies are treated as additional input, not output to the method.

A facility for declaring arguments to an object to be readable and/or writeable is given. This can give the effect of public variables, as in C++, but is a lot more flexible. If a variable is followed by `?` in the header, it is publicly readable, if followed by `^`, it is publicly writeable. These annotations are a shorthand for declaring methods in which messages of the same name as the argument are received on its input handles. For example,

```
#p(a?,b^,c?^)→(P,Q) {}
```

is equivalent to:

```
#p(a,b,c)→(P,Q)
{
P.a→x | x←a;
Q.a→x | x←a;
P.b(x) | b←x;
Q.b(x) | b←x;
P.c→x | x←c;
Q.c→x | x←c;
P.c(x) | c←x;
Q.c(x) | c←x;
P$, Q$ ||
}
```

This also shows an implicit destructor method (one with empty right-hand side) which is added when the outputs are all object handles and no alternative destructor is given. The greater flexibility than a simple public/private distinction is given by allowing `?` and `^` to be annotated with the names of the output handles to which they apply (if none is given, it is assumed they apply to all). So:

```
#p(a?{P},b^{Q},c?{Q}^)→(P,Q) {}
```

is equivalent to:

```
#p(a,b,c)→(P,Q)
{
P.a→x | x←a;
Q.b(x) | b←x;
Q.c→x | x←c;
P.c(x) | c←x;
Q.c(x) | c←x;
P$, Q$ ||
}
```

Clearly it is possible to provide any number of classes of privileged access. Extending this, messages read from sets of output object handles or channels are allowed, which indicate a read from any one of the members of the set. So, for example, the method:

```
{P,Q}.message | statement;
```

is a shorthand form of:

```
P.message | statement;
Q.message | statement;
```

Another shorthand allows conditions which from part of a left-hand side to be spread across several methods, so

```
condition1 [ condition2 | statement1;
             condition3 | statement2
           ]
```

is shorthand for:

```
condition1, condition2 | statement1;
condition1, condition3 | statement2;
```

There is no ordering on these conditions following the expansion, so `condition1` may refer to variables which is an argument of a message received in `condition2` and `condition3` (so long as it is in both), for example `condition1` could be `a>b`, `condition2` could be `X.m(a,b)` and `condition3` could be `Y.f(a).g(b)`. Embedded occurrences of this construct may be used.

The next modification removes the need for an explicit name for an output variable. The method

```
S.p(a)-|> <Value>, …
```

where `-`, `|` and `>` are separate symbols of the language, but the possibility of their combination into an arrow-like shape is deliberate, is equivalent to

```
S.p(a)→x | x←<Value>, …
```

The `p(a)-` message pattern may occur anywhere on a left-hand side, so long as there is only one anonymous return on that left-hand side. `<Value>` is any expression evaluating to a value.

Message patterns may be given without being attached to a particular output handle, in which case it is taken to be a shorthand for methods attaching them to every handle, so

```
#p(sum,count)→(Priv,Ord)
{
inc(x)      | sum←sum+x;
Priv.dec(x) | sum←sum-x;
balance    -|> sum, count←count+1
}
```

is a shorter form of:

```
#p(sum,count)→(Priv,Ord)
{
Priv.inc(x)     | sum←sum+x;
Ord.inc(x)      | sum←sum+x;
Priv.dec(x)     | sum←sum-x;
Priv.balance→r | r←sum, count←count+1;
Ord.balance→r  | r←sum, count←count+1;
Priv$, Ord$ ||
}
```

9. Cells, Pointers and Functional Programming

Objects may be linked together with handles in a way similar to the linking of cells with pointers in standard imperative programming data structures. So:

```
#empty()→Self
{
Isempty-|>=true;
Cons(X)-|>list(X,Self)
}

#list(Head?,Tail?)→Self
{
Isempty-|>=false;
Cons(X)-|>list(X,Self)
}
```

provides lists of objects using the linked list method. If `L` represents a list, then the head of the tail of the tail of the list (assuming its length is known to be more than two) is obtained by `((L.Tail).Tail).Head`, while a list of three objects, `A`, `B` and `C` is obtained by `((empty().Cons(A)).Cons(B)).Cons(C)`. Note that we allow `f(<Args>).<Message>` to be the equivalent of `f(<Args>)→X,X.<Message>`, and `(<Expression>).<Message>` equivalent to `X.<Message>` where `<Expression>` evaluates to `X`.

