1

NON-INTERFERENCE PROPERTIES OF A CONCURRENT OBJECT-BASED LANGUAGE: PROOFS BASED ON AN OPERATIONAL SEMANTICS

S. J. Hodges and C. B. Jones

Department of Computer Science,
Manchester University, M13 9PL, UK

ABSTRACT

A carefully designed concurrent object-oriented language can provide a suitable target for a compositional design process that copes with the interference inherent with concurrency. The semantics of a particular object-based design language have been written using operational semantics and by a mapping to the first-order polyadic π-calculus. This paper focuses on the operational semantics and indicates how proofs of some key properties about interference can be based on such a definition: the final theorems prove that certain equivalences –which can be used to enhance concurrency– are sound. The limitations of the operational semantics based proofs are discussed as is the interest in transferring intuitions from these proofs to ones based on a mapping to the π-calculus.

1 INTRODUCTION

The results in this paper relate to a broad programme of research which aims to provide compositional development methods for concurrent programs; this context is described in [Jon96]. In that reference, some concepts from object-oriented languages are argued to be useful in taming interference; the concurrent object-based design language proposed is known as $\pi o \beta \lambda$. The specific aspect of the research programme which is addressed here is the soundness of equivalences which can be used to increase concurrency in $\pi o \beta \lambda$; [Jon96] also discusses the application of rely-/guarantee-conditions to reasoning about com-

plex interference in object-based programs but no further mention of these more powerful ways of reasoning about interference is made in the current paper.

Motivation for –and an operational semantics definition of– $\pi o \beta \lambda$ are given in Section 2 as are examples of two equivalences used in [Jon96]; Section 3 contains a soundness argument for the equivalence concerning return statement commutation. Section 4 contains a description of related work as well as comments on the viability of the operational semantics approach and alternatives.

2 AN OBJECT-BASED LANGUAGE

This section introduces and defines the $\pi o \beta \lambda$ design notation; it also motivates two equivalences concerned with concurrency.

2.1 Overview

It is easiest to envisage the scope of $\pi o \beta \lambda$ by considering a programming example even though the language is actually intended to be employed as a design notation via which programs written in languages like POOL [Ame89], ABCL [Yon90], Beta [MMN93] or Modula-3 [Nel91] can be developed. The $\pi o \beta \lambda$ class *Sort* in Figure 1 provides a (sequential) facility for maintaining a sequence of integers in ascending order. (In the envisaged development approach [Jon96], this program would actually be the result of a development process not unlike that envisaged by Hoare in [Hoa75] where the representations of recursive data structures are considered; proof obligations of a sequential development processes like [Jon90] suffice.)

A class is a template for object structure and behaviour: it lists the instance variables with their corresponding types and defines each method's parameters, result type and implementation. An instance variable can have one of three types: integer, Boolean or (typed) reference. A reference value is the identity of another object; the special value nil is used to indicate when no reference is being held. Any variable declared to be a reference must be identified as either shared or unique. The latter is written as a keyword (unique) if required, the default is shared. A variable marked as unique can only be assigned a reference of a newly created object, and it is prohibited to duplicate its contents: unique variables cannot appear on the right hand side of an assignment statement, be passed as arguments to a method or be returned as a result. These restrictions ensure

```
Sort class
vars v: ℕ ← 0; l: unique ref(Sort) ← nil
insert(x: ℕ) method
  begin
    if is-nil(l) then (v ← x; l ← new Sort)
    elif v ≤ x then l.insert(x)
    else (l.insert(v); v ← x)
    fi;
    return
  end
test(x: ℕ) method : 𝔹
  if is-nil(l) ∨ x ≤ v then return false
  elif x = v then return true
  else return l.test(x)
  fi
```

Figure 1 Example Program *Sort* – sequential

that the object identity being held is unknown to any other object. Objects of a class (objects corresponding to the class description) can be generated by executing a new statement which creates a new object with which a unique reference is associated, and returns this reference as a result. As implied above, all objects of a class share the same structure and behaviour, however, each possesses its own copy of the instance variables, and it is on these copies that the methods operate.

An object can invoke[1] a method of any object to which it holds a reference. The syntax for method invocation is $\alpha.m(\tilde{x})$, where α is the identity of the object, m is the method name and $\tilde{x}$ is the list of parameters. When an object accepts a method invocation the client is held in a *rendezvous*. The *rendezvous* is completed when a value is returned; in the simplest case this is by a return statement. (The delegate statement allows an object to transfer the responsibility for answering a method call to another object, without itself waiting for the result – see below.) In addition to the statements described above, $\pi o \beta \lambda$ provides a normal repertoire of simple statements.

It follows from the above that an object can be in one of three states: *quiescent* (dormant), *waiting* (held in *rendezvous*) or *active* (executing a method body). Methods can only be invoked in an object which is in the quiescent state;

[1]The terms 'method invocation' and 'method call' are used interchangeably below.

Figure 2 A linked-list of *Sort* objects

Figure 3 The transfer of control during $\alpha.test(4)$ – sequential

therefore –in $\pi o \beta \lambda$– at most one method can be active at any one time in a given object.

