Specifying Distributed Software Architectures

Jeff Magee, Naranker Dulay, Susan Eisenbach and Jeff Kramer

Department of Computing
Imperial College
London SW7 2BZ, United Kingdom
jnm, nd, se, jk@doc.ic.ac.uk

Abstract. There is a real need for clear and sound design specifications of distributed systems at the architectural level. This is the level of the design which deals with the high-level organisation of computational elements and the interactions between those elements. The paper presents the Darwin notation for specifying this high-level organisation. Darwin 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 supports the specification of both static structures and dynamic structures which may evolve during execution. The central abstractions managed by Darwin are components and services. Services are the means by which components interact.

In addition to its use in specifying the architecture of a distributed system, Darwin has an operational semantics for the elaboration of specifications such that they may be used at runtime to direct the construction of the desired system. 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 services in Darwin leads to an elegant and concise π-calculus model of Darwin's operational semantics. The model is used to argue the correctness of the Darwin elaboration process. The overall objective is to provide a soundly based notation for specifying and constructing distributed software architectures.

This paper will appear in the Fifth European Software Engineering Conference, ESEC '95 on 26 September 1995 in Barcelona.

1 Introduction

It has been recently recognised within the Software Engineering community, most notably by Garlan and Shaw [1] and Perry and Wolf [2], that when systems are constructed from many components, the organisation or architecture of the overall system presents a new set of design problems. One of the architectural concerns identified by Garlan and Perry [3] is the high-level description of systems based on graphs of interacting components. They identify *components* as the primary points of computation in a system and *connectors* to define the interactions between these components. Our work addresses this concern in the context of distributed systems and in particular stresses the management of system structure.

We are concerned with the provision of sound and practical means for the design and construction of distributed systems. To this end we have been involved in the development and use of structural configuration languages [4, 5, 6] as a means of specifying 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 architecture 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 specification as opposed to a structural description. Others that have also adopted a similar use of configuration languages for distributed systems include Polylith [7], Durra [8] and LEAP [9].

The version of Darwin used in this paper is the latest in a line of configuration languages. Darwin is a declarative language which is intended to be a general purpose notation for specifying the structure of systems composed from diverse components using diverse interaction mechanisms. It is currently being used in the context of the Regis system [10] which supports multiple interaction primitives and in the Sysman project [11] with ANSAware [12] which uses remote object invocation for component interaction. Darwin allows the specification of both static structures fixed during system initialisation and dynamic structures which evolve as execution progresses. An earlier version of Darwin was used in conjunction with the REX distributed systems platform [5]. The version described here differs in its treatment of dynamic structures and in the ability to deal with diverse interaction mechanisms.

Distributed programs can be constructed directly from their Darwin specifications. Darwin thus has an operational interpretation such that elaboration at runtime of the Darwin specification results in a distributed set of interconnected primitive components. In contrast with its predecessors, CONIC [4] and REX [5] which had centralised sequential interpretations, Darwin has a distributed and concurrent interpretation permitting the construction of large distributed systems in an efficient manner. In addition, it allows physical distribution to be specified completely orthogonally to logical structure. Darwin allows interaction with external management agents [11] which can direct structural changes in response to changing system requirements whether operational or evolutionary.

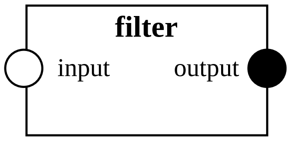
The aim that Darwin be general purpose requires that there should be a clear and well specified division of responsibilities between Darwin and the primitive components it configures. The requirement that Darwin should be capable of concurrent elaboration demands that the re must be a clear and unambiguous model 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 [13], Robin Milner's calculus of mobile processes. The reasons for choosing this formalism are discussed in the concluding sections of the paper. The paper initially describes the basic features of Darwin and outlines how these are modelled in the π-calculus. This basic model is then used to prove some properties of Darwin configurations. The paper demonstrates that the basic π-calculus model can be extended to incorporate those parts of Darwin concerned with dynamic structures and concludes by comparing the Darwin/π-calculus approach we have adopted with related work.

