IEEE Proc. First International Enterprise Distributed Object Computing Workshop (EDOC'97), Gold Coast, Queensland, Australia, Oct. 1997, pp 118-128

Interactive Configuration Management for Distributed Object Systems

Halldor Fosså
half@tibco.com
TIBCO
Gracechurch House
55 Gracechurch Street
GB-London EC3V 0JA

Morris Sloman
mss@doc.ic.ac.uk
Department of Computing
Imperial College
180 Queen's Gate
GB-London SW7 2BZ

Abstract

This paper describes an environment for interactive configuration management of the software components comprising a distributed enterprise application. The environment permits one or more managers to view and modify the structure of components in terms of component instances, their allocation to hardware nodes and the bindings between their interfaces. Our graphical management is based upon the Darwin configuration language which can be used to create the initial system. It supports hierarchical composition of CORBA components to form a composite distributed application or service. When this structure has been modified interactively, a persistent specification of the configuration can be saved to backing store. This can be used to determine unreachable or failed components and, if necessary, to recreate them. The configuration management environment is integrated into an overall domain-oriented platform for enterprise management. The paper illustrates the use of the management system in terms of a simple banking example, and outlines our implementation based on a CORBA platform.

Keywords: *Hierarchical components, binding, graphical life-cycle management, persistent configurations, CORBA*

1 Introduction

Distributed enterprise applications will typically consist of many software components on various computers connected to the network. The components may themselves be fairly complex distributed services or subsystems. Hierarchical composition is a powerful mechanism for specifying these components in terms of object or component instances, allocation to hardware and binding of interfaces. The services offered by a component can be accessed via its interfaces, as specified by an interface definition language, but clients of the component need not know its internal structure. It should also be possible to treat a component specification as a class or template from which multiple instances can be created to allow reuse.

Managers of an application will need to deal with operational changes such as replacement of components which have failed, introducing new instances of existing classes to cater for system expansion as well as new component classes to cater for new functionality. For many large scale enterprise applications, such as telecommunications, banking or process control, these changes should be performed dynamically without shutting down the complete system. In order to manage these configuration changes, a human manager must be able to view the current state of a distributed component and then interactively modify its structure.

We have previously described *configuration management* [1] as the construction of distributed applications from components with well-defined communications interfaces using the Darwin configuration language [2], which is based on *components* which can be both clients and servers. Interfaces can be bound by third-party (e.g. a manager) as well as first-party (i.e. the component itself). In order to permit third-party binding, components acting as clients must explicitly declare the interfaces they *require* in order to use an external service as well as the interfaces which they *provide* as servers. *Primitive* components perform application-specific functionality implemented in a programming language (e.g. C++). *Composite* components contain instances of primitive or composite components, specify bindings between required and provided interfaces of internal component instances and map their interfaces to those made visible outside the component.

The CORBA standard [3] from the OMG includes a Specification of Common Object Services [4] which addresses some of these issues in the *Life Cycle Service.* This allows graphs of related objects to be created and maintained. Our approach to component creation is similar, but we advocate third-party composition and binding to manage system structure explicitly. We have used the Darwin language with several different distribution platforms [5], and this paper describes the configuration of CORBA components. While CORBA IDL defines the interfaces of components, Darwin is used orthogonally to specify the structure and bindings of an application.

The Darwin configuration language mainly serves the purpose of constructing the initial configuration, although it supports some programmed, incremental configuration changes. The work described in this paper shows how we made the full abstractions of the language available to permit interactive run-time reconfiguration of applications via the ICON tools [6]. This allows a manager to monitor and modify the structure and interconnecting bindings of a composite component and to invoke configuration operations provided at a component interface. The ICON environment detects and indicates component failures, and permits a saved configuration to be restored from persistent storage after a failure. However, the running configuration state is determined from querying the running components and does not rely on the Darwin configuration program.

The configuration concepts are integrated in a domain-based management environment [7], which allows managed interfaces to be grouped and named for interactive management. Typical domains could be used to group the set of hosts on a particular LAN or the resources belonging to a department but domains are untyped. Managed objects (including other domains) only become members of domains when they are explicitly included into their parent domain with a unique local name. This allows path names to be used to name managed objects, and such names can be translated across a distributed domain structure, which forms an arbitrary directed graph.

This paper uses a simple personal banking application, described in section 2, to illustrate configuration management. The management user interface is presented in section 3 followed by the main features for interactive configuration evolution in section 4. Interactive configuration maintenance in the presence of failure is described in section 5. An overview of our implementation architecture is given in section 6, and related work is reviewed in section 7. Section 8 concludes the paper with a discussion of the main contributions and future directions of our research.

2 Configuration representation

2.1 Bank example

The bank example application consists of a number of branches which hold accounts for customers. Customers enter into a branch to open the account initially and make deposits. They can also use an ATM (Automated Teller Machine), connected to some of the branches, to withdraw money from their account. The relevant concepts are modelled in Darwin as software components which we configure to form a network of branches and accounts. Clients instantiate customer components which bind to a branch and request that an account be opened in their name.The ATMs are implemented as user interfaces which communicate with a branch to validate a customer's account details and debit the account balance as a result of transactions.

