System Structuring: A Convergence of Theory and Practice?

Jeff Magee, Susan Eisenbach & Jeff Kramer
Department of Computing
Imperial College
London SW7 2BZ

jnm,jk,se@doc.ic.ac.uk

Abstract

Darwin is a general purpose structuring tool of use in building complex distributed systems from diverse components and diverse component interaction mechanisms. It is in essence a declarative binding language which can be used to define hierarchic compositions of interconnected components. Distribution is dealt with orthogonally to system structuring. The language allows the specification of both static structures and dynamic structures which evolve during execution. The central abstractions managed by Darwin are components and services. Bindings are formed by manipulating references to services.

The paper describes the operational semantics of Darwin in terms of the π-calculus, Milner's calculus of mobile processes. The correspondence between the treatment of names in the π-calculus and the management of service references in Darwin leads to an elegant and concise π-calculus model of Darwin's operational semantics. The model has proved useful in arguing the correctness of Darwin implementations and in designing extensions to Darwin and reasoning about their behaviour. The paper discusses the reasons why other formalisms fail to capture elegantly the system structuring concepts on which Darwin is based.

1. Introduction

Our research is concerned with the provision of sound and practical means for the construction of distributed systems. It has resulted in the development of configuration languages as a means of describing and subsequently managing system structure. The languages we have developed have in common the notion of a component as the basic element from which systems are constructed. Complex components are constructed by composing in parallel more elementary components and as a result, the overall structure of a system is described as a hierarchical composition of primitive components which at execution time may be located on distributed computers. These primitive components have a behavioural description as opposed to a structural description. Their behaviour is usually specified in a conventional sequential programming language.

The initial work on system structuring resulted in the CONIC Toolkit [1] which contained a simple configuration language. This language was limited in that it could only be used to specify configurations of primitive components which were written in a special purpose programming language and which could interact by a fixed set of communication primitives. The CONIC configuration language allowed the definition of system structures which were fixed at system instantiation time but which could be modified by an external configuration manager. The manager was responsible for ensuring the safety of changes it made. The

successor to CONIC was developed in the context of the REX project[2]. This configuration language permitted components to be implemented in a range of programming languages but still limited component interaction to a fixed set of communication primitives. The REX configuration language allowed a programmer to specify arbitrary changes to the initially specified system structure. These changes could result in inconsistent and unsafe system structures. The configuration languages of both CONIC and REX had a centralised implementation. The configuration description was elaborated at a single node which then issued commands to instantiate and bind components on remote nodes.

Darwin[3] is the latest in this line of configuration languages. Darwin is a declarative language which is intended to be a general purpose tool which can be used to configure systems from diverse components using diverse interaction mechanisms. It is currently being used in the context of the Regis system[4] which supports multiple interaction primitives and with ANSAware[5] which uses remote object invocation for component interaction. Darwin allows the description of both static structures fixed during system initialisation and dynamic structures which evolve as execution progresses. It does not support the arbitrary change operation incorporated by REX but allows interaction with external management agents. In contrast with its predecessors, CONIC and REX, Darwin has a distributed implementation which permits the concurrent elaboration of a system. It allows physical distribution to be specified completely orthogonally to logical structure. In common with a number of similar systems[6,7,8], REX and CONIC permitted distribution only at the top-level of the system configuration description.

The goal that Darwin be general purpose requires that there should be a clear and well specified division of responsibilities between Darwin and the primitive components it configures. The goal that Darwin should be capable of concurrent evaluation demands that there must be a clear and unambiguous specification of Darwin's operational behaviour against which implementations can be validated. We have attempted to satisfy both these requirements by modelling Darwin in the π-calculus[9], Robin Milner's calculus of mobile processes.The reasons for choosing this formalism are discussed in the concluding sections of the paper. The remainder of the paper provides an introduction to both Darwin and the π-calculus and then outlines how Darwin is modelled in the π-calculus and demonstrates how this model can be used to prove some properties of Darwin configurations. In later sections, the paper describes how more advanced features of Darwin concerned with dynamic configuration are modelled without invalidating the properties demonstrated for the basic model. The paper concludes with a critical evaluation of the work.

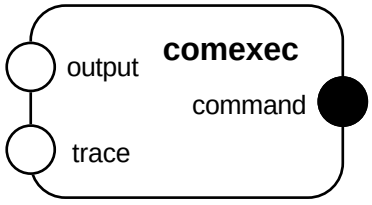
2. Darwin

A distributed program consists 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. The configuration language Darwin provides such a structuring tool. Darwin allows distributed 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 primitive computational components and these in turn can be configured into more complex composite types. In the following, Darwin examples are taken from an Active Badge[10] 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.

2.1 Components and Services

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 specification specifies the set of services required and provided by a component together with types of these services (enclosed in angle brackets).




```
component comexec {
  require  trace      <event bstatus>;
           output     <port  smsg>;
  provide  command    <entry comT repT>;
}
```

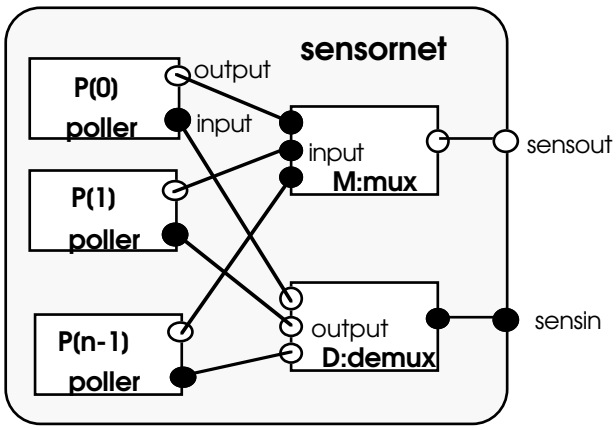
Figure 1 - Component Type

By convention in the Regis system, 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 component 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. When used with ANSAware[5], the angle brackets contain the names of IDL specifications. 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.

2.2 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 sensor.




```
component sensornet(int n) {
  provide sensin   <port smsg>;
  require  sensout <port smsg>;

  array P[n]:poller;
  inst
     M:mux;
     D:demux;
  forall i:0..n-1 {
     inst P[i] @ i+1;
     bind
        P[i].output -- M.input[i];
        D.output[i] -- P[i].input;
  }
  bind
     M.output -- sensout;
     sensin    -- D.input;
}
```

Figure 2 - Composite Component Type

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 e.g. *sensin--D.input.*

The Darwin compiler checks that bindings are only made between required and provided services with compatible types (in the example that the provided and required service use *port* message passing and that the message type is *smsg*). Where necessary, the compiler infers the type of interface objects which are not explicitly typed. 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.

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 runtime system.

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. In addition to replication, Darwin supports conditional configuration.

So far, we have described how the Darwin configuration language may be used to describe static structures of component instances. Before examining more advanced features which permit the description of dynamic structures we will show how this basic subset of Darwin is modelled in the π-calculus.

3. π-calculus

The π-calculus[9] is an elementary calculus for describing and analysing concurrent systems with evolving communication structure. In this paper, we use the simple monadic form of the calculus as described in the following.

