Presented at the International Conference on Configurable Distributed Systems (ICCDS'96), Annapolis, Maryland, May 1996. Proceedings published by IEEE Press.

Implementing Interactive Configuration Management for Distributed Systems

Halldor Fosså Morris Sloman

Department of Computing, Imperial College, 180 Queen's Gate, London, SW7 2BZ, UK.

Email: {h.fossa, m.sloman}@doc.ic.ac.uk

Abstract

This paper describes an environment for interactive configuration management of software for distributed applications and services. Configuration management involves creating the components which form a distributed system; allocating these components to physical nodes and binding the interfaces of the components to each other or to existing services. Components and their required and provided interfaces are represented and accessible in the system-wide domain structure. We describe a graphical configuration environment, based on the Darwin configuration language, which is used to create the required configurations and maintain the configuration structure as part of the overall systems management infrastructure. The paper describes a simple example to show how a system can be initially constructed and subsequently reconfigured at run-time without shutting down the system.

Keywords: *Component creation, interface binding, dynamic configuration, management user interface, domains*

1 Introduction

Configuring software components in distributed applications or services entails specifying the required object instances, bindings between their interfaces, bindings to external required services, and allocating components to physical nodes. Large distributed systems (e.g. telecommunications, multi-media or banking applications) introduce additional configuration management problems. These systems cannot be completely shut down for reconfiguration but must be dynamically reconfigured while the system is in operation. There is a further need to access and reconfigure resources and services controlled by different organisations. These systems are too large and complex to be managed by a single human manager. Consequently, we require the ability not only to partition configuration responsibility amongst an organisation's managers but also to permit controlled access to limited configuration capabilities by managers in different organisations.

This paper describes the implementation support to enable a manager to interactively configure the software components forming a distributed service using the Darwin configuration notation. This specifies the structure of the service as a *composite* component type which defines internal primitive or composite component instances and interface bindings [1]. A *primitive* component would be implemented in a distributed programming environment, and has application-dependent functionality. The functionality of a composite component is defined by its constituent primitive components. The external view of both composite and primitive components is in terms of their interfaces – those *required* by clients and those *provided* by servers.

Managers need to locate these interfaces to be able to bind them so they are registered in a *domain service* [2]. This is similar to a hierarchical directory system in a typical file service. *Domains* provide a means of grouping objects (i.e. interface references) and specifying a common policy which applies to the objects in the domain. An object is given a local name within a domain and an icon may also be associated with it. If a domain holds an interface reference to a component, the component is said to be a direct member of that domain and the domain is said to be its parent. A domain may be a member of another domain and is then said to be a subdomain. Policies which apply to a parent domain normally propagate to subdomains under it.

An object can be included in multiple domains (with possibly different local names in each domain) and so can have multiple parents. The domain hierarchy is not a tree but an arbitrary graph. An object's direct and indirect parents form an ancestor hierarchy and a domain's direct and indirect subdomains form a descendant hierarchy. The domain service supports operations to create and delete domains, include and remove objects, list domain members, query objects' parent sets and translate between path names and object references [3]. The domain service can be used to partition management responsibility by grouping those objects for which a manager is responsible or to control access to them by means of an authorisation policy [4] which specifies what configuration operations the manager can perform on the objects. Domains would be used to group interfaces which a manager is permitted to bind, files representing new component types which can be created or physical nodes to which components can be allocated.

A graphical user interface permits a human manager to locate objects they wish to manage by browsing through the domain hierarchy. Once located, composite components may be inspected and their internal configuration of interconnected component instances modified. New applications can be constructed by interactively creating component instances and binding their interfaces to those already registered in the domain service [5].

The Darwin configuration language was primarily aimed at defining the configuration of a composite component or service at design time, although it is able to specify some pre-programmed configuration changes. The work described in this paper shows how we made the full abstractions of the Darwin language available at run-time to permit interactive management of applications and services. A problem that had to be overcome was that the existing support for the Darwin configuration facilities stripped the design-time composite component abstractions down to the minimum required to create the required structure. The hierarchical composite components are essentially "flattened" to optimise run-time performance, but this implies that the knowledge of the system structure is typically lost. Our aim was to represent and augment this knowledge of system structure so as to support run-time interactive configuration management.

In Section 2, we use a simplified telephone exchange as a means of describing the Darwin language environment and configuration concepts. The overall environment and tools for interactive configuration management is presented in Section 3. Section 4 describes the support for evolving and maintaining configurations. Further details of the implementation are discussed in Section 5, along with plans for future work. Finally, we outline related work and conclusions.

2 Darwin Language Environment

2.1 Telephone Example

In this paper we will make use of a simulated telephone example. Telephones are connected to an exchange which sets up the calls between telephones (Figure 1). All components are implemented in software, and the telephones have graphical user interfaces, e.g. buttons to dial and text windows to talk to the connected party.

