Configuring Object-Based Distributed Programs in REX

Jeff Kramer, Jeff Magee, Morris Sloman, Naranker Dulay

Department of Computing,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate, London SW7 2BZ, UK.

November 1991

Abstract

The popularity of the object oriented programming paradigm has stimulated research into its use for parallel and distributed programming. The major issues which impact such use are concurrency control, object interfaces, binding and inheritance. This paper discusses the relative merits of current solutions to these issues and describes an approach based on the use of active objects with essentially explicit interfaces and bindings, and composition as a pragmatic alternative to inheritance. The key feature of our approach is the use of a configuration language to define program structure as a set of objects and their bindings. The configuration language includes facilities for hierarchic definition of composite objects, for parameterisation of objects, for replication of both object instances and interface interaction points, for conditional configurations with evaluation of guards at object instantiation, and even for recursive definition of objects. This separate and explicit description of program structure complements the object oriented concepts yet is missing from most other approaches. This approach, termed Configuration Oriented Programming, is illustrated by examples from the REX environment for the development of distributable software. This environment is being developed by the REX collaborative ESPRIT II project [REX 89].

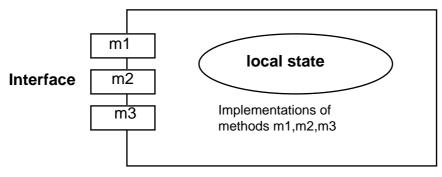
Keywords: Concurrency, Configuration Language, Object Oriented, Distributed System, Module Interconnection, Software Development

1. Introduction

Assessment of the impact of Object Oriented Programming (OOP) has varied considerably. On the one hand it has been hailed as a major revolution in programming paradigms, and on the other it is considered to be merely an extension to the use of Abstract Data Types. Reality is somewhere in between. There is no doubt that OOP has taken a number of sound and useful concepts and combined them in a form which has captured the popular imagination [Wegner 90]:

- Encapsulation permits the association of state and behaviour to form an object. This
 provides the benefit of abstraction during design and simplifies independent construction.
 This is essentially the concept of Abstract Data Types.
- ii) *Classes* serve as templates from which object instances are created. Classes thereby permit the use of a single behaviour specification for the set of all object instances created from that class.
- iii) *Inheritance* provides a mechanism for the reuse of the behaviour of one class in the definition of a new class. It establishes a class hierarchy which represents a generalisation / specialisation relationship between parent / child classes.

The design approach is essentially one of modelling, with real world entities being modelled as objects. The class hierarchy describes the relationship between the behaviours of the required objects. The underlying execution model is that of objects manipulating their local state and affecting others by sending messages which invoke methods (operations) in a client-server relationship. Those methods available for invocation are made visible at the object interface (Figure 1). All these are very appealing notions.



Behaviour: state changes in response to method invocation

Figure 1. - An Object in OOP

However, there are some weaknesses. These seem to emanate from the lack of an

explicit description of the program structure in OOP. We argue that a separate and explicit description of program structure, in terms of object instances and their interactions, facilitates program description, construction and modification, and that it complements the object oriented concepts yet is missing from most other approaches.

Object Oriented Design (OOD) methods concentrate on the definition of classes in an inheritance hierarchy. This hierarchy defines the relation between *classes*, but ignores the actual structure of the required program. This needs to be defined in terms of *object instances*. Explicit program structure, in terms of object instances and the interconnections or bindings between them, is a natural outcome of traditional design techniques such as the system structure diagram of JSD [Jackson 83], Data Flow Diagrams and Structure Charts of SASD [Page-Jones 80]). This resulting design structure of objects instances can be used to determine the classes required to perform the application processing [Kramer 90a], and *then* organised into an inheritance hierarchy to try to maximise software reuse.

To some extent, this lack of an explicit structural description in OOD is excused by the use of dynamic object creation and dynamic binding between objects. However, it is very difficult to determine the current overall structure of a program as it is embedded in the object code as parameters, instructions for instances to be created and references to objects to which bindings are required. This paper will show that even such dynamic programs can benefit by making the possible object instance structures more explicit.

OOP is also offered as one of the prime technologies expected to cope with large-scale applications and the issues of "megaprogramming" [Wegner 90]. Megaprogramming is programming in the large, with multiple personnel and multiple computers and where the programs are expected to have a long lifetime. Structural descriptions are ideally suited to programming in the large [DeRemer 76] as they provide a clear and useful reference abstraction for the personnel involved in the distributed development process, including program construction, evolution (modification) and software management.

What of concurrency and distribution? The success of OOP has encouraged researchers and language developers to examine its application for parallel and distributed programming. At first sight the model seems ideal: concurrent, distributable objects sending messages to invoke one anothers' methods. However, there are a number of key issues which impact its use in a distributed environment. For instance, at what granularity level should concurrency be provided? Should objects become active entities like processes, or should there rather be the ability to create threads of execution within objects? Here too we believe that the combination of OOP concepts and structural descriptions has much to offer. Others have also recognised the importance of highlighting the structural configuration view in a distributed environment [Barbacci 88, Goguen 86, Kaplan 88, Kramer 85, Leblanc 82, LeBlanc 85, Lee 86, Nehmer

87, Purtilo 88].

