A Concurrent Programming Model using Single-assignment, Single-writer, Multiple-reader Variables

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

Abstract This paper describes a concurrent programming language, Aldwych, and its underlying model. The basis of the language is that the concurrency is provided by shared single-assignment variables, each of which is constrained by the terms of the language to have exactly one writer though it may have several readers. Through derived forms, this simple model provides the basis of a flexible concurrent object-oriented language.

Introduction

Underneath various concurrent programming models lie differences in how processes share data. Unrestricted read/write access to shared store leads to the classic problems of concurrency and mechanisms such as semaphores for mutual exclusion, discussion of which still dominates concurrency as conventionally taught [Ma & Kr 99]. Shared rewriteable store remains an essential part of concurrency in Java, with the need for its synchronized methods though access to store is restricted by object-orientation. The popular Linda model [Ca & Ge 89] has no direct sharing between threads of variables, but has shared access to a database which any thread can write to, read from or delete from.While these models are popular because they are an evolutionary step from standard sequential von Neumann computing, the lack of restriction on access to shared data makes concurrency seem complicated, and leaves a large burden for the programmer to manage.

A different approach to concurrency uses models of computing which are inherently concurrent. The π-calculus [Mil 92] has received much discussion recently as one such model, and at least one programming language has been built from it [Pi & Tu 97], as functional programming languages have been built using the λ-calculus as an underlying model. The only data in the π-calculus are channels, and channels are used to send messages which are themselves channels between processes.

We sugges the π-calculus is not an ideal model for building a concurrent language because of the unrestricted nature of access to its channels. A channel in the π-calculus could be viewed as a restricted form of a Linda database, one which may hold only one tuple. At any time the channel is either full with one value in which case the only operation possible is for that value to be read and removed, or empty in which case the only operation possible is for a value to be put into it. The original π-calculus is synchronous, so the process putting a value in a channel must wait until it is taken out by another, but asynchronous versions in which there is no such wait have also been considered [Ho & To 91]. The problem is that any process that has access to a channel may both read and write to it. Read and write access may be spread arbitarily as a process with access to one channel may spread that access to any other process which shares any other channel with it. Thus in any non-trivial computation it could soon become difficult to trace which process wrote what to which channel.

We propose a model of concurrent computation which is considerably more restricted than this. As the history of sequential programming has shown us, with the development first of structured then of object oriented programming, restriction of access in a programming language strengthens it. We can more easily build higher level conceptual abstractions, both formally and informally, if we can be sure of lack of interference from unexpected side-effects.

In our model, shared locations may only be written to once and only one process has write access to a shared location at any one time. Other processes have read-only access to locations, and reading has no effect on the value of the location, so, as it is not consumed, writing to a location can be regarded as multicasting to all that location's readers. A process which needs the value of a location may have to wait until it learns of that location's value, but there are none of the problems associated with concurrent systems where shared locations may change their value. As every location has exactly one writer, in the absence of deadlocks and non-terminating computations, locations will always get written to but there may be an arbitary delay before a reader learns of the value written to a location. This allows for implementation in a distributed system where it is the responsibility of a location to inform processes suspended waiting its value of that value.

Logic Variables and Futures

The single assignment variable is often referred to as the "logic variable" [Hari 99] as the first extensive use of the concept occurred in logic programming. Another form of single-assignment variable is the "future" of parallel functional programming [Hals 85] and concurrent object-oriented programming [Lieb 87] – a placeholder for a value being computed concurrently which may be embdedded in structures or passed as an argument before its value is known. The conventional model of logic programming, however, makes no distinction between readers and writers of a variable, whereas a future has exactly one writer, the computation calculating its value. Our locations as described above then are more like futures than logic variables.

The key advantage logic variables offer over futures is "back communication" [Greg 87]. This refers to the writer of a variable writing to it a structure containing variables which are intended to be written to by readers of the original variable. This allows for the two-way communication between processes which is essential for distributed computing. A future allows only for one-way communication since a computation may set a future to a structure containing a future but must also set up a computation to give a value to that embedded future.

Back communication seems incompatible with insisting that variables have one writer but letting them have multiple readers. A structure containing a variable intended for back-communication will be accessed by all readers of the original variable set to that structure without distinction. To overcome this, Saraswat et al proposed a language, Janus, in which logic variables had exactly one reader as well as exactly one writer [Sara 90]. We propose a model which is less restrictive than this, so multicasting is not lost.

We also avoid the introduction of additional constructs such as the cells and procedure references of Oz [Hari 99] which are not describable using logic variables and the logic programming model. So we develop a language which relies heavily on derived forms, but has a direct translation to concurrent logic programming [Shap 89]. There are no direct higher-order procedures, unlike Oz. A program consists of a fixed set of process descriptors and an initial process. We shall assume the process descriptors are available globally, though in a highly distributed implementation of the language this assumption may need to be reconsidered. At present the language is implemented through a translator to a concurrent logic language, but it is not our intention that the language be seen as a "front-end" to concurrent logic programming, so it need not necessarily be implemented in that way.