The syntax `L~Tail~Tail~Head` and `empty()~Cons(A)~Cons(B)~Cons(C)` is provided as an alternative to the above, where in general, `L~f(<Args>)` is a value which, as with `L.f(<Args>)` before, evaluates to `X`, where `L.f(<Args>)→X` is added to the right-hand side of a method. However, in this context any further message sent is taken to be sent not to `L`, but to the reply from `f(<Args>)`, so while the value `L.f(<Args>).g(<Args>)` is equivalent to `X` where `L` is sent first `f(<Args>)` then `g(<Args>)→X`, `L~f(<Args>)~g(<Args>)` is equivalent to `Y` where `L` is sent `f(<Args>)→X`, then `X` is sent `g(<Args>)→Y`.

Juxtaposition of two values is also used as a syntax for message sending, with the distinction between juxtaposition and `~` roughly equivalent to that between using `^` and using `.` for sending messages, that is it sends a message which is the result of an evaluation. The message sent is a nil-name message with reply, in other words it matches the left-hand side pattern `(<Args>)→R` (empty tuple name, one reply). So, for example, `d ← (f(a) P.m(b) g(c))` is equivalent to `(f(a)→X, P.m(b)→Y, X.(Y)→Z, g(c)→W, Z.(W)→d)`. Juxtaposition is left-associative, and this means there is a distinction between the expressions `F(<Expression>)` and `F (<Expression>)` (with a space between the `F` and the bracket) with the former treating `F` as a process name, taking one input argument the value of `<Expression>` and having one output argument which is returned as the value of the expression, while the latter treats it as a variable representing an object to which the value of `<Expression>` is sent as the input argument to a nil-name message, the output argument being the value of the expression. The effect is to enable functional programing in the higher order style [Bi & Wa 88] to be embedded within Aldwych.

A function is an entity which when presented with a value returns a result, and in the higher order style such an entity is itself a first class citizen of the language. An object which is restricted to taking nil-messages and which does not change its internal state at any stage may be regarded as a function. An important aspect of functional programming is the idea of currying where a function may be supplied with a partial number of its arguments and return a function specialised to those arguments, in general given a function `f` which takes n arguments, `f` a_1 a_2 … a_{n-1} returns a function `f'` which when supplied with the final argument a_n gives the same result as `f` when given all n arguments a_1 … a_n. A similar argument applies to `f'`, and so on, hence functions need only take one argument supplied by juxtaposition.

In Aldwych, a header of the form:

`#p(`a_1`,…,`a_k`)[`a_{k+1} … a_{k+n}`]→(`v_1`,…,`v_m`)`

followed by a set of methods appropriate for a header of the form `#p(`a_1`,…,`a_{k+n}`)→(`v_1`,…,`v_m`)` translates to a set of process declarations, starting with:

```
#p(a₁,…,aₖ)→Self
{
Self.(Val)→Return | p₁(a₁,…,aₖ,Val)→Return;
Self$ ||
}
```

which could be written in the briefer style introduced above:

```
#p(a₁,…,aₖ)→Self
{
(Val)-|> p₁(a₁,…,aₖ,Val)
}
```

and follows with (in the briefer style again):

```
#p₁(a₁,…,aₖ,aₖ₊₁)→Self
{
(Val)-|> p₂(a₁,…,aₖ,aₖ₊₁,Val)
}
```

and so on, ending with with one with header `#`p_n`(`a_1`,…,`a_{k+n}`)→(`v_1`,…,`v_m`)` and the original body. So if `P` is the output of p_m`(`a_1`,…,`a_{k+m}`)`, `P X` will be `Y` where the message `(X)→Y` is sent to `P`, which is the output of p_{m+1}`(`a_1`,…,`a_{k+m}`,X)`, which is `P` specialised with the next argument set to `X`, and may be used as a first class object.

