Programming a Bounded Buffer using the *Object* and *Path Expression* Constructs of Path Pascal

R. D. DOWSING AND R. ELLIOTT*
School of Information Systems, University of East Anglia, Norwich NR4 7TJ

We present a number of examples of the programming of a bounded buffer using the object and path expression constructs of the language Path Pascal, taking these examples as the basis for an examination of the methods of use of path expressions as a means of specifying synchronisation requirements for concurrent processes. We compare objects and path expressions with monitors and conditions, and consider the derivation of path expressions and of informal correctness demonstrations for them. Also considered is the strategy of 'distributing' the synchronisation requirements for a structured object to its components, and its effects on concurrency of access to the object, on its indeterminacy of behaviour, and on implementation costs.

Received February 1985

1. INTRODUCTION

It may plausibly be claimed that the central questions in the design of multi-process programs and of multiprogramming languages turn on the problem of disciplined synchronisation and resource sharing among a set of asynchronous concurrent processes. One of the simplest and most commonly considered multiprogramming tasks is that of programming a bounded buffer for use as a means of communication between distinct 'producer' and 'consumer' processes executing concurrently. In the following we present and discuss a number of possible ways of programming such a buffer, using for the most part the *object* and associated *path expression* constructs of the multiprogramming language Path Pascal,[1,3] but using also, for the purposes of comparison, the 'monitor' construct and the associated 'condition' variables described by Hoare.[2]

2. PATH PASCAL

Path Pascal[1,3] is a multiprogramming language based on Pascal but with extensions for the definition and use of *processes* and *objects*. Processes may be declared, like procedures and functions, as program units; an individual instance of a process thus declared may then be both created and initiated by a statement which is syntactically indistinguishable from a procedure call. Processes created by these means run in parallel with each other and with the main program itself. A Path Pascal object is comparable to the *monitor* described by Hoare[2] in that it is an encapsulated shared data structure with an associated set of *operations* (*entry* routines in Path Pascal) which provide the only permissible means of access to the structure by the surrounding processes. However, whereas synchronisation between operations on a monitor is achieved by a general mutual exclusion discipline together with primitive *signal* and *wait* operations on special *condition* variables, in Path Pascal all synchronisation constraints on an object's operations are specified by the so-called *path expression* associated with the object, which mentions the operations by name but which is textually separate from the code defining them. A path expression can express constraints on, *inter alia*, the order in which operations are performed and the degree of concurrency among operation performances. The following section describes path expressions in greater detail, but we summarise here the distinctive features of the method of specifying synchronisation constraints with path expressions.

(1) It emphasises in the program code a separation between the functionality of an operation and the synchronisation constraints on that operation.

(2) It permits the definition of synchronisation disciplines which, in some respects, are more liberal than those permitted by the monitor: in particular, it is possible for several operation performances by processes on a single object to proceed concurrently.

(3) It provides a natural means of expressing synchronisation constraints arising from the need to maintain the integrity of shared resources and data.

In Path Pascal the object is a category of type, like the array or record, and thus object variables may be created statically using ordinary variable declarations, or dynamically, using the *new* allocation routine. For information on Path Pascal not covered in this or in the following section, the interested reader is referred to Ref. 1 or 3.

3. PATH EXPRESSIONS

We present in this section an informal outline of the syntax and semantics of *path declarations* in Path Pascal, which we refer to throughout as *path expressions*. Our treatment is somewhat novel, being based throughout on the (intuitively appealing but formally undefined) concept of *operations* associated with an object, whereas the semantics given for path expressions in Refs. 1 and 3 are defined operationally, in terms of semaphores. We restrict the discussion here to those features of path expressions of which use is made in the sequel.

