Dynamic Structure in Software Architectures

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

jnm,jk@doc.ic.ac.uk

Abstract

Much of the recent work on Architecture Description Languages (ADL) has concentrated on specifying organisations of components and connectors which are static. When the ADL specification is used to drive system construction, then the structure of the resulting system in terms of its component instances and their interconnection is fixed. This paper examines ADL features which permit the description of dynamic software architectures in which the organisation of components and connectors may change during system execution.

The paper outlines examples of language features which support dynamic structure. These examples are taken from Darwin, a language used to describe distributed system structure. An operational semantics for these features is presented in the π-calculus, together with a discussion of their advantages and limitations. The paper discusses some general approaches to dynamic architecture description suggested by these examples.

1 Introduction

Software architecture is intended to describe "*..the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time* [8]. It is a critical design concern when bridging the gap between requirements and implementations [7,8,9,10,25,28]. Architectural Description Languages (ADLs) are notations for expressing and representing architectural designs and styles. They describe the high level structure of a system in terms of components and component interactions, often referred to as structural models. To date, much of this work has concentrated on providing precise descriptions of connectors which provide the "glue" for combining components into systems[9,10] and accommodating diverse connector types [2,28]. As exemplified by UniCon[28], they describe the overall system structure by declaring a static set of component instances and connectors.

SIGSOFT '96 CA, USA

For a number of years we have been involved in developing configuration specification languages for use in the design and construction of complex and scalable distributed systems [15,17,18]. These languages have much in common with ADLs in describing a system as a configuration of connected component instances. They differ mainly in restricting connectors to those compatible with distribution [3,26]. Darwin, the latest in a line of configuration languages, is a *declarative* language which is intended to be a general purpose notation for specifying the structure of distributed systems composed from diverse components using diverse interaction mechanisms. It deliberately divides the description of structure from that of computation and interaction in order to provide a clear separation of concerns. Darwin supports software composition [24] through the description of generic software architectures which can be elaborated and instantiated to form specific executable architectures. It is currently being used in the context of the Regis system[19] which supports multiple interaction primitives and in the Sysman project[6] with CORBA[29] which uses remote object invocation for component interaction. Darwin's declarative property has facilitated the provision of an operational semantics and the ability to reason about structural aspects such as the correctness of the distributed algorithm used for the elaboration of Darwin software [20].

Darwin can be used to specify static system structures in much the same way as UniCon. Unlike UniCon, through the use of conditional and replicator constructs, Darwin allows parameters to determine the system structure at initialisation time. UniCon fixes structure in the specification. Further, Darwin has features which permit the description of dynamic structures which evolve as execution progresses. Structural evolution includes changes in both the bindings (connections) between components and the set of component instances. These structural changes can be expressed without violating the declarative nature of Darwin, thereby facilitating both a semantic description and reasoning. This paper concentrates on Darwin's dynamic features to act as a focus for discussion of general approaches to the specification of dynamic software architectures. The motivation is to widen the scope of applicability of ADLs and thereby gain the associated

benefits for dynamic and evolving architectures. The paper describes two specific techniques used in Darwin to capture dynamic structures, lazy instantiation and direct dynamic instantiation. In addition, dynamic binding is discussed with respect to open systems and abstract services. The π-calculus[21] is used to specify these dynamic constructs. This builds on earlier work which developed a π-calculus model for the static aspects of Darwin and demonstrated the correctness of the elaboration of Darwin programs[20]. The π-calculus is used here to ensure that the dynamic constructs are compatible with the existing elaboration scheme. It is used to give an operational semantics to Darwin constructs, and is not intended for direct consumption by the system architects who are the target users of Darwin.

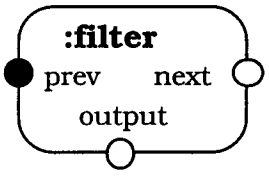
The next section gives a brief overview of Darwin. A more comprehensive description may be found in [19,20]. Section 3 outlines how the basic static features of Darwin are given an operational semantics in the π-calculus. The remaining sections extend this operational specifiation to capture the dynamic aspects of Darwin. Section 4 describes lazy instantiation, a way of describing structures in which the size can vary dynamically within a constrained pattern. Section 5 describes direct dynamic instantiation which permits unconstrained structural evolution. Section 6 outlines open systems binding which permits the specification of interactions with external systems. Section 7. describes the role of dynamic binding. The paper concludes with some general observations and conclusions on specifying dynamic structures.