2 Darwin

A distributed system consists of multiple concurrently executing and interacting computational components. Typically, a system consists of a limited set of component types with multiple instances of these types. The task of specifying the system 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. It has both a graphical and textual representation. Darwin allows distributed programs to be constructed from hierarchically structured specifications 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. Components interact by accessing services. This section gives a brief overview of Darwin before a more precise description using π-calculus is given in the following section.

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. For example, the component of figure 1 is a *filter* component which **provides** a single service *output* and **requires** a single service *input*. The diagrammatic convention used here is that filled in circles represent services provided by a component and empty circles represent services required by a component. The type of the service is specified in angle brackets. In the example, the communication mechanism used to implement the service is a *stream* and the datatype communicated is *char*. Darwin does not interpret service type information, it is used by the underlying distributed platform. In the Regis system [10], this information is used to directly select the correct communication code. When used with a more conventional distributed systems platform such as ANSAware, the service type names an IDL specification which is used to generate the correct client and server stubs.



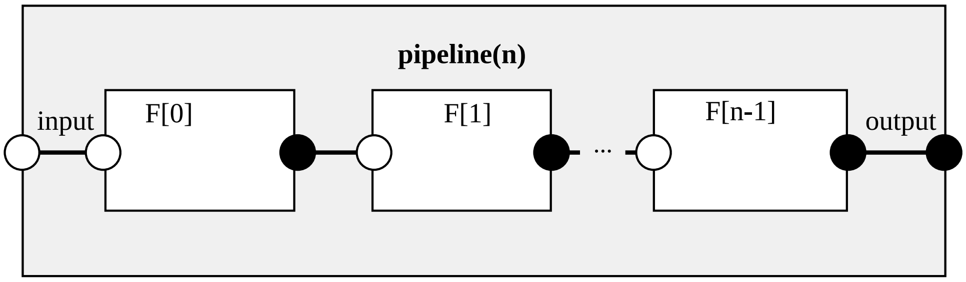

```
component filter{
  provide output<stream char>;
  require input<stream char>;
}
```

Fig. 1. component type *filter*

In general, a component may provide many services and require many services. It should be noted that the names of required and provided services are local to the component type specification. A component does not need to know the global names of external services or where they are to be found in the distributed environment. Components may thus be implemented and tested independently of the rest of the system of which they will form a part. We call this property *context independence*. It permits the reuse of components during construction (through multiple instantiation) and simplifies replacement during maintenance.

2.2 Composite Components

The primary purpose of the Darwin configuration language is to allow system architects to construct composite components from both basic computational components and from other composite components. The resulting system 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 example of figure 2 defines a variable length pipeline of filter instances in which the input of each instance is bound to its predecessor's output. Bindings between requirements and provisions are declared by the bind statement. For example, the *input* of each *filter* component instance $F[k+1]$ is bound to the *output* of its predecessor $F[k]$ by the statement **bind** $F[k+1].input$ -- $F[k].output$. 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 filter $F[0]$ requirement *input* which is bound to *input*. Similarly services provided internally which are required outside are bound to an interface service provision e.g. *output* -- $F[n-1].output$. The Darwin design [14] and construction tools check that




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

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

Fig. 2. composite component *pipeline*

bindings are only made between required and provided services with compatible types. The compatibility test invoked is determined by the target distributed systems platform. Where necessary, the Darwin tools infer the type of interface services which are not explicitly typed. In general, many requirements may be bound to a single provision. A particular requirement may be bound to a single provision only. It should be noted that a service may transmit or receive information or do both. The many requirements to a single provision binding pattern may thus describe either one-to-many or many-to-one communication depending on the interaction mechanism used to implement the service. For example, streams and events in Regis are one-to-many interaction types while ports and entries (similar to Ada entries) are many-to-one.

The example of figure 2 locates each filter instance $F[k]$ on a different host computer by means of the annotation $@k+1$. The integer machine identifiers are mapped to real machine addresses by the runtime system for Darwin. This level of indirection in mapping permits portable specifications. In general, instances are located at the machine on which the enclosing component is elaborated unless they are annotated. The reader is referred to [10] for further (and more realistic) examples of Darwin configuration programs.