These comments should help to clarify most aspects of the sequential version of *Sort*. A sequence of integers is represented by a linked-list of *Sort* objects (cf. Figure 2). The left-most object behaves as a server containing the whole queue but, in fact, each object holds a single element of the sequence (in v) and a unique reference to the next object in the list (in l). The method *insert* places its argument such that the resulting sequence is in ascending order; while *test* searches the sequence for its argument.[2] The implementation of both these methods is sequential: at most one object is active at any one time. Figure 3 illustrates how the single thread of control is passed from object to object during an invocation of the *test* method.

Concurrency can be introduced into this example by applying two equivalences. The *insert* method given in Figure 1 is sequential: its client is held in a *rendezvous* until the effect of the insert has passed down the list structure to the appropriate point and the return statements have been executed in every object on the way back up the list. If the return statement of *insert* is commuted to the beginning of the method as in Figure 4, the client is able to continue its computation concurrently with the activity of the insertion. Furthermore, as the insertion progresses down the list, objects 'up stream' of the operation are free to accept further method calls. One can thus imagine a whole series of *insert* operations trickling down the list structure concurrently. It is interesting to note that this behaviour would be difficult to specify and would require some form of auxiliary variable.

[2]The return statement in Figure 1 has a method call in the place of an expression, which strictly does not conform to the strict syntax of $\pi o \beta \lambda$. One simple remedy would be to assign the result of this call to a temporary variable and return the value of that variable. Since this is straightforward, and adds nothing to the language, it is preferred here to rely on the reader's comprehension.

```
Sort class
vars v: ℕ ← 0; l: unique ref(Sort) ← nil
insert(x: ℕ) method
  begin
    return;
    if is-nil(l) then (v ← x; l ← new Sort)
    elif v ≤ x then l.insert(x)
    else (l.insert(v); v ← x)
    fi
  end
test(x: ℕ) method : 𝔹
  if is-nil(l) ∨ x ≤ v then return false
  elif x = v then return true
  else delegate l.test(x)
  fi
```

Figure 4 The concurrent implementation of *Sort*

As with any program transformation, it is important that the derived *Sort* class behaves in the same way as the original. The soundness of the equivalence relies on the fact that method invocations can *not* overtake each other as they pass down the list; this is ensured by the use of unique references and the restriction that at most one method can be active in an object at any one time. Equivalent behaviour could not be guaranteed if, for example, the *test* method had some 'fast path' vector of references to every tenth item in the list: sharing of references would allow objects to interfere and the behaviour of code with a commuted return statement would not correspond to that of the sequential version.

When considering possible interference between objects it is useful to partition the object structure into non-interfering regions called *islands*. Figure 5 illustrates the nesting of islands in a linked-list structure. An island can be defined as a set of objects (accessible from a *bridge* object) reflecting the transitive closure of the stored references (this is formalized in Section 3). Since no references are held in or out of the island, the bridge is the sole access point to the island. Activity within the island cannot therefore interfere with the outside world and it is this property that ensures that a whole collection of *Sort* objects appears to its client as a single server. The $\pi o \beta \lambda$ restrictions on unique references (see Definition 5 below) guarantee that a unique object is necessarily a bridge to an island.

Figure 5 Islands

It is not possible to apply the return commutation equivalence to the *test* method because the client must be held until a result can be returned. It is, however, possible to avoid the entire list being 'locked' throughout the duration of a *test* method. In the sequential implementation, invocations of the *test* method in successive instances of *Sort* run down the list structure until either the value being sought is found or the end of the list is reached; at this point the Boolean result is passed back up the list; when the result reaches the object at the head of the list it is passed to the client. If instead each object has the option to *delegate* the responsibility of answering the client, it is possible for the first object in the list to accept further method calls. Again one can imagine a sequence of *test* method calls progressing down the list concurrently. Notice however that although the linear structure of the list prevents overtaking, it is possible for invocations to be answered in a different order from that in which they were accepted. For example –in the situation illustrated in Figure 2– if two invocations are accepted in the order *test*(4) followed by *test*(1), it is possible for the result of the second call to be returned before the first has completed. Although this would constitute a modified behaviour when viewed from an all-seeing spectator, no $\pi o\beta\lambda$ program can detect the difference. The transformed implementation of *test* is given in Figure 4.[3]

Because return statements do not have to come at the end of methods and the use of delegate statements, $\pi o\beta\lambda$ is an object-based language which permits concurrency.

2.2 Semantics

The syntax of the subset of $\pi o\beta\lambda$ which is considered in this paper can be inferred from the reduction rules below; obvious *context conditions* such as the use of defined classes and methods, agreement on names of instance variables within methods and respect for the typing of references are assumed. Furthermore, the checks on the use of unique references are static. This can be straightforwardly formalized using –for example– the methods in [Bru93]. Type rules for $\pi o\beta\lambda$ have been written (by SJH) but will be presented elsewhere.

[3]Space limitations prevent further discussion of the delegate equivalence in this version of the paper.

For the semantics itself, two levels of Structured Operational Semantic (SOS) reductions are defined [Plo81]. Some mild VDM notation (constructed objects, maps, etc.) is used in this paper – see [Jon90].

Object internal reductions rely on a state containing values of instance variables of the object (this is modelled as a 'map' from *Id* to *Val*).

