Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: Cambridge (0223) 323010
INTERNATIONAL: +44 223 323010
FAX: +44 223 359779
E-MAIL: apm@ansa.co.uk

ANSA Phase III

Using Path Expressions as Concurrency Guards

Owen Rees

Abstract

This document describes part of a concurrency control model to be added to the ANSA computational model and DPL. The concurrency control model can be used in non-atomic computations or for the ordering phase of atomic computations as described in AR.004.

The concurrency control model is based upon path expressions. This document includes a description of the requirements for concurrency control, a survey of the various forms of path expression, a proposal for the elements of a concurrency control model, a syntax for that model and an implementation strategy with examples taken from a prototype implementation.

Various options have been identified, pursuing some of these requires considerable additional work.

APM.1010.01 **Approved** 25 January 1994

Technical Report

Distribution:
Supersedes:
Superseded by:

Using Path Expressions as Concurrency Guards

Technical Report

Using Path Expressions as Concurrency Guards

Owen Rees

APM.1010.01

25 January 1994

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Requirements for Concurrency Control

The concurrency control model described in this document is intended for use with the ANSA computational model [APM.1001.01 93] and is designed to fit into that context.

Interaction in ANSA is described in terms of invocation requests and responses. The concurrency control manager determines whether or not an activity requesting an invocation may proceed immediately or if some other action must be taken. The concurrency generation mechanisms of the ANSA computational model define the relative progress of activities in terms of activities that are waiting. Whatever other options are available, delaying an activity until the concurrency control manager is prepared to permit it to proceed must be possible.

Although the concurrency control mechanisms may be used by themselves, they are being developed as part of the atomic operation infrastructure to support the ordering phase of the concurrency control manager [APM.1004.01 93].

There are two major requirements for concurrency control:

- protecting the service abstraction
- the provision of synchronization services

In addition to these requirements, concurrency control has an effect on resource usage and may be called on to help to limit resource consumption.

1.1 Protecting the abstraction

1.1.1 Abstract and concrete states

A service, as seen by its users, may be in one of a number of states. These are abstract states that correspond to the potential future behaviours of the service. The implementation will also have a number of states, the concrete states of the object that provides the service[1].

There may be many concrete states corresponding to each abstract state, and there may be concrete states that do not correspond to any valid abstract state.

The transitions from one abstract state to another are caused by invocations of the operations in the interfaces of the object. These invocations include the evaluations of forms which operate on the concrete state of the object. It is necessary to ensure that whenever there are no evaluations in progress, the concrete state of the object corresponds to some abstract state. It is also

1. The terms 'concrete' and 'abstract' are relative to one another. The implementation may be in terms of the states of services which are abstract with respect to the implementation of those services.

important to ensure that the concrete state corresponds to an abstract state that is consistent with the initial state and the invocation history.

The model described here does not distinguish between activities. Such distinctions are important when considering separation between atomic activities [APM.1004.01 93].

The requirement for concurrency control depends upon the effect of evaluating forms within the same object concurrently. The granularity of the evaluation units is usually left as an unconstrained implementation issue but this is unsatisfactory as will be seen in the examples below. This issue will be considered in more detail below as part of the discussion of integration with the computational model.

1.1.2 Examples

The need for concurrency control in order to ensure that the concrete state corresponds to an appropriate abstract state can be illustrated with a simple example.

The example (in DPL [APM.1014.01 93] [APM.1001.01 93]) is a stack of strings with conventional push and pop operations and also an operation that reports the current depth of the stack.

```
object
  [ ListOfString= List.of(String);
    ss =
      object
        [ state:ListOfString := ListOfString.new();
          depth:Integer := 0;
          interface
            ( push(item:String):()
                [ state := state.cons(item);
                  depth := depth.add(1);
                  ()
                ]
              pop():(String)->noMore()
                [ head tail = state.carcdr(); % preempted here
                  state := tail;
                  depth := depth.subtract(1);
                  head
                ]
             depth():(Integer) [ depth ]
           )
        ];

   etc.
    ]
```

Concrete states in which the variable `depth` does not match the length of the list of items do not correspond to any abstract state. Even with a quite coarse unit of evaluation for the basic forms, concurrency control is required to avoid these erroneous states.

Suppose that the stack is not empty and one `pop` invocation is preempted at the point indicated above (i.e. after the first invocation has captured the old state but not changed the state). If another `pop` invocation occurs in such a way that the new invocation proceeds to completion before the preempted evaluation proceeds, then the stack will be left in an erroneous state. Both pop operations will have returned the same string, they will have removed only one item from the `state` list but the `depth` counter will have been decremented by two. This erroneous state can become visible because the `depth` counter can become negative and this can be seen through the `depth` operation.

If the `depth` variable is removed and the `depth` operation measures the length of the `state` list, the concrete states which correspond to no abstract state are eliminated. This still leaves the possibility of the state of the stack and hence the outcomes of future operations not corresponding to the history of its invocations.

The simple solution in this case is to impose the constraint that at most one of `push` and `pop` be evaluated at a time. Whether or not this constraint needs be applied to `depth` as well depends upon the granularity of evaluation units mentioned above. If concurrent ‘read’s of `depth` are guaranteed all to deliver the correct value and also ‘reading’ `depth` concurrently with an assignment to it is guaranteed to return either the old or the new value and not to disturb the assignment then no constraint is necessary. (The question of whether or not it is more efficient to impose an unnecessary constraint in cases like this will be discussed later.)

1.2 Services with synchronization behaviour

In order to exploit concurrency, it is necessary to have a way for one activity to synchronize with another. If a number of activities are cooperating to perform some task then it will usually be necessary for at least one activity to wait for other activities to complete some part of the task before proceeding.

A concurrency control mechanism which can delay the start of the evaluation of the body of an operation can be combined with ‘wait for result’ semantics of interrogations to provide a synchronization mechanism.

The resulting service may do nothing other than provide synchronisation or it may combine some other behaviour with the synchronization.

A simple binary semaphore can be constructed out of two operations which do nothing except return and a mechanism that ensures that the bodies of the operations are evaluated alternately. Although the operations always return the same result and their bodies neither access nor update any state, the evaluation of an invoker for one of these operations must not complete until the evaluation of the body has been permitted to take place.

The interrogations included in the ANSA computational model have the required property and therefore synchronization can be provided through interfaces with activation constraints applied to their interrogations. Announcements do not have the required property and therefore cannot be used in this way. Although “non blocking calls” can be constructed using the ANSA computational model elements, this depends on the result collection being provided as an interrogation which uses a concurrency guard to synchronise with the result depositing operation.

In addition to the simple binary semaphore, various other services with a synchronization aspect to their behaviour will be presented later in this document. The examples will include services that need to protect their abstractions as well as providing a synchronization service. Eventcounts and

Sequencers [REED79] can be provided as services and will be included in the examples.

1.3 Resource usage

Since activities necessarily consume resources, allowing multiple activities changes the pattern of resource usage. The usual problem introduced by concurrency is that activities need storage resources at the same time rather than being able to use the same resources at different times.

In terms of resource usage, an invocation from a remote client is typically more costly than an invocation from a local client. There are cases of local invocation that need not consume any additional storage resources. If it can be determined that an operation is invoking itself in a tail-recursive context then no additional storage is required.

Both the identity of the invoking activity and the location from which the invocation is taking place are important factors for resource usage. Neither of these are factors in protecting a service abstraction nor in providing a synchronising service. Therefore, although a concurrency control mechanism may limit the consumption of resources, resource management will not be a goal of the concurrency control mechanism.

For the same reasons, the form of concurrency control that will be described in this document should not be considered as an important resource management mechanism, and should not be considered a substitute for a properly thought out resource management strategy.

2 Integration with Computational Model

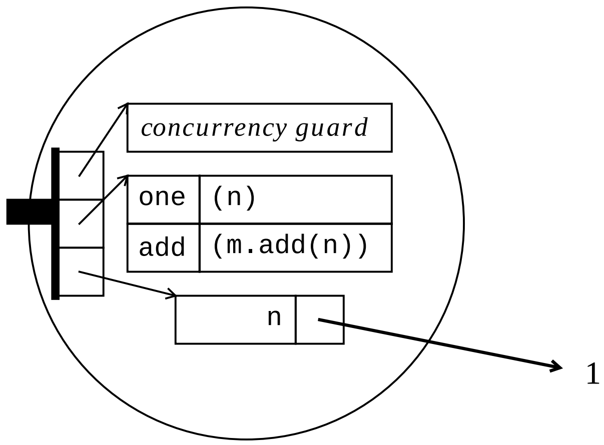
The ANSA computational model [APM.1001.01 93] describes where concurrency control mechanisms may be placed. This chapter introduces the basic elements of a concurrency control model which may be used in that context.

The proposal described here is based upon some experiments which have been performed using a prototype concurrency control mechanism added to the DPLTools implementation. The main purpose has been to establish the semantics of a useful set of elements. Two syntaxes are presented below, and used in the examples, one is an infix style based on the syntax used by Andler [ANDLER79], the other is a prefix style. Both syntaxes were supported by the prototype implementation.

2.1 Invocation events

The description of the ANSA Computational Model [APM.1001.01 93] shows concurrency guards associated with interfaces, as illustrated in Figure 2.1

Figure 2.1: Interface with concurrency guard



The major aim of this work is to describe a concurrency guard model that fits into the ANSA computational model. Secondary aims are to relate this model to previous work and to demonstrate the feasibility of the model by producing a prototype implementation.

The description of the evaluation of an invoker explained when the request, activate and terminate events occur.

> The arrival of the invocation request at an interface is the request event for invocation of the operation named in the request. The activate event occurs after the request event at a time determined by the concurrency guard for the interface. After the activate event, the body of the operation is evaluated in the naming context defined by the interface augmented by

constant bindings of the argument names in the signature of the operation to the parameters in the request. The terminate event of the invocation occurs when the evaluation of the body is complete and delivers its outcome.

When the request event has occurred, the concurrency guard is tested. If activation of an invocation of the named operation is permissible then the activate event occurs and the activity is allowed to proceed. The activate event changes the state of the concurrency guard. The state change and granting of permission must be performed indivisibly with respect to other uses of this concurrency guard.

If activation of an invocation of the named operation is not permissible then the state of the concurrency guard is not changed and the activity is not permitted to proceed. In this case it must be arranged that the concurrency guard will be tested again later. There are various ways to achieve this effect and any mechanism or combination of mechanisms that achieves the required effect may be used. Mechanisms usually involve either a state change trigger or repeated attempts at the test.