In section 2 of this paper, we examine the major issues which impact the use of OOP in a parallel and distributed environment: concurrency control, object interaction, interfaces, binding and inheritance. We discuss the relative merits of current solutions to these issues and describe and justify an approach based on the use of active objects with essentially explicit interfaces and bindings, and composition as a pragmatic alternative to inheritance. An explicit configuration language is used to define program structure as a set of objects and their bindings. This configuration language is separate from that used for programming objects as context-independent types (like classes) with well-defined interfaces.

Since a key feature of the approach is its explicit and separate expression of software structure, it is termed **Configuration Oriented Programming**. This emphasis on software architecture leads to clear and flexible designs, and produces object-based systems which are comprehensible, maintainable and amenable to change. The configuration language includes facilities for hierarchic definition of composite objects, for parameterisation of objects, for replication of both object instances and interface interaction points, for conditional configurations with evaluation of guards at object instantiation, and even for recursive definition of objects. In addition, the approach supports system evolution and change, expressible at the configuration level as changes to the configuration of object instances and/or their bindings. The utility and versatility of the approach is explained and illustrated in section 3 using a number of examples. Finally, section 4 concludes by summarising the approach and giving the current status of the work.

2. Concurrency and Distribution Issues

In this section, we overview and discuss some of the main issues which arise in the use of object oriented concepts in a parallel and distributed environment. We make no attempt to present a survey, but rather to describe the main concepts and their relative merits.

2.1 Object Concurrency and Interaction

There are two main approaches to the provision of concurrency in object based languages: either by the introduction of multiple threads of execution within an object, or by considering the objects themselves as active but sequential (cf. processes). We discuss these approaches, together with some of the mixed styles.

¹Wegner [87] makes a distinction between object based, class based and object oriented languages. Object based languages support encapsulation in the form of objects, class based extends this to include classes and object oriented further extends this to include inheritance. Strictly speaking, our language can be classified as class based.

Thread Model of Concurrency

Concurrency can be introduced by permitting objects to create new threads of execution using statements such as **cobegin...coend** [Nierstrasz 87]. This can be used in a client object to create multiple threads each of which can in turn send messages to invoke methods on other server objects, which could themselves introduce further concurrency, thereby forming a "tree" of execution threads through objects (Figure 2). At a particular server object, concurrency can occur through multiple invocation of its methods by different execution threads. This thread model thus provides a grain of concurrency finer than that of an object, and requires additional concurrency control primitives (such as semaphores) for synchronisation and to control access to shared data [Steigerwald 90]. Use of such synchronisation primitives is generally difficult and error-prone. Although some improvement has been made by permitting separate expression of this synchronisation in systems such as Guide [??Decouchant 81??], they can still require highly complex expressions.

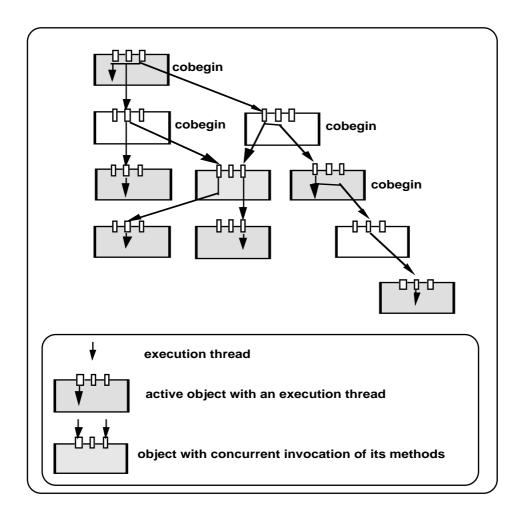


Figure 2. - Snapshot of a Program of Objects using the Thread Model of Concurrency

A variation of this approach is the use of asynchronous method invocation, which permits the client to continue execution and creates another thread of execution at the invoked server object. Although this appears to be attractive for distribution in providing an asynchronous form of communication, it is rather too primitive (low-level). For the common case of invocation with results, the programmer is left with the task of matching replies [Agha 86, Yonezawa 86].

Hence, although these approaches provide a fine grain of concurrency, they can be referred to as the "assembler level" of concurrent OOP.

Active Object Model of Concurrency

In order to make the concurrency control clearer, the grain of active concurrency can be made coarser and restricted to coincide with an object. Objects thus support only one active thread. A program would consist of sequential active objects (cf. processes) which communicate and synchronise via passive objects (cf. monitors). Concurrency is then controlled by queuing invocations at monitor objects combined with the ability to suspend a thread of execution through the use of synchronisation variables such as condition variables. Although this approach to concurrency is easier for the programmer, it suffers from being non-uniform in its use of two forms of object, and its restriction to a form of indirect interaction via monitor objects. Furthermore, monitors are more suited to interaction through shared memory than in a distributed environment.

This leads us to seek a more uniform form of concurrency which is clear and well suited to distribution. Such an approach is the use of sequential active objects (cf. processes) which communicate directly with one another by remote method invocation. A distributed program then consists of a number of distributed communicating objects, which can handle one invocation at a time. This has sacrificed the fine-grain concurrency of the concurrent threads approach, and adopted the object as the grain of concurrency and distribution. In so doing, the need for additional synchronisation primitives within an object are obviated and instead all that is necessary is the ability for an object to wait on a selection of possible invocations/messages.

