On the Integration of Object-Oriented and Process-Oriented Computation in Persistent Environments

R.Morrison, A.L.Brown, R.Carrick, R.Connor & A.Dearle

Department of Computational Science, University of St Andrews,
North Haugh, St Andrews, Scotland KY16 9SS

Abstract

Persistent programming is concerned with the construction of large and long lived systems of data [1,2]. Such systems have traditionally required concurrent access for two reasons. The first is that of speed, be it access speed for multiple users or execution speed for parallel activities. The second reason for concurrency is to control the complexity of large systems by decomposing them into parallel activities.

This process-oriented approach to system construction has much in common with the object-oriented approach. We will demonstrate, in this paper, the facilities of the language Napier [18] which allows the integration of the two methodologies along with a persistent environment to provide concurrently accessed object-oriented databases.

1 Introduction

One of the major motivations for concurrent activity, be it user or machine, 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 increases to require even greater power.

There is, however, a second major motivation for concurrency. Many of the activities that we wish to model are inherently parallel and if we wish to capture the essential nature of these real world activities then language primitives powerful enough to model them are required. 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[7]. Since most of the early operating systems modelled in this manner ran on uni-processor machines this modelling was not done to increase speed. It was done to simplify the complexity of the system being built in order to gain greater insight into its operation. This process-oriented method of modelling, first applied to operating systems, has now been applied to database systems, graphics systems and general problems in computer science[13].

The process-oriented methodology yields a new style of program construction and understanding in which 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.

The best known process-oriented methodology is that of actors [12] where there are only two kinds of object in the system's universe of discourse, actors and values. The attractiveness of the actor model is in its simplicity and most of the work on integrating the paradigms has been based on superimposing an actor model on an object-oriented language [11].

Here we will take a different approach and use a strong type system to unify the concepts of process and inheritance. We will 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.

2 Processes and Objects

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 th 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 will now demonstrate, by example, the facilities of the language Napier which make this integration possible.

3 Processes and Objects in Napier

In Napier, the type system is similar to one suggested by Cardelli and Wegner [6]. It contains a number of base types such as integer, real, string and boolean and a number of constructors like vectors, structures, first class procedures, abstract data types, environments and processes. The type system is polymorphic and we will introduce the relevent details as we progress through examples. The inheritance method is that of Cardelli [5].

The model of concurrency in Napier is based on CSP [13] and Ada [14]. 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. For example, an integer object to which we may read and write could be defined by specifying the following type

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

That is, objects (processes) of this type have two entries 'write' and 'read'. 'write' takes as a parameter the integer value to be written and 'read' returns the integer value contained in the process. 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. For example

receive write (n : int) **do** i := n

When a rendezvous is established the body of the particular entry is executed. In this case it assigns the parameter value 'n' to the variable 'i'. From the time the call is made until the completion of the rendezvous the caller is suspended. If, however, a **receive** clause is executed before its entry has been called then the callee is suspended until the entry is called. Each **receive** clause defines a body for the entry allowing entry names to have many bodies in a process.

Processes are created and activated by associating a process body (a void clause) with a process type. For example,

```
let int_object = shared_int_object with
     begin
               let i := 0
               while true do
               begin
                      receive write (val : int) do i := val
                      receive read ( -> int) do i
               end
     end       !create a new process running in parallel
```

will create a process of the type 'shared_int_object'. The process once created starts to execute immediately and in parallel with the process which created it. A name may be given to the process by declaring it, as above. In the example, the process 'int_object' will loop servicing requests for writing and reading in strict order. Thus it has the state embodied in its local variable 'i' and its operations are 'read' and 'write'.

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

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

To execute the **select** clause all the boolean expressions are evaluated in order. An option is open i it does not contain a boolean expression or the boolean expression is **true**. Otherwise it is closed. On of the open options is chosen for execution non-deterministically subject to the constraint that if th clause to be executed is an entry clause it will only be chosen if the entry can be received immediately If none of the options can be immediately executed, eg if there is no entry pending, the process wait until one can be.

4 Polymorphic Processes

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

```
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

! create the process objects
let int_object = object_generator [int] (3)
let string_object = object_generator [string] ("Ronald")

!send messages to the processes
int_object (write) (-4213) ; string_object (write) ("Ronald")
```

In this example, a polymorphic procedure 'object_generator' is 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. Thus 'int_object' has the type shared_object [int] and 'string_object' the type shared_object [string]. These are not the same type. However, the entry procedures manipulate integers and strings respectively and may be used where procedures of these types are appropriate.

Thus 'int_object' and 'string_object' are handles on processes executing in parallel with the rest of the system. The processes will loop forever receiving requests to 'write' and 'read' in any order. The processes themselves will ensure mutual exclusion of multiple calls.

This example has been deliberately kept simple to illustrate the process mechanism. More generally, the object 'i' represents a generalised database of any type and 'read' and 'write' represent the operations on the database. To be more useful, extra operations on the database would be required. However, the example does illustrate how the object concurrency abstraction mechanism need only be written once and then applied to any type of database. This has benefits in terms of software economics [4].