A system in the π-calculus is a collection of independent processes which communicate via channels. Channels or links are referred to by name. Names are the most primitives entity in the calculus, they have no structure. There are an infinite number of names, represented here by lowercase letters. Processes are built from names as follows:

action terms *A*::=	$\bar{x}z.P$	Output the name *z* along the link named *x* then execute process *P.*
	$x(y).P$	Input a name, call it *y,* along the link named *x* and then execute *P* (binds all free occurrences of *y* in *P*).
terms *P*::=	$A_1 + \ldots + A_n$	Alternative action $(n \geq 0)$, execute one of *A*. When $n = 0$ the sum is written as **0** and means stop.
	$P_1 \mid P_2$	Composition, P_1 and P_2 execute concurrently. The operation is commutative and associative.
	$(\nu y)\,P$	Restriction, introduces a new name *y* with scope *P* (binds all free occurrences of *y* in *P*).
	$!P$	Replication, provide any number of copies of P. It satisfies the equation $!P \equiv P \mid !P$. Recursion can be coded as replication and so need not be included as a separate method for building processes. Recursion will be used when it makes examples clearer.

Computation in the π-calculus is expressed by the following reduction rule:

$$(\ldots + x(y).P_1 + \ldots) \mid (\ldots + \bar{x}z.P_2 + \ldots) \rightarrow P_1\{z/y\} \mid P_2.$$

Sending *z* along channel *x* reduces the left hand side to $P_1 \mid P_2$ with all free occurrences of *y* in P_1 replaced by *z*. The following is a simple example of applying the reduction rule:

$$\bar{x}z.0 \mid x(y).y(s).0 \rightarrow z(s).0 .$$

For reasons of conciseness, in the remainder of the paper we will omit the stop process.0 in an agent and write $z(s)$ in place of $z(s).0$.

4. Modelling Darwin in the π-calculus

Our purpose in modelling Darwin in the π-calculus is to provide a precise operational semantics for the language. We wish to demonstrate that elaboration of Darwin configuration programs result in the required set of primitive component instances and set of intercomponent bindings at runtime. Further, we wish to demonstrate that this elaboration process is correct when concurrently executed. The model should define precisely that which is the responsibility of the Darwin program and that which must be carried out by the components configured by the Darwin program. For simplicity, in the following, the types of Darwin services and type discipline for binding is ignored. In the concluding section, we discuss how the model may be simply extended to capture the notion of service type in Darwin.

4.1 Requirements, Provision & Binding

In this section, an interpretation in the π-calculus is given for each of the Darwin syntactic constructs concerned with requiring, providing and binding services. With these, we can examine the operational effect of binding a simple configuration which has no hierarchic structure.

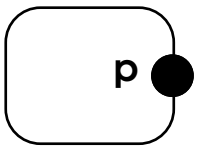
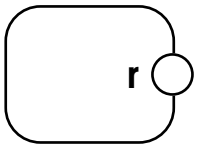
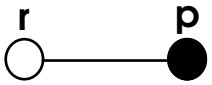
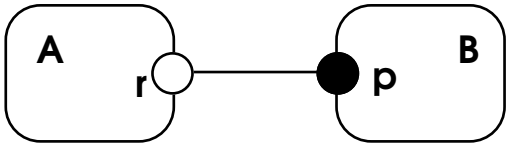
Darwin	π-calculus
 **provide** p;	$PROV(p,s) \stackrel{\text{def}}{=} \; !(p(x).\bar{x}s)$ where *s* - service reference *x* - location at which s is required *p* - access name
 **require** r;	$REQ(r,o) \stackrel{\text{def}}{=} \; r(y).\bar{y}o$ where *o*- location at which service is required *y* - name of service provider *r* - access name
 **bind** r -- p;	$BIND(r,p) \stackrel{\text{def}}{=} \; \bar{r}p$ where *r* - name of requirement *p* - name of provision

Figure 3 - provide, require & bind

From figure 3, it can be seen that the declaration of a provided service, ***provide*** *p,* in Darwin is modelled in the π-calculus as the agent *PROV(p,s)* which is accessed by the name *p* and manages the service *s*. The service *s* is simply the name or reference to a service which must be implemented by a component. Darwin is not concerned with how the service *s* is implemented, it is concerned with placing *s* where it is required by clients of the service. The agent *PROV* thus receives the location *x* at which the service is required and sends *s* to that location. Since there may be more than one client of the service, the agent *PROV* is defined to be a replicated process (!) which will repeatedly send out the service reference each time a location is received.

The declaration of a required service, ***require*** *r,* is modelled by the agent *REQ(r,o)* which is accessed by the name *r* and which manages the location *o* at which the service is required. Again, Darwin is not concerned with how a client component uses a service, it must ensure that a reference to the service is placed at some location in the client component. The *REQ* agent receives the access name to a *PROV* agent and sends the location *o* to that agent. A requirement in Darwin may only be bound to a single service and so the agent *REQ* sends out the location *o* precisely once.

The binding construct in Darwin is modelled by the *BIND* agent which simply sends the access name of the *PROV* agent to the *REQ* agent. It should noted that agents or processes in the π-calculus cannot be directly named, instead agents are accessed by named channels. Although Darwin names instances of components, these names are only used to qualify the names of the service they provide or require. To illustrate this, the simple Darwin configuration of figure 4 is translated into the π-calculus.




```
inst
  A:Client;
  B:Server;
bind
  A.r -- B.p;
```

Figure 4 - Client Server configuration

Each primitive component is represented by an agent which is a composition of the *PROV* and *REQ* agents which manage its service requirements and provisions and the agents which define its behaviour. A primitive component is simply a component which has no Darwin defined substructure of components. The *Server* component type of figure 4 is represented by the π-calculus agent:

$$Server(p) \stackrel{\text{def}}{=} (\nu s)\ (PROV(p,s) \mid Server'(s))\ .$$

in which *Server'* represents the user implemented behaviour of the *Server* component which realises the services *s*. Similarly, the *Client* component type is represented by the agent:

$$Client(r) \stackrel{\text{def}}{=} (\nu o)\ (REQ(r, o) \mid Client'(o))\ .$$

The configuration of figure 4 can thus be represented in π-calculus by the parallel composition of a *Client*, *Server* and *BIND* agent:

$$(\nu a_r, b_p)\ (Client(a_r) \mid Server(b_p) \mid BIND(a_r, b_p))\ .$$

The instance names A and B are used only to qualify and thus rename the requirement *r* and provision *p* of the *Client* and *Server* component types. The expression above is a precise translation of the Darwin configuration of figure 4. However, to demonstrate that the model is correct, it must be shown that the client instance A will get the service reference provided by the server B when the configuration program is executed. Substituting the definitions for *Client* and *Server* and dropping the quoted user defined behaviour agents since they play no part in the binding process the π-calculus description of figure 4 becomes:

$$(\nu a_r, b_p)\ ((\nu o)\, REQ(a_r, o) \mid (\nu s)\, PROV(b_p, s) \mid BIND(a_r, b_p))\ .$$