The *basic operations* of an object are its entry routines, that is its externally accessible procedures, functions, and processes. *Composite operations* of two kinds, *selective* and *sequential*, may be specified for an object, defined in terms of its basic operations and of other composite operations associated with it. Given operations, basic or

* To whom correspondence should be addressed.

composite, $p_1, p_2, \ldots, p_m$, associated with the object, then the construct

$$p_1, p_2, \ldots, p_m$$

specifies a *selective* composite operation associated with the object, while the construct

$$p_1; p_2; \ldots; p_m$$

specifies a *sequential* composite operation associated with the object. Thus all composite operations are ultimately constructed from the object's basic operations. In complex cases. ';' (regarded as an operator) binds more tightly than ',' (similarly regarded): parentheses may be used in the usual way to override these priorities, or may be introduced simply to make them explicit. Thus, for example, the construct

$$p_1; p_2, p_3$$

specifies a selective operation with a sequential component, namely

$$(p_1; p_2), p_3$$

rather than specifying a sequential operation with a selective component, for the specification of which the use of parentheses would be mandatory:

$$p_1; (p_2, p_3).$$

Any operation p, basic or composite, possibly itself a component of some composite operation, may have specified a *concurrency constraint* associated with it, indicated by a bracketing of the form

$$K:(p),$$

where K is some positive integer constant.

A path expression consists of an operation specification, possibly including embedded concurrency constraint specifications, enclosed in the brackets

path ... **end.**

Path Pascal requires each object type specification to include a path expression, which is required to mention each of the object's basic operations (entry routines) at least once.

The semantics of path expressions are based on the notion of a *performance* of a composite operation. For a selective operation

$$p_1, p_2, \ldots, p_m$$

each performance of any component operation p_i ($i = 1, 2, \ldots, m$) constitutes a single performance of the whole composite operation. A performance of the sequential operation

$$p_1; p_2; \ldots; p_m$$

consists of a complete performance of p_1, followed by a *corresponding* complete performance of p_2, and so on, the whole sequential operation performance being completed with the completion of the corresponding performance of p_m. In this context, for any positive integer n, the nth performances – according to the temporal ordering from the time of the object's creation – of each of the component operations $p_1, p_2, \ldots, p_m$ are considered to *correspond* to each other (but to no other performance of any of these operations).

There are two kinds of synchronisation constraint which an object's path expression may impose on the performance by processes of its operations; as follows,

(1) *Sequencing Constraints.* If operations p_i, p_j are immediately successive component operations of some sequential composite operation specified in the path expression, that is, if the path expression includes the fragment

$$\ldots p_i; p_j \ldots,$$

then each performance of p_j is constrained to start *after* completion of the corresponding performance of p_i. (In other words, each performance of an operation which is a component of some sequential operation can only occur as a legitimate part of some performance of that sequential operation. Thus constraints of this kind may be regarded as being implicit in the notion of a sequential composite operation.)

(2) *Concurrency Constraints.* If the path expression includes the specification of an operation p which has an associated concurrency constraint specification

$$K:(p),$$

Then at most K performances of the operation p may be simultaneously in progress at any given moment.

If the initiation of an attempted performance of one of the object's basic operations would give rise to the violation of a constraint of either of the above kinds – whether the constraint in question applies directly to the basic operation itself or to some composite operation of which it is a component – then the initiation of the attempted performance is delayed until the avoidance of all such violations can be guaranteed.

3. VERSIONS OF THE BOUNDED BUFFER

We first present the different versions of the program code for the bounded buffer without comment, after which we discuss their design and behaviour.

The program fragments which follow are all assumed to be located in an environment in which the following are defined:

- a positive integer constant N, representing the number of 'slots' in the buffer;
- a type *element*, taken to be the type of the individual data passing through the buffer;
- a subrange type *range* with the definition:

$$range = 1..N.$$

Each version defines a type called *bounded_buffer*. Version 1 is coded in a Pascal-like language with monitors and conditions; all the others are coded in Path Pascal.

