

Regis: A Constructive Development Environment for Distributed Programs

Jeff Magee, Naranker Dulay, Jeff Kramer

Department of Computing, Imperial College, London SW7 2BZ, UK.

Abstract

Regis is a programming environment aimed at supporting the development and execution of 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 distributed programs using the configuration language Darwin. The paper describes programming in Regis through a set of small example programs drawn from the implementation of an Active Badge system.

Keywords

configuration programming, dynamic reconfiguration, inter-process communication, distributed 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 and 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] we 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 succeeded Conic, permitted the implementation of computational components in a range of sequential programming languages; however, these components were also restricted to a fixed set of intercommunication primitives. The Regis system described here removes the latter restriction by allowing components to interact through user specified communication primitives. In essence, Regis allows the separation of 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. In contrast, the configuration language provided in the REX system 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 of only 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 accommodate 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. The paper describes the basic elements of the Regis environment using a set of example programs. These examples are taken from an Active Badge[8] system implemented by one of the authors in the Regis programming environment using hardware from Olivetti Research Laboratories in Cambridge. Active Badges emit and receive infrared signals which are

received/transmitted by a network of infrared sensors connected to workstations. The system permits the location and paging of badge wearers within a building.

In the following, we look at configuration, communication and computation in the Regis environment and then see how these elements are combined to form distributed programs. The more advanced features of the Regis environment concerned with dynamic and open systems binding are then examined before we conclude by discussing our experience with using the environment.

2. Program Configuration

As mentioned, 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

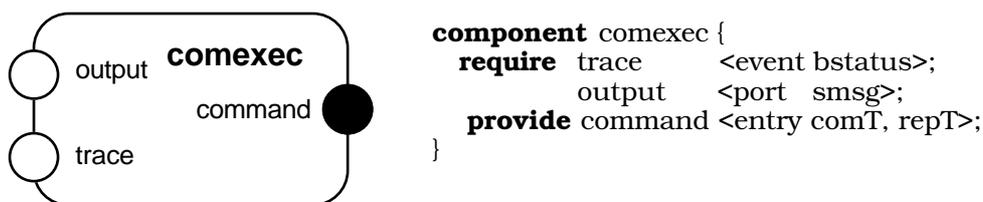


Figure 1 - Component Type

Darwin views components in terms of both the services they provide to allow other components to interact with them and the services they require to interact with other components. The component of Figure 1 provides one service (depicted by a filled in circle) and requires two external services to support that service (the empty circles). The service provided is badge command execution. Commands are issued to a badge to set off its internal beeper or to illuminate its status LEDs. The Darwin component interface specifies the set of services required and provided by a component together with the types of these services

(enclosed in angle brackets). By convention, the first word of the type specification is the interaction mechanism class which has been used to implement the service. For example, *command* accepts *entry* calls with a request of type *comT* and a reply of *repT*. To execute a command, it is necessary to first locate a badge. Consequently, the command execution badge requires the *trace* service. Location information in the badge system is an event stream where an event represents a change of badge location. Consequently, the interaction mechanism for *trace* is *event* and the data type of each event is *bstatus*. Similarly, to execute the command once the badge is found, the component must send a message to the sensor network. The requirement for this service is represented by *output* which uses the Regis *port* message transmission primitives.

Note that the component *comexec* does not need to know the names of external services or where they may be found. It may be implemented and tested independently of the rest of the badge system. We call this property context independence. It permits the reuse of components during construction and simplifies replacement during maintenance.

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 program is a hierarchically structured composite component which, when elaborated at execution time, results in a collection of concurrently (potentially distributed) 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 services required by one component with the services provided by others, can be visualised as filling in the empty circles of a component with the solid circles provided by other components.

The composite component of Figure 2 controls the interface to the network of infrared badge sensors. Each requirement (empty circle) in this example is for a *port* (named *output*) to send messages to, and each provision (filled in circle) is a *port* from which a component receives messages (named *input*). Bindings between requirements and provisions are declared by the Darwin ***bind*** statement. For example, the output of each *poller* component is bound to an input of the multiplexer component *M* by the statement ***bind*** *P[i].output -- M.input[i]*. Requirements which cannot be satisfied inside the component can be made visible at a higher level by binding them to an interface requirement as has been done in the example for multiplexer *M* requirement *output* which is bound to *sensout*.. Similarly services provided internally which are required outside are bound to an interface service provision eg. *sensin--D.input*.