Substituting the definitions of figure 3 for *REQ* and *BIND*:

$$\equiv (\nu a_r, b_p)\ ((\nu o)\ (a_r(y).\bar{y}o) \mid (\nu s)\, PROV(b_p, s) \mid \bar{a}_r b_p)\ .$$

Applying the reduction rule for the communication along a_r:

$$\rightarrow (\nu b_p)\ ((\nu o)\, \bar{b}_p o \mid (\nu s)\, PROV(b_p, s))\ .$$

Expanding *PROV* (using $!P \equiv P \mid !P$) to spin off one replica:

$$\equiv (\nu b_p)\ ((\nu o)\, \bar{b}_p o \mid (\nu s)\ (b_p(x).\bar{x}s \mid PROV(b_p, s)))\ .$$

Finally, applying the reduction rule for communication along b_p:

$$\rightarrow \underline{(\nu o, s, b_p)\ (\bar{o}s \mid PROV(b_p, s))}\ .$$

The expression describing the client server system thus reduces to an expression which sends the service *s* to the required location *o* in parallel with the *PROV* agent and of course *Server'* and *Client'*. Before the client can use the service it must perform an input action. A possible definition for *Client'* would be: $Client'(o) \stackrel{\text{def}}{=} o(x).Client''$. The system:

$$(\nu o, s, b_p)\ (\bar{o}s \mid PROV(b_p, s) \mid Client'(o) \mid Server'(s))$$

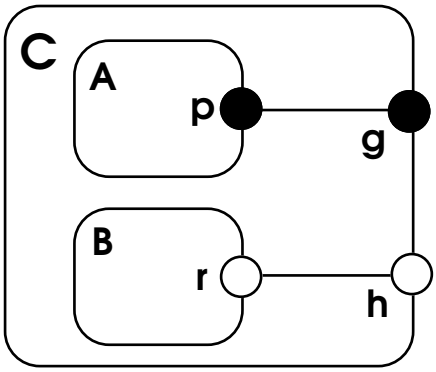
then reduces to:

$$\rightarrow (\nu s, b_p)\ (PROV(b_p, s) \mid Client"\{s/x\} \mid Server'(s))$$

which is the desired result of an instance of the server component executing in parallel with a client component in which every occurrence of the local name *x* has been replaced with the name *s*, the reference to the required service. In practice, a Darwin implementation can compute the number of requirements bound to a provision and so the number of replicas of the agent *PROV* is known. Consequently, the configuration process can terminate and the resources it uses can be recovered. It should be noted the model described permits binding and instantiation to proceed concurrently. Components which try to use a service will be blocked until they are bound to that service (i.e.. they must input the service reference as in *Client'*).

4.2 Composite components & Interface binding

So far, we have described how primitive components and the provision, requirement and binding of services are modelled in the π-calculus. As outlined in section 2, complex systems are specified in Darwin as a hierarchic structure of composite components. Composite components are defined by declaring both the instances of other components they contain and the bindings between these internal component instances. The interface to a composite component is defined in the same way as for a primitive component. It is a set of provided and required services. These *interface* requirements and provisions are bound to internal requirements and provisions as depicted in figure 5.

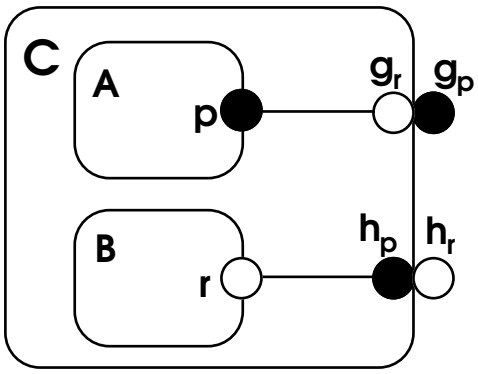



```
component C {
  provide g;
  require h;
  inst
     A:T1;
     B:T2;
  bind
      g -- A.p;
      B.r -- h;
}
```

Figure 5 - Composite Component

Interface requires and provides are modelled in the π-calculus as agents which both require and provide a service. An interface **provide** requires binding to a service internal to the composite component and provides that service externally. Similarly, an interface **require** provides a service internal to the component and requires to be bound to an externally provided service.

Figure 6 depicts these situations together with the π-calculus agent used to model interface requires and provides.



π-calculus for interface agent

$$VAR(m_p, m_r) \stackrel{\text{def}}{=} m_r(x).!(m_p(y).\bar{x}y)$$

where

m_r - access name for require side of *m*.

m_p - access name for provide side of *m*.

x - name bound to m_r.

y - location service required at.

Figure 6 - Interface agent

The interface agent *VAR* initially waits to receive the name *x* on channel m_r. It is then reduced to a process which repeatedly receives a name *y* on channel m_p and sends it to the channel *x* it initially received. We will define an agent *PASS* and observe in the following that binding to a provision reduces *VAR* agents to *PASS* agents. The *PASS* agent repeatedly receives a name *y* on channel *m* and sends that name to channel *n*.

$$PASS(m, n) \stackrel{\text{def}}{=} !(m(y).\bar{n}y)$$

Figure 7 examines the situations in which interface agents are bound to internally provided and required services.To simplify the equations, the agents representing internal component instances are not shown. As mentioned above, binding an interface **provide** to an internal **provide** results in the *VAR* agent being reduced to a *PASS* agent which forwards binding requests on to the *PROV* agent representing the internal **provide**. Binding an internal **require** to an interface require results in the *REQ* agent being reduced to the binding request $\bar{h}_p o$, where *o* is the location where the service is required. Binding many requirements to an interface **require** results in a set of binding requests in parallel with the *VAR* agent. When the *VAR* agent is bound to a provision and reduced to a *PASS* agent, these requests are forwarded to the provision as shown below:

$$\bar{m}o \mid PASS(m, n) \equiv \bar{m}o \mid m(y).\bar{n}y \mid PASS(m, n)$$
$$\rightarrow \bar{n}o \mid PASS(m, n)\ .$$

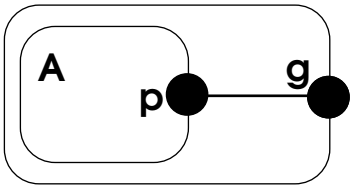
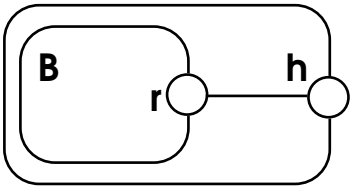
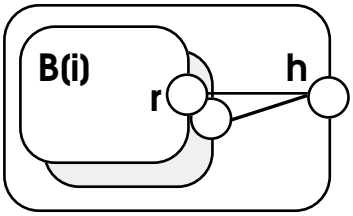
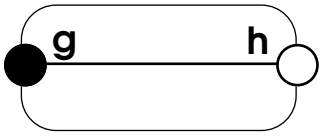
Darwin	π-calculus
 **bind** g -- A.p	Binding an interface **provide** to an internal **provide**. $VAR(g_p, g_r) \mid PROV(a_p, s) \mid BIND(g_r, a_p)$ $\equiv g_r(x).!(g_p(y).\bar{x}y) \mid PROV(a_p, s) \mid \bar{g}_r a_p$ $\rightarrow !(g_p(y).\bar{a}_p y) \mid PROV(a_p, s)$ $\equiv PASS(g_p, a_p) \mid PROV(a_p, s)$
 **bind** B.r -- h	Binding an internal **require** to an interface **require**. $REQ(b_r, o) \mid VAR(h_p, h_r) \mid BIND(b_r, h_p)$ $\equiv b_r(y).\bar{y}o \mid VAR(h_p, h_r) \mid \bar{b}_r h_p$ $\rightarrow \bar{h}_p o \mid VAR(h_p, h_r)$
 **forall** i:1..n **bind** B[i].r -- h	Binding many internal requirements to an interface **require** reduces to: $\bar{h}_p o_1 \mid \ldots \mid \bar{h}_p o_n \mid VAR(h_p, h_r)$