The use of active sequential process objects is our preferred approach. The interface for interaction with an object is discussed below in section 2.2. How can more complex objects be constructed which encapsulate inherently parallel activities? We propose a simple approach which permits the definition of a composite object as a composition of primitive, active sequential objects (Figure 3). Composition is discussed in section 2.3.

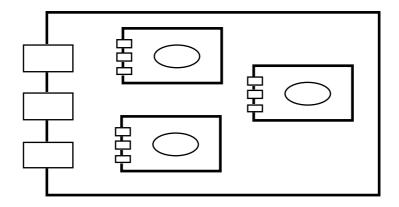


Fig. 3 Composite Objects Encapsulating Concurrency

2.2 Object Interfaces

Modularisation is a necessary facility not only for distribution, but also for sound software engineering. One of its recommended principles is that the interface to a module should explicitly define all the means of affecting its behaviour and state. We now examine and assess object interfaces in this light.

Objects interact by method invocation. An object interface is usually described by those methods which it offers (figure 1). However, its use of the methods of other objects is not descibed explicitly at the object interface but embedded internally in the object code. This is analogous to the publication of offered service interfaces in distributed systems but the hiding of service requirements. We believe that both the "services" *provided* and *required* are necessary for the description of object behaviour and ought to be explicitly defined at the object interface. The interface can thus clearly and explicitly identify the methods offered and used, the types of data associated with each, and even the form of interaction (such as uni- or bidirectional).

This form of interface definition can be extended to object naming. In the same way that, from the point of view of the invoked object, an invocation is from an anonymous (unnamed) object, so calls on other objects can be anonymous by making them indirect. Object code need never refer directly to other objects, but rather indirectly via remote references declared at its interface. Object binding can be performed separately. In this way, an object achieves context independence which facilitates its use and reuse in different contexts. In practice we define an object interface in terms of:

- i) the set of typed ports representing the methods that the object provides, and
- ii) the set of typed remote port references representing the methods *required* from other objects (Figure 4).

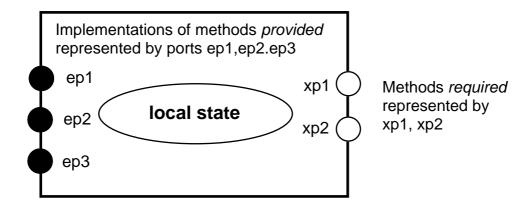


Figure 4. - An Object with explicit interfaces

2.3 Program structure: Object Creation and Binding

The structure of a program can be thought of as consisting of a number of object instances bound together according to their ability to communicate i.e. interconnections are bindings between required and provided interfaces. In a distributed environment, the objects can reside on different machines and communication can be remote. In addition it is necessary to allocate objects to physical nodes in the network either explicitly by the programmer or implicitly by the system.

The most common Object Oriented approach is dynamic object instantiation and binding. The distributed configuration starts as a single object which dynamically creates other objects locally or remotely. The binding between object interfaces is direct, by reference. These references can be obtained as instantiation parameters from the parent object which created it, as parameters in methods invoked on the object or by querying some form of name server or trader to locate a suitable server instance. Object creation, binding and allocation are embedded in the object programming language.

On the other hand, program structure can be thought of as essentially static. It can be elaborated initially as the distributed program configuration, which is unchanged during program execution. Object creation, binding and allocation can be specified either within the object programming language or specified separately in a configuration language (cf. Module Interconnection Language).

Whereas the dynamic approach is undoubtedly more flexible in expressing computations in which the structure changes over time, it has problems in the creation of the initial program

structure and in managing long running programs (such as embedded systems) where explicit structure is beneficial. Conversely, the static approach causes difficulties in programming systems which change in response to user demands (such as transaction systems).

The approach advocated in this paper is a synthesis of these two approaches. The structure of object instances can be explicitly and separately described. However, this description includes the description of structural changes (object creation/deletion and binding) which can be invoked by objects to perform dynamic change. The structure is thus available explicitly for management purposes but can change in response to the demands of the application computation. Furthermore, in order to retain the flexibility of dynamic binding where appropriate, traditional communication is enhanced by the ability to send port references in messages. This is discussed and illustrated in section 3.

2.4 Inheritance and Composition

Inheritance in Object Oriented Programming languages permits objects to share both behaviour and data with their parent superclasses. An example of the use of shared behaviour would be objects instantiated from subclasses of a parent superclass QUEUE which supported methods for queue insertion and deletion. These objects could then share the ability to be inserted and removed. However, while it is easy to see the utility of this mechanism in constructing complex sequential object behaviour, it is less simple in relation to parallel and distributed systems as object encapsulation can be violated. For instance, a subclass may refer to variables or local private operations in its superclass, or even to superclasses further up the inheritance hierarchy. Such use of shared class variables and procedures cannot be efficiently supported in distributed systems. Wegner [Wegner 87] has gone so far as to state that "...distribution is inconsistent with inheritance".

On the other hand, composition seems to offer a viable alternative to inheritance. Composition is a technique for *constructing* systems, and its inverse, decomposition is a means of *designing* systems. Composition provides a powerful means of abstraction in that it permits a collection of components to be treated as a single component. We therefore support the view that composition provides a sensible and efficient alternative to inheritance for distributed programming [Raj 89]. Section 4 presents an example to support this view.

