Langauge Design Issues in Supporting Process-Oriented Computation in Persistent Environments

R.Morrison, C.J.Barter[++], A.L.Brown, R.Carrick,

R.Connor, A.Dearle, A.J.Hurst[+] & M.J.Livesey

Department of Computational Science, University of St Andrews, North Haugh, St Andrews, Scotland KY16 9SS. Tel 0334 76161. e-mail : ron%uk.ac.st-and.cs@ukc

[+] Department of Computer Science, Australian National University, G.P.O. Box 4, Canberra, ACT 2601, Australia. Tel 062 49 5111

[++] Department of Computer Science, University of Adelaide, Box 498, G.P.O., Adelaide, South Australia 5001. Tel 08 228 5333

Abstract

The problems of shared access to large bodies of information raise difficulties in the understanding and semantics of concurrency, distribution and stability. When the information is held in a persistent object store, these difficulties of understanding are extended to the interaction of the concepts of persistence and store with those above. Some of the difficulty is in deciding at what level the architecture or language operates, be it hardware or software. Other difficulties arise in the complexity of the problems of concurrency.

In this paper we identify these difficulties and clarify them with regard to database programming language design. We propose a model of concurrency that may be used as a solution to the problems. It is based on the rendezvous of Ada and integrated with a polymorphic type system with inheritance. This yields a process-oriented approach to system construction that has much in common with the object-oriented approach. We will demonstrate the facilities of the language Napier which allows the integration of these two methodologies along with a persistent environment to provide concurrently accessed object-oriented databases.

1. Introduction

In our attempts to design and build a persistent information space architecture (PISA) [1] we have identified a number of interacting and sometimes conflicting problems with regard to persistence, stores, concurrency, distribution, transactions and stability. Some of the difficulty is in deciding at what level the persistent architecture operates, be it a hardware or software architecture. Other difficulties arise in the complexity of the problems of concurrency. Our goal is to identify these difficulties, clarify them and to design a programming language based on our new understanding. Here we report on our work in designing such a language, often called a database programming language. In particular we identify the issues of locality and acces through context, process creation and the independence of process from procedures and demonstrate how a polymorphic type system with inheritance may be used along with a process type to integrate the notions of object-oriented and processes-oriented programming. In a persistent environment this allows the programming of object-oriented databases.

2. A Persistent Information Space Architecture

2.1 Persistence

The persistence of data has been defined to be the length of time for which the data exists and is useable[2]. In a persistent system the use of all data is independent of its persistence. Here we extend this notion of persistence to abstract over all the physical attributes of data, for example where data is kept, how long it is kept and in what form it is kept. The advantage of persistence is that it eliminates the artifical differences between programming language models of data and environment models of data. A persistent system will automatically control the movement of data to where it is required and the programmer is therefore freed from the burden of organising this movement. This results is a significant saving in the amount of code to be written and maintained in an application. It also has intellectual benefits in that the

programmer is presented with one mapping of the data from the real world to the data model instead of the three mappings amongst the real world, the program model and the environment model. The final advantage of persistent systems is that they can present a uniform protection mechanism, such as a type system, so often lacking between programming languages and their environment. The advantages of persistence are discussed at greater length elsewhere [3,15,16] and will not labour them further here. It is sufficient to say that by ensuring that all data objects are persistent and that the persistence of data is invisible to the programmer, then this level of abstraction yields powerful software engineering gains in the life of large systems[2].

At this level of abstraction all physical properties of the data are invisible to the user since persistence is a concept that abstracts over them. It is important to distinguish between the conceptual and physical problems of building and using persistent spaces in order to identify the areas on which we need to concentrate to achieve true persistence. In the following discussion we will pursue this theme, separating the conceptual or logical properties of the persistent information space from the physical ones. This is not always an easy task especially with regard to concurrency.

We wish to build a total system capable of providing for all programming activity. Our traditional view of the persistent information space is that it will subsume the functions of a plethora of mechanisms currently supported by components such as command languages, editors, file systems, compilers and interpreters, linkage editors and binders, debuggers, DBMS sublanguages and graphics libraries[1]. The information space is composed of objects, which may be simple or highly structured, defined by the universe of discourse of the type system of the PISA architecture. The major conceptual requirement is that the information space must be unbounded. That is, the user has the facility to create persistent objects forever. Most modern programming languages, database systems and operating

systems provide this facility and if we are to unify these mechanisms then the persistent space must also provide it.

Our previous work[2] has shown that one of the main difficulties in using an unbounded space is in remembering an unbounded number of names. Traditional solutions to this problem have included block structure in programming languages, hierarchal file directories in operating systems and data dictionaries in database systems. None of these solutions is totally satisfactory and often other mechanisms, such as those for module construction to augment block structure in programming languages, have arisen. Mechanisms for controlling and using this unbounded name space must be made available to the user.