2 Darwin

Darwin allows distributed programs to be specified as a hierarchic construction of components. 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. Each inter-component interaction is represented by a binding between a required service and a provided service. Darwin has both a graphical and textual representation. The Darwin specification of a system architecture is used as a framework for structuring behavioural specifications during design and analysis and is used directly to drive system building during construction.

2.1 Components & 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 **provide**s a single service *prev* and **require**s two services *next* and *output*. 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 interaction mechanism used to implement the service is a port which accepts messages of type *int*. Darwin does not interpret service type information. Service type information is either interpreted by the underlying behaviour specification formalism used during design and analysis[4] or, as in the example, denotes a communication mechanism supported by the underlying distributed platform used in building an implementation. In the Regis system[19], this information is used to directly select the correct communication code. In addition to a number of predefined communication classes, Regis permits users to define their own. When used with a more conventional distributed platforms such as CORBA[29] based systems, the service type names an IDL specification which is the used to generate the correct client and server stubs.




```
component filter {
  provide prev   <port,int>;
  require next   <port,int>,
          output <port,int>;
}
```

Figure 1 - component type *filter*

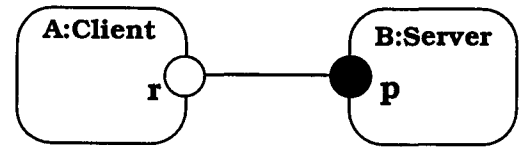
In general, a component may provide many services and require many services. It should be noted that the names of required and provided services are local to the component type specification. A component does not need to know the global names of external services or where they are to be found in the distributed environment. Components may thus be specified, 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 Instantiation & Binding

The primary purpose of the Darwin configuration language is to allow system architects to construct composite component types from both instances of basic computational components and 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. Composite components and systems are thus formed in Darwin by declaring instances of components and binding the services required by one

component to the services provided in another as shown in figure 2 for a simple client server system.

A binding is only legal if the service type of the requirement matches the service type of the provision. As noted before, Darwin only manages service types, it does not interpret them so the matching predicate must be supplied by the system being used to specify service type. In the current Darwin toolset the default is to do a simple name equivalence test. Many requirements may be bound to a single service provision (many-to-one), however, a service requirement may only be bound to a single service provision. The problem of using Darwin to describe architectures using multicast is dealt with later in the paper.




```
component Server {
  provide p;
}

component Client {
  require r;
}
```

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

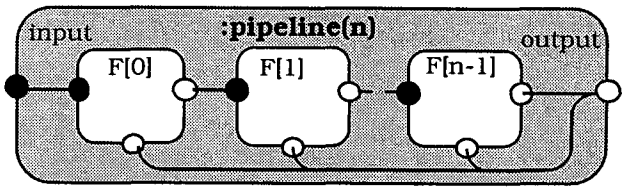
Figure 2 - Client Server configuration

2.3 Guarded and replicated configurations

The example of figure 3 defines a variable length pipeline of filter instances in which the input of each instance is bound to it predecessors output. The length of the pipeline is determined by a parameter to the composite component which is substituted at elaboration time of the Darwin configuration program.

The pipeline component type is implemented by an array of filter instances dimensioned by the ***array*** declaration. The replicator construct ***forall*** *range* declares the actual instances and their bindings. Each instance must be declared explicitly since they may have different parameter values (although not in this example). The guard construct ***when*** *expression*, only includes associated bindings and instances in an elaborated system if the associated expression evaluates to true.

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[n-1]* requirement *next* which is bound to *output*. Similarly services provided internally which are required outside are bound to an interface service provision e.g. *input -- F[0].prev*. Since an interface requirement represents an external provision, it is consistent that many internal requirements may be bound to an interface requirement e.g. *F[k].output -- output*.