$\Sigma = Id \xrightarrow{m} Val$

The only basic values considered here are natural numbers and Booleans; values in *Oid* correspond to references.

$Val = \mathbb{B} \mid \mathbb{N} \mid Oid$

Lower case Greek letters (α, β, γ) are used for members of *Oid*; lower case Roman letters for members of *Id*.

The internal reductions are marked by $\xrightarrow{s}$ and define a relation over $(Stmt^* \times \Sigma)$. The rules are:[4]

$$\boxed{\leftarrow} \quad \frac{}{((x \leftarrow e;\ l), \sigma) \xrightarrow{s} (l, \sigma \dagger \{x \mapsto [\![e]\!]\sigma\})}$$

The execution of an assignment statement $(x \leftarrow e)$ causes x to be assigned the valuation of e in the state σ ($[\![e]\!]\sigma$); all the other variables defined in σ are left unchanged (this is the semantics of VDM's 'overwrite' operator). Thus the reduction rule defines a relation over $(Stmt^* \times \Sigma)$ which reflects the execution of the first statement in the sequence. Because $\pi o \beta \lambda$ expressions are side-effect free their valuation cannot cause a state change.

$$\boxed{compound} \quad \frac{}{((\langle sl \rangle;\ l), \sigma) \xrightarrow{s} (sl \frown l, \sigma)}$$

Where $\frown$ is concatenation with signature $X^* \times X^* \to X^*$.

$$\boxed{\mathsf{if}_t} \quad \frac{[\![e]\!]\sigma = \mathsf{true}}{((\mathsf{if}\ e\ \mathsf{then}\ S_t\ \mathsf{else}\ S_f;\ l), \sigma) \xrightarrow{s} (S_t;\ l, \sigma)}$$

[4] In the interests of readability, this definition –being small– is written with concrete syntax expressions of $\pi o \beta \lambda$ rather than the more pedantic abstract syntax (e.g. $\langle sl \rangle$ for *mk-Compound*(sl) and $x \leftarrow e$ for *mk-Assign*(x, e)).

$$\boxed{\mathsf{if}_f}\ \frac{[\![e]\!]\sigma = \mathsf{false}}{((\mathsf{if}\ e\ \mathsf{then}\ S_t\ \mathsf{else}\ S_f;\ l), \sigma) \xrightarrow{s} (S_f;\ l, \sigma)}$$

The global reductions require information about all of the objects which have been created – this is given in *Omap*.

$$Omap = Oid \xrightarrow{m} Oinfo$$

$$\begin{array}{lll} Oinfo :: & cn & : Id \\ & status & : Status \\ & rest & : Stmt^* \\ & state & : \Sigma \\ & client & : [Oid] \end{array}$$

$$Status = \textsc{Act} \mid \textsc{Quies} \mid Wait$$

$$Wait :: var : Id$$

The *cn* component contains the class name (which indexes *Cmap* – see below); *status* indicates which 'state' the object is in[5]; *rest* contains the statements yet to be executed by the currently active method; *state* contains the indexed values of the instance variables; and *client* contains the *Oid* of the place to where the value will be returned by a method (if any). [6]

The global reductions also require the class definitions to hand.

$$Cmap = Id \xrightarrow{m} Cdef$$

$$\begin{array}{lll} Cdef :: & vs & : Id \xrightarrow{m} Type \\ & initvals & : Id \xrightarrow{m} Val \\ & mm & : Id \xrightarrow{m} Mdef \end{array}$$

$$Type = \textsc{Bool} \mid \textsc{Nat} \mid UniqueRef \mid SharedRef$$

$$UniqueRef :: c : Id$$

$$SharedRef :: c : Id$$

$$\begin{array}{lll} Mdef :: & r & : [Type] \\ & pl & : (Id \times Type) \\ & b & : Stmt \end{array}$$

[5] We use WAIT(r) as convenient notation for $mk\text{-}Wait(r)$, where r is the variable to which the result of the method call will be assigned.

[6] Tuple notation is used where convenient even though *Oinfo* is strictly a composite object (e.g. $(c, b, l, \sigma, \omega)$ is written instead of $mk\text{-}Oinfo(c, b, l, \sigma, \omega)$).

The following functions are required in the reduction rules given below.

$$
\begin{array}{lcl}
\mathit{init}(C, c) & \triangleq & \mathit{initvals}(C(c)) \\
\mathit{vars}(C, c) & \triangleq & \mathsf{dom}\ \mathit{vs}(C(c)) \\
\mathit{body}(C, c, m) & \triangleq & b(\mathit{mm}(C(c))(m)) \\
\mathit{param}(C, c, m) & \triangleq & \mathit{pl}(\mathit{mm}(C(c))(m))_1
\end{array}
$$

The global reductions are marked $\xrightarrow{g}$ and, for a given *Cmap*, define a relation over *Omap*. The overall semantics of a *System* is given by the global reductions through which an *Omap* evolves. The reductions may or may not terminate. The *initial state* must be an *Omap* which records a collection of objects (which match the given *Cmap*) all of which are initially active and for which each *Oinfo* has a nil *client* field.