The Darwin compiler checks that bindings are only made between required and provided services with compatible interaction classes and datatypes (in the example that the provided and required service use *port* message passing and that the message type is *smsg*). Note that the interaction and datatype information specified at the component interface is optional in Darwin. Where necessary, the compiler infers the type of interface objects which are not explicitly typed. This enables Darwin descriptions to be concise and reusable.

The *forall* construct of Figure 2 is used to declare an array of *poller* instances and their bindings. Instance arrays may be multi-dimensional; the *array* declaration is used to specify the dimensions and bounds.

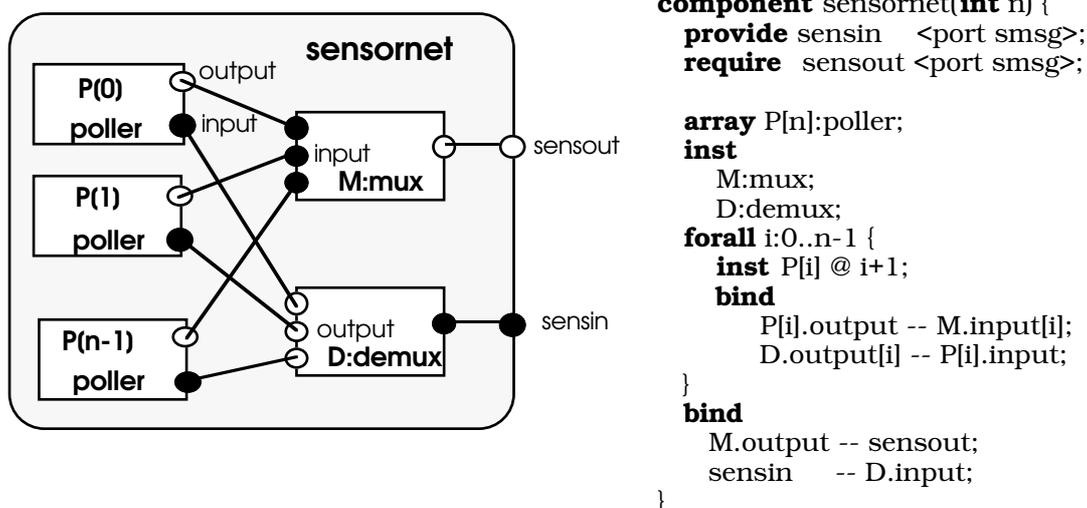


Figure 2 - Composite Component Type

Each *poller* component is located on a different workstation and controls a multi-drop RS232 line of infrared sensors. The poller component requires a service to output badge location sighting messages and provides an input on which to transmit command messages. In general, many requirements may be bound to a single provided service. However, in this case each poller instance output is bound to a separate input port to allow the multiplex component *M* to identify the sensor network in the outgoing message. Pollers are distributed by the expression *inst P[i]@ i+1* which locates each instance *P[i]* on a separate machine *i+1*. The integer machine identifiers are mapped to real workstations by the Regis runtime system as described later in the paper.

From the example, it can be seen that components may be parameterised and that parameters can be used to determine the internal structure of composite components. In this case the parameter determines the number of poller instances.

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 dynamic structures we will examine how communication and computation are catered for in Regis.