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

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

Figure 3 - composite component type *pipeline*

This section has given an outline of the basic features of Darwin needed to define static architectures. Darwin also permits recursively defined components and allows component types as parameters so that template component types can be defined[19]. However, in this paper we concentrate on the dynamic aspects of Darwin and only a minimal set of the static features have been covered as a necessary basis for discussing the dynamic constructs.

3 π-calculus model for Darwin

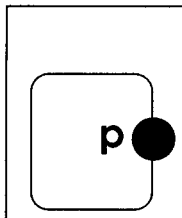
In the following, an operational specification for the main static aspects of Darwin is given in the π-calculus[21]. The correspondence between the treatment of names in the π-calculus and the management of services in Darwin leads to an elegant and concise π-calculus model of Darwin's operational semantics[20]. For the reader unfamiliar with π-calculus, a brief overview of the notation used here may be found in the Appendix.

Note that the separation of concerns - structure from computation and interaction - facilitates the provision of an operational semantics for Darwin as a "pure" structural language. We avoid consideration of the interactions (cf. connectors [2]) and/or components (cf. CHAM [12] or LTS [4] models), but focus on services and binding. This does not preclude specification of the interaction and/or computation, but allows us to modularise our approach. This focus is matched by that of the π-calculus. Furthermore, as

we will see later, it facilitates the specification of the dynamic aspects of architecture in Darwin.

3.1 Provide & Require

Provide - 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 Darwin name p and manages the service s as shown below:



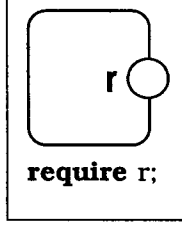
provide p;

$$PROV(p, s) \stackrel{def}{=} \; !(p(x).\bar{x}s)$$

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 *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.

Require - The declaration of a required service, *require r,* is modelled by the agent *REQ(r,o)* which is accessed by the Darwin 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 as show below:



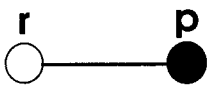
require r;

$$REQ(r, o) \stackrel{def}{=} \; r(y).\bar{y}o$$

where
o- location at which service is required
y - name of service provider
r - access name

3.2 Binding & Components

Bind - The binding construct in Darwin is modelled in the π-calculus by the *BIND* agent which simply sends the access name of the *PROV* agent to the *REQ* agent.



bind r -- p;

$$BIND(r, p) \stackrel{def}{=} \; \bar{r}p$$

where
r - name of requirement
p - name of provision

Initially, we will ignore the fact that *PROV* and *REQ* agents are always contained within a component and examine the effect of binding on these agents. Firstly, the composition of *REQ* and *BIND*:

Substituting definitions:

$$REQ(r, o) \mid BIND(r, p) \equiv r(y).\bar{y}o \mid \bar{r}p$$

Communication along r: $\underline{\rightarrow \bar{p}o}$ ---------- (1)

In other words, the composition of a *REQ* agent with a *BIND* agent produces a binding request of the form $\bar{p}o$, in which the location at which the service name is required o is sent along the access channel p of the *PROV* agent. When composed with the *PROV* agent, the binding request results in a binding as follows:

Using $!P \equiv P \mid !P$:

$$\bar{p}o \mid PROV(p, s) \equiv \bar{p}o \mid p(x).\bar{x}s \mid PROV(p, s)$$

Communication along p: $\underline{\rightarrow \bar{o}s \mid PROV(p, s)}$ ---- (2)

The result of composing a *REQ*, *PROV* and *BIND* agent is thus the binding $\bar{o}s$ in which the name of the service s is sent to the place o where it is required. The *PROV* agent remains to allow further bindings.