Version 1 (after Hoare[2])

```
bounded_buffer =
  monitor
  var
    buf: array [range] of element;
    inp, outp: range; nn: 0..N;
    nonfull, nonempty: condition;
  procedure append (x: element);
    begin
    if nn = N then nonfull.wait;
    buf [inp]:= x;
    inp:= inp mod N+1; nn:= nn+1;
    nonempty.send
    end;
```

```
  procedure remove (var x: element);
    begin
    if nn = 0 then nonempty.wait;
    x:= buf [outp];
    outp:= outp mod N+1; nn:= nn-1;
    nonfull.send
    end;
  begin {initialisation}
  inp:= 1; outp:= 1; nn:= 0
  end
```

Version 2

```
bounded-buffer =
  object
  path N: (append; remove) end;
  var
    buf: array [range] of element;
    inp, outp: range;
  entry procedure append (x:element);
    begin
    buf [inp]:= x;
    inp:= inp mod N+1
    end;
  entry procedure remove (var x:element);
    begin
    x:= buf [outp];
    outp:= outp mod N+1
    end;
  init;
    begin inp:= 1; outp:= 1 end
  end
```

Version 3 (after Kolstad & Campbell[1, 3])

– as Version 2, but with the path expression ('**path** ... **end**') replaced with:

```
  path N: (1: (append); 1: (remove)) end
```

Version 4

We make use of an auxiliary type *shared_index*, with the definition:

```
shared-index =
  object
  path 1: (next_ind) end;
  var
    i: range;
  entry function next_ind: range;
    begin
    next_ind:= i;
    i:= i mod N+1
    end;
  init;
    begin i:= 1 end
  end
```

The main definition is then as follows:

```
bounded_buffer =
  object
  path N: (append; remove) end;
  var
    buf: array [range] of element;
    inp, outp: shared_index;
  entry procedure append (x: element);
    begin
    buf [inp.next_ind]:= x
    end;
  entry procedure remove (var x: element);
    begin
    x:= buf [outp.next_ind]
    end;
  {no explicit initialisation required in this version}
end
```

Version 5

We use auxiliary types *shared_index*. as defined for Version 4, and *single_slot*, with the definition:

```
single_slot =
  object
  path 1: (store; fetch) end;
  var
    value: element;
  entry procedure store (x: element);
    begin
    value:= x
    end;
  entry procedure fetch (var x: element);
    begin
    x:= value
    end;
  end
```

The main definition is then as follows:

```
bounded_buffer =
  object
  path append, remove end;
    {i.e no restriction at this level}
  var
    buf: array [range] of single-slot;
    inp, outp: shared_index;
  entry procedure append (x: element);
    begin
    buf [inp.next_ind].store (x)
    end;
  entry procedure remove (var x: element);
    begin
    buf [outp.next_ind].fetch (x)
    end;
  end
```

4. DISCUSSION OF DIFFERENT VERSIONS OF THE BOUNDED BUFFER

Version 1

Version 1, using the monitor construct, is in essence the same as that presented by Hoare.[2] It is noteworthy that it is not possible to give for Version 1 an 'equivalent' coding in terms of the Path Pascal object, since even in this simple case the monitor regime, involving relaxation of the mutual exclusion rule for processes waiting on condition variables, gives rise to a synchronisation discipline whose dynamic complexity is too great for it to be expressed in terms of path expressions.

Version 2

The most straightforward Path Pascal coding for the buffer, suitable for use in an environment in which there

is a single producer and a single consumer process, and which may be considered to be as close as Path Pascal allows to the previous version, is given as Version 2. Comparison of these first two versions highlights the two most striking differences between programming with the object/path expression constructs and programming with the monitor/condition constructs.

(1) The use of the object/path expression constructs admits the possibility of both producer and consumer processes being simultaneously active within the buffer object: an *append* operation accessing one slot of the buffer and a *remove* operation accessing a different slot are allowed to proceed concurrently with one another, with neither imposing any delay on the other. The performance of a buffer operation by a process is only delayed when this is required for the maintenance of the buffer's overall functional integrity, that is, for the producer when the buffer is full or for the consumer when it is empty. The mutual exclusion rule for a monitor on the other hand means that even when they are accessing separate buffer slots, an *append* and a *remove* operation must be forced to proceed in strict, temporally non-overlaping sequence, achieved by delaying one or other operation, if necessary.

