Formal Methods in System Design 8, 105–122 (1996)
© 1996 Kluwer Academic Publishers. Manufactured in The Netherlands.

Accommodating Interference in the Formal Design of Concurrent Object-Based Programs

C.B. JONES cbj@cs.man.ac.uk
Department of Computer Science, University of Manchester, Manchester, U.K.

Abstract. This paper is about formal development methods for concurrent programs. Interference is the bane of the quest for compositional methods for concurrency. Concepts from object-oriented languages are argued to be a promising way of taming interference. Two approaches to development are described which are applicable to differing degrees of interference.

Keywords: formal development, concurrency, object-based languages, rely/guarantee-conditions

1. Introduction

Researchers have long sought to exploit parallelism in computers in order to obtain solutions to problems in shorter time than is possible on single processors. ‘Speed-up’ work which has relied on data parallelism—as in SIMD architectures—has been relatively successful. The reasons for this success are at least in part attributable to the relative ease of programming SIMD applications. The design of successful MIMD machine architectures has proved more difficult and the hunt for programming methods suitable for such machines has proved far more elusive. As well as those systems where parallelism is sought as a way of obtaining speed-up, there are applications whose external behaviour requires concurrency because many users are allowed to interact with the systems at the same time (e.g., airline reservation systems, bank teller systems). In the design of such applications the overall system specification has to address the issue of potential interference between the different users. This paper advocates the handling of concurrency by design methods which create concurrent object-oriented programs; it is essentially about shared variable (MIMD) concurrency. Two approaches to the design of such programs are described below.

This paper should then be judged as a contribution to ways of developing a class of concurrent programs (it is unlikely that any single method will be able to realistically claim to cover the myriad forms of concurrency which are required in systems design). In particular, this paper offers two compositional development approaches for shared-variable programs. It is based on the general argument that some concepts of object-oriented programming appear to offer useful control of the interference which is inherent with concurrency.

In order to understand the notion of compositionality, it is necessary to say something about the way in which the design of a system can be recorded in a top-down presentation. All components—from the overall system itself to the lowest level of detail—should be defined by specifications. A design step should provide a way of satisfying a specification which is not yet at a level where it can be directly programmed; in general, such design steps introduce new components. A development method is said to be *compositional* if the fact that a design step satisfies its specification can be proved solely on the basis of the specifications of any constituent components. The most important technical consequence of compositionality is that reasoning can be local: the proof of correctness of one design step should be possible without knowledge of the eventual internal structure of its new components. A key practical consequence is that errors can be detected early in development rather than lurking until encountered after much further work has been expended on a flawed design. To some extent, compositionality also facilitates separate development of components; but poorly chosen interfaces will always result in difficulties of separating concerns and unrealisable specifications will cause reconsideration of interfaces whatever formalism is applied.

For sequential—as opposed to concurrent—computer systems, compositional development methods like VDM have achieved some success. It is the *interference* which is inherent with concurrency which has made it difficult to find compositional development methods for programs which run in parallel. Furthermore, it is an unfortunate corollary of interference that the developer needs to be concerned with the issue of *granularity.* A further claim for the approaches described in this paper is that the control of granularity is left in the hands of the designer rather than being dictated by the development method itself.

Section 2 describes a design approach which is appropriate in cases where interference is limited. The approach involves the use of features from object-oriented languages to show how recursive data structures can be represented by families of objects. The overall approach is to develop first a sequential program and then to show that an equivalent concurrent program can be found.

More troublesome forms of interference are considered in Section 3. Earlier papers propose using rely/guarantee-conditions to document and reason about intimate interference between shared-variable programs; the second development approach shows how these ideas can be used for concurrent object-oriented languages. (The task undertaken in Section 3 is also a speed-up application because systems where interference is exhibited at the outermost level tend to be large.)

Both approaches can be considered to be in the 'posit and prove' school in which formalism is intended to support the verification of design steps which a developer might anyway wish to make. There is here—however—a greater use of transformation ideas than can be found in earlier work on sequential development in VDM.

Section 4 relates this paper to other research and describes the status of work which justifies the two design approaches by providing a formal semantics.

This paper unifies material from two earlier conference papers [22, 23] but this amalgamation and work on the semantics of the design language have resulted in a number of changes. These range from different keywords in the language to simplifications of the approaches in both major sections below: Section 2 presents a more transformational view of the design of data structures than was the case in [22] where invariants on object graphs were seen as the main tool for controlling interference; in Section 3 there is a direct use of rely/guarantee-conditions in contrast to the special logic of [23]. The same applications as were used in the earlier papers are retained as examples: this affords comparison also with the wider literature.