Communication of Simple Values

A process in our model consists of a name and an ordered collection of location references (which we can call variables). There is a fixed set of process descriptors, one for each possible name of process. A process descriptor gives formal parameters for a process of its name, indicating for each parameter whether a process of that name has read or write access to it. It also gives a set of rewrite rules. Each rewrite rule consists of a left-hand side (lhs) and a right-hand side (rhs). The lhs of a rule consists of read commands, or more properly ask commands, since they ask whether a variable has a particular value. The rhs consists of write commands and new process creations. A computation consists of a set of processes. When the variables in a process are sufficiently bound such that all the asks on the lhs of a rule in the process descriptor are answered positively the process is transformed to the rhs of the rule. In both cases formal parameters are substituted by actual parameters, with any parameter that occurs only in the rule being substituted by a new variable inaccessible anywhere else in the computation. For simplicity we will for now ignore the case where there is no rule to cover a situation, that is when at least one ask is answered negatively for every rule.

To keep to the property that any variable has exactly one writer, each rule must give a writer for each argument a process has write access to. The rule may write a value to the variable itself or create a new process which writes to that variable. A variable which occurs only on the rhs must occur in exactly one write position.

Here is the simplest example of a process descriptor:

```
#p(x,ŷ) {x=a || y=b}
```

This indicates that any `p` process has two arguments, is a reader to its first and a writer to its second. After its first becomes the constant `"a"` it writes the constant `"b"` to its second.

Here is an example with two rules:

```
#p(x,ŷ) {x=a || y=b; x=c || y=d}
```

Again, a `p` process has two arguments, is a reader to its first and a writer to its second. If its first becomes `"a"` it writes `"b"` to its second, if its first becomes `"c"` it writes `"d"` to its second.

A rule may have more than one ask on its lhs, in which case all asks must be answered positively for the rule to be applied. So the following

```
#q(x,y,ẑ) {x=a, y=b || z=c}
```

indicates that a `q` process has three arguments, and is a reader to its first two, a writer to its third. After its first becomes `"a"` and its second becomes `"b"`, `"c"` is written to its third.

The model is indeterministic because more than one rule may apply:

```
#q(x,y,ẑ) {x=a || z=c; y=b || z=d}
```

Here, a `q` process has three arguments, is a reader to its first two and a writer to its second. If the first is `"a"` then `"c"` is written to the third, if the first is `"b"` then `"d"` is written to the third. But either rule could be applied if the first is `"a"` and also the second is `"b"`.

While `z=y` writes the constant `"y"` to the variable `z`, `z←y` assigns the variable `y` to the variable `z`. The effect is that remaining occurences of `z`, which must be read occurrences, become further read occurrences of `y`. So

```
#r(w,x,y,ẑ) {w=a || z←x; w=b || z←y}
```

means an `r` process has four arguments, with read access to the first three and write access to the fourth. If the first becomes equal to `"a"`, the fourth argument becomes equal to the second argument, either the value already written to the second argument, or if no value has yet been written to it, the variable in the fourth argument will become written to whatever is written to the variable of the second argument later. Similarly with the third argument if the first becomes `"b"`. Note that since variables never have their values rewritten, we do not need to be concerned over whether we are writing references to values or values themselves into variables.

For an example showing the creation of a new process:

```
#r(w,x,y,ẑ) {w=a || p(x,ẑ); w=b || p(y,ẑ)}
```

Here when the first argument to a four-argument `r` process becomes `"a"`, a new `p` process is set up whose first (read) argument is the second argument of the `r` process and whose second (write) argument is the fourth argument of the `r` process, when the first argument of the `r` process is `"b"`, a new `p` process is set up whose first argument is the third argument of the `r` process. Note that a rule for an `r` process here has either to write to `z` directly using = or ←, or has to create exactly one new process with `z` in the output position. It is not necessary to use all read arguments in a rule however, as the above example shows. One way of thinking of the rules above is that an `r` process becomes a `p` process on rewriting using either of the rules.

For an example with a local variable

```
#q(x,y,ẑ) {x=a || p(y,ŵ), p(w,ẑ)}
```

sets up two `p` processes when the first argument to a `q` process becomes `"a"`, with `w` the link between them, written by one and read by the other. Any such variable local to the rhs of a rule must have one writer but may have any number of readers. If there is an existing variable of the same name, local variables like `w` should be considered as given new unique names – there is no "variable capture".

A variable may be used as a read argument even if its value is known through an ask, for example in

```
#q(x,y,ẑ) {x=a || p(y,ŵ), q(w,x,ẑ)}
```

the second argument of the recursive call to `q` must be `"a"` on its creation.