10. Initialisation, Default Values and Delegation

Many concurrent logic languages have an "otherwise" construct which separates clauses. Computation may only commit to a clause following the "otherwise" if variables have been sufficiently bound to rule out the possibility of committing to any clauses before it. We use the colon character as Aldwych's "otherwise" (we are keeping to the rule that language constructs use non-alphanumeric characters only), thus:

```
#max(a,b)<
{
a>b ||> a;
:
    ||> b
}
```

Here the otherwise gives us a form of if-then-else. This also shows an extension to the "anonymous" return (< is used for anonymous return of a value in a header), allowing the return value for a whole process to be anonymous, giving a more functional appearance to the code. The symbol < may be used as a value elsewhere if it is necessary to use the anonymous output of a recursive call as input.

Delegation [Lieb 87] has been recommended as an alternative to inheritance in a concurrent object-oriented system. The idea is that one of the arguments to an object is another object termed a "proxy", any messages an object cannot handle are passed to its proxy. Here is an example of its use to define the class of royal elephants, which are like ordinary elephants in every way except their colour is white:

```
#RoyalElephant(Proxy)~
{
colour-|>=white;
:
?m | Proxy^m
}
```

The ~ indicates an anonymous object handle return, the above could be written more fully

```
#RoyalElephant(Proxy)→Self
{
Self.colour→return | return=white;
:
Self?m | Proxy^m
}
```

A ~ as a value in an expression is used to mean `Self` as above, similar to < as above, while ~`<Mess>` is equivalent to `Self.<Mess>`. The delegation effect depends on royal elephant objects being declared with an appropriate proxy. We introduce a syntax allowing objects to have local variables which are initialised when the objects are created. In this case, a royal elephant class is declared by:

```
#RoyalElephant(Args)~
= Elephant(Args)→Proxy, <(Proxy)
{
colour-|>=white;
:
?m | Proxy^m
}
```

In general, an = following a header indicates there is some initialisation code, this code must write values to all local input arguments, and use all local output arguments. An argument may be local to just the initialisation code provided it has exactly one writer and at least one reader within that code. The < indicates the declaration of local arguments, which are treated in the methods just like other arguments to the object. In effect, any call `RoyalElephant(Args)→E` is transformed to one `Elephant(Args)→Proxy, RoyalElephant'(Args,Proxy)→E`, where `RoyalElephant'` has the methods declared for `RoyalElephant`. It is assumed that `Args` hold the arguments needed for any elephant. So each royal elephant object has its own elephant object which acts as its proxy.

An alternative way of declaring this is:

```
#RoyalElephant(Args,Proxy←Elephant(Args))~
{
colour-|>=white;
:
?m | Proxy^m
}
```

Here `Proxy` is an argument with a default value. In general, a default value argument is declared in a header by following it with ←`<Expression>`, where `<Expression>` is any expression using the arguments of the header as input. It has a similar effect to a local variable, but the default may be overridden when an object is created, thus the call `RoyalElephant(Args)→E` works as before, but `RoyalElephant(Args,Proxy←FunnyElephant(Args))→E` creates a royal elephant object whose proxy is a funny elephant object, in other words it is equivalent to `FunnyElephant(Args)→Proxy, RoyalElephant'(Args,Proxy)→E`.