The specifications of the bank and branch composite components are shown in Figure 1, using both the Darwin textual and graphical notation. A component *class* is shown as a rounded box with the class name on the inside, whereas a subcomponent *instance* is a rectangle with an instance name on the outside. A required interface is denoted by an empty circle and has to be bound to a provided interface de-

```
component bank (int branches=2, int atms=1) {
  array office[branches];

  forall i:0..branches {
    inst office[i] : branch(i, (i<atms)) @ i;
  }
}

component branch (int sortcode, int hasATM) {
  inst desk: counter(sortcode);

  provide open;

  bind open -- desk.open;

  when (hasATM) {
    inst till: atm;

    bind
      till.branch -- desk.socket;
  }
}
```

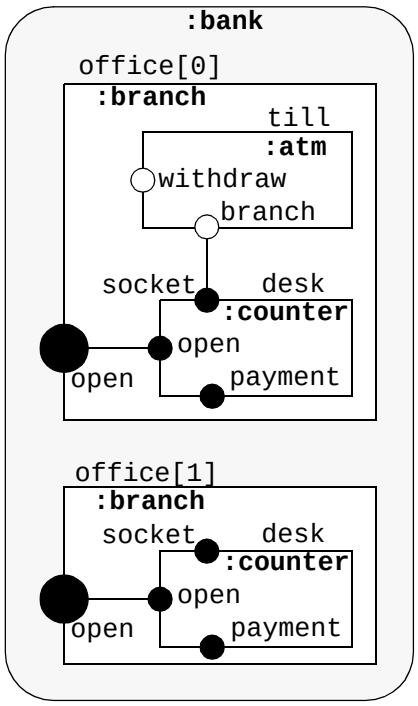

Figure 1. Personal banking system

```
component account (int number) {
  provide deposit <credit>;
  provide withdraw <debit>;
  provide current <balance>;
}

component person (char* identity) {
  require open <new_account>;
}

component customer (char* id, char* branch) {
  import open@branch;
  inst me : person(id);
  bind me.open -- open;
}
```

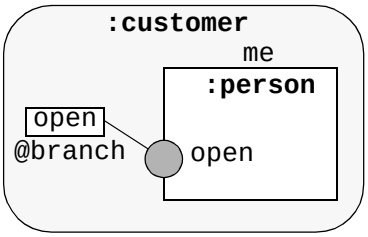

Figure 2. Darwin components for customers and accounts

noted by a black, filled circle. It is this explicit representation of a component's required interface which underlies the ability to perform third-party binding in Darwin or by an external configuration manager. A branch can optionally have an ATM which is bound to a socket at the desk, as is the case for `office[0]`. We will show later how an ATM is created and bound as part of interactive configuration evolution. Interfaces can also be bound hierarchically (e.g. the `open` interface of the branch) to expose an otherwise internal interface of a subcomponent (e.g. `desk.open`).

The bank system is very dynamic, and accounts are only created by a branch when customers invoke the relevant `counter` interface. Customers appear spontaneously, instantiate components and perform bindings. Figure 2 shows how customers bind to their branch by a client-initiated or first-party **`import`** which can be used to create a new account.

A composite configuration specification such as the `bank` component in Figure 1 is translated by a Darwin compiler into an executable program which instantiates the subcomponents and performs the necessary bindings. Both composite components implemented by Darwin and primitive components implemented by the programmer can subsequently be instantiated and bound together by the interactive configuration manager.

2.2 Darwin representation

We now describe how configurations are represented in the interactive domain-based management environment. Central to this representation is the ability for managers to view the initial or evolved configuration state without any previous knowledge or reference to the Darwin code that was used to elaborate the initial configuration structure.

An instance of a component class is represented as a configuration domain which is a managed object with a domain interface, e.g. for including and removing domain members, but can be queried for information on internal component instances and bindings to support interactive configuration management. It is a component of the application rather than the domain service, although it can be represented as part of the domain hierarchy.

Configuration domains are used to name the configuration elements by reusing the identifiers of the Darwin specification. Subcomponent instances are named using their

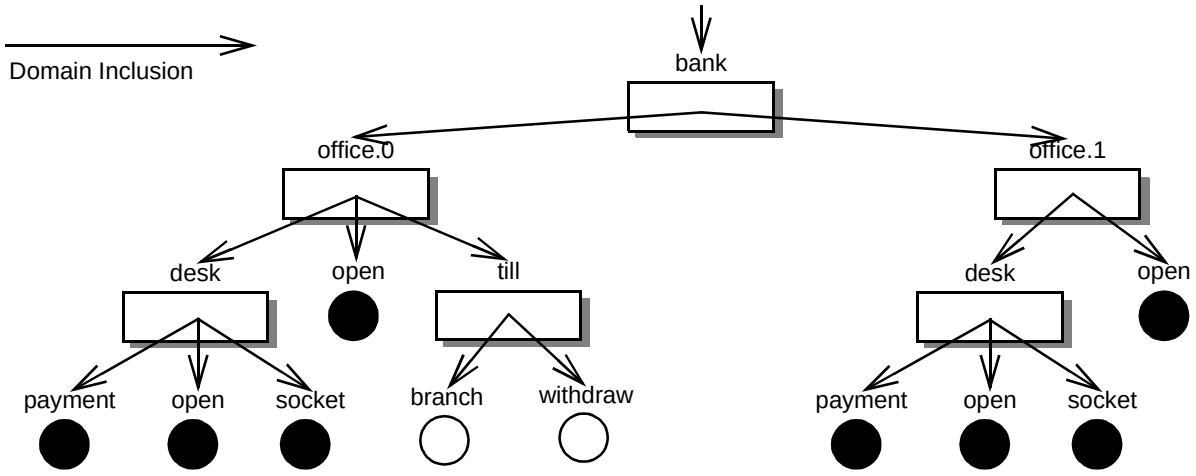

Figure 3. Configuration Domain Hierarchy

instance names, although we remove square brackets from array indices (e.g. `office[0]` is mapped to office.0), since square brackets have special meaning in our management scripting environment. This is shown in Figure 3 which depicts the configuration representation which results when the `bank` component from Figure 1 is instantiated.

The configuration domain of the top-level component class `bank` instantiated by the configuration program is included into a previously existing management or component domain. It is given a user-chosen, unique name which can default to the class name if there is only one such instance in the parent domain. Figure 3 also shows that instances of both primitive and composite component classes are represented by a configuration domain, although primitive components do not themselves contain subcomponent domains.