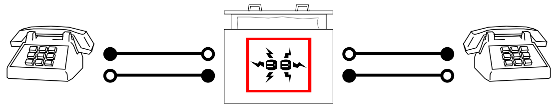

Figure 1. Telephones and exchange

An exchange consists of an exchange switch (`eswitch`) and a number of line units. The switch handles the connections between line units. The Darwin description of the exchange and a diagram showing an exchange instantiated with two line units are shown in Figure 2. A required interface (e.g. `s.out[0]`) is represented as a hollow circle and is usually bound to a provided interface represented by a filled circle (e.g. `lu[0].sin`). Note that the composite component `exchange` has two `in` and two `out` interfaces which are bound to interfaces of internal `lineunit` components. Binding a composite component interface to a subcomponent interface means the interface is really implemented by the subcomponent.

```
component exchange (int max) {
    provide in[max];
    require out[max];

    array lu[max]: lineunit;
    inst s: eswitch (max);

    forall i: 0 .. max-1 {
        inst lu[i]: lineunit  (i);

        bind
            lu[i].pin -- in[i];
            lu[i].pout -- out[i];
            lu[i].sout -- s.in[i];
            s.out[i] -- lu[i].sin;
            lu[i].ctl -- s.control;
    }
}
```

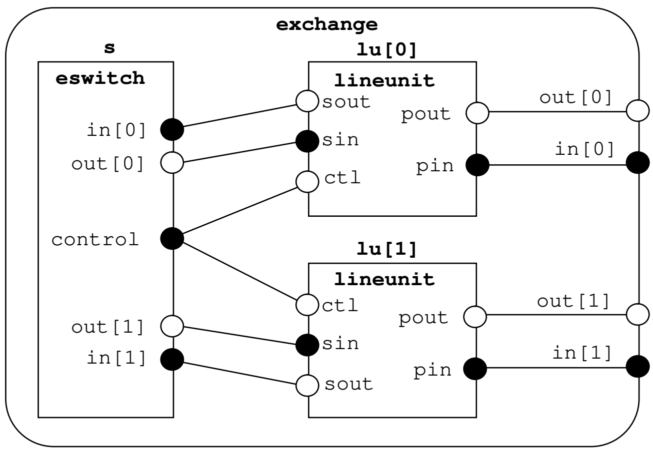

Figure 2. Darwin description of telephone exchange

The exchange component can now be bound to telephone sets to form a simple telephone system, as shown in Figure 3 in which the two connected telephones can make a call to each other via the single exchange. We do not show how multiple exchanges can be connected together. Note that the bindings within the exchange mean that a connected telephone only communicates with its line unit, and that a connected call is achieved by explicit forwarding of incoming data (in the switch and the line units) to simulate a real telephone system. The line unit signals off-hook and dialling from the phone and simulates ringing signals for the phone. If the callee is busy because the phone is off the hook, then the callee's line unit will respond with an engaged signal. Data received is forwarded via the switch to the other phone.

In Figure 3, the component `system` has been instantiated with exchange capacity of 3, but only with two telephones. The interfaces `in[2]` and `out[2]` represent unused capacity in the exchange for one telephone. We will show later how this is used when performing interactive configuration management.

```
component system (int phones, int max) {
  inst
    xchange: exchange(max);
  array ph[phones]: phone;

  forall i: 0 .. phones-1 {
    inst
      ph[i]: phone (i);
    bind
      xchange.out[i] -- ph[i].in;
      ph[i].out -- xchange.in[i];
 }
}
```

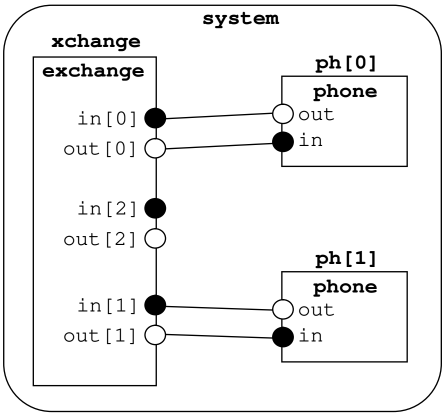

Figure 3. Simple telephone system

2.2 Configuration Domains

We represent the configured application as a part of the domain structure to enable a manager to locate the interfaces on which he wishes to invoke management operations. Every *configurable component instance* is represented by a *configuration domain.* This has similar operations to a management domain, except arbitrary interfaces cannot be included within a configuration domain and it is implemented as part of the configured application itself rather than within the domain service. A configuration domain contains the required and provided interfaces of the component it represents as well as the configuration domains representing the internal subcomponents. A primitive component has no subcomponents so its configuration domain only contains its interfaces. A manager can perform configuration operations on these interfaces or open internal composite components. Note that subcomponents can be distributed on different nodes.