The notation used in the designs below is known as $\pi o\beta\lambda$ (pronounced 'Pobble' as in Edward Lear's poem!) to mark its debt both to POOL and to the π-calculus. It is intended

```
Primes class
vars max: ℕ; sr: unique ref(Sift)
init (n: ℕ)
   begin
      vars ctr: ℕ
      max ← n;
      sr ← new Sift;
      ctr ← 2;
      while ctr ≤ max do  sr.setup(ctr); ctr ← ctr + 1 od
   end
test(n: ℕ) method res: 𝔹
   return sr.test(n)
end  Primes

Sift class
vars m: ℕ ← 0; l: unique ref(Sift) ← nil
setup(n: ℕ) method
   begin
      return ;
      if l = nil then (l ← new Sift; m ← n)
      else  if ¬ m div  n then l.setup(n) fi
      fi
   end
test(n: ℕ) method res:𝔹
   if l = nil ∨ n < m then return false
   elif m = n then return true
   else delegate l.test(n)
   fi
end  Sift
```

Figure 1. Example primes program.

that $\pi o \beta \lambda$ should be used to develop programs in languages like POOL [1], ABCL [41], Beta [25] or Modula-3 [7].

Even though $\pi o \beta \lambda$ is not itself intended as a programming language, it is easy to see its scope by looking at a programming example.

The program presented in figure 1 provides a way of deciding whether natural numbers—up to some stated maximum—are prime or composite. (A development broadly following the approach of Section 2 of this program is given in [23].) Classes (here *Primes* and *Sift*) are templates which define families of objects. The class definitions fix the instance variables and their type, initialisation and the methods associated with the class. Objects corresponding to the class template can be created by executing a new statement which creates a new object with which a unique reference is associated and returns that reference as a result. The initialisation code of the class (if any) is executed to establish the starting state of the object (where there is no initialisation statement, the instance variables are normally provided with initial values which can be thought of as 'syntactic sugar' for an initialisation section).

Each object has its own copy of the instance variables and it is these copies to which reference is made by the methods of the class. The methods of a particular object are invoked using a syntax which has the unique reference associated with that object followed by '.' and the name of the method with any parameters to be passed. At most one method can be active per object at any one time—so an object becomes free for a further method call only when a method has finished execution. (One consequence of this is that $\pi o\beta\lambda$ programs would deadlock if recursive calls were attempted: the approach described in Section 2 carefully obviates this danger.)

Both the initialisation part and the body of each method is a single statement but this statement can be a block. A method call establishes a *rendez-vous* between client and server. The client has to wait if the object is busy; a *rendez-vous* finishes when a value is returned (in the simplest case this is by a **return** statement of the invoked method; but see **delegate** below). In addition to method invocation and new statements, there is a repertoire of conventional imperative statement types which can be used in method bodies.

These comments should clarify most aspects of *Primes*: its initialisation creates a sieve which can be seen as a linked-list of objects conforming to the class *Sift*. Both the reference in *sr* of *Primes* and *l* of each *Sift* object are marked as **unique** which prohibits copying (Section 2 shows that this is useful when reifying recursive data structures). The body of the method *setup* in *Sift* contains a return statement as its first action. This releases the client from its *rendez-vous*, enabling it to make further progress. One can therefore see the activity of the initialisation part of *Primes* as causing a series of *setup* methods to ripple down the linked-list instances of *Sift*. It is, of course, desirable to achieve the same sort of concurrency with the *test* methods. Here, however, the problem is slightly different in that the client must be held in a *rendez-vous* pending the return of a value; but it is still desirable to complete the execution of the body of *test* of *Sift* so that other clients can invoke methods of that particular object. This effect is achieved by using the **delegate** statement which has the effect of passing the responsibility for returning a value to the next object.

There are some facets of $\pi o\beta\lambda$ which are explored in examples below: these include the parallel statement, the use of shared references and marking classes as immutable (method guards and exceptions are available in $\pi o\beta\lambda$ but are not needed in the examples in this paper).

Concurrent object-oriented languages are seen here as an approach to implementation; although their syntax is used in the overall specifications, it is not considered useful to employ gratuitous algorithmic concepts (e.g., **new**) in specifications themselves—these bring unnecessary operational reasoning and the need to consider a global state.

Many researchers have observed that language restrictions are a crucial weapon in taming interference. The basis for the argument that some aspects of object-oriented languages provide appropriate constraints rests upon the following points.

- The instance variables of an object are safe from any interference (recall the restriction that only one method is active per object).
- Unique references provide a way of insulating other objects from interference even though they have an independent existence.
- It is only in the case of shared references that the full danger of interference is felt. Such references can be used to simulate shared variables and two active methods (necessarily in distinct client objects) can interfere with each other by the method calls they make to a server whose reference they share. The interference results from changes to the instance variables of the server but even here the actual interference can be constrained by the methods available for that class of object.

A similar series of observations can be made about the control of granularity.

The approach followed in the next section capitalizes on the first two observations; the impact of the third is seen in Section 3.

2. Avoiding interference

Section 3 shows how interference can be specified in a way which makes it possible to present compositional developments of concurrent programs. That approach is not, however, easy to use and it is certainly true that the best thing to do with interference in a design is to minimize it.

In some cases where parallelism is employed for speed-up (rather than as an inherent part of, say, a distributed system), it is possible to avoid interference by ensuring that data structures are sufficiently isolated from one another. As the preceding section has indicated, concurrent OO languages in general—and $\pi o\beta\lambda$ in particular—provide ways of indicating that data structures should be insulated from interference. In this class of problems, it is appropriate to develop a sequential program from a specification and to introduce concurrency by showing that a concurrent program is observationally equivalent to the sequential version.