2.3 Component interfaces

CORBA IDL can be used to define interfaces for component classes in Darwin. These interfaces are represented as members of the instance's configuration domain. Figure 3 shows how this is achieved by using the interface names from the Darwin specification as these are already unique names within a component. Primitive components create provided interfaces which are arbitrary communication endpoints. The only prerequisite is that provided interfaces can be globally referenced, as such references are passed to the require-side for binding (see §2.4).

Required interfaces must be made explicit to permit third-party binding, rather than just object references as in CORBA, so they are represented by predefined managed objects called *bindable requirements*. They are instances of a library-implemented interface which stores a reference to a provided interface on behalf of the owning client, and they are created and located with their enclosing component. They are remotely invokable for managers to perform binding. They also allow an external rebinding operation to synchronise with the internal code of primitive client component so that the binding is actually performed after the client has explicitly marked the bindable requirement as safe for unbinding or rebinding. This will be done on completion of the interactions forming part of an application dependent sequence when the client is effectively in a quiescent state. The bindable requirement can also be queried by a remote manager to determine its current state.

A required interface of a composite component can be hierarchically mapped onto multiple required interfaces of internal conponents as explained below, although this is not used in our bank example - see [5] and [6] for examples.

2.4 Interface binding

R-to-P (Require-to-Provide) binding between clients and servers supplies a requirement with a reference to a provided interface. This represents an internal binding within a composite component. The Darwin-implemented components look up the interface references of provided interfaces in their owning component domain and pass it to the bindable requirement when invoking its `bind` operation. As soon as such a reference is obtained by the bindable requirement, it can engage in a binding protocol to establish a connection or session directly to the referenced communication endpoint, and can start sending data or invoking operations.

Such R-to-P binding is shown between branch and socket in Figure 4, which depicts binding representations in the lower half of the hierarchy of Figure 3. It shows how the `branch` interface of the ATM `till` in office.0 is bound to the socket interface of the counter desk. If a client tries to use an unbound interface it will be blocked, which permits synchronisation during initial configuration elaboration or interactive unbinding.

Hierarchical P-to-P (Provide-to-Provide) binding maps a composite component's interface to that of an internal component. It is achieved by including the provision of the inner component into the enclosing component domain. The provided interface thus is a member of two different parent domains, potentially with different names i.e. the binding is indicated by the references being equal so entails very little overhead.

Hierarchical R-to-R (Require-to-Require) binding enables multiple internal requirements to be transparently bound by the same externally provided interface. It is achieved by creating a hierarchical requirement at the outer level, which maintains a binding set of internal require-

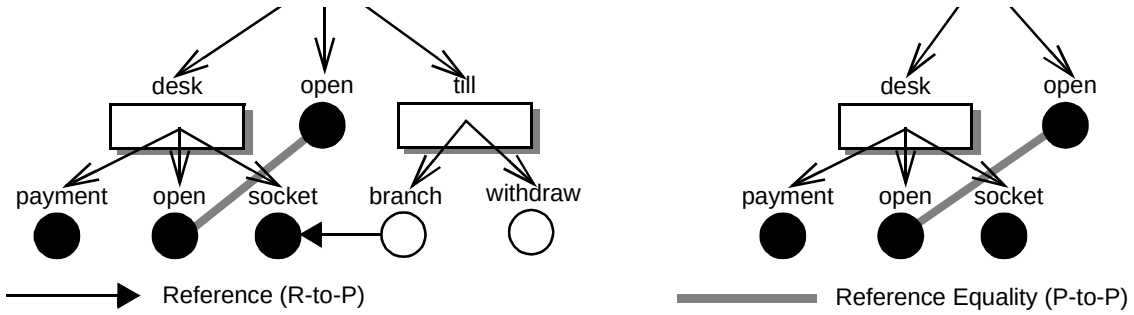

Figure 4. Binding representation

ments onto which it maps. The members of the binding set can be defined in an initial Darwin specification as well as changed interactively.

3 Management user interface

Our model of interactive configuration management allows external human managers to interact and modify running configured systems through a management user interface tool set. The configuration state is represented in the domain management environment, and ICON supports various tools in order to depict and manage the components and objects in this structure.

3.1 Domain browser

The backbone of the management user interface is a domain browser. It allows a domain structure to be displayed as trees. A domain browser window is shown in Figure 5, which has an icon canvas on the left with direct domain members, and a descendant tree of indirect subcomponents.

The tree is depicted as a *treemap* [8], which enables the structure to shrink-to-fit to the size available in the window without any scrolling. The structure in Figure 5 also shows how each component subtree can be expanded and collapsed at will by the user, e.g. the bank branches shown are expanded except office.3.

The domain browser has full support for manipulating domain membership, with drag-and-drop and double-clicking for invoking operations. All operations are scripted and fully configurable in the management environment, and user-defined operations can have typed arguments associated with them which the browser uses to create dialogue boxes for generic invocation support. In ICON we can use this to configure objects and components individually, as well as perform structural configuration such as object allocation (via drag-and-drop) and binding.

The domain browser view of Figure 5 shows a bank composite component with four branches. The configuration state shown has developed over the life-time of the system. Two customer components (CustomerHal and CustomerSteve) have appeared and bound themselves to branch office.1. Although we cannot directly see the bindings, we can see the accounts they have opened (i.e. jsc and hal) at the branch office.1. The state depicted in Figure 5 shows how configuration changes at run-time have been caused from within the primitive components themselves. Our configuration representation and user interface support can detect and display changes to the configuration state independently of who or what initiated them.

The domain browser is a generic user interface for domain-based management, and does not have any specialised or built-in support for our view of interactive configuration. In particular, despite being able to query and change bindings as part of generic invocation support, the domain browser does not offer graphical or notational support for Darwin-style bindings and configurations, as this is supported in the Beagle graphical configuration manager.