If the mechanism involves a pool of waiting activities to be tested when a state change occurs, then the test for permission to proceed and the adding of the activity to the pool must be performed indivisibly with respect to other uses of the concurrency guard.

If the mechanism involves repeated tests then the existence of an activity that is waiting to be allowed to enter an operation must not prevent other activities from making progress.

2.2 Syntax and semantics

This section proposes that the basic elements of the concurrency control model be taken from Andler’s PPEs.

2.2.1 Semantics of the elements

The semantics are presented informally here with the aid of examples.

2.2.1.1 Underlying model

Basic concept is the “path”. The model consists of a primitive path constructor and operations that construct a path from one or more paths.

A path could be modelled as a set of sequences of events (almost - you need a notion of ‘exits’ as well). There are quite a lot of constraints on what constitutes a valid path (e.g. initial events must all be activate events, every activate must have a matching terminate later in the sequence etc.) Possible states of a concurrency guard are also sets of sequences of events but less constrained than paths.

2.2.1.2 path elements

An informal summary of the path elements (in the syntax based on Andler’s notation). The description of how to translate to a state machine is probably easier to understand than an exposition in some process algebra or mathematical formulae. The algebraic properties also give some helpful hints.

opname	activate(opname) then terminate(opname) then exit

path1 ; path2	path1 up to an exit then path2
path1 + path2	any sequence permitted by either path1 or path2
path1 , path2	path1 interleaved with path2, exits are wherever both paths have an exit.
path *	exit or path ; (path *) - i.e. zero or more complete traversals of path.
{ path }	exit or path , { path } - i.e. zero or more interleaved copies of path. Note that this differs from Andler who defined this construct to mean one or more.
path [predicate]	if predicate then path - i.e. if the predicate is true then a sequence in the path may be followed otherwise it may not. The predicate controls entry to the path, it need not be true for any subsequent steps.

2.2.1.3 *Predicates*

Note: The specification of predicates is likely to change in view of the issues described in §3.3.

For each operation in the interface guarded by the path, there is a constant binding of the name of the operation to an interface of type

```
type( req():(Integer)
      act():(Integer)
      term():(Integer)
    )
```

These bindings are in scope in all predicates in the path but are not in scope in the bodies of the operations of the interface.

The three operations deliver the current values of the request, activation and termination counters respectively for the operation to whose name the interface is bound. The request counter will have been incremented for the operation being considered for activation.

The prototype implementation permitted this value to be captured. This counter is not guaranteed to appear to be monotonic if this is done since the reversion of the counter when remote invocation requests are deferred has also been implemented.

A predicate must be a form (see [APM.1001.01 93] Chapter 3) conforming to

```
->true() ->false()
```

An outcome named `true` indicates that the predicate is true for the purpose indicated above.

The context for evaluation of the predicate is the context in which the interface constructor containing the path expression is evaluated augmented by the bindings giving access to the event counters as described above.

This is as implemented in the prototype and will certainly need to be changed. It means that the predicates can observe and update the same variable bindings as can the operation bodies. The predicates are also combined predicate and update forms; these present difficulties which are described in §3.3.6.

2.2.1.4 Examples

If prefix + and - are used to indicate activate and terminate events respectively then the sequences permitted by some example path expressions (alternatives on separate lines) are:

```
A         +A -A

A;B       +A -A +B -B

A+B       +A -A
          +B -B

A,B       +A -A +B -B
          +A +B -A -B
          +A +B -B -A
          +B +A -A -B
          +B +A -B -A
          +B -B +A -A

A*        +A -A +A -A +A -A …

{A}       +A -A +A -A +A -A …
          +A -A +A -A +A +A …
          +A -A +A +A -A -A …
          +A -A +A +A -A +A …
          …
          +A +A +A +A +A +A …

(A+B)*    +A -A +A -A +A -A …
          +B -B +A -A +A -A …
          +A -A +B -B +A -A …
          +B -B +B -B +A -A …
          +A -A +A -A +B -B …
          +B -B +A -A +B -B …
          +A -A +B -B +B -B …
          +B -B +B -B +B -B …

( A[A.act().subtract(B.act()).lessThan(2)] +
  B[B.act().lessThan(A.act())]
)*

          +A -A +B -B +A -A +B -B …
          +A -A +B -B +A -A +A -A …
          +A -A +A -A +B -B +B -B …
          +A -A +A -A +B -B +A -A …
```

2.2.2 Algebraic properties

Implicit universal quantification over all paths in the following:

2.2.2.1 ‘;’ is associative

It is neither commutative nor idempotent.

```
(path1 ; path2) ; path3 = path1 ; (path2 ; path3)
```

This means that we can leave out the brackets without any problems.

2.2.2.2 *‘,’ is associative and commutative*

It is not idempotent.

```
(path1 , path2) , path3 = path1 , (path2 , path3)

path1 , path2 = path2 , path1
```

2.2.2.3 *‘+’ is associative, commutative and idempotent*

```
(path1 + path2) + path3 = path1 + (path2 + path3)

path1 + path2 = path2 + path1

path1 + path1 = path1
```

2.2.2.4 *‘;’ distributes over ‘+’*

```
path1 ; (path2 + path3) = (path1 ; path2) + (path1 ; path3)

(path1 + path2) ; path3 = (path1 ; path3) + (path2 ; path3)
```

This may help to clarify the meaning of +.

‘;’ does not distribute over ‘,’. Neither ‘+’ nor ‘,’ distributes over the other nor over ‘;’.

2.2.2.5 *identity elements for the operators*

Andler [ANDLER79] does not provide a way to write the identity elements for the operations described above. Habermann [HABERMANN75] uses epsilon ε as the identity element for ‘;’ in an explanation but not in the notation itself.

Given the explanations above, `exit` will be used to mean a path which can exit immediately but do nothing else. The path which cannot be entered (false predicate) will be written `blocked`.

The identity element for ‘;’ and also for ‘,’ is `exit`:

```
path1 ; exit = path1
exit ; path1 = path1
path1 , exit = path1
exit , path1 = path1
```

The identity element for ‘+’ is `blocked`:

```
path1 + blocked = path1
blocked + path1 = path1

blocked ; path1 = blocked
path1 ; blocked  neither path1 nor blocked - path1 with exits removed.
```

2.2.2.6 *Other miscellaneous equivalences*

(These are unproven but are correct for all cases considered to date.)

```
{path1},{path2} = {path1+path2}

(path1 ; path2)[pred] = path1[pred] ; path2

path1** = path1*

{{path1}} = {path1}
```

```
{path1*} = {path1}

{path1}* = {path1}
```

2.2.3 Syntax

Two syntaxes are described here, the first is a prefix notation, the second an infix notation based on Andler's syntax. Both were supported by the prototype, the choice made by attribute name as explained below.

For both syntaxes, the non-terminals `attributeName`, `attributeBlock`, `number` and `expressionList` are as used in the DPL syntax given in the DPL Programmers' Manual [APM.1014.01 93].

The notation used to describe the ANSA Computational Model [APM.1001.01 93] adopts the ordering predicate syntax as part of an interface constructor.

2.2.3.1 Ordering predicate syntax

attributeName	=	'ordered'
attributeBlock	=	'(' ordered-string ')'
ordered-string	=	'"' path-expr '"'
path-expr	=	operator '(' pe-list ')'
operator	=	choice | concurrent | sequence
choice	=	'cho' | 'choice'
concurrent	=	'con' | 'concurrent'
sequence	=	'seq' | 'sequence'
pe-list	=	{ pe-expression }
pe-expression	=	pe-composite | pe-unit-expr
pe-composite	=	pe-composer ':' pe-unit-expr
pe-unit-expr	=	operation-name | path-expr
pe-composer	=	'*' | number | '[' expressionList ']'

For example, a path appropriate to a testable latched event (which is described in more detail below) would be written in this syntax as:

```
interface
<ordered ("seq( *:test open *:cho(test open enter))")>
( test()…
  open()…
  enter()…
)
```

This concurrency guard permits arbitrarily many invocations of `test` followed by an invocation of open followed by arbitrarily many invocations of `test`, open and `enter` in any order. All of the invocations are restricted to one at a time.

This syntax has some deficiencies, the '**`*:`**' construct is context sensitive if considered as a function from a path to a path. The numeric multiplier facility has been restricted to the well defined part - i.e. literal integer multipliers.

2.2.3.2 PPE syntax

```
attributeName    =  'path'
attributeBlock   =  '(' pathString ')'
pathString       =  "" ppe ""
ppe              =  ppe-con [ '+' ppe ]
ppe-con          =  ppe-seq [ ',' ppe-con ]
ppe-seq          =  ppe-seq-rpt [ ';' ppe-seq ]
ppe-seq-rpt      =  ppe-pred [ '*' ]
ppe-pred         =  ppe-term [ '[' expressionList ']' ]
ppe-term         =  opname | '(' ppe ')' | '{' ppe '}'
                 =
                 =
                 =
                 =
```

The example given above would be written in this syntax as:

```
interface
<path ("test*;open;(test+open+enter)*")>
( test()…
  open()…
  enter()…
)
```

2.3 Options and their complexity

The path expressions described above have many features, some of which introduce considerable complexity in both specification and implementation. This section outlines options for simpler subsets. The examples in the next section and the discussion in subsequent chapters should clarify the issues mentioned here.

The option ultimately chosen will depend upon establishing which features are valuable is solving realistic problems. A particular intended use of the concurrency control mechanisms is in support of the work on atomicity and that work will provide important input for deciding which option to adopt.

2.3.1 Default case options

If, as currently proposed, the path expression may be omitted altogether, then the default path must be specified. The absence of a path expression may be taken to imply no concurrency control, i.e. arbitrarily many concurrent copies of any operations in the interface. This would be equivalent to the path

```
<path ("{ choice between all operations in the interface }")>
```

The implementation of this path is trivial and the run-time mechanism has minimal cost. Whatever other simplifications are adopted, this can be provided as a special case.

2.3.2 Basic paths

Path expressions composed of the following operators are straightforward to analyse and can be implemented using a finite state machine driven mechanism.

```
opname

path1 ; path2

path1 + path2

path1 , path2

path *
```

The simplest option is to include only these operators in addition to the default case.

2.3.3 Arbitrarily concurrent paths

Considerable additional complexity is introduced by including

```
{ path }
```

This operator permits arbitrarily many concurrent copies of a path which means that the number of states of the concurrency control manager is no longer bounded.

Analysis of path expressions containing this operator is considerably harder than for the basic path constructs.

The current implementation of paths containing this operator involves additional complexity in the run-time support functions and significant additional run-time evaluations to deal with entry to and exit from this construct. The unbounded state space also means that dynamic storage allocation is required.