The π-calculus [13] is an elementary calculus for describing and analysing concurrent systems with evolving communication structure. In this paper, we use the simple monadic form. 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 primitive entities 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 ::= $\overline{x}z.P$	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 ::= $A_1 + \ldots + A_n$	Alternative action $n \geq 0$, execute one of A. When $n = 0$, it is written as $\mathbf{0}$ 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 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:

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

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:

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

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

Fig. 3. π-calculus

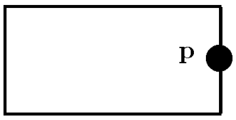
3 Modelling Darwin in the π-calculus

Our purpose in modelling Darwin in the π-calculus is to provide a precise semantics for the language. We wish to demonstrate that a Darwin configuration program correctly specifies the set of primitive component instances and set of intercomponent bindings required at runtime. Further, we wish to demonstrate that this elaboration process is correct when executed concurrently. 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. 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 [15], but unavailable in the simple monadic form of the π-calculus used in this paper. For simplicity, in the following, the types of Darwin services and type discipline for binding are omitted. A brief overview for readers unfamiliar with the π-calculus is given in figure 3.

3.1 Provide, Require, Bind

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 elaboration of a simple configuration which has no hierarchic structure.

Provide The declaration of a provided service, **provide** p, in Darwin is modelled in the π-calculus as the agent $\text{P}\small{\text{ROV}}(p, s)$ which is accessed by the Darwin name p and manages the service s as shown below:



provide p;

$\textsc{Prov}(p, s) \stackrel{\text{def}}{=} !(p(x).\overline{x}s)$

where

s – service reference

x – location at which s is required

p – access name

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 other components which use the service. The agent $\textsc{Prov}$ 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 $\textsc{Prov}$ is defined to be a replicated process (!) which will repeatedly send out the service reference each time a location is received.

Require The declaration of a required service, **require** r, is modelled by the agent $\textsc{Req}(r, l)$ which is accessed by the Darwin name r and which manages the location l 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 $\textsc{Req}$ agent receives the access name to a $\textsc{Prov}$ agent and sends the location l to that agent. A requirement in Darwin may only be bound to a single service and so the agent $\textsc{Req}$ sends out the location l precisely once as shown below:

 **require** r;	$\textsc{Req}(r, l) \stackrel{\text{def}}{=} r(y).\overline{y}l$ where l – location at which service is required y – name of service provider r – access name

Bind The binding construct in Darwin is modelled by the $\textsc{Bind}$ agent which simply sends the access name of the $\textsc{Prov}$ agent to the $\textsc{Req}$ agent.

 **bind** r -- p;	$\textsc{Bind}(r, p) \stackrel{\text{def}}{=} \overline{r}p$ where r – name of requirement p – name of provision

Initially, we will ignore the fact that $\textsc{Prov}$ and $\textsc{Req}$ agents are always contained within a component and examine the effect of binding on these agents. Firstly, the composition of $\textsc{Req}$ and $\textsc{Bind}$:

$$\begin{array}{llr} \text{Substituting definitions:} & \textsc{Req}(r, l) \mid \textsc{Bind}(r, p) \equiv r(y).\overline{y}l \mid \overline{r}p & \\ \text{Communication along } r\text{:} & \qquad\qquad\qquad\qquad \to \overline{p}l & (1) \end{array}$$

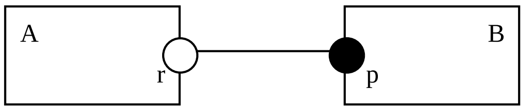
In other words, the composition of a $\textsc{Req}$ agent with a $\textsc{Bind}$ agent produces a binding request of the form $\overline{p}l$, in which the location at which the service name is required l is sent along the access channel p of the $\textsc{Prov}$ agent. When composed with the $\textsc{Prov}$ agent, the binding request results in a binding as follows:

$$\begin{array}{llr} \text{Using } !P = P \mid !P\text{:} & \overline{p}l \mid \textsc{Prov}(p, s) \equiv \overline{p}l \mid p(x).\overline{x}s \mid \textsc{Prov}(p, s) & \\ \text{Communication along } p\text{:} & \qquad\qquad\qquad \to \overline{l}s \mid \textsc{Prov}(p, s) & (2) \end{array}$$

The result of composing a $\textsc{Req}$, $\textsc{Prov}$ and $\textsc{Bind}$ agent is thus the binding in which the name of the service s is sent to the place l where it is required. The $\textsc{Prov}$ agent remains to allow further bindings. The following section looks at the effect of these agents in the context of a simple Darwin configuration.

