Constraining Interference in an Object-Based Design Method

C. B. Jones

Department of Computer Science
Manchester University, M13 9PL, UK
cbj@cs.man.ac.uk

Abstract. This paper is the first of a series which are intended to contribute to tractable development methods for concurrent programs by exploring ways in which object-based language concepts can be used to provide a compositional development method for concurrent programs. The property of a (formal) development method which gives the development process the potential for productivity is compositionality; interference is what makes it difficult to find compositional development methods for concurrent systems. This paper shows how object-based concepts can be used to provide a designer with control over interference; it also proposes a transformational style of development (for systems with limited interference) in which concurrency is introduced only in the final stages of design. The essential idea here is to show that certain object graphs limit interference.

A companion paper discusses the problems of interference more fully and shows how a suitable logic can be used to reason about those systems where interference plays an essential role. There again, concepts are used in the design notation which are taken from object-oriented languages since they offer control of granularity and way of pinpointing interference. A third paper is in preparation which defines the semantics of the object-based design notation.

1 Introduction

The most difficult aspect of finding tractable development methods for concurrent systems is to provide a useful notion of compositionality which facilitates the division of work. Compositionality can be defined as follows (adapted from [Zwi88])

> A development method is *compositional* if the fact that a design step satisfies a given specification can be justified on the basis of the specifications of any constituent components without knowledge of their construction

Earlier work on shared-variable concurrency (see [Jon83] which is significantly extended in [Stø90, Stø91a, Stø91b]) used rely and guarantee conditions both to describe and to reason about *interference*. The fixed format of these specifications was rejected in [Jon91] in favour of a logic with operators which use predicates of pairs of states (this is similar to Lamport's TLA [Lam90, Lam91]). But the proofs in [Jon91] remain long-winded and earlier work has been dogged by issues like atomicity (granularity) and questions about where invariants etc. are supposed to hold.

In common with many others, the current author sees language restrictions as a way of constraining concurrency; in particular, the aim here is to reduce the number of proof

obligations in development. The current approach uses concepts of object-oriented languages in order to constrain interference and fix a level of granularity.[1] It is not, however, the aim to add yet one more language to those claiming to be object-oriented; the development method envisaged here ought to be usable for programs in languages such as ABCL [Yon90], Modula-3 [Nel91], Beta [KMMN91] or UFO [Sar92]. The claim is that some carefully chosen subset of object-oriented concepts makes the design of concurrent programs more tractable than in arbitrary shared-variable languages (or even languages like CSP). The move to an object-based language has not made the interference logic redundant it has only reduced the need for interference arguments; [Jon93b] explores the situation where interference is essential.

The design notation used in this paper is heavily influenced by the programming language 'POOL' (see Section 4 for references and some comparative notes); it also reflects discussions with colleagues at Manchester University. Most of the features of the language are presented by examples. Points of interest include the following. *Classes* have *methods* only one of which may be active at any one time (for a particular instance); invocation of methods is synchronous but methods can return before they complete and this releases the invoking process from the *rendezvous*. Consider Figure 1: this can be read as an object-oriented program (which is actually developed from a specification in Section 2). The programming task which is considered concerns sorting: a priority queue delivers – and removes – its smallest value via a remove method (*rem*); new values can be added by another method (*add*). Programs obtain a reference to (an instance of) a priority queue with a new *Priq* statement. In fact, the created queue can be a linked list of instances of *Priq* but the using program would have no way of detecting this. Each instance has two variables containing a value and a link (possibly nil) to the next element.

```
Priq class
vars m: [ℕ] ← nil; l: private ref(Priq) ← nil
add(e: ℕ) method
  return
  if m = nil then (m ← e; l ← new Priq)
  elif m < e then l!add(e)
  else (l!add(m); m ← e)
  fi
rem() method r: ℕ
  return m
  if m ≠ nil then m ← l!rem()
                  if m = nil then l ← nil
                  fi
  fi
```

Fig. 1. Example $\pi o \beta \lambda$ program: *Priq*

[1] The idea to use object-oriented languages was made more tempting by the positive experience of building a theorem proving assistant [JJLM91] in Smalltalk and more recent discussions about exploiting parallel hardware and tackling a multi-user version of *mural*.

