A Constructive Development Environment for Parallel and Distributed Programs

Jeff Magee, Naranker Dulay, Jeff Kramer

Department of Computing,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate, London SW7 2BZ, UK.
Email: jnm@uk.ac.ic.doc
Tel: +44-71 589 5111 x5040
Fax:+44-71 581 5024

Abstract

Regis is a programming environment aimed at supporting the development and execution of parallel and distributed programs. It embodies a constructive approach to the development of programs based on separating program structure from communication and computation. The emphasis is on constructing programs from multiple parallel computational components which cooperate to achieve the overall goal. The environment is designed to easily accommodate multiple communication mechanisms and primitives. Both the computational and communication elements of Regis programs are programmed in the Object Oriented programming language C++. The elements are combined into parallel and distributed programs using the configuration language *Darwin*. The paper describes programming in Regis through a set of small example programs.

Keywords

parallel programming, distributed programming, inter-process communication, parallel programming language, software development environment.

1. Introduction

The *Regis* environment we describe in this paper has evolved from our research into "configuration" programming[1]. The premise of this approach is that a separate, explicit structural (configuration) description is essential for all phases in the software development process for distributed programs, from system specification as a configuration of component specifications to implementation as a set of interacting computational components. Configuration programming thus separates the description of program structure from the programming of computational components. In common with others[2,3,4] have found this to be crucial in managing the complexity of large parallel and distributed programs. The Regis environment extends our previous work in two major areas: by separating communication from computation and in the support for dynamic program structures.

The Conic system[5] separated structure from component behaviour but was restricted to a single programming language (Pascal) augmented with a fixed set of communication primitives for defining computational components. REX[6], which suceeded Conic, permitted the implementation of computational components in a range of sequential programming languages, however, these components were restricted to a fixed set of intercommunication primitives. The Regis system we described here removes the latter restriction by allowing components to interact through user specified communication primitives. In essence, Regis allows the separation between configuration, computation and communication while its predecessors considered computation and communication as integral.

The configuration language included in the Conic system permitted only the definition of static component graphs. The set of component instances and their interconnections was fixed at system startup time. However, Conic did allow distributed programs to be interactively configured and modified at run-time through the agency of an external configuration manager. The configuration language provided in the REX system in contrast allowed the user to define arbitrary configuration operations which could be invoked at system runtime to modify and extend the initial structure. REX attempted to include the functionality of the configuration manager in the configuration language. However, this generality obscured the clarity of the configuration description since it provided a precise description only of the initial structure. The subsequent structure depended on the exact sequence of configuration operations which had been executed. We feel that Regis is a more satisfactory compromise between the desire for a precise description of program structure and the need for dynamic component instantiation in some applications. Regis provides lazy component instantiation and direct dynamic component instantiation as the only two methods of programming dynamic structures. These limited forms of dynamic instantiation have been found, so far, to satisfy the needs of applications. The result is that the Regis configuration describes the potential structure of an application which may or may not be completely elaborated at runtime. The need for arbitrary evolutionary change to

structure which REX attempted to accomodate is catered for by providing hooks into the external Open Systems environment in which Regis programs execute.

The computational components in Regis are provided by C++ objects. Computational components interact via communication objects again programmed in C++. The framework in which these objects concurrently execute is programmed in the configuration language *Darwin.* This is essentially the language described in [7]. The version used here differs only in its treatment of dynamic instantiation and in the way generalised communication is handled. In the following, the basic elements of the Regis environment are described and illustrated by a set of examples[1] . We look at configuration, communication and computation in the Regis environment and then see how these elements are combined to form parallel and distributed programs. The more advanced features of the Regis environment concerned with dynamic and generic structures are then examined before we conclude by discussing our experience with using the environment.

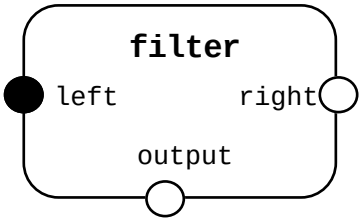
2. Program Configuration

From the previous section, we have seen that programs in Regis consist of multiple concurrently executing and interacting computational components. Typically, a program consists of a limited set of component types with multiple instances of these types. The task of describing a program as a collection of components with complex interconnection patterns quickly becomes unmanageable without the help of some structuring tools. In Regis, the structuring tool is the configuration language Darwin. Darwin allows parallel programs to be constructed from hierarchically structured configuration descriptions of the set of component instances and their interconnections. Composite component types are constructed from the basic computational components and these in turn can be configured into more complex composite types.

Components

Darwin views components in terms of both the communication objects they provide to allow other components to interact with them and the communication objects they require to interact with other components. For example, the component of Figure 1 is a *filter* component which provides the communication object *left* and requires two communication objects *right* and *output.* The diagrammatic convention used here is that filled in circles represent communication objects provided by a component and empty circles represent communication objects required by a component.

[1] We have reused some of the examples from [7] so that the interested reader may easily follow the evolution of the Darwin configuration language.