2.1. *Developing a sequential implementation*

The development of a sequential program from its specification is sketched in this section. This is presented only in outline because the approach used here is close to standard formal development methods. In fact, what is presented could broadly be viewed as a VDM development in the style of [21]: the notation for sets and maps is adopted from that source, as are the notions of data reification and operation-decomposition. The only notation which might trouble a reader not conversant with VDM are: method specifications are marked with a rd/wr frame; (in post-conditions) the value of a variable prior to an operation is marked with a hook (e.g., $\overleftarrow{st}$) to contrast with the value after the operation which is undecorated (e.g., st); the notation for 'maps' or finite functions ($Key \xrightarrow{m} Data$) and the associated update operator ($\dagger$); and the use of brackets to denote the extension of a type (e.g., $[Key]$) with an optional nil value.

An abstract specification for a symbol table problem can be written as follows.

```
*Tab* class
vars $st$: $(Key \xrightarrow{m} Data) \leftarrow \{\,\}$
$insert(k\colon Key,\ d\colon Data)$ method
  wr $st$: $Key \xrightarrow{m} Data$
  post $st = \overleftarrow{st} \dagger \{k \mapsto d\}$
$search(k\colon Key)$ method $res\colon Data$
  rd $st$: $Key \xrightarrow{m} Data$
  pre $k \in$ dom $st$
  post $res = st(k)$
end *Tab*
```

As in a Larch (cf. [14]) 'interface language', the specification is framed here in the development language.

A normal design step would be to reify a data structure towards one which could be more efficiently handled in an implementation: the map $Key \xrightarrow{m} Data$ can be represented by a (self-embedding) recursive tree such as *TabInst*.

$$
\begin{aligned}
&TabInst :: k : [Key] \\
&\qquad\quad d : [Data] \\
&\qquad\quad l\ : [TabInst] \\
&\qquad\quad r\ : [TabInst] \\
&\mathsf{inv}\ (mk\text{-}TabInst(k, d, l, r)) \\
&\quad \triangleq (k = \mathsf{nil} \Leftrightarrow d = \mathsf{nil}) \wedge (k = \mathsf{nil} \Rightarrow l = r = \mathsf{nil}) \\
&\quad \wedge (\forall lk \in coll(l) \cdot lk < k) \wedge (\forall rk \in coll(r) \cdot k < rk)
\end{aligned}
$$

Where *coll* is the obvious function to collect the set of keys; its signature is

$$coll\colon [TabInst] \to Key\text{-}\mathsf{set}$$

The definition of *TabInst* can be thought of as defining a set

$$TabInst = \{mk\text{-}TabInst(k, d, l, r) \mid k \in Key \wedge \cdots \wedge (k = \mathsf{nil} \Leftrightarrow d = \mathsf{nil}) \wedge \cdots\}$$

Notice how the constructor function *mk-TabInst* is used as a pattern in the definition of the type invariant.

The relationship of the representation to the abstraction is given by a 'retrieve' function (again using the constructor as a pattern in the ordered case construct).

$$retrm\colon [TabInst] \to (Key \xrightarrow{m} Data)$$

$$
\begin{aligned}
&retrm(t) \triangleq \\
&\quad \mathsf{cases}\ t\ \mathsf{of} \\
&\quad \mathsf{nil} \qquad\qquad\qquad\quad\ \to \{\,\}, \\
&\quad mk\text{-}TabInst(\mathsf{nil}, d, l, r) \to \{\,\}, \\
&\quad mk\text{-}TabInst(k, d, l, r)\ \ \to retrm(l) \cup \{k \mapsto d\} \cup retrm(r) \\
&\quad \mathsf{end}
\end{aligned}
$$

One could now record this design step by writing a new definition of *Tab* (which is claimed to satisfy that at the beginning of this section)

```
Tab class
vars st: [TabInst] ← nil
insert(k: Key, d: Data) method ...
search(k: Key) method res: Data ...
end Tab
```

The idea here, however, is to represent the recursive type *TabInst* as a collection of objects. The fact that they are self-embedding is indicated by marking those instance variables which contain references to other objects as unique (such references cannot be copied so no sharing is possible). Thus

```
Tab class
vars mk: Key ← nil; md: Data ← nil;
                              l: unique ref(Tab) ← nil; r: unique ref(Tab) ← nil
insert(k: Key, d: Data) method ...
search(k: Key) method res: Data ...
end Tab
```

is the design whose satisfaction is actually of interest.

It is not difficult to see that the following code achieves the required effect for the *insert* method.

```
insert(k: Key, d: Data) method
  begin
    if mk = nil then (mk ← k; md ← d)
    elif mk = k then md ← d
    elif k < mk then (if l = nil then l ← new Tab fi ; l.insert(k, d))
    else (if r = nil then r ← new Tab fi ; r.insert(k, d))
    fi
    ;
    return
  end
```

To justify this formally requires proof rules for object creation and method call (this problem is addressed in [13]).