3. Communication

Component interaction is supported in Regis by interaction classes defined using the C++ template facility. A service provided by a component is realised by an instance of one of these classes. This instance is remotely referenced by the component which *requires* the service. Figure 3 is the C++ description of the Regis *port* interaction class previously mentioned in relation to Figures 1 & 2.

```
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 environment. Consequently, the implementor of interaction classes in the Regis framework must describe not only the interaction class but also a class which can be used to access objects of that class remotely. These remote access classes are by convention named by adding the suffix *ref* to the name of the interaction 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);

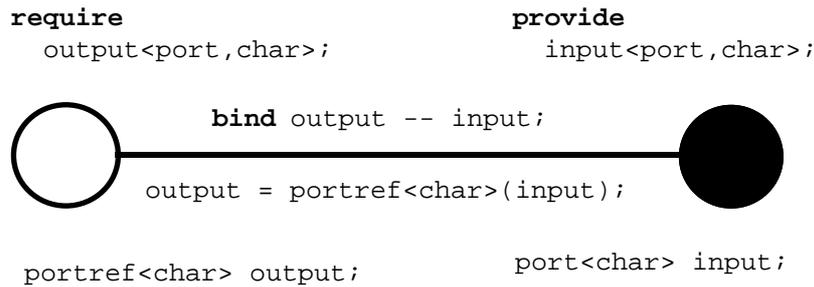
    void out(T &msg);
    void outv(T msg[],int n);

    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.

Darwin



C++

Figure 5 - Interaction Objects in Darwin & C++

For computational components, required services are implemented by reference objects and provided services by interaction objects. 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. Regis provides a light-weight reliable transmission protocol for communication between remote processes. This ensures that messages are not lost due to transient link failures and that a FIFO delivery order is maintained. We have chosen not to use the TCP protocol since most implementations of UNIX limit the number of open TCP connections for a process to a small number. The figures of Table 1 for remote communication are for the situation in which no transmission errors occur. 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.send(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 interaction template classes which currently includes bidirectional request-reply entries, events, streams and tuple spaces in addition to ports. We have chosen to use interaction 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 an interaction class and a remote reference class, we have the opportunity to optimise the communication between objects of these classes. Figure 6 which summarises the *event* interaction class illustrates that the data transfer direction is not necessarily from the required to the provided service as with ports and that the model can support one-to-many communication in addition to many-to-one communication. Clients of an event service receive event messages (using *wait*) with the *Eclass* for which they have been enabled. For example in the badge system, *Eclass* is the badge identity and a client component can choose to receive location events concerned with a particular badge. The event mechanism is implemented by passing references to client private ports to the event interaction object. These details are hidden from programmers using the event class.

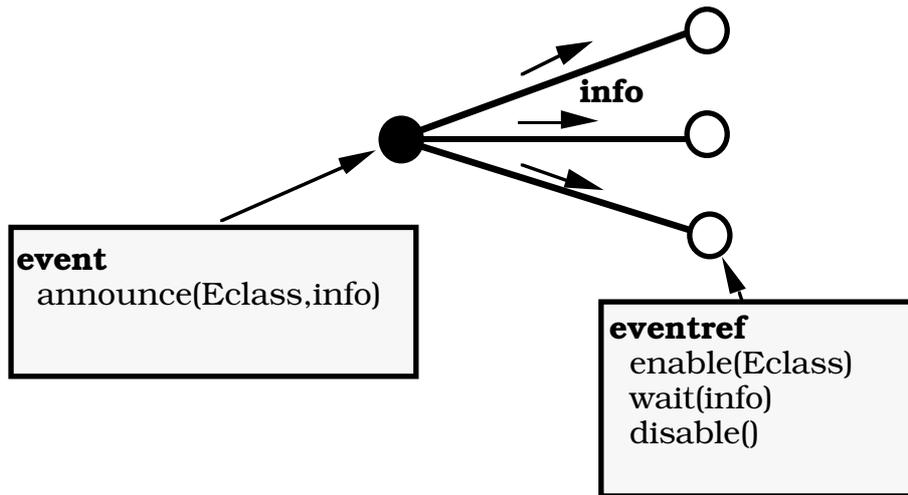


Figure 6 - Event Interaction

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 7 which gives the C++ implementation of the *demux* component used in Figure 2. 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. The *demux* program of Figure 7 receives messages of type *msg* into the local variable *M*. The message is then sent to an output depending on the network identity embedded in the message.