A second example of a generalised index from any ordered type to any type, modelled as a process is given in Appendix I. For the present, we will concentrate on the problem of the 5 dining philosophers [8] and model it using process objects. Before that, however, we must introduce a form of vector initialisation in Napier.

```
let square = proc (i : int -> int) ; i * i
```

defines a procedure that will return the square of its parameter value. A vector initialised by

```
let squares = vector 1 to 10 using square
```

would consist of ten elements indexed from 1 to 10. The elements themselves are initialised by calling the procedure with the index value and using the result as the element value. In this case each element will have a value that is the square of its index.

The solution to the dining philosophers problem is similar to the one proposed by Hoare [13]. In the system there are three types of objects, forks, philosophers and a room in which they dine. Each philosopher sits at a particular unique seat.

Forks have a very simple existence, being picked up and put down in order by one of two philosophers. The five forks modelled as a vector of processes can be defined by

```
type fork is process (pickup, putdown : proc () )

let fork_generator = proc (i : int -> fork)
  fork with
       while true do
       begin
              receive pickup () do {}
              receive putdown () do {}
       end

let forks = vector 0 to 4 using fork_generator
```

Thus, we now have five processes, forks (0) .. forks (4) executing in parallel. Notice that the forks will receive messages from anyone and it is therefore up to the philosophers not to abuse this.

The room has an equally simple existence. Philosophers may enter the room to eat and leave after eating. To avoid deadlock, but not starvation, at most four philosophers are allowed in the room at any one time. The room may be modelled by

```
type Room is process (enter, exit : proc () )
```

```
let room = Room with
  begin
       let occupancy := 0
       while true do
                select
                occupancy < 4      : receive enter () do occupancy := occupancy + 1
                                   : receive exit () do occupancy := occupancy - 1
                selected
  end
```

Philosophers enter the room, pick up the left hand and then the right hand fork, eat, put down the forks and leave the room. We must model each philosopher so that the philosopher picks up and puts down the correct forks only. The following will do this

```
type philosopher is process ()

let philosopher_generator = proc (i : int -> philosopher)
  philosopher with
       while true do
       begin
                room (enter) () ; forks (i, pickup) () ; forks ((i + 1) rem 5, pickup) ()
                forks (i, putdown) () ; forks ((i + 1) rem 5, putdown) () ; room (exit) ()
       end

let philosophers = vector 0 to 4 using philosopher_generator
```

Notice that within the closure of each philosopher there is the integer 'i' which is in effect the identity of the process and ensures that the correct forks are selected. Putting the fragments together yields the total solution.

5 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 [5] 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 shared_int_object is process (write : proc (int ), read : proc ( -> int) )
type write_shared_int_object is process (write : proc (int) )

let Write_object = proc [t ≤ write_shared_int_object] (A : t ; value : int)
                       A (write) (value)

let ron = write_shared_int_object with ...
     ! create a process of type write_shared_int_object

Write_object [write_shared_int_object ] (ron, -42)
     ! pass it to the procedure Write_object andwith the value to be written

let fred = shared_int_object with ... ; Write_object [shared_int_object ] (fred, 55)
```

The procedure 'Write_object' takes as a parameter an object of type 't' which is a process with at least the entry 'write'. In the example, the procedure is called twice with a process parameter. The first 'ron' has exactly the entry 'write' whereas the second 'fred' has more.

Inside the procedure, only the entry 'write' 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.

6 Persistence and Object-Oriented Databases

We have defined the persistence of data 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. We have discussed the advantages of persistence elsewhere [3,16,17] and will not labour them 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. The figure often quoted is 30% of the total cost of a system throughout its life cycle[2].

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, PS. When a program terminates all its data objects may be destroyed except those that it has arranged to be reachable from PS.

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 'object_generator' procedure and the 'int_object' process from previous examples in the persistent store and retrieve them later. We will assume that these objects have already been declared. Storing them in the persistent store may be done by