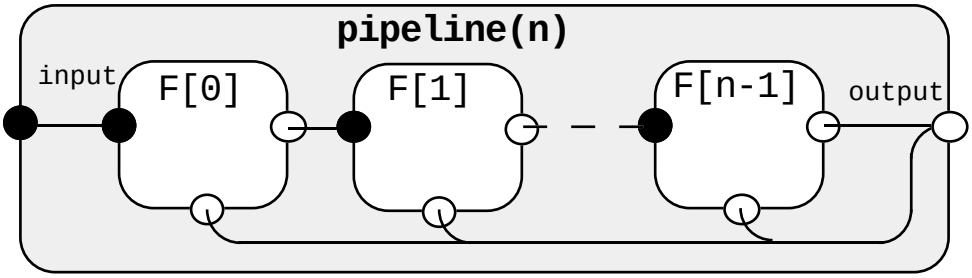

```
component filter {
  provide left   <port,int>;
  require right  <port,int>,
          output <port,int>;
}
```

Figure 1 - Darwin description of filter component.

In the Darwin description of the *filter* component, the required and provided communication object names are annotated with a type description <*port, int*>. These annotations, as we will see in the next section, are used in the generation of class headers for computational components implemented in C++. They indicate that the communication objects provided and required are message ports which accept messages of the integer type *int.* Ports are one of the standard communication object classes provided by Regis.

Composite Components

The primary purpose of the Darwin configuration language is to allow programmers to construct composite components from both basic computational components and from other composite components. The resulting parallel program is a hierarchically structured composite component which when elaborated at execution time results in a collection of concurrently executing computational component instances. Darwin is a declarative notation. Composite components are defined by declaring both the instances of other components they contain and the bindings between those components. Bindings, which associate the communication objects required by one component with the communication objects provided by others, can be visualised as filling in the empty circles of a component with the solid circles provided by other components. The example of Figure 2 of a *pipeline* component constructed from the *filter* component of Figure 1 illustrates the Darwin ***bind*** statement.




```
component pipeline (int n) {
  provide input;
  require output;

  array F[n]:filter;
  forall k:0..n-1 {
    inst F[k];
    bind F[k].output -- output;
    when k<n-1
      bind F[k].right -- F[k+1].left;
  }
  bind input -- F[0].left;
       F[n-1].right -- output;
}
```

Figure 2 - Darwin description of composite pipeline component.

The Darwin ***bind*** statement is also used to described how component interface provisions are implemented by internal component instances (e.g. Figure 2, ***bind*** *input -- F[0].left*) and to export requirements to the component interface (e.g. Figure 2, ***bind*** *F[k].output -- output*). As can be seen from this last example, many required communication objects can be satisfied by one provided object. However, a required object may only be bound to a single provided object. The Darwin compiler checks that bindings are only made between compatible communication objects (in this case that the provided port and required port are both of the same type *int*). Where necessary, the compiler infers the type of interface objects which are not explicitly typed (e.g.. *input* and *output*). The ***forall*** construct of Figure 2 is used to declare an array of *filter* instances and their bindings. The ***when*** construct allows conditional declaration of both instances and bindings, although in this case, only a conditional binding is declared. Instance arrays may be multi-dimensional, the ***array*** declaration is used to specify the dimensions.

So far, we have described how the Regis configuration language (Darwin) may be used to describe static structures of component instances. Before examining more advanced features which permit the description of recursive, generic and dynamic structures we will examine how communication and computation are catered for in Regis.

3. Communication

Inter-component communication is supported in Regis by communication objects defined by C++ template classes. A communication object is contained within the component which *provides* the object and is remotely referenced by the component which *requires* it. Figure 3 is the C++ description of the Regis ***port*** communication object previously mentioned in relation to Figure 1.

```
template <class T>
class port: public portbase {
public:
      void in(T &msg);            // receive message  of type T into msg
      int  inv(T msg[], int n);   // receive   inv elements of vector   (maximum n )

      void out(T &msg);           // synchronous send  msg of type T
      void outv(T msg[],int n);   // synchronous send  of n elements of msg

      void send(T &msg);          // asynchronous send  msg of type T
      void sendv(T msg[],int n); // asynchronous send  of n elements of msg
};
```

Figure 3 - Template class for Regis Port Objects

Port objects are really queues of messages of a particular type *T*. Messages may be queued to a port object (*out, send*) and removed from a port object (*in*). A communication object named *input* for integer messages would be declared in C++ as:

```
port <int> input;
```

The operation: `input.send(3)` would queue an integer message with the value 3 to the port *input*. This send is asynchronous in the sense that it does not block its calling process. The synchronous operation *out* blocks its calling process until the message has been received. Variable length messages may be transferred through ports using the vector primitives (*inv, outv, sendv*). The operations on ports are implemented using the underlying class *portbase* which supplies untyped send and receive operations. In addition, *portbase* supplies methods common to all ports. These methods are used to selectively wait on a set of ports and to determine whether messages are queued to a port.