2.5 Summary

We have discussed the relative merits of the main approaches to the issues of object concurrency and interaction, object interfaces and binding, and to inheritance, particularly as applied to a parallel and distributed environment. We have argued for the use of active objects and explicit interfaces and bindings. In particular, we argue that an explicit configuration language is required to define program structure as a set of objects and their bindings. This configuration language should be separate from that used for programming objects as context-independent types (classes). This language should also support hierarchic composition as this is a more flexible and powerful than inheritance for constructing distributed applications. In the following section, we describe our configuration oriented object-based approach, using a number of simple examples to illustrate its utility and versatility.

3. Configuration Oriented Programming in REX.

The main concepts of configuration oriented programming discussed in the previous section, namely those of explicit structure and hierarchic composition, are illustrated by examples from the REX environment for the development of distributed programs [REX 89]. We concentrate on the configuration facilities provided by Darwin, the REX configuration language. Darwin includes facilities for hierarchic definition of composite objects, for parameterisation of objects, for multiple instantiation of both objects and interface interaction points, for dynamic binding and instance creation, for conditional configurations with evaluation of guards at object instantiation, and even for recursive definition of objects. This work owes much to earlier experience using the Conic configuration language [Magee 89].

Producer - Consumer

Figure 5 depicts the structure of a simple producer-consumer system together with the description of that structure in Darwin. The basic structuring entity in Darwin is the component. Components may be parameterised with the basic types int, real and string. Components are composed from other component types. The specific types used to construct a component are declared by:

```
use <list of component types>
```

In the example, the component producers is constructed from the component types consumer and producer. Instances of these component types are declared by:

```
\verb"inst" < instance \ name > : < component \ type > \ (< parameters >) \ . or sometimes to avoid inventing unnecessary names:
```

```
inst <component type> (<parameters>)
```

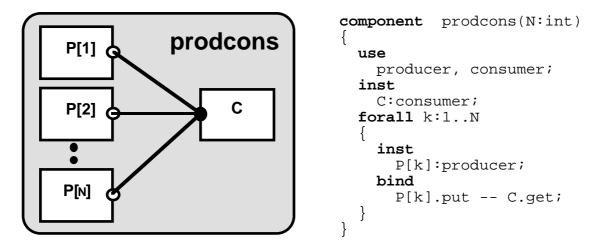


Figure 5 - Darwin Producer-Consumer program.

Instance declaration is analogous to the declaration of variables in convential sequential programming languages. The example declares one instance C of the consumer component type and using the replicator <code>forall</code>, a set of instances P[k] of producer where k ranges from 1 to N. These component types do not require actual parameters. Connections between instances are declared by:

```
bind <instance>.<service required> -- <instance>.<service provided>.
In the example, a connection is made from each producer instance P[k].put to the consumer instance C.get.
```

Components in Darwin may be constructed from other component types, as above, or they may be process types programmed in a sequential programming language augmented with message passing operations. In the following we will use Pascal to program process types. Figure 6 gives the programs for the producer and consumer process types of Figure 5.

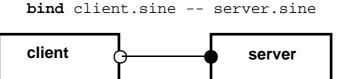
```
process consumer;
process producer;
                                provide get;
require put;
  put:@port (integer,real);
                                  get:port (integer,real);
                                   i:integer; r:real;
  i:integer;
                                begin
begin
 for i:=1 to 100 do
                                 loop
                                    ge?
                                           (i, r);
         (i, sqrt(i));
    put
                                     writeln(i,r);
end.
                                  end;
                                end.
```

Figure 6 - Producer and Consumer Process programs

Services in REX are provided via ports which are queues of messages. For example, the consumer process of Figure 6 queues messages consisting of an integer value and real value. Messages are transferred from the port into a processes local variables via the receive operation. In this case the operation is get?(i,r). The compiler or preprocessor ensures that the types of variables mentioned in a receive operation are compatible with the port on which the operation is performed. Ports are accessed remotely by remote references. A remote reference type is declared by prefixing the port declaration with an @ symbol. In the example, the variable put holds a reference value which refers to a port to which messages consisting of an integer and a real value can be sent. Messages are sent to a remote port by performing a send operation on a remote port reference. In the example, the send operation is put!(i,sqrt(i)). Again, the compiler checks that the values sent are compatible with the declared port reference type.

It should now be clear, that the effect of a bind declaration in Darwin is to assign the port reference value **provide**d by one process into the port reference value **require**d by another process. These bindings are typed checked using structural type descriptions supplied by process compilers to the Darwin compiler. Port references may also be sent in messages to allow more complex communication protocols than the simple unidirectional protocol of Figure 6. For example, the client and server processes of Figure 3 interact by means of a remote rendezvous protocol.