In the class *Priq*, the *new* method is implicit; all that happens when an instance is created is that the instance variables (m and l) are initialized. Once created, there are two methods which can be invoked for an instance of the *Priq* class: *add* puts its argument into the queue and *rem* – which takes no arguments – returns the smallest value contained in the queue. Methods are invoked by expressions like $l!add(7)$ (where l is a reference to an instance of *Priq*). The semantics dictates that only one method can be active at any time in a particular instance of *Priq*.[2] Notice that the return statements occur at the the beginning of the *add* and *rem* methods. This releases the user from the *rendezvous* and lets the remaining code of the method run in parallel with other activity of the invoking program. Furthermore, once – say – the call to the next *add* has been released, the method terminates and its instance is available for other method calls. One can picture a whole sequence of *add* and *rem* methods rippling along the linked-list structure. The fact that the activity can never get out of order is important and results from the object graph which is created. Marking the contained references as private makes it easier to establish results about the object graphs. Were $\pi o\beta\lambda$ a programming language, all sorts of concrete syntax details would have to be resolved – here, a rather relaxed syntax is used with line breaks playing a meaningful part. (The abstract syntax of the language used here is given in an appendix of [Jon92].) The reader should remember that $\pi o\beta\lambda$ is intended as a design notation to be used to develop programs in a language where issues like parsing have received due attention.

In addition to the return statement, there is a yield statement which provides a way of delegating the responsibility to answer a method invocation. As in Figure 1, objects (instances of classes) are created by activating new for a class name; in $\pi o\beta\lambda$ explicit methods for new can be written; the language does not offer inheritance.

In addition to the language presentation herein, it is to some extent true that the search for a development method has been driven by examples: the approach has been to find plausible development steps and then to look for formal rules which justify them. This is largely motivated by the experience which shows that the thing which makes formal development work like mathematics is finding the right steps of development; detailing the proofs of individual steps is less rewarding. One key insight was the realization that assertions (invariants etc.) about the object graphs created by object references are central to the explanation of many algorithms. This paper looks at two topologies in Sections 2 and 3; both use decompositions which are justified by rules which support a 'promotion' of properties about instances to properties about collections of instances. This can be compared with the way in which an inference rule for a while statement can be used to infer results about a composite statement from properties of its components. The need – in the case of more general (DAG-like) topologies – to cope with interference is studied in [Jon93b].

There are at least two options for giving the semantics: mapping to Milner's Polyadic π-calculus [Mil92] or a resumption semantics which fits the way methods work here (cf. [AR89, pp111]; see also [Wol88, AR92]). Since the mapping to the π-calculus is quite far advanced (see [Jon93a]), the working name for the design notation is $\pi o\beta\lambda$.

---

[2] It can be useful to think of classes as blocks which can be multiply instantiated; each instance has local (instance) variables and procedures (methods); the instance variables can only be accessed or changed by the methods; methods are called (invoked) by sending messages.

2 Linked-lists of objects

The first development example in this paper illustrates the object-based nature of the programming language and the role that this plays in developing programs. What follows is a step-wise development of a program which stores each element of a queue as a local variable in an instance of an object; these objects are organized into a linked-list. Because the specifications are simpler, the first steps of development assume sequential execution within a queue (there might – however – be other concurrent threads); concurrency within a queue is considered in the final development step where its use is justified by arguing that it provides the same visible behaviour as the sequential implementation.

Specification

As in a Larch [GHW85, GH93] 'interface language', the design notation is used here to provide a framework for the specification which is given as a class definition. The methods are specified by pre- and post-conditions in a style similar to that used in VDM [Jon90].[3] The separation of the assumptions that a developer can make into pre-conditions should be noted; this is mirrored by the separation of assumptions about interference in [Jon93b]. In post-conditions, hooked identifiers refer to the value of the instance variables before execution of the method and undecorated identifiers refer to the values after execution of the method. Thus

$$b = \overleftarrow{b} \cup \{e\}$$

requires that the value of the instance variable b after an invocation of *add* is the bag union of the value of that variable before execution of the method with a unit bag containing the value of the parameter. Notice that *rem* is a partial method and – as in VDM – the post-condition can be undefined if its pre-condition is not satisfied. (The external clauses from VDM operations are barely necessary in the context of a class but there are places where one really ought note that some variables are read-only.) Values of type bag etc. and operators like $\cup$ are part of the specification language.

Priq class
vars b: $\mathbb{N}$-bag $\leftarrow \{\}$
add(e: $\mathbb{N}$) method
  post $b = \overleftarrow{b} \cup \{e\}$
rem() method r: $\mathbb{N}$
  pre $b \neq \{\}$
  post $r = min(\overleftarrow{b}) \wedge b = \overleftarrow{b} - \{r\}$

Just as in VDM, 'satisfiability' proof obligations can be generated for each method specification.

Straightforward data reification

It is possible to represent the bag abstraction b by an ascending sequence. This step of data reification is sketched here in order to afford comparison with the reification to a linked-list which follows. The objects concerned are

[3] The classes here can be compared with modules in VDM-SL [BSI92, Daw91].