3.2 Components and non-hierarchical configurations

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. This is illustrated by translating the simple non-hierarchical Darwin configuration of figure 4 into the π-calculus.




```
component Server {
  provide p;
}

component Client {
  require r;
}
```

```
component System {
inst
  A:Client;
  B:Server;
bind
  A.r -- B.p
}
```

Fig. 4. Client Server configuration

Components 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:

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

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:

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

Note that the scope of the service name s is local to the *Server* agent and similarly, the name of the place at which the service is required l is also local to the *Client*. As can be seen in the following, binding extends the scope of these names.

3.3 Non-hierarchical configuration

The configuration of the *System* component of figure 4 is represented in π-calculus by the parallel composition of a *Client*, *Server* and BIND agent:

$$System \stackrel{\text{def}}{=} (\nu\ r_A, p_B)(Client(r_A) \mid Server(p_B) \mid \textsc{Bind}(r_A, p_B)).$$

The instance names A and B in figure 4 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. 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 is elaborated. 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:

$$(\nu\ r_A, p_B)((\nu\ l)\textsc{Req}(r_A, l) \mid (\nu\ s)\textsc{Prov}(p_B, s) \mid \textsc{Bind}(r_A, p_B))$$

$$\begin{array}{ll} \rightarrow (\nu\ p_B)((\nu\ l)\overline{p_B}l \mid (\nu\ s)\textsc{Prov}(p_B, s)) & \text{Applying (1)} \\ \rightarrow (\nu\ l, s, p_B)(\overline{l}s \mid \textsc{Prov}(p_B, s)) & \text{Applying (2)} \end{array}$$

The expression describing the client server system thus reduces to an expression which sends the service s to the required location l 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:

$$Client'(l) \stackrel{\text{def}}{=} l(x).Client''.$$

$$\begin{array}{ll} \text{The system:} & (\nu\ l, s, p_B)(\overline{l}s \mid \textsc{Prov}(p_B, s) \mid Client'(l) \mid Server'(s)) \\ \text{then reduces to:} & \rightarrow (\nu\ s, p_B)\textsc{Prov}(p_B, s) \mid Client''(s/x) \mid Server'(s) \end{array}$$

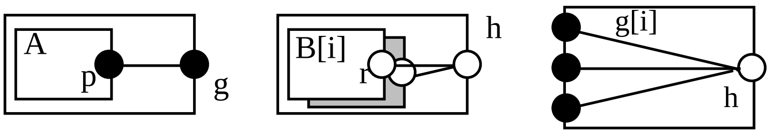
which is the desired result of an instance of the server component executing in parallel with a client component in which every occurrence of the local name x has been replaced with the name s, the reference to the required service. In practice, a Darwin implementation can compute the number of requirements 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'*).

3.4 Composite components and hierarchic binding

Hierarchic binding occurs in a composite component to bind the interface provisions and requirements to the constituent component instances. These hierarchic bindings take one of the three forms depicted in figure 5.

Case (a) is the export of an internal provision to form part of the interface of the composite component. An interface **provide** may only be bound to a single

(a) **(b)** **(c)**



(a)
```
provide g;
bind g -- A.p;
```

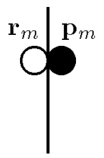
(b)
```
require h[n];
forall i:0..n-1
  bind B[i].r -- h;
```

(c)
```
provide g[n];
require h;
forall i:0..n-1
  bind g[i] -- h;
```

Fig. 5. Hierarchic bindings

instance **provide** since a service can only be implemented by a single primitive component instance. Case (b) is the situation where one or more internal instance **require**s (of the same service type) are bound to an interface **require** to reflect the situation that the required service is to be provided outside the composite component. The remaining case (c) is the situation where one or more interface **provide**s are not implemented inside the composite component but are bound to an interface **require**. These connection only components are useful for defining connection patterns and in addition, they may form the base case of recursively defined structures.