An *Omap*, O, and a *Cmap*, C, are 'coherent' if and only if all classes mentioned in O are defined in C (this is taken for granted below). Formally, $\forall (c, b, l, \sigma, \omega) \in \mathsf{rng}\ O \cdot c \in \mathsf{dom}\ C$. Coherence is initially ensured and can be seen to be maintained by subsequent reductions.

An *Omap* O can be converted into an *alpha-equivalent Omap* O' by substituting all occurrences of an *Oid* in O by another *Oid* not in O.

The first global reduction rule shows how a step of the local reductions can be reflected at the global level.

$$
\boxed{s \rightsquigarrow g}\ \frac{\begin{array}{c} O(\alpha) = (c, \textsc{Act}, l, \sigma, \omega) \\ (l, \sigma) \xrightarrow{s} (l', \sigma') \end{array}}{C \vdash O \xrightarrow{g} O \dagger \{\alpha \mapsto (c, \textsc{Act}, l', \sigma', \omega)\}}
$$

This shows how a $\xrightarrow{s}$ reduction in a single object affects the global state.

The reduction rule which fixes the semantics of the new statement creates a new entry in the *Omap*. (The type rules and coherence guarantee that $c \in \mathsf{dom}\ C$.) The new object's instance variables are initialized to their default values.

$$
\boxed{\mathsf{new}}\ \frac{\begin{array}{c} O(\alpha) = (c_\alpha, \textsc{Act}, (v \leftarrow \mathsf{new}\ c;\ l), \sigma, \omega) \\ \beta \notin \mathsf{dom}\ O \end{array}}{C \vdash O \xrightarrow{g} O \dagger \left\{ \begin{array}{l} \alpha \mapsto (c_\alpha, \textsc{Act}, l, \sigma \dagger \{v \mapsto \beta\}, \omega), \\ \beta \mapsto (c, \textsc{Quies}, [\,], \mathit{init}(C, c), \mathsf{nil}) \end{array} \right\}}
$$

As well as reflecting the execution of the new statement in the α object, this reduction establishes a new (β) object.

Method invocation forces the client object into the waiting state retaining the variable to which the result will be assigned. The (single) argument is added to the server's state before the method is executed.

$$\boxed{\textsf{call}}\ \frac{\begin{array}{l} O(\alpha) = (c_\alpha, \textsc{Act}, (r \leftarrow v.m(e);\ l_\alpha), \sigma_\alpha, \omega) \\ \sigma_\alpha(v) = \beta \\ O(\beta) = (c_\beta, \textsc{Quies}, [\,], \sigma_\beta, \textsf{nil}) \\ \sigma'_\beta = \sigma_\beta \dagger \{param(C, c_\beta, m) \mapsto [\![e]\!]\sigma_\alpha\} \end{array}}{C \vdash O \xrightarrow{g} O \dagger \left\{ \begin{array}{l} \alpha \mapsto (c_\alpha, \textsc{Wait}(r), l_\alpha, \sigma_\alpha, \omega), \\ \beta \mapsto (c_\beta, \textsc{Act}, body(C, c_\beta, m), \sigma'_\beta, \alpha) \end{array} \right\}}$$

Notice that the coherence of (C, O) is preserved.

The return statement links the client to the server again and releases the latter from the *rendezvous*; if l_β is non-empty, it will run concurrently with l_α.

$$\boxed{\textsf{ret}}\ \frac{\begin{array}{l} O(\alpha) = (c_\alpha, \textsc{Wait}(r), l_\alpha, \sigma_\alpha, \omega) \\ O(\beta) = (c_\beta, \textsc{Act}, (\textsf{return}\ (e');\ l_\beta), \sigma_\beta, \alpha) \end{array}}{C \vdash O \xrightarrow{g} O \dagger \left\{ \begin{array}{l} \alpha \mapsto (c_\alpha, \textsc{Act}, l_\alpha, \sigma_\alpha \dagger \{r \mapsto [\![e']\!]\sigma_\beta\}, \omega), \\ \beta \mapsto (c_\beta, \textsc{Act}, l_\beta, \sigma_\beta, \textsf{nil}) \end{array} \right\}}$$

As described in Section 2.1, the delegate statement passes the task of returning a value to another object. Notice that the α object runs concurrently with the β object and that the object indexed by α can accept new method calls after l_α is complete.

$$\boxed{\textsf{del}}\ \frac{\begin{array}{l} O(\alpha) = (c_\alpha, \textsc{Act}, (\textsf{delegate}\ v.m(e);\ l_\alpha), \sigma_\alpha, \omega) \\ \sigma_\alpha(v) = \beta \\ O(\beta) = (c_\beta, \textsc{Quies}, [\,], \sigma_\beta, \textsf{nil}) \\ \sigma'_\beta = \sigma_\beta \dagger \{param(C, c_\beta, m) \mapsto [\![e]\!]\sigma_\alpha\} \end{array}}{C \vdash O \xrightarrow{g} O \dagger \left\{ \begin{array}{l} \alpha \mapsto (c_\alpha, \textsc{Act}, l_\alpha, \sigma_\alpha, \textsf{nil}), \\ \beta \mapsto (c_\beta, \textsc{Act}, body(C, c_\beta, m), \sigma'_\beta, \omega) \end{array} \right\}}$$

When an object has completed the execution of a method its state is purged of the method parameters (if any) before going quiescent.

$$\boxed{\text{quies}}\ \frac{\begin{array}{c} O(\alpha) = (c_\alpha, \textsc{Act}, [\,], \sigma_\alpha, \mathsf{nil}) \\ \sigma'_\alpha = \mathit{vars}(C, c_\alpha) \lhd \sigma_\alpha \end{array}}{C \vdash O \xrightarrow{g} O \dagger \{\alpha \mapsto (c_\alpha, \textsc{Quies}, [\,], \sigma'_\alpha, \mathsf{nil})\}}$$

The concurrency of $\pi o \beta \lambda$ is clear from the SOS because several objects may be able to progress at any given moment. The order in which they actually progress is non-deterministic.