We define context to be the manner in which the persistent information space controls the unbounded number of names but will defer discussion of how context may be defined in the architecture language until later. The important point is that the unbounded name space and the ability to impose context are identified as conceptual requirements of the persistent information space.

2.2 Stores

In all our work so far the persistent information space has been built in the form of a store[5,7] for which we have identified the following desirable properties:

a an unbounded capacity to store objects

b infinite speed

c stability

There are, of course, a number of problems in building and using a persistent store with the properties above. The conceptual problems of persistent stores arise out of how the store is used and the physical problems arise from the engineering difficulties in building the store. We will look at each of the requirements on the persistent store given above to identify these conceptual and physical problems.

2.2.1 Unbounded capacity to store objects

The need for unbounded capacity comes from the unbounded nature of the modelling performed by the programmer in the persistent store. The conceptual problems of using an unbounded store are the same as for an unbounded persistent space. That is, we must contend with an unbounded number of storage identifiers.

Of course, we cannot build a store with an infinite capacity. Any attempt to give the illusion of such a store will be built out of components, which may be of disparate technologies, and may include software technologies such as stacks and garbage collectors to reclaim unused space. This is, however, an engineering constraint and therefore a physical view of the store although it does introduce the notions of locality and distribution which may not be visible at the persistent space level.

A data object can be resident in a local store or in a distributed store. Two data objects reside in the same locality if they live in the same physical store and are distributed from one another otherwise. By combining locality and distribution we can compose very large stores. Note, however, that locality and context are not necessarily equivalent since a context can easily spread over a number of localities or indeed a locality can contain many whole or partial contexts. Only when an object in the contextual name space is mapped onto a locality in the physical store do context and locality coincide. Context is a property of the conceptual persistent space whereas locality and distribution are properties of the physical store.

In object-oriented programming languages, context and locality are often identical. Indeed it is this simplification of the universe that gives such systems much of their appeal.

2.2.2 Infinite speed

An advantage of an infinitely fast store is that it can be operated sequentially since there is no speed advantage in operating it concurrently. The programmer need

never take account of where information is stored and multiple copies of objects, for speed trade offs, are unnecessary since there is no speed advantage to moving the object closer to its point of use. Therefore the object may always reside in the one place. It follows that the notion of locality and distribution are unnecessary with infinitely fast stores. We note, however, that infinite speed does not solve any of our conceptual problems, only technological ones and still leaves difficulties in implementing such concepts as atomic transactions.

Again we cannot build this infinitely fast store. The only way we know of approaching this is to duplicate components and make them operate concurrently. This again gives rise to locality and distribution but also adds a new dimension at the physical level - that of concurrency. This form of concurrency, for speed advantage, is an engineering decision and not fundamental to the operation of the persistent store. That is, this kind of concurrency is introduced to speed up the store and not to solve conceptual problems.

We will argue later that there is a second need for concurrency which does effect the conceptual persistent information space.

2.2.3 Stability

All users of the persistent store would wish it to be stable. That is, in the event of a system failure no data will be lost. The mechanism ensures that data is always kept (or copied) on non volatile storage devices. This, of course, is only an illusion since no system can guarantee that even the non volatile devices are free from corruption by malice or error. Usually, however, an adequate level of reliability can be provided for any system.

Stability is a property of the physical store medium and is therefore a physical property of the information space and hidden from the user. In some systems stability and transactions are synonymous leading to some confusion of how the concept of stability arose. We will see in the next section how to separate the two concepts.

From the above discussion we contend that the main conceptual problem that we have in using the persistent information space is that of context. That is, how is the unbounded name space partitioned in order that we can master the complexity of a potentially unbounded number of names? The physical problems centre around how to build an infinite stable store. The issues of locality and distribution, that is where an object lives, allow us to simulate unbounded capacity out of smaller components. At this level, concurrency allows us to simulate higher speed out of slower components and stability can be simulated by a number of techniques such as incremental and total dumping.

2.3 Concurrency

We have argued above that a major motivation for concurrent activity is execution speed. The need for concurrency increases as machines approach their theoretical speed limit at the same time as the complexity of the applications becomes great enough to require even greater power.

There is, however, a second major need for concurrency. Many of the activities that we wish to model are inherently parallel. One of the major breakthroughs in the design and understanding of operating systems was gained by modelling the system as a set of co-operating sequential processes[8]. Since most of the early operating systems modelled in this manner ran on uni-processor machines this modelling was not done to simulate infinite speed. It was done to simplify the complexity of the system being built in order to gain greater insight into its operation. This method of modelling, first applied to operating systems, has now been applied to database systems, graphics systems and general problems in computer science[11]. It yields a new style of program construction and understanding and therefore can no longer be regarded as a physical property of the store.