Client - Server



```
process client;
                                   process server;
                                   provide sine;
require sine;
                                   var
var
                                     sine:port(real,@port(real));
  sine:@port(real,@port(real));
                                     srep : @port(real);
  crep : port(real);
  angle, sinval: real;
                                     angle:real;
                                   begin
begin
                                      sine?(angle, srep);
   sine!(angle,@crep);
                                      srep!(sin(angle));
   crep?(sinval);
                                   end.
end.
```

Figure 7 - Client - Server

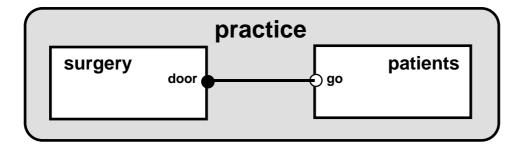
The client process sends a message to the server consisting of a real value and a reference to its local port crep. The reference to crep is computed by the @ operator. The server process receives the real value into the variable angle and the reference value into the variable srep. Srep is used to send the computed sine value back to the client. Note that the protocol works for any number of clients connected to the server. However, the syntax for declaring port reference types requires simplification to enhance the readability of programs. With no change to the semantics of the underlying operations, an alternative syntax for declaring port references is provided such that <code>->type_list</code> is equivalent to @port(type_list). Furthermore, since rendezvous interaction is very common the compiler provides an implicit reply port for the caller together with an implicit port reference variable for the callee of a remote rendezvous. The program of figure 7 now becomes the program of Figure 8. Rex also provides an extended rendezvous and selective receive statements[Magee 91].

```
process server;
process client;
                             provide sine;
require sine;
var
                               sine:port(real->real);
  sine:->real->real;
                              angle:real;
  angle,sinval:real;
                             begin
begin
                                . . . . . . .
   sine!(angle->sinval);
                                sine?(angle->sin(angle));
end.
                             end.
```

Figure 8 - Client - Server (alternative suntax)

General practioner's group practice

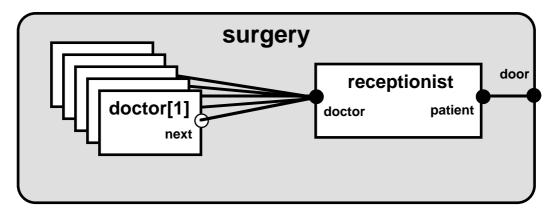
This example program models the workings of a general medical practioner's group practice. The practice consists of a surgery and a set of patients. Patients when they become sick visit the doctors' surgery. At the receptionist's desk they wait for one of the doctors to become free. When a doctor is free he examines the patient, diagnoses his or her illness and sends the patient off with a prescription. The doctor then returns to the receptionist to collect another patient. The example is a modified from that specified by ??? [].



```
component practice(ndoc,npat:int)
{
  use surgery,patients;
  inst
    surgery(ndoc) @loc="skid";
    patients(npat) @loc="bench";
  bind
    patients.go -- surgery.door;
}
```

Figure 9 - GP's group practice

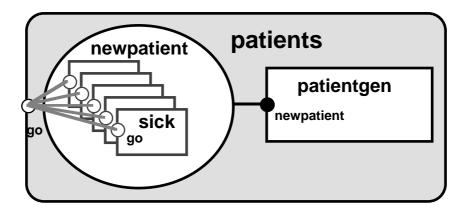
Figure 9 gives the top-level decomposition in which the practice is divided into two components, the surgery and the patients. The component is parameterised with the number of doctors in the practice ndoc and the number of patients to be treated npat before the program terminates. The Darwin program is annotated with information on how the practice component is to be executed in a distributed environment. The component surgery is to be instantiated on the machine named skid and patients on bench. By default, if a location for an instance is not specified with the @loc annotation, the instance is located with its enclosing component. The surgery is constructed from the process types doctor and receptionist as shown in Figure 10. The receptionist process provides two ports doctor and patient on which are queued respectively, free doctor processes and sick patient processes which have "entered" through the door of the surgery. Note that interfaces to composite components are specified using provide and require in precisely the same way that process interfaces are specified. Composite component interfaces are implemented by binding them to a process instance ports. For example in Figure 10 door is bound to receptionist.patient.



```
component surgery(ndoc:int)
{
  provide door;
  use doctor, receptionist;
  inst receptionist;
  bind door -- receptionist.patient;
  forall i:1..ndoc
  {
    inst doctor[i](i);
    bind doctor[i].next -- receptionist.doctor;
  }
}
```

Figure 10 - The Doctors' Surgery

The last composite component in the program is the set of patients. This component can be implemented in two different ways. We could statically declare an array of patient processes which delayed for some period of time and then became sick or we can use a patient generator task to dynamically create sick patients. We will choose the latter technique since in addition to showing dynamic structure creation in Darwin it uses memory more efficiently. The patients component is described in Figure 11.



```
component patients(npat:int)
{
   require go;
   use patient,patientgen;
   inst patientgen(npat);

   op newpatient(patid,npat:int->void)
   {
      inst sick:patient(patid,npat);
      bind sick.go -- go;
   }
   bind patientgen.newpatient--newpatient;
}
```

Figure 11 - The Patients

The patientgen process instance periodically invokes the configuration operation newpatient to create a new sick patient. Each new patient process has its requirement for a reference value (sick.go) associated with the interface requirement (go) of the patients composite component. At runtime, the required value will be a reference to the receptionist's port receptionist.patient. Configuration operations are invoked by the normal Rex message passing operation as shown (Figure 12) in the code for the patient generator process. Void indicates that a message is returned from the configuration operation for synchronisation purposes, however, it contains no information. The newpatient reference variable type of Figure 12 is thus used to send a message consisting of two integers and a reference to a port used only for synchronisation. The synchronisation port in the example is not declared explicitly, it is provided by the compiler for the operation:

```
newpatient!(i,npat->void).
```