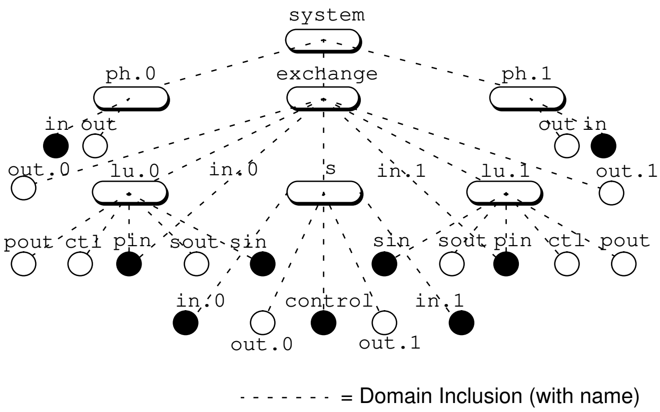

Figure 4. Configuration domain hierarchy

Figure 4 shows the configuration domain hierarchy resulting from instantiating the exchange depicted in Figure 3 (only two line units are shown). The names given to members must be unique, which is achieved by using the component instance name, not the type name. We separate array indices from the array name with a ".", so that lu.0 represents lu[0] from the Darwin specification, as the brackets [] have special meanings to most of our tools. The top-level component system is included into some existing domain, e.g. the creating user's User Personal Domain (UPD) [6], and can be given a name when instantiated. This name defaults to the type name, i.e. system. When naming interfaces for including them into configuration domains, we can use the same names as in the Darwin description, as Darwin enforces the necessary uniqueness properties.

2.3 Interfaces and Binding

Binding a required interface to a provided one makes the address of the provided interface known to the required one so that operations can be invoked on it. Many object-oriented systems only have explicit provided interfaces which are registered with a directory or name service. The required interface is an internal interface reference which is not visible outside the component. It is therefore not possible to perform "third-party" bindings from external managers.

We implement an explicit required interface as a communication object called a *bindable reference* on which a bind operation can be invoked by a local or remote manager or the run-time system performing configuration. When a component is created, its required interfaces are included into the configuration domain to make them known, e.g. for binding. When binding is performed, the required interface can perform binding-related actions which we call a binding protocol. Such a protocol can create a state-oriented connection to the provided interface [7], e.g. for reliable communications or according to some Quality-of-Service parameters, or it can initiate procedures for access control.

The details of the provided interface, its implementation and communication semantics are independent of its Darwin description. In our ANSAware implementation, a provided interface is mapped to an interface instance, i.e. serving an ANSA IDL, and the creating component must include a reference for the interface into the component's configuration domain where it can be looked up for later binding. When an operation is invoked on an unbound required interface, the component will block until the interface is bound to a provided interface, which ensures component synchronisation during configuration creation. A required interface can also be rebound later at run-time by a third party and the component will then start using the new binding. We have implemented a facility to allow the component to control when it will allow rebinding, which we describe later.

Both required and provided interfaces implement additional management operations to support remote retrieval of type information which can used to validate type compatibility of a binding.

The overall semantics and operations supported by the communication object which actually implement the required and provided interfaces can be extended by application programmers. For required interfaces, the only condition is that they inherit the relevant IDL-defined class for bindable references, which is part of the Darwin support library. Several different interaction mechanisms are available, including bi-directional ANSA remote procedure call communication, unidirectional messages and events. In the example described in this paper, uni-directional communication is used to simulate the signalling that takes place in a real telephone system.

We will now explain the three kinds of bindings that are permitted in the Darwin language, and how they are supported in our environment.

Require-to-Provide (R-P)

R-P binding is permitted in Darwin between interfaces at the same level, i.e. belonging to subcomponents in the same composite component. It represents the actual establishment of a communication link and is achieved by invoking the `bind` operation available for bindable references. Both one-to-one and many-to-one R-P bindings are permitted, as it is necessary to be able to bind multiple clients to the same provided interface of a server. In the `exchange` component of Figure 2, the lineunit `sout` interfaces are each bound to their own `in` interface at the switch (i.e. one-to-one), but the `ctl` interfaces are all bound to the single `control` interface. This is represented in Figure 5, which augments the bottom half of the diagram from Figure 4 with additional binding information.