In the process-oriented methodology the system is decomposed into activities that operate in parallel. Each activity is self contained, communicating with other activities via messages which are synchronised by some protocol. This style of

programming is very similar to the object-oriented style except that it does not usually make use of inheritance mechanisms. For this reason we say that it is more akin to a data abstraction methodology than an object-oriented one.

In order to accommodate this wish to model using concurrency, a host of languages have been invented or proposed that include the notion as part of their universe of discourse. Thus concurrency in the persistent information space sense is not a conceptual requirement of the space but of the manner in which we wish to model. This is equivalent to deciding that we wish to use other data objects such as arrays or functions to model with. However, since we wish to unify the activity of operating systems and database systems with our information space it would be wise to have concurrency as a conceptual requirement of the language or languages supporting the persistent space.

We therefore make concurrency a conceptual requirement of the PISA architecture rather than the information space itself which we traditionally view as figure 1.

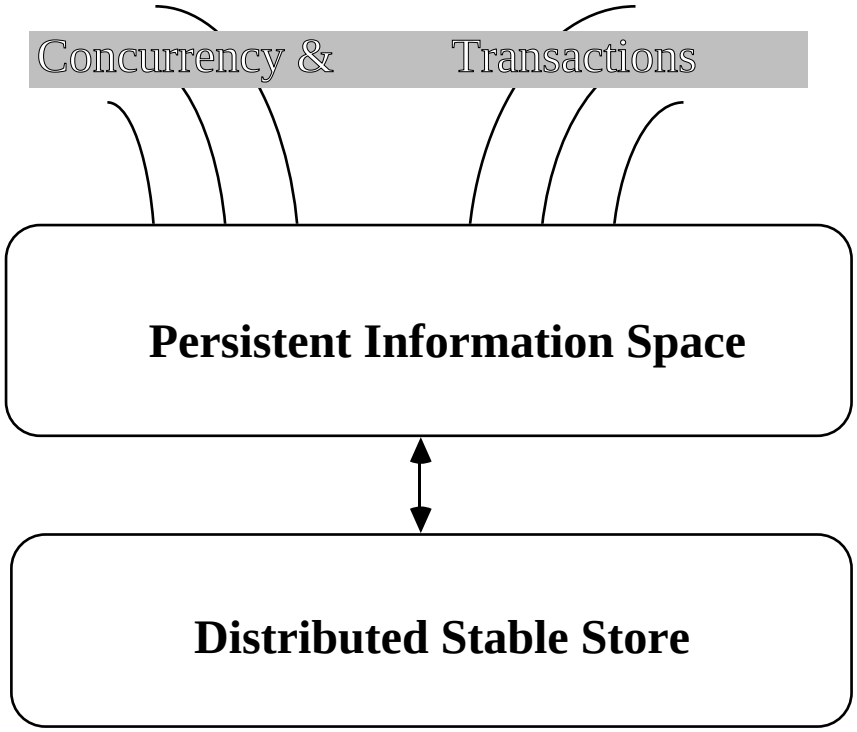

Figure 1 A Persistent Information Space Architecture

The persistent space is composed of objects defined by the universe of discourse of the PISA language. For the present the only requirement that we have of that language is that it must support concurrent computation, an unbounded name space and a context mechanism. At this level the programmer has no notion of where the data resides, be it locally, on disk or on a remote processor, how long the data may be kept or in what form it may be stored. An important consequence of this is that true persistence does not allow distribution to be seen at the conceptual level.

At a lower level the information space is supported by a stable store. This store may be distributed over many storage devices and machines and is likely to be built out of many disparate technologies both software and hardware. Thus locality and distribution are appropriate concepts here.

We subscribe to the view that it is premature to build mechanisms for atomic transactions into the low level stable store [1,9,10,14]. The stable store provides a very primitive form of transaction that allows the system to recover from unexpected errors, be they hardware or software. Thus it provides primitive (one level) recovery but not atomicity or general (multi process) reversability. There is little agreement on an appropriate generalised transaction mechanism and for the present we deem it safer to build sophisticated mechanisms for atomic transactions, at a higher level of abstraction, using the concurrency and stability primitives.

Users of the persistent information space access data via concurrency and transaction protocols. This is done by encapsulating the data in concurrency and/or transaction abstractions. The focus of this paper is to look at how concurrency may be integrated with the persistent information space. In particular what primitives are sufficient to support concurrency in persistent languages and how are they integrated with a polymorphic type system and a persistent store to build the abstraction mechanisms described above?

3. Concurrency Models

In designing a concurrent language for use in programming the persistent store we have two conceptual problems. The first is how to impose context on the unbounded name space. The second is how to specify concurrent activity, that is separate threads of control.

There are many different styles of concurrency in modern programming languages. The applicative languages such as SASL[18] have implicit concurrency due to the fact that they are referentially transparent. This style of concurrency is transparent to the user and will merely add speed to the execution of the programs. It is difficult to see how the applicative languages can make full use of a persistent store since the store would have to remain static to ensure referential transparency.