2.2. *An equivalent parallel program*

As forewarned, the *insert* method above is sequential: its client is held in a *rendez-vous* until the effect of an insert has passed all the way down the tree to the appropriate point and the return statements have been executed in the *insert* method of each object on the way back up the tree. This is where the first equivalence rule comes in; the preceding *insert* can be transformed into that of figure 2 by using Equivalence 1. The basic idea is to identify statements which can be executed concurrently with no observable difference of behaviour by checking that they are immune from interference.

A preliminary definition fixes what can(not) be done with unique references.

Definition 1. A unique reference is defined to be one which is never 'copied' nor which has general (unshared) references passed over it—neither in nor out (since one can't pass unique references, this restricts to references to 'immutable' objects).

```
Tab class
vars mk: Key ← nil; md: Data ← nil;
                                    l: unique ref(Tab) ← nil; r: unique ref(Tab) ← nil
insert(k: Key, d: Data) method
  begin
    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 Tab fi; l.insert(k, d))
    else (if r = nil then r ← new Tab fi; r.insert(k, d))
    fi
  end
search(k: Key) method res: Data
  if k = mk then return md
  elif k < mk then delegate l.search(k)
  else delegate r.search(k)
  fi
end Tab
```

Figure 2. Symbol table program.

Equivalence 1. S; return e is equivalent to return e; S providing

- S contains no return or delegate statements and always terminates;
- e is a simple expression (i.e., no method calls: compare Equivalence 2) and is not affected by S; and
- every method invoked by S belongs to objects reached by unique references.

Not all programs are intended to terminate; even where they are, termination is not a syntactically checkable property; but it is in the spirit of the development method envisaged that termination would be proved for relevant methods. (This point does however make it doubtful whether the kind of equivalences being considered are suitable for automatic application by a compiler.)

The sequential code for the *search* method is

```
search(k: Key) method res: Data
  if k = mk then return md
  elif k < mk then return l.search(k)
  else return r.search(k)
  fi
```

This again holds its client in a *rendez-vous* until the appropriate value is found and returned—furthermore, the whole tree is locked until the active methods terminate. It is not possible to use Equivalence 1 here because the value to be returned must be located before the client's needs can be satisfied (cf. the restriction on Equivalence 1 that e be a simple expression); it is however possible to 'delegate' the task of returning the value and to terminate the statement (and thus the method) thereby unlocking the tree: this is facilitated by Equivalence 2 which shows that the *search* above is observationally equivalent to that in figure 2.

Equivalence 2. return $l.m(x)$ is equivalent to delegate $l.m(x)$ providing

- $l.m(x)$ terminates; and
- l is a unique reference.

In conclusion, it is worth commenting that it is very difficult even to specify the program in figure 2. Any attempt to use pre/post-conditions would have to overcome the problem that both the initial and final 'states' are combinations of values and unfinished activity. Such a specification would at least need some form of auxiliary variable. So the approach of introducing parallelism by showing a program which is equivalent to a sequential program (whose specification was simple) has avoided considerable complication.

The work on justifications of Equivalences 1 and 2 is reviewed in Section 4.

3. Controlling interference

This section considers the sort of program where interference has to be lived with: earlier papers (e.g. [19, 20]) introduced the idea of describing and reasoning about interference using rely/guarantee-conditions; these ideas are significantly developed in—for example—[9, 35, 40]. The basic observation is that interference can be recorded in specifications in a way which makes it possible to formulate compositional proof rules for parallel constructs. This leads to a concept of recording assumptions which a developer can make and commitments that his or her code must fulfill. The specific proposal uses a four-tuple of predicates (p, r, g, q):

- a pre-condition p is a predicate of a single state and identifies assumptions the developer is entitled to make about the initial state in which the specified program will be invoked;
- a rely-condition r is a predicate of two states and characterises the interference which the developed code must tolerate—this can be viewed as an invitation to the developer to assume that any pair of states which differ as a result of interference will be constrained by the relation given by r;
- a guarantee-condition g is a predicate of two states and defines a restriction on any state transitions which the developed code can make;
- a post-condition q is a predicate of two states and characterises the required input/output relation of the component (VDM has always used relational post-conditions in preference to ones on a single state).

A variety of proof rules are presented in the cited papers. It is not difficult to guess their general form (e.g., components must tolerate interference from other components and from the environment in which the components are combined) but their specific formulation leads to some delicate issues (see [10]).

This section explores how the ideas for recording and reasoning about interference can be used in the framework of concurrent OO languages. It is shown that this framework reduces the weight of proof even in the case of the general interference encountered in this section.