The port operations we have described so far are only available to invoking processes which are co-located with the port object. That is, they are resident on the same processor and in the same address space such that they can either name a port directly or use a pointer to it. This is of limited use in a distributed memory environment. Consequently, the implementer of communication objects in the Regis framework must describe not only the object class but also a class which can be used to access that communication object remotely. These remote access classes are by convention named by adding the suffix *ref* to the name of the communication

object class it is providing access to. Regis provides the template class *portref* for remote access to port objects. As can be seen from Figure 4, we have chosen to support only the sending operations in the remote access class for ports. Although, a remote receive operation could in theory be implemented, this would not be efficient, nor would it lead to clear program design.

```
template <class T>
class portref : public remref {
public:
      portref();
      portref( port<T> &P);

      void out(T &msg);
      void outv(T msg[],int n);

      void send(T &msg);
      void sendv(T msg[],int n);
};
```

Figure 4 - Template class for Port remote access

The base class for port references is the *remref* class provided by the Regis run-time system. *Remref*, short for remote reference, provides operations to transfer data between machines and to remotely invoke operations. In addition, *remref* , ensures that the system can detect invalid remote references (i.e. subclasses of *remref*) to objects which no longer exist. From Figure 4, we can see that a remote reference value can be constructed from a port object. In effect, this makes a binding between a port reference and a port object. Figure 5. summarises the relationship between the declaration of communication objects in Darwin and C++ and the relationship between binding and remote references.

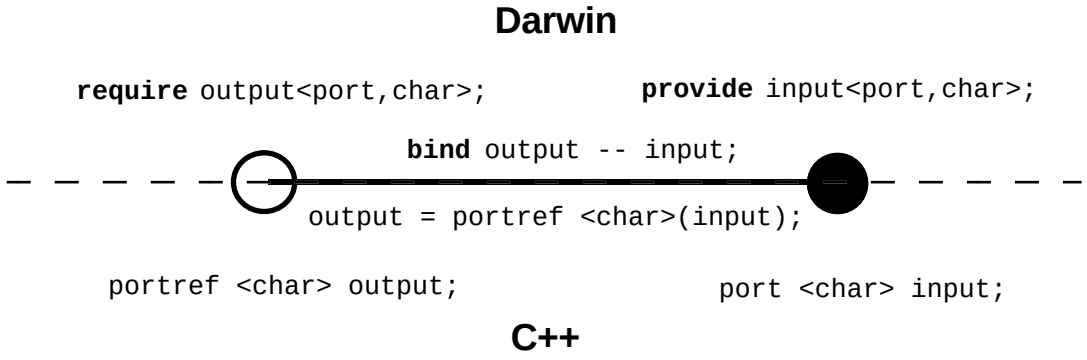


Figure 5 - Communication Objects in Darwin & C++

For computational components, required communication objects become reference objects and provided communication objects become object declarations at the C++ implementation level. Bindings are made by associating the reference object with the actual communication object. The reader should note that this simple correspondence between Darwin require/provide interfaces and the C++ objects and reference objects is only true for the base level of computational components. Composite component interfaces have no concrete representation at runtime. They are an artifact of program structuring which incur no runtime resource overhead. Similarly, a Darwin binding may result in many C++ reference to object bindings. Table 1 summarises the performance of the Regis port communication primitives in our environment of Sun SPARC IPX workstations connected by ethernet. Remote communication is implemented on top of Sun's implementation of the UDP/IP datagram protocol. The local communication times are dominated by the time taken to perform a light-weight thread context switch on the SPARC architecture.

Test	Message Size(bytes)	Local	Remote
Synchronous X.out(M) -> X.in(M).	1	118uS	1.89mS
	100	126uS	2.05mS
	1000	197uS	3.01mS
Asynchronous X.out(M) -> X.in(M).	1	121uS	0.98mS
	100	131uS	1.16mS
	1000	182uS	2.17mS

Table 1 - Port communication performance

In this section, we have illustrated how communication is supported in Regis by C++ objects. In particular, we have used the Regis *port* object as an example. Ports can be used to transfer messages with complex datatypes as well as the simple base types used here. Reference objects may be sent in messages to allow bindings to be set up dynamically. Regis has a library of communication template classes which currently includes bidirectional request-reply ports and streams in addition to ports. We have chosen to use communication template classes in Regis rather than directly supporting remote method invocation since it gives us the flexibility to use different communication mechanisms in the same program and in addition allows us to optimise communication for a particular mechanism. For example, it would be inefficient to implement the asynchronous *send* operation on a remote port by using a synchronous remote method invocation. By specifying communication in terms of both a communication class and a remote reference class, we have the opportunity to optimise the interaction between objects of these classes.