Figure 7 - Hierarchic binding

4.3 Intra-component interface binding

In addition to binding interface **requires** and **provides** to internal requirements and provisions, they may be bound to each other to form binding only or connection components as depicted in figure 8. These connection only components are useful for defining connection patterns such as perfect-shuffle and, in addition, they may form the base case of recursively defined structures. Although it may seem that a provision is being bound to a requirement in contravention of the rule that binding is always from requirement to provision, in fact we are binding the internal names of interface entities, which have the opposite sense to their external names. These intra-component bindings reduce to *PASS* agents which forward binding requests through the component (figure 8).

Darwin	π-calculus


component D {
provide g;
require h;
bind g -- h;
}
	Binding an interface **provide** to an interface **require**.
	$VAR(g_p, g_r) \mid VAR(h_p, h_r) \mid BIND(g_r, h_p)$
	$\rightarrow !(g_p(y).\bar{h}_p y) \mid VAR(h_p, h_r)$
	$\equiv PASS(g_p, h_p) \mid VAR(h_p, h_r)$

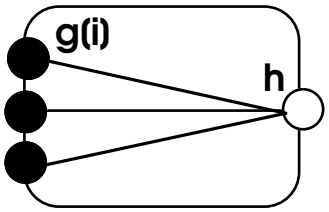
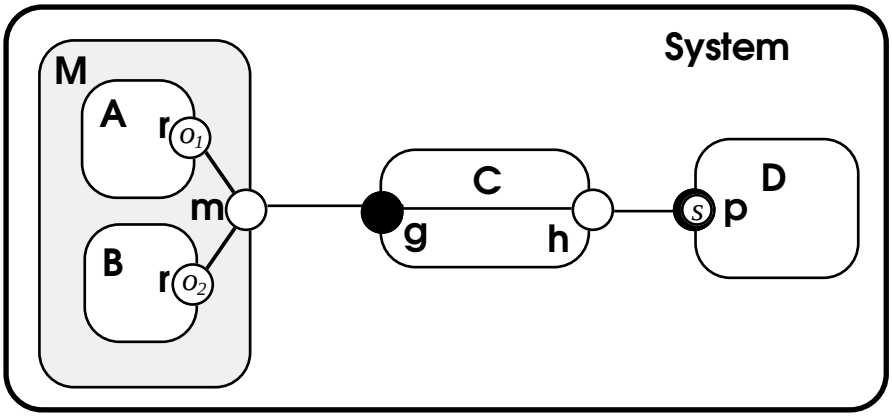
 **forall** i:1..n **bind** g[i] -- h;	Binding many interface **provides** to an interface **require** reduces to**:** $PASS(g_{p_1}, h_p) \mid \ldots \mid PASS(g_{p_n}, h_p) \mid VAR(h_p, h_r)$

Figure 8 - Intra-component bindings

4.4 Program elaboration example

In the previous subsections, π-calculus agents have been defined for each of the Darwin syntactic constructs for declaring components, services and bindings. In addition, the results of combining these agents in the ways permitted by Darwin's grammar has been determined. We can now ascertain the effect of elaborating complex configuration specifications and check that the correct result is obtained. In particular, it is necessary to demonstrate that complex configurations reduce to a system of primitive component instances in which service references have been correctly placed where they are required. The correctness of the elaboration process must be independent of the order of component instantiation or binding actions since elaboration of Darwin programs typically takes place in a distributed setting.

In the following, we demonstrate the elaboration process for Darwin programs by an example. The next section, establishes a correctness condition for elaboration and shows informally that it is satisfied in the π-calculus model for Darwin. The Darwin specification for the example is given in Figure 9. The system consists of a composite component connected through a connection only component to a server component.




```
component Mclient {
  require m;
  inst
    A:Client;
    B:Client;
  bind
    m -- A.r;
    m -- B.r;
}
```

```
component Conn {
  provide g
  require h;
  bind
    g -- h;
}
```

```
component System {
  inst
    M:Mclient;
    C:Conn;
    D:Server;
  bind
     M.m -- C.g;
     C.h -- D.p;
}
```

Figure 9 - Elaboration example

To show that elaboration occurs correctly, we must show that the service name *s* provided by server *D* is sent to the two locations, o_1 & o_2 , where it is required. Firstly, using the results for hierarchic binding of requirements from figure 7, we can show that the *Mclient* component will reduce to the following π-calculus expression which consists of two binding requests and the agent representing the interface requirement *m* (omitting primitive component agents as usual):

$$Mclient(m_r) \stackrel{\text{def}}{=} (\nu o_1, o_2, m_p)\ (\overline{m}_p o_1 \mid \overline{m}_p o_2 \mid VAR(m_p, m_r))\ .$$

Using the results for intra-component binding from figure 8, the component *Conn* is expressed below:

$$Conn(g_p, h_r) \stackrel{\text{def}}{=} (\nu h_p)\ (PASS(g_p, h_p) \mid VAR(h_p, h_r))\ .$$

The primitive component *Server* is as defined in section 4.1 with the omission of *Server'* which represents its behaviour:

$$Server(p) \stackrel{\text{def}}{=} (\nu s)\, PROV(p, s)\ .$$

The *System* component is simply a composition of the above components and the agents representing the bindings between these components.

$$System \stackrel{\text{def}}{=} (\nu m_r, g_p, h_r, p)\ (Mclient(m_r) \mid Conn(g_p, h_r) \mid Server(p) \mid \overline{m}_r g_p \mid \overline{h}_r p)$$

Initially, we will reduce the composition of *Mclient* and *Conn* and then compose the result of this reduction with *Server.* The subexpression is:

$$(\nu m_r, g_p, h_r)\ (Mclient(m_r) \mid Conn(g_p, h_r) \mid \overline{m}_r g_p)\ .$$

Substituting for *Mclient*:

$$\equiv (\nu m_r, g_p, h_r)\ ((\nu o_1, o_2, m_p)\ (\overline{m}_p o_1 \mid \overline{m}_p o_2 \mid VAR(m_p, m_r)) \mid Conn(g_p, h_r) \mid \overline{m}_r g_p)\ .$$