```

// Darwin interface description
component demux ( ) {
    provide input <port smsg>;
    require output[8] <port smsg>;
}



---



// generated from Darwin description
class demux : public process {
public:
    port<smsg> input;
    portref<smsg> output[(8)];
    demux();
};



---



// constructor implements the computational function
demux::demux( ){
    for(;;) {
        smsg M;
        input.in(M);
        output[M.network].send(M);
    }
}

```

Figure 7 - Demux C++ computational component

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 communicate via interaction objects as described in the previous section. The body 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 interaction 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.

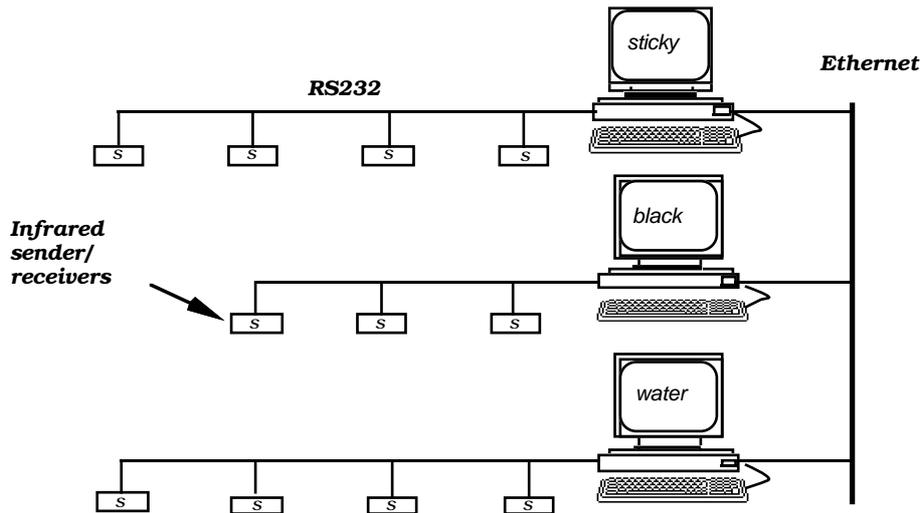


Figure 8 - Active Badge system Sensor Network

The program of Figure 2 uses the mapping annotation *inst P[i]@ i+1* which places each poller instance *P[i]* on a separate processor numbered *i*. 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 or it may be 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]. However, in the Active Badge system, a mapping file must be specified so that pollers execute on the workstations to which sensors are connected. Consequently to map the program of Figure 2 to the hardware configuration of Figure 8, the following mapping file must be specified:

```
map 1 "sticky.doc.ic.ac.uk"
map 2 "black.doc.ic.ac.uk"
map 3 "water.doc.ic.ac.uk"
```

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 ensures that protection is not violated. The mapping file may also include information on the resources and environment required on a remote processor (e.g. specifying the location of an Xwindow server for displays). *Red* creates the specified environment and acquires the required resources before creating the remote process.

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. For example, the number of poller component instances from Figure 2 is determined by the actual value specified for its parameter *n* at instantiation time. Regis also permits the development of programs in which the process structure changes as execution proceeds. In particular, components whether composite or primitive may be instantiated as a result of requests from other components. Figure 9 shows an example from the Active Badge system of the implementation of command execution composite component *comexec*. A *badge* component is created to deal with each new request received by *comexec*. This *badge* component deals with locating the physical badge, reserving the nearest sensor to transmit the infrared command and implements a protocol to ensure that commands are reliably executed. As mentioned in the foregoing, commands are issued to a badge to set off its internal beeper or to illuminate its status LEDs.

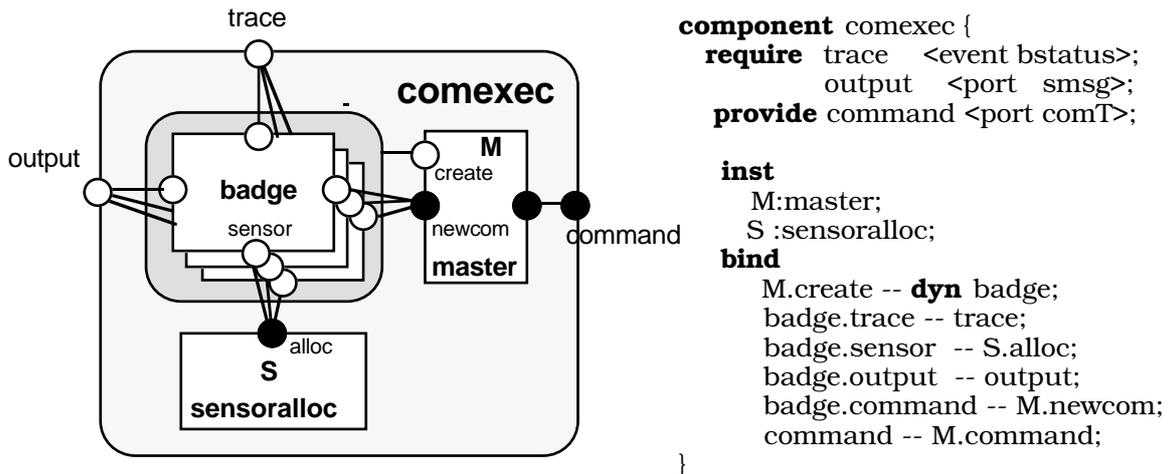


Figure 9 - Dynamic component instantiation

The *master* component *M* is responsible for creating badge components. It has a requirement for a dynamic instantiation service (*dyn*) which passes a single parameter of type *int* as shown below in the Darwin interface specification for *master*.