```
let e = environment ()              !create a new environment
in e let O_G = object_generator           !store the bindings O_G and
in e let Int_object = int_object    !Int_object in the environment e
in PS let ee = e                    !store the environment e in PS
```

The PS environment contains a binding e : **env** and the ee environment contains the bindings O_G : **proc** [t] (init : t -> shared_object [t]) and Int_object : shared_object [int].

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

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

use PS with ee : env in
use ee with   O_G : proc [t] (init : t -> shared_object [t]),
        Int_object : shared_int_object in
begin
  let first = Int_object (read) ()
  let synchronised_int = O_G [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 'object_generator' procedure, which is called 'O_G' 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.

The example demonstrates that it is possible to store data in the form of a process object in the persistent store. This gives a very primitive form of object-oriented database where the objects are processes that receive and send messages. More traditionally databases are concerned with the stability of data, for integrity, and transaction mechanisms for safe concurrent use.

For stability we provide a low level primitive 'stabilise' that ensures that data is copied onto a stable medium. However, we subscribe to the view that it is premature to build mechanisms for atomic transactions into the low level stable store [1,9,10,15]. 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 which may be provided as demonstrated above.

7 Conclusions

The purpose of this paper was to show how the language Napier has integrated the object-oriented and process-oriented design paradigms. This was achieved by illustrating the similarities in the paradigms. By using a type system that allows both inheritance and process as a type in its universe of discourse it was shown how to construct process objects in the object-oriented style.

Furthermore, the Napier system is persistent. That is, it allows all data objects, including processes to persist. Using this persistence, we have demonstrated a method for constructing concurrently accessed object-oriented databases.

8 Acknowledgements

A number of people have help in our understanding of the integration of type, inheritance, concurrency, transactions, stability and persistence. The most prominent of those are Chris Barter, John Hurst, Robin Stanton and Mike Livesey who are all due our thanks. We would also like thank our collaborators in the PISA project and in particular Malcolm Atkinson and Francis Wai who are also working on similar problems. The system is available on request. The work was supported by SERC grants GR/D 4326.6, GR/D 47790 and GR/D 8823.

9 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),360-365.
3. Atkinson, M.P. & Morrison, R. "Procedures as persistent data objects". ACM.TOPLAS 7,4 (October 1985), 539-559.
4. Boehm, B.W. "Understanding and controlling software costs". 10th IFIP World Congress, Dublin (September 1986), 703-714. North-Holland, Amsterdam.
5. Cardelli, L. "A semantics of multiple inheritance". In **Lecture Notes in Computer Science**. 173, 51-67. Springer-Verlag (1984).
6. Cardelli, L. & Wegner, P. "On understanding types, data abstraction and polymorphism". ACM.Computing Surveys 17, 4 (December 1985), 471-523.

7. Dijkstra, E.W. "The structure of THE multiprogramming system". Comm.ACM 11, 5 (May 1968), 341-346.
8. Dijkstra, E.W. "Cooperating sequential processes". In **Programming Languages** (editor F. Genuys). Academic Press, London. (1968), 43-112.
9. Fredrich, M. & Older, W. "HELIX : the architecture of a distributed file system". 4th Conf. on Distributed Computer Systems. (May 1984), 422-431.
10. Gammage, N.D., Kamel, R.F. & Casey, L.M. "Remote Rendezvous". Software, Practice & Experience 17, 10 (1987), 741-755.
11. Goldberg, A. & Robson, D. **SMALLTALK-80 The language and its implementation**. Addison-Wesley, London. (1983).
12. Hewitt, C.E. "Viewing control structures as patterns of message passing". J. Artificial Intelligence 8, 3 (June 1977), 323-364.
13. Hoare, C.A.R. "Communicating sequential processes". Comm.ACM 21, 8 (August 1978), 666-677.
14. Ichbiah et al., **The Programming Language Ada Reference Manual.** in **Lecture Notes in Computer Science**. 155. Springer-Verlag (1983).
15. Krablin, G.L. "Building flexible multilevel transactions in a distributed persistent environment". proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117.
16. Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P. "An integrated graphics programming environment". 4th UK Eurographics Conference, Glasgow (March 1986). In Computer Graphics Forum 5, 2 (June 1986),147-157.
17. 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 environment". IEEE 8th International Conference on Software Engineering, London (August 1985),166-172.
18. Morrison, R., Brown, A.L., Carrick, R., Connor, R. & Dearle, A. **Napier Reference Manual**. Dept of Computational Science. University of St Andrews.
19. Wegner, P. "Dimensions of object-based language design". OOPSLA 87 (1987), 168-182.

Appendix I

```
type  general_index [KEY, VALUE] is process (
                                Enter       : proc (KEY, VALUE),
                                Lookup      : proc (KEY -> VALUE) )

let generate_general_index = proc [Key, Value] (less_than : proc (Key, Key -> bool) ;
                                fail_value : Value -> general_index [Key, Value])
general_index [Key, Value] with
begin
  rec type index is variant (node : Node ; tip : null)
  & Node is structure (key : Key ; value : Value ; left, right : index)

  let null_index = index (tip : nil)
       !Compute the empty index by injecting the nil value into the variant

  let i := null_index
       !This is the internal index structure initialisation

  rec let enter = proc (k : Key ; v : Value ; i : index  -> index)
  !Enter the value into the binary tree indexed by key 'k'
  if i is tip then index (node : Node (k, v, null_index, null_index)) else
  case true of
  less_than (k,i'node (key) )        : { i'node (left) := enter (k, v, i'node (left)) ; i }
  k = i'node (key)              : { i'node (value) := v ; i }
  default                            : { i'node (right) := enter (k, v, i'node (right)) ; i }

  let lookup = proc (k : Key ; i : index -> Value)
  !lookup the value in the binary tree
  begin
       let head := i
       while head is node and k ≠ head'node (key)  do
                head := if less_than (k, head'node (key) )          then head'node (left)
                                                              else head'node (right)
       if head is node then head'node (value) else fail_value
  end

  while true do
       select
                : receive Enter (key : Key ; value : Value) do i := enter (key, value, i)
                : receive Lookup (key : Key -> Value) do lookup (key, i)
       selected
end
```