In case (a) and (c), the interface **provide** must act like a Req agent for bindings inside the composite component and as a Prov agent for external bindings made at the next level of the configuration. The opposite is necessary in case (b) and (c), where the interface require must act like a Prov agent internally and a Req agent externally [16]. Interface **require**s and **provide**s are modelled in the π-calculus by the Var agent which combines the behaviour of Req and Prov as shown below.

	$\text{Var}(p_m, r_m) \stackrel{\text{def}}{=} r_m(x).!(p_m(y).\overline{x}y)$ where r_m – access name for require side of m p_m – access name for provide side of m x – name bound to r_m y – location service required at

We will show in the following that the effect of binding a Var agent is to create an agent which simply passes on binding requests.

$$\text{Var}(p_m, r_m) \mid \text{Bind}(r_m, p) \equiv r_m(x).!(p_m(y).\overline{x}y) \mid \overline{r_m}p$$

$$\begin{array}{ll} \text{Communication along } r_m: & \rightarrow !(p_m(y).\overline{p}y) \\ \text{Defining } \text{Pass}(m,n) \stackrel{\text{def}}{=} !(m(y).\overline{n}y): & \equiv \text{Pass}(p_m, p) \end{array} \tag{3}$$

Composing a binding request for m_p with this PASS agent transforms it into a binding request for p as follows.

$$\begin{array}{lr} \text{Using } !P = P \mid !P: & \overline{p_m}l \mid \text{PASS}(p_m, p) \equiv \overline{p_m}l \mid p_m(y).\overline{p}y \mid \text{PASS}(p_m, p) \\ \text{Communication along } p_m: & \rightarrow \overline{p}l \mid \text{PASS}(p_m, p) \end{array} \tag{4}$$

The intuition here is that when a VAR agent is bound to the access name of a provided service (either PROV or VAR) it is transformed into a PASS agent which forwards binding requests to that service.

3.5 Correctness of Program Elaboration

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 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. For example, the system of figure 6 should reduce to a system in which the service reference s has been placed in the required places l_1 and l_2.

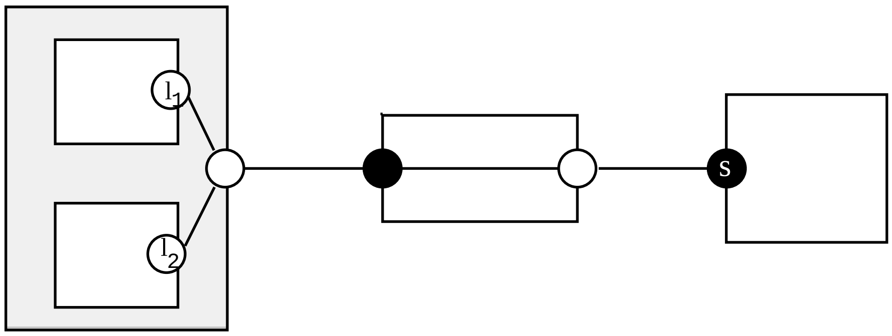


Fig. 6. System with hierarchic bindings

In general, 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. Figure 7 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 7. The correctness condition

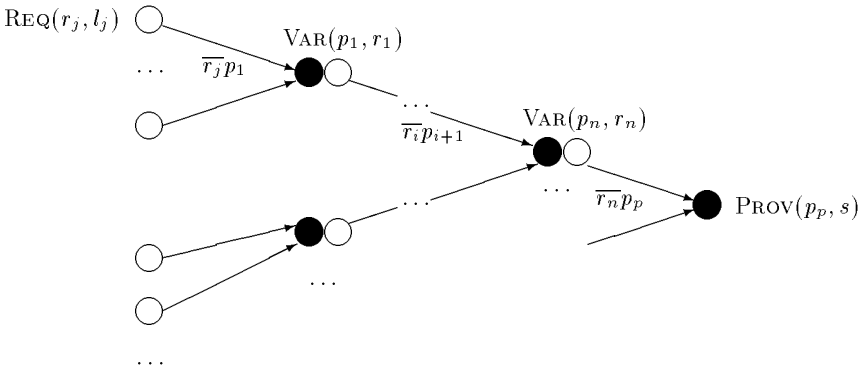


Fig. 7. General Configuration Graph

for elaborating a Darwin configuration is thus:

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

To prove this, we must demonstrate that the following system, where $i : 1..n-1$, produces the binding $\overline{l_j}s$:

$$\mathrm{REQ}(r_j, l_j) \mid \overline{r_j}p_1 \mid \ldots \mid \mathrm{VAR}(p_i, r_i) \mid \overline{r_i}p_{i+1} \mid \ldots \mid \mathrm{VAR}(p_n, r_n) \mid \overline{r_n}p_p \mid \mathrm{PROV}(p_p, s).$$

$$\begin{array}{lll}
\mathrm{REQ}(r_j, l_j) \mid \overline{r_j}p_1 & \rightarrow \overline{p_1}l_j & \text{Using 1} \\
\mathrm{VAR}(p_i, r_i) \mid \overline{r_i}p_{i+1} & \rightarrow \mathrm{PASS}(p_i, p_{i+1}) & \text{Using 3} \\
\overline{p_1}l_j \mid \mathrm{PASS}(p_i, p_{i+1}) & \rightarrow \overline{p_n}l_j \mid \mathrm{PASS}(p_i, p_{i+1}) & \text{Using 4} \\
\mathrm{VAR}(p_n, r_n) \mid \overline{r_n}p_p & \rightarrow \mathrm{PASS}(p_n, p_p) & \text{Using 3} \\
\overline{p_n}l_j \, \mathrm{PASS}(p_n, p_p) & \rightarrow \overline{p_p}l_j \, \mathrm{PASS}(p_n, p_p) & \text{Using 4}
\end{array}$$

Dropping the PASS processes, the system becomes:

$$\overline{p_p}l_j \mid \mathrm{PROV}(p_p, s) \qquad \text{Using 2}$$

The above π-calculus model 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.

The Darwin compiler cannot statically detect two categories of incorrect bindings. These incorrect bindings can therefore 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 3.2, this simply causes the component containing that requirement to be blocked. The more interesting binding error is depicted in figure 8 in which a requirement is bound to a cycle of interface entities.

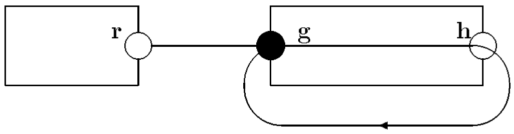


Fig. 8. Cyclic binding error

While the simple case of cyclic binding depicted in figure 8 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 8 is:

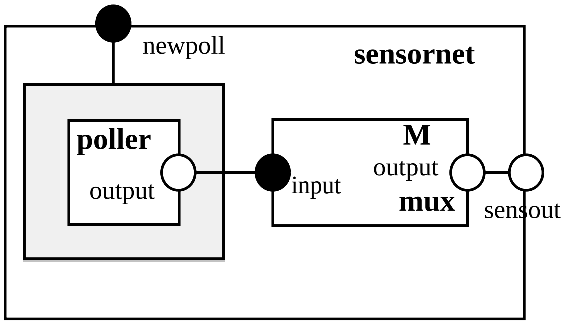
$$\begin{aligned} &\textsc{Req}(r,l) \mid \overline{r}p_g \mid \textsc{Var}(p_g, r_g) \mid \overline{r_g}p_h \mid \textsc{Var}(p_h, r_h) \mid \overline{r_h}p_g \\ &\rightarrow \overline{p_g}l \mid \textsc{Pass}(p_g, p_h) \mid \textsc{Pass}(p_h, p_g) \\ &\rightarrow \overline{p_h}l \mid \textsc{Pass}(p_g, p_h) \mid \textsc{Pass}(p_h, p_g) \\ &\rightarrow \overline{p_g}l \mid \textsc{Pass}(p_g, p_h) \mid \textsc{Pass}(p_h, p_g) \end{aligned}$$

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.