Store semantics languages have themselves split into two paradigms. The first is a shared store semantics where the whole store is available for use by all customers. The use must be synchronised to avoid indeterminate results. The shared store model roughly characterises a multiprocessor system where many processors share the same store.

The second paradigm is that of message passing where independent tasks have their own store and communicate with other tasks by sending messages to them. The message passing paradigm roughly models a distributed system where separate processors, perhaps with their own local store, communicate over a communications channel.

It is generally accepted that where large amounts of communication are required then the shared store model is more efficient in speed since local store is usually faster than a communications channel. However, where large amounts of computation are performed between communications then the message passing model may be appropriate.

Our dilemma should now be apparent. For persistence we wish to have an unbounded name space upon which we can impose some context mechanism. Thus the shared store model would seem more appropriate. On the other hand we know that this unbounded information space will be constructed out of components and it

would seem sensible to build this in from the beginning to allow for expansion. For this the message passing model is more appropriate.

The answer is to have a model that will allow the programmer the freedom to choose.

3.1 Napier Model

We have proposed and built the language Napier [17] in which the persistent store may be regarded as an unbounded collection of objects, each one sharable among the active processes in the system. We will use this language to demonstrate the concepts necessary to allow polymorphic, persistent processes.

In Napier, all data is persistent. That is, data is kept for as long as it is useable. This can be determined from the fact that it is reachable by the computation of the transitive closure of objects from the persistence root, called PS. When a process terminates all its data objects may be destroyed except those that the process has arranged to be reachable from PS. It should be noted that the persistent store may, in general, be a graph since it is a generalised data structure and it may be distributed over many machines. Given such a model of the information space we must define mechanisms for context and concurrency.

3.1.1 Context

Context is controlled by two methods in Napier, one static and one dynamic. The block structure of the language allows objects to be hidden to the outside world. Thus within the context of the block the name has a unique interpretation. Block structure forms a tree of contexts which may be used to segment the unbounded information space. The persistent space, however, is a graph and not necessarily a tree and that would suggest that block structure is not powerful enough to model all the required contexts.

In most programming languages it is recognised that block structure is not sufficient for all our modelling needs. In languages with higher order functions, such

as Napier, we can overcome this deficiency by another method. For example, we could write a block that returned a procedure which held within its closure a hidden object. A counter is a good example of this.

```
let successor = begin
                    let next := 0
                    proc ( -> int)
                    begin
                        next := next + 1
                        next
                    end
                end
```

The block expression, when executed, returns a procedure which contains the variable 'next' in its closure. This procedure is declared with the name 'successor'. 'next' is initialised in the block expression. On exit from the block the object 'next' cannot be destroyed as it will be required if the procedure 'successor' is called. Each call of 'successor' yields the next integer in the sequence. The procedure does this by remembering, in its closure, the last value it returned, in 'next', and calculating the new value from it. Languages with such semantics are called block retention languages and any block structured language with higher order functions falls into this category. By adding some concurrency control to the system we could ensure mutual exclusion on the access to 'next' and extend the block to a monitor[12]. However, although powerful, the combination would be difficult to use without more syntactic support.

Although the above method allows users to dynamically create and manipulate contexts, the technique is essentially static since the scope of the objects must be defined by the programmer and may never be changed. A second method of context control is provided in Napier. The technique is similar to block structure except that we are allowed to dynamically nest the blocks. To do this the data type environment is used.

Objects of type environment are collections of bindings, that is name-value pairs. The distinguished point of the persistence graph, PS is of type **env**. Objects of

type **env** belong to the infinite union of all labelled cross products of typed name-value pairs.Environments are designed to perform two roles. The first is to act as a context mechanism in the form of a collection of bindings, like a block. All environments have the same union type and are assignment compatible. On dynamic projection the particular environment is determined to have a specific type. This allows the mesting of environments to be constructed dynamically.

The second major use for environments is as an extensibility mechanism to control change in the system. Bindings may be added and subtracted dynamically to allow this. Environments are best demonstrated by example. We will create an environment that contains a counter and two procedures, one to increment the counter and one to decrement it. This may be done by

```
let e = environment ()
in e let count := 0
!we have now placed the binding count : int = 0 in the environment e
use e with count : int in
begin
        in e let add = proc (n : int -> int) ; {count := count + n ; count}
        in e let subtract = proc (n : int -> int) ; {count := count - n ; count}
end
!the environment now has three bindings, count, add and subtract
```

The **use** clause binds an environment and its field names to the clause following the **in**. In the above the name 'count' is available in the block as if it had been declared in the immediate enclosing block. The binding occurs at run time since in general the environment value, 'e', may be any expression evaluating to an environment. The binding is therefore dynamic and is similar to projection out of a union. The difference is that here we only require a partial match on the fields and other fields not mentioned in the **use** clause are invisible in the qualified clause and may not be used.