3.1. Specification and initial design steps

As in Section 2, the development begins with a specification which is embedded in a $\pi o \beta \lambda$ class. The task to be implemented by the *Primes* class is to provide a method (*test*) which determines the primality of natural numbers up to some stated maximum; this maximum is provided as a parameter to the initialisation of the *Primes* class.

```
Primes class
vars $max$: $\mathbb{N}$
init ($n$: $\mathbb{N}$)
    wr $max$: $\mathbb{N}$
    post $max = n$
$test(n$: $\mathbb{N})$ method $res$: $\mathbb{B}$
    rd $max$: $\mathbb{N}$
    pre $2 \leq n \leq max$
    post $res \Leftrightarrow is\text{-}prime(n)$
end Primes
```

An obvious first step of design would be to introduce into the *Primes* class a set *ps* which contains all of the prime numbers up to the stated maximum. This can be viewed as a step of reification.

```
Primes class
vars $max$: $\mathbb{N}$; $ps$: $\mathbb{N}$-set
init ($n$: $\mathbb{N}$)
    wr $max, ps$
    post $max = n \wedge ps = \{2 \leq i \leq max \mid is\text{-}prime(i)\}$
$test(n$: $\mathbb{N})$ method $res$: $\mathbb{B}$
    rd $max, ps$
    pre $2 \leq n \leq max$
    post $res \Leftrightarrow (n \in ps)$
end Primes
```

The example program in figure 1 is one possible implementation of this *Primes* class and its development—in the style of Section 2—is presented in [23]; here the aim is to develop an implementation which exhibits more parallelism and exposes the problem of interference.

3.2. *Documenting and reasoning about interference*

The aim of the remainder of this development is to build a sieve (in the style of Eratosthenes) in which parallel objects remove all composite numbers from a set which is initialised to contain all natural numbers up to the maximum value required. The *Rem*(i) objects (roughly $\sqrt{max}$ of them) run in parallel and are each responsible for removing all multiples (above two) of their index i from the set. The access by the *Rem*(i) objects to the set is achieved by introducing a new class *Sieve* whose reference is shared between the *Rem*(i) objects. Notice that the need to share this reference means that it cannot be marked as unique; which, of course, inhibits the use of equivalence rules like those of Section 2. (The reader might expect to see a method in *Sieve* which deletes elements from the set: at this stage of development, removal is handled at the specification level; after a further reification, the deletion method will be placed in lower level objects.)

```
Primes class
vars $max$: $\mathbb{N}$; $sr$: ref($Sieve$)
init ($n$: $\mathbb{N}$)
  begin
    $sr \leftarrow$ new $Sieve(max)$;
      $\{sr.ps = \{2, \ldots, max\}\}$
    for all $i \in \{2, \ldots, \lceil\sqrt{max}\rceil\}$ parallel new $Rem(i, sr)$
      $\{sr.ps = \{2 \le i \le max \mid is\text{-}prime(i)\}\}$
  end
$test(n$: $\mathbb{N})$ method $res$:$\mathbb{B}$
  rd $max, sr$
  pre $2 \le n \le max$
  post $res \Leftrightarrow (n \in sr.ps)$
end Primes

Sieve class
vars $max$: $\mathbb{N}$; $ps$: $\mathbb{N}$-set
init ($n$: $\mathbb{N}$)
  post $max = n \wedge ps = \{2, \ldots, max\}$
end Sieve
```

The interesting step is to record a specification of the class *Rem*(i) which permits inference of the second annotation in the initialisation portion of *Primes* from that which precedes the parallel statement. In order to provide insight into the formulation of rely/guarantee-conditions it is useful to consider a plausible—but erroneous—post-condition for a sequential version of *Rem*; assume *mults*(i) yields a set of multiples of i up to *max*; one might think of writing

$$\overleftharpoon{ps} - ps = mults(i)$$

Even for a sequential implementation this post-condition would not be correct because not all of the multiples of i would have been present when the ith instance of *Rem*(i) began execution. One could split the putative post-condition into a requirement that certain values have been removed

$$ps \cap mults(i) = \{\ \}$$

and a requirement that no non-multiples are removed

$$\overleftharpoon{ps} - ps \subseteq mults(i)$$

The first of these two conditions is perfectly acceptable as a post-condition for *Rem*(*i*) which has to exist in an environment of limited (see below) interference. Of course, it is not in itself enough because it could be implemented by a process which removed all values from *ps*. In the case of interference, the second condition (which could be used as an additional conjunct in the post-condition of a sequential *Rem*) can be used as a guarantee-condition for *Rem*. But, in the case of a program which has to live in an interfering environment, there is an additional problem: no program can ensure that all elements of *mults*(*i*) will be absent on termination if another process can put values into *ps*. The assumption that this does not occur can be documented as a rely-condition for *Rem*(*i*); which then also has to be conjoined to the guarantee-condition.

(In another paper—joint with Pierre Collette—conditions like $ps \subseteq \overleftharpoon{ps}$ are not repeated in both the rely and guarantee-conditions but are viewed as *evolution conditions* attached to the state and can be considered to be an implicit conjunct of every rely and guarantee-condition. Evolution conditions can be compared to single state *invariants* in VDM which can be considered to be an implied conjunct of every pre and post-condition.)