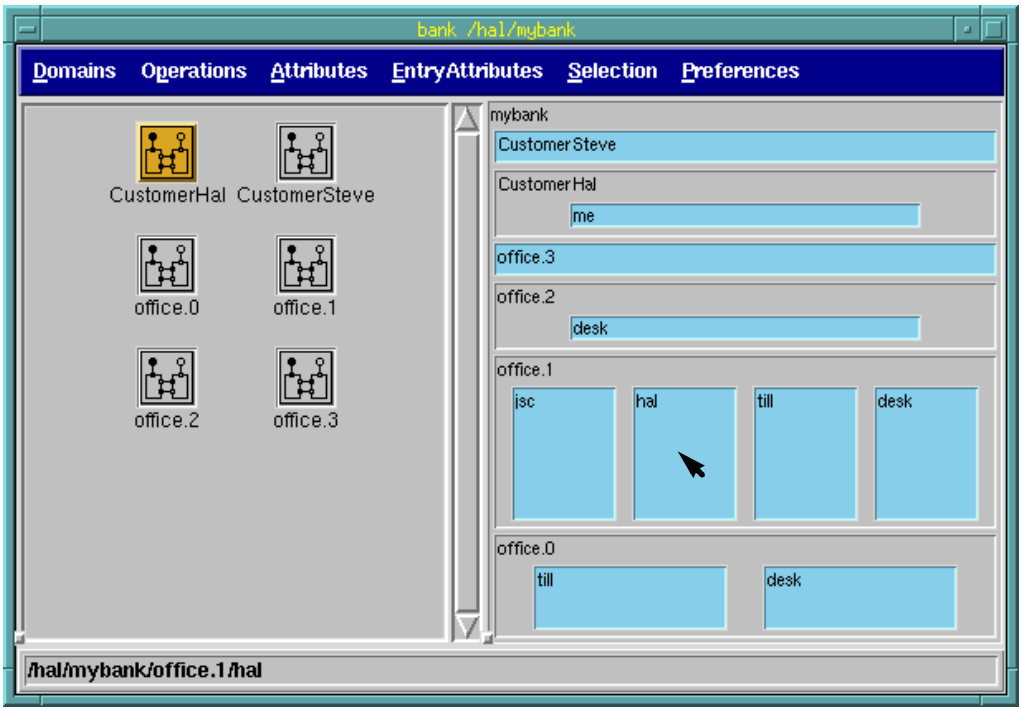

Figure 5. Domain browser view

3.2 Beagle configuration manager

The interactive, graphical configuration manager called Beagle is specifically designed to support all aspects of our configuration model. It retrieves a configuration domain in the management environment, and derives the subcomponents, interfaces and inter-component bindings. This enables Beagle to depict graphically the configuration state of a running application.

The configuration state of the branch office.0 is shown in Figure 6. The configuration view uses the graphical notation used for Darwin, and bindings are depicted as lines. A component-level interface is depicted as a larger circle, as it cannot be drawn across the edge of the window. The bound requirements are filled in grey (actually green on a colour screen), to indicate that they will not block their owning component threads when invoked.

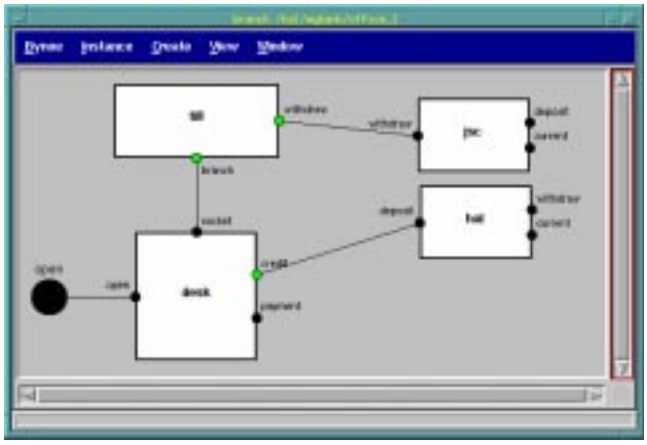

Figure 6. Beagle configuration view

The configuration view in Figure 6 shows two account transactions in progress: One customer is withdrawing money from the account jsc using the ATM, which has bound its requirement withdraw to the relevant account interface. At the counter desk, a customer is making a deposit into the account hal, for which the counter has dynamically declared a requirement credit (not present in the original Darwin code) and bound this to the deposit interface of the account component. All configuration elements i.e. components, interfaces and bindings, are monitored and visualised in Beagle by querying the actual components.

Our management user interfaces are integrated so that when the browser is used to navigate to a configuration domain, it can inform Beagle, which responds by opening the component's configuration view. The domain browser and Beagle also interwork in their use of the selection for copying and pasting. Drag-and-drop is possible across the two management applications.

4 Configuration evolution

Interactive configuration evolution allows the manager to perform deliberate and external modification of a system to change its capacity or functionality during the life-time of long-running systems such as the bank application.

4.1 Component creation

Component creation is supported in ICON in several ways: We have already seen how components can be instantiated as a result of the running components invoking each other to make use of application-provided creation services, i.e. for opening new accounts. In our example, the branch counter creates new account components in response to invocations by customers. Such application-specific creation services could also be invoked by the interactive manager using generic operation support from within the domain browser.