Output handles may have the default value `$`. Consider the case where there are three classes of access to a `p` object, `A`, `B` and `C`. Its header would be `#p(<Args>)→(A,B,C)`, and if we wish to produce an object called `PwithA` having `A`-access to `P` only, we write `p(<Args>)→(PwithA,_,_)`. However, using the default we have header `#p(<Args>)→(A$,B$,C$)` and the `A`-access is given by `p(<Args>)→(A→PwithA)`. Knowing how to access an object in a particular way involves knowing the name of the relevant output handle argument. Such names could be withheld in public for security reasons.

Note that == is used for initialisation where there is no body, in effect declaring a macro, for example (using anonymous output):

```
#square(x) <==> x*x;
```

This can also be seen as a way of avoiding "single method" processes like:

```
#square(x)<
{
||>x*x
}
```

11. More Control

The next addition to the language deals with the fact that programs in concurrent logic languages tend to have a tangled structure of mutually recursive procedure. We allow anonymous embedded procedures, with their bodies written at the end of right-hand side of the method where they are called. This reduces the name space of procedures, and takes away the goto-like jumping between the mutually recursive procedures. In general, an embedded procedure is declared by adding a set of methods, with the same syntax as top-level methods, at the end of the method which calls it. The embedded procedure has the same header (including local arguments) as the calling procedure with the addition of the anonymous return if there is one and it hasn't been given a value. A call to it is made in the place of the recursive call, with the arguments the recursive call would have had. So, the following declares a process with one input and one output channel which reads a series of constants, deleting those which are enclosed by the constants `stop`: and `start`:

```
#delbetween(in)→out
{
in.stop | {
           in.start || ;
           :
           in?m     | ;
           in$      ||| out$
          }
:
in?m | out^m;
in$ || out$
}
```

A double bar on an embedded procedure method indicates that recursion reverts to the embedding procedure, a triple bar means that there is no recursion. In general, given there is no limit on the level of embedding, the number of bars is one greater than the number of embeddings which are exited. Using mutual recursion would make the above:

```
#delbetween(in)→out
{
in.stop || delbetween1(in)→out;
:
in?m | out^m;
in$ || out$
}

#delbetween1(in)→out
{
in.start || delbetween(in)→out;
:
in?m     | ;
in$      || out$
}
```

Embedded procedures may have local variables which are declared in a similar way to local variables for top-level procedures, so, for example, the following is similar to the above but outputs a separate stream which sends a list of the number of characters deleted on after each occurence of stop:

```
#delcountbetween(in)→(out,count)
{
in.stop | num←0,<(num)
         {
          in.start ||  count^num;
          :
          in?m     |   num←num+1;
          in$      ||| count^num$, out$
         }
:
in?m | out^m;
in$ || out$, count$
}
```

Here the equivalent is:

```
#delcountbetween(in)→(out,count)
{
in.stop || delcountbetween1(in,0)→(out,count);
:
in?m | out^m;
in$ || out$, count$
}

#delcountbetween1(in,num)→(out,count)
{
in.start || count^num, delcountbetween(in)→(out,count);
:
in?m     |  num←num+1;
in$      || count^num$, out$
}
```

Any output local variable must have an explicit writer in any method whose number of bars is such that the variable is no longer in scope and hence not an argument to the mutually recursive call.

A variant of the embedded procedure gives an if-then-else construct (which could be expressed awkwardly using the previous syntax), which overcomes some of the problems of "flat" guards (conditionals on the left hand side of methods being only system primitives, not user-defined expressions). The if-then-else takes the form `<Decl>?<Cond>:<Statement>:<Statement>`. Here `<Decl>` is an optional statement of local variables, taking the same form as previous local declarations, `<Cond>` is an expression which evaluates to `true` or `false`, and `<Statement>` has the same structure as the right-hand side of a method (including the possibility of further embedding). `<Cond>` as part of the statement preceding the if-then-else, thus having variable values as input into the procedure, but the `<Statement>`s describe the embedded procedure, and thus have variable values as changed by any assignments. The embedded procedure has an extra argument, the result of the expression `<Cond>`, and just two non-recursive methods given by the `<Statement>`s, the first taken if `<Cond>` evaluates to `true`, the second if it evaluates to `false`. As an example, here is a version of list filtering, using the linked list form of lists given above, and the filtering function passed in as an argument:

```
#filter(Func,InList)→OutList
{
|| ? InList.Isempty
        : Empty()→OutList;
        : InList.Head→H.Tail→T,<(H,T)
          ? Func H
              : filter(Func,T).Cons(H)→OutList;
              : filter(Func,T)→OutList
}
```