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 2 is represented by the π-calculus agent:

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

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:

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

The configuration of the *System* component of figure 2 can now be represented by the parallel composition of a *Client*, *Server* and *BIND* agent:

$$System \stackrel{def}{=} (\nu a_r, b_p)(Client(a_r) \mid Server(b_p) \mid BIND(a_r, b_p))$$

---------- (3)

This can be reduced using (1) & (2) to demonstrate that the correct binding between client and server is made:

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

The term $\bar{o}s$ sends the service s to the required location o. Before the client can use the service it must perform an input action. A possible definition for *Client'* would be:

$$Client'(o) \stackrel{def}{=} (\nu x)(o(x).Client'')$$

The client server system then reduces to:

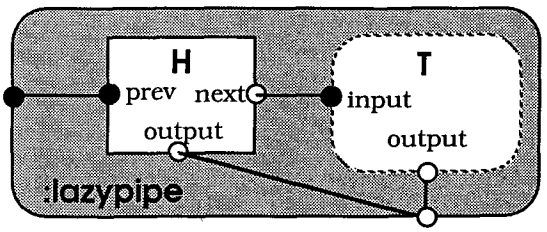
$$\rightarrow (\nu s, b_p)((\nu x)Client''\{s/x\} \mid Server(b_p))$$

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.

Interface requirements and provision such as those used in figure 3 require an extension to the π-calculus model for Darwin developed in the above. However, this is not necessary for the following discussion of the dynamic features of Darwin and consequently will be omitted. A complete π-calculus model for the static elements of Darwin may be found together with a general correctness proof for the elaboration of Darwin configuration descriptions may be found in [20].

4 Lazy instantiation

Darwin provides two main 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 4.




```
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 4 - Lazy instantiation example

The program of figure 4 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* (*i.e. 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. Consequently, while the structure of the pipeline is fixed as in figure 3, the size of the pipeline can increase at runtime while that of figure 3 is determined at initialisation by its parameter. In this simple example, there is no guarantee that the elaboration process will terminate. Instances are created as required by computation.

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, when communication is initiated, triggers instantiation of the lazy instance and its associated bindings. For example, consider the system of figure 2 with lazy instantiation of the server component:

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

The π-calculus model for this system becomes:

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

in which the *Client* is bound to the dummy provision d (cf. specification (3)). This reduces to:

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

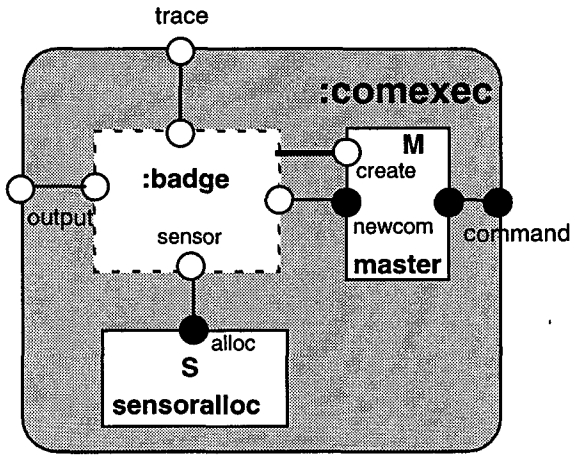
where o is the location at which the client requires the service as in section 2. Note that instantiation of the *Server* component is guarded by the prefix $b_p(y)$ and that the binding returned to the client is the name of this prefix. In a more complex system, the prefix would also guard the binding 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 $o(x)$. 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:

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

The agent *LAZY* outputs the signal true (t) or false(f) 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. A more satisfactory solution, and one which closely models the current Darwin 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[22] but it is beyond the scope of this paper.

The combination of lazy instantiation and recursion can be used to describe a wide range of commonly occurring distributed parallel processing structures (e.g. search trees, combining trees, divide & conquer). The advantage of this technique of specifying dynamic structure is that the configuration description is a precise specification which describes the potential structure at execution time. The components used in the structure need not be aware of whether they are being used in a statically or lazily elaborated structure. Lazy instantiation by itself is commonly found in distributed systems, where to save resources, servers are only created on demand (e.g. Unix *Netd*).

The limitation of this technique is that it is not possible to describe structures with cyclic bindings, for example, a ring of filter components. Inserting a new component into a ring requires breaking an existing ring interconnection by deleting an existing binding. Introducing an unbind or rebind operation into Darwin would make it an imperative rather than a declarative notation in which system structure would depend on the elaboration sequence of the Darwin description. In addition, rebinding without coordination between components could cause undesirable application side effects arising from trying to change a binding while it is being used for interaction. Imperative reconfiguration in which the system is directed to change structure is handled at a management level of systems using Darwin[6]. Essentially, as discussed later, a managed system exports rebinding services which control access to existing bindings and synchronise change with component interaction.