4 Darwin and Dynamic Architectures

In the previous section, we have described the basic features of Darwin, concerned with binding, instantiation and hierarchy, and their semantics in the π-calculus. These features allow the specification of static structures which do not change once elaborated. Darwin also has the ability to specify architectures which change at runtime using lazy and direct dynamic instantiation. In the following, we briefly describe the direct dynamic instantiation facility and its π-calculus model.

Direct dynamic instantiation permits the definition of structures which can evolve in an arbitrary way. 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 specification. Figure 9 is an example of a component which creates new *poller* components in response to the requests of an external manager. The example comes from a distributed system which monitors the location of Active Badges [17]. Active Badges emit infrared signals which are picked up by sensors distributed around a building. Each *poller* component monitors a string of sensors. New *poller* components need to be created as the system is extended.




```
component sensornet {
  require sensout<port smsg>;
  provide newpoll<dyn int>;

  inst
    M:mux;
  bind
    M.output -- sensout;
    poller.output -- M.input;
    newpoll-- dyn poller;
}
```

Fig. 9. Dynamic instantiation

The provided service *newpoll* is bound to the service **dyn** *poller* to satisfy this need. Invoking the service creates a new *poller* instance and passes it a single integer parameter. Note that in figure 9, bindings are specified for the component type *poller* 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 *poller* 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, nevertheless, dynamically created components may provide services. 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.