A more concise form involving implicit recursion and anonymous output is:

```
#filter(Func,InList)~
{
| ? InList.Isempty
       :| <=Empty();
       :  InList.Head→H.Tail→InList,<(H)
          ? Func H
              : <=~Cons(H);:
}
```

where <= is a new construct introduced to assign a value to the anonymous object output, equivalent to Actors "become".

Without the embedded procedures, this would be:

```
#filter(Func,InList)~ ==> filter1(InList.Isempty,Func,InList);


#filter1(t,Func,InList)→OutList
{
t=true  || Empty()→OutList;
t=false || InList.Head→H, filter2(Func H,H,Func,InList.Tail)→OutList
}

#filter2(t,H,Func,InList)→OutList
{
t=true  || filter(Func,InList).Cons(H)→OutList;
t=false || filter(Func,InList)→OutList
}
```

A further control structure gives sequencing. A statement may take the form takes the form `+<Decl><Statement>;<Statement>`, with `<Decl>` again an optional declaration of further local variables taken from the first of the `<Statement>`s added to the arguments of the embedded procedure, and the second `<Statement>` being the sole method, entered without condition, of this embedded procedure. Again, the `<Statement>`s may contain further embedding. For example, here is a method which on receipt of a 1-input 0-output-argument message `stop` on handle `Self` stacks up all further messages until it receives a 0-input 1-output message `start`, returns the argument of the `stop` message to the sender of that `start` message, and then pulls the messages back off the stack, and stacks them on `Self` for reuse:

```
Self.stop(mess) |+<()→Stack, <(mess,Stack)
                  {
                   Self.start-||>mess;
                   :
                   Self?m | Stack^m
                  };
                  {
                   Stack?m | Self^m;
                   Stack$ ||
                  }
```

`Stack` is an output handle from the second embedded loop, and an input to the first. This example shows how a local object may be used as a message stack. In [Hunt 95a] we give a similar example showing how it may be used as a queue. It shows more generally that message queues can be handled, we are not restricted to dealing with messages in the order they arrive. The translation to non-embedded processes makes the method:

```
Self.stop(mess) || stackup(Args,mess,Stack)→Stack;
```

with definitions:

```
#stackup(Args,mess,InStack)→(Self,OutStack)
{
Self.start-||>mess, unstack(Args)→(Self,OutStack);
:
Self?m | InStack^m
}

#unstack(Args)→(Self,OutStack)
{
OutStack?m | Self^m;
OutStack$ || orig(Args)→Self
}
```

where `orig` is the name of the original process of which the original method is part, assuming *`Args`* are its inputs and `Self` its only output.

12. Locks and Priorities

We have stated that multiple references to objects are converted to merged channels, so for example

```
#p(a,b,P)→Q
{
|| f(a,P,R)→c, g(b,Q)→d, h(c,P)→R, k(d,P,Q,R)→Q
}
```

which could be written

```
#p(a,b,P)~ ==>Q, f(a,P,R)->c, g(b,Q)->d, h(c,P)->R, k(d,P,Q,R)->Q;
```

becomes

```
#p(a,b,q) <==> p,
  f(a)->(p1,r1,c), g(b)->(q1,d), h(c,r)->p2, k(d,q2)->(p3,q3,r2),
  merge(q,q1)->q4, merge(q3,q4)->q2, merge(p1,p2)->p4, merge(p3,p4)->p,
  merge(r1,r2)->r;
```

which gives equal access to all sharers of an object, but we can give the effect of locking an object to a process and excluding other processes from accessing it by using append in the place of merge. Consider the following:

```
#p(a,b,q) <==> p,
  f(a)->(p1,r1,c), g(b)->(q1,d), h(c,r)->p2, k(d,q2)->(p3,q3,r),
  merge(q,q3)->q4, append(q1,q4)->q2, merge(p1,p3)->p4, append(p4,p2)->p;
```