The environment mechanism provides a contextual naming scheme that can be composed dynamically. The **use** clauses can be nested and the environments involved calculated dynamically and therefore the name bindings can be constructed dynamically. This does not yield full dynamic scoping in the Lisp sense since all the

objects in the individual environments are statically bound. The technique complements the block structure in the language and completes the context mechanisms required for persistent information spaces.

For information to outlive the process it must be reachable from the distinguished root, PS. In the above case we could do this by adding e to PS by

in PS **let** ee = e

!add the binding ee : **env** = e to the environment PS

and to retrieve it again we could use

use PS **with** ee : **env in** ...

Having defined a mechanism for the conceptual problem of context we must now define one for concurrency.

3.1.2 Concurrency

The model of concurrency in Napier is based on CSP [11] and Ada [13]. The choice of this model reflects our own preferences in concurrency. The system should be regarded as an exampler of how to integrate persistence and concurrency and not to determine the sole model of concurrency.

Process is a type in the language. The type defines the process interface that the process presents to the world. The interface consists of a set of procedures (names) called entries. External to the process the entries act like first-class procedures. Inside the process the entries do not act like procedures but are used to establish rendezvous with calling processes. The syntax of process type is

<process_type> :: = **process** ([< entry_list>])

<entry_list> :: = <identifier_list> : < proc_type> [;<entry_list>]

For example, the counter given earlier may be extended to a safely updated counter by specifying the type

type shared_counter **is process** (add, subtract : **proc** (int ->int))

and creating an instance of the type. Outside the process, entries act like first class procedures and may be called. For example, if an object of type 'shared_counter' with the name 'this_counter' existed we could communicate with it by

this_counter (add) (3)

which would add 3 to the value and yield the new total. Thus the entries are typed as procedures outside the process to which they belong. The parameter list is evaluated in the calling process, before the rendezvous takes place, which is suspended until the rendezvous is complete.

Within a process the entry name is used to establish a rendezvous. The **receive** clause uses the entry name, its formal parameter list and a clause to be executed during the rendezvous. The syntax is

<entry_clause> ::= **receive** <identifier> ([<parameter_list>]) **do** <clause>

For example

```
receive add (n : int -> int) do
begin
        count := count + n
        count
end
```

When a rendezvous is established the body of the particular entry is executed. During this time the caller is suspended. If, however, a **receive** clause is executed

before it has been called then the callee is suspended until the entry is called. Each **receive** clause defines a body for the entry. Thus each entry name may have many bodies in a process.

Processes are created and activated by associating a process body (a void clause) with a process name. The syntax is

<clause> :: = <identifier> **with** <clause>

For example

```
let this_counter = shared_counter with
        while true do
        begin
                receive add (n : int -> int) do
                begin
                        count := count + n
                        count
                end
                receive subtract (n : int -> int) do
                begin
                        count := count - n
                        count
                end
        end
```

The process once created starts to execute immediately and in parallel with the main body which created it. A name may be given to the process by declaring it, as above.

In the example, the process 'this_counter' will loop servicing requests for adding and subtracting in strict order.

To implement the rendezvous there is a separate queue of waiting processes for every entry. These queues are serviced on a first-come first-served basis. Non-determinism in the system is provided by a **select** clause. The syntax is

<select_clause>::=**select**

<guarded_command>{;<guarded_command>}***selected**

<guarded_command> ::= [<guard>] : <guard_clause>

<guard_clause> ::= <entry_clause> | <clause>

<guard> ::= <boolean_clause>

For example

```
select
occupancy < 4       : receive enter () do occupancy := occupancy + 1
                    : receive exit () do occupancy := occupancy - 1          selected
```

A guarded command is open if it does not contain a guard or the boolean expression is **true**. Otherwise it is closed. To execute the **select** clause all the guards are evaluated in order. This determines the open guarded commands. One of these is chosen for execution non-derministically, subject to the constraint that if the guard clause is an entry clause it will only be chosen if the entry can be received immediately. If none of the guarded commands can be immediately executed the process waits until one can be.

A process becomes completed when it finishes the execution of its body. If an entry call is made to a completed process an exception is raised in the calling process. The same is true if the called task terminates while the caller is on an entry queue. If an exception is raised during a rendezvous it is propagated in both processes. Finally if the caller completes before the callee completes, the callee will complete its part of the rendezvous.

Two processes are equal only if they are the same process. They have the same type if they have the same entries.

We can now use a shared object example to see how it may be programmed in Napier.

```
type shared_int_object is process (write : proc (int), read : proc ( -> int) )

let int_object = shared_int_object with
        begin
                let i := 0
                while true do
                        select
```

```
                    :       receive write (val : int) do i := val
                    :       receive read ( -> int) do i
                selected
        end     !create a new process running in parallel
```

Thus 'int_object' is a handle on a process of type 'shared_int_object' which is executing in parallel with the rest of the system. The process will loop forever receiving requests to 'write' and 'read' in any order. The process itself will ensure mutual exclusion of multiple calls.