For an example with multicasting

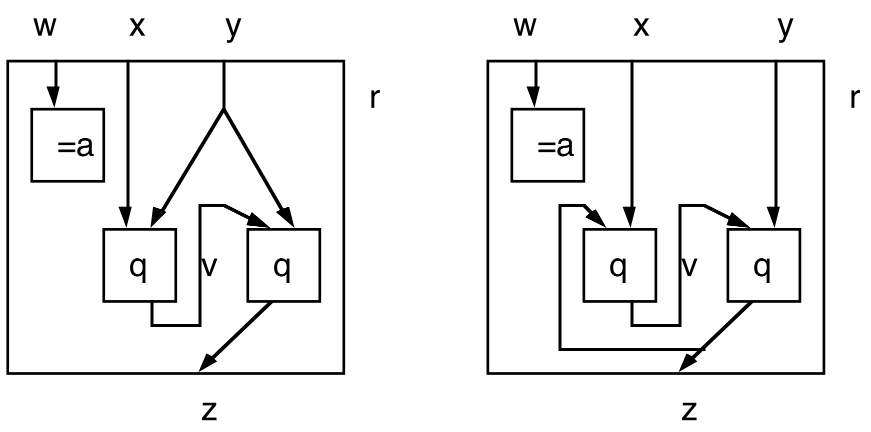
```
#r(w,x,y,ẑ) {w=a || q(x,y,v̂), q(v,y,ẑ)}
```

Here the third argument of the `r` process is sent to both `q` processes created when the first argument becomes `"a"`. Unlike Janus, this can happen before this third argument has been given a value. Multicasting can also occur with an argument to which a process has write access occurring in a read position in a new process:

```
#r(w,x,y,ẑ) {w=a || q(z,x,v̂), q(v,y,ẑ)}
```

Here the `z` written by the second `q` process is multicast by being sent out of the `r` process as well as into the first `q` process. As the two `q` processes run concurrently, the circular dataflow is permitted, though it can lead to deadlock. The use of a read access argument in a write position is not permitted however (until we consider single bar rules later).

The situation in these two multicasting examples is perhaps best appreciated when show as dataflow diagrams of the sort we used in our previous work on logic program synthesis [Hunt 86]:



Communication of Compound Values

Locations in our model may store not just constants but tuples containing references to further locations. Here is a description of a process with a read and write argument, which reacts when the read argument becomes an `f` tuple with two arguments by writing a `g` tuple with the same two arguments reversed to its write argument:

```
#p(x,ŷ) {x=f(u,v) || y=g(v,u)}
```

Tuple arguments on the lhs of a rule are treated as additional read arguments on the rhs. If an argument to a tuple on the rhs is local to that rhs, it must also occur in exactly one write position in a new process creation in order to ensure that every variable has a writer:

```
#q(x,y,ẑ) {x=f(u,v) || p(u,ŵ), z=h(v,y,w)}
```

The arguments to tuples may themselves be tuples, and asks on the lhs may combine asks on an argument with asks on a sub-argument, for example:

```
#q(x,y,ẑ) {x=f(u,v), v=g(w) || p(u,t̂), z=h(w,y,t)}
```

means a `q` process reacts only when its first argument becomes `f(j,g(k))` for any variables `j` and `k`.

Tuples as described here do not allow for back-communication. When a process reads a tuple it has only read access to its arguments. When a process writes a tuple it has to write its arguments or pass write access to processes it has created. For a tuple to be used for back-communication, a process which reads its value must be given write access to one or more of its arguments. We could imagine `m=f(x,ŷ)` writing to `m` a tuple whose second argument is intended for back communication, but suppose `m` is passed to a process that multicasts it, which reader of the tuple will write to `y`?

Back Communication

Back communication is achieved by having a separate set of variables which may hold tuples with back-communication arguments. These tuples are referred to as *queries* as they may be considered queries sent to other processes to send an answer back. So the variables holding them are referred to as *query variables*. The header of a process descriptor will indicate which of its arguments are query variables, and which are standard variables. Query variable arguments have read or write access like standard arguments, but it is a compiler error to multicast them. It is a compiler error to use a query variable in a standard argument position in a new process creation, and a compiler error to use a query variable as one of the arguments of a tuple. A query, however, may have query variables amongst its arguments.

A process rule which asks for the value of a query argument it has read access to has write access to all of its reply arguments. If a rule does not have an ask on a query variable it has read access to, it must pass that variable to exactly one new process, which is then passed the write access to its reply arguments, or alternatively the query can be passed on as an argument to a further query. In

```
#q(m,x,y)
{
 x=a, m⇒f(u,v,ẑ) || r(u,v,y,ẑ);
 x=b || p(m,y)
}
```

the first argument to a `q` process is a query, which is indicated by writing it in italics. When the second argument to the `q` process is `"a"`, the first rule is used which creates an `r` process to answer the query in the first argument. When the second argument to the `q` process is `"b"`, the query is passed on to a `p` process. Passing on a query like this is a form of delegation [Lieb 87]. The ⇒ character is used in the place of = in the first rule to indicate that the query `m` is being broken down and answered in this rule. = is used with query variables to view them without answering them. This is done below:

```
#p(m,x)
{
 m⇒f(w,y,ẑ) || r(x,y,w,ẑ);
 m=g(w,y,ẑ) || r(z,y,w,v̂), q(m,v,x);
 m⇒h(w,y,ẑ) || n=k(w,ẑ), q(n,y,x)
}
```

Here, a `p` process checks whether its first argument is an `f` query, a `g` query or an `h` query. In the first case it answers it directly using an `r` process to generate the answer. In the second case, it delegates it to a `q` process to answer, but one of the arguments to the `q` process is generated from an `r` process which itself requires the reply to the query. Note that when a process looks at the value of a query but does not answer it, that is when its value is asked for using =, it has read access only to all the arguments of that query including those that are indicated as reply arguments. So in the second rule above there is read access to `z`, but write access to `z` is passed on to the `q` process. A query which is read but not answered must be passed on to exactly one process to keep the rule that there must be exactly one writer for every variable. The third rule above shows a case where the query is answered by formulating and asking a further query.