$AscSeq = \mathbb{N}^*$
inv $(b) \triangleq is\text{-}ascending(b)$

The invariant is a restriction on the elements which are in the set $AscSeq$ ($is\text{-}ascending$ – and other simple functions – are taken to be obvious).[4]

The relationship between this representation and the abstract objects is defined

$retr : AscSeq \to \mathbb{N}\text{-}Bag$
$retr(b) \triangleq bagof(b)$

$bagof : X^* \to X\text{-}Bag$
$bagof(t) \triangleq \{e \mapsto \text{card}\{i \in \text{inds } t \mid t(i) = e\} \mid e \in \text{elems } t\}$

This representation is 'adequate' (there is at least one element of $AscSeq$ which corresponds – under $retr$ – to each element of $\mathbb{N}$-bag). The methods of $Priq$ can be specified on this representation as follows.

$Priq$ class
vars b: $AscSeq \leftarrow []$
$add(e: \mathbb{N})$ method
  post $\exists i \in \text{inds } b \cdot b(i) = e \wedge del(b, i) = \overleftarrow{b}$
$rem()$ method r: $\mathbb{N}$
  pre $b \neq []$
  post $r = \text{hd } \overleftarrow{b} \wedge b = \text{tl } \overleftarrow{b}$

$del(t, i) \triangleq t(1, \ldots, i-1) \frown t(i+1, \ldots, \text{len } t)$

The correctness of such a step can be justified by further rules (operation domain/result) of [Jon90].

It is worth taking this opportunity to reflect on where the invariant must hold: a user would presumably accept an implementation of add which put new elements at the end of a list and then sorted it. Thus an invariant does not have to be true mid-operation: it is really a way of abbreviating pre-/post-conditions. It would be possible to develop a sequential implementation – using decomposition rules to justify that the use of while statements etc. – which satisfies this intermediate specification.

Reification involving class instances

The main line of object-based development is now considered (i.e. the reification to $AscSeq$ is ignored and the reference point for this step is the initial specification in terms of a bag). Here again, the first design step focuses on the development of the data structure and finding an appropriate invariant is a key issue. This development step employs multiple instances of class $Priq$; they form a linked-list with the l variable in one instance pointing to the next; the local variables (m) of the instances collectively represent the bag b. The use of references

[4] Throughout this paper, VDM notation [Jon90] is used for sequences, maps etc.

necessitates talking about a global state ($\sigma \in \Sigma$). This is viewed as a map from references to instances

$$\Sigma = Ref \xrightarrow{m} Inst$$

and variable names are applied as selectors to objects of $Inst$ (e.g. if p is a reference to an instance of $Priq$, then $m(\sigma(p))$ is a natural number). The state is a Curried argument to functions which depend on the global state. The predicate $is\text{-}linked\text{-}list(p, l)(\sigma)$ is true if the instance pointed to by p (in σ) is the start of a linked-list via the references contained in the l variables of each instance. Although the objective here is to talk about linked-lists etc. without needing to think at the reference level, this predicate can be defined in terms of Σ as follows.[5]

$$is\text{-}linked\text{-}list : Ref \times Name \to \Sigma \to \mathbb{B}$$

$$\begin{aligned}
&is\text{-}linked\text{-}list(p, l)(\sigma) \triangleq \\
&\quad \exists pl \in Ref^* \cdot \\
&\qquad pl(1) = p \wedge l(\sigma(pl(\mathsf{len}\, pl))) = \mathsf{nil} \wedge \\
&\qquad \forall i \in \{1, \ldots, \mathsf{len}\, pl - 1\} \cdot pl(i+1) = l(\sigma(pl(i)))
\end{aligned}$$

Similarly, a function to extract a sequence from a linked list is $extract\text{-}seq(p, l, n)$ which generates a sequence of the (non-nil) n values from instances linked by the l references.

$$extract\text{-}seq : Ref \times Name \times Name \to \Sigma \to X^*$$

$$\begin{aligned}
&extract\text{-}seq(p, l, n)(\sigma) \triangleq \\
&\quad \mathsf{if}\ p = \mathsf{nil}\ \mathsf{then}\ [\,] \\
&\quad \mathsf{elif}\ n(\sigma(p)) = \mathsf{nil}\ \mathsf{then}\ extract\text{-}seq(l(\sigma(p)), l, n)(\sigma) \\
&\quad \mathsf{else}\ [n(\sigma(p))] \frown extract\text{-}seq(l(\sigma(p)), l, n)(\sigma) \\
&\quad \mathsf{fi}
\end{aligned}$$

This can be used to define the set of references which can be reached from a reference.