Several straightforward properties of the semantics are noted.

Remark 1 The *Omap* only grows (objects are not garbage collected). Therefore there are never any 'dangling references'.[7]

$$\forall (c, b, l, \sigma, \omega) \in \mathsf{rng}\ O \cdot \forall \beta \in \mathit{Oid} \cdot \beta \in \mathsf{rng}\,\sigma \ \Rightarrow\ \beta \in \mathsf{dom}\ O$$

Remark 2 The class of an object remains unchanged after creation.

Lemma 3 The objects which are waiting for values to be returned are in one-to-one correspondence with the objects which will (providing they terminate) provide such a value.

$$\begin{array}{l} (\forall \beta, \beta' \in \mathsf{dom}\ O \cdot \beta \neq \beta' \ \Rightarrow\ \mathit{client}(O(\beta)) \neq \mathit{client}(O(\beta'))) \wedge \\ \forall \alpha \in \mathsf{dom}\ O \cdot \forall r \in \mathsf{dom}\ \mathit{state}(O(\alpha)) \cdot \\ \quad \mathit{status}(O(\alpha)) = \textsc{Wait}(r) \ \Leftrightarrow\ \exists \beta \in \mathsf{dom}\ O \cdot \mathit{client}(O(\beta)) = \alpha \end{array}$$

Proof By computational induction on the SOS reductions: the base case follows from the fact that there are no waiting objects in the initial state; the induction on the $\xrightarrow{g}$ relation is straightforward (the only non-trivial case is the del rule where the *client* name is moved from one object to another).

2.3 Unique References

Definition 4 An immutable object is derived from a class whose methods have no side-effects (neither directly by modifying its own instance variables

[7] Of course, there is nothing to stop an implementation disposing of redundant objects but this must be done in a way which avoids dangling references.

nor indirectly changing those of another object via a method invocation). Thus, once initialized, an immutable object's state remains unchanged. This property allows immutable objects to be freely copied.

Definition 5 A unique reference must never be copied nor have references to mutable objects passed over it – neither in nor out.

The uniqueness of an *Oid* in an *Omap* is defined by a function *is-unique*:

$$\textit{is-unique} : \textit{Oid} \times \textit{Omap} \to \mathbb{B}$$
$$\textit{is-unique}(\gamma, O) \quad \triangleq \quad \exists! \, (\alpha, v) \in (\textit{Oid} \times \textit{Id}) \cdot \gamma = \textit{state}(O(\alpha))(v)$$

Lemma 6 Instance variables which are marked unique contain –in any O which can arise– either nil or a $\gamma \in \textit{Oid}$ such that $\textit{is-unique}(\gamma, O)$.

Proof Follows from $\pi o \beta \lambda$ restrictions on unique references (see Definition 5) and inspection of the reduction rules.

2.4 Islands

This section firms up the informal discussion of islands that was based on Figure 5. Functions *refs-from* and *refs-to* are defined as follows.

$$\textit{refs-from} : \textit{Oid} \times \textit{Omap} \to \textit{Oid}\text{-set}$$
$$\textit{refs-from}(\alpha, O) \quad \triangleq \quad \mathsf{rng}\,(\textit{state}(O(\alpha)) \rhd \textit{Oid})$$

$$\textit{refs-to} : \textit{Oid} \times \textit{Omap} \to \textit{Oid}\text{-set}$$
$$\textit{refs-to}(\alpha, O) \quad \triangleq \quad \{\beta \in \mathsf{dom}\ O \mid \alpha \in \mathsf{rng}\,(\textit{state}(O(\beta)))\}$$

The function $\textit{island}(\gamma, O)$ finds the smallest island containing any *Oid* which either refers to anything within the island, or is referred to from the island *except* the *Oid* of the (unique) object which refers to γ.

$island : Oid \times Omap \to Oid\text{-set}$

$island(\gamma, O) \quad \triangleq$
  let $\alpha \in Oid$ be s.t. $\gamma \in \mathsf{rng}\,(state(O(\alpha)))$ in
  let A be the smallest set s.t.
  $\gamma \in A \wedge \alpha \notin A \wedge$
  $\forall \beta \in A \cdot refs\text{-}to(\beta, O) \subseteq A \cup \{\alpha\} \wedge refs\text{-}from(\beta, O) \subseteq A$

pre $is\text{-}unique(\gamma, O)$

The three lemmas which follow show when reductions over an *Omap* can be separated into sets of reductions working on disjoint parts of the *Omap*.