2.3.4 Predicates

Predicates introduce conceptual problems rather than implementation complexity. The major issue is in defining what is in scope in the predicate. This is discussed in detail in Chapter 3.

A particular problem is whether or not the evaluation of a predicate could require a remote invocation. Given access transparency, it is difficult to define the restrictions that are necessary to prohibit remote invocations since the concept of remoteness is not directly available.

2.3.4.1 What can be predicated

```
path [ predicate ]
```

If predicates can be applied to arbitrary paths then it may be necessary to evaluate a logical composition of the predicates. The conditions under which this occurs are defined in §3.3.

A restriction which is simple to explain and which would eliminate such cases is to allow only operation names to be predicated rather than arbitrary paths. This would also mean that the name of the operation being considered for activation would be known, this is not necessarily the case otherwise.

2.3.4.2 Extended predicates

EOPPEs [WARNE89] introduced an extension to allow “predicates” to make state changes but this introduces significant problems. The question of when the state change occurs being the most serious. The issues are described in §3.3.

Adopting the restriction of predicates to operation names would eliminate the more difficult cases of defining when state changes may occur.

2.3.5 Options

The options for path expressions, roughly in increasing order of complexity, generally in both specification and implementation, are:

- Basic path expressions (only)
- Basic + opname predicate (no update)
- Basic + path predicate
- Basic + { }
- Basic + { } + opname predicate
- Basic + { } + path predicate
- Basic + opname predicate + update
- Basic + { } + opname predicate + update
- Basic + path predicate + update
- Basic + { } + path predicate + update

Basic path expressions with predicates which may refer only to activate and terminate counters is the choice for further development.

2.4 Examples

The literature on path expressions includes many examples. These are used to demonstrate how particular problems can be solved using the particular notation being proposed. The examples are presented here both to show how they appear in the notation defined above and also as a basis for determining whether or not all of the features described above need be supported.

2.4.1 Exclusive read and write

Campbell and Habermann [CAMPBELL74] introduced an example which “… specifies a series of executions by processes of the procedures read and write in unpredictable order, none of which overlap in time.” In the original notation this was written:

```
path read, write end
```

In the notation(s) proposed in this document, this can be written in either of the two forms:

```
interface <ordered ("seq(*:cho(read write))")>
  ( … ops … )

interface <path ("(read+write)*")>
  ( … ops … )
```

2.4.2 Concurrent reads exclusive write

That example was immediately followed by another which "… specifies a series of executions by processes of the procedures read and write in unpredictable order. Read executions may overlap other read executions but write executions may not overlap other read or write executions. Reading, once started, will continue for as long as there are processes invoking read and at least one process executing read."

```
path {read}, write end
```

This can be written in either of the two forms:

```
interface
  <ordered ("seq(*:cho( con(*:read) write))")>
  ( …ops…)

interface
  <path ("({read}+write)*")>
  ( …ops… )
```

2.4.3 Current operation priority

The next example was the first of a series of examples intended to demonstrate various priority and fairness schemes. This example specifies "… a policy in which, once writing commences, all processes requesting writing will proceed provided they do so one at a time."

```
path {read}, {WRITE} end
     where WRITE = begin write end
     path write end
```

This approach can be used directly in the translation. Note that the various signatures and also the arguments to the invocation would need to be filled in, in order to complete the example.

```
[ x=interface <ordered ("seq(*:write)")>
    ( write… );
  interface <ordered ("seq(*:cho( ind(*:read) ind(*:write)))")>
  ( write(…):(…) [ x.write(…) ]
    read…   ) ]

[ x=interface <path ("write*")> ( write… );
  interface <path ("({read}+{write})*")>
  ( write(…):(…) [ x.write(…) ]
    read…
  )
]
```

Using predicates avoids the indirection but it is not immediately obvious whether or not this path expression has the same effect as the solution given above.

```
[ reader:Boolean := true;
  interface
  <path ("( {read}
            [ after (write.req()
                     .greaterThan(write.act()))
              handle
              ( true()  [ reader.if() ]
                false() [ reader:=true; ->true() ]
              )
            ] +
            write [ reader:=false; ->true() ]
          )*")>
  ( …ops…)
]
```

2.4.4 Fair read and write

The next example introduces a notion of fairness. “Another strategy is to give individual read and write invocations by processes an equal chance of executing first. This can be achieved by having each process invoke a READ or a WRITE procedure which first obtains permission before performing the read or write.”

```
path requestread, requestwrite end
path {openread; read}, write end
where requestread  = begin openread end
      requestwrite = begin write end
      READ         = begin requestread; read end
      WRITE        = begin requestwrite end
```

The direct translation is:

```
[ x = interface <path ("({openread; read} + write)*")\>
       ( openread():() []
         read(…):(…) …real read operation…
         write(…):(…) …real write operation…
       );
   y = interface <path ("(requestread+requestwrite)*")>
       ( requestread():() [x.openread()]
         requestwrite(…):(…) [x.write(…)]
       );
   interface
   ( read(…):(…) [ y.requestread(); x.read() ]
     write(…):(…) [ y.requestwrite(…) ]
   )
]
```

Note that the example above will let the first writer which arrives go as soon as all the active reads finish. The write is guaranteed to go before any reads or writes which arrive later.

This effect cannot be achieved with predicates applied to a single level interface structure - the first writer needs to be distinguished from any other writer that arrives before the active reads complete. The solution with a two level structure and predicates is:

```
[ x = interface <path ("({read} + write)*")>
      ( read(…):(…) …real read operation…
        write(…):(…) …real write operation…
      );
  interface <path
("{read[write.term().equal(write.act())]},write*")>
  ( read(…):(…) [ x.read(…) ]
    write(…):(…) [ x.write(…) ]
  )
]
```

In this case, the inner interface path expression enforces the required protection, the outer interface provides the “fairness” wrapping.

2.4.5 Writer priority

The next example was a demonstration that a problem solved by another technique could also be solved using path expressions. “Suppose we require the strategy that writing should have priority over reading. This is the readers and writers problem solved by Courtois Heymans and Parnas [COURTOIS71] and requires that when writing is requested, no further reading should be granted and writing should start as soon as the current reading is finished.”

```
path readattempt end
     path requestread, {requestwrite} end
     path {openread; read}, write end
     where readattempt = begin requestread end
           requestread = begin openread end
           requestwrite = begin write end
           READ = begin readattempt; read end
           WRITE = begin requestwrite end
```

Note that in order for the `readattempt` mechanism to do what appears to be intended, a waiting `requestwrite` must be granted permission to proceed indivisibly with the termination of the `requestread` that was holding it up. If this is not the case then the `readattempt` may terminate allowing another `readattempt` which may permit its nested `requestread` to overtake the `requestwrite`.

There is, once again, a direct translation which would have the same characteristics as noted above. Since the underlying model does not guarantee to activate a waiting activity indivisibly with a termination, the translation will not be presented here.

The required constraint can be expressed simply using a predicate on a single level interface. A read may not proceed if there is a writer waiting. The solution was described by Warne [WARNE89] under the title “strong writer preference” and is:

```
interface
  <path ("( {read[write.req().equal(write.act())]}
          + write )*")>
  ( …ops… )
```

2.4.6 Reader priority

The last example in the series devoted to reader and writer problems was introduced as follows. “Finally, let reading have priority over writing, i.e., when reading is requested, from that moment no further write requests should be granted and reading should begin as soon as the current writing has finished. A write request, on the other hand, should not stop the flow of reading.” For both this and the previous example, it is left to the reader to deduce when the lower priority operation may proceed.

```
path writeattempt end
path {requestread}, requestwrite end
path {read}, (openwrite ; write ) end
where requestread = begin openwrite end
      writeattempt = begin requestwrite end
      requestread = begin read end
      READ = begin requestread end
      WRITE = begin writeattempt ; write end
```

This example is very similar to the previous one and has a similar activation issue. The direct translation is straightforward but with the same problem as before.

The solution using predicates is similar to the previous example:

```
interface
  <path ("( {read}
          + write[read.req().equal(read.act())] )*")>
  ( …ops… )
```

2.4.7 One place buffer

Campbell and Habermann introduced a one place buffer as an example of an object accessible to more than one process that requires synchronization. In this example, the path expression was embedded in a type definition (in an arbitrary syntax not intended to be a proposal for a programming language). Note that the “type declaration” is being used as a factory.

```
type buffer;
message frame;
path write; read end
operations
  procedure read (returns message m) : m := frame;
  procedure write (accepts message m) : frame := m
end type
```

The direct translation into DPL is:

```
[ frame:Message := emptyMessage;
  interface <path ("(write ; read)*")>
  ( write(m:Message):() [frame := m;()]
    read():(Message) [frame]
  )
]
```

The only problem is the need for `emptyMessage` which is never used (and not defined here either). Since DPL does not permit uninitialised variables to exist, the only way to avoid the use of a dummy message is to wrap up the incoming messages in a wrapper that indicates that a message is present and then use a dummy wrapper as the initial value of the variable.

2.4.8 Binary semaphore

This example illustrates the use of empty operation bodies to provide a service with a simple synchronizing effect.

```
interface <ordered ("seq(*:seq(claim release))")>
( claim():() []
  release():() []
)
```

The path expression guarantees that evaluations of the claim and release operations will occur alternately. This means that after claim has been invoked, another claim cannot complete until after a release has completed.

2.4.9 Latched event

```
interface <ordered ("seq(open *:cho(open enter))")>
( open():() []
  enter():() []
)
```

In this example, invocations of enter will wait until an invocation of open has completed. Once an open has been completed, any number of open and enter invocations will be permitted in any order. This path expression constrains these invocations to take place one at a time which is not strictly necessary.

2.4.10 Latched event with test

```
[ open   = interface (test()->open()   [->open()]);
  closed = interface (test()->closed() [->closed()]);
  state:type(test() ->open() ->closed()) := closed;
  interface <ordered ("seq(*:test open
                         *:cho(open test enter))")>
    ( test() ->open() ->closed() [ state.test() ]
      open():() [ state := open; ()]
      enter():() []
    )
]
```

This example is similar to the previous example but a `test` operation has been added. The `test` and open operations close over the same binding of `state` thus allowing the open operation to change this binding so that the `test` operation can return a different outcome.

If more than one such interface is required then the whole of the example could be made the body of an operation in an interface that provides an event factory. Having the creation of the variable binding inside the factory operation gives each event a distinct binding of `state`. The constant bindings of open and `closed` could be moved outside the operation and thus shared by all the events created by the factory interface.