(2) The use of an object with a path expression effectively ensures the maintenance in the program code of a clear distinction between, on the one hand, the functionality of the object's operations, that is the specific manipulations they perform on the associated data, and on the other hand the synchronisation discipline imposed on these operations. Using these constructs, each operation may be coded exactly as if it were located in a single-process von Neumann program, that is with reference solely to the required data manipulations; the necessary synchronisation restrictions are then expressed separately in the object's path expression. Use of the monitor construct, however, requires not only the declaration of special 'condition' data and possibly other control data (such as *nn*, the count of currently filled slots in Version 1), but also that code performing appropriate manipulations on these extra data be embedded at appropriate points in the code of the monitor operations. (The other side of this coin – which we mention, but do not discuss further – is that the monitor construct allows the expression of more complex, dynamically varying synchronisation requirements in the code of its operations: in Path Pascal such requirements can only be expressed by the declaration – internally to the object under consideration – of further objects with suitable path expressions.)

The path expression of Version 2 may be derived informally by considering the most basic requirements for the buffer's functional integrity in the context of an abstract view both of its internal structure and of its external relationships. That is, we characterise its internal structure simply by saying that the buffer has a fixed number of slots (N), each capable of holding a single value, all such values being of the same type (*element*), while we characterise its external relationships simply by saying that it is accessible by a single producer and a single consumer, of which the former may *append* to it a sequence of values of the appropriate type, while the latter may *remove* from it a similarly characterised sequence. Then the most basic requirement for the buffer's functional integrity is that precisely the same values should eventually be *removed*, as have, earlier, been *appended*. (Strictly, we should say 'the same instances of values' rather than just 'the same values', to cover the case where a particular value occurs several times.) Thus, ignoring both the orderings of the input and output value sequences, and also the details of the buffer's internal workings, such as the strategy for the placement of individual values in particular slots, we can immediately identify two essential constraints on the history of the buffer's interactions with the processes outside it.

(1) Avoidance of buffer 'underflow': a value can never properly be *removed* without having previously been *appended*.

(2) Avoidance of buffer 'overflow': at any given time the buffer can hold at most N values, that is, no more than N values can properly have been *appended* without also having been *removed*.

Spelling this out more explicitly in terms of the synchronisation of *remove* and *append* operations, we have the following requirements:

(1) Each *remove* operation can only be initiated after completion of a 'corresponding' *append* operation.

(2) The number of initiated *append* operations (taken over the lifetime of the buffer to date) must never exceed by more than N the number of completed *remove* operations.

The sequencing constraint expressed by the fragment '*append*; *remove*' of the path expression in Version 2 is evidently equivalent to requirement (1), while the enclosing resource constraint 'N: (...)' is evidently equivalent to requirement (2).

The preceding discussion shows how consideration of the buffer's behaviour requirements even in a very abstract form may suggest the appropriate form for the associated path expression. Strictly, however, this discussion shows only that the path expression in Version 2 expresses conditions *necessary* for the buffer's correct functioning, but we have not yet shown that these conditions are also *sufficient* for this purpose. To do this, it is sufficient to demonstrate that the values *appended* and those *removed* correspond, not merely as unordered *sets*, but also as (temporally ordered) *sequences*. To be more specific, it is sufficient to show that, for any positive integer n, the nth value *appended* to the buffer is identical to the nth value *removed* from it. To do this, the internal workings of the buffer must be considered in greater detail than hitherto: In particular, it is necessary to show that the buffer is used *cyclically* by both producer and consumer, that is, that both the nth *append* and the nth *remove* operation act exclusively on the $(((n-1)\ \mathbf{mod}\ N)+1)$th slot of the internal array *buf*. It is important at this stage to note that we can speak unambiguously here of the nth values *appended* and *removed*, since we have assumed that there is only one producer and only one consumer, and hence at most one performance of each operation can possibly be in progress at any given moment. On the basis of this, and of the further observation that the index *inp* is (after initialisation) manipulated only by the *append* operation, and from a conventional analysis of the code of that operation (and of the initialisation routine) it can be seen fairly easily, by a simple induction, that the nth value *appended* is indeed deposited in the $(((n-1)\ \mathbf{mod}\ N)+1)$th slot, as required. A similar argument with regard to the