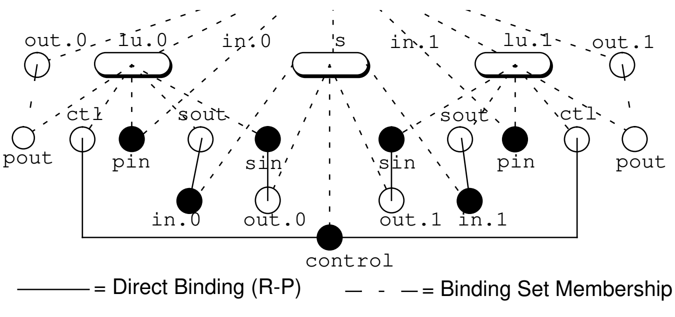

Figure 5. Representation of binding information

One-to-many (R-to-P) bindings are not directly expressible in Darwin – a required interface can only be bound to a single provided interface to avoid the difficulties in transparently enforcing consistency in group communications. To achieve the effect of multicast group communication, multiple required interfaces, each representing a multicast receiver, can be bound to a provided interface on a disseminator component which explicitly implements the required multicast service. The receivers register with this service as part of their binding protocol and the multicast server can use a underlying multicast protocol or implement the service as multiple one-to-one messages, as appropriate. This interaction pattern has been impemented in a set of event communication objects.

Provide-to-Provide (P-P)

P-P binding is a one-to-one mapping from a subcomponent's provided interface to a provided interface of the composite component (e.g. `lu[i].pin -- in[i]`). Binding to the composite's interface `in[i]` is equivalent to binding to the internal interface it maps onto. This is achieved by including the provided interface from the subcomponent's domain into the enclosing configuration domain and giving it a local name corresponding to that of the composite's provided interface. For example, in Figure 4 the `lu.0/pin` and `lu.1/pin` interfaces are included into the xchange domain with the names `in.0` and `in.1` respectively. The same interface reference is included in both the subcomponent and the enclosing component domains. The two entries have different names, but they are the same interface references, so binding a required interface to the provided interface in either domain is equivalent.

Require-to-Require (R-R)

The R-R binding represents making an interface from the internal level available at the composite level, similarly to P-P binding (e.g. `lu[i].pout -- out[i]` in Figure 2). However, a composite-level required interface can be bound to *several* internal required interfaces. This permits is all the contained interfaces to be bound when the one at the composite level is bound. The composite’s internal structure is opaque to users who only see *one* required interface and bind it in the normal way[1].

This is achieved by creating an intermediate required interface which stores (references to) the required interfaces it has been bound to at the lower level in a *binding set.* When this intermediate required interface is bound to a provided interface, it will also bind the required interfaces in its binding set to this provided interface.

The exchange has required interfaces `s/out.0` and `s/out.1` which need to be bound to telephones, but they represent the underlying required interfaces `lu.0/pout` and `lu.1/pout`. In our example, the R-R binding is one-to-one, but we still create the intermediate binding sets, in order to support R-to-R binding performed interactively at run-time. When `out.0` is bound to a provided interface (of a telephone), it forwards this binding to its binding set members, i.e. `lu.0/pout`. The same is true if and when `out.0` is rebound at run-time.

3 Interactive Configuration Management

3.1 Viewing a Configuration

A prerequisite for interactive configuration management is that the human manager can view an existing system

1. This is important when certain users only have rights to use/bind to a component’s interfaces, but no access to its internal structure. This permission is typically only given to the owner/creator of an application and certain managers.

configuration. Our architecture represents the structure of the application as a sub-tree in the system-wide domain structure. The mapping of component instances to configuration domain hierarchies allow the generic domain service tools, e.g. command shells and domain browser, to be used to get a basic view of the configuration. In particular, the domain browser is used to locate components and navigate within a component hierarchy. Figure 6 shows a domain browser view of the `system` component. The middle display shows the direct members of the domain, whereas the left and right treemaps show its ancestor and descendant domain hierarchies, respectively.

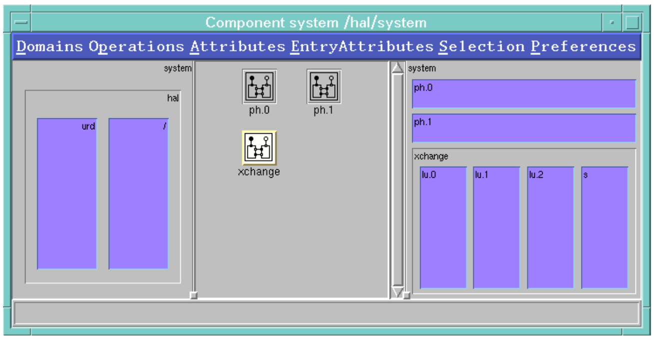

Figure 6. Domain browser view

Our graphical configuration tool, called *beagle*, displays the structure of a composite component representation held in a configuration domain e.g. double-clicking on the icon labelled `xchange` in Figure 6 will open the beagle configuration view shown in Figure 7. When retrieving the configuration structure of a component, beagle checks all the members of its configuration domain. These fall into one of three categories, a required interface, a provided interface or a subcomponent. Below we describe the steps required for interpreting the structure of the telephone `exchange` component instance shown in Figure 7.