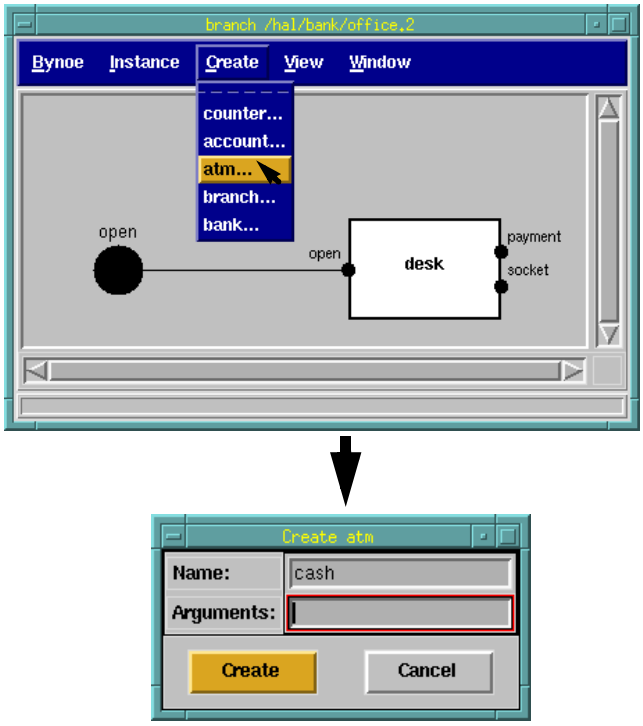

Figure 7. Beagle component instantiation

Components can be created from their program template by dragging their file icon in the domain browser onto a physical host. This can be used to introduce new, previously unknown component classes into the configured system. In particular, such allocation is performed when the initial, top-level component is instantiated and included into an existing management domain. When such new component instances are included into previously running component domains, they become interactively created

subcomponents, which might have been implemented separately from the existing configuration into which they are included.

A component can have pre-programmed classes from which instances can be created interactively by Beagle, i.e. the configuration domain acts as an *abstract factory* for an application. Such instantiation is shown in Figure 7, where a new ATM machine is created in one of the branches initially without one. When the highlighted item in the menu is chosen in Figure 7, the dialogue box shown on the right is displayed. The user can supply a new instance name, and any arguments to the component class (none in the case of the `atm` component class).

When the instance name cash has been supplied in the dialogue box and Create is pressed, the new instance is included as a subcomponent of office.2. These actions, like any sequence of configuration operations, can be viewed as they take place in Beagle, resulting in the configuration view shown in Figure 8. It shows the new instance with unbound requirements as white, empty circles. The `branch` required interface of the new ATM must be bound to a socket before it can be used.

Beagle visualises all configuration changes by monitoring components and their interfaces. It receives events when the configuration of displayed components change.The Create menu pulled down in the top half of Figure 7 also indicates that the bank application has no knowledge of the `customer` or `person` component classes, which are configured and instantiated independently of the bank component.

4.2 Interactive binding

We can now bind the newly created ATM component to the socket interface at the branch counter. The graphical action for performing this binding in Beagle is depicted in Figure 8, where a line is drawn on the configuration canvas to bind the relevant interfaces. Beagle supports both this intuitive form of binding when both interfaces are visible on the same configuration canvas and another form of binding which uses drag-and-drop to bind interfaces in different configuration or management application views. This latter form is shown in Figure 13.

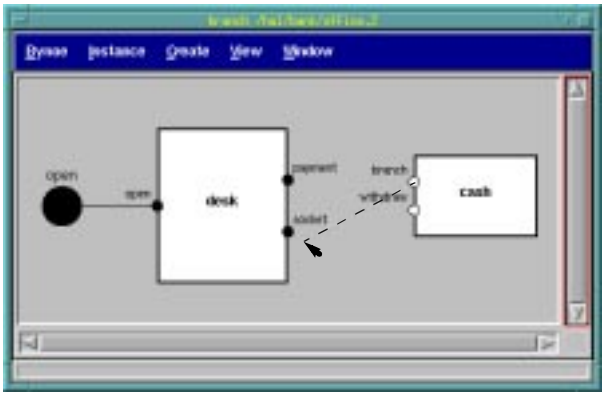

Figure 8. Beagle interactive binding

When the binding in Figure 8 is performed, the `branch` and `socket` interfaces are connected by the solid black line indicating a binding and the requirement is filled with green (or grey) to show its bound status. The ATM can then be used by customers for withdrawing cash from their account held at a different branch, shown in Figure 9. This causes the binding of the ATM's `withdraw` requirement to a provision of the account, although the account component is not shown in Figure 9.

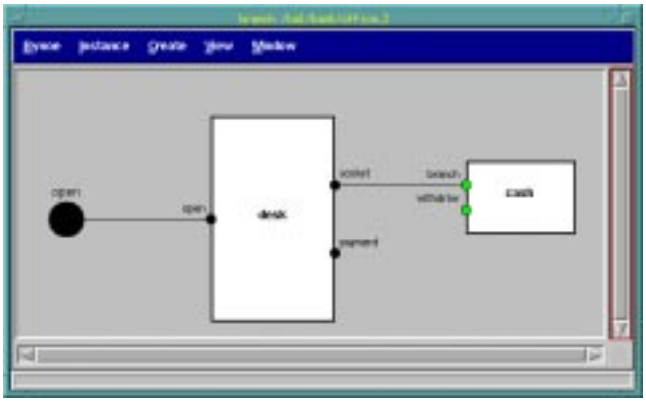

Figure 9. Run-time bindings at ATM

4.3 Safe unbinding

Our bindable requirements support safe un- and rebinding. Our model for safe unbinding is employed in the bank example to enable the ATM to safely complete the withdrawal in progress in Figure 9. If the interactive configuration manager unbinds (or rebinds) the ATM socket interface, the existing binding is critical to the application and remains in use until the transaction completes. This is detected and visualised in Beagle as a (yellow) dashed line as shown in Figure 10. For interactive rebinding, the new binding would be shown as a stippled line to the new provided interface.