An alternative but semantically identical declaration for newpatient would be:

```
newpatient: @port(integer,integer,@port()).

process patientgen(npat:integer);
require newpatient;
var
newpatient: ->integer,integer->void;
i:integer;
```

```
begin
  for i:= 1 to npat do
  begin
    delay(1 * seconds);
    newpatient!(i,npat->void);
  end;
end.
```

Figure 12 - patient generator process

The remaining three process types share the following type definitions:

```
type
  name = integer;
  illness = integer;
  prescription = integer;
```

The patient process of Figure 13 simply sends a message consisting of the name of the patient and his/her illness and then waits for the prescription. The last patient process created by the patient generator terminates the distributed program when id = npat.

```
process patient(id:name; npat:integer);
require go;
var
   go: ->name,illness->prescription;
   I:illness; P:prescription;
begin
   writeln("patient ",id:1," Sick");
   I:=id*id;   {illness is id²}
   go!(id,I->P);
   writeln("patient ",id:1," got Prescription ",P:1);
   if id=npat then halt;
end.
```

Figure 13 - patient process

The doctor process of Figure 14 contains the most complex port type description we have as yet introduced. This may be easily understood with the aid of the message sequence chart of Figure 14 which is really just an alternative way of laying out the type declaration. The doctor process sends a synchronisation message to the receptionist consisting of a reference to its own implicit port. The receptionist replies with the details of a patient ie. name, illness and a prescription port reference. The doctor returns a prescription directly to the patient using the prescription reference. The alternative syntax for the port reference variable next would be:

```
next: @port(@port(name,illness,@port(prescription)))
```

Message Sequence Diagram doctor receptionist name, illness doctor receptionist prescription process doctor(id:integer); patient doctor require next; next:->void->name,illness->prescription; I:illness; N:name; P:prescription; Preply:->prescription; begin loop next!(void->N,I,Preply); writeln("Doctor ",id:1," Treating ",N:1); delay(4*seconds); P:=-I ;{diagnose patient's illness} Preply!(P); end; end.

Figure 14 - doctor process

The remaining receptionist process acts as a mailbox (Figure 15). On one side there is a queue of sick patients and on the other, a queue of free doctors. The receptionist repeatedly gets a patient and forwards it to a doctor for diagnosis.

```
process receptionist;
provide doctor,patient;
var
  doctor: port(void->name,illness->prescription);
  patient:port(name,illness->prescription);
  I:illness; N:integer; Preply:->prescription;
begin
  loop
     patient?(N,I,Preply);
     doctor?(void->N,I,Preply);
  end;
end.
```

Figure 15 - receptionist process

The *General practioner's group practice* is an example of a multi-server system. The patients being the clients and the doctors the servers. The server processes could just as easily return a reference to a record of ports representing, for example, a file access interface. The reader who has attempted to program a multi-server in a language such as Ada or Occam (or even REX's predecessor CONIC) will appreciate the elegance of the program.

Composition as an alternative to inheritance

Composition as an alternative to inheritance

The modularity of the previous example means that it would be trivial to replace the doctors' surgery with a single doctor process. However, this doctor process would not have precisely the same form as the program of Figure 14 since it would not require the receptionist access protocol. The single or lone doctor program would be as described in Figure 16.

```
process lonedoctor(id:integer);
require next;
var
  patient:port(name,illness->prescription);
  I:illness; N:name;
  P:prescription; Preply:->prescription;
begin
  loop
     patient?(N,I,Preply);
     writeln("Doctor ",id:1," Treating ",N:1);
     delay(4*seconds); P:=-I ; {diagnose patient's illness}
     Preply!(P);
  end;
end.
```

Figure 16 - lonedoctor process

Suppose we had started with the lonedoctor process and then wished to reuse its functionality in the multi-doctor surgery? In a sequential object oriented system such as Smalltalk, we would make groupdoctor a subclass of lonedoctor and thus let it inherit the functionality of lonedoctor ie. lonedoctor class groupdoctor {....}. In Darwin, the same effect is achieved by composition as shown in Figure 17. The process raccess provides the additional receptionist access protocol necessary to let lonedocter work in the surgery environment. Note that groupdoctor has precisely the same functionily as the doctor process of Figure 14. Admittedly, this is less elegant than the inheritance mechanism, however, it is difficult to see how the inheritance mechanism would deal with the composition of synchronisation in other than simple cases. The attempt made in the Guide language[] is less than satisfactory since it deals only with a subset of the possible synchronisation interactions.

```
component groupdoctor(id:int)
   doctor
                                   require next;
                                   use lonedoctor, raccess;
                           next
                    R
                                      L:lonedoctor(id);
                                      R:raccess;
                                   bind
                                     R.next -- next;
                                     R.lonedoctor -- L.patient;
process raccess;
                                 }
require next, lonedoctor;
var
  next:->void->name,illness->prescription;
  lonedoctor:->name,illness->prescription
  I:illness; N:name;
  P:prescription; Preply:->prescription;
begin
  loop
     next!(void->N,I,Preply);
     lonedoctor!(N,I->P);
     Preply!(P);
  end;
end.
```