Required interfaces can be queried for the members of their binding set (which should be empty for primitive components) to deduce R-R bindings from subcomponents. The exchange contains `out.0` and `out.1` which each have one entry in their binding sets (`lu.0/pout` and `lu.1/pout` respectively). These required interfaces can also be queried for what provided interface they are bound to (if any), corresponding to R-P binding. We only show the R-P bindings between subcomponents but not those between the composite level interfaces and other external interfaces.

A provided interface was either created by a primitive component, or comes about as a P-P binding from a subcomponent. The exchange contains provided interfaces `in.0` and `in.1`, which were created initially as `lu.0/pin` and `lu.1/pin`, but beagle needs to examine the subcomponents to deduce a P-P binding.

For subcomponents, beagle looks up their interfaces within their component domain. If any of their required interfaces appear in the binding set of a required interface of the composite component, it can deduce a R-R binding. This is the case inside `lu.0`, where `pout` is a member of `out.0`'s binding set. Also, beagle queries `ctl` and `sout`, which are (R-P) bound to interfaces `control` and `in.0` found inside the switch `s`. It also works out the P-P binding from `lu.0/pin` to `in.0`. This process is repeated for all subcomponents and bindings. Beagle uses the deduced configuration structure to create the graphical view shown in Figure 7. The larger circles which are not attached to any boxes represent the interfaces of the composite component. The interfaces attached to boxes are automatically laid out to minimise the number of intersecting lines [8].

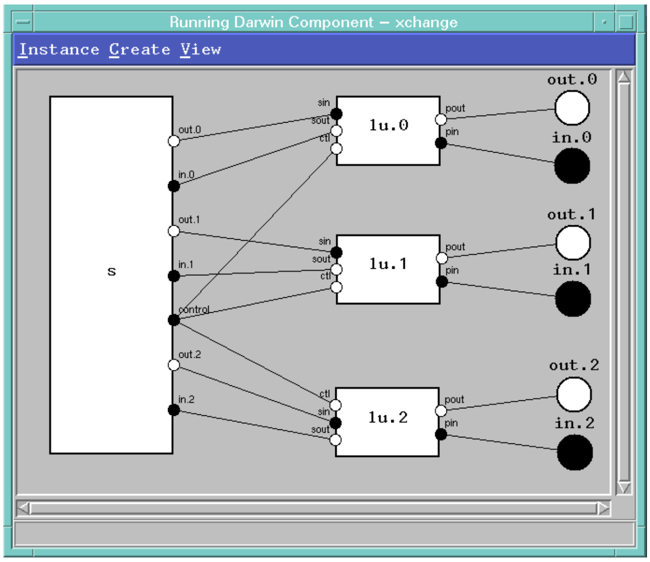

Figure 7. Graphical configuration view

3.2 Component Creation

It is possible to create new components interactively in two ways:

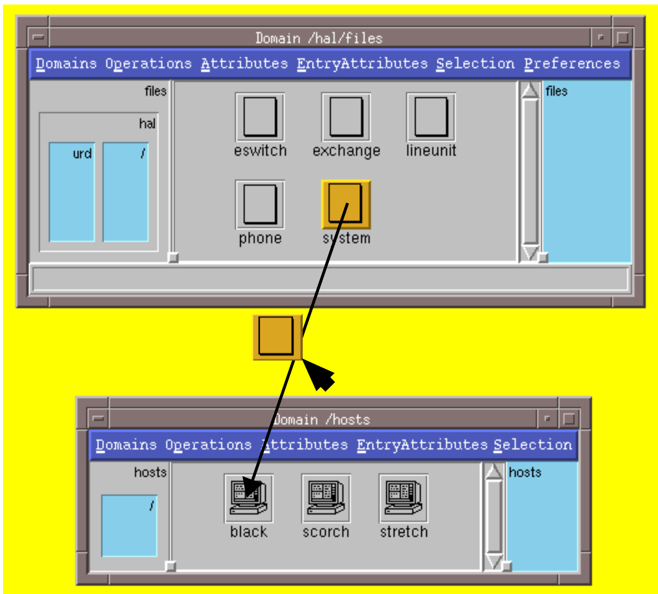

Figure 8. Drag-and-drop component execution

Operating System Creation

A component can be started using operating system facilities, and then bound into an existing application. In practice, this means executing or starting a new process. This method will allow the most flexible control over such issues as component location and allocated operating system resources.