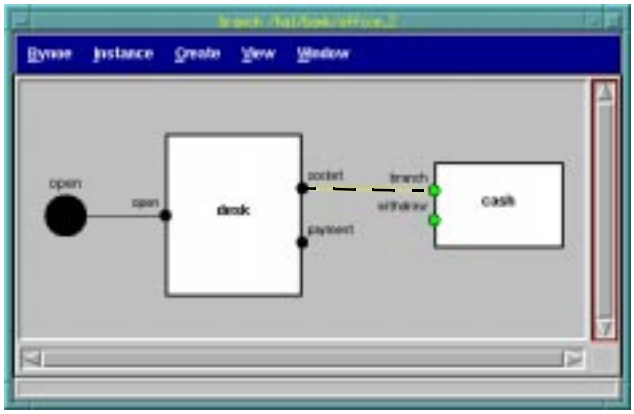

Figure 10. Safe unbinding in progress

Eventually, the transaction will complete and the ATM component marks the branch requirement as safe for unbinding. The yellow line disappears, and any new stippled binding would become solid. This allows rebinding to directly cause a requirement to use a new provided interface without ever blocking. Unbinding is useful to isolate a component before it is destroyed, and in the case of the pending unbinding of Figure 10, the configuration state reverts back to that of Figure 8.

5 Configuration maintenance

Interactive configuration maintenance concerns the management of running configurations and maintaining their functionality in the presence of failures or generally unwanted configuration changes.

5.1 Persistent configurations

Our approach permits the interactive configuration manager to create *persistent configurations* which are representations of the structural or configurational aspects of a running application. These are saved as disk files so they can survive common failure conditions such as process and host failures. They can be used to recreate failed composite configurations, including their interconnecting bindings. Figure 11 shows Beagle invoking the running component domains to save their own configuration state.

There is no automatic saving of the internal state of running primitive components. However, provided they have been programmed to support invocation of operations to save application dependent state, these would be invoked at the same time as saving configuration information. Our system does not address fault-tolerance in the sense of automated masking of failures [9], but allows the interactive configuration manager to view and manage the effects of failed configurations.

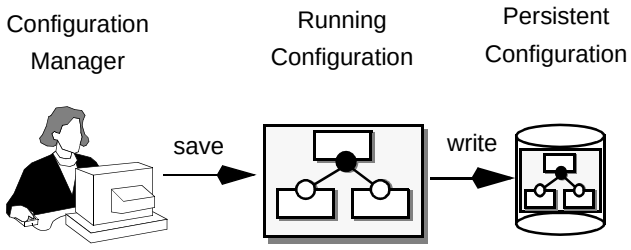

Figure 11. Creating persistent configurations

The interactive configuration manager can choose to save a composite component's configuration when it is in a state which one might want to be able to recreate later. In our bank example, such a state might be the configuration of a branch at the end of a working day, when all transactions of that day are completed. In this section, we assume that such a persistent configuration was saved when the transactions in progress in Figure 6 have completed. In the bank example, our account components are implemented to save their account balance when the configuration state is saved. Although a more realistic implementation would have accounts save an additional log of transactions, our example shows how persistence is achieved at both structural and computational levels.

5.2 Configuration failure detection

Our management architecture enables simple component failures to be detected and then depicted by Beagle. By storing intended configuration information and comparing this to the state of the actual, running configuration, Beagle can indicate failed or unreachable components, e.g. by using red colour.

The effect in Beagle of the failure of the branch office.1 is shown in Figure 12. Such failure can come about as a result of accidental or deliberate, temporary termination, e.g. temporary branch closure. Beagle will close any open configuration view of the failed component, and indicate the failed subcomponent with a thick, red, stippled edge around the component box. This is because its members, e.g. required and provided interfaces cannot be determined from the failed component, but its name and role as a component is determined from redundant configuration information held by the intact parent component.

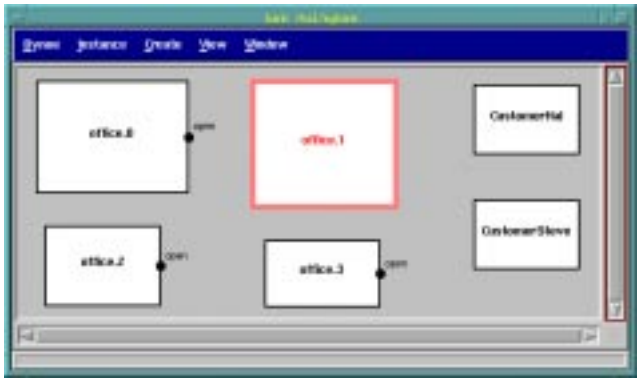

Figure 12. Failed branch in bank

5.3 Configuration remedy

A remedy available after the branch failure of Figure 12 could be to install a temporary ATM machine. We have already shown how this can be achieved within a branch, whereas the lower half of Figure 13 shows how such a temporary ATM has been created as a subcomponent of the bank component. This ATM is bound to the socket interface in the branch office.2 using drag-and-drop, where the idea is that a full provided circle is dragged onto an empty circle to fill the requirement.

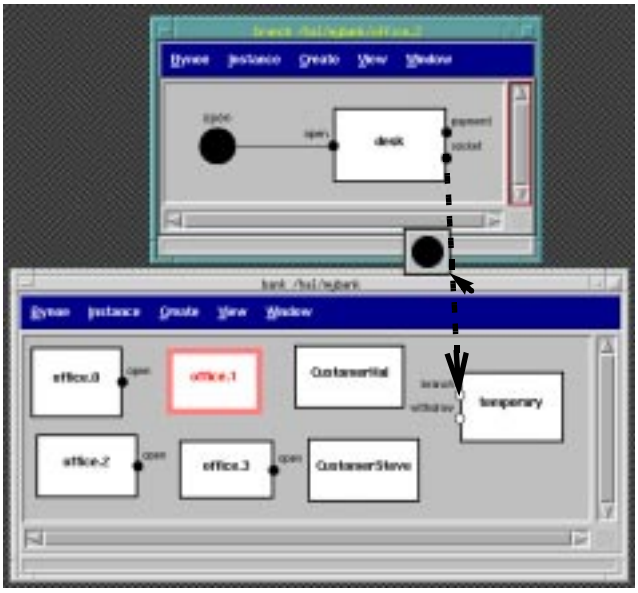

Figure 13. Drag-and-drop binding

5.4 Component recreation

In due course, the failed branch can be recreated using the previously saved persistent configuration. This will also restore the accounts dynamically instantiated at the branch along with their previously saved balances. Such recreation in full is useful since customers know their account numbers in the form of interface references. Recreating the composite structure means that individual interfaces can be referenced in the same way as before the failure occurred.