```
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 5 - Direct dynamic instantiation example

5 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 5 is an example taken from an infrared Active Badge location system [11] implemented using Darwin and Regis [19]. The *comexec* component forms part of the badge management server which runs on a local area network and uses a set of fixed infrared tranceivers to communicate with the mobile badges. *Comexec* has a supervisor-worker architecture and is responsible for sending commands to badges. Commands activate paging signals (audible and visual) on the active badge. A *badge* worker component is created to deal with each new request received by *comexec* to send a command to an active badge. This *badge* component deals with locating the physical badge, reserving the nearest transceiver to transmit the infrared command and implements a protocol to ensure that commands are reliably executed. The *master* supervisor 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 5 by binding it to the component type *badge* prefixed by the keyword **dyn**. i.e. *M.create -- **dyn** badge*. Note that in figure 5, 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. This would otherwise result in the potentially ambiguous situation of requiring a service from multiple providers. 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.

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 2 and modify it so that *Client* components can be created dynamically through the service *d*:

```
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 $\bar{m}$ 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.

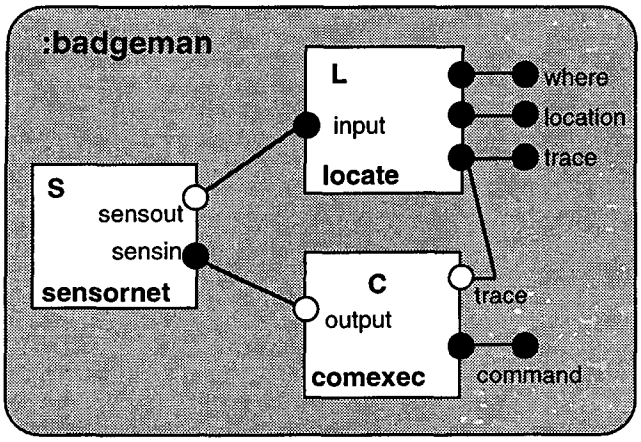
$$(\nu b_p, d, m)(Server(b_p) \mid PROV(d, m) \mid !(m.(\nu r)(Client(r) \mid BIND(r, b_p))))$$

Note that the way in which a component is instantiated, statically or dynamically, does not change the definition of that component.

6 Open systems binding

It is intended that systems constructed 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. Bindings achieved by export and import are orthogonal to the inter-component and hierarchical bindings used to express Darwin structures. Consequently, exported and imported services are not drawn at the boundary of a component but are depicted as in figure 6.

The example program of figure 6 includes the *comexec* component described earlier into a composite component which constitutes 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 **imports**.




```
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 6 - Exporting services

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

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

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 4. Requirements are initially bound to a dummy provision *i*.

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

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.

The set of exported and imported services provide a clear specification of the boundary between a system specified in Darwin and the external environment in which it exists. Both lazily and dynamically instantiated components may import and export services allowing this interface with the environment to be changed dynamically over time.

7 Dynamic Binding

7.1 Interaction specific binding

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.

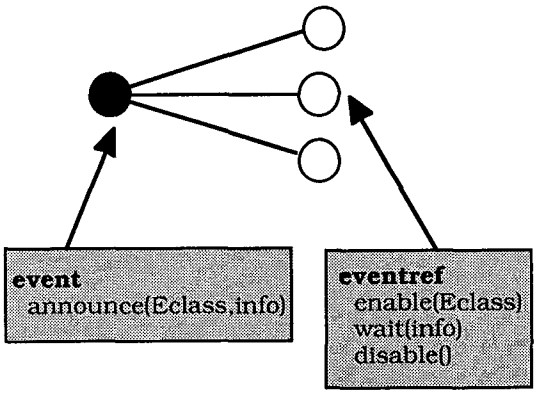
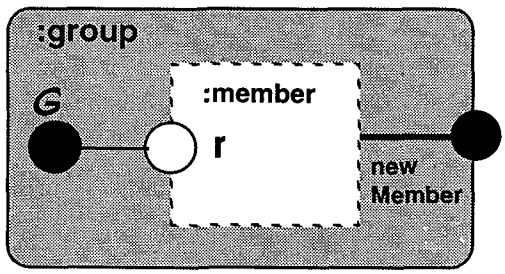


Figure 7 - Event interaction mechanism

An example of extended binding is found in the Regis event distribution mechanism which allows the provider of an event service to transmit information to many receivers as shown in figure 7. 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.