Note that the `q` and `p` processes above have no arguments with write access. This means their only output is through replies to queries they receive as their first argument. The use of back-communication means we can no longer describe the behaviour of a process rule through a simple dataflow diagram as used previously.

Further Aspects of the Model

The model described above needs some minor additions in order for it to be the basis of a general programming language. Arithmetic operators are allowed on the rhs of rules so, for example, `x←y+z` waits till `y` and `z` have had numerical values written to them, then adds those values and writes the results to `x`. The right-hand side of the ← may be any variable with write access, the left-hand side any arithmetic expression with the usual operator precedence rules applying. The lhs of a rule may include comparison operators on variables (either process arguments or additional variables from tuples or queries), such as `x>y`. Any comparison operator must evaluate positively as well as all ask operations for a rule to become applicable. A comparison operator can only evaluate when both its arguments have been written to. The operation x? on the lhs where x is any variable means the rule does not become applicable until x has been written to. This can be used for synchronisation. A **:** following a rule is read as “otherwise”, meaning that rules following it only become applicable if there has been sufficient writing to variables to make all rules before it inapplicable. Variable and process names may be any string of alphanumeric characters starting with an alphabetical one, and strings of digits represent numerical constants. The following, for example, is a process descriptor to evaluate factorials:

```
#fact(n,f̂) {n>0 || f←1; :|| fact(m,r̂), m←n-1, f←r*n}
```

We have constructed a programming language called Aldwych which adds derived form to this underlying model, much as Pict builds from the π-calculus by adding derived forms [Pi & Tu 97].

For use with ascii, p(x,ŷ) can be written p(x)->y, and q(x,ŷ,ẑ) can be written q(x)->(y,z) and so on. x←exp can be written x<-exp for exp any arithmetic expression. We shall see later how queries, which above use italics and the symbol ⇒, are dealt with.

To reduce the name space and give a more functional appearance to code, p(x) as a new process argument or in an arithmetic expression can be written in the place of v and p(x,v̂) written elsewhere in the rule, where v is a variable which does not otherwise occur in the rule. Similarly, any arithmetic expression exp can be written directly as an argument to a new process in the place of v being the argument and v<-exp occurring separately. =f(x,y) as an argument passes the tuple f(x,y) as the argument, since f(x,y) on its own would mean z where there is a process f(x,y,ẑ). On the lhs, x=f(u,g(v)) is a shorthand for x=f(u,w),w=g(v). The form x=f(u,w=g(v)) is also allowed to give a layered pattern where a variable representing g(v) is required on the rhs. The factorial example can be written in Aldwych (showing the reduction of name space and also the form for when only ascii characters are available):

```
#fact(n)->f
{
 n>0 || f<-n*fact(n-1);
 :
 || f<-1
}
```

The otherwise operator is best written in a line of its own like this so that it stands out. More concise ways of writing factorial are given later in this paper.

A notation for expressing priorities between processes may be needed in some cases (we argue this in respect of concurrent logic programming in [Hunt 91]). For mapping the abstract concurrency onto real parallel architecture, a mapping notation such as that employed by PCN [Fost 92] may be used. Notations for priority and process mapping should be kept orthogonal to the abstract concurrency and are not discussed further here.

Process State

Processes as described so far are short-lived. As soon as a rule is applied, they disappear, though possibly creating new processes when they do. Conceptually, however, it helps to think of a process created when a rule is applied as continuing the original process, particularly if it has the same name and hence same process description. This is how the Actors model [Agha 86] works: an actor creates the actor to receive the next message in a stream of messages which are conceptually sent to a single actor. It is also the basis for viewing concurrent logic programming in an object-oriented way [Sh & Ta 83], [Hunt 95]: the arguments in a clause head are considered to represent the state of an object, with the arguments in a recursive call in the clause body representing the state as updated by the actions represented by the clause.

To encourage thinking in these conceptual terms, Aldwych has “single bar” rules in which there is an implicit recursive process creation. When the lhs and rhs of a rule are separated by `|` rather than `||`, a process with the same process descriptor as the process the rule in in is created when the rule is applied. The arguments to this recursive call are by default the arguments to the original process. However, if `x` is a read access argument to the process, `x←exp` in a single-bar rule is equivalent to `x′←exp` with `x′` replacing `x` in the recursive call (also `x=val`, where `val` is a constant or a tuple, is equivalent to `x′=val` with the replacement). If `y` is a write access argument to the process, `y←exp` in a single-bar rule is equivalent to replacing `y` in the recursive call and all other occurences of `y` on the rhs (including within `exp`) by `y′`.