$$reach : Ref \times Name \to \Sigma \to X^*$$

$$reach(p, l)(\sigma) \triangleq \mathsf{elems}\ extract\text{-}seq(p, l, l)(\sigma)$$

The data type invariant can then be defined as follows.

$$inv : Ref \to \Sigma \to \mathbb{B}$$

$$\begin{aligned}
&inv(p)(\sigma) \triangleq \\
&\quad is\text{-}linked\text{-}list(p, l)(\sigma) \wedge is\text{-}ascending(extract\text{-}seq(p, l, m)(\sigma)) \wedge \\
&\quad \forall r \in reach(p, l)(\sigma) \cdot l(\sigma(r)) = \mathsf{nil} \Leftrightarrow m(\sigma(r)) = \mathsf{nil}
\end{aligned}$$

[5] It would be possible to pass a lambda expression (or simply make l a constant) in order to avoid passing a name to *is-linked-list*.

The invariant is considered to be true only between method invocations (rather than during the execution of a method). The retrieve function is as follows.

$$retr : Ref \to \Sigma \to \mathbb{N}\text{-}Bag$$

$$retr(p)(\sigma) \triangleq bagof(extract\text{-}seq(p, l, m)(\sigma))$$

It is now possible to specify *Priq* on the linked-lists.[6]

Priq class
vars m: $[\mathbb{N}] \leftarrow$ nil; l: private ref($Priq$) $\leftarrow$ nil
$add(e: \mathbb{N})$ method
  post let $\overleftarrow{b} = extract\text{-}seq(\text{self}, l, m)(\overleftarrow{\sigma})$ in
    let $b = extract\text{-}seq(\text{self}, l, m)(\sigma)$ in
    $\exists i \in \text{inds } b \cdot b(i) = e \wedge del(b, i) = \overleftarrow{b}$
$rem()$ method r: $\mathbb{N}$
  pre $extract\text{-}seq(\text{self}, l, m)(\sigma) \neq []$
  post let $\overleftarrow{b} = extract\text{-}seq(\text{self}, l, m)(\overleftarrow{\sigma})$ in
    let $b = extract\text{-}seq(\text{self}, l, m)(\sigma)$ in
    $r = \text{hd } \overleftarrow{b} \wedge b = \text{tl } \overleftarrow{b}$

Any user of a *Priq* would be unaware that the implementation involved multiple instances; since the references are private (cannot be copied) they are invisible and free from danger of interference. In order to state the pre- and post-conditions, the sequences are extracted from the state with a reference to the current instance (self) providing the start of the list. A simple generalization of standard refinement rules covers such reification steps.

Operation decomposition

The next step of development is to look at code which satisfies the above specifications: they are decomposed into executable statements.

[6] Notice m can contain a VDM-like nil; for the *Ref* type, a nil value is a normal null reference; there is a sort of pun here since a 'real' object-oriented language would anyway make all values into objects.

```
Priq class
vars m: [ℕ] ← nil; l: private ref(Priq) ← nil
add(e: ℕ) method
  if m = nil then (m ← e; l ← new Priq)
  elif m < e then l!add(e)
  else (l!add(m); m ← e)
  fi
  return
rem() method r: ℕ
  t: ℕ
  t ← m
  if t ≠ nil then m ← l!rem()
                  if m = nil then l ← nil
                  fi
  fi
  return t
```

The inductive justification of this decomposition relies on rules which promote assumptions on one instance of the class to collections of such instances; the linear reference topology allows a structural induction argument about the recursive calls to methods. The base case for *add* – which starts with b as the empty sequence – is straightforward (p and l are both nil). The inductive step assumes that the recursive call to $l!add(m)$ performs according to specification. Notice that *inv* above implies that there can not be a loop in the reference chain which is important since otherwise calls to *add* would deadlock. Notice also that it is not necessary to rely on *pre-rem*: the implementation happens to deliver a nil result if the method is used outside its intended domain.

Equivalent code

As mentioned above, the initial steps of this development have not employed concurrency within a queue: in the preceding code, *add* and *rem* hold the invoking process in a *rendezvous* until they complete and a method call at the head of the list does not complete until all recursive calls terminate. (Recall that only one method can be active in each instance of a method at any one time.) Parallelism can be achieved by letting – for example – *rem* return the local m before it ripples through bringing up values as required; the invoking process is released from the *rendezvous* and its subsequent code can run in parallel with the *Priq* methods. Furthermore, this applies to instances of *Priq* within one queue: once *rem* has obtained a value from the next element in the queue, it can terminate making it possible for either of the methods of this instance to be invoked. Because of the linear reference topology controlled by private refs, no other thread of control can interfere with the queue.