7.2 Abstract Services

Direct dynamic instantiation permits sets of anonymous components to be created at runtime and in addition, through type specific bindings, it permits the definition of the service environment for these components. If these components wish to interact directly then they can exchange service references through a third party component. The disadvantage is that, as mentioned above, the Darwin description does not capture these bindings. A limited solution to allow dynamically instantiated components to interact directly exists in Darwin for the case of group communication as shown in Figure 8.




```
component group {
  provide newMember;
  service G;
  bind
     Member.r -- G;
     newMember -- dyn member;
}
```

Figure 8 - Group Service

Member components may be created dynamically via the service *newMember*. They interact directly via the abstract service *G* which in an implementation becomes a group address for multicast communication.

8 Discussion and Conclusions

The paper has described two techniques for capturing dynamic structure in Darwin, an architectural description language for distributed systems. The first, lazy instantiation (figure 9a), is restricted in its application to situations where the designer can predict precisely the way system structure will evolve during execution. These structures must have acyclic bindings and have in practice been found to be mainly useful in the domain of parallel processing. The second technique, direct dynamic instantiation (figure 9b), allows arbitrary structures but permits the context or environment of the components to be precisely captured in the structural description. These two techniques are motivated by our desire to retain a declarative notation and to capture as much as is possible of the structure of a system in this notation while realising that for some systems, the system structure at runtime must be determined by data input at runtime. In addition, we have discussed support for dynamic binding both inside and outside Darwin.

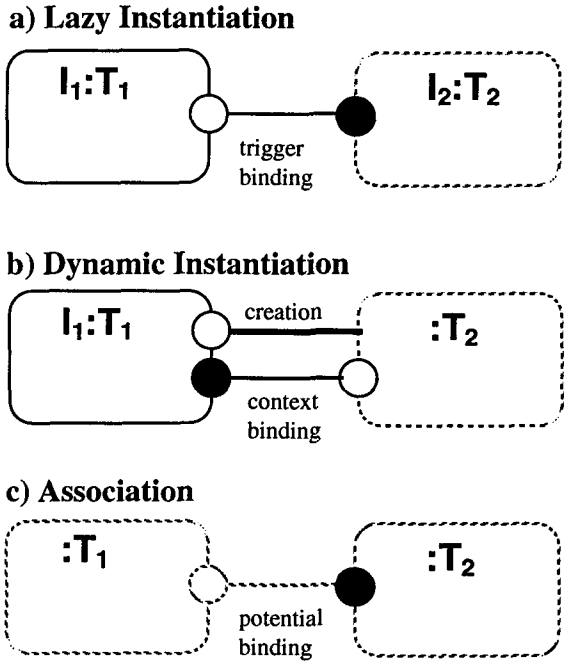


Figure 9 - Summary of Dynamic Structure in Darwin

Rapide [16] is one of the few architectural languages which also supports dynamic architectures, providing facilities for both dynamic connections and instantiation of components. Although there are many similarities in its structural aspects (architectural sublanguage), it is an event based language aimed at prototyping systems for examination of properties such as synchronisation and timing. As such, it is expected to execute and produce representative event patterns for the particular application. With Darwin we have deliberately

focused purely on the structural aspects and aimed to provide a language for both specifying and constructing distributed applications.

Retaining the declarative property for an ADL is important we believe in simplifying the analysis and possible transformation of architectures. However, it is critical in an ADL intended for use in construction of distributed systems. In particular, it permits concurrent and decentralised elaboration which is essential if ADLs are to be applied to large systems. An imperative ADL would impose strict ordering requirements on elaboration. However, where the architecture of a system must change in response to external influence at runtime then some imperative constructs are required. How do we retain the benefits of a declarative notation while permitting such change?