The composition of the binding $\overline{m}_r g_p$ with $VAR(m_p, m_r)$ reduces to $PASS(m_p, g_p)$ (see figure 8). The composition of the requests $\overline{m}_p o_1$ and $\overline{m}_p o_2$ with this *PASS* agent transforms these requests to $\overline{g}_p o_1$ and $\overline{g}_p o_2$ respectively (as shown in section 4.2). Dropping the *PASS* agent, since it plays no further part in the elaboration process, and substituting the definition of *Conn* the above expression reduces to:

$$\rightarrow (\nu g_p, h_r, o_1, o_2)\ (\overline{g}_p o_1 \mid \overline{g}_p o_2 \mid (\nu h_p)\ (PASS(g_p, h_p) \mid VAR(h_p, h_r)))\ .$$

The requests $\overline{g}_p o_1$ and $\overline{g}_p o_2$ are transformed again by *PASS* to give (again dropping the *PASS*):

$$\rightarrow (\nu h_p, h_r, o_1, o_2)\ (\overline{h}_p o_1 \mid \overline{h}_p o_2 \mid VAR(h_p, h_r))\ .$$

Composing this with the *Server* agent and substituting its definition:

$$(\nu h_p, h_r, o_1, o_2, p)\ (\overline{h}_p o_1 \mid \overline{h}_p o_2 \mid VAR(h_p, h_r) \mid (\nu s)\, PROV(p, s) \mid \overline{h}_r p)\ .$$

The composition of $\overline{h}_r p$ with $VAR(h_p, h_r)$ reduces to $PASS(h_p, p)$. This *PASS* agent reduces the binding requests $\overline{h}_p o_1$ and $\overline{h}_p o_2$ to $\overline{p} o_1$ and $\overline{p} o_2$ respectively. Discarding the *PASS* agent gives:

$$(\nu o_1, o_2, p)\ (\overline{p} o_1 \mid \overline{p} o_2 \mid (\nu s)\, PROV(p, s))$$

As described previously in section 4.1, spinning off copies of the replicated agent *PROV,* reduces the expression to:

$$\underline{(\nu o_1, o_2, p, s)\ (\overline{o}_1 s \mid \overline{o}_2 s \mid PROV(p, s))\ .}$$

This is of course a system which sends the service name *s* to the two locations o_1 & o_2, where it is required.

4.5 Correctness of Program Elaboration

From the example of the previous section, it should be apparent that a Darwin configuration program specifies a set of tree structured directed acyclic graphs in which the leaf vertices are requirements and the root vertices are provisions. Vertices at intermediate levels of a tree are interface provisions and requirements. The arcs represent the bindings between requirements and provisions. The example of figure 9 can be represented by a single tree since there is a single provided service. Figure 10 depicts the general case for a configuration which provides a single service *s*. Configurations which provide multiple services simply consist of multiple trees of the form shown in figure 10.

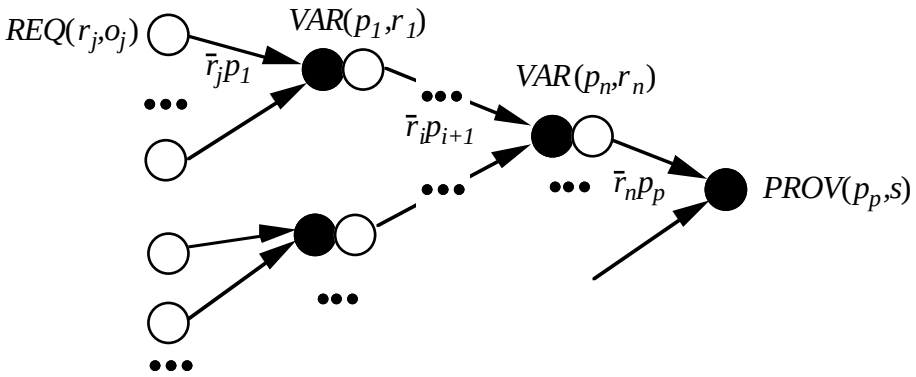


Figure 10 - General Configuration Graph

The correctness condition for elaborating a Darwin configuration is thus:

> If there is a path in the configuration graph from the requirement $REQ(r_j, o_j)$ to a provision $PROV(p_p, s)$ then elaboration should result in the binding $\bar{o}_j s$.

To prove this, we must demonstrate that the following system produces $\bar{o}_j s$:

$$REQ(r_j, o_j) \mid \bar{r}_j p_1 \mid \ldots \mid VAR(p_i, r_i) \mid \bar{r}_i p_{i+1} \mid \ldots \text{ where } i\text{: } 1..\ n\text{-}1$$
$$\ldots \mid VAR(p_n, r_n) \mid \bar{r}_n p_p \mid PROV(p_p, s)\ .$$

From figure 7: $REQ(r_j, o_j) \mid \bar{r}_j p_1 \rightarrow \bar{p}_1 o_j$

and from figure 8: $VAR(p_i, r_i) \mid \bar{r}_i p_{i+1} \rightarrow PASS(p_i, p_{i+1})$

From section 4.2: $\bar{p}_1 o_j \mid PASS(p_i, p_{i+1}) \rightarrow \bar{p}_n o_j \mid PASS(p_i, p_{i+1})$ where i: *1.. n-1.*

Similarly: $VAR(p_n, r_n) \mid \bar{r}_n p_p \rightarrow PASS(p_n, p_p)$

and $\bar{p}_n o_j \mid PASS(p_n, p_p) \rightarrow \bar{p}_p o_j \mid PASS(p_n, p_p)$.

Dropping the *PASS* processes, the system becomes:

$$\underline{\bar{p}_p o_j \mid PROV(p_p, s) \rightarrow \bar{o}_j s \mid PROV(p_p, s)}\ .$$

The above π-calculus model of Darwin elaboration is an abstract specification of the distributed elaboration algorithm implemented in the Regis system. In Regis, asynchronous message passing is used to send the locations at which service references are required to the providers of services. These messages are forwarded by processes representing interfaces. In the Regis implementation of Darwin elaboration, the *PASS* and *PROV* processes are implemented by a single elaboration manager process per component. When component parameters are substituted and conditional configuration guards evaluated, the number of bindings managed by these processes can be computed and the elaboration computation can thus be terminated. In the π-calculus model, we have chosen to ignore the detail of *PASS* and *VAR* process termination.

In addition to transmitting locations towards the provider of a service (left to right in figure 10), the Regis implementation transmits service references back towards requirements since in many cases this can reduce the amount of remote communications.This behaviour is captured by the π-calculus model since the system $PASS(q, p) \mid PROV(p, s)$ has equivalent communication behaviour to the system $PROV(q, s) \mid PROV(p, s)$. When composed with either $\bar{q}o$ or $\bar{p}o$ both systems reduce these binding requests to $\bar{o}s$. In the above proof, we could thus substitute *PROV* processes for *PASS* processes which models the backward transmission of the service references.

The Darwin compiler cannot statically detect two categories of incorrect bindings. These incorrect bindings can thus occur during elaboration. It is instructive to compare the behaviour we can determine from the π-calculus model with the behaviour we observe in the Regis implementation for these situations. The first category is simply the situation where a requirement is not bound. As noted in section 4.1, this simply causes the component containing that requirement to be blocked. The more interesting binding error is depicted in Figure 11 in which a requirement is bound to a cycle of interface entities.