The argument for the correctness of this step follows from a transformation which permits moving statements

$$S; \text{return } e \quad \rightsquigarrow \quad \text{return } e; S \tag{1}$$

providing e is not affected by S and S only changes (other than its own state) states reachable by private references. Thus the preceding code is equivalent to that in Figure 1 of Section 1.

This step uses algebraic laws to re-order code to give an observationally equivalent parallel program to the one which was first specified. Apart from offering what is hopefully

an intuitive development route, this has obviated the need to write post-conditions for the concurrent behaviour of the methods. It is not immediately obvious how to write such post-conditions because at the point at which an execution of a method begins, methods on other instances might still be active (such post-conditions appear to need something like Lamport's 'prophesy variables').

The final code behaves in much the same way as *BUBLAT* (cf. [CLW79]) did in earlier work on 'interference' proofs (e.g. [Stø90]) but there is much less 'mechanism' visible here – further steps of development could bring in the extra variables of the earlier code if so desired.

Alternatives

A couple of general observations can be made even after this simple example. There is a reliance above on the fact that the values (in $\mathbb{N}$) are immutable; while this is taken for granted in non-OO-languages, it is not the norm in the OO-world (cf. open issue 2 in Section 4). If the element values could change, such changes would need to be constrained by interference assertions like those used in [Jon93b].

It must be conceded that – thus far – it would be possible to use a development method in which objects can be guarded from interference by encapsulation and then to have a compiler generate the actual class instances. The reason for taking the approach of creating the instances and reasoning about (non-)interference is that it prepares for the more general approach in [Jon93b]. It is – for example – interesting to consider what would go wrong with the above development if a 'fast path' vector of pointers to every tenth element in the list existed. The sharing of pointers which would result would undermine the transformation shown in Equation 1 and observational equivalence would not be guaranteed. Extensions to reason about such interference would need extra variables in which counts of readers and writers could be maintained.

3 Tree-structured topologies

The programming task specified below is similar to that in the preceding section but it shows that references defining a tree-like object graph can be used as a basis for reasoning; the developed program also introduces a new statement of the language.

Specification

The example of building a simple symbol table is used in [Ame89]; its specification is very simple.

Symtab class
vars st: $(Key \xrightarrow{m} Data) \leftarrow \{\}$
$insert(k\text{: } Key,\ d\text{: } Data)$ method
  post $st = \overleftarrow{st} \dagger \{k \mapsto d\}$
$search(k\text{: } Key)$ method $res\text{: } Data$
  pre $k \in \mathrm{dom}\, st$
  post $res = st(k)$

Reification

The first design idea is to represent the map as a binary tree.

$$
\begin{array}{l}
Tree :: mk : [Key] \\
\qquad\quad md : [Data] \\
\qquad\quad l \;\; : [Tree] \\
\qquad\quad r \;\; : [Tree] \\
\mathsf{inv}\,(mk\text{-}Tree(mk, md, l, r)) \triangleq \\
\quad (mk = \mathsf{nil} \Leftrightarrow md = \mathsf{nil}) \wedge (mk = \mathsf{nil} \Rightarrow l = r = \mathsf{nil})
\end{array}
$$

Over which an invariant might be defined

$$
\begin{array}{l}
is\text{-}ordered\text{-}tree : Tree \to \mathbb{B} \\
\\
is\text{-}ordered\text{-}tree(mk\text{-}Tree(mk, md, l, r)) \triangleq \\
\quad \mathsf{if}\ mk = \mathsf{nil} \\
\quad \mathsf{then\ true} \\
\quad \mathsf{else}\ (\forall lk \in coll(l) \cdot lk < mk) \wedge (\forall rk \in coll(r) \cdot mk < rk) \wedge \\
\qquad (l \neq \mathsf{nil} \Rightarrow is\text{-}ordered\text{-}tree(l)) \wedge (r \neq \mathsf{nil} \Rightarrow is\text{-}ordered\text{-}tree(r)) \\
\quad \mathsf{fi}
\end{array}
$$

where the *coll* function simply collects the set of *Keys*

$$
\begin{array}{l}
coll : [Tree] \to Key\text{-}\mathsf{set} \\
\\
coll(t) \triangleq \\
\quad \mathsf{cases}\ t\ \mathsf{of} \\
\quad \mathsf{nil} \qquad\qquad\qquad\quad \to \{\}, \\
\quad mk\text{-}Tree(\mathsf{nil}, md, l, r) \to \{\}, \\
\quad mk\text{-}Tree(mk, md, l, r) \to coll(l) \cup \{mk\} \cup coll(r) \\
\quad \mathsf{end}
\end{array}
$$