Meta-level Configuration

Those parts of the system that are fixed can be specified in the usual way in the Darwin language using direct dynamic instantiation mechanism to specify those parts of the system which depend on factors not determinable when the system is created. However, direct dynamic instantiation may be driven by interpreting a Darwin script at runtime. The structure of the interpreter and the services it can access is thus specified by the original Darwin specification. The structure of the overall system is this original Darwin specification plus the set of meta-level Darwin specifications which the original system executes as a result of either internal or external reconfiguration events. These reconfiguration events capture the imperative quality of change while the system specification remains declarative. Others, such as the reconfiguration plans of Clipper [1] and the reconfiguration actions of Durra [3], describe a similar use of imperative commands in response to system events, though with less support for maintaining the integrity of the system than that which we propose.

The execution of a meta-level Darwin specification may require the deletion of both components and bindings created by a previous specification. Previous work has identified an approach which minimises the disruption to a system while such change is accomplished [14]. This meta-level approach has been used in a prototype management system for distributed applications [6] in which the configuration manager is essentially a Darwin interpreter. This configuration manager uses generic direct dynamic instantiation and third party rebinding services to construct and modify applications. The instantiation service is generic in the sense that the type of component to be created is a parameter to the service.

Constrained Configuration

In some cases, where the timescale of dynamic change is short and the structure of connected components changes as a result of the computation being performed by the system, the meta-level configuration approach may exhibit too high a resource overhead. It may thus not be possible to describe the exact structure of the system at each stage of its evolution. In this case, it is possible using structural constraints to capture more of the structure in the specification than that currently provided by the direct dynamic instantiation technique. For example, a configuration constraint can specify an upper bound on the number of dynamically created instances. As shown in figure 9c, we can constrain the potential bindings between components by declaring the allowed associations between the requirements and provisions of component types. Association is used here to mean that an actual binding may occur at runtime. It is similar to the association concept found in class diagrams e.g. OMT[29]. In general, configuration constraints can limit both the number and type of dynamically created instances and the possible bindings between these instances. An earlier experiment used Prolog to specify constraints on Conic configurations [30]. Constraints did not play a part in the construction of a system from its architectural specification, however they were used to ensure that implementations conformed to their specification as in [23]. Others, such as the Raven configuration management system [5], have shown that constraints can be useful for recognising valid structures and for performing repairs.

In conclusion...

This paper has described our approach and the associated rationale for using a declarative ADL, Darwin, for distributed software architectures. In particular, we have concentrated on the dynamic features of Darwin, using it to illustrate some of the possibilities and problems of supporting constrained and unconstrained structural evolution, including open systems. We have tried to be precise in the features we have defined, providing an operational semantics in the π-calculus. Although the design and use of Darwin has been aimed at distributable software, we believe that the underlying concepts of components and binding forming the primitives of a "pure" structural language, independent of the interaction mechanisms between components, is more general and can be applied to more conventional program structures.

Acknowledgments

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 EPSRC (Grant Ref: GR/J52693) and the CEC (Framework IV ARES Project) for their financial support.

References

[1] B. Agnew, C. Hofmeister, J. Purtilo, *Planning for Change: a Reconfiguration Language for Distributed Systems*, Distributed Systems Engineering Journal, Vol. 1, No. 5., pp 313-322.

[2] R. Allan, D. Garlan, *Formalizing Architectural Connection*, Proc. of 16th International Conference on Software Engineering, Sorrento, May 1994.

[3] M. Barbacci, C. Weinstock, D, Doubleday, M. Gardner and R Lichota, *Durra: a structure description language for developing distributed applications*, IEE Software Engineering Journal, Vol. 8, No. 2, March 1993, pp 83-94

[4] S.C. Cheung, J. Kramer, *An Integrated Method for Effective Behaviour Analysis of Distributed Systems*, Proc. of 16th International Conference on Software Engineering, Sorrento, May 1994, pp 309-322.

[5] T. Coatta and G. Neufeld, *Distributed Configuration Management using Composite Objects and Constraints*, Distributed Systems Engineering Journal, Vol. 1, No. 5., pp 294-303.