As recreation commences, the first indication of this in Beagle is that it changes the red, stippled edge of the component office.1 in Figure 12 to the black line of running, components. After the full branch configuration has been re-elaborated, the component can be opened up to show the recreated structure, which is the same as the one saved after completion of transactions in Figure 6.

After the original branch configuration has been recreated, the temporary ATM machine installed is no longer needed. It can therefore be unbound from its branch and destroyed. As we showed in Figure 10, such unbinding is implemented to be safe with regard to any cash withdrawals that are in progress. Once any transaction and thus the unbinding completes, the temporary ATM can be destroyed safely.

6 Implementation architecture

The ICON tools and management services are implemented using Orbix [10] and allow CORBA [3] components to be configured. Provided interfaces can be arbitrary IDL-defined interfaces, which are bound to bindable requirements employed by Orbix clients. Extensive use is made of the Dynamic Invocation Interface (DII) which allows a more elegant implementation than with ANSAware as used in an earlier version [5]. We have also implemented asynchronous event notification on top of Orbix. This is shown in Figure 14. The host manager acts as an agent on each host to monitor state and maintain information of the parent domains of local components and interfaces.

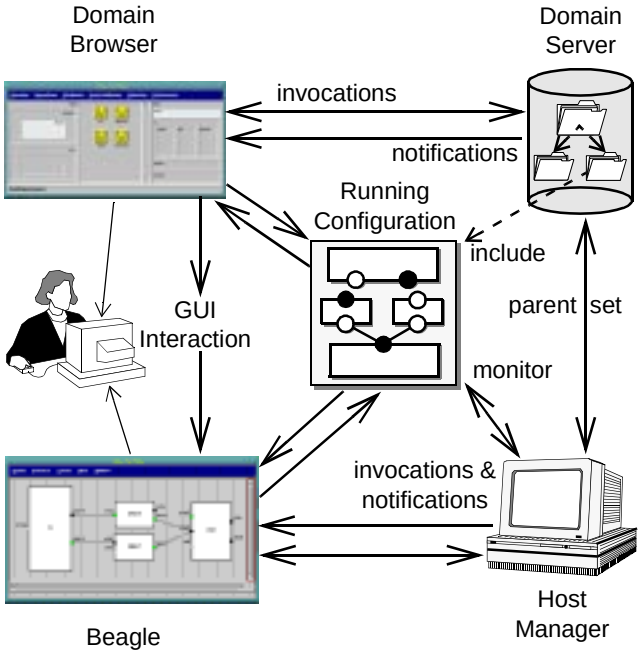

Figure 14. Architecture overview

We have used Tcl/Tk [11] extensively for our integrated management user interface tool suite. The interpreters have been augmented with generic client- and server-side capabilities in Orbix. The tool set is loosely coupled and extensible. For example, colleagues have implemented a management policy editor [12] which interworks with our graphical configuration environment.

The initial configuration elaboration is performed by the executable generated by our Darwin compiler. This creates all the processes required by the configuration and sets up the required bindings.A performance penalty of our implementation is caused by the binding mechanism which uses the native Orbix invocation protocol. This requires all components to consult a daemon for binding establishment. This daemon can easily become a bottleneck during initial configuration, and contributes a delay of nearly a second for every component which is launched. The use of CORBA IIOP (Internet Inter-ORB Protocol) [3] in the future would allow components to by-pass the daemon, but the Orbix IIOP implementation was not sufficiently stable when we embarked upon our implementation work.

Once components have been created and their bindings established, there is little performance overhead for configurable components. Communication is direct from a primitive component to the bound provided interface as the

hierarchical binding structure is “flattened” at run-time. The configuration representation itself is maintained outside the execution context of primitive components, and reconfiguration is implemented as efficient Orbix invocations.

7 Related work

Other authors have similar views of configurations which use composition and third-party binding of interfaces, notably Durra [13], RCMS [14] and Clipper [15]. However, they use configuration languages mainly to set up initial configurations and only have some limited forms of programmed reconfiguration. Although Clipper allows reconfiguration plans to be implemented and executed dynamically, there is little support for long-term evolution of configured applications. Also, these systems do not integrate with an open management architecture with support for interactive, graphical management tools. Durra is posited as useful for fault-tolerant systems, since components can save their state and recover it when recreated. This is similar to our view that primitive components can be implemented to be persistent. However, there is no provision in Durra for persistent configurations.

Our support for configurations uses graphical tools extensively. Other systems incorporate facilities for such design- and implementation-time configuration, e.g. the PRISMA system [17] or VDAB [18]. However, such systems do not generally enable consistent abstractions to be used for run-time, interactive configuration management. Where such graphical configuration support is available, it is commonly engineered into the systems directly [18] as opposed to being available to multiple, external configuration managers.

The CORBA Common Object Services specify interfaces for manipulating graphs of related objects. The Life Cycle Services furthermore support moving and copying such objects, although this focuses on passive objects as opposed to active components. These mechanisms are currently not available in many ORBs, and must be seen as statements of aspiration rather than proven technology.The underlying CORBA ORB merges binding and creation inappropriately, and the lack of third-party binding support makes configuration management unnecessarily difficult during the software life cycle.