4. Computation

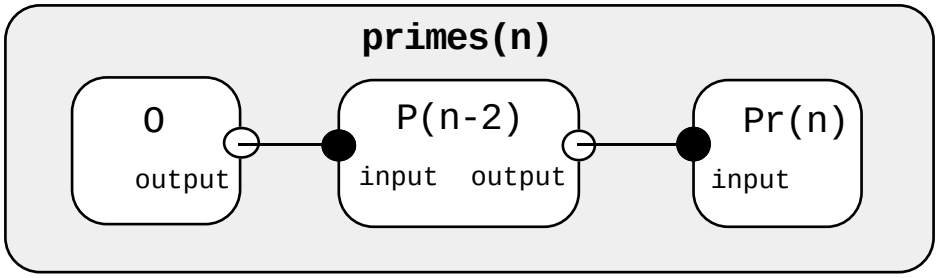
The computational components of a Regis program are again programmed in C++. Computational components execute as lightweight threads or processes. In our current workstation implementation, many computational components may execute inside a Unix heavyweight process. Lightweight threads are implemented by the Regis *process* class. Computational components must be implemented as a subclass of *process* as shown in Figure 6 which gives the C++ implementation of the filter component defined in Figure 1. The class header for this component is directly generated from the Darwin declaration. The class constructor forms the body of the component and is supplied by the programmer.

```
// generated from Darwin description
class filter : public process {
public:
  port<int> left;
  portref<int> right;
  portref<int> output;
  filter();
};
```

```
// constructor implements component computational function
filter::filter() {
  int first, val;
  left.in(first);
  output.out(first);
  for(;;) {
    left.in(val);
    if (val%first) right.send(val);
  }
}
```

Figure 6 - Filter C++ computational component

The filter program of Figure 6 transfers the first value it receives from *left* to *output*. Subsequent values are transferred from left to right if they are not multiples of the first value received. The pipeline of Figure 2 when used with this implementation of the filter process forms a program to compute prime numbers as shown in Figure 7. The additional computational components are a process to generate odd numbers(*odds*) and a process to print out the prime numbers (*print*). The code for these processes is also described in Figure 7. The program is terminated when the first *n* prime numbers have been printed out by the call to *end_program()*. Regis programs also terminate when all their constituent computational components have terminated. However, in this case, since the filter processes are programmed to execute endlessly, the forced termination using *end_program()* is required.




```
component odds {
 require output<port,int>;
}

component print (int n) {
  provide input <port,int>;
}

component primes(int n) {
  inst
    O:odds;
    P:pipeline(n-2);
    Pr:print(n);
  bind
    O.output -- P.input;
    P.output -- Pr.input;
}
```

```
odds::odds () {
 int i;
 for(i=3;;i+=2)
   output.out(i);
}

print::print(int n){
  cout << 2 << '\n';
  while(--n){
    int prime;
    input.in(prime);
    cout << prime << '\n';
  }
  end_program();
}
```

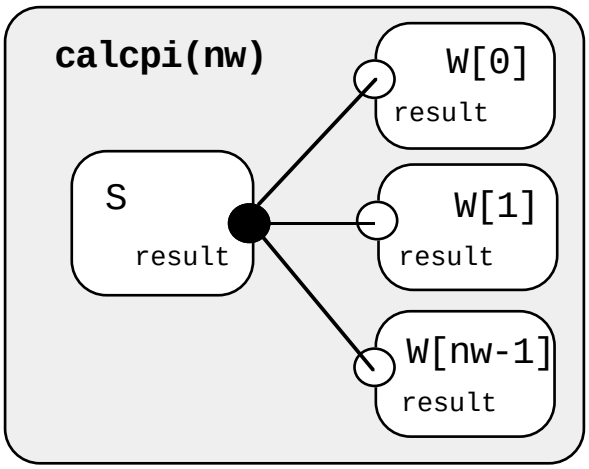
Figure 7 - Primes Sieve of Eratosthenes

In summary, Regis computational components are active objects in the sense that they incorporate a thread of control. The thread mechanism, as discussed in the foregoing, is inherited from the *process* class. These process objects interact through communication objects as described in the previous section. The function of a computational component is implemented as the constructor method for a C++ class derived from *process.* Although not shown here, a programmer can directly embed C++ declarations in Darwin component descriptions to allow the generation of process class headers which have more than communication objects as members.

5. Distributed Execution

So far we have examined how Regis programs are constructed from a Darwin configuration description. This description is a hierarchically structured specification of interconnected component instances. When elaborated at execution time, this specification results in a network of intercommunicating active objects. We now examine how Regis programs are executed in a distributed system, in our case, a collection of workstations connected by a local area network. To illustrate distributed execution, we will use an example program which computes an

approximation to π by calculating the area under the curve $4/(1+x^2)$ between 0 and 1 using numerical integration. This computation has been used by Lewis and El-Rewini [8] to compare a wide range of parallel programming languages, tools and environments. The program is structured as a set of workers which each compute a part of the integration and a supervisor which combines the results. Figure 8 outlines both the Darwin structural description for this program and the C++ computational components.




```
component supervisor(int nw){
 provide result <port,double>;
}

component worker(int id, int nw,
                 int intervals){
  require result <port,double>;
}

component calcpi(int nw){
 const int intervals=400000
 array W[nw]:worker;
 inst S:supervisor(nw);
 forall k:0..nw-1 {
  inst W[k](k,nw,intervals) @ k;
  bind W[k].result--S.result;
 }
}
```

```
supervisor::supervisor(int nw){
 double area=0.0;
 for (int i=0; i<nw; i++) {
  double tmp;
  result.in(tmp);
  area+=tmp;
 }
 printf("Approx pi %20.15lf\n",area);
 exit();
}

worker::worker (int id, int nw,
                int intervals){
 double area=0.0;
 double width=1.0/intervals;
 for (int i=id; i<intervals; i+=nw){
  double x=(i+0.5)*width;
  area+=width*(4.0/(1.0+x*x));
 }
 result.send(area);
 exit();
}
```