```

component master {
  require create <dyn int>;
  .....
}

```

The parameter specifies the identity of the physical badge that the newly created component will deal with (ie **component** badge(int id) {...}). The *master's* requirement is satisfied in the configuration program of Figure 9 by binding it to the component type *badge* prefixed by the keyword *dyn*. ie. *M.create -- dyn badge*. The implementation of the *master* component type is shown in Figure 10 illustrating the way new component instances are invoked. Dynamically created components and indeed all components terminate when they exit their constructor body. Subsequently, their storage is garbage collected. If it was required that the *badge* component be created on another processor and not co-located with the *master* then the statement *create.at(i)* would have been inserted in the code before the request to create a new instance.

```

master::master() {
  for(;;) {
    comT B;
    command.in(B); // receive new command
    create.inst(B.id); // create new instance
    output.announce(B.id,B); // send command to the new instance
  }
}

```

Figure 10 - Dynamic component invocation

Note that in Figure 9, bindings are specified for the component type *badge* rather than for instances of this type as is usually the case. These type specific bindings serve to define the environment in which the dynamically created instances of *badge* will execute. The interfaces for dynamically created components types may only usefully declare a requirement for services. Since dynamically created instances are essentially anonymous, it would not be possible within Darwin to declare bindings to services they provided. Dynamically created components may provide services; however, access to these services is achieved by passing references to interaction objects in messages to form bindings dynamically. These bindings cannot be captured by the configuration description. 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 components while describing more permanent relationships explicitly in the configuration description. 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. Type specific bindings have a more general use than that described above. In essence, they describe the default bindings for all instances of a given type, whether declared statically or created dynamically, within the scope of a composite component. This default may be overridden by the bindings declared for a specific instance. This facility has proved useful in programming concise configuration descriptions.

In addition to the dynamic instantiation facility described above, Regis also provides a mechanism termed lazy instantiation. Component instances may be declared in the configuration program, however, they are not actually instantiated until another component tries to use one of the services they provide. For example, if in Figure 11 the command execution component instance had been declared as:

```
inst C: dyn comexec;
```

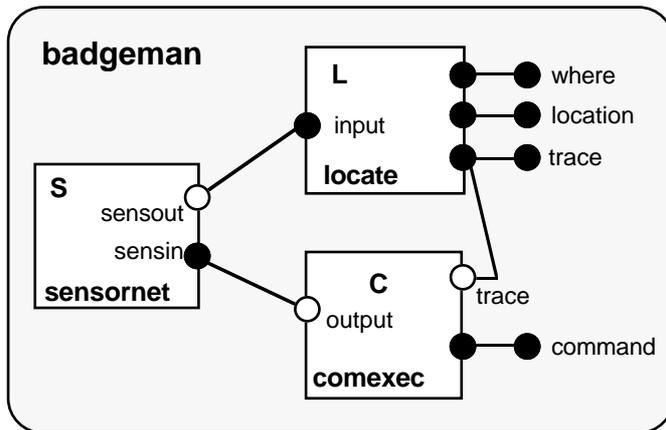
then the instance C would not be created until the command service it provides is requested by an external client. In this way, potential configurations can be expressed, however, the resources need not be allocated until the service provided by that configuration is required. Lazy instantiation captures the requirement in distributed programs for servers to be create on demand (cf. Unix *Inetd*).

7. Open Systems Binding

The *sensornet* and *comexec* components of Figures 2 & 9 respectively, form part of a badge manager server. This server provides a set of services, listed below, which are required by user instantiated client programs:

where	returns current location of all badges,
location	badge events, selected by location,
trace	badge events, selected by badge identity.
command	to execute a command on a badge.

These services are **exported** from the *badgeman* component of Figure 11 and advertised in an external namespace. When the component *badgeman* is executed, the services named in the export system are registered in a name server by recording the name of the service (e.g. “badge/where”) and the remote reference to the interaction object which supports that service. The *bind* statement is used to associate exports with services provided by component instances (e.g. *bind where -- L.where*).