Here only when the `g` process has terminated the channel `q1` (which it will do when it has no more references to it) and all `q1`'s messages have been processed will the `k` process reach messages coming to it on channel `q3` from `q` and `q2`. `g` has a high priority lock on the object `Q`. On the other hand, messages on `p2` will be handled on `p` only when `p1` and `p3` have been terminated. `h` has a low priority lock on object `P`. We prefix an object reference by `!:` to give it a high priority lock, and `&:` to give it a low priority lock, so the above pattern of appends and merges is given by:

```
#p(a,b,P)~==>Q, f(a,P,R)->c, g(b,!:Q)->d, h(c,&:P)->R, k(d,P,Q,R)->Q;
```

Note that the locking of the `P` object is only within the processes initiated by the `p` process. When `h` has access to `P`, it would still share it with any process granted access to it outside `p`. Since `p` is itself the provider of the `Q` object, however, it can grant an absolute lock on it, as it can to the purely local `R`. A `!!:` will give priority higher than a `!:`, and so on, adding extra `!`s and `&`s as necessary. `!:P` as a statement will priority lock the reference to `P` in an implicit recursive call. `P1<-!P` enables a group of processes to share `P` using the name `P1` and lock out other processes.

If the colon is omitted from the locking notation, the object references are combined using a biased merge rather than an append. A biased merge is one which is indeterminate, but given a choice of two possible mergers will always choose one over the other. Some concurrent logic languages make use of a primitive `var` or `unknown` which succeeds if its single argument is an unbound variable, and fails otherwise, thus:

```
merge([H|T],L,M) :- M=[H|M1], merge(T,L,M1).

merge(L,[H|T],M) :- var(L) | M=[H|M1], merge(L,T,M1).

merge([],L,M) :- M=L.

merge(L,[],M) :- M=L.
```

is a merger which is biased towards the first stream since it will only accept items from the second when the first is an unbound variable, that is there are no items ready to accept on it. The concurrent logic language KL1, however, makes use of a construct called `alternatively`, making the above:

```
merge([H|T],L,M) :- M=[H|M1], merge(T,L,M1).

alternatively.

merge(L,[H|T],M) :- var(L) | M=[H|M1], merge(L,T,M1).

merge([],L,M) :- M=L.

merge(L,[],M) :- M=L.
```

Clauses following `alternatively` are used only when arguments are insufficiently bound to enable commitment to any clauses before it. Unlike `otherwise`, though, there is no wait for variables to become bound, computation may commit to a clause following `alternatively` even if variable binding would eventually allow commitment to one before.

Methods may be prioritised in Aldwych by preceding them with one of more `!`s for high priority and `&`s for low priority, the prioritising working by introducing the equivalent of KL1's `alternatively` between sets of clauses with different priority. Thus a biased merge equivalent to the above could be written as:

```
#biasedmerge(c1,c2)→out
{
! c1?m | out^m;
  c2?m | out^m;
  c1$ || out←c2;
  c2$ || out←c1
}
```

The introduction of priorities to methods marks the point at which we move away from features which are ultimately describable in the logic of concurrent logic languages. We believe that the core of the language must have a logic semantics and thus do not consider in detail more extralogical features. As we have pointed out in another paper [Hunt 91] the introduction of a prioritising mechanism over processes is useful in some cases, such as search algorithms; KL1 has a simple priority mechanism similar to the one we proposed. For practical use as a coordination language, a notation which gives direct control of the mapping of processors to processes is necessary. We propose the symbol `@`, attached to process creation for this, as in PCN [FOT 92], but used earlier for implementing systolic algorithms in a concurrent logic language [Shap 84].