index *outp* and the *remove* operation establishes the corresponding result for this latter operation. Having thus shown that the *n*th append and the *n*th remove both operate on the same slot of the buffer, we are in a position to show that the path expression associated with the buffer guarantees that these operations stand in the necessary relationship to each other in the context of the buffer's overall history. Specifically, it is necessary to show that both the following are guaranteed to hold.

(1) The *n*th *append* is necessarily completed before initiation of the *n*th *remove* is permitted.

(2) Given (1), no other operation on the $((n-1)\ \mathbf{mod}\ N)+1$)th slot of the buffer can intervene between the performance of the *n*th *append* and the performance of the *n*th *remove*: that is, the *n*th *remove* is necessarily completed before the initiation of the next *append* – i.e. the $(n+N)$th *append* overall – which operates on this slot.

These two constraints are precisely those defined by the sequencing and concurrency constraints in the path expression of Version 2: '*append*; *remove*' (1), and '*N*: (...)' (2). Thus we complete the informal demonstration of the adequacy of Version 2 for the buffer's functional integrity.

We showed earlier that the constraints imposed by the path expression are certainly necessary to the buffer's functional integrity as well as sufficient, and thus we have effectively shown also that the buffer in Version 2 is *efficient*, in the sense that it imposes no unnecessary delay on the processes using it. This in turn implies that it is deadlock-free, but we pursue such issues no further.

We have thus seen that we can only establish the integrity of operation of the buffer in Version 2 – that is establish that values are neither lost nor spuriously generated within it – by means of a detailed consideration of the possible history of performances of its operations, by which means, we may note in passing, we establish also the correspondence between the order in which values are *append*ed and that in which they are *remove*d. We observed earlier that the use of the object/path expression constructs promotes a reflection in program code of a separation of concerns, namely the concerns, on the one hand, of the functionality of operations and, on the other, of their synchronisation. However, our detailed consideration of the process of establishing correctness in this – very simple – case of the cyclic buffer indicates that this separation is at best a mixed blessing, since the required correctness can be established only by consideration of the concerns of functionality and synchronisation, not separately, but explicitly in conjunction with one another. Indeed, it might be argued that, not only from the point of view of proving correctness, but even from that of intuitive understanding, the monitor/signal version, where the count of values currently held in the buffer – referred to as *nn* in Version 1 – appears as an explicitly declared variable, explicitly manipulated in the code of the buffer operations, is actually preferable to the object/path expression version, where such data are represented in the semaphores – or some other suitable mechanism – used to implement the path expression, but which are hidden from the programmer's view.

Version 3

If the bounded buffer is located in an environment in which either the number of (concurrently executing) producer processes, or the number of consumer processes, is greater than 1, then the buffer of Version 2 is inadequate: as we have seen, the correctness of Version 2 depends on the assumption that there is a single producer, and a single consumer, and hence that at most 1 *append* operation and at most 1 *remove* operation on the buffer are in progress at any given time. If however we explicitly impose constraints on the buffer operations *append* and *remove*, ensuring that at any given time at most one producer and at most one consumer can be engaged in each operation, then we re-establish the only one of the assumptions for Version 2 which is violated by the introduction of additional producers and consumers. Thus we see that, for an environment in which there may be several producers and several consumers, a satisfactory coding for the buffer is given by Version 3, where mutual exclusion and hence strict sequencing are imposed on performances of the *append* operation, and likewise on performances of the *remove* operation, by applying to each the concurrency constraint:

1:(...).