Figure 8 - Program to calculate approximate value of π

The *calcpi* program is parameterised with the number of worker components. This determines the degree of parallelism. Each *worker* component instance is mapped to a different

logical processor by the annotation @k in the line: **`inst`** `W[k](k,nw,intervals) @ k.` The general form of the mapping annotation is `@ integer_expression.` By default, a component is mapped to the same location as its enclosing composite component and the top-level component is mapped to processor 0. Processor 0 is by default the processor location at which the program is initiated. Effectively, we are using the positive integers to denote logical processors. These logical processors are then mapped to the actual physical workstations by the Regis execution environment. A user may specify a detailed mapping by supplying a mapping file, however, more usually it is left to the environment to select an appropriate set of workstations. The Regis execution environment maintains a set of candidate workstations and allocates these to programs based on the current CPU loading of those workstations. The load sharing algorithm is outlined in [9]. A program such as *calcpi* is initiated by a user on his local workstation and this then requests extra workstations when necessary. Currently, the entire program code is loaded at each workstation, but only those components required at that location are instantiated. The Regis execution environment is implemented by a daemon (***red*** - Regis Execution Daemon) process running at each candidate workstation. The daemon copies the program code where necessary and insures that protection is not violated. Figure 9 illustrates an execution scenario in which the command **calcpi 3** executed at a workstation named *skid* causes execution at three workstations (*skid, bench & water*).

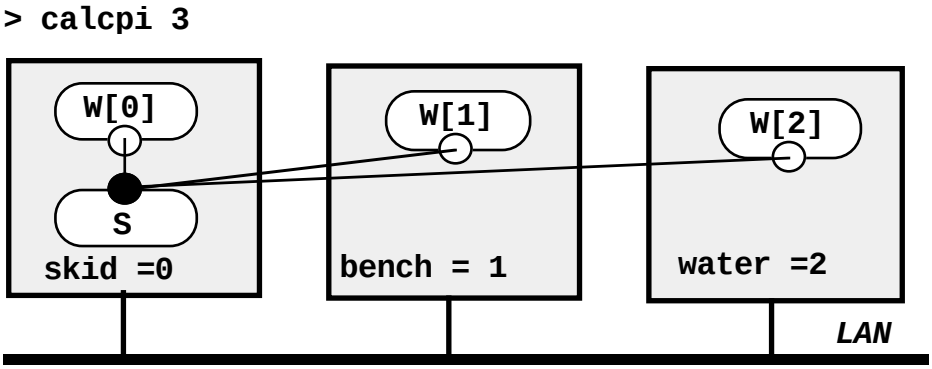


Figure 9 - Program to calculate approximate value of π

We have chosen integer expressions as the most general way of expressing the mapping of a Regis program to the underlying multicomputer. A user may encapsulate a particular set of mappings by implementing these expressions as C++ functions. For example, on a torus connected network, it would be natural to have the functions *north(), south(), east()* and *west()* which would be used to locate components relative to each other. To facilitate programming these abstractions, Regis supplies a standard function *here()* which returns the logical processor location of the enclosing component. Any level of the Darwin configuration program may be annotated with mapping expressions. Consequently, composite components complete with mappings can be developed as part of a library of distributed components. Separating the partitioning of the program structure for distribution using logical locations as

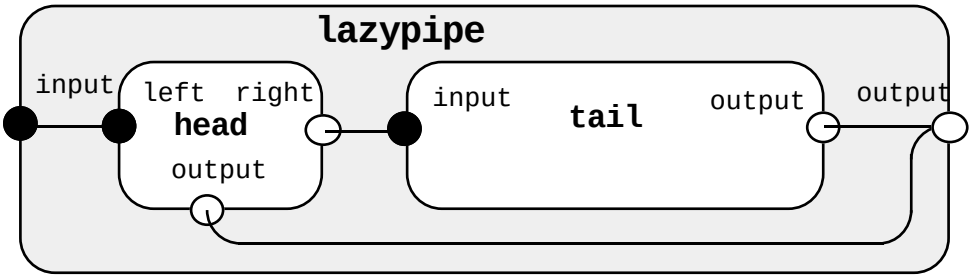
opposed to directly specifying physical machine addresses gives the execution environment the opportunity to load share and in addition makes Regis programs portable between different workstation environments.