Further work on the incorporation of much more sophisticated forms of control over computational resources than our simple numerical priority may be necessary in a language which is to obtain practical use in implementing open systems. Kahn and Miller [Ka & Mi 88] note the incorporation of mechanisms to deal with hardware failure is an important area for further research if actors and concurrent logic languages are to be used in this context. Miller and Drexler [Mi & Dr 88] take the issue of hardware control further by suggesting that the encapsulation and trade of hardware resources by objects, analogous to the way object-oriented programming confers ownership of software resources, is a necessity in the future development of open systems languages. The “funding” mechanism of Spawn [WHHK 92], an experimental system inspired by these ideas, suggests a way in which prioritising may be made more sophisticated.

In an open system, as Kahn and Miller note [Ka & Mi 88], the global association of process names with process declarations is unacceptable, we would expect a local association, though some global conventions are necessary. As they suggest, this indicates the development of a module system is necessary (in fact practical concurrent logic languages like KL1 and Strand have a simple module system). If the language is to be used for multi-agent systems, restrictions need to be placed on the types and structures of messages that are exchanged, and the structures of the state of objects, for example as proposed by Shoham [Shoh 93] who develops a language for Agent-Oriented Programming from an Actors-like basis.

13. Conclusion

The origins of Aldwych lie with attempts to package concurrent logic languages in a more palatable way. The author was impressed with the flexibility of these languages for expressing parallel algorithms and abstract concepts in concurrency that were not easily expressed in other languages, yet was concerned the languages were attracting little interest outside their own developers. The resulting language developed as one concept followed from another and the language took on a life of its own. What is presented here amounts to a jotting down of some of the ideas that emerged rather than a fully-fledged design for a language, but what can be seen emerging is a language which fits in closely with Hewitt and Agha’s Actor concepts [Hewi 77], [Agha 90].

One aim throughout has been to keep to the principle that all programs in the language may be converted to a concurrent logic form, providing a logical semantics. Another aim has been to avoid constricting the flexibility of the underlying logic language, thus avoiding the problems noted by Kahn [Kahn 89] with previous attempts to build a concurrent object-oriented language on top of a concurrent logic language. The main driving force was a desire to cut down the name space of concurrent logic programs by making a much greater use of embedded code structures, and to exploit what could be achieved once it is accepted that variables in concurrent logic language almost always have an input/output direction.

The coordination interface is important, not only for the growing importance of coordination as we move into the age of open systems, but also because it enables many practical aspects (not only input/output of text and graphics, but also, for example arrays) to be separated out and brought in with library procedures. These could be implemented in other languages, but would fit in smoothly so long as they have the limited interface required.

14. References

[Agha 90] G.Agha. Concurrent object-oriented programming. *Comm. ACM 33*, 9 pp.125-141.

[Bi & Wa 88] R.Bird and P.Wadler. *Introduction to Functional Programming*, Prentice Hall.

[Ca & Ge 89] N.Carriero and D.Gelernter. Linda in context. *Comm. ACM 32*, 4 pp.444-458.

[Ci & Ha 96] P.Ciancarini and C.Hankin (eds) *Co-ordination Languages and Models*, Springer *LNCS 1061*.

[Clar 88] K.L.Clark. PARLOG and its applications. *IEEE Trans. Soft. Eng. 14*, 12 pp.1792-1804.

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

[FOT 92] I.Foster, R.Olson and S.Tuecke. Productive parallel programming: the PCN approach. *Scientific Programming 1*, 1 pp.51-66. Reprinted in *Programming Languages for Parallel Processing* D.B.Skillicorn and D.Talia (eds), IEEE Press, 1995.

[Fr & Ag 94] S.Frølund and G.Agha. Abstracting interactions based on message sets. In In *Object-Based Models and Languages for Concurrent Systems* P.Ciancarini, O.Nierstratz and A.Yonezawa (eds), Springer *LNCS 924*, pp107-124.