[6] S. Crane, N. Dulay, H. Fosså, J. Kramer, J. Magee, M. Sloman, K. Twidle, *Configuration Management for Distributed Systems*, Proc. of the IFIP/IEEE International Symposium on Integrated Network Management (ISINM 95), Santa Barbara, May 1995, Chapman and Hall 1995.

[7] D. Garlan and D. Perry, *Software Architecture: Practice, Potential and Pitfalls (Panel Introduction)*, Proc. of 16th International Conference on Software Engineering, Sorrento, May 1994, pp 363-354.

[8] D. Garlan, *First International Workshop on Architectures for Software Systems: Workshop Summary*, ACM, SIGSOFT Software Engineering Notes, SEN 1995.

[9] D. Garlan and D.E. Perry, *Introduction to the Special Issue on Software Architecture*, IEEE Transactions on Software Engineering, 21 (4), April 1995, pp 269-274.

[10] D. Garlan, M Shaw, *An Introduction to Software Architecture*, in Advances in Software Engineering and Knowledge Engineering, Vol. I, ed. Ambriola and Tortora, World Scientific Publishing Co., 1993.

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

[12] P. Inverardi, A.L. Wolf, *Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model*, IEEE Transactions on Software Engineering, 21 (4), April 1995, pp 373-386.

[13] Keng Ng, Jeff Kramer, Jeff Magee and Naranker Dulay, *The Software Architect's Assistant - A Visual Environment for Distributed Programming*, Proceedings of Hawaii International Conference on System Sciences (HICSS-28), January 1995.

[14] J. Kramer and J. Magee, *The Evolving Philosophers Problem: Dynamic Change Management*, IEEE Transactions on Software Engineering, 16(11), 1990, pp 1293-1306.

[15] 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.

[16] D.C. Luckham et al., *Specification and Analysis of Software Architecture using Rapide*, IEEE Transactions on Software Engineering, 21 (4), April 1995, pp 336-355.

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

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

[19] J.Magee, N. Dulay, J. Kramer, *Regis: A Constructive Development Environment for Distributed Programs*, Distributed Systems Engineering Journal, Vol. 1, No. 5., pp 304-312.

[20] J.Magee, N. Dulay, S. Eisenbach and J. Kramer, *Specifying Distributed Software Architectures*, Proc. of 5th European Software Engineering Conference, ESEC '95, Barcelona, September 1995.

[21] 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.

[22] 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.

[23] N.H. Minsky, *Law-governed systems*. The IEE Software Engineering Journal, September 1991.

[24] O.Nierstrasz and T.D.Meijler, *Research Directions in Software Composition*, ACM Computing Surveys, Vol. 27, No. 2, June 1995, pp 262-264.

[25] D.E. Perry, A.L. Wolf, *Foundations for the study of Software Architectures*, ACM SIGSOFT, Software Engineering Notes, 17 (4), 1992, pp 40-52.

[26] J.M. Purtilo, *The POLYLITH Software Bus*, ACM Transactions on Programming Languages 4, pp 151-174.

[27] J. Rumbagh, M. Blaha, W. Premerlani, F. Eddy, W. Lorenson, *Object-Oriented Modelling and Design*, Prentice-Hall Internation Editions, 1991

[28] M. Shaw et al., *Abstractions for Software Architecture and Tools to Support Them*, IEEE Transactions on Software Engineering, 21 (4), April 1995, pp 314-335.

[29] *The Common Object Request Broker: Architecture and Specification*, Object Management Goup, Document OMG 91.12.1, December 1991.

[30] A.J. Young, and J. N. Magee, *A Flexible Approach to Evolution of Reconfigurable Systems*, Proc. of 1st IEE Int. Workshop on Configurable Distributed Systems, London, March 1992, pp 152-163.

Appendix -The π -calculus

The π-calculus[11] 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 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:

A::= action terms	$\bar{x}z.P$	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).
P::= terms	$A_1 + \ldots + A_n$	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, we will omit the stop process **0** in an agent and write $z(s)$ in place of $z(s).0$.