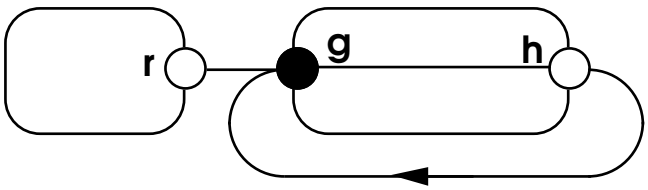


Figure 11 - Cyclic binding error

While the simple case of cyclic binding depicted in figure 11 can be statically detected, the general case cannot be statically detected when separate compilation, parameterisation and conditional configuration are taken into account. The π-calculus model of the system of figure 11 is:

$$\begin{aligned} REQ(r, o) \mid \bar{r}g_p \mid VAR(g_p, g_r) \mid \bar{g}_r h_p \mid VAR(h_p, h_r) \mid \bar{h}_r g_p \\ \rightarrow \bar{g}_p o \mid PASS(g_p, h_p) \mid PASS(h_p, g_p) \\ \rightarrow \bar{h}_p o \mid PASS(g_p, h_p) \mid PASS(h_p, g_p) \\ \rightarrow \bar{g}_p o \mid PASS(g_p, h_p) \mid PASS(h_p, g_p) \end{aligned}$$

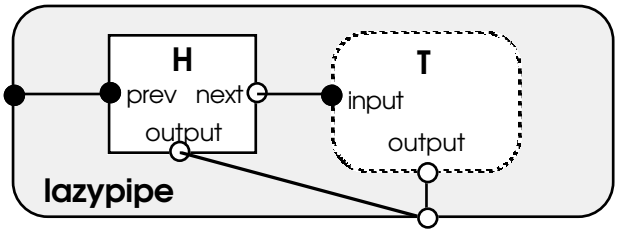
As shown above, the system reduces to a system which continuously circulates the binding request. The behaviour observed in early versions the Regis system was that elaboration manager processes continuously circulate binding messages. The current version detects the error and raises a runtime exception.

5. Extending the basic model

In the previous section, we have described how the basic features of Darwin, concerned with binding, instantiation and hierarchy, are modelled in the π-calculus. In the following, we demonstrate how the features of Darwin concerned with describing dynamic structures and open systems binding are modelled in the framework of the basic model. The section concludes by briefly outlining the way component interaction mechanisms fit into the framework of the model.

5.1 Lazy instantiation

Darwin provides two mechanisms for describing dynamic structures which can evolve at run-time as opposed to static structures which are defined at instantiation/elaboration time. The first of these is lazy instantiation in which the component providing a service is not instantiated until a user of that service attempts to access the service. The combination of lazy instantiation with recursion allows the description of potentially unbounded structures as shown in the example of figure 12.




```
component lazypipe {
  provide input;
  require output;
  inst
    H:filter;
    T:dyn lazypipe;
  bind
     input -- H.prev;
     H.next -- T.input;
     H.output -- output;
     T.output -- output;
}
```

Figure 12 - Lazy instantiation example

The program of figure 12 generates a pipeline of *filter* components. Initially a single instance *H* of the *filter* component is instantiated. The *next* requirement of this instance *H* is bound to the *input* provision of the instance *T* (*H.next* -- *T.input*) which is again a pipeline. T is not immediately instantiated since it is declared lazy by the keyword **dyn**. When the *H* instance attempts to access the service provided through *T.input* the instantiation of *T* is triggered along with the bind actions for the requirements of *T* (i.e.. *T.output* -- *output*). The pipeline of *filter* instances is thus extended as required.

Lazy instantiation is modelled by using a dummy provision to which the clients of a server are initially bound. This dummy provision, in response to a binding request, returns the name of a prefix which triggers instantiation of the lazy instance and its associated bindings. For example, consider the system of figure 4 with lazy instantiation of the server component e.g.

```
inst
    A:Client;
    B: dyn Server;
bind A.r -- B.p;
```

The π-calculus model for this system becomes:

$$(\nu a_r, b_p, d)\ (Client(a_r) \mid PROV(d, b_p) \mid b_p(y).\ (\bar{b}_p y \mid Server(b_p))\ \mid \bar{a}_r d)$$

in which the *Client* is bound to the dummy provision *d*. This reduces to:

$$\rightarrow\ (\nu b_p, o)\ (\bar{o} b_p \mid Client'(o) \mid b_p(y).\ (\bar{b}_p y \mid Server(b_p))\ )$$

where *o* is the location at which the client requires the service as in section 4.1. Note that instantiation of the *Server* component is guarded by the prefix $b_p(y)$ and that the binding returned to the client is the name of this prefix. In a more complex system, the prefix would also guard the binding actions for the requirements of the lazy instantiated component. In addition, the component may have more than one provision which triggers instantiation, in which case the prefix becomes a sum of alternative actions. To deal with lazy instantiation, the *Client* must perform a more complex action than simply inputting the binding by the action $o(x)$ as described in section 4.1. The *Client* must decide if the binding is to a lazy service and if so perform what is in effect a rebind. Thus *Client'* becomes:

$$Client'(o) \stackrel{\text{def}}{=} \quad o(x).LAZY(x, t, f) \mid\ (t.\bar{x}o.o(x).Client'' \quad f.Client'')\ .$$

The agent *LAZY* outputs the signal true (*t*) or false(*t*) depending on whether or not *x* names a lazily instantiated service. Definition of this agent is not trivial since names in π-calculus are primitive entities which have no structure. Consequently, *LAZY* must maintain the finite set of all names which refer to services and the finite set of all names which refer to lazy services to make a decision. Additionally, we need to extend the π-calclus defined in section 3 with the match operator originally defined in [9]. A more satisfactory solution, and one which closely models the Regis implementation, is to give service names a structure and encode the lazy property in this structure. This can easily be achieved in the polyadic π-calculus[11] but it is beyond the scope of this paper.

It should be noted that the introduction of lazy instantiation has not changed the elaboration model described in the previous section since lazy services are simply represented by a *PROV* agent which does not in any way affect elaboration and binding.