Nested objects like *Tree* have, in $\pi o \beta \lambda$, to be represented by structures built with references. An invariant must specify that the reference structure forms a genuine tree (*is-linked-tree*) and that the *Tree* obtained by using *extract-tree* on the instances satisfies *is-ordered-tree*.

$$
\begin{array}{l}
inv : Ref \to \Sigma \to \mathbb{B} \\
\\
inv(p)(\sigma) \triangleq \\
\quad is\text{-}linked\text{-}tree(p, l, r)(\sigma) \wedge is\text{-}ordered\text{-}tree(extract\text{-}tree(p, l, r, mk)(\sigma))
\end{array}
$$

The functions *is-linked-tree* and *extract-tree* can be defined in a similar way to *is-linked-list* above.[7] The retrieve function follows.

$$
\begin{array}{l}
retr : Ref \to \Sigma \to (Key \xrightarrow{m} Data) \\
\\
retr(p)(\sigma) \triangleq retrm(extract\text{-}tree(p, l, r, km)(\sigma))
\end{array}
$$

[7] It might, however, be worth passing lambda expressions rather than names to define the link tracing.

$retrm : [Tree] \to (Key \xrightarrow{m} Data)$

$retrm(t) \triangleq$
  cases t of
  nil $\to \{\}$,
  $mk\text{-}Tree(\text{nil}, md, l, r) \to \{\}$,
  $mk\text{-}Tree(mk, md, l, r) \to retrm(l) \cup \{mk \mapsto md\} \cup retrm(r)$
  end

The methods are respecified as follows.

$Symtab$ class
vars mk: $Key \leftarrow$ nil; md: $Data \leftarrow$ nil;
  l: private ref($Symtab$) $\leftarrow$ nil; r: private ref($Symtab$) $\leftarrow$ nil
$insert(k: Key, d: Data)$ method
  post $retr(extract\text{-}tree(\text{self}, l, r, mk)(\sigma)) =$
    $retr(extract\text{-}tree(\text{self}, l, r, mk)(\overleftarrow{\sigma})) \dagger \{k \mapsto d\}$
$search(k: Key)$ method $res: Data$
  pre $k \in \text{dom } retr(extract\text{-}tree(\text{self}, l, r, mk)(\sigma))$
  post $res = (retr(extract\text{-}tree(\text{self}, l, r, mk)(\sigma)))(k)$

Operation decomposition

It is straightforward to provide code which satisfies the specifications above.

$Symtab$ class
vars mk: $Key \leftarrow$ nil; md: $Data \leftarrow$ nil;
  l: private ref($Symtab$) $\leftarrow$ nil; r: private ref($Symtab$) $\leftarrow$ nil
$insert(k: Key, d: Data)$ method
  if $mk = \text{nil}$ then $(mk \leftarrow k;\ md \leftarrow d)$
  elif $mk = k$ then $md \leftarrow d$
  elif $k < mk$ then (if $l = \text{nil}$ then $l \leftarrow$ new $Symtab$ fi $l!insert(k, d)$)
  else (if $r = \text{nil}$ then $r \leftarrow$ new $Symtab$ fi $r!insert(k, d)$)
  fi
  return
$search(k: Key)$ method $res: Data$
  pre $k \in \text{dom } retr(\text{self})$
  if $k = mk$ then return md
  elif $k < mk$ then return $l!search(k)$
  else return $r!search(k)$
  fi

The argument that this code satisfies its specification uses structural induction over the tree topology.

Equivalent code

As in Section 3, the above code is sequential (within one instance of a tree). The transformation in Equation 1 can be used to justify moving the return to the beginning of $insert$.

There is, however, a problem with re-ordering the statements of *search*: no result can be returned until it has been found so the caller of the method has to be held up. But an instance of *Symtab* can be used by another process if the task of delivering a result is delegated (to another instance). This is exactly the semantics of the yield statement. The equivalence used is

$$\text{return } l!m(x) \quad \rightsquigarrow \quad \text{yield } l!m(x) \tag{2}$$

providing l is a private reference and only references via private references. Thus the above code can be transformed into the following.

```
Symtab class
vars mk: Key ← nil; md: Data ← nil;
                    l: private ref(Symtab) ← nil; r: private ref(Symtab) ← nil
insert(k: Key, d: Data) method
  return
  if mk = nil then (mk ← k; md ← d)
  elif mk = k then md ← d
  elif k < mk then (if l = nil then l ← new Symtab fi l!insert(k, d))
  else (if r = nil then r ← new Symtab fi r!insert(k, d))
  fi
search(k: Key) method res: Data
  if k = mk then return md
  elif k < mk then yield l!search(k)
  else yield r!search(k)
  fi
```