Because of the way the *Sieve* object has been separated, it is necessary to refer to the set *ps* in the specification which follows as *sr.ps*. The specification of *Rem* can now be given.

```
Rem class
init (i: ℕ, sr: ref(Sieve))
  rely $sr.ps \subseteq \overleftharpoon{sr.ps}$
  guar $\overleftharpoon{sr.ps} - sr.ps \subseteq mults(i) \wedge sr.ps \subseteq \overleftharpoon{sr.ps}$
  post $mults(i) \cap sr.ps = \{\ \}$
end Rem
```

The parallel statement which is used in *Primes* above creates a family of *Rem* objects: two parameters are passed to the initialisation, the first being the index *i* for which that instance is responsible and the second being the shared reference to the *Sieve* class. Having invoked all of these processes, the parallel statement does not terminate until all of the invoked processes have terminated.

The formal proof rule for such parallel statements is concerned with the four predicates forming the specification (i.e., the pre and post-conditions in the normal sense and rely and guarantee-conditions); see [10, 19, 34] for such rules. Here, the argument is outlined as follows. Since the rely-condition of one instance of *Rem* is also a guarantee-condition of all other instances (and no interference comes from the outside environment), one can deduce that the conjunction of the post-conditions holds on termination of the parallel statement. The second annotation in *Primes* is thus established because the guarantee-condition of *Rem* shows that no primes are removed.

A further step of reification can now be undertaken to arrange that *Sieve* creates *max* objects of class *El* which each reflect whether one particular number is currently to be considered a member of the set *ps* or not (thus reifying a set into its characteristic function). This design step results in *Sieve* needing to store a 'look up' table between the natural numbers and the references to the *El* objects. Thus the first step is to document the implementation of *test* of *Primes* so that it looks up the index of the *n*th object when it applies *test*.

```
Primes class
vars $max$: $\mathbb{N}$; $sr$: ref($Sieve$)
  ⋮
test($n$: $\mathbb{N}$) method $res$: $\mathbb{B}$
  return($sr.lu(n)$).$test$()
end Primes
```

The reification of *Sieve* then becomes

```
Sieve immutable class
vars $max$: $\mathbb{N}$; $v$: $\mathbb{N} \xrightarrow{m}$ ref($El$)
init ($n$: $\mathbb{N}$)
  begin
      $max \leftarrow n$;
      for all $i \in \{2, \ldots, max\}$parallel $v(i) \leftarrow$ new $El$
         $\{retr(v) = \{2, \ldots, max\}\}$
  end
lu($n$: $\mathbb{N}$) method $res$: ref($El$)
      $\{2 \leq n \leq max\}$
  return $v(n)$
end Sieve
```

As indicated above, the *Sieve* method itself does not contain a method to delete elements; that is performed by methods in other objects; *Sieve* provides the conversion from natural numbers to references via the *lu* method.

Notice that the *Sieve* class is marked as immutable. This has several effects on the semantics of $\pi o \beta \lambda$. One interesting consideration is that multiple instances of this class could exist without changing the effect of *Primes*. This observation makes it possible to copy *Sieve* in order to resolve any performance bottleneck on the use of the *lu* method.

The reasoning required for the initialisation portion of the *Sieve* class's parallel statement is a simplified form of that which was conducted above.

The *El* class can now be implemented as follows.

```
El class
vars b: 𝔹 ← true
test() method res: 𝔹
  return b
del() method
  begin
    b ← false;
    return
  end
end El
```

Notice that the earlier guarantee condition $ps \subseteq \overleftarrow{ps}$ follows from the fact that *test* maintains the value of *b* and *del* can only make a value false. It is therefore a consequence that—once created—no element can re-enter the set having been deleted.

In a final step of operation decomposition—which again uses a simplified form of the reasoning about parallel constructs above—the *Rem* class can be implemented as follows

```
Rem class
init (i: ℕ, sr: ref(Sieve))
  for all m ∈ {2, ..., ⌊max/i⌋} parallel (sr.lu(i * m)).del()
end Rem
```

Notice it would be possible to make minor improvements by—for example—commuting the return statement in the *del* method of *El* (cf. Equivalence 1).

It must be emphasised that the more complicated design approach which had to be used here was resorted to because the straightforward approach of Section 2 is not appropriate for dealing with the sort of acyclic directed graph (DAG) which is used in this implementation.

It is a corollary of the development method proposed here that the level of granularity is in the hands of the designer.

4. Discussion

This section puts what is covered in the body of the paper into a wider context.

4.1. Related work

It is useful to record the differences between $\pi o\beta\lambda$ and POOL which is the language with most claim to $\pi o\beta\lambda$'s parentage. An overview of the work on POOL is given in [1]; Pierre America and Jan Rutten wrote a combined doctoral thesis [3] which contains a collection of papers on the formal aspects of the POOL project including their work on a denotational semantics.