For convenience we can rename the entry procedures outside the process. For example

```
let Write = int_object (write) ; let Read = int_object (read)
```

and we can pass the procedures 'Write' and 'Read' to other components of the system. We may wish to establish a rendezvous by calling the procedure. For example

```
Write (2)
```

Processes are also first class data objects and may be generated as the result of a procedure. For example

```
let int_object_generator = proc ( init : int -> shared_int_object)
        shared_int_object with
        begin
            let i := init
            while true do
                select
                    :       receive write (val : int) do i := val
                    :       receive read ( -> int) do i
                selected
        end

let this_object = int_object_generator (2)
let that_object = int_object_generator (-42)
```

creates two processes running in parallel with the main stream. These two processes both have the same type 'shared_counter' and may be used where objects of this type are required.

In the following example we demonstrate that by integrating the process concept with the polymorphic type system, we can define polymorphic process generators.

```
type shared_object [t] is process (write : proc (t), read : proc ( -> t) )

let object_generator = proc [t] ( init : t -> shared_object [t])
                shared_object [t] with
                begin
                        let i := init
                        while true do
                                select
                                        :       receive write (val : t) do i := val
                                        :       receive read ( -> t) do i
                                selected
                end

let int_object = object_generator [int] (3)
let string_object = object_generator [string] ("Ronald")

int_object (write) (-4213)
```

In this example a polymorphic procedure 'object_generator' has been used to generate processes of a particular type. The procedure is a generic form quantified by the type 't'. It takes as a parameter an object of the type 't' and returns an active process of type 'shared_object' parameterised by 't'. The procedure must be given a particular type and an object of that type to operate correctly. 'int_object' has the type shared_object [int] and 'string_object' the type shared_object [string] which are not the same. However, the entry procedures manipulate integers and strings respectively and may be used where procedures of these types are appropriate.

It should be noted that these generators and the processes themselves are more powerful than the generics and tasks of Ada. In Napier. the processes are first class objects and may be passed around or stored in a complex data structure in the persistent environment. This is not possible in Ada.

3.1.3 Locality and Distribution

So far all the processes that we have described are lightweight in that they share the address space of their creators. It so happens that in the 'shared_counter' example, the processes do not use any free variables and create their own environment, thus making it look heavyweight in nature. This we can use to our advantage to give both lightweight and heavyweight processes in the persistent store.

Our ideal model of the persistent architecture is an unbounded space fragmented by the context mechanism. We know, however, that the system will be built out of localities and we can accommodate this by arranging that localities are always controlled by one context. By making the environment our context mechanism, we can achieve the correct mixture of bindings necessary to support a distributed system.

To support the distribution mechanism we require at least two procedures to be built into the system. They are

let copy = **proc** [t] (item : t -> t)

let copy_to_env = **proc** [t] (environment : env ; N : name[t] ; item : t)

The 'copy' procedure makes a copy of the object. That is, it copies the transitive closure of the object to ensure that it will work correctly. The 'copy_to_env' procedure makes a copy of the object using the copy procedure and moves it into the same locality as the environment placing a new binding in the environment. The new binding is the name 'N' to the copy of the value 'item'.

The 'copy_to_env' procedure is essentially the bootstrap mechanism for a new locality being added to the persistent store. Initially there is one distinguished point PS. To add a new locality, a binding is placed in PS or any environment reachable from PS, so that it may be seen by all users. Binding to this new environment, and thus locality, is performed by the environment binding mechanism described above.

The placing of the new locality in the environment is equivalent to plugging a new component in to the system and must be performed by a low level implementation as it is not possible at this level of abstraction.

3.1.4 Stability

Each locality is stabilised independently in the system. When the standard procedure 'stabilise' is called the locality in which the processes is operating is stabilised. This may be done automatically or by the user. The 'stabilise' procedure calculates which objects are in the locality that it is going to operate on by computing the transitive closure of all the objects local to the environment. Cross locality pointers are ignored. The procedure is made available to users so that higher level transaction mechanisms may be built. Krablin[14] has shown how this may be done in the language CPS-algol.