Our user interface for executing a new component can use drag-and-drop to allocate a component to a physical node, as shown in Figure 8. The icon representing the executable file for the system component (see Figure 3) is dragged onto the machine icon black which creates the initial telephone system. A benefit is of this mechanism is that it can be used to instantiate components (composite or primitive) which are not defined or implemented in any previously running application.

Application-Supported Creation

An existing application might be asked to create a new instance of a predefined component type. All our composite components have the ability to instantiate a component type defined within the overall application. This is shown in Figure 9, where a telephone instance called `newphone` is about to be created.

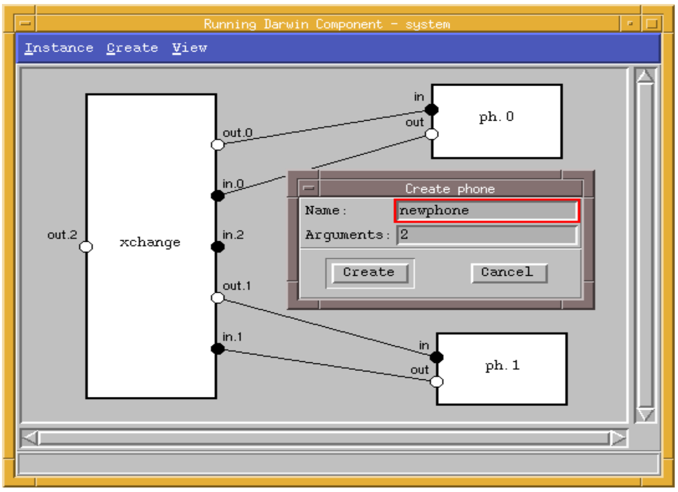

Figure 9. Application-supported component creation

An application will only be able to instantiate component types which are defined in the application (e.g. phone, exchange). Creating a component instance this way allows the creating (enclosing) component to map the new component instance to a physical node depending on (co)location criteria of the overall application.

Destroying components is essentially preformed directly on the component itself, either via a (remote) destroy operation or via operating system facilities. Bindings to and from the component that is to be destroyed must be managed by the configuration management tool before destruction. Component destruction should entail destruction of enclosing component instances, and components should destroy their component domains upon destruction. Furthermore, the domain system will (attempt to) remove invalid interface references from the domain structure.

3.3 Interactive Binding

The main form of binding change that is useful at run-time is modifying a R-P binding. A required interface can be unbound or rebound. Unbinding causes the old binding to be discarded, and the application will block until it is bound again. Rebinding does not require explicit unbinding first, and the application uses the new binding without blocking. Direct rebinding is more common at run-time, although unbinding can be useful for freezing applications during certain types of configuration change.

R-P binding is commonly used after creating a new component at run-time. In Figure 10, the `newphone` has been created in the system component, and it is being bound to the exchange. Binding is achieved within a configuration window by dragging a line between the interfaces, or it can be achieved between different windows (e.g. domain browser views) using drag-and-drop.

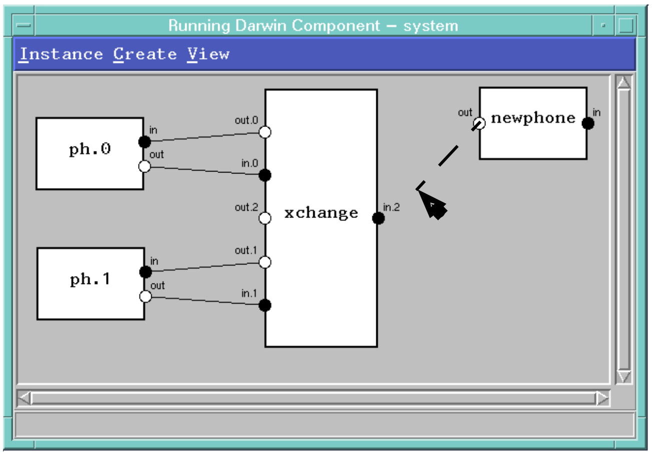

Figure 10. Binding of new subcomponent

It is necessary for a component to control when its binding may be changed as it forms an integral part of the component state, e.g. a transaction may span multiple invocations on an interface and so the binding can only be changed when the transaction has completed. A component can set its required interface to be safe for rebinding– so it can be changed by a configuration manager. The interface is marked as critical to indicate that it cannot be changed. This mechanism allows the configuration manager to perform the rebinding, but the new binding will only comes into use when the component marks the required interface safe. When this happens, i.e. when a new binding comes into effect at component level, some time later, the required interface can emit an event to this effect, e.g. to the configuration manager.