Lemma 7 Given a coherent (C, O) pair, and an object β such that

$rest(O(\beta)) = x \leftarrow e;\ l$

then any O' with $rest(O'(\beta)) = l$ which results from $C \vdash O \xrightarrow{g}{}^{*} O'$ can be derived by separating O into the β component (I) and its complement (E). Thus with

$I = \{\beta\} \lhd O \quad C \vdash I \xrightarrow{g} I'$
$E = \{\beta\} \lhd\!\!\!\cdot\, O \quad C \vdash E \xrightarrow{g}{}^{*} E'$

then

$O' = I' \cup E'$

Proof Follows from the $\leftarrow$ rule of the SOS: interpretation of assignments makes no use of the references out of the portion of O indexed by β so there is no interference between I and E.

Lemma 8 Given a coherent (C, O) pair, and an object β such that

$rest(O(\beta)) = v \leftarrow \mathsf{new}\ c;\ l$

then any O' with $rest(O'(\beta)) = l$ which results from $C \vdash O \xrightarrow{g}{}^{*} O'$ can be derived by separating O into the β component (I) and its complement (E). Thus with

$I = \{\beta\} \lhd O \quad C \vdash I \xrightarrow{g} I'$
$E = \{\beta\} \lhd\!\!\!\cdot\, O \quad C \vdash E \xrightarrow{g}{}^{*} E'$

there are I'', E'' which are respective alpha-equivalents[8] of I', E' such that
$O' = I'' \cup E''$

Proof Follows from the new rule of the SOS: interpretation of new statements makes no use of the references out of the portion of O indexed by β so there is no interference between I and E.

Lemma 9 Given a coherent (C, O) pair and $\gamma \in Oid$, if *is-unique*(γ, O), then for any S which does not contain return or delegate statements, any O' which results from

$$C \vdash O \dagger \{\gamma \mapsto (c_\gamma, \textsc{Act}, S;\ l, \sigma_\gamma, \omega_\gamma)\} \xrightarrow{g}{}^{*}\ O'$$

with $rest(O'(\gamma)) = l$, can be derived by separating O into its island I and its complement E. Thus with

$$I = island(\gamma, O) \lhd O \quad C \vdash I \xrightarrow{g}{}^{*}\ I'$$
$$E = island(\gamma, O) \lhd\!\!\cdot\ O \quad C \vdash E \xrightarrow{g}{}^{*}\ E'$$

there are I'', E'' which are respective alpha-equivalents of I', E' such that
$O' = I'' \cup E''$

Proof Follows by induction on the $\xrightarrow{g}$ relation – there are no references in or out of $island(\gamma, O)$ so there is no interference between the two parts.

3 SOUNDNESS OF THE EQUIVALENCES

The overall strategy in these proofs is to look for confluence of the SOS. The first equivalence rule used in Section 2.1 concerns repositioning return statements. This rule is formalized and its soundness proved in this section.

Equivalence 10 S; return e is equivalent to return e; S providing

- e is not affected by S;
- S contains no return or delegate statements;

[8] This is only necessary because of the way new members of *Oid* are chosen.

Figure 6 Untransformed behaviour

Figure 7 Behaviour after return transformation

- S always terminates; and
- methods invoked by S belong to objects reached by unique references.

The proof of soundness is separated into two stages: the first examines the effect of the equivalence on a single object while the second addresses the equivalence applied to the class definition. The essence of the proof is to observe that a premature return increases parallelism by letting the client resume while the server and its subsidiary servers are still active; but the parallelism is only over objects which run in islands disjoint from the client and they cannot influence each other.

Theorem 11 From any O^u and O^t which differ only because a return transformation has been applied, their reductions will converge to a common *Omap*.

Proof Definition 5 does not completely preclude the passing of references over unique references so it is worth disposing of the issue of immutable reference passing at the outset. When an immutable reference is passed over a unique reference, a uniquely named copy of the passed object could be created (recall comment on alpha-equivalence) and thus, although they are nominally shared, immutable objects can be made to appear to their clients as if they were unique objects.

Figures 6 and 7 illustrate the effect of the return transformation applied to the first (β) object of a server. Notice that more than one client could contain a reference to the β server (it is the references within the *Sort* objects which are constrained to be unique).

The labels on the two figures mark different positions in the execution of the method call, and correspond to the suffices used on the object configurations

below. The situation of interest for Equivalence 10 is a coherent (C, O_1^u) pair where some client object α is waiting for a return from a server object β.

$O_1^u(\alpha) = (c_\alpha, \textsc{Wait}(r), l_\alpha, \sigma_\alpha, \omega)$

$O_1^u(\beta) = (c_\beta, \textsc{Act}, (S;\ \mathsf{return}\ (e');\ l_\beta), \sigma_\beta, \alpha)$

Lemma 3 ensures that the α client has a unique matching β server. The α object remains waiting until β is ready to return (recall that the third side-condition of Equivalence 10 ensures that S does terminate and fairness is assumed).