6. Dynamic Configuration

The programs we have described so far have a static structure which is determined by the set of actual parameters given to the program at initiation time. Regis also permits the development of programs in which the process structure changes as execution proceeds. Two extensions to Darwin are required to accommodate programs with dynamic structure. The first is lazy instantiation which together with recursion allows the description of structures whose size is determined dynamically. The second is direct dynamic instantiation of components.

Lazy instantiation

To illustrate lazy instantiation, we will return to the pipeline example of Figure 2. This had a fixed number of *filter* instances determined by the parameter *n*. However, suppose that we did not know apriori the number of filter elements required, for example, if we wanted to compute primes up to some limit rather than the first N. In this case, the number can only be determined during execution. The program of Figure 10 is a version of the pipeline component defined recursively which extends as it executes.




```
component lazypipe {
  provide input;
  require output;
  inst
    head:filter;
    tail:dyn lazypipe @ (here()+1)%MAX;
  bind
    input -- head.left;
    head.right -- tail.input; // use triggers tail instantiation
    head.output -- output;
    tail.output -- output;
}
```

Figure 10 - Lazily instantiated pipeline component

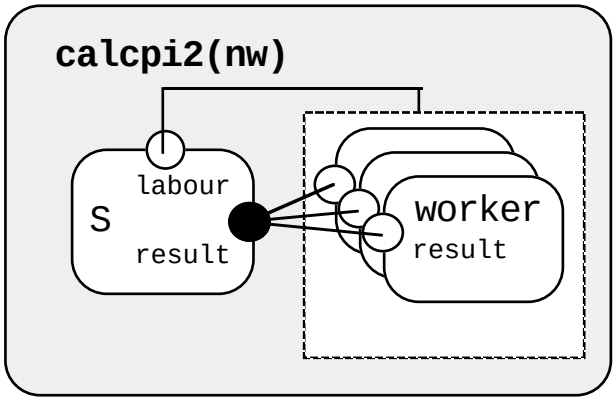
The lazily instantiated pipeline is defined as having a *head filter* component and a *tail* component which is itself a *lazypipe*. The *tail* instance declaration is prefixed by the keyword ***dyn*** indicating that this component should not be immediately created at program initiation time. The *tail* component creation is triggered by the head component sending a message to it. That is, use of the binding *head.right -- tail.input* causes the tail to be instantiated. If used to compute prime numbers up to some limit, the *lazypipe* component would create a *filter* instance for each prime found[2] . The program could be terminated by a *print* component in a similar fashion to that used in the program of Figure 7. Figure 10 illustrates the use of the *here()* function. Each tail is created at the next location (modulo MAX) to the current location. The combination of lazy instantiation and recursion can be used to describe a wide range of commonly occurring diustributed parallel processing structures (e.g. search trees, combining trees, divide & conquer). The advantage of this technique of specifying dynamic structure is that the configuration description is a precise specification which describes the potential structure at execution time. The components used in the structure need not be aware of whether they are being used in a statically or lazily elaborated structure.

Dynamic instantiation

Lazy instantiation, although elegant where applicable, is generally only useful in conjunction with recursive structures. It does not permit parameters to be passed to newly created instances. To allow the full generality of dynamic configuration, Regis permits components, both composite and computational, to be created directly at execution time. Figure 11 gives a dynamically instantiated version of the *calcpi* program of Figure 8. While the *worker* component remains the same, the *supervisor* component is modified to instantiate the workers directly. To create components directly, a component must specify that it requires that service. In Figure 11, the *supervisor* component specifies this as: `require labour <component,int,int,int>`. This is bound in the *calcpi2* program to the system provided object ***dyn*** *worker* which creates new instances of type worker when it is invoked. The supervisor invokes worker instantiation by the call `labour.inst(...)`. The location at which the new worker instance is to be created is specified by the `labour.at(i)` call. It should be noted that Figure 11 includes the type specific binding `worker.result -- S.result`. This binding applies to all instances of type worker created within the composite component *calcpi2*. That is, when a new instance of worker is created, it will automatically be bound to the supervisor instance. Type specific bindings are useful in contexts other than dynamic instantiation. They can be used to specify default bindings which are over-ridden by instance specific bindings. For example, we could have used type specific bindings to specify that all

[2] It should be noted that in a more realistic program the number of filter components created could be dramatically decreased by passing each filter the limit as a parameter and including a test to insure that a message was only sent to the right(thus creating a new instance) if its value squared was less than the limit.

instances of type *filter* in the program of Figure 2 had their output bound to the enclosing component interface *output*. .




```
component supervisor(int nw){
 provide
   result <port,double>;
 require
   labour <component,int,int,int>;
}

component worker(int id, int nw,
                 int intervals){
  require result <port,double>;
}

component calcpi2(int nw){
  inst
    S:supervisor(nw);
  bind
    worker.result -- S.result;
    S.labour -- dyn worker;
}
```

```
supervisor::supervisor(int nw){
 const int intervals=400000
 double area=0.0;
 for (int i=0; i<nw; i++) {
  labour.at(i);
  labour.inst(i,nw,intervals);
 }
 for (int i=0; i<nw; i++) {
  double tmp;
  result.in(tmp);
  area+=tmp;
 }
 printf("Approx pi %20.15lf\n",area);
 exit();
}
```