In the telephone example, a telephone (`ph.0`) marks its binding to the exchange (from `ph.0/out` to `xchange/in.0`) as safe when a call is completed, i.e. when the caller hangs up the receiver. Similarly, the exchange marks its binding to the telephone as safe at the same time. This allows the telephone to be rebound to a different pair of exchange interfaces, which means that it answers a different number. This corresponds to moving house and plugging one’s old phone into a new socket. Any attempt to change a binding while a call is in progress will be deferred until the call is completed.

It might also be useful to have more powerful rebinding mechanism, which does *not* depend on application synchronisation, i.e. a forced rebinding. This could be useful when applications have got into a inconsistent state, e.g. some bindings to provided interfaces which no longer exist. This is rather a powerful functionality and its use must be strictly controlled.

The other two forms of binding, R-R and P-P, are not as useful at run-time as R-to-P bindings. A scenario for creating a P-P binding is when an internal provided interface would be useful as a provided interface of the composite component - for example, it might be desirable to make the control interface of the switch visible at the interface of the overall exchange which can be achieved by including it into the configuration domain of the composite component.

R-R binding might be useful when a new subcomponent is created inside a composite component, and the new component has a required interface which should be bindable at the enclosing level. If the subcomponent's required interface is included into the binding set of an existing required interface of the composite component, any existing binding will be propagated down to the new binding set entry.

4 Maintaining a Configuration

4.1 Actual versus Intended Configuration State

We have considered how an actual configuration can be retrieved at run-time, and how it can be modified. However, the structure that actually exists could change in an unwanted way as a result of breaks in communication links or in machine failures. We therefore distinguish between the *intended* and *actual* state of a configuration. So far we have assumed that the two are identical. Instead, by having a persistent representation of the intended configuration, it is possible to maintain the actual configuration in the presence of failure.

At program start-up time, the Darwin environment should not only create the initial configuration structure, but also record what this intended structure is. When the structure is modified as a result of changes made (e.g. by the configuration tools), this intended structure should be modified accordingly. This information can be used to detect failures, e.g. by automated monitoring or from within the configuration tool. When a discrepancy between the intended and actual configuration is detected, the configuration tool could display a warning message, or post an event notification. Automated managers could furthermore be used to attempt to recreate the intended structure without human intervention being required.

4.2 Generating Code for Configuration

The configuration tool can be used to modify component instances to an extent where they no longer have any resemblance to the Darwin description used at design-time to generate the component. The configuration can generate Darwin code which effectively defines a new Darwin component template, so that the same component can be instantiated at a later time.

Darwin has a number of general programming language constructs which allow structured configuration programming, e.g. loops and conditionals. When these are used, it is not possible in general to predict the exact outcome of instantiating the component, as it will depend on instantiation parameters or indeterministic arguments, e.g. current time. It is not possible, in general, to recreate the Darwin code relating to these loops and conditionals.

```
component modified_system {
    array ph[2] : phone;
    inst    xchange: exchange(3);
            newphone : phone (2)
            ph[0] : phone(0);
            ph[1] : phone(1);
    bind    xchange.out[0] -- ph[0].in;
            ph[0].out -- xchange.in[0];
            xchange.out[1] -- ph[1].in;
            ph[1].out -- xchange.in[1];
            xchange.out[2] -- newphone.in;
            newphone.out -- xchange.in[2];
}
```

Figure 11. Generated Darwin description

However, it is simple to create a Darwin description that will recreate the intended configuration using a sequence of inst and bind statements to achieve the intended effect. We have designed such a facility, which can also cope with arrays of interfaces and component instances. If applied to the modified telephone system component shown in Figure 10 with the newphone bound to the exchange, the resulting Darwin code for the component could be as shown in Figure 11.

5 Implementation & Future Work

The basic Darwin environment described in this document has been implemented in a prototype architecture based on ANSAware. Anything can be a component to Darwin, as long as the Darwin environment knows how to instantiate a component template. The component mapping chosen was that primitive component templates were expected to be executable programs. Component creation is achieved by running the executable representing the component template, supplying it with any instantiation arguments. The Darwin compiler itself generates code to implement the composite components and emits support code for primitive components. This mapping had the benefit of being very flexible, since primitive components could be implemented in different languages (e.g. C or Tcl).

We have also implemented the tools and support for most of the interactive configuration management described in this document. The mechanisms for viewing and modifying the configuration structure at run-time is working. The graphical management interfaces for configuration and domain browsing are implemented as seperate, but integrated tools in Tcl/Tk [9]. For this purpose, we have extended the basic Tcl interpreter with commands to enable ANSA client access.

The support for maintaining the *intended* configuration in stable store, along with a facility for generating Darwin source code from such a description is being developed.

Future work will concentrate on the use of the Orbix [10] implementation of CORBA as an environment for domain-oriented configuration management. Certain features of the CORBA standard will make the implementation more flexible, e.g. the Dynamic Invocation Interface (DII) will allow more complete client access to managed interfaces.