Version 4

For the more complex environment in which the numbers of producers and of consumers exceed one, there is a possible alternative approach to that, represented by Version 3, of minimal refinement of the single producer/single consumer version. This alternative approach involves the adoption of what might be considered a more 'fundamentalist' attitude to the problem of data protection in a multi-process environment. The principle embodied in this attitude is that any variable located in an environment from which it is potentially simultaneously accessible by more than one process should be explicitly protected by making it into an object with a suitable path expression to guarantee its functional integrity. In the case of the bounded buffer, the particular circumstance which renders Version 2 inadequate in the multi-producer, multi-consumer environment is the possibility of concurrent accesses to – and hence possible corruption of – the index variables *inp* and *outp*. Rather than avoid this danger by imposing synchronisation restrictions on the operations which access these variables, as in Version 3, we may choose instead to make the variables themselves into objects, whose path expressions constrain all accessing operations on them to occur in strict linear sequence. Adopting this approach leads us to Version 4. We should note however that in terms of the interface between the buffer and its environment, the semantics of the buffer of Version 4 differ from those of the buffer of Version 3. In Version 4 there is no ordering imposed by the buffer at the outer level on the performance by processes of *append* operations, or similarly on the performance of *remove* operations. Thus, while with Version 3 – where we still have these orderings – we can go on to say that the sequences of values determined in each case are the same, with Version 4 – where the existence of (total) orderings is not guaranteed – it is not necessarily meaningful to talk of any correspondence between them. Indeed, even

in the case where, for example, the actual pattern of use of the buffer by the processes surrounding it gives rise to strict orderings on the initiations of *append* and on the completions of *remove* operations, there is nevertheless no guarantee that the sequences of values determined by these orderings will be equivalent. It is perfectly possible that the relative speeds of – say – a pair of *append* operations within the buffer may be such that the one initiated first actually gains access to the inner object *inp* later than the other. It is also perfectly possible that the order in which *append* operations are initiated should differ from that of their completion. Thus we see that, relative to Version 3. Version 4 permits a greater degree of concurrency of activity within the buffer, at the expense of a (possibly harmless) degree of indeterminacy in the ordering of values as they pass through it.

Version 5

With the previous version, we introduced the approach of guaranteeing the integrity of operation of a structured object by ensuring that we have suitable guarantees of integrity for each of its components. We may push this approach a stage further than in the previous version by applying it to the individual elements of the array *buf.* each of which may be regarded as a single-slot buffer whose integrity of operation is fairly obviously characterised adequately by the path expression

path 1: (*store*; *fetch*) **end**

Thus we arrive at the buffer coding of Version 5, where there is a 'null' path expression – that is, no synchronisation constraint – at the outer level, and where the integrity of operation of the whole depends entirely on that expressed by the path expressions associated with the component indices and individual buffer slots of the larger object. Here of course, the degree of concurrency, and the associated degree of indeterminacy of ordering, of performances of the buffer operations are both greatly increased with respect even to Version 4: ordering the *append*s according to the times of their initiation, and assuming further that this ordering is equivalent to that of their gaining access to the index *inp*, it is possible – if highly unlikely in most implementations – that the *n*th *append* should gain access to the $(((n-1)\ \mathbf{mod}\ N)+1)$th slot *after* the $(n+N)$th *append*, which also uses that slot. Whether such increases in the degrees of concurrency and of indeterminacy are matter of any consequence or not is dependent on the buffer's intended context of use, rather than on any consideration of its own integrity.

In Version 3 we have a data structure for which the accessing operations are subject to a relatively complex set of functional integrity constraints, expressed by a correspondingly complex path expression: in Version 5 these complex constraints have effectively been weakened somewhat and 'distributed' to the atomic components of the data structure, by encapsulating each component as an object in its own right, with its own limited set of accessing operations, constrained by a comparatively simple path expression, which effectively expresses the 'local' functional integrity requirements. Although in fact we have thus developed Version 5 from Version 3 here, it is possible to regard the former as being the more primitive of the two, developed in two layers 'top-down' on the basis of the overall functional requirements: Version 3 might then be regarded as a refinement or optimisation of Version 5, motivated either by the addition of further overall requirements, such as a requirement to establish congruent orderings on the input and output sequences, or by the desire to reduce the overheads associated with the synchronisation, which are considered in the following section.