2.4.11 Eventcounts and sequencers

And here to round things off is an implementation of Reed and Kanodia's Eventcounts and Sequencers [REED79].

```
object
  [ Sequencer = type( ticket():(Integer) );
    Eventcount = type( advance():()
                       await(n:Integer):()
                       read():(Integer) );
    interface
      ( sequencer():(Sequencer)
          [ n:Integer := 0;
         interface <ordered ("sequence( *:sequence( ticket ) )")>
            ( ticket():(Integer) [ n := n.add(1) ] )
          ]
        eventcount():(Eventcount)
          [ count:Integer := 0;
            waiters:Integer := 0;
            s1 = interface <ordered ("seq(*:seq(c1 r1))")>
            (c1():()() r1():()());
            s2 = interface <ordered ("seq(*:seq(r2 c2))")>
            (c2():()() r2():()());
            ec =
              interface
                ( read():(Integer) ( s1.c1(); count; s1.r1() )
                  advance():() [ s1.c1();
                                 count := count.add(1);
                                 after (waiters.greaterThan(0))
                                 handle ( true()  [ s2.r2() ]
                                          false() [ s1.r1() ] )
                               ]
                  await(n:Integer):()
                    [ s1.c1();
                      after ( count.lessThan(n) )
                      handle
                        ( true() [ waiters := waiters.add(1);
                                   s1.r1();
                                 s2.c2(); % This is the real wait.
                                   after( [waiters :=
                                     waiters.subtract(1)]
                                          .greaterThan(0) )
                                   handle ( true()  [ s2.r2() ]
                                            false() [ s1.r1() ]
                                          );
                                   ec.await(n) ]
                          false() [ s1.r1() ]
                        )
                    ]
                )
          ]
      )
  ]
```

2.5 Evaluation unit granularity

The current definition of the computational model does not explain the unit of granularity for evaluation. There are a number of issues to be considered before a final decision is made.

The requirement is to define the effect of evaluating forms in an object concurrently. What are the primitive ‘operations’ which don’t get messed up.

Most of the questions are about resolving and updating bindings.

Do all instances of concurrent name resolutions of constant bindings necessarily give the same answers as if done one at a time? If this case is unsafe then concurrency guards need to be used to protect everything, including the use of literal constants.

Same question for resolving names of variable bindings. Not quite so bad if this is unsafe.

Resolution of name concurrent with updating its variable binding. Does the resolution necessarily give either the ‘before’ or the ‘after’ value? Is the updating guaranteed not to be disturbed? This can easily be unsafe - if the representation of an interface reference is larger than the storage access granularity then you can get a mixture of before and after. The model should not exclude the use of shared memory multiprocessors nor should it make their use difficult.

Concurrent updating of a binding. The transient name resolution error of the last case becomes persistent. The representation can get left in a mixed state.

The stack example used earlier illustrates that the choice of evaluation granularity affects the path expression needed on an interface.

```
object
  [ ListOfString= List.of(String);
    ss =
      object
        [ state:ListOfString := ListOfString.new();
          depth:Integer := 0;
          interface
            ( push(item:String):()
                [ state := state.cons(item);
                  depth := depth.add(1);
                  ()
                ]
              pop():(String)->noMore()
                [ head tail = state.carcdr(); % preempted here
                  state := tail;
                  depth := depth.subtract(1);
                  head
                ]
             depth():(Integer) [ depth ]
            )
        ];
     x y = (ss.depth() || ss.depth() );
     etc.
  ]
```

Which of the following path expressions should be used depends on the answers to the questions above.

2.5.1 Concurrent constant reads unsafe

If concurrent constant reads are unsafe then the line

```
x y = (ss.depth() || ss.depth() );
```

would have to be rewritten

```
x y = [ss1 = ss; (ss.depth() || ss1.depth() )];
```

in order to avoid the concurrent reads of ss. The same path expression as for unsafe concurrent variable reads would also be needed.

2.5.2 Concurrent variable read unsafe

If concurrent reads of constant bindings are safe but concurrent reads of variable bindings are not then the required guard is:

```
seq( *:cho( push pop depth ) )
```

This path expression restricts evaluations of the operations to one at a time. This is safe whatever unit of evaluation granularity has been chosen.

2.5.3 Concurrent variable reads safe

If concurrent reads of variable bindings are safe but a write concurrent with a read is not then the required guard is:

```
seq( *:cho( push pop con( *:depth ) ) )
```

In this case, evaluations of the body of depth may be allowed to proceed concurrently provided that neither push nor pop are concurrent with them. The concurrent resolutions of depth will deliver the current value provided that neither update is in progress.

2.5.4 Concurrent variable read and write safe

If it is safe to read a variable binding concurrently with a write of it then the following guard may be used.

```
con( seq( *:cho( push pop ) ) *:depth )
```

In this case evaluations of depth may proceed concurrently with other evaluations of depth and with one evaluation of either push or pop. Each resolution of depth will deliver either the new or old value and, given that the activities involved have invoked the operations in such a way that this concurrency can take place, either value is consistent with a possible sequential ordering.

2.5.5 Concurrent variable writes safe

The final case is where concurrent writes to a variable binding are safe. The stack example cannot take advantage of this case since both of the operations that update bindings update two bindings based upon the old states of those bindings. This case can, however, be exploited by a simple container:

```
n:Integer := 0;
interface <path ("{put+get}")>
( put(m:Integer):(Integer) ( n := m )
  get():(Integer) ( n )
)
```

In this case, evaluations of put may be concurrent provided that the final state of n is the value of the argument to one of the invocations.

This case is particularly interesting since the path expression never needs to delay an activation and can therefore be implemented by the absence of any concurrency control mechanism.

2.5.6 Chosen granularity

The option described in §2.5.3 has been chosen as the guarantee that will be required by the computational model.

2.6 Efficiency and unnecessary constraints

The discussion of the appropriate concurrency control expression in the previous section was based upon two assumptions.

All concurrency control forms are of equivalent cost

Maximum concurrency within correctness is required

As will be seen in the next chapter, the first of these assumptions is not generally correct. In some cases, it may be appropriate to impose a concurrency constraint that is more restrictive than necessary for correctness but which has a more efficient implementation.

In the case of the stack example given above, the constraint

```
seq( *:cho( push pop depth ) )
```

has a finite state implementation as described in Chapter 4. The other constraints such as

```
con( seq( *:cho( push pop ) ) *:depth )
```

do not have a finite state implementation using that strategy.

Other implementations, including developments of the strategy proposed below, may change the cost of the various forms of concurrency control expression. It is not clear that the issue described here can be eliminated altogether.

3 Path expressions

The proposed concurrency control mechanism is based upon path expressions. This chapter discusses the various kinds of path expression that have been described in the literature.

3.1 History

3.1.1 Original work

Path expressions were proposed by Campbell and Habermann [CAMPBELL74] as a synchronization mechanism appropriate to abstract data types. It is noteworthy that SIMULA 67 is identified as being the language with the most appropriate type definition concept (object oriented languages had not been invented at that time). This early work included a number of restrictions in order to permit an implementation in terms of semaphores.

3.1.2 Consistent extensions

Habermann [HABERMANN75] described some algebraic properties of path expressions and outlined a strategy for translating path expressions to finite state machines. This approach had been mentioned but not pursued in the earlier paper. Habermann also introduced several new features: conditional path elements, priority choice, connected paths and parallel paths. He demonstrated the use of these features by introducing a number of examples.

Flon and Habermann [FLON76] set out to show that path expressions simplify the verification of concurrent systems. They discuss the relationship between concrete and abstract states of an object and the benefits of being able to restrict the set of execution histories for an object.

Andler proposed Predicate Path Expressions (PPEs) [ANDLER79] as a consolidation of the various published extensions of the original path expressions. The objective was to develop path expressions as a “synchronization tool that allows for easier specification of common synchronization problems”, to develop verification techniques for path expressions and to make an efficient implementation of path expressions available.

All of these developments retained the basic concepts of the original path expressions but there were also developments that did not.

3.1.3 A different approach

Campbell proposed Open Path Expressions (OPEs) [CAMPBELL77] which were significantly different from the original path expressions. In the words of Headington and Oldehoeft [HEADINGTON85] “OPEs have semantics fundamentally opposite from that of most path notations derived from RPEs” (RPE “regular path expression” being their name for the path expressions of

Campbell and Habermann [CAMPBELL74]). This fundamental change is difficult to discern given the similarity of the titles of the papers and is a potential source of confusion.

Headington and Oldehoeft added the predicates of PPEs to OPEs to produce Open Predicate Path Expressions (OPPEs). Their preference for OPEs over PPEs and consequent creation of OPPEs seems to be based upon the desire to have the finite parallelism operator of OPEs - `n:(list)`. Their interest in dataflow architectures appears to be another important factor. The example intended to demonstrate the superiority of OPPEs over both OPEs and PPEs is flawed in that their PPE solution

```
def wb = req(B) - auth(b)
    busya = auth(A) - term(A)
    busyb = auth(B) - term(B)
path ({(A[busya=0 and wb=0]
        B[busyb=0])[busya+busyb<2]})*
```

is unnecessarily complicated. In particular, the OPPE solution

```
def wb = req(B) - auth(b)
path 2:(1:(A)[wb=0], 1:(B)) end
```

should be contrasted with the PPE solution

```
def wb = req(B) - auth(b)
path ((A[wb=0])*, B*)
```

which uses the collateral execution operator ‘,’ which they “have chosen not to discuss”. Even within their own restriction the PPE solution can be simplified - the predicate `[busya+busyb<2]` is unnecessary.

3.2 OPs and PPEs compared

3.2.1 Basic path constructs

The Ordering Predicates (OPs) defined in 2 are very similar to Andler’s PPEs. Some OP operators correspond directly to PPE operators and there are some OP constructs that are artefacts of the prefix style of syntax. OPs, unlike PPEs, include constructs that correspond to the identity elements of the operators. OPs also have some constructs and some operators which do not have any analogy in PPEs.

3.2.1.1 Equivalent constructs

The basic PPE constructs which correspond directly to OP constructs are:

```
PPE            OP

opname         opname
path1;path2    seq(path1 path2)
path*          seq(*:path)
path1+path2    cho(path1 path2)
path1,path2    con(path1 path2)
{path}         con(path *:path) - i.e. PPEs require one or more
```

Since ';', '+', and ',' are all associative, the OP forms with more than two items have a straightforward interpretation. E.g.:

```
path1;path2;path3seq(path1 path2 path3)
```

3.2.1.2 *Syntax artefacts*

Since OP operators take an arbitrary length list of operands, it is possible to have a single operand. These cases have no analogy in PPEs which use binary infix operators.