[Gele 85] D.Gelernter. Generative communication in Linda. *ACM Trans. Prog. Lang Sys. 7*, 1 pp.81-112.

[Gele 89] D.Gelernter. Multiple tuple spaces in Linda. In *Parallel Languages and Architectures Europe (PARLE-89)*, E.Odijk, M.Rem, J-C.Syre (eds) Springer *LNCS 366* pp.20-27.

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

[Hals 85] R.H.Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Prog. Lang Sys. 7*, 4 pp.501-538.

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

[Hewi 85] C.Hewitt. The challenge of open systems, *Byte*, April 1985, pp.223-233.

[He & Ag 90] C.Hewitt and G.Agha. Guarded horn clauses: are they deductive and logical? In *Artificial Intelligence at MIT Vol.1*, P.H.Winston and S.A.Shellard (eds), MIT Press, pp.582-593.

[He & Li 84] C.Hewitt and H.Lieberman. Design issues in parallel architectures for artificial intelligence. In *Proc. 28th IEEE Computer Society Int. Conf,*, San Francisco CA, pp.418-423.

[Hoar 78] C.A.R.Hoare, Communicating sequential processes. *Comm. ACM 21*, 8 pp.666-677.

[Hunt 91] M.Huntbach. Parallel Branch and Bound Search in Parlog. *Int. J. of Parallel Programming 20*, 4 1991 pp.299-314.

[Hunt 95a] M.Huntbach. The concurrent object-oriented language Braid. In *ACM Symp. on Applied Computing*, Nashville TN, pp.140-146.

[Hunt 95b] M.Huntbach. An introduction to RGDC as a concurrent object-oriented programming language. *J. Obj. Oriented Prog.*, Sep. 1995.

[Kahn 89] K.M.Kahn. Objects – a fresh look. In *Proceedings of the Third European Conference on Object-Oriented Programming (ECOOP 89)* S.Cook (ed), Cambridge University Press.

[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, pp.291-312.

[Land 65] P.J.Landin. A correspondence between Algol-60 and Church's Lambda notation. *Comm. ACM 8*, 2 pp.89-101.

[Lieb 87] H.Lieberman. Concurrent object-oriented programming in Act 1. In *Object-Oriented Concurrent Programing* A.Yonezawa and M.Tokoro (eds) MIT Press, pp.9-36.

[Me & Wi 91] J.Meseguer and T.Winkler. Parallel programming in Maude. In *Research Directions in High-Level Parallel Programming Languages*, J.P.Banâtre and D.Le Métayer (eds), SPringer *LNCS 574*, pp.253-293.

[Mi & Dr 88] M.S.Miller and K.E.Drexler. Markets and computations: agoric open systems. In *The Ecology of Computation*, B.A.Huberman (ed), Elsevier, pp.133-176.

[Mi & Le 94] N.H.Minsky and J.Leichter. Law-governed Linda as a co-ordination model. In *Object-Based Models and Languages for Concurrent Systems* P.Ciancarini, O.Nierstratz and A.Yonezawa (eds), Springer *LNCS 924*, pp.125-146.

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

[Shap 84] E.Shapiro. Systolic programming: a paradigm for parallel processing. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, Tokyo, pp.458-471.

[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.

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

[Thor 95] J.Thornley. Declarative Ada: parallel dataflow programming in a familiar context. *ACM Computer Science Conf.*, Nashville TN, pp.73-80.

[Wern 91] E.Werner. The design of multi-agent systems. In *Decentralised AI 3: Proc. Third European Workshop on Modelling Automous Agents in a Multi-Agent World*.

[WHHK 92] C.A.Waldspurger, T.Hogg, B.A.Huberman, J.O.Kephart and W.S.Stornetta. Spawn: a distributed computational economy. *IEEE Trans. on Soft. Eng 18*, 2 p.103-116.