Figure 11 - Dynamically instantiated components

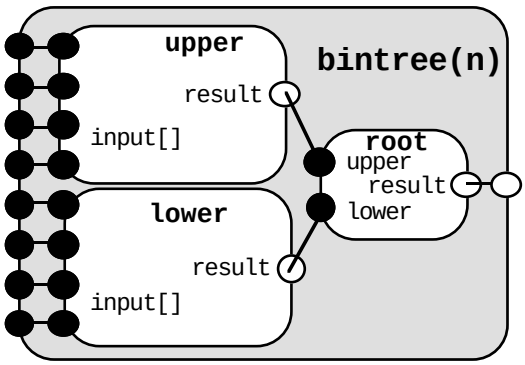
It is worth noting that the configuration description *calcpi2* is shorter than that of the original statically configured *calcpi*. In using direct dynamic instantiation, we have moved information from the structural description into the supervisor computational component. In general, we have found this reduces the clarity and thus comprehensibility of Regis programs. Where possible, we prefer to describe structure explicitly. Direct dynamic instantiation is thus a compromise which permits dynamic structures while retaining some information in the configuration description of the sort of structure being created. Dynamic instantiation is usually used in combination with sending remote references in messages to dynamically establish bindings. Again, this serves to obscure program structure, but is necessary for very dynamic and irregular structures. We have found that it is usually the case that more manageable

programs result from restricting dynamic bindings to temporary transactional relationships between component while describing more permanent relationships explicitly in the configuration description.

7. Generic Components

One of the primary considerations in the design of Regis and its subsequent development was the requirement that commonly used parallel and distributed computing paradigms and functions should be capable of description in a reusable form. We believe that this is crucial in reducing the complexity of this form of programming. A user should not begin programming from the basic elements each time but should be able to select components from a library. Components are specified in a way which makes it easy to plug them into different programs since they specify both the communication objects they require as well as those they provide. Internally they use only local names to communicate with the external environment. We term this property *context independence* since it allows components to be developed independently of the context in which they will execute. However, we would liked to be able to reuse the structural forms or skeletons of parallel programs as well as overall parallel computational objects. To this end, the Darwin configuration language allows the description of generic components which can be parameterised with component types. For example, Figure 12 gives the generic form of a binary tree component with *n* inputs and one output.

```
component bnode {
   provide upper <<alpha>>,
           lower <<alpha>>;
   require result <<alpha>>;
}
```




```
component bintree(int n, component bnode) {
  provide input[n]; require result;
  when n==1
    bind result -- input[0];
  when n>1 {
    inst
     root: bnode;
     upper:bintree(n/2,bnode);
     lower:bintree(n/2+n%2,bnode)@here()+n/2;
    bind
     root.result -- result;
     upper.result -- root.upper;
     lower.result -- root.lower;
    forall i:0..n/2-1
     bind input[i] -- upper.input[i];
    forall i:0..n/2+n%2-1
      bind input[i+n/2] -- lower.input[i];
  }
}
```

Figure 12 - Generic binary tree component

Any component which conforms to the description of *bnode* in Figure 12 can be substituted for

it in the *bintree* component at instantiation time. The type of communication nodes provided and required by *bnode* have been specified using a type variable <alpha>. The description simply states that a component will conform to *bnode* if it provides two communication objects and requires one communication objects and that the types of these three are compatible. The Darwin compiler incorporates a simple form of the polymorphic type checking algorithm that ensures that any component substituted into *bintree* must be compatible with the *input[]* and *result* objects provided and required by the component. The *bintree* generic component can be used to construct the parallel form of any binary associative operator. For example, a component defined as follows:

```
component add {
   provide A<port,int>, B<port,int>;
   require sum<port,int>;
}
```

which produced the sum of it inputs *A* and *B* on its output *sum* could be used to construct an adder *paradd* which summed its N inputs in $O(\log_2 N)$ time by the declaration:

```
inst paradd:bintree(N,add);
```

Generic components describe not only a logical interconnection structure but also information on how that structure should be mapped to the physical system. For example, the component of Figure 12 has a simple mapping expression which ensures that nodes at the same depth of the tree execute on different processors.

8. Conclusion

The paper has given an overview of the Regis support environment for parallel and distributed programs. The environment is distinguished from its predecessors CONIC and REX firstly, by the ability to easily incorporate different interaction mechanisms through the general support for communication objects and secondly, by the support included for dynamic configuration. To date, we have developed classes to support many to one communication (e.g. ports) and one to many communication (e.g. streams are provided by one component and may be read by many components). Work is in progress to provide support for reliable group multicast communication. The goal of this work is to allow configurable highly available systems[10] to be programmed in Regis.