Thus

```
#r(w,x,y)->z
{x=a | v<-q(w,y), y<-q(z,v), x=b, z<-p(z)}
```

is equivalent to

```
#r(w,x,y,ẑ)
{x=a || q(w,y,v̂), q(z′,v,ŷ′), x′=b, p(z′,ẑ), r(w,x′,y′,ẑ′)}
```

Here `y<-p(x)` is equivalent to `p(x,ŷ)` since `p(x)` on its own can be an expression equivalent to `v` with a separate `p(x,v̂)` with `v` an otherwise unused variable, so `y<-p(x)` is equivalent to `y<-v,p(x,v̂)`, which is equivalent to `p(x,ŷ)`.

Using this, the factorial process above can be written as:

```
#fact(n)->f {n>0 | f<-n*f, n<-n-1; :|| f<-1}
```

If you are not used to the Aldwych style, the behaviour of a process with this description may not be immediately apparent, but a tail recursive version of factorial:

```
#fact(n,acc)->f {n>0 | acc<-acc*n, n<-n-1; :|| f<-acc}
```

can easily be seen in terms of the while loop:

```
while(n>0)
       {
        acc<-acc*n;
        n<-n-1
       }
return acc
```

and in general a pair of rules of the form

```
{ cond | assignments; :|| return<-value}
```

can be seen as the while loop

```
while(cond) do assignments; return value
```

operating on updateable store.

This is the basis on which an imperative style of programming can be achieved in Aldwych. The idea is extended in the full Aldwych language with some further derived forms not described here.

Stream Communication

In the object-oriented view of concurrent logic programming, recursive calls representing the continuation of a state are coupled with lists representing continuous communication between long-lived processes. If we represent lists by the tuple `list(head,tail)` then the following is a long-lived version of our first process descriptor which took in `"a"` and returned `"b"`:

```
#p(x,ŷ) {x=list(h,t), h=a | y=list(=b,y), x<-t}
```

which translates to:

```
#p(x,ŷ) {x=list(h,t), h=a || y=list(=b,y′), p(t,ŷ′)}
```

This describes a process which has two arguments, an input stream and an output stream. Each time it receives `"a"` on the input stream, it outputs `"b"` on the output stream.

For a more convenient list syntax, Aldwych uses `:` as an infix cons operator, resulting in the following representing the `"a"` to `"b"` stream converter:

```
#p(x,y) {x=m:t, m=a | y=(=b):y, x<-t}
```

However, in most cases direct use of the cons operator is not required due to a derived form introduced to handle streams. On the lhs, `x.`*val* for any constant or tuple *val* is equivalent to `x=h:t,h=`*val* with all occurrences of `x` on the rhs replaced by `t` (where `t` is a new variable), except the occurrence in `x<-`*exp* for assigning a new value of `x` in the implicit recursive call. This means that `x.`*val* on the lhs can be considered as checking the value at the head of stream `x` and consuming it. On the rhs, `y.`*val* needs to represent sending *val* on stream `y`, so it is equivalent to `y<-`*val*`:y`, which assigns *val*`:y` to the `y` position of the whole process, while leaving `y` otherwise to represent the `y` output of the implicit recursive call. This means the `"a"` to `"b"` stream converter can be described simply by

```
#p(x,ŷ) {x.a | y.b}
```

The only problem here is that there is no rule to deal with stream termination. The full converter is written:

```
#p(x,ŷ) {x.a | y.b; x$ || y$}
```

On the lhs `x$` means “check stream `x` is end-of-stream”, while on the rhs `x$` means “set stream `x` to end-of-stream”.

If `x` is a process argument with read access, `x.`*val* is permitted on the rhs of a single-bar rule, and has the effect that the `x` position in the implicit recursive call is set to *val*`:x`. The effect is to treat `x` as a stack, with `x.`*val* on the rhs the push operation, and `x.`*val* on the lhs the pop operation. The equivalent of the top operation which looks at the top of the stack or the front of the stream without consuming it is `x/.`*val* on the lhs. Like `x.`*val*, this is equivalent to `x=h:t,h=`*val* on the lhs, but instead of `x` being set to `t` on the lhs, it remains *val*`:t` there.

Since `x.a` means send/receive the constant `"a"`, we need separate forms to receive any value from stream `x`, and send any value to stream `x`. On the rhs, `x^`*exp* evaluates the expression *exp* and sends it on stream `x`, that is it writes `e:x′` to `x` if the process has write access to `x`, or sets the `x` position in the implicit recursive call to `e:x` if `x` is a read access argument to the process, where `e` is the result of evaluating *exp*. On the lhs of a rule, `x?v` evaluates positively when `x` is bound to `h:t`, and makes `v` a reader of `h` and `x` a reader of `t` on the rhs of the rule. Again, a lookahead form is available, so `x/?v` evaluates positively when `x` is bound to `h:t`, makes `v` a reader of `t`, but keeps `x` referring to `h:t` on the rhs.