A simple interpretation is to make all operators applied to a single element be the identity function. This means that `seq(path)`, `cho(path)` and `con(path)` are all equivalent to `path`

3.2.1.3 *Identity elements*

`seq(), cho(), con() -` these forms can reasonably be interpreted as the identity element for the given operation. This would mean that `seq()` and `con()` were the "empty action" used by Habermann [HABERMANN75] in transforming to remove ambiguities and that `cho()` is a "dead end" or blocked path as described in §2.2.2.5 (this is how these forms were interpreted by the prototype implementation).

3.2.1.4 *Multiple choice*

The OP constructs `seq( *:path )` and `con( *:path )` have been related to PPE constructs `path*` and `{path}` respectively but there is no likely analogy for the construct `cho(*:path)`.

Since choice is idempotent[1], any number greater than one of occurrences of the expression is equivalent to one occurrence. Therefore, there are only two reasonable interpretations of `cho(*:path)`, either `cho( cho( ) path )` which simplifies to `path` or `cho( seq( ) path )`. The prototype implementation adopted the first interpretation, but further thought suggests that the second might be more useful. The second interpretation provides an "optional path" construct although the syntax does not make this clear.

3.2.1.5 *Numeric multipliers*

The numeric multiplier feature of OPs has no analogue in PPEs. For literal positive integer, the interpretation is as if the path were included the specified number of times. Extending this interpretation to a zero multiplier is straightforward, it is as if the path had not been written at all.

There are a number of unanswered questions about possible extended forms of numeric multiplier which will need to be resolved if they are to be supported. For example, is the following usage permissible?

```
[ x = interface
   ( op(n:Integer):(type( x():() y():()))
       [ interface <ordered ("seq(*:seq(n:x y))")>
            ( x():() []
              y():() []
            )
       ]  )
]
```

1. in the mathematical sense - $\oplus$ is idempotent iff $\forall\ x \in \mathrm{dom}\ \oplus \cdot x \oplus x = x$

This example would require that the value of the multiplier be captured at the time of creation of the interface. Allowing a multiplier to refer to a variable binding presents even greater difficulties, as it would allow the multiplier value to change dynamically.

3.3 Predicates

The predicates introduced in PPEs are an additional way to control entry to a path. A true predicate permits entry to a path, as if the predicate were not present. False predicates do not have such an obvious interpretation.

3.3.1 False predicates

There are three possible ways to interpret false predicates. False predicates in Andler's PPEs prohibit the taking of all paths starting with the predicated form. The second option is to interpret a path with a false predicate as a short circuit. The difference between these two can be illustrated by an example containing a constant false predicate.

```
    cho( seq( [->false()]:op1 op2) seq(op3 op4))
1)= cho( seq( op3 op4) )
2)= cho( op2 seq(op3 op4) )
```

The third option is to interpret a path with a false predicate as the identity element for the closest enclosing operator. As was pointed out above, the identity element for choice differs from the identity element for sequence, so this is the least satisfactory option.

The short circuit interpretation can be translated into a form where predicates are interpreted as blocking by a simple syntactic expansion:

```
[P]:path  →  cho( [P]:path [not P]:seq() )
```

The path

```
seq( *:seq( [P1]:seq(a *:a) seq(b *:b)
            [P2]:seq(c *:c) seq(d *:d) ) )
```

with predicates interpreted as blocking cannot be expressed in terms of short circuit predicates (at least, no solution has been found).

Since blocking predicates are at least as powerful as short circuit predicates and the converse has not been shown, predicates will be taken to have blocking semantics, as they do in Andler's PPEs.

3.3.2 Combination of predicates by path operators

When a path operator is applied to paths that contain predicates, the resulting path may have logical combinations of those predicates guarding activation of some of the operations.

For example:

```
(1)      (A*[pa] + B*[pb]); Z
(2)      (C*[pc] , D*[pd]); Z
(3)      ((A*[pa] + B*[pb]) , C*[pc]); Z
```

In the first example, the effect of the repeated sequence operator combines with the effect of the predicate to give a “go on if true” effect. Two of these are combined by the choice operator to allow **Z** to be activated immediately if either pa or pb is true.

In the second example, choice is replaced by concurrent and this allows Z to be activated first if both pc and pd are true.

The third example is just a combination of the first two to illustrate that arbitrarily complex combinations of predicates may be constructed. In this case the initial guard for Z is $(pa \vee pb) \wedge pc$.

The combining effect of the path operators is dependent on the semantics of predicates. The examples given here use the semantics adopted at the end of the previous section. If the short circuit semantics had been adopted it would have meant that Z could be activated immediately without reference to any predicates for all of the examples - indeed, the problem with those semantics is the difficulty of attaching a guard to Z in the way described here.

The combining effect of path operators on predicates will have important consequences for the issues raised below.

3.3.3 When are predicates evaluated

Predicates, to be of any practical use, must depend upon mutable state. This means that the relative times of changing of state and evaluating of predicates must be considered. This, in its turn, depends upon what state can be observed by predicates.

A number of problems arise if predicates can observe state that can change while the predicate is being evaluated. Since combinations of predicates may need to be tested as was described above, it is possible for the test of a combination of predicates to deliver a result which is consistent with neither the initial nor the final values of the state observed by the predicates even if each individual predicate evaluation is indivisible with respect to those state changes.

Since activities may be delayed as a result of a predicate test, it is necessary to arrange that the predicate is tested again later. The effect of a path expression with a predicate should not depend upon whether the retest is caused by a timer or by the concurrency control mechanisms detecting a change of state. In order to give the implementer some freedom to choose an efficient implementation, the definition of predicates should be chosen so as not to require a specific implementation mechanism.

A further question is whether or not the specification should define whether or not predicates may be evaluated more than once between state changes and whether or not such multiple evaluations must take place.

Avoiding onerous restrictions on when or how many times a predicate may be evaluated will be an important factor to be remembered in the discussions of extended predicates and the state that may be observed by a predicate.

3.3.4 Implicit event counters

In Andler’s PPEs, “the predicates are expressed entirely in terms of implicit counters req(g), act(g) and term(g) …” [ANDLER79]. This differs from Habermann’s conditionals [HABERMANN75] in which “the operands are either constants or fieldnames of the type definition in which the path expression is defined".

The path expression part of PPEs, OPEs, OPPEs, EOPPEs and Ordering Predicates can all be defined in terms of the activate and terminate events, as was shown in section §2.2.1.4, but not in terms of the counters. These events can be considered to occur at the interface to the service provider and so can be considered to occur in a sequence without any problems of different sequences being used by different observers.[1]

The introduction of predicates makes the event counters accessible and in the words of Headington and Oldehoeft "These predefined counters are non-negative, monotonically increasing integers ...".

Distribution introduces a problem for the request events and their associated counter (but only because the counter is made accessible to predicates). The problem is in when and where the events are considered to occur. For the activate and terminate events there is no problem - they can be considered to occur at the interface which is necessarily at the location of the operation, and to occur when the concurrency guard (also at that location) updates its view of what is permitted to happen.

The situation is not so simple for the request event. If the event is considered to occur at the client when a client invokes an operation, then there is a problem if the client and server are not co-located. In such a case, relativity tells us that the event cannot be observed by the server (and thus be included in the count used by the predicate) until the server enters the forward light cone of the event. In practice, the server cannot find out until a message arrives from the client, and this information is usually conveyed implicitly in the invocation request message. The suggests that the request event must be considered to occur at the server.

The various kinds of path expression all assume that delayed activities can be woken up at will. This, combined with the monotonic counter and the fact that the request event occurs before evaluation of the predicate, limits the options for when the event can be considered to occur. The server cannot consider the event to occur until it accepts responsibility for performing the invocation. If it is busy and chooses to ignore incoming requests (in the expectation of retransmission) or to send some sort of 'try again later' response then it should not consider the request to have occurred since it is not in a position to proceed with the invocation. This means that any such decision cannot depend upon whether or not the operation would be permitted to proceed by the concurrency guard since this cannot be determined until after the request event which must be after the decision has been made.

It would be possible to adopt a client retry strategy if the predicate could be evaluated as if the request event had occurred but allowing the state to revert if the request is to be rejected rather than being accepted. This has an impact on the predicate extension described below.

There is a further problem with the visibility of these counters. Any policy which includes the possibility of rejection of invocations that cannot proceed must also define whether or not any of the counters are incremented. If it is to be possible for the client that invokes an operation to receive a response

1. The relationship between replicated services supported by a group mechanism and the requirements of activate and terminate counters has not yet been studied in detail.

indicating rejection then it must be decided whether or not the counters are incremented for that operation. This may lead to an apparent contradiction between the state of the path-following part of the concurrency manager and the state of the counters.

3.3.5 State observable from predicates

As noted above, Headington proposed that the state of the object being guarded should be visible to a condition (which was like a predicate but with ‘else if’ and ‘else’). Restrictions were imposed on what state might be observed in order to ensure that there was no possibility of an operation changing state that was concurrently being observed by a condition.

Applying this restriction to the path expressions presented above would require an analysis of the predicates and operation bodies of the kind that is required to determine separation requirements.

The current proposal will prohibit observing by predicates of state that may be changed by operations.

3.3.6 Extended predicates

Habermann’s conditionals and Andler’s predicates allow observation but not modification of state. If a predicate is permitted to include an assignment form then a number of questions arise. In particular, there are a number of problems introduced by the possibility of a state change caused by evaluation of a predicate which has a false outcome.

A predicate might be evaluated many times before it eventually delivers a true outcome. If the predicate has some incremental effect the state will depend on the implementation strategy which chooses when, and how often, to evaluate the predicate.

A predicate which has been evaluated and delivered a false outcome might not be evaluated again (e.g. `((a[pa]+b[pb]),c);(a+b+c)*)`.

It becomes possible to write a predicate for which the eventual true outcome depends upon there having been an earlier false outcome. The task of analysing the predicate in order to optimise the concurrency control mechanism becomes harder (such a predicate must be evaluated even if it is already known that it will be false).

When the path combining operators are applied to paths that contain predicates, it turns out that the problem of when state changes can occur applies to true predicates as well as to false predicates. A true outcome from a predicate does not necessarily lead to activation of an operation since it may be necessary for some other predicate also to be true for the activation to take place. A false outcome does not necessarily mean that the activation being contemplated will not take place since the truth of some other predicate may be sufficient.

The problem of when the request counter is incremented was described above. If predicates can make state changes then they can capture the value of the request counter thus making reverting to the previous state a non-trivial issue. Although atomicity mechanisms that can deal with reversion to earlier states have been developed in related work [APM.1004.01 93], such mechanisms will depend upon an underlying concurrency control mechanism that this work is intended to provide.