Darwin configuration programs are elaborated in the Regis environment by a highly parallel and distributed algorithm. In contrast, the REX implementation of Darwin used a sequential algorithm. The new parallel algorithm permits an elegant and efficient implementation of both lazy and direct dynamic instantiation. It can be observed that configuration programs of simple functional components which communicate using streams and which use only static or lazy instantiation are equivalent to first order functional programs (cf. Kahn and MacQueen dataflow[11]). We intend to formally examine the relationship between the configuration

language and other programming paradigms using the π calculus semantics specified for Darwin[12].

The use of a configuration language naturally poses the question as to whether the activity of structuring parallel programs would be better accomplished using a graphics based tool. Our experience with ConicDraw[13], a visual programming tool for Conic, suggests that graphic representations are valuable as an aid to comprehension and as a framework to meaningfully display status and performance data on executing programs. However, visual programming of large regular graph structures is a tedious activity best left to the concise descriptions afforded by a textually based configuration language. The developers of the Poker environment, a visual programming environment for distributed memory parallel programs, have also met this problem[14]. However, Poker does not have a textually based configuration language. Our current approach is to develop a graphics based program design tool in which the textual description can be generated from a graphic description and vice-versa[15]. Each component of a program may thus be designed and viewed in whichever representation a developer is most comfortable with.

Regis has been in use at Imperial College since November 1992. It is used in both undergraduate and postgraduate laboratories for distributed and parallel programming courses and as a vehicle for postgraduate research. It has proved invaluable in the design of laboratory exercises since it allows instructors to constrain the solution space for an exercise by specifying both the structure of the overall solution and key component interfaces. It is also proving a useful research tool since the approach of separating configuration, communication and computation allows researchers to quickly construct a framework in which to test their ideas. It is planned in the near future to make a version of Regis publicly available via anonymous FTP. Our previous experience with disseminating the Conic toolkit demonstrated convincingly to us that the benefits of user feedback outweigh by far the disadvantages and workload involved in dealing with user queries and problems.

Acknowledgements

The authors would like to acknowledge discussions with our colleagues in the Parallel and Distributed Systems Group during the formulation of the ideas behind Regis, Stephen Crane for helping with the design and implementation and Kevin Twidle for implementation of the Regis execution environment. We gratefully acknowledge the DTI (Grant Ref: IED 410/36/2) for their financial support.

References

[1] J.Kramer, J.Magee, "Dynamic Configuration for Distributed Systems", IEEE Transactions on Software Engineering, SE-11 (4), April 1985, pp. 424-436.

[2] 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.

[3] M. Barbacci, C. Weinstock, D, Doubleday, M. Gardner and R Lichota, “Durra: a structure description language for developing distributed applications”, IEE Software Engineering Journal, Vol. 8, No. 2, March 1993, pp83-94.

[4] C. Hofmeister, E. White and J. Purtillo, “ Surgeon: a packager for dynamically reconfigurable distributed applications”, IEE Software Engineering Journal, Vol. 8, No. 2, March 1993, pp95-101.

[5] J. Magee, J. Kramer and M. Sloman, "Constructing Distributed Systems in Conic" IEEE Transactions on Software Engineering, SE-15 (6), 1989.

[6] J. Kramer, J. Magee, M.Sloman, N.Dulay, "Configuring Object-Based Distributed Programs in REX", IEE Software Engineering Journal, Vol. 7, 2, March 1992, pp139-149.

[7] J. Magee, N. Dulay and J. Kramer, “Structuring Parallel and Distributed Programs”, IEE Software Engineering Journal, Vol. 8, No. 2, March 1993, pp73-82

[8] T.G. Lewis and H. El-Rewini, “Introduction to Parallel Computing”, Prentice- Hall International Editions, 1992.

[9] O. Kremien, J. Kramer and J. Magee, “Scalable Load-Sharing for Distributed Systems”, Proc. of HICSS-26, Hawaii, Jan 1993, pp632-641.

[10] F. Cristian, “Automatic reconfiguration in the presence of failures”, IEE Software Engineering Journal, Vol. 8, No. 2, March 1993, pp53-60.

[11] G. Kahn and D. MacQueen, “Coroutines and Networks of Parallel Processes”, Information Processing 77, edited by B. Glichrist, North-Holland.

[12] S. Eisenbach, R. Paterson, “π-Calculus Semantics for the Concurrent Configuration Language Darwin”, HICSS-26, Hawaii, January 1993.

[13] J. Kramer, J. Magee and K. Ng,. (1989). "Graphical Configuration Programming", IEEE Computer, 22(10), 53-65.

[14] D. Notkin, L. Snyder, D. Socha et al., “ Experiences with Poker”, Proc. of ACM/SIGPLAN PPEALS, pp 10-20, July 1988.

[15] J. Kramer, J. Magee, K. Ng and M. Sloman, “The System Architect’s Assistant for Design and Construction of Distributed Systems”, To be published in Proceedings of the Fourth IEEE Workshop on Future Trends of Distributed Computing Systems, Sept. 2-24, 1993, Lisbon, Portugal, IEEE Computer Society Press.