The main changes from POOL (see [2]) are:

- In POOL methods have a body (which is a statement which shows—for instances of the class—when a *rendez-vous* can occur as well as executing autonomous code between method invocations); in $\pi o\beta\lambda$, methods can be guarded.
- The new message to a class can be matched by an explicit initialisation in $\pi o\beta\lambda$.
- References in $\pi o\beta\lambda$ are typed.
- Methods in $\pi o\beta\lambda$ which do not return a value are distinguished from those which do.
- The **delegate** statement is new in $\pi o\beta\lambda$.
- The **parallel** statement is also new but is an obvious extension.
- POOL has a local call; this could easily be added to $\pi o\beta\lambda$.
- $\pi o\beta\lambda$ has no inheritance.

The development approaches presented here are unlike any in the POOL literature. (A proof method for the full *rendez-vous* mechanism of POOL is given in [11]: but this multi-level approach is not compositional in a useful sense.)

The Eiffel language [29] incorporates of pre and post-conditions within an object-oriented language. Hogg uses the idea of 'islands' in [18] which are connected with the idea of unique references above.

The equivalence based approach of Section 2 is related to the approach to data structures described in [17]. Furthermore, the idea illustrated in that section to undertake developments which first employ sequential reasoning and then use equivalences to admit concurrency is similar to ideas presented by Lipton [28], Lengauer [27], Janssen, Poel and Zwiers [24], Xu and He [39, 40] and even has echoes of the well-known UNITY approach [8] or the Refinement Calculus for Reactive Systems described by Back in [4]. Equivalence laws are elevated to a language definition style in [16, 33] (see also [31]).

4.2. *Further work*

It would be interesting to add to Equivalences 1 and 2 (or even to relax their overly strict side-conditions) but the author does not yet plan to emulate [16, 33] in claiming completeness as an algebraic language definition.

There is clearly a need to publish a formal semantics for $\pi o\beta\lambda$ and a justification of *inter alia* the equivalence rules used in Section 2. This—with an emphasis on the π-calculus [30]—has in fact been the focus of much of the research over the last two years and will be reported elsewhere. What is currently available (to be published in the proceedings of the Schloß Dagstuhl workshop on 'Object-Orientation with Parallelism and Persistence', April 1995) is a Structured Operational Semantics definition of $\pi o\beta\lambda$ and proofs of the general equivalences. David Walker has published [36, 37] proofs based on the π-calculus of specific instances of the equivalences.

As well as the semantics and proofs of the equivalences, it will be necessary to proceed to a similar exercise on the rely/guarantee-conditions. These can be modelled on the related work on non-object-oriented languages (most recently [10]).

It is also important to investigate the expressive power of $\pi o\beta\lambda$ and there are experiments already under way in this direction. It has already been seen useful to add guards to methods which control when they are available for invocation and a form of exception mechanism to signal to the client that the server is not available for a particular form of method invocation. In this area, the author wishes to investigate the connections with [26] of which he has only recently become aware.

Another area where further work is clearly necessary is bringing ideas of progress arguments and fairness into the $\pi o\beta\lambda$ development method. It is also interesting to note that the sieve of Section 3 would be closer to that of Eratosthenes were each *Rem* to begin with

```
if $\neg(sr.lu(i)).test()$then skip else ...
```

Specifying the expectation that the absence of i guarantees eventual removal of its multiples is not possible with the current guarantee-conditions.

In general, the author is well aware that all development methods face the challenge of 'scaling up' and the reader is asked to remember that this effort is seen as part of a much larger programme in which a general attack on interference has been begun using concepts from object-oriented languages but much more remains to be done. Tool support is also an important issue for future research because of the need to manipulate multi-part specifications and control the consistency of proofs with versions of such specifications.

Acknowledgments

Input to the material in [22, 23] is acknowledged in those papers; the $\pi o\beta\lambda$ work is being pursued in collaboration with Steve Hodges and Haruo Yamaguchi whose comments are gratefully acknowledged. Pierre Collette and Kohei Honda provided thorough reviews of a draft of this paper. Rustan Leino and two anonymous referees provided detailed comments on the penultimate draft. The authors thanks go to these scientists for giving their time.

The author's research is supported by grants from EPSRC and the exchange scheme funded by the Royal Society, British Council and JSPS. The final version of this paper was produced whilst the author was at the Newton Institute for Mathematical Sciences (Cambridge)—their hospitality and that of Gonville and Caius College is gratefully acknowledged.

References

1. P. America, "Issues in the design of a parallel object-oriented language," *Formal Aspects of Computing*, Vol. 1, No. 4, 1989.
2. P. America, "Formal techniques for parallel object-oriented languages," in (Baeten and Groote, 1991), pp. 1–17, 1991.
3. P. America and J. Rutten, "A parallel object-oriented language: Design and semantic foundations," PhD thesis, Free University of Amsterdam, 1989.
4. R.J.R. Back, "Refinement calculus, part II: Parallel and reactive systems," in (de Bakker et al., 1990), pp. 67–93, 1989.