The examples used to justify path expressions must be studied to determine whether or not the benefits of the assignment extension are worth the cost of producing an adequate definition.

4 State Machine Implementation

Path expressions can be translated into a state machine based implementation. Path expressions that do not contain the repeated concurrency construct `{ path }` (`con( *:path )`) can be translated into a finite state machine based implementation.

4.1 Finite state translation

For those path expressions that can be translated to a finite state machine control mechanism, the activate and terminate events correspond to transition arcs between states. For simplicity, the states will be numbered. Each state corresponds to a set of permitted future sequences of events.

In the finite state case, the current state of the concurrency guard can be represented as just a state number. While it may be convenient to cache the outcomes of predicates which have been evaluated since the last state change, this need not be done and the effect of the concurrency control mechanism should not depend upon whether or not it is done.

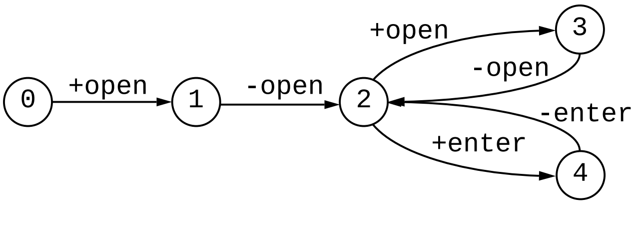
4.1.1 Example - latched event

The FSM for the latched event example for which the path expression was

```
seq( open *:cho( open enter ) )
```

has the transition graph shown in 4.1. As in the previous examples, the activate events are identified by a '+' before the name, the terminate events by a '-'.

Figure 4.1: Transition graph for latched event



The states are numbered from zero and state zero is always the initial state.

From state zero, the only permitted transition is by the activate event of open which goes to state 1. From state 1, the only permitted transition is by the terminate event of open to state 2. From state 2, two activate events are permitted with the corresponding terminate events returning to state 2.

4.1.2 FSM construction

Creating a finite state machine from a path expression can be done in a number of stages.

The strategy that has been adopted is to use an intermediate form of finite state machine which has a number of “exit states” as well as the internal transitions. These “exit states” are used to construct new transition graphs out of existing transition graphs. The FSM with exits (hereinafter called XFSM) is converted to the FSM required by the run-time mechanism as a final stage by deleting the exit state information.

4.1.2.1 Operation name

```
PPE       opname
OP        opname
```

Activate then terminate then exit.

4.2 shows the transition graph for operation name a. Exit states are indicated by boxes, in this case there is a single exit - state 2.

Figure 4.2: Transition graph for operation name a

4.1.2.2 Sequence

```
PPE       path1;path2
OP        seq(path1 path2)
```

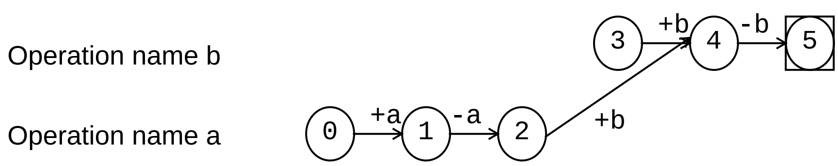
All of the transitions from state zero of path2 are added to each exit of path1, ‘and’ing the transition predicate with the exit predicate to form the predicate for the new transition. All the states in path2 are renumbered by adding the number of states in path1.

The exits of the new transition graph are the exits of the renumbered path2. If state zero of path2 (in its original numbering) is an exit then all of the exits of path1 are exits of the new path but with predicates constructed by ‘and’ing the exit predicate of state zero of path2 with their exiting predicate.

This construction formula deals with multiple exits from path1 and also with cases where path1 and path2 contain repeated sequences and predicates.

4.3 shows how this constructs the transition graph for the sequence a; b. State 2 was an exit for the operation name a but now has the same transitions as state 3 which was state zero for operation name b.

Figure 4.3: Transition graph for sequence a; b



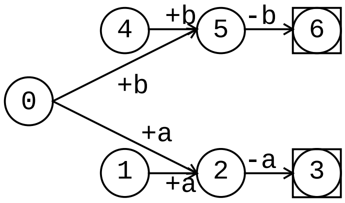
4.1.2.3 Choice

```
PPE       path1+path2
OP        cho(path1 path2)
```

A new state zero is created with the union of the transitions of the states zero of path1 and path2. If either old state zero was an exit then so is the new state zero, the predicate for the new zero exit is the 'or' of the predicates of the old zero exits (not being an exit is equivalent to having a predicate of 'false'). All other state which were exit states remain exit states. All of the old states are renumbered. Creating a new state zero is necessary to deal with a choice between repeated sequences.

4.4 illustrates the construction of the graph for the choice **a+b**.

Figure 4.4: Transition graph for choice a+b



4.1.2.4 *Concurrent*

```
PPE      path1,path2
OP       con(path1 path2)
```

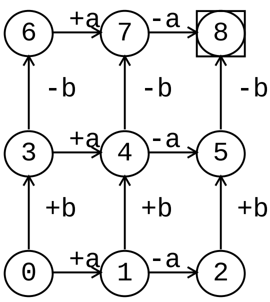
A sort of state product transformation. Applied to n state and m state machines, result is an n*m state machine.

The new graph has a state for each (s1,s2) pair where s1 is a state from path1 and s2 a state from path2. If path1 has a transition T from state s1a to s1b then for each state s2 in path2 there is a transition T from (s1a,s2) to (s1b,s2). A similar rule applies to transitions in path2.

A state (s1,s2) is an exit if s1 and s2 are both exits of their respective paths, the predicate for the new exit is formed by 'and'ing the predicates of the original exits.

4.5 shows the graph for `a, b` - state (s1,s2) is numbered s1+3*s2.

Figure 4.5: Transition graph for concurrent a, b



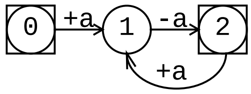
4.1.2.5 *Repeated sequence*

```
PPE      path1*
OP       seq(*:path1)
```

Make state zero an unconditional exit, add all state zero transitions to each exit state.

4.6 shows the transition graph for a*. State zero has been made an exit and the +a transition of state zero has been added to state 2.

Figure 4.6: Transition graph for repeated sequence a*



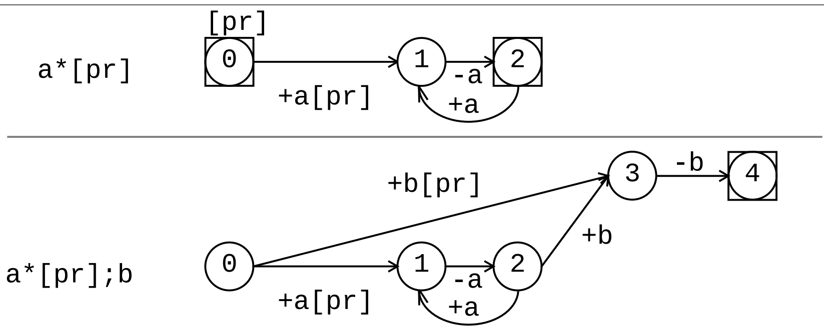
4.1.2.6 *Predicate*

```
PPE       path1[predicate]
OP        seq([predicate]:path1)
```

A new state zero copied from the old state zero but with the predicate added to all the transitions from the old state zero. If the old state zero was an exit then the new state zero is also an exit but with the predicate added. The new predicate is added by ‘and’ing it with any existing predicate for the transition (or exit).

4.7 shows the graph for `a*[pr];b`. A new state zero has been added with a predicated transition and exit, the old state zero was unused and so removed. The sequence construction is applied leaving the graph as shown.

Figure 4.7: Transition graph for predicated path a*[pr];b



4.1.2.7 *Identity elements*

```
PPE
OP        seq() con() cho()
```

The identity for seq and con is a graph with only state zero. State zero is an exit and the graph has no transitions.

The identity for cho is a graph with only state zero. State zero is not an exit and the graph has no transitions.

4.8 shows the graphs of the identity elements.

Figure 4.8: Transition graphs for identity elements

4.2 Unbounded state translation

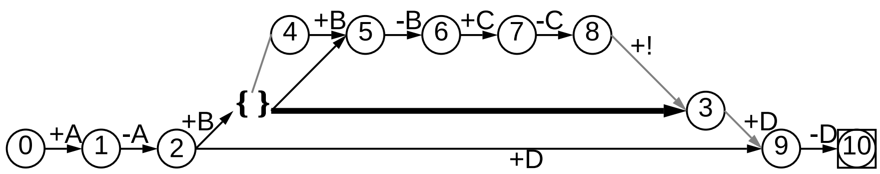
The use of `{path}` causes the number of states to be unbounded so a different approach is needed. It is no longer possible to represent the current state of a concurrency guard as a single number.

The basic state machine approach can be extended to cope with this case by replacing the single state number with a list (representing a bag) of active states, a restart state and an exit state. This technique can be applied recursively to deal with nested concurrent repetitions.

4.2.1 Example

The graphs can be augmented for this extension.

Figure 4.9: Graph for A;{B;C};D



In state 2 there are two transitions. One of these is to state 9 on `+D`, the other occurs on `+B` into a multiple state. The thick arrow points to the exit state, the dotted line to the restart state and the normal arrow points to the active state (in general there may be several such arrows).

Starting from state 0, the permitted events and states reached are:

```
+A => 1
+A -A => 2
+A -A +D => 9
+A -A +B => (3 4 5)
```

The first two events must be `+A` and `-A` ending in state 2. After that, there is the choice described above. If the `+B` event occurs then the subsequent state is represented as `(3 4 5)`. The first and second elements of the list being the exit and restart states, the third being the one active state.

In a multiple state, the permitted transitions are all of the transitions of all of the active states, all of the transitions of the restart state and, if all of the active states are at an exit point (indicated by a transition named '!'), the transitions of the exit state. State 5 is not an exit point so the permitted transitions are `+B` from state 4 and `-B` from state 5.

```
+A -A +B => (3 4 5)
+A -A +B +B => (3 4 5 5)
+A -A +B -B => (3 4 6)
```

The `-B` transition changes the active state to state 6. The `+B` transition adds a new active state 5.