For any O_2^u $(C \vdash O_1^u \xrightarrow{g}{}^* O_2^u)$ such that

$O_2^u(\beta) = (c_\beta, \textsc{Act}, (\mathsf{return}\ (e');\ l_\beta), \sigma_{\beta 2}, \alpha)$

Note that c_β is known not to change from Remark 2 (this is taken for granted below) and $O_2^u(\alpha) = O_1^u(\alpha)$. Then by ret, $C \vdash O_2^u \xrightarrow{g} O_3^u$, where

$$O_3^u = O_2^u \dagger \left\{ \begin{array}{l} \alpha \mapsto (c_\alpha, \textsc{Act}, l_\alpha, \sigma_\alpha \dagger \{r \mapsto [\![e']\!]\sigma_{\beta 2}\}, \omega), \\ \beta \mapsto (c_\beta, \textsc{Act}, l_\beta, \sigma_{\beta 2}, \mathsf{nil}) \end{array} \right\}$$

Now consider the situation if the statements of β were commuted in O_1^u.

$O_a^t = O_1^u \dagger \{\beta \mapsto (c_\beta, \textsc{Act}, (\mathsf{return}\ (e');\ S;\ l_\beta), \sigma_\beta, \alpha)\}$

By ret this can release α from waiting: $C \vdash O_a^t \xrightarrow{g} O_b^t$, where

$$O_b^t = O_a^t \dagger \left\{ \begin{array}{l} \alpha \mapsto (c_\alpha, \textsc{Act}, l_\alpha, \sigma_\alpha \dagger \{r \mapsto [\![e']\!]\sigma_\beta\}, \omega), \\ \beta \mapsto (c_\beta, \textsc{Act}, (S;\ l_\beta), \sigma_\beta, \mathsf{nil}) \end{array} \right\}$$

First, observe $[\![e]\!]\sigma_{\beta 2} = [\![e]\!]\sigma_\beta$ because the first side condition of Equivalence 10 says that e is not affected by S. Now, it would be possible to select reductions from O_b^t which postpone the possibility of progressing the α object and achieve

$C \vdash O_b^t \xrightarrow{g}{}^* O_3^u$

The remaining question is whether any reductions from O_b^t –which interleave the α/β steps– can give rise to a configuration in which execution of S is complete but which are *not* reachable from O_3^u.

To show that *all* O_c^t $(O_b^t \xrightarrow{g}{}^* O_c^t$, such that $rest(O_c^t(\beta)) = l_\beta)$ can be reached from O_3^u, a structural induction over S is conducted.

1. If S is an assignment statement, say $x \leftarrow f$, then

$\sigma_{\beta 2} = \sigma_\beta \dagger \{x \mapsto [\![f]\!]\sigma_\beta\}$

since nothing can interfere with the *state* in the β object

$O_c^t(\beta) = (c_\beta, \textsc{Act}, l_\beta, \sigma_\beta \dagger \{x \mapsto [\![f]\!]\sigma_\beta\}, \mathsf{nil}) = O_3^u(\beta)$

Furthermore, by Lemma 7, any progress on executing l_α in O_c^t can be made independently by $\xrightarrow{g}{}^*$ reductions from O_3^u.

2. If S is a new statement, say $v \leftarrow \mathsf{new}\ c$, then for some $\gamma \notin \mathsf{dom}\ O_b^t$ (which could be the same as in O_1^u since $\mathsf{dom}\ O_1^u = \mathsf{dom}\ O_b^t$ – otherwise remove the difference by forming alpha-equivalents).

$\sigma_{\beta 2} = \sigma_\beta \dagger \{v \mapsto \gamma\} = \mathit{state}(O_3^u(\beta))$

$O_c^t(\gamma) = (c, \textsc{Act}, [\,], \mathit{init}(C, c), \mathsf{nil}) = O_3^u(\gamma)$

By Lemma 8, any progress on executing l_α in O_c^t can be be made independently by $\xrightarrow{g}{}^*$ reductions from O_3^u.

3. The case where S is a call statement, say $r \leftarrow v.m(e)$, is the one where significant extra parallelism arises. From the fourth side condition on Equivalence 10 it is known that $\sigma_\beta(v)$ is a unique reference. Let

$O(\sigma_\beta(v)) = (c_\gamma, \textsc{Act}, [\,], \sigma_\gamma, \mathsf{nil})$

Since the third side condition of Equivalence 10 guarantees termination of S, the method body $b(mm(C(c_\gamma))(m))$ can be split around its return statement as $S_1; \mathsf{return}\ e; S_2$; Lemma 9, ensures no interference with either S_1, S_2 and thus any progress on executing l_α in O_c^t can be made independently by $\xrightarrow{g}{}^*$ reductions from O_3^u.

4. Neither return nor delegate statements are allowed by the second side condition on Equivalence 10.

5. If $S \in \mathit{Compound}$, use structural induction.

6. If $S \in \mathit{If}$, use structural induction.

Notice that cases 1, 5 and 6 reflect the argument in [Jon96] that $\pi o \beta \lambda$ insulates against interference on instance variables; cases 2 and 3 indicate that unique references also provide insulation from interference.

Theorem 12 Class definitions can be changed in accordance with Equivalence 10.

Proof Islands form a nested structure over which structural induction (using Theorem 11) can be performed.