3.1.5 Polymorphic persistent processes

In the following example we demonstrate that by integrating the process concept with the type system, we can define polymorphic persistent processes. The example is that of readers and writers accessing a shared database. The procedure that creates the database is given an initial value for the database, which is copied to remove any aliases, along with the database type. The procedure creates a process to control access to the database and returns the procedures 'read' and 'write', in a structure, which may be used to access the database in a controlled manner. Indeed since the database is always copied it is the only manner in which it may be accessed. The example can be extended to allow alteration to parts of the database but that is not relevant here. The algorithm is taken from Barnes [4] page 228.

```
let Readers_Writers = proc [t] (init : t -> structure (Read : proc ( -> t) ; Write : proc (t) ) )
begin
    type Control is process (start : proc (int), stop_read, write_it, stop_write : proc () )

    let control = Control with
```

```
        22
        begin
                let readers := 0 ; let writers := 0 ; let READ = 0 ; let WRITE = 1
                while true do
                        select
                                writers = 0   : receive start (service : int) do
                                                if service = READ                then readers
:= readers + 1
                                                                                 else writers := 1
                                              : receive stop_read () ; readers := readers - 1
                                readers = 0           : receive write_it ()
                                              : receive stop_write () ; writers := 0
                        selected
        end

        let item := copy (init)

        struct (        Read =      proc ( -> t)
                                    begin
                                        control (start) (READ)
                                        let result = copy (item)
                                        control (stop_read) ()
                                        result
                                    end ;
                        Write =     proc (X : t)
                                    begin
                                        control (start) (WRITE)
                                        control (write_it) ()
                                        item := copy (X)
                                        control (stop_write) ()
                                    end )
end

let synchronised_integer = Readers_Writers [int] (0)
let Read = synchronized_integer (Read) ; let Write = synchronized_integer (Write)
```

In this example, with the call Readers_Writers [int] (0), the user process creates a database which is a synchronised integer with initial value zero. The two database operations are then given local names 'Read' and 'Write', which will give controlled access to the integer.

While the process itself ensures mutual exclusion of access to its entries, the procedures 'Read' and 'Write' may be made available to any number of other (user) processes, to provide concurrent interfaces to the database. Because such procedures execute within the closure of their creator (in this case the procedure 'Readers_Writers'), care must be taken with access to shared variables in this closure;

in this example, only 'item' is shared, and access to it is controlled by the process 'Control'.

This example has been kept deliberately simple to illustrate the mechanism. To be more general, the interface would have to be extended to allow selective updating and access to a more general database. However, the example does illustrate how the concurrency abstraction mechanism need only be written once and then applied to any type of database. This has benefits in terms of software re-use and economics. A solution to a second example, the ring buffer problem, is given in Appendix I.

Processes, and procedures that generate processes, may be stored in the persistent store. Thus when the program that activates a process terminates, the process itself may remain active. As an example we will store the 'Readers_Writers' procedure and the 'int_object' process from previous examples in the persistent store and retrieve them later. Storing them in the persistent store may be done by

```
let e = environment ()
in e let R_W = Readers_Writers
in e let Int_object = int_object
in PS let ee = e
```

Thus the PS environment contains a binding ee : **env** and this environment contains the bindings R_W : **proc** [t] (init : t -> **structure** (Read : **proc** (-> t) ; Write : **proc** (t))) and Int_object : **process** (write : **proc** (int), read : **proc** (-> int)).

These objects may be retrieved from the persistent store by the following

```
type shared_int_object is process (write : proc (int), read : proc ( -> int) )

use PS with ee : env in
use ee with    R_W : proc [t] (init : t -> structure (Read : proc ( -> t) ; Write : proc
(t) ) ),
               Int_object : shared_int_object in
begin
       let first = Int_object (read) ()
       let synchronised_int = R_W [int] (first)
       ...
end
```

In this example, the integer that is encapsulated in the 'Int_object' process is used to initialise the database generated by the 'Readers_Writers' procedure, which is called 'R_W' here. It should be obvious that the process 'Int_object' is always active. Getting it out of the persistent store allows the current process to communicate with it.

3.1.6 Protocols and inheritance

The entry list for a process specifies its type and can be considered as the protocol through which it may be accessed. By utilising the multiple inheritance scheme of Cardelli [6] we can place process types in the type lattice and define a partial ordering of processes. Thus it is possible to define procedures that will operate on processes with at least a given defined protocol. If the process has a more specialised type then that may also be used. For example

```
type RON is process (add : proc (int -> int) )
type FRED is process (add, subtract : proc (int -> int) )

let Ric = proc [t ≤ RON] (A : t )
        begin
                .
                let q = A (add (2))
                .
        end

let ron = RON with ...                          ! create a process of type RON
Ric [RON] (ron)                                 ! pass it to the procedure Ric
let fred = FRED with ... ; Ric [FRED] (fred)
```

The procedure 'Ric' takes as a parameter an object of type 't' which is a process with at least an entry 'add' of the correct type. In the example, the procedure is called twice with a process parameter. The first 'ron' has exactly the entry 'add' whereas the second 'fred' has more. Inside the procedure, only the entry 'add' may be used. By using this subtype inheritance we can abstract over entry protocols that are common to processes.

Such an inheritance mechanism is important in object-oriented programming. Although the inheritance mechanism is available for all data types in Napier it is particularly important for processes since they are the main system construction type for self contained objects. By structuring the system into co-operating processes and using the other type constructors as data abstraction mechanisms we can impose an object-oriented methodology.