```

component badgeman {
  export
    where    @ "badge/where",
    location @ "badge/location",
    trace    @ "badge/trace",
    command @ "badge/command";
  inst
    S:sensornet(3);
    L:locate;
    C:comexec;
  bind
    where -- L.where;
    location -- L.location;
    trace -- L.trace;
    command -- C.command;
    S.sensout -- L.input;
    C.output -- S.senin;
    C.trace -- L.trace;
}

```

Figure 11 - Exporting services

Darwin contains an *import* construct which is used to bind to externally provided or exported services. So for example, a client program which displays the location of badge wearers contains a statement to import the “badge/where” service as shown in Figure 12. Again bindings are type checked, this time dynamically. Type descriptions are recorded in the name service along with the service name and reference.

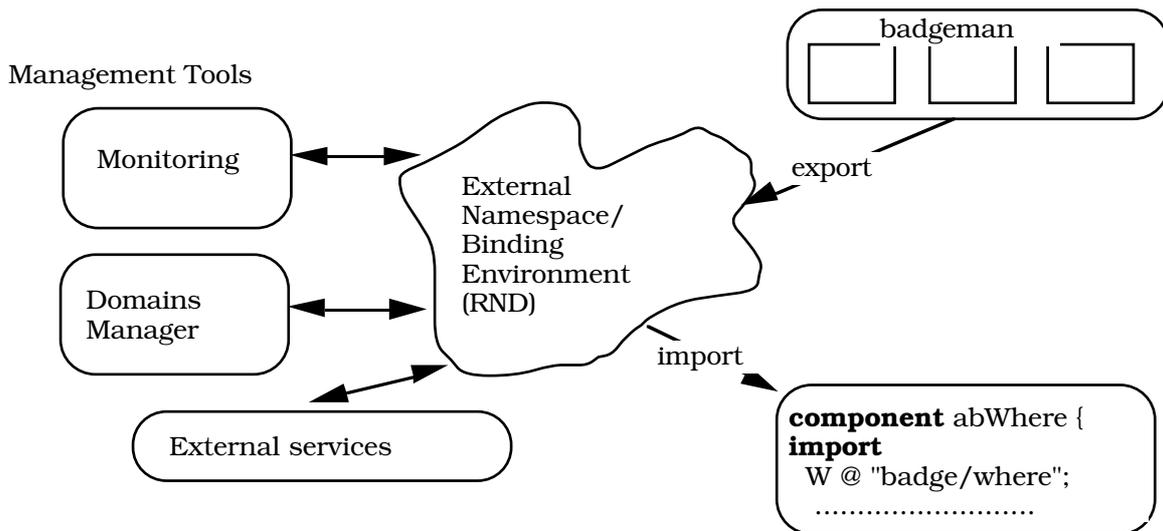


Figure 12 - Open Systems Binding Environment

Currently, the name service is provided by a program written using Regis (RND - Regis Name Daemon), however, there is no reason why Regis programs should not interact with other name servers such as the ANSA Trader. Work has been carried under the SYSMAN project [10] in using Darwin to structure ANSA programs. In addition as shown in Figure 12, the open systems binding mechanism is used to allow a range of tools to interact with Regis programs. The Domain Management tool will allow the dynamic configuration management of Regis programs. It exploits the fact that any Regis service can be exported including dynamic component instantiation. This together with a third part binding service, which is simply a new type of interaction object, has allowed external dynamic management of configurations to be easily integrated with the system [11].

8. Conclusion

The paper has given an overview of the Regis support environment for distributed programs. The environment is distinguished from its predecessors CONIC and REX firstly, by the ability to easily incorporate different communication mechanisms through the general support for interaction 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 entries) and one-to-many communication (e.g. event streams are provided by one component and may be received by many components). Work is in progress to provide support for reliable group multicast communication. The goal of this work is to exploit the Regis configuration facilities to support configurable highly available services [12] through the use of replicated servers programmed in Regis.

As described, Darwin provides only lazy component instantiation and direct dynamic component instantiation as the methods of programming dynamic structures. Components are expected to terminate when they complete their computation and are then garbage collected. This constructive approach avoids many of the ordering problems associated with operational languages which include destructive operations such as unbind and component deletion. The result is that Darwin is concise declarative language with a clear and precise semantics [13]. For instance, its declarative nature has made it relatively easy to provide a parallel elaboration algorithm for Darwin specifications and to enable us to reason about its correctness. Other systems which employ configuration languages have concentrated on other aspects. For instance, the module interconnection language of POLYLITH [4] employs a software bus for interconnection and communication. It supports component heterogeneity for mixed-language programming. Dynamic reconfiguration is supported using Clipper [14], an associated operational language (based on C++).