Making use of these, the following is a process descriptor for indeterminate stream merger:

```
#imerge(s1,s2)->m
{
 s1?v | m^v;
 s2?v | m^v;
 s1$ || m<-s2;
 s2$ || m<-s1
}
```

while merger of ordered streams with the elimination of duplicates is described by:

```
#omerge(s1,s2)->m
{
 s1?u, s2/?v, u<v | m^u;
 s1/?u, s2?v, v<u | m^v;
 s1?u, s2?v, u==v | m^u;
 s1$ || m<-s2;
 s2$ || m<-s1
}
```

It is possible both to send and receive multiple values on a stream in a rule. On the rhs, x.val1.val2 is equivalent to writing val1:val2:x′ to x where val1 and val2 are any values, and : is right associative. On the lhs, x.val1.val2 is equivalent to x=h1:x′,h1=val1,x′=h2:x″,h2=val2 with x replaced by x″ on the rhs. Combinations of the stream operators are possible, so x.val^exp on the rhs writes val:e:x′ to x, for any value val and expression exp where e is the write argument from the evaluation of exp. Combinations may include the lookahead operator on the lhs, so x?u/.val on the lhs becomes applicable when x has been written to h:t with t further written to val′:x′, then x is replaced by val′:x′ on the rhs.

Streams of Queries and Objects

Because the arguments to a query may themselves be queries, a query variable may be set to a stream of queries. However, such a stream must have only one reader and one writer, whereas a standard stream may have many readers. Apart from this streams of queries may be handled like streams as described above. Here is a simple example:

```
#cell(s,n)
{
 s.set(m) | n<-m;
 s.get(m̂) | m<-n;
 s$ ||
}
```

which is equivalent to:

```
#cell(s,n)
{
 s=list(h,s1), h⇒set(m) || cell(s1,m);
 s=list(h,s1), h⇒get(m̂) || m←n, cell(s1,n);
 s=nil ||
}
```

To make this clearer, an infix cons symbol has not been used for lists here. So the query variable *s* which a cell process has read access to is bound by the process which has write access to *s* to a list consisting of `set(m)` and `get(m̂)` queries. The cell process has read access to the `m` in `set(m)`, but write access to the `m` in `get(m̂)`. So a process set up by `cell(s,n)` can be considered a form of mutable variable, with `n` initialising it, a `set(m)` message sent on *s* changing the value it stores to `m`, and a `get(m̂)` message on *s* used by the process which has write access to *s* to retrieve the current value of the mutable variable. This might be considered to have limited use, since *s* can have only one writer, and a mutable variable for a single process can be represented by a simple argument in the process call changed to update it in the recursive call. However, consider the following:

```
p(…,ŝ′,…), q(…,ŝ″,…), merge(s′,s″,ŝ), cell(s,n)
```

where a `merge` process is like the indeterministic merge above, except that it merges streams of queries rather than streams of values. The result is that the query stream *s* of the cell process is a merger of two separate streams of queries *s′* and *s″* written by the `p` and `q` processes respectively. So `p` and `q` may be considered as sharing access to a single mutable variable.

The idea of a shared mutable variable here is just a simple example of what can be seen more generally as an object. An object has its own state which may be altered in response to receiving a message, but it has complete control over its state (encapsulation). This is what we have here with object state represented by process arguments which a process may change in response to reading a query from the query stream it has read access to. There can be as many references as needed to an object in other processes, each represented by the process having write access to a stream of queries, with the queries being merged together to form the single stream which is read by the object process. The rule that a query has a single reader and a single writer is not broken since the merge process is the single reader of the query, which delegates that query to the object process. If there are more than two references to an object, a tree of binary merges of query streams could be built up which enables us to keep a description of the language in terms of the underlying model. A more efficient alternative would be to provide multiway merges as a primitive in the implementation of the language.

However, having to deal explicitly with streams of queries is too low level. Additionally, it is counter-intuitive to represent an object which should be considered one of the inputs to a process by a stream which is an output of the process. Therefore we introduce "object variables" which are translated automatically to query variables. An object variable is a query variable but with its polarity reversed, and refers only to streams of queries. An object variable with write access is represented by a query variable with read access. An object variable with read access is represented by a query variable with write access. Multiple occurrences of an object variable with read access are translated into separate query variables with write access whose sole reader is a merge process which writes the query stream to which the write access of the variable was translated.

So a `p` and `q` process sharing access to a `cell` object can be written as:

```
p(…,S,…), q(…,S,…), cell(n)->S
```

which is translated automatically to the form above. Note that object variables are signified by having an initial capital letter. The call `cell(n)->S` may be considered as producing the object reference `S`, referring to a new cell initialised with `n`.