Two of the possible sequences that may follow the `-B` transition are shown below. In the first, a `+D` may occur when the only active state is state 8 since this is an exit state. The second illustrates that a `+B` may also occur in that state.

```
+A -A +B -B => (3 4 6)
+A -A +B -B +C => (3 4 7)
+A -A +B -B +C -C => (3 4 8)
+A -A +B -B +C -C +D => 9
+A -A +B -B +C -C +D -D => 10

+A -A +B -B => (3 4 6)
+A -A +B -B +C => (3 4 7)
+A -A +B -B +C -C => (3 4 8)
+A -A +B -B +C -C +B => (3 4 5 8)
```

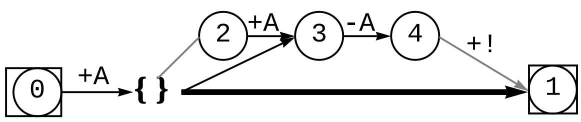
As a final illustration, here is the sequence leading up to **(3 4 7 5)** and the three permitted transitions for that state.

```
+A -A +B +B => (3 4 5 5)
+A -A +B +B -B => (3 4 6 5)
+A -A +B +B -B +C => (3 4 7 5)
+A -A +B +B -B +C +B => (3 4 5 7 5)
+A -A +B +B -B +C -C => (3 4 8 5)
+A -A +B +B -B +C -B => (3 4 7 6)
```

4.2.2 Multiple concurrent construction

New states zero and one are added and the numbers of all the existing states are incremented by 2. The new state zero has transitions constructed from the transitions of the old state zero by copying the operation names and predicates but replacing the destination with a multiple state on which the exit state is state 1, the restart state is 2 (the old state 0) and the active state is the destination of the original transition. Every exit of the old path is made an exit point for the multiple state - the predicate being attached if there is one. The new path has the new states 0 and 1 as unconditional exits.

Figure 4.10: Graph for {A}



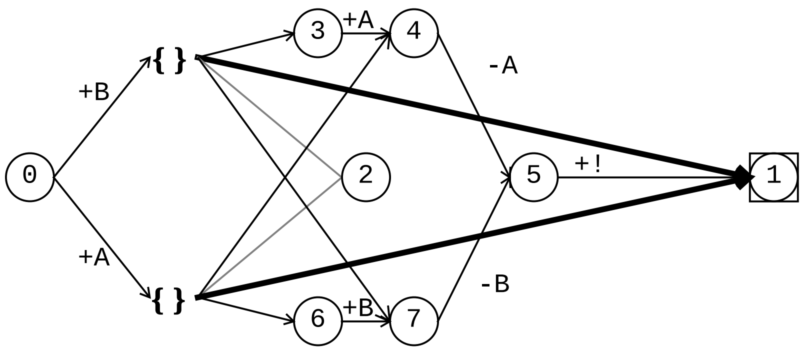
4.2.3 Modified concurrent

The product transformation shown before for the concurrent operator cannot be applied to graphs that contain the multiple state nodes created to deal with multiple concurrent paths. The multiple state mechanism can, however, be applied to provide an alternative concurrent construction to be used when required.

New states zero one and two are added and all the old states are renumbered appropriately. For each transition in each of the old states zero, the new state zero has a transition with the same operation name and predicate. The destination for the new transition is a multiple state with the new state one as exit, the new state two as restart and two active states. One of the active states is the original destination of the transition being copied, the other is the state that was state zero of the other XFSM. The new state one is an exit, if both old states zero were exits then the new state zero is also an exit with the old predicates ‘and’ed.

Although it would not normally be used for this simple case, the construction for `A,B` would be:

Figure 4.11: Graph for multiple state form of A,B



Note that state 2 is a dead end and is used in this case for the restart state in order to avoid having to introduce another kind of node.

4.3 Reduction

There are a number of simplifications that can be applied to the graphs. These simplifications serve two purposes. The more important is to make an automaton driven by the graph deterministic by eliminating states with more than one transition with any given operation name. The other purpose is to reduce the size of the graph so that the construction functions have less work to do.

4.3.1 Removing duplicated states

States can be merged if they are equivalent but not the same. Two states are equivalent if they are the same or if they have equivalent sets of transitions. Two sets of transitions are equivalent if for each transition in each set there is an equivalent transition in the other. Two transitions are equivalent if they have the same operation name, equivalent predicates and lead to equivalent states.

There may be equivalent cycles in the graph and it is important that these be detected and resolved. An example of such a cycle is given below.

Note that if a merge has occurred then the test can be repeated since there may now be a new match.

4.3.2 Removing unused states

Start with a list of all the state numbers, go through states in turn crossing out states reached from here. Any states left are unused.

Note: This algorithm may leave unreachable cycles but this just costs storage, the control mechanism still does the right thing.

4.3.3 Removing ambiguities

This is the reduction that transforms non-deterministic automata into deterministic automata.

The XFSM is searched for a state that has more than one transition with the same operation name. If the transitions do not have predicates then this ambiguity can be eliminated by creating a new state as the destination. The transitions from this new state are the union of the transitions of the states which were the destinations of the ambiguous transitions. This transformation is illustrated by an example below.

Ambiguous transitions with predicates are for future study.

4.3.4 Multiple states

Transitions to multiple states are equivalent if the restart states are equivalent, the exit states are equivalent and the bags of active states are equivalent.

Other reductions involving multiple states are for further study.

4.4 Examples

This section demonstrates the construction of the FSMs for some examples. This shows how the individual steps described above combine in practice.

Note: Additional examples are needed to show why the combination rules are as they are when in the obvious cases there will be an immediate reduction step.

4.4.1 sequence(*:choice(a b) *:choice(a b))

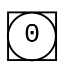
This example contains a redundant element, it is equivalent to

```
sequence( *:choice(a b) )
```

The major purpose of this example is to show how the XFSM is reduced to the same form as for the simpler expression.

Firstly a sequence identity XFSM is created to which the sequence items will be prefixed

Figure 4.12: Step 1 - sequence identity

Identity for sequence 

This has no duplicated states, unused states or ambiguous transitions and so cannot be reduced.

A choice identity XFSM is created as the initial value for constructing the second of the choices. An XFSM for the operation name b is created and then combined with the choice identity XFSM to make a choice between them and this is then reduced in two steps to the same XFSM as for operation name b

The XFSM for operation name a is then created, this is joined to the “choice(b)” XFSM and the resulting XFSM is reduced

The next step is to combine the XFSM for “choice(a b)” with the sequence identity to make the XFSM for “sequence(*:choice(a b))”.

Figure 4.13: Construction and reduction of cho(b)

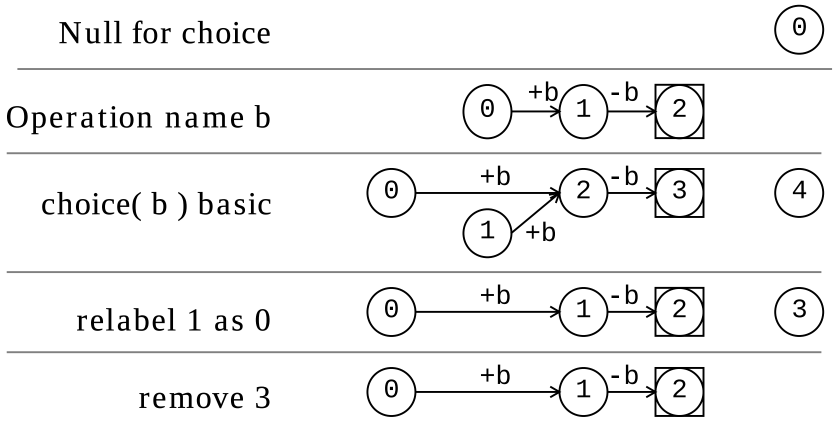
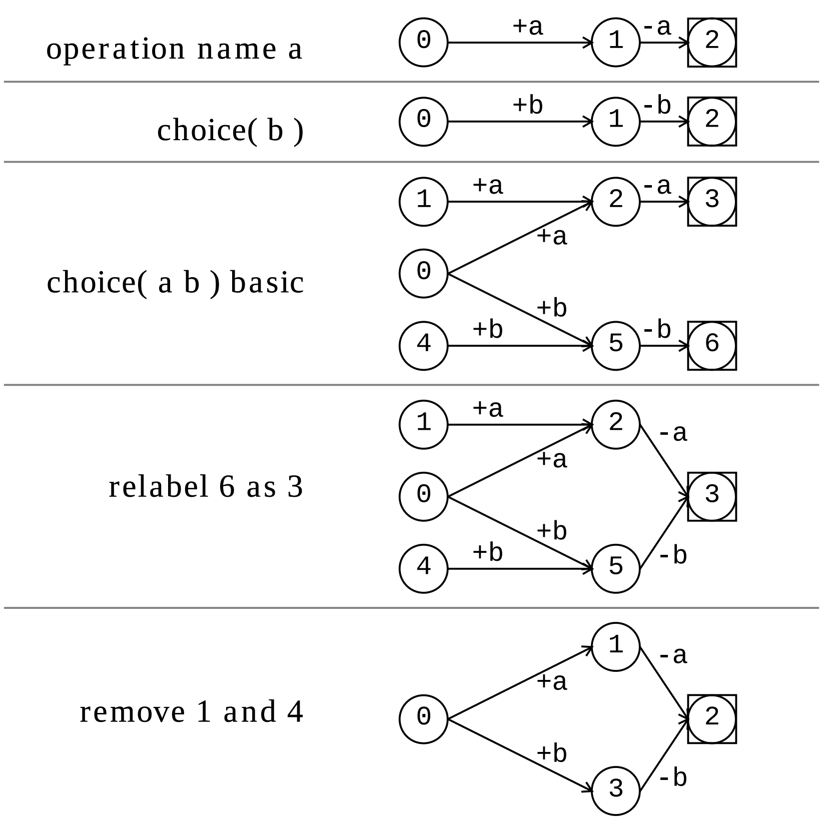


Figure 4.14: Construction and reduction of cho(a b)



Note the repeated sequence transformation applied to "choice(a b)".

Another "choice(a b)" XFSM is constructed as before, converted to a loop and prefixed to the XFSM created above and simplifications are applied as before.

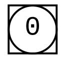
Note that the intermediate stages contain ambiguities, for example, state 0 of the basic XFSM has two "+a" transitions and two "+b" transitions.

The first reduction is to relabel state 2 as the equivalent state 0.