5. IMPLEMENTATION OF SYNCHRONISATION

The implementation of the synchronisation constraints expressed by an object's path expression may be achieved by attaching prologue and epilogue sequences of semaphore operations to the code for the basic operations (entry routines) of the object itself, as described in detail in Ref. 3. For example, the path expression of Version 3:

path *N*: (1: (*append*); 1: (*remove*)) **end**

can be implemented by attaching semaphore operations to the code for the entry procedures *append* and *remove* as follows:

$P(S3)$	$P(S4)$
$P(S2)$	$P(S1)$
body of *append*	body of *remove*
$V(S1)$	$V(S2)$
$V(S3)$	$V(S4)$

where in this case the semaphores and their initial values are

$S1$ initially	0	– the 'underflow' semaphore;
$S2$ initially	N	– the 'overflow' semaphore;
$S3$ initially	1	– the 'mutual exclusion' semaphore for *append*;
$S4$ initially	1	– the 'mutual exclusion' semaphore for *remove*.

Thus a measure of the overheads incurred by the use of a particular object or path expression is given by the number of semaphores required to implement it. We give in Table 1 below this measure for each of the Path Pascal versions of the bounded buffer, which shows clearly the profligacy in this respect of Version 5 for any but the smallest values of N.

Table 1. No. of semaphores required for bounded buffer

Version	1	2	3	4	5
No. of semaphores	—	2	4	4	$2*N+2$

6. CONCLUSIONS

From the various versions of the bounded buffer object a number of points emerge. Firstly, the object/path expression discipline presents two advantages over the monitor/signal discipline: path expressions often have a natural and intuitively appealing derivation from simple considerations of functional integrity, and their use enforces a clear reflection in program code of the separation of the concerns of synchronisation and of functionality for an object's operations. However, this latter may possibly also be regarded as something of a disadvantage, since in the consideration of program correctness these concerns are often intimately linked

and must then necessarily be considered in conjunction rather than separately.

The discussion of the later versions shows that there is a spectrum of possible approaches to the programming of a shared data structure. At one end of this spectrum, all synchronisation constraints are expressed at the outermost level, with the access to the individual components programmed conventionally: at the other end of the spectrum, the specification of synchronisation constraints is distributed to the individual components, which are then themselves treated as sharable objects with controlled access. The earlier discussion shows that, as we move across this spectrum, there is an increase in the degree of concurrency of access at the outermost level (but it should be appreciated that this will not normally imply any significant increase in overall performance levels), and an increase also in the degree of indeterminacy of externally perceived behaviour (to a level which in some contexts might be unacceptably anarchic), and an increase finally in the implementation overheads associated with the synchronisation requirements.

Finally, the variety of possible approaches to what is, from an intuitive point of view, a rather straightforward problem, may be taken as bearing witness to the difficulty of establishing a simple set of general design principles for multiprogramming problems.

Acknowledgements

One of the authors (R. Elliott) acknowledges with gratitude the support of a studentship awarded by the SERC (UK). The treatment of path expressions in Path Pascal is developed from work undertaken under an earlier research grant, also awarded by the SERC.

REFERENCES

1. R. B. Kolstad & R. H. Campbell, 'Path Pascal user manual,' *ACM. Sigplan Notices* **15** (9), 15–24 (1980).
2. C. A. R. Hoare, 'Monitors: an operating system structuring concept', *CACM* **17**, 549–557 (1974).
3. R. B. Kolstad and R. H. Campbell, 'Path Pascal user manual', Department of Computer Science, University of Illinois at Champaign–Urbana (1980).