Figure 17 - Constructed groupdoctor program

Guarded Configuration Programs

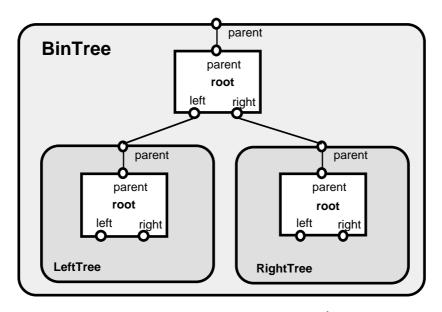
Now that we have the lone doctor process of Figure 16, it would be sensible to use it, rather than the surgery component when the parameter of the practice program ndoc=1. This avoids the overhead of the receptionist in a single doctor surgery. This can be accomplished using the Darwin **when** construct as shown in Figure 18.

```
component practice(ndoc,npat:int)
{
  use surgery,patients,lonedoctor;
  inst patients(npat) @loc="bench";
  when ndoc==1 {
    inst lonedoctor(1) @loc="skid";
    bind patients.go -- lonedoctor.patient;
  }
  when ndoc>1 {
    inst surgery(ndoc) @loc="skid";
    bind patients.go -- surgery.door;
}
```

Figure 18 - Guarded practice program

Recursive Configuration Programs

Guards are also required to terminate recursive configuration programs. Recursive configurations are commonly used to describe regular structures such as trees. Figure 19 is the Darwin description of a binary tree of component instances of type node. RightTree and LeftTree are both instances of the enclosing type BinTree giving the recursive structure definition. Further examples of recursive structures defined in Darwin may be found in {}.



```
component BinTree(depth:int) = {
  provide parent;
  use node;
  inst root : node;
  bind root.parent -- parent;

when depth>1 {
  inst LeftTree :BinTree(depth-1);
      RightTree:BinTree(depth-1);
  bind root.left -- LeftTree.parent;
      root.right -- RightTree.parent;
  }
}
```

Figure 19 - Recursive Binary Tree Component

4. Conclusions

Object Oriented programming has lead to a proliferation of literature, languages, tools and, unfortunately, hype comparable only with that of structured programming. A reasoned and careful evaluation is necessary to extract the real benefits that OOP offers, especially if one seeks to extend it to cater for concurrency and distribution as well.

In this paper we have briefly discussed conventional OOP and found it wanting in its support for describing program configuration or structure. We have discussed the key issues which impact its use in a distributed environment viz. concurrency control, object interfaces, binding and inheritance, pointing out the relative merits of current solutions to these issues. For instance, most object oriented languages do not support explicit interfaces of services *required* by an object, only those services offered. We believe that both are needed for management purposes and to fully identify object functionality.

We have introduced and illustrated the utility of our Configuration Oriented Programming approach, based on the use of a separate and explicit configuration language to define program structure as a set of active objects (components) with explicit interfaces and bindings, together with the ability to compose them hierarchically to build up more complex components. This provides an alternative to inheritance for reuse of behaviour in distributed systems.

In the discussion of object concurrency and interaction, we made the distinction between the threads and active (process) object models, and indicated our preference for the latter. Conventional OOP with inheritance does not scale for large distributed applications but is still appropriate as a programming technique for individual sequential components. It therefore seems appropriate that it should be possible to pass simple objects (with their state and behaviour) in messages and as invocation parameters. This provides a powerful yet practical means of communication, even in a distributed environment. Since the port interface specifies the types of object which it can accept or produce, it can ensure that the behaviour for communicated objects is also available and need not be sent.

The REX work is heavily based on Conic, with which we have had extensive experience for a number of years. This has provided us with convincing evidence of the utility of the configuration approach for distributed program design [Kramer 90a], construction [Magee 89], evolution [Kramer 90b] and management using graphic tools such as ConicDraw [Kramer 89]. However, a number of limitations to Conic and its implementation have also been recognised. It provided support for *post hoc* evolutionary change but no linguistic support for dynamic programmed change, where the configuration changes are embedded in the configuration description (as illustrated in the exchange switch). Component interfaces in Conic are simpler and do not support port sets and the associated binding rules. Finally, there is no integrated

support for components written in different programming languages. These limitations are being addressed in REX and its configuration language, Darwin.

The current status of the work is that Darwin has been implemented and the communication primitives have been embedded in components written in C or C++. Colleagues in the REX project are embedding the primitives in other languages such as Modula 2.

Some particular aspects of megaprogramming have been neglected. For instance, persistence and atomicity of objects and actions is beyond scope of this paper, but there is promising research in this area (eg. see Arjuna [Shrivastava 89]). Nevertheless, we believe that configuring object-based distributable programs offers a realistic framework towards providing the necessary support for large distributed systems.

Acknowledgements

Acknowledgement is made to our colleagues at Imperial College (Steve Crane, Essie Cheung, Anthony Finkelstein, Keng Ng, and Kevin Twidle) and also to our partners in the REX project (at T.U. Berlin, GMD Karlsruhe, Siemens, Stollman, Karlsruhe University and 2i in Germany, Intracom and Intrasoft in Greece, Tecsi in France, and PRG Oxford in the UK) for their contribution to the work described in this paper. We gratefully acknowledge the SERC ACME Directorate under grant GE/E/62394, the SERC under grant GE/F/04605 and the CEC in the REX Project (2080) for their financial support.