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 [15], 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 [16]. 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 [17]. 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. A version of Regis is publicly available via anonymous FTP ([dse.doc.ic.ac.uk](ftp://dse.doc.ic.ac.uk)).

Acknowledgements

The authors would like to acknowledge discussions with our colleagues in the Distributed Software Engineering Section 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 and Olivetti Research Laboratories (Cambridge) for their donation of the Active Badge hardware.

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, pp 83-94.
- [4] J. Purtillo, "The POLYLITH Software Bus", ACM Transactions on Programming Languages and Systems, 16 (1), January 1994, pp 151-174.
- [5] J. Magee, J. Kramer and M. Sloman, "Constructing Distributed Systems in Conic" IEEE Transactions on Software Engineering, SE-15 (6), 1989, pp. 663-675.
- [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] Harter A., Hopper A., "A Distributed Location System for the Active Office", IEEE Network, Jan./Feb. 1994, pp. 62-70.
- [9] O. Kremien, J. Kramer and J. Magee, "Scalable and Adaptive Load Sharing for Distributed Systems", IEEE Parallel and Distributed Technology , 1 (3), August 1993, pp. 62-70.
- [10] M. Sloman, J. Magee, K. Twidle , J. Kramer, "An Architecture for Managing Distributed Systems", Proc. 4th IEEE Workshop on Future Trends of Distributed Computing Systems, Lisbon, Sep. 1993, pps. 40-46.
- [11] S. Crane, N. Dulay, H. Fosså, J. Kramer, J. Magee , M. Sloman, K. Twidle ,, "Configuration Management for Distributed Systems", ICSTM.CM 1.1 V2.1, Department of Computing Research Report, 1994.
- [12] F. Cristian, "Automatic reconfiguration in the presence of failures", IEE Software

Engineering Journal, Vol. 8, No. 2, March 1993, pp53-60.

- [13] J. Magee, J. Kramer and S. Eisenbach, "System Structuring: a Convergence of Theory and Practice?", Dagstuhl Workshop on Unifying Theory and Practice in Distributed Systems, September 1994, to appear.
- [14] B. Agnew C. Hofmeister, J. Purtillo, "Planning for Change: A Reconfiguration Language for Distributed Systems", this issue.
- [15] J. Kramer, J. Magee and K. Ng,. (1989). "Graphical Configuration Programming", IEEE Computer, 22(10), pp. 53-65.
- [16] D. Notkin, L. Snyder, D. Socha et al., " Experiences with Poker", Proc. of ACM/SIGPLAN PPEALS, pp 10-20, July 1988.
- [17] J. Kramer, J. Magee, K. Ng and M. Sloman, "The System Architect's Assistant for Design and Construction of Distributed Systems", Proceedings of the Fourth IEEE Workshop on Future Trends of Distributed Computing Systems, Lisbon, Portugal, Sept. 1993, pp. 284-290.