The inheritance mechanism is also important for controlling change in the system. For example, a process may be changed to give more entries in its interface without altering the procedures that work on at least the supertype. This allows dynamic change to the active processes in the system subject to the supertype constraints.

4. Conclusions

Wegner [19] has defined an object-oriented language to have three essential features. They are

a. the ability to define objects as a set of operations and a state that remembers the effect of the operations.

b. the objects can be categorised by class (type).

c. there is an inheritance mechanism for defining superclasses and subclasses.

In comparison with this, processes traditionally have a state and a method of communicating with each other. By making process a type in the modelling language and ensuring that the type system allows inheritance, then the methodologies can be conveniently integrated. To illustrate this, the table below compares the essential features of both styles.

Object-Oriented	**Process-Oriented**
objects	processes
object state	process state

object type	process type
object operations	process entries
object inheritance	process protocols

We have presented a model of a persistent information space which is composed of objects. The information space is unbounded in that the user has the capacity to create objects forever. A context mechanism is introduced in order to control the unbounded name space.

It is argued that at a persistent level of architecture, concepts such as locality, distribution and stability are physical properties of how we might build an unbounded information space. By using one of the contextual mechanisms of the system to incorporate locality we can provide all the necessary functionality by two procedures 'copy_to_env' and 'stabilise'.

The Napier system is multiparadigm in that we can construct processes to distinguish between locality and context or we can build using the object-oriented methodology which we call process-oriented.In this approach we use a strong type system to unify the concepts of process and inheritance. We allow static data constructors but use process as the main object constructor and integrate this with a persistent environment to yield concurrent object-oriented databases. It should be noted however that process-oriented computation still requires a contextual mechanism such as environments to control the name space.

5. Acknowledgements

This work was initiated during the study leave periods of Chris Barter and John Hurst taken in St Andrews. We are grateful to our collaborators in the PISA project, particularily Francis Wai and Malcolm Atkinson at Glasgow University who are also working on these problems and with whom we have shared many ideas. We would also like to thank Robin Stanton for his many comments and improvements on this paper. The work was supported by SERC grants GR/D 4326.6, GR/D 47790 and GR/D 8823.

6. References

[1]. Atkinson, M.P., Morrison, R. & Pratten, G.D.

Designing a persistent information space architecture. 10th IFIP World Congress, Dublin (September 1986),115-120. North-Holland, Amsterdam.

[2]. Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.

An approach to persistent programming. Computer Journal 26,4 (November 1983),360-3

[3]. Atkinson, M.P. & Morrison, R.

Procedures as persistent data objects. ACM.TOPLAS 7,4 (October 1985),539-559.

[4]. Barnes, J.G.P.

Programming in Ada. 2nd Edition. Addison-Wesley (1984).

[5]. Brown, A.L. & Cockshott, W.P.

The CPOMS reference manual. The Universities of Glasgow and St Andrews PPRR-13. (

[6]. Cardelli, L.

A semantics of multiple inheritance. In **Lecture Notes in Computer Science**. 173, 51-67. Springer-Verlag (1984).

[7]. Cockshott, W.P., Atkinson, M.P., Bailey, P.J., Chisholm, K.J. & Morrison, R. The persiste

[8]. Dijkstra, E.W.

The structure of THE multiprogramming system. Comm.ACM 11, 5 (May 1968), 341-346.

[9]. Fredrich, M. & Older, W.

HELIX : the architecture of a distributed file system. 4th Conf. on Distributed Computer S

[10]. Gammage, N.D., Kamel, R.F. & Casey, L.M.

Remote Rendezvous. Software, Practice & Experience 17, 10 (1987), 741-755.

[11]. Hoare, C.A.R.

Communicating sequential processes. Comm.ACM 21, 8 (August 1978), 666-677.

[12]. Hoare, C.A.R.

Monitors : an operating system structuring concept. Comm.ACM 17, 10 (1974), 549-557.

[13]. Ichbiah et al.,

The Programming Language Ada Reference Manual. ANSI/MIL-STD-1815A-1983. Also **Lecture Notes in Computer Science**. 155. Springer-Verlag (1983).

[14]. Krablin, G.L.

Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117.

[15]. Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P.

An integrated graphics programming environment. 4th UK Eurographics Conference

[16]. Morrison, R., Bailey, P.J., Brown, A.L., Dearle, A. & Atkinson, M.P.

A persistent store as an enabling technology for an integrated project support environmer

[17]. Morrison, R., Brown, A.L., Carrick, R., Connor, R. & Dearle, A.

Napier Reference Manual. Universities of Glasgow and St Andrews (1988).

[18]. Turner, D.A.

SASL language manual. University of St.Andrews CS/79/3 (1979).

[19]. Wegner, P. "Dimensions of object-based language design". OOPSLA 87 (1987), 168-182.