4 Relationship of $\pi o\beta\lambda$ to POOL

This section comments on the differences between $\pi o\beta\lambda$ and the language which inspired its creation. A useful overview of the work on POOL is [Ame89]. Pierre America and Jan Rutten produced a combined doctoral thesis [AR89] which contains a collection of papers (some published elsewhere) on the formal aspects of the POOL project including their work on (metric) denotational semantics. A proof theory for a sequential version of POOL is given in [Ame86], while [AdB90] addresses proofs about process creation in a language called P which is more like CSP or CCS in the way that communication is a single event without any way to return a value. A proof method for the full *rendezvous* mechanism of POOL is given in [dB91]: but this multi-level approach is not compositional in a useful sense.

The main changes from POOL (see [Ame89, Ame91]) are:

1. In $\pi o\beta\lambda$, methods do not have a body (which, in POOL, is a statement which shows – for instances of the class – when a *rendezvous* can occur as well as executing autonomous code between method invocations); the examples here were longer with a body and it rarely did anything interesting; one can simulate the effect of this body by code in methods and switches etc.
2. The new message to a class can be defined by an explicit method in $\pi o\beta\lambda$.
3. Methods in $\pi o\beta\lambda$ which do not return a value are distinguished from those which do.
4. The *yield* statement is new in $\pi o\beta\lambda$.

5. The *Parallel* statement is also new but is an obvious extension.
6. References in $\pi o\beta\lambda$ are typed.
7. POOL has a local call; this could easily be added to $\pi o\beta\lambda$.
8. Clearly, $\pi o\beta\lambda$ needs some way of controlling conditional 'firing' of methods.

The development method presented here is not like any in the POOL literature. The approach illustrated in the current paper is the way that developments can first employ normal sequential reasoning based on pre-/post-conditions and then use transformations to admit concurrency (similar ideas are present in the works of Lipton [Lip75], Lengauer [Len82], Zwiers [JPZ91], Xu/He [XH91, Xu92] and the well-known UNITY approach [CM88]; equivalence laws are given in [HHJ+87, RH86]; see [OA91]).

Some open issues in $\pi o\beta\lambda$ are:

1. Methods could be divided into those which have a side-effect and those which are purely functional – this is done in UFO [Sar92].
2. It is not clear whether it would be worth distinguishing mutable values from what are constants in other languages – this affects the need for interference assertions (cf. the infamous *ordered-collection* example).
3. So far, $\pi o\beta\lambda$ has not used the (ST) trick of defining operators (e.g. $+, \neg$) as methods; since there are no 'block expressions' the option to do the same for while does not exist.
4. Block statements and exceptions might be added (exceptions could be in the style of VDM's exit).
5. The language has no inheritance yet (it is tempting to try something like 'theory morphisms' – cf. [JJLM91] – because inheritance is often used to solve too many problems at once).
6. There is some case for adding constant (e.g. numeric) channel names (cf. [Jon93b]).

5 Discussion

Clearly there is much more work to be done. Apart from considering other examples, the major activity is to complete the companion paper which provides a semantics for $\pi o\beta\lambda$ in terms of the π-calculus (this approach results from technical difficulties with a more conventional operational or denotational semantics which are discussed further in [Jon93a]). This will be the basis on which the proof obligations are to be justified.

Acknowledgements

The author is grateful to Mario Wolczko, Carlos Figueiredo, Trevor Hopkins, John Sargeant, Michael Fisher and John Gurd for stimulating discussions on topics related to the implementation of object-based languages and machine architectures. The incentive provided by the discussions with the 'Object-Z' group at the University of Queensland is also remembered. Ketil Stølen prompted the use of predicates like *is-linked-list* during an enjoyable visit to Munich. Anders Ravn made useful comments on a draft of this paper and Kohei Honda provided a detailed criticism of both content and presentation style. Feedback from the 1992 meeting of IFIP WG 2.3 was stimulating as were the questions on a trip to NWPC in Bergen and at a seminar in Oslo. The support of a Senior Fellowship from the SERC is gratefully acknowledged.

References

[AdB90] P. America and F. de Boer. A proof system for process creation. In *[BJ90]*, pages 303–332, 1990.

[Ame86] Pierre America. A proof theory for a sequential version of POOL. Technical Report 0188, Philips Research Laboratories, Philips Research Laboratories, Nederlandse Philips Bedrijven, B.V., September 1986.

[Ame89] Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4), 1989.

[Ame91] P. America. Formal techniques for parallel object-oriented languages. In *[BG91]*, pages 1–17, 1991.