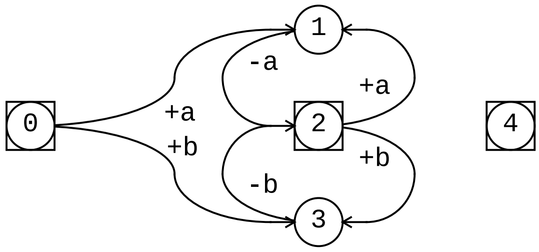
The second reduction is to relabel state 3 as state 0. Detecting that these two states are equivalent requires that the cycles through states 1 and 4 are

Figure 4.15: Construction and reduction of seq(*:cho(a b))

Null for sequence



seq(*:cho(a b)) basic



relabel 2 as 0

remove 3 (was 4)

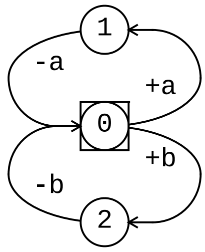


Figure 4.16: Initial form of seq(*:cho(a b) *:cho(a b))

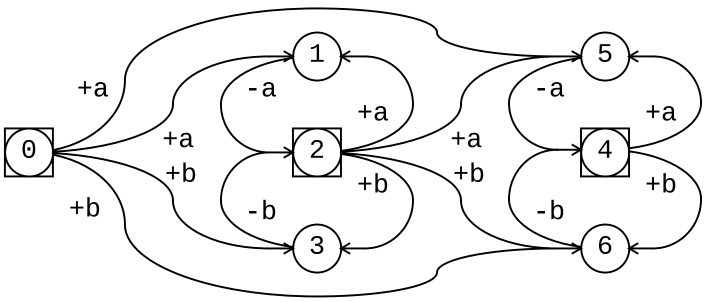
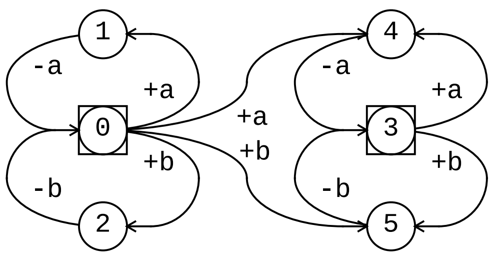


Figure 4.17: First reduction of seq(*:cho(a b) *:cho(a b))

relabel 2 as 0

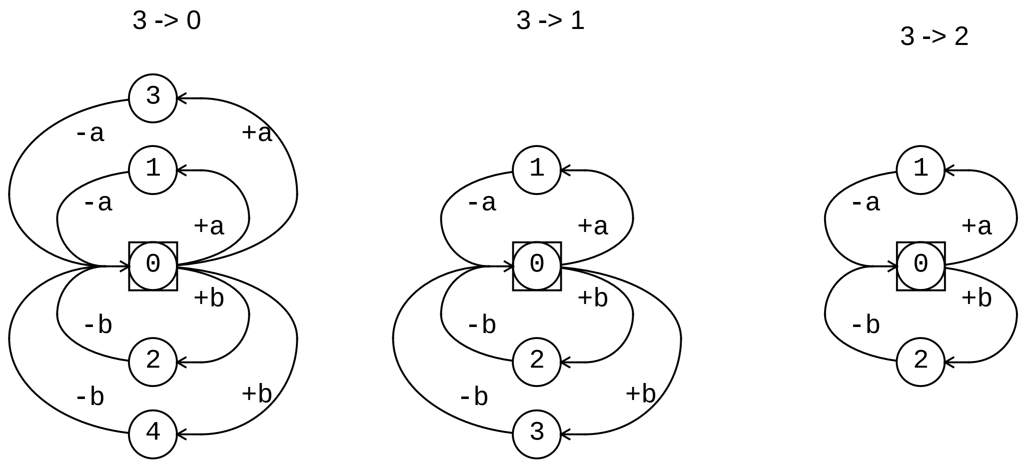


recognised as being equivalent and that the cycles through states 2 and 5 are also recognised as equivalent.

Applying the ambiguity elimination rather than equivalent state elimination would not lead to a reduction of this XFSM since the equivalent cycles would cause new ambiguous states to be generated indefinitely.

The new state 3 is then relabelled as state 1 leaving a new state 3 which can be relabelled as state 2. The final XFSM is the same as for “seq(*:cho(a b))”.

Figure 4.18: Reduction of seq(*:cho(a b) *:cho(a b)) completed



Note: The algorithm implemented in the prototype copes with this case, other more complex cases have not yet been tested.

4.4.2 choice(seq(a b c) seq(a x y))

This example is of a choice between sequences that have a common prefix.

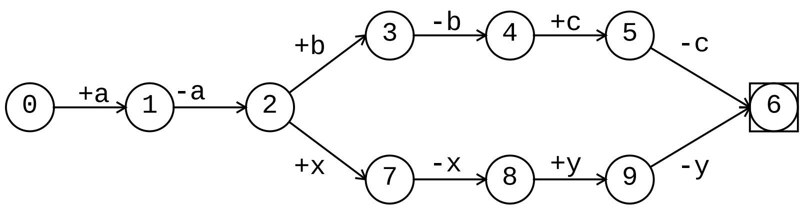
The following two expressions are equivalent and the reduction to equivalent FSMs will be described here.

```
choice( seq( a b c ) seq( a x y ) )

seq( a choice( seq( b c ) seq( x y ) ) )
```

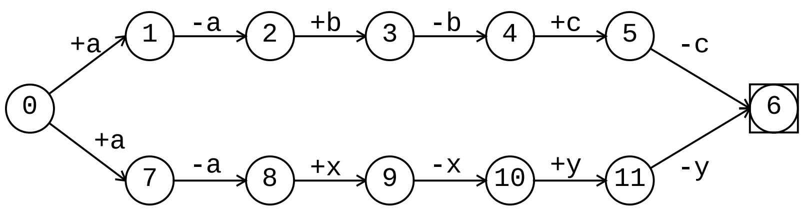
The XFSM generated for the second of these two expressions is:

Figure 4.19: seq(a choice(seq(b c) seq(x y)))



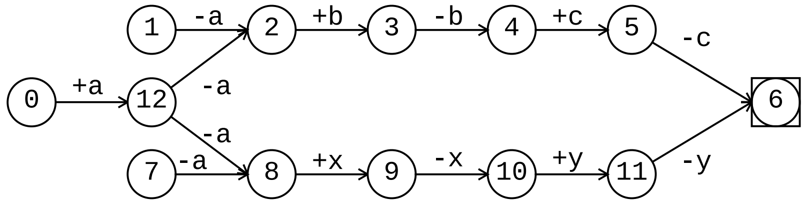
An equivalent XFSM should be generated for the first expression. Applying the construction rules and the simple reductions leads to an XFSM with an ambiguity - there are two transitions from state 0 labelled “+a”.

Figure 4.20: choice(seq(a b c) seq(a x y))



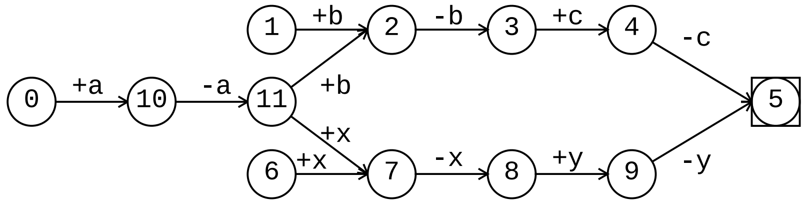
The ambiguity is factored out by creating a new state as the destination of what was the ambiguous transition. All of the transitions out of states reached from the ambiguous transition are made transitions out of the new state.

Figure 4.21: Factor out +a from state 0



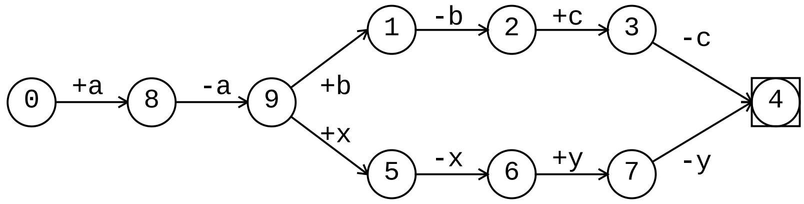
This new XFSM can be reduced as before - this will eliminate the unused states 1 and 7. The resulting XFSM still has an ambiguity to be factored out.

Figure 4.22: Factor out -a from state 10



After a removing the unused states 1 and 6, there are no more ambiguities and the XFSM is the same as for the alternative expression except for the numbering of the states.

Figure 4.23: choice(seq(a b c) seq(a x y)) reduced



It has not yet been determined whether or not the ambiguity removal mechanism will terminate. The prototype implementation imposed an arbitrary limit of two times the number of states in the initial XFSM on the number of times an ambiguity will be factored out. If there is still an ambiguity after the limit is reached then the path expression to FSM translator gives up and a “too complicated” error will be generated.

References

[ANDLER79]

Andler S, **Predicate Path Expressions**, *Proc. 6th ACM Symp. on Principles of Programming Languages pp226-236*, (Jan. 1979).

[APM.1001.01 93]

The ANSA Computational Model, *APM.1001.01*, Architecture Projects Management Ltd, Cambridge (1993).

[APM.1004.01 93]

ANSA Atomic Activity Model and Infrastructure, *APM.1004.01* Architecture Projects Management Ltd. Cambridge, (1993).

[APM.1014.01 93]

DPL Programmers' Manual, *APM.1014.01*, Architecture Projects Management Ltd. Cambridge (1993).

[CAMPBELL74]

Campbell R H, Habermann A N; **The specification of process synchronization by path expressions**, *LNCS 16 pp89-102*, Springer-Verlag (1974).

[CAMPBELL77]

Campbell R H; **Path Expressions: A technique for specifying process synchronization**, *Report UIUCDCS-R-77-863*, Dept. Comp. Sci., Univ. Illinois at Urbana-Champaign (May 1977).

[COURTOIS71]

Courtois P J, Heymans F, Parnas D L; **Concurrent control with 'readers' and 'writers'**; *CACM 14(10)667-688* (October 1971).

[FLON76]

Flon L, Habermann A N; **Towards the construction of verifiable software systems**, *Proc. ACM Conference on Data: Abstraction, Definition and Structure, SIGPlan Notices 11(Special Issue):141-148*, (March 1976)

[HABERMANN75]

Habermann A N; **Path Expressions**, *Technical Report*, Department of computer science, Carnegie-Mellon University, Pittsburgh, PA (June 1975).

[HEADINGTON85]

Headington M R, Oldehoeft A E; **Open predicate path expressions and their implementation in highly parallel computing environments**, *Proc. 1985 International Conference on Parallel Processing*, IEEE (1985).

[REED79]

Reed D P, Kanodia R K; **Synchronization with Eventcounts and Sequencers**, *CACM 22(2) 115-123* (February 1979).

[WARNE89]

Warne J P; **Extended Open Predicate Path Expressions in DPL**, *ANSA Project Report TI.72.02* Architecture Projects Management Ltd. (1989).