Object Variables and Object Descriptions

Object variables have exactly the same rules regarding readers and writers, including with single-bar rules, as standard variables. All the necessary conversion to streams and mergers is done automatically. In fact query variables are almost dispensed with in written Aldwych, featuring only in the underlying programming model into which it is translated, and as local variables. Process descriptors may not have query arguments in written Aldwych, but they do remain as local variables within rules. Here is an example:

```
#obj(x,y,P)->S
{
 S.mess1(z)->r | q(x,y,z)->r;
 S.mess2(z)->r | P.mess3(x,y,z)->r;
 S.mess4(u,v)  | x<-r(u,v,P);
 :
 S?m | P^m;
}
```

As can be seen, queries to an object are read using the same notation as used for streams of values (`.` and `?`), except the object reference must be one which appears to the right of the `->` in the process descriptor header. A query is more conventionally termed a message in object-oriented terms.

In the description above, an `obj` object takes three arguments, two values `x` and `y`, and an object `P`. If it receives a `mess1` message with one input argument `z` and one reply argument `r`, it computes the value of `r` by setting up a `q` process with its `x` and `y` arguments and the `z` argument from the message as input. If it receives a `mess2` message with one input argument and one reply argument, it returns the reply from a `mess3` message sent to the `P` object (another way of thinking of this is that the write access to `r` is passed to the object represented by `P`). A `mess4` message with two input arguments but no reply causes the value of `x` in the state of the `obj` object to be changed. The final rule applies only if none of the previous rules apply (due to the `:`), that is a message is received which is not of the type covered in the previous rules. In this case it is delegated to `P` to handle. Here `m` is a query variable, and is detected as such from its context on the lhs. The only thing that is permitted to be done to a query variable is to send it in this way to another object. Note that `P^m` sends the query in variable `m` to the object referenced by `P`, and should be distinguished from `P.m` which would send the constant `"m"` to `P`.

The following is the underlying representation of this in the Aldwych model:

```
#obj(x,y,p̂,s)
{
 s=m:s′, m⇒mess1(z,r̂) || q(x,y,z,r̂), obj(x,y,p̂,s′);
 s=m:s′, m⇒mess2(z,r̂) || p←n:p′, n=mess3(x,y,z,r̂),
                          obj(x,y,p̂′,s′);
 s=m:s′, m⇒mess4(u,v) || r(u,v,p̂′,x̂′), obj(x′,y,p̂″,s′),
                          merge(p′,p″,p̂);
 :
 s=m:s′ || p←m:p′, obj(x,y,p̂′,s′);
 s$ || p$;
}
```

The last rule is added automatically and works as an object destructor. When the stream of queries representing an object becomes end-of-stream, that indicates there are no further references to the object. In any rule end-of-stream is written to any query variables it has write access to which otherwise are unused, so that the reader of the stream will read end-of-stream if there are no other references to it. This works to give garbage collection of objects, although it cannot cope with circular object references (an object which references a second object which references the first object, either directly or indirectly through further objects).

Object variables can be passed in messages to objects, both forwarded with them, and returned in reply. The ability to pass object variables in messages and use object variables received in messages as arguments to new processes and further messages enables Aldwych to deliver dynamic concurrency, in which communication topologies evolve during evaluation.

An object which is the writer to an object variable in messages which it receives can be considered an “object-factory” or class. For example, the following:

```
#objclass()->S
{
 S.new(x,y,P)->R | obj(x,y,P)->R
}
```

in conjunction with the descriptor for `obj` objects above is the descriptor of an object which represents a class for `obj` objects. If `C` is an object created by `objclass()->C`, then `C.new(a,b,P)->Q` is equivalent to a direct call `obj(a,b,P)->Q`. Alternatively, consider:

```
#sobjclass(x,y)->S
{
 S.new(P)->R | obj(x,y,P)->R
}
```

Now, a call `sobjclass(a,b)->C` creates a class referenced by `C` for producing `obj` objects specialised with their first two arguments set to `a` and `b`, and only the third argument specified each time a new one is created by `C.new(P)->Q`. This is the basis on which higher-order programming in Aldwych is achieved. The basic idea is extended in the full language by some further derived forms not described here.

Beyond Object-Oriented Programming

In conventional object-oriented programming, including conventional concurrent object-oriented programming, there is a just a single rule for each type of message an object can receive ([Fr & Ag 94] describes an exception to this). While an object descriptor in Aldwych can be limited only to have rules of this form, this is at the discretion of the programmer and more complex rules are also possible.

A simple use of the flexible nature of message patterns in Aldwych is to have an object represented by more than one streams of queries, for example:

```
#mobj(x,y)->(S,T)
{
 S.p(z)->r | r<-f1(x,y,z);
 T.p(z)->r | r<-f2(x,y,z);
 S.q(z)->r | r<-g(z,y,z);
 T.r(z)->r | r<-h(z,y,z);
 {S,T}.s(z)->r | r<-k(z,y,z)
}
```

Now, if a new object is created by `mobj(a,b)->(P,Q)`, `P` and `Q` represent two different views of that object. Whichever view is appropriate may be passed on to other objects. `P` represents a view which can respond to `p`, `q` and `s` messages, while `Q` represents a view which can respond to `p`, `r` and `s` messages. The `Q` view responds to `p` messages in a different way than the `P` view, using an `f2` rather than an `f1` process, but both respond to `s` messages the same way. The last rule introduces some new syntax, it is just a shorthand for:

```
 S.s(z)->r | r<-k(z,y,z);
 T.s(z)->r | r<-k(z,y,z)
```

This is just one example of several pieces of syntax in Aldwych but not described in this paper which provide shorthand ways of writing commonly occurring patterns of code. This one enables an arbitary number of types of access right to be given to an object, which clearly gives a lot more programmer flexibility than the fixed **public**, **private** and **protected**, or similar, of conventional object-oriented languages.