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 use the system of figure 2 and modify it so that *Client* components can be created through the service d:

```
component System {
    provide d <dyn>;
    inst B:Server;
    bind
        d -- dyn 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 $\overline{m}$ will cause a new replica of the *Client* component to be instantiated together with its associated bind action. In general, the action would be $\overline{m}\ \vec{x}$ where $\vec{x}$ represents the vector of parameters for the newly instantiated component.

$$(\nu\ p_B, d, m)(Server(p_B) \mid \textsc{Prov}(d, m) \mid !(m().(\nu\ r)(Client(r) \mid \overline{r}p_B)))$$

Dynamic instantiation does not change the basic model of section 3. 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 Discussion and Conclusions

Darwin has little impact on the internal structure and behaviour of the primitive components it configures. Components may be sequential, concurrent or distributed. They are only required to supply the names or references of services and accept bindings. The π-calculus description of Darwin clearly illustrates this separation between architecture and computation/communication in systems constructed using Darwin. Unlike Allan and Garlan [18], we do 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 [19]. 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 3 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 4 showed that one of the Darwin features concerned with dynamic configuration could easily be modelled without disturbing the basic elaboration algorithm. Further extensions can be tested against the criteria that they do not adversely affect or complicate elaboration. Work is currently in progress to provide open systems binding, the ability to manage group communication abstractions and component migration. We are also extending the definition of the Darwin language together with its π-calculus semantics to capture the notion of architectural styles [20].

We have chosen to ignore component parameterisation in arriving at the π-calculus model. 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.

One of the major benefits of using the π-calculus to model Darwin has been our increased understanding of the role and nature of configuration languages. Rice and Seidman [21] chose to use the Z specification language [22] as a means for modelling component instantiation, interconnection, and hierarchical composition for configuration languages such as Conic. We felt that the process algebras might be more appropriate to model component interaction and elaboration. Initially, we attempted to define the semantics of Conic using the CCS [23] and CSP [24] formalisms. While it 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 consider 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 [25] 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.

Finally, we would like to emphasise that, together with others, we have accumulated extensive experience in using Darwin for constructing distributed systems [10, 11]. We are therefore confident in proposing Darwin as both a practical and sound means for specifying and manipulating the software architecture of distributed systems.

The authors would like to acknowledge discussions with our colleagues in the Distributed Software Engineering Section Group during the formulation of these ideas. We gratefully acknowledge the DTI (Grant Ref: IED 410/36/2) and the EPSRC (Grant Ref: GR/J52693) for their financial support.

References

1. D. Garlan and M Shaw, *An Introduction to Software Architecture*, in Advances in Software Engineering and Knowledge Engineering, Vol. 1, ed. Ambriola and Tortora, World Scientific Publishing Co., 1993.
2. D.E. Perry and A.L. Wolf, *Foundations for the study of Software Architectures*, ACM SIGSOFT, Software Engineering Notes, 17 (4), 1992, pp 40-52.
3. D. Garlan and D. Perry, *Software Architecture: Practice, Potential and Pitfalls* (Panel Introduction), Proc. of 16th Intl. Conf. on Software Engineering, S orrento, May 1994.
4. J. Magee, J. Kramer, and M. Sloman, *Constructing Distributed Systems in Conic*, IEEE Transactions on Software Engineering, SE-15 (6), 1989.
5. J. Kramer, J. Magee, M. Sloman and N. Dulay, *Configuring Object-Based Distributed Programs in* REX, IEE Software Engineering Journal, Vol. 7, 2, March 1992, pp139-149.
6. J. Magee, N. Dulay and J. Kramer, *Structuring Parallel and Distributed Programs*, IEE Software Engineering Journal, Vol. 8, No. 2, March 1993, pp73-82.
7. J.M. Purtilo, *The POLYLITH Software Bus*, ACM Transactions on Programming Languages, 16(1), January 1994, pp 151-174.
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. H. Graves, *Lockheed Environment for Automatic Programming*, Proc. of KBSE 91, 6th IEEE Knowledge Based Software Engineering Conference, 1991, pp 68-76.
10. J.Magee, N. Dulay and J. Kramer, *Regis: A Constructive Development Environment for Distributed Programs*, Distributed Systems Engineering Journal, to appear.
11. S. Crane, N. Dulay, H. Fosså, J. Kramer, J. Magee, M. Sloman and K. Twidle, *Configuration Management for Distributed Systems*, to be presented at ISINM 95.
12. ANSA*ware 4.1: Application Programming in* ANSA*ware*, Document RM.102.02, Architecture Projects Management Agency, Poseidon House, Cambridge Feb. 1993.
13. R.Milner, J. Parrow, and D.Walker, *A calculus of mobile processes, Parts I and II*, Journal of Information and Computation, Vol. 100, pp 1-40 and pp 41-77, 1992.
14. K. Ng, J. Kramer, J. Magee and N. Dulay, *The Software Architect's Assistant - A Visual Environment for Distributed Programming*, HICSS-28, January 1995.
15. 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.
16. S. Eisenbach and R. Paterson, *π-Calculus Semantics for the Concurrent Configuration Language Darwin*, HICSS-26, January 1993.
17. A. Harter and A. Hopper, *A Distributed Location System for the Active Office*, IEEE Network, Jan./Feb. 1994, pp. 62-70.
18. R. Allan and D. Garlan, *Formalizing Architectural Connection*, Proc. of 16th International Conference on Software Engineering, Sorrento, May 1994.

19. M. Radestock and S. Eisenbach, *What Do You Get From a π-calculus Semantics?*, PARLE 94, Springer-Verlag, LNCS No. 817, pp635-647, 1994.
20. G. Abowd, R. Allen and D. Garlan, *Using style to give meaning to software architecture*, In Proceedings of the SIGSOFT'93: Foundations of Software Engineering, Software Engineering Notes 118(3), pp.9-20, ACM Press, Dec. 1993.
21. M.D. Rice and S.B. Seidman, *A Formal Model for Module Interconnection Languages*, IEEE Transactions on Software Engineering, 20 (1), 1994, 88-101.
22. J.M. Spivey, *The Z Notation, a Reference Manual*, Prentice Hall, Englewood Cliffs, N.J., 1989.
23. R. Milner, *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, N.J., 1989.
24. C.A.R. Hoare, *Communicating sequential processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.
25. R. Milner, *Elements of Interaction - Turing Award Lecture*, CACM, Vol 36, No. 1, January 1993, pp78-79.

This article was processed using the LaTeX macro package with LLNCS style