4 DISCUSSION

The proofs in this paper are based on an SOS definition of $\pi o \beta \lambda$. The first semantics written for $\pi o \beta \lambda$ was given by mapping to the π-calculus [MPW92, Mil92]. Such a semantics is recorded (with some informal proofs of the general equivalences) in [Jon93b, Jon93a, Jon94]; but completely formal proofs of the general equivalences are difficult to obtain because of the need to state –for the side conditions– what can *not* happen. The initial reservation about the use of SOS was that there is no natural algebra of such definitions. This has been mitigated here by the choice of lemmas. There are also several issues (including fairness) which require further study in any of the possible semantics.

There is a significant corpus of work that tackles the relationship between object-oriented languages and various process calculi. David Walker is a major contributor to this line of research: [Wal91] provides a mapping from two POOL-like languages to the monadic π-calculus; [Wal95] offers a mapping of a POOL-variant into the polyadic π-calculus and develops a close correspondence with an operational semantics; [Wal93] turns its attention to $\pi o \beta \lambda$ and provides both a two-level operational semantics and a mapping to the higher-order π-calculus (again noting a close correspondence); [Wal94] has a similar translation and tackles the justification of specific examples of the $\pi o \beta \lambda$ equivalences using weak bi-simulation; more recently, [LW95] discuss the correctness of specific examples of the equivalences presented above; and [PW95] reviews conditions for determinacy in $\pi o \beta \lambda$. Nestmann and Steffan use a confluence argument about $\pi o \beta \lambda$ in their paper in these proceedings. All of these proofs tackle specific instances of the equivalence rules which will not ultimately suffice for the programme set out in [Jon96] where the general transformations are to be applied by developers who need have no knowledge of the semantics (and certainly cannot be expected to have the skills of the cited authors in constructing proofs thereon). One of the current authors (CBJ) is now interested in tackling the general results based on the π-calculus semantics again (cf. [Jon93a, Jon94]) using the intuitions from the SOS proofs: this train of thought has already led to experiments with state indices to process expressions.

The $\pi o\beta\lambda$ design notation is most strongly related to POOL [Ame89] to which a detailed comparison is provided in [Jon96]. The use of the word 'islands' was prompted by Hogg (cf. [Hog91]) who had earlier come up with a concept which the current authors found independently. An interesting article on specification languages for objects has recently appeared [SSC95].

Much work remains to be done to support the full programme set out in [Jon96]. For example, as the interference logic used there stabilises, it will be necessary to undertake justification of its inference rules and this will require proofs about the relationship between logical expressions and $\pi o\beta\lambda$ statements.

Acknowledgements Apart from the people whose work is cited above, Tony Hoare, Kohei Honda, Robin Milner, Benjamin Pierce and Members of IFIP's WG 2.3 have all provided valuable comments on earlier presentations or drafts. The current authors are particularly grateful to Pierre Collette for a very detailed set of comments on a full draft of this paper.

The first author is funded by an EPSRC post-graduate studentship. The second author's research is supported by grants from EPSRC. The final version of this paper was produced whilst the second author was at the Newton Institute for Mathematical Sciences (Cambridge) – their hospitality and that of Gonville and Caius College where the second author is a Visiting Fellow is gratefully acknowledged. Both authors have had travel to Japan funded by grants from the scheme funded by the Royal Society, British Council and JSPS.

REFERENCES

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

[Bru93] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 1:1–81, 1993.

[Hoa75] C. A. R. Hoare. Recursive data structures. *International Journal of Computer & Information Sciences*, 4(2):105–132, June 1975.

[Hog91] John Hogg. Islands: Aliasing protection in object-oriented languages. In *[Pae91]*, 1991.

[Jon90] C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, second edition, 1990. ISBN 0-13-880733-7.

[Jon93a] C. B. Jones. A pi-calculus semantics for an object-based design notation. In E. Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer-Verlag, 1993.

[Jon93b] C. B. Jones. Reasoning about interference in an object-based design method. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93*, volume 670 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1993.

[Jon94] C. B. Jones. Process algebra arguments about an object-based design notation. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*, chapter 14. Prentice-Hall, 1994.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[LW95] X. Liu and D. Walker. Confluence of processes and systems of objects. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzback, editors, *TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*, pages 217–231. Springer-Verlag, 1995.

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

[MMN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object Oriented Programming in the Beta Programming Language.* Addison-Wesley, 1993.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

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

[Pae91] Andreas Paepcke, editor. *OOPSLA'91.* ACM, ACM Press, November 1991.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

[PW95] Anna Philippou and David Walker. On sharing and determinacy in concurrent systems. 1995. CONCUR'95.

[SSC95] A. Sernadas, C. Sernadas, and J. F. Costa. Object specification logic. *Journal of Logic and Computation*, 5(5):603–630, 1995.

[Wal91] D. Walker. π-calculus semantics for object-oriented programming languages. In T. Ito and A. R. Meyer, editors, *TACS'91*, volume 526 of *Lecture Notes in Computer Science*, pages 532–547. Springer-Verlag, 1991.

[Wal93] D. Walker. Process calculus and parallel object-oriented programming languages. In *In T. Casavant (ed), Parallel Computers: Theory and Practice*. Computer Society Press, 1993.

[Wal94] D. Walker. Algebraic proofs of properties of objects, 1994. Proceedings of ESOP'94.

[Wal95] D. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.

[Yon90] Akinori Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990. ISBN 0-262-24029-7.