5. J.A. Bergstra and L.M.G. Feijs (Eds.), "Algebraic methods II: Theory tools and applications," Vol. 490 of *Lecture Notes in Computer Science*, Springer-Verlag, 1991.
6. J.C.M. Baeten and J.F. Groote (Eds.), "CONCUR'91—Proceedings of the 2nd International Conference on concurrency theory," Vol. 527 of *Lecture Notes in Computer Science*, Springer-Verlag, 1991.
7. L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson, "Modula-3 report," Technical Report 31, DEC Systems Research Center, Palo Alto, California, April 1988.
8. K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.
9. P. Collette, "Composition of assumption-commitment specifications in a UNITY style," *Science of Computer Programming*, Vol. 23, pp. 107–126, 1994.
10. P. Collette, "Design of compositional proof systems based on assumption-commitment specifications—Application to UNITY," PhD thesis, Louvain-la-Neuve, June 1994.
11. F.S. de Boer, "Reasoning about dynamically evolving process structure," PhD thesis, Free University of Amsterdam, 1991.
12. J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (Eds.), "Stepwise refinement of distributed systems," Vol. 430 of *Lecture Notes in Computer Science*, Springer-Verlag, 1990.
13. C.C. de Figueiredo, "A proof system for a sequential object-based language," PhD thesis, University of Manchester, August 1994.
14. J.V. Guttag and J.J. Horning, "Larch: Languages and tools for formal specification," *Texts and Monographs in Computer Science*, Springer-Verlag, 1993, ISBN 0-387-94006-5/ISBN 3-540-94006-5.
15. M.-C. Gaudel and J.-P. Jouannaud (Eds.), "TAPSOFT'93: Theory and practice of software development," Vol. 668 of *Lecture Notes in Computer Science*, Springer-Verlag, 1993.
16. 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*, Vol. 30, No. 8, pp. 672–687, 1987, see Corrigenda in *Communications of the ACM*, Vol. 30, No. 9, p. 770.
17. C.A.R. Hoare, "Recursive data structures," *International Journal of Computer & Information Sciences*, Vol. 4, No. 2, pp. 105–132, 1975.
18. J. Hogg, "Islands: Aliasing protection in object-oriented languages," in (Paepcke, 1991), 1991.
19. C.B. Jones, "Development methods for computer programs including a notion of interference," PhD thesis, Oxford University, June 1981, Printed as: Programming Research Group, Technical Monograph 25.
20. C.B. Jones, "Specification and design of (parallel) programs," in *Proceedings of IFIP'83*, North-Holland, 1983, pp. 321–332.
21. C.B. Jones, *Systematic Software Development using VDM*, Prentice-Hall International, second edition, 1990, ISBN 0-13-880733-7.
22. C.B. Jones, "Constraining interference in an object-based design method," in (Gaudel and Jouannand, 1993), pp. 136–150, 1993.
23. C.B. Jones, "Reasoning about interference in an object-based design method," in (Woodcock and Larsen, 1993), pp. 1–18, 1993.
24. W. Janssen, M. Poel, and J. Zwiers, "Action systems and action refinement in the development of parallel systems," in (Baeten and Groote, 1991), pp. 298–316, 1991.
25. 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, September 1991.
26. K.R.M. Leino, "Toward reliable modular programs," PhD thesis, California Institute of Technology, 1995.
27. C. Lengauer, "A methodology for programming with concurrency," PhD thesis, Computer Systems Research Group, University of Toronto, 1982.
28. R.J. Lipton, "Reduction: A method of proving properties of parallel programs," *Communications of the ACM*, Vol. 12, pp. 717–721, 1975.
29. B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988.
30. R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes," *Information and Computation*, Vol. 100, pp. 1–77, 1992.
31. E.-R. Olderog and K.R. Apt, "Using transformations to verify parallel programs," in (Bergstra and Feijs, 1991), pp. 55–82, 1991.
32. A. Paepcke (Ed.), *OOPSLA'91*, ACM, ACM Press, November 1991.

33. A.W. Roscoe and C.A.R. Hoare, "Laws of occam programming," Monograph PRG-53, Oxford University Computing Laboratory, Programming Research Group, February 1986.
34. K. Stølen, "Development of parallel programs on shared data-structures," PhD thesis, Manchester University, 1990, available as UMCS-91-1-1.
35. K. Stølen, "A method for the development of totally correct shared-state parallel programs," in (Baeten and Groote, 1991), pp. 510–525, 1991.
36. D. Walker, "Process calculus and parallel object-oriented programming languages," in International Summer Institute on Parallel Computer Architectures, Languages, and Algorithms, Prague, 1993.
37. D. Walker, "Confluence of processes and systems of objects," July 1995, CAAP'95.
38. J.C.P. Woodcock and P.G. Larsen (Eds.), "FME'93: Industrial-strength formal methods," Vol. 670 of *Lecture Notes in Computer Science*, Springer-Verlag, 1993.
39. Q. Xu and J. He, "A theory of state-based parallel programming by refinement: Part I," in J. Morris (Ed.), *Proceedings of The Fourth BCS-FACS Refinement Workshop*, Springer-Verlag, 1991.
40. Q. Xu, "A theory of State-based parallel programming," PhD thesis, Oxford University, 1992.
41. A. Yonezawa (Ed.), *ABCL: An Object-Oriented Concurrent System*, MIT Press, 1990, ISBN 0-262-24029-7.