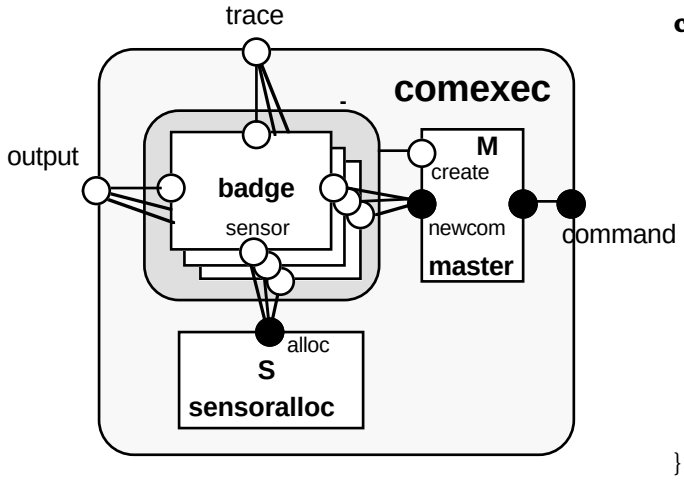
5.2 Direct dynamic instantiation

Lazy instantiation permits a structure to evolve according to a fixed pattern. Direct dynamic instantiation permits the definition of structures which can evolve in an arbitrary way. Of course, much less of the runtime structure is captured in the Darwin program. In practice, we have found that dynamic instantiation can be used in a way which balances flexibility at run-time with the advantages of retaining a structural description. Figure 13 is an example from the Active Badge system of the implementation of the component depicted in figure 1. 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. 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 master's requirement is satisfied in the configuration program of figure 13 by binding it to the component type *badge* prefixed by the keyword **dyn**. i.e. *M.create* **-- *dyn*** *badge*. Note that in figure 13, bindings are specified for the component type *badge* rather than for instances of this type as is usual. 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 provide. Dynamically created components may provide services, however, access to these services is achieved by passing service references in messages to form bindings dynamically. These bindings cannot be captured by the Darwin program.




```
component comexec {
  require  trace    <event bstatus>;
           output   <port  smsg>;
  provide command <port comT>;

  inst
     M:master;
     S :sensoralloc;
  bind
      M.create -- dyn badge;
      badge.trace -- trace;
      badge.sensor  -- S.alloc;
      badge.output  -- output;
      badge.command -- M.newcom;
      command -- M.command;
}
```

Figure 13 - Direct dynamic instantiation example

Dynamic instantiation is modelled in the π-calculus by a *PROV* agent which supplies the name of the instantiation service. This instantiation service triggers one of the copies of a replicated process. As an example of modelling dynamic instantiation, we will again use the client-server system of figure 4 and modify it so that *Client* components can be created through the service *d*:

```
provide d <dyn>;
inst B:Server;
bind
    d -- client;
    client.r -- B.p;
```

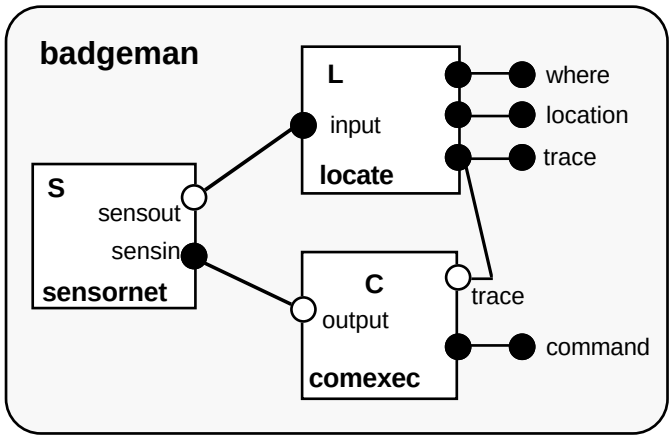
The π-calculus model for this system is shown below. The *PROV* agent will return the name *m* in response to a binding request. A client which performs the action $\bar{m}$ will cause a new replica of the Client component to be instantiated together with its associated bind action. In general, the action would be $\bar{m}\vec{x}$ where $\vec{x}$ represents the vector of parameters for the newly instantiated component.

$$(\nu b_p, d, m)\ (Server(b_p) \mid PROV(d, m) \mid !(m.(\nu r)\ (Client(r) \mid \bar{r} b_p)))$$

Dynamic instantiation does not change the basic model of section 4. It is represented by a *PROV* agent which is treated and bound in the same way as other *PROV* agents. Note that the way in which a component is instantiated, statically or dynamically, does not change the definition of that component.

5.3 Open systems binding

It is intended that systems implemented using Darwin interwork with external systems in an open systems environment. Darwin, as described so far, has a closed namespace which would not permit services implemented inside a system to be externally accessed. Consequently, Darwin has the facility both to export service names into an external namespace with an associated external name and to import names from an external namespace.




```
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.sensin;
     C.trace -- L.trace;
}
```

Figure 14 - Exporting services

The example program of figure 14 combines the *sensornet* and *comexec* components described earlier with a *locate* component to construct the badge system management server. This server exports a set of services with associated external names (in quotes). Exports are bound to a service provided internally (e.g. *where -- L.where*). Darwin components may also import external services. For example, the statement **import** *w@"badge/where"*, would return a reference to the *where* service exported by an instance of *badgeman*. Internal requirements may be bound to **import**s.

Modelling the Darwin **export** construct in π-calculus is reasonably straight forward. The Darwin statement **export** *e @ n* is modelled as follows:

$$EXPORT(e, n) \stackrel{\text{def}}{=} \quad (\nu o)\ (REQ(e, o) \mid o(x).REG(n, x))\ .$$

The *EXPORT* agent consists of a *REQ* agent in parallel with *REG*. The *REG* agent encapsulates the details of registering the service in a nameserver with the external name *n*. When a provided service is bound to the export, the resulting binding $\bar{o}s$ will reduce the *EXPORT* agent to *REG(n,s)*. Exports behave in the same way as requirements during binding and elaboration.

The π-calculus model for **import** *i @ n* is complicated by the fact that we do not wish elaboration of all or part of a Darwin program to be suspended while an external service reference is fetched from a nameserver. Consequently, the definition of *IMPORT* uses the lazy instantiation mechanism described in section 5.1. Requirements are initially bound to a dummy provision *i*.

$$IMPORT(i, n) \stackrel{\text{def}}{=} \quad (\nu d)\ (PROV(i, d) \mid d(y).\ (\bar{d}y \mid\ (\nu z)\ LUP(n, z) \mid z(s).PROV(d, s))\ )$$

An attempt to use the service represented by the *IMPORT* agent triggers the *LUP* agent which looks up the reference to an external service with the name *n*. When it gets the service name, *LUP* performs the action $\bar{z}s$ which enables the agent *PROV* to return the service name *s* in response to binding requests. Imports behave in the same way as provisions during binding and elaboration.

5.4 Component interaction

Components interact or communicate using the bindings to services computed by the elaboration of the Darwin program. The binding patterns established by Darwin are many-to-one in that one or more requirements for a service may be bound to the provider of that service. However, for some component interaction mechanisms, further binding may be required. This is not part of Darwin and may be treated orthogonally to the elaboration model developed in the foregoing. However, the π- calculus may be used to model this additional binding process.