A suite of management platforms address the problem of maintaining program and installation files on networked computers, notably Depot [19] and hobgoblin [20]. Depot supports object creation by executing programs, which is related to our different forms of component instantiation. hobgoblin also allows the checking of directory configurations against persistent configuration models. However, these systems nevertheless do not directly address configuration management as the description, composition and interactive management of *running* software components.

8 Conclusions

The work presented in this paper draws upon years of experience with configuration notations and graphical configuration programming. The Darwin language itself was originally developed for Regis [2] and colleagues have developed graphical design support for the language [16]. The key features and contributions of our approach are:

Interactive management of configuration structure

Our model allows the explicit configuration structure to be preserved at run-time. Configured programs and systems can be viewed and monitored, and interactive managers can evolve and maintain systems in terms of their configuration structure.

Consistent configuration abstractions

The Darwin concepts of components, interfaces and bindings are fully represented in the interactive management environment. All aspects of a configuration can be dynamically and interactively modified, including extensive and graphical capabilities for component creation, and full support for interactive binding and safe rebinding.

Textual and graphical configuration

We support interactive configuration management using both textual and graphical notations, and we enable one representation to be translated into another. Darwin code for an evolved, running configuration can be generated interactively for viewing in the off-line design editor and be fed back into the software design phase.

Interactive and programmed reconfiguration

Reconfiguration is supported in the ICON system both interactively and from within programs. Limited programmed reconfiguration can be implemented in the Darwin language, or arbitrary configuration changes can be instrumented in component behaviour. The outcome of reconfiguration actions are independent of what initiated them, and can be monitored by interactive configuration managers, even if they are not the initiators of the reconfiguration operations.

Interactive management of configuration failure

We support interactive management in the presence of configuration failure, and allow failures to be detected and visualised. The structural aspects of applications can be recreated following failure, and with appropriate application support can be used to make configured programs persistent both at the level of configuration and application or computational level.

Management integration

We have instrumented interactive configuration management within a domain-based management platform, but have yet to integrate this with the security enforcement mechanisms developed for this environment. This would permit various levels of authorisation to be assigned to dif-

ferent managers. The performance of our system could also be made to benefit from the use of an external event and monitoring service [22]. Finally, the activities of different configuration managers should be explicitly synchronised, and the use of *management roles* [23] provides a framework in which this will be addressed.

9 Acknowledgements

This work was funded by EPSRC grant GR/J 52693 and by British Telecom through the project *Management of Multiservice Networks*. We acknowledge the contribution of our colleagues at Imperial College for their valuable comments and suggestions, in particular Steve Crane, Kevin Twidle and Nat Pryce.

10 References

[1] 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

[2] 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

[3] Object Management Group: *Common Object Request Broker Architecture and Specification*, Version 2.0, July 1995

[4] Object Management Group: *Common Object Services Specification*, Revised Version, March 1995

[5] H. Fosså & M. Sloman: *Implementation of Interactive Configuration Management for Distributed Systems*, In Proceedings of the Third International Conference on Configurable Distributed Systems, IEEE Press, pp. 44-51, May 1996

[6] H. Fosså: *Interactive Configuration Management for Distributed Systems*, PhD Thesis, Imperial College of Science, Technology & Medicine, July 1997

[7] 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

[8] B. Johnson: *TreeViz: Treemap Visualization of Hierarchically Structured Information*, In Proceedings of ACM CHI'92 on Human Factors in Computing, pp. 83-91, ACM, New York, 1992

[9] F. Cristian: *Understanding Fault-Tolerant Distributed Systems*, Communications of the ACM, vol. 34, no. 2, February 1991

[10] Iona Technologies: *Orbix 2 - Reference Guide*, P2, Iona Technologies Limited, The Iona Building, 8-10 Lower Pembroke Street, Dublin 2, Ireland, October 1996

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

[12] D. Marriott & M. Sloman: *Implementation of a Management Agent for Interpreting Obligation Policy*, In 7th International Workshop on Distributed Systems Operations and Management (DSOM'96), Italy, October 1996

[13] 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

[14] 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

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

[16] K. Ng, J. Kramer & J. Magee: *A CASE Tool for Software Architecture Design*, Journal of Automated Software Engineering, Vol. 3, No 3/4, Kluwer Academic Publishers, pp 261-284, August 1996

[17] J. Berghoff, O. Drobnik, A. Lingnau & C. Mönch: *Agent-Based Configuration Management of Distributed Applications*, In Proceedings of 3rd International Conference on Configurable Distributed Systems, IEEE Press, pp 52-59, May 1996

[18] H-W. Gellersen: *Graphical Design Support For DCE Applications*, DCE - The OSF Distributed Computing Environment - Client/Server Model and Beyond, LNCS, Springer Verlag, pp 267-278, October 1993

[19] W. Collyer & W. Wong: *Depot - A Tool for Managing Software Environments*, Proceedings of Sixth System Administration Conference, p. 153, October 1992

[20] K. Rich & S. Leadley: *hobgoblin: A File and Directory Auditor*, Proceedings of the Fifth Large Installation Systems Administration Conference, p. 199, September 1991

[21] N. Yialelis: *Domain-Based Security for Distributed Object Systems*, PhD Thesis, Imperial College of Science, Technology & Medicine, August 1996

[22] M. Mansouri-Samani & M. Sloman: *A Configurable Event Service for Distributed Systems*, In Proceedings of 3rd International Conference on Configurable Distributed Systems, IEEE Press, pp. 210-217, May 1996

[23] E. Lupu & M. Sloman: *Towards A Role Based Framework For Distributed Systems Management*, Journal of Network and Systems Management, vol. 5, no. 1, January 1997