References

- [Agha 86] G. Agha, "Actors: A Model of Concurrent Computation in Distributed Systems", MIT Press, Cambridge, Mass. 1986.
- [Barbacci 88] M.R.Barbacci, C.B.Weinstock, and J.M.Wing, "Programming at the Processor
 Memory Switch Level", Proc. of 10th IEEE Int. Conf. on Software Engineering, Singapore, April 1988.
- [DeRemer 76] F. DeRemer, H.H.Kron. "Programming-in-the-large Versus Programming-in-the-small, IEEE Trans. Software Engineering", Vol. SE-2, 2, June 1976.
- [Goguen 83] J.A.Goguen. "Reusing and Interconnecting Software Components", IEEE Computer, (Designing for Adaptability), Vol. 19, 2, February 1986.
- [Jackson 83] M.A.Jackson, "System Development", Prentice Hall 1983.
- [Kaplan 88] S. Kaplan, G. Kaiser, "Garp: Graph Abstractions for Concurrent Programming", *ESOP* '88, Nancy, France, March 1988, Springer-Verlag, pp. 191-205.
- [Kramer 85] J.Kramer, J.Magee, "Dynamic Configuration for Distributed Systems", IEEE Transactions on Software Engineering, SE-11 (4), April 1985, pp. 424-436.

- [Kramer 89] J. Kramer, J. Magee, K. Ng, "Graphical Configuration Programming", IEEE Computer, 22(10), October 1989, 53-65.
- [Kramer 90a] J. Kramer, J. Magee, A. Finkelstein, "A Constructive Approach to the Design of Distributed Systems", to be presented at the 10th Int. Conf. on Distributed Computing Systems, May 1990.
- [Kramer 90b] J. Kramer, J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management", IEEE Transactions on Software Engineering, SE-16 (11), Nov. 1990, pp. 1293-1306.
- [Leblanc 82] R.J. Leblanc and A.B. MacCabe, "The Design of a Programming Language based on a Connectivity Network", Proc. 3rd Int. Conf. On Distributed Computing Systems, 1982.
- [LeBlanc 85] T. LeBlanc and S. Friedberg. "HPC: A model of structure and change in distributed systems". IEEE Trans. on Computers, Vol. C-34, 12, December 1985.
- [Lee 86] I. Lee, N. Prywes, B. Szymanski, "Partitioning of Massive/Real-Time Programs for Parallel Processing", in Advances in Computers, ed. M.C. Yovits, Vol.25, Academic Press 1986.
- [Magee 89] J.Magee, J.Kramer, and M.Sloman, "Constructing Distributed Systems in Conic" IEEE Transactions on Software Engineering, SE-15 (6), June 1989.
- [Magee 90] J.Magee, J.Kramer, M.Sloman and N. Dulay, "An Overview of the REX Software Architecture", Proc. of 2nd IEEE Computer Society Workshop on Future Trends of Distributed Computer Systems, Cairo, Oct. 1990.
- [Nehmer 87] J. Nehmer, D. Haban, F. Mattern, D. Wybranietz, D. Rombach, "Key Concepts of the INCAS Multicomputer Project", IEEE Transactions on Software Engineering, SE-13 (8), August 1987.
- [Nierstrasz 87] O.M. Nierstrasz, "Active Objects in Hybrid", Proc. OOPSLA '87, Sigplan Notices, Vol 22, No 12, December 1987, pp243-253.
- [Page-Jones 88] M. Page-Jones, "Practical Guide to Structured Systems Design", Prentice Hall International Editions, 1988.
- [Purtilo 88] J. Purtilo, "A Software Interconnection Technology", Computer Science Dept., University of Maryland, TR-2139, 1988.
- [Raj 89] R.K.Raj and H.M.Levy, "A Compositional Model for Software Reuse", The Computer Journal, 32 (4), 1989, 312-322.
- [REX 89] REX Technical Annexe, ESPRIT Project 2080, European Economic Commission, March 1989.
- [Steigerwald 90] R. Steigerwald, "Concurrent Programming in Smalltalk-80", ACM Sigplan Notices, Vol 25, No 8, Aug 90, pp 27-36.
- [Shrivastava 89] S.K.Shrivastava, G.N.Dixon, G.D.Partington, "An Overview of Arjuna, A Programming System for Reliable Distributed Computing", TR 298, Computing Laboratory, University of Newcastle-upon-Tyne, Nov. 89.

- [Wegner 87] P.Wegner, "Dimensions of Object-Based Language Design", Proc. of OOPSLA '87, Oct. 1987, 168-182.
- [Wegner 90] P.Wegner, "Concepts and Paradigms of Object-Oriented Programming", OOPS Messanger (ACM SIGPLAN), Vol. 1, NO. 1, August 1990, 7-87.
- [Yonezawa 86] A. Yonezawa, J-P Briot, E. Shibayama, "Object-Oriented Programming in ABCL/1", Proc. OOPSLA '86, ACM SIGPLAN Notices, Vol 21, No 11, November 1986, pp258-268.