The stacking effect, seen previously with `s.m` on the rhs where `s` is a read-access argument, can be used with object arguments. `S.mess(args)->reply` where `S` is an object variable with write access has the effect of stacking the message `mess(args)->reply` on the `S` stream, equivalent to conventional object-oriented "send a message to self".

For examples of rules which respond only to combinations of messages (only the lhs of the rules is given):

```
S.f(u)->v.g(w)->z | …
```

responds only when `S` receives an `f` message followed by a `g` message and must give a reply to both by writing to `v` and `z` on the rhs.

```
S.f(u)->v, T.g(w)->z ~ | …
```

responds when the `S` stream receives an `f` message and the `T` stream a `g` message, again both must be replied to on the rhs.

```
S.f(u)->v/.g(w;z) | …
```

responds only when an `f` message is followed by a `g` message, but involves a look-ahead at the message stream to the `g` message. The `g` message is not answered in this rule. The syntax `g(w;z)` rather than `g(w)->z` is used to make this clearer. Although `z` is the return value of the `g` message, it is not written to on the lhs, because the `g` message is not consumed in this rule but left for a further rule application.

Conclusions

The language Aldwych has an operational model which is simpler than that proposed for other abstract concurrent languages. The language allows for interaction between processes, and the dynamic configuration of communication links.

In order to give a flavour of the language without overwhelming the reader with detail, it has been described in a fairly informal style. It would require a longer paper to describe formally each of the features mentioned here, and the precise way it interacts with other features, although the model which underlies the language is strong enough to allow such a formal description to be built. There are several further features of the language not covered here, but which take the same approach as that used here, that is using derived forms which can be translated to the simpler operational model. The aim of the derived forms introduced is to overcome verbosity, in particular the verbosity which is found in the direct use of concurrent logic languages for object-oriented programming [Sh & Ta 83] which is the ancestor of this work.

The language has been developed experimentally, so an implementation exists, compiling into the concurrent logic language KLIC [Roku 96]. The development of the language has proceeded by using it, observing programming patterns, and adding new derived forms where a particular pattern of programming is common, and its intended effect is easier to see when it is written using the derived form.

The complete set of derived forms has not been finalised yet, and there is an obvious tradeoff of the convenience of having a higher-level construct against the complication which it brings into the language. We would prefer to promote wider experimentation and discussion of the language before settling on a fixed set of constructs. This paper has described the basic model, which has remained stable for some years, and some suggestions as to how it could be extended.

References

[Agha 86] G.Agha. Concurrent object-oriented programming. *Comm. ACM* 33, 9 pp.125-141.

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

[Fost 92] I.Foster, R.Olson and S.Tuecke. Productive parallel programming: the PCN approach. *Scientific Programming* 1, 1 pp.51-66.

[Fr & Ag 94] S.Frølund and G.Agha. Abstracting interactions based on message sets. In *Object-Based Models and Languages for Concurrent Systems* P.Ciancarini, O.Nierstratz and A.Yonezawa (eds), *LNCS 924*, pp.107-124.

[Greg 87] S.Gregory. *Parallel Logic Programming in PARLOG.* Adison-Wesley.

[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*.

[Ho & To 91] K.Honda and M.Tokoro. An object calculus for asyncronous communication. In *Proceedings of ECOOP'91*, P.America (ed.), *LNCS 512*, pp.133-147.

[Hunt 86]. M.Huntbach. Program synthesis by inductive inference. *Proc. 7th European Conf. on Artificial Intelligence*.

[Hunt 91] M.Huntbach. Parallel branch-and-bound search in Parlog. *Int. J. of Parallel Programming* 20, 4 pp.299-314.

[Hunt 95] M.Huntbach. An introduction to RGDC as a concurrent object-oriented language. *J. Object Oriented Programming*. Sep. 1995.

[Lieb 87] H.Lieberman. Concurrent object-oriented programming in Act1. In *Object-Oriented Concurrent Programming* A.Yonezawa and M.Tokoro (eds), MIT Press, pp.9-36.

[Ma & Kr 99] J.Magee and J.Kramer. *Concurrency*. Wiley.

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

[Pi & Tu 97] B.C.Pierce and D.N.Turner. *Pict: a Programming Language Based on the Pi-Calculus*. Indiana University CSCI tech. Rep. 476.

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

[Sara 90] V.A.Saraswat, K.Kahn and J.Levy. Janus: a step towards distributed constraint programming. In *Proc. North American Conf. on Logic Programming*,. S.Debray and M.Hermenegildo (eds), MIT Press, pp.431-446.

[Shap 89] E.Shapiro. The family of concurrent logic programming languages. *ACM Comp. Surveys*, 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.