We will probably change the mapping of Darwin components to be based on C++ source code: A Darwin component template is mapped into a C++ class, which can be instantiated at run-time without needing to know about its implementation. A feature of our existing mapping is that a primitive component instance is mapped to a running Unix process. For large-scale systems, this a natural and useful component mapping, as an operating system process is the typical unit of failure. However, it is convenient to be able to configure components as light-weight threads, as has been done in the Regis [1] system. In our CORBA implementation, we will support primitive components as light-weight threads, and single, multi-threaded processes as composite components. We are also planning further integration with general domain-oriented management, e.g. access control management for Darwin-configured applications.

6 Conclusions and Related Work

In this paper we have described our implementation of an interactive configuration management environment, with both notational and graphical support for component creation and establishing communication bindings. The graphical tools enable a manager to visualise and modify the configuration structure of an object-based distributed service or application which can be used as the basis for monitoring and changing the system. However this type of interactive management is very powerful and needs to be strictly controlled. Access control is needed as to which components a manager can reconfigure and what configuration operations can be performed on them.

Our management tools all share the domain browser as the means for domain hierarchy display, navigation and manipulation. Moreover, our configuration tool can interwork with other management tools, e.g. a management policy editor [4], which is part of the management infrastructure.

Our approach in this paper is the result of integrating work our colleagues and we have done over a number of years in the areas of distributed systems management in general, and configuration management in particular. The Darwin notation for describing the structure of distributed systems evolved in the REX project from experiences with Conic. Also, our graphical tool support for viewing and manipulating configurations is based on experiences made from these projects [8].

There are a number of systems which support a configuration notation [11] [12] and some support the concept of composition [13]. However, none integrate the concept of graphical interactive management with a notation for defining composite objects nor do they permit transformation between graphical and textual configuration notations.

Acknowledgments

This work was supported by the SysMan ESPRIT project (7026). We acknowledge the contribution of our colleagues at Imperial College for their valuable comments and suggestions, in particular Nat Pryce and Kevin Twidle for help with implementation work.

References

[1] J. Magee, N. Dulay & J. Kramer: *Regis: A Constructive Development Environment for Distributed Programs*. IEE/IOP/BCS Distributed Systems Engineering Journal, 1(5), 304-312, September 1994

[2] M. Sloman: *Policy Driven Management for Distributed Systems*, Journal of Network and Systems Management, Vol.2, No. 4, 1994, Plenum Press pp 333–360, 1994

[3] K. Becker, U. Raabe, M. Sloman & K. Twidle (eds.) *Domain and Policy Service Specification*. IDSM Deliverable D6, SysMan Deliverable MA2V2, 1993

[4] D. Marriott & M. Sloman: *Management Policy Service for Distributed Systems*, In Proceedings of IEEE 3rd International Conference on Services in Distributed and Networked Environments (SDNE 96), pp. 2-9, June 1996

[5] S. Crane, N. Dulay, H. Fosså, J. Kramer, J. Magee, M. Sloman & K. Twidle: *Configuration Management for Distributed Software Services*. In Proceedings of IFIP/IEEE International Symposium on Integrated Network Management, Chapman & Hall, May 1995

[6] J. Moffett & M. Sloman: *User and Mechanism Views of Distributed System Management*, IEE/IOP/BCS Distributed Systems Engineering, 1:1, 37–47, 1994

[7] N. Pryce & S. Crane: *A Uniform Approach to Configuration and Communication in Distributed Systems*. In Proceedings of 3rd International Conference on Configurable Distributed Systems, IEEE Press, May 1996

[8] J. Kramer, J. Magee, K. Ng & M. Sloman: *The System Architect's Assistant for Design and Construction of Distributed Systems*. In Proceedings of 4th IEEE Workshop on Future Trends of Distributed Computing Systems, 284-290, 1993

[9] J. Ousterhout: *Tcl and the Tk Toolkit*, Addison-Wesley, ISBN 0-201-633337-X, 1994

[10] Iona Technologies: *Orbix - A Technical Overview*. The O'Reilly Institute, Westland Row, Dublin 2, Ireland, 1992

[11] B. Agnew, C. Hofmeister & J. Purtilo: *Planning for Change*, IEE/IOP/BCS Distributed Systems Engineering Journal, 1(5), 313–322, September 1994

[12] M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner & R. Lichota: *Durra: a structure description language for developing distributed applications*, IEE Software Engineering Journal, vol. 8 pp. 83–94, 1993

[13] T. Coatta & G. Neufeld: *Distributed Configuration Management using Composite Objects and Constraints*, IEE/IOP/ BCS Distributed Systems Engineering Journal, 1(5), 294–303, September 1994