[AR89] Pierre America and Jan Rutten. *A Parallel Object-Oriented Language: Design and Semantic Foundations*. PhD thesis, Free University of Amsterdam, 1989.

[AR92] Pierre America and Jan Rutten. A layered semantics for a parallel object-oriented language. *Formal Aspects of Computing*, 4(4):376–408, 1992.

[BF91] J. A. Bergstra and L. M. G. Feijs, editors. *Algebraic Methods II: Theory Tools and Applications*, volume 490 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[BG91] J. C. M. Baeten and J. F. Groote, editors. *CONCUR'91 – Proceedings of the 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[BJ90] M. Broy and C. B. Jones, editors. *Programming Concepts and Methods*. North-Holland, 1990.

[BSI92] BSI. VDM specification language protostandard. Technical Report N-231, BSI IST/5/19, 1992.

[CLW79] K. M. Chung, F. Luccio, and C. K. Wong. A new permutation algorithm for bubble memories. Technical Report RC 7633, IBM Research Division, 1979.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Daw91] J. Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.

[dB91] Frank S. de Boer. *Reasoning about Dynamically Evolving Process Structure*. PhD thesis, Free University of Amsterdam, 1991.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[GHW85] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report 5, DEC, SRC, July 1985.

[HHJ+87] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sørensen, J. M. Spivey, and B. A. Sufrin. The laws of programming. *Communications of the ACM*, 30(8):672–687, August 1987. see Corrigenda in Communications of the ACM, 30(9): 770.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[Jon83] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon91] C. B. Jones. Interference resumed. In P. Bailes, editor, *Engineering Safe Software*, pages 31–56. Australian Computer Society, 1991.

[Jon92] C. B. Jones. An object-based design method for concurrent programs. Technical Report UMCS-92-12-1, Manchester University, 1992.

[Jon93a] C. B. Jones. Giving semantics to an object-based design notation. In *CONCUR'93*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[Jon93b] C. B. Jones. Reasoning about interference in an object-based design method. In *FME'93*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[JPZ91] W. Janssen, M. Poel, and J. Zwiers. Action systems and action refinement in the development of parallel systems. In *[BG91]*, pages 298–316, 1991.

[KMMN91] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. Object oriented programming in the Beta programming language. Technical report, University of Oslo and others, September 1991.

[Lam90] L. Lamport. A temporal logic of actions. Technical Report 57, Digital Equipment Corporation, Systems Research Center, 1990.

[Lam91] L. Lamport. The temporal logic of actions. Technical Report 79, Digital, SRC, 1991.

[Len82] C. Lengauer. *A Methodology for Programming with Concurrency*. PhD thesis, Computer Systems Research Group, University of Toronto, 1982.

[Lip75] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 12:717–721, 1975.

[Mil92] R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*. Springer-Verlag, 1992.

[Nel91] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[OA91] E.-R. Olderog and K. R. Apt. Using transformations to verify parallel programs. In *[BF91]*, pages 55–82, 1991.

[PT91] S. Prehn and W. J. Toetenel, editors. *VDM'91 – Formal Software Development Methods. Proceedings of the 4th International Symposium of VDM Europe, Noordwijkerhout, The Netherlands, October 1991. Vol.1: Conference Contributions*, volume 551 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[RH86] A. W. Roscoe and C. A. R. Hoare. Laws of occam programming. Monograph PRG-53, Oxford University Computing Laboratory, Programming Research Group, February 1986.

[Sar92] J. Sargeant. UFO – united functions and objects draft language description. Technical Report UMCS-92-4-3, Manchester University, 1992.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, Manchester University, 1990. available as UMCS-91-1-1.

[Stø91a] K. Stølen. A Method for the Development of Totally Correct Shared-State Parallel Programs. In *[BG91]*, pages 510–525, 1991.

[Stø91b] K. Stølen. An Attempt to Reason About Shared-State Concurrency in the Style of VDM. In *[PT91]*, pages 324–342, 1991.

[Wol88] Mario I. Wolczko. *Semantics of Object-Oriented Languages*. PhD thesis, Department of Computer Science, University of Manchester, January 1988.

[XH91] Qiwen Xu and Jifeng He. A theory of state-based parallel programming by refinement: Part I. In J. Morris, editor, *Proceedings of The Fourth BCS-FACS Refinement Workshop*. Springer-Verlag, 1991.

[Xu92] Qiwen Xu. *A Theory of State-based Parallel Programming*. PhD thesis, Oxford University, 1992.

[Yon90] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.

[Zwi88] J. Zwiers. *Compositionality, Concurrency and Partial Correctness: Proof theories for networks of processes, and their connections*. PhD thesis, Technical University Eindhoven, 1988. available as LNCS 321, Springer-Verlag.