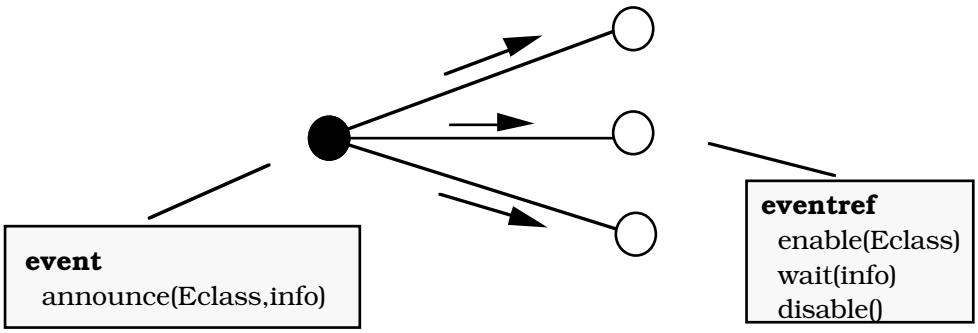


Figure 15 - Event interaction mechanism

An example of extended binding is found in the the Regis event distribution mechanism which allows the provider of an event service to transmit information to many receivers as shown in figure 15. This requires a one-to-many binding which is constructed by the event mechanism using the Darwin many-to-one binding. Darwin gives each client of the event service the name of that service *s*. To enable the receipt of event information messages, the client sends the name of a private channel *y* to the event service ($\bar{s}y$) together with the event class the client is interested in. The event service maintains a set of the private channel names of enabled clients. It then uses this set to send information message (with the selected event class *Eclass*) to clients. Component interaction mechanisms may thus easily extend the binding supported by Darwin.

6. Discussion & Conclusions

In the introduction, we identified two objectives for the π-calculus model of Darwin. They were firstly, that it should clearly specify the division of responsibilities between Darwin and the primitive components it configures and secondly, that it should provide a clear specification of Darwin's operational behaviour.

The basic model developed in section 4 has little impact on the internal structure and behaviour of primitive components. They may be concurrent and distributed. They are only required to supply the names or references of services and accept bindings. This illustrates clearly the separation between configuration and computation/communication in systems constructed using Darwin. Components requiring services needed to be extended to accommodate lazy instantiation. In this situation, components requiring services must perform a rebind action. However, the model clearly specifies the binding and instantiation actions taken by Darwin. Section 5.4 demonstrated that the basic many-to-one binding patterns established by Darwin can easily be extended by additional binding actions taken by component interaction mechanisms.The model does not make any assumptions about the way instantiated primitive components interact. We have deliberately not considered in any detail the modelling of component interaction mechanisms. However, some of the communication mechanisms supported by the Regis system have been modelled in detail in the π-calculus[12]. These interaction models do not impact the configuration of Darwin programs but rather their runtime behaviour. We can thus modularise our reasoning about Darwin/Regis programs or indeed any distributed system using Darwin for configuration support.

Section 4 described a general model of the elaboration of Darwin programs. It demonstrated that for correct configurations this elaboration resulted in the correct bindings between primitive components requiring services and those providing them. In addition, the model could be used to examine the behaviour of incorrect configurations. The fact that this behaviour agrees with that observed in an implementation gives some additional confidence in the validity of the model. Section 5 showed that the Darwin features concerned with dynamic configuration and open systems binding could be easily modelled without disturbing the basic elaboration algorithm. This has given us confidence in the design of these features. Further extensions can be tested against the criteria that they do not adversely affect or complicate elaboration. Work is currently in progress to provide the ability to manage group communication abstractions in the Darwin setting and to cater for component migration.

We have chosen to ignore two major aspects of the Darwin configuration language in arriving at the π-calculus model. These are firstly, component parameterisation and secondly, the concept of service type supported by Darwin. Component parameters can determine the final

structure of a system through the conditional and replicator constructs. While these could be modelled directly in π-calculus the resulting model is clumsy and obscures the intuitions that can be obtained from the current model. We have found it more convenient to consider parameter substitution and the resulting conditional guard and replicator evaluation as a phase (similar to macro expansion) which occurs before concurrent elaboration. Section 2 briefly described the way service types can be associated with **provide** and **require**. Darwin supports static checking to ensure only bindings between compatible requirements and provisions are allowed. Service types can be modelled using the concepts of *sort* and *sorting* provided by the polyadic π-calculus, but unavailable in the simple monadic form of the π-calculus used in this paper. However, the main reason for ignoring these two aspects of Darwin is that they do not directly impact the concurrent behaviour of Darwin programs, which is the primary focus of the paper.

One of the major benefits of this work in using the π-calculus to model Darwin has been our increased understanding of the role and nature of configuration languages. Initially, we attempted to define the semantics of the CONIC system using the CCS[13] and CSP[14] formalisms. While its was possible to reason about the behaviour of the set of communicating processes resulting from the elaboration of a configuration program, we were unable to develop a satisfactory model for the elaboration process itself. It now seems clear that this was due to the inability in these formalisms to describe evolving or dynamic structures. However, at the time, CONIC supported only the definition of static structures and it did not occur to us to think of elaboration as a computation requiring the mobility of processes or channels. In fact, the CONIC system did not treat channels as first class objects which could be transmitted in messages and the elaboration process was sequential. The requirement that Darwin be a general purpose configuration language led us to develop a more general model for binding which involved the management of service references. The requirement that the elaboration process be distributed meant that these service references must be freely transmitted between processes in messages. Milner[15] stresses the fundamental importance of naming or reference in concurrent computation and considers the π-calculus as the beginnings of a tractable theory for reference. It is consequently not surprising that Darwin, a language primarily concerned with reference and binding, can be elegantly modelled in the π-calculus, demonstrating in this instance, an encouraging convergence between the development of the theory of concurrent computation and the practice of constructing concurrent and distributed systems.

Acknowledgements

The authors would like to acknowledge discussions with our colleagues in the Distributed Software Engineering Section Group during the formulation of these ideas and in particularly Naranker Dulay, who with two of the authors is responsible for the design and implementation of the Darwin configuration language. 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. Magee, J. Kramer and M. Sloman, *Constructing Distributed Systems in Conic*, IEEE Transactions on Software Engineering, SE-15 (6), 1989.

[2] 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.

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

[4] J.Magee, N. Dulay, J. Kramer, *Regis: A Constructive Development Environment for Distributed Programs*, Dristubuted Systems Engineering Journal, to appear.

[5] ANSAware 4.1: *Application Programming in ANSAware*, *Document* RM.102.02, Architecture Projects Management Agency, Poseidon House, Castle Park, Cambridge CM3 0RD, UK, Feb. 1993.

[6] Inmos Ltd, *OCCAM 2 reference manual*, Prentice Hall, 1988.

[7] 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.

[8] 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.

[9] R.Milner, J. Parrow, and D.Walker, *A calculus of mobile processes, Parts I and II*, Journal of Information and Computation, Vol. 100, pp1-40 and pp41-77, 1992.

[10] Harter A., Hopper A., *A Distributed Location System for the Active Office*, IEEE Network, Jan./ Feb. 1994, pp. 62-70.

[11] R. Milner, *The polyadic π-calculus: a tutorial*, in Logic and Algebra of Specification, ed. F.L. Bauer, W. Brauer and H. Schwichttenberg, Springer Verlag, 1993, pp203-246.

[12] M. Radestock and S. Eisenbach, *What Do You Get From a π-calculus Semantics?*, PARLE 94, Springer-Verlag, Lecture Notes in Computer Science No. 817, pp635-647, 1994.

[13] R. Milner, *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, N.J., 1989.

[14] C.A.R. Hoare, *Communicating sequential processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.

[15] R. Milner, *Elements of Interaction - Turing Award Lecture*, CACM, Vol 36, No. 1, Jamuary 1993, pp78-79.