SOFA/DCUP: Architecture for Component Trading and Dynamic Updating

František Plášil[1,2], Dušan Bálek[1,2], Radovan Janeček[1,2]

[1] *Charles University*
Faculty of Mathematics and Physics,
Department of Software Engineering
Malostranské náměstí 25, 118 00 Prague 1,
Czech Republic
phone: +420-2-2191 4266
fax: +420-2-532 742
e-mail:{plasil, janecek}@nenya.ms.mff.cuni.cz

[2] *Institute of Computer Science*
Czech Academy of Sciences
Pod vodárenskou věží
180 00 Prague 8
Czech Republic
phone: +420-2-6605 3291
fax: +420-2-858 5789
e-mail: {balek, plasil}@uivt.cas.cz

www: http://nenya.ms.mff.cuni.cz

Keywords: software component, dynamic updating, Java, CORBA, Module Interconnection Language, IDL, externalization, electronic commerce

Abstract. In this paper, the authors address some of the challenges of the current technologies in the area of component-based programming and automated software downloading. These challenges include: component updating at runtime of affected applications, adopting the "true-push" model in order to allow for silent software modification (e.g. for removing minor implementation errors), and finding a way to integrate these technologies and electronic commerce in software components. To respond to these challenges, the SOFA (*SOF*tware *A*ppliances) architecture, the SOFA component model and its extension, DCUP (*D*ynamic *C*omponent *UP*dating), are introduced. SOFA and DCUP provide a small set of well scaling orthogonal abstractions which address three areas: the background for electronic commerce, the component model, and support for dynamic component updating in running applications. The updating granularity can scale anything from minor implementation changes to a major reconfiguration. In contrast with the usual belief that it is difficult to map abstractions supporting component based programming to concrete computer systems, the abstractions proposed by DCUP are very easy to map to the Java and CORBA programming environments.

1 Introduction

It is generally believed that in the potentially near future, many software applications will be integrated from reusable components and that there will be a (mostly electronic) market in such components. Inherent in this idea, in addition to all the issues related to the electronic commerce itself (trading, selection, security, etc.), is the necessity to customize such a component in accordance to specific requirements of a particular application and to set the necessary interconnection with other parts of the application. This issue is also studied in connection with frameworks [CACM97, FSJ98, PREE96]. What still remains as a particular challenge is the component updating (initiated by the component's provider) with minimal human effort/interaction at the end-user side. The issue is even more complex in the case where the updating has to take place at runtime, as is necessary e.g. in real-time applications.

1.1 Component-based Programming

There are at least two strong arguments for employing component based programming: (1) Around the world there are a number of software modules which offer services, therefore reusing them is desirable in order to facilitate the software development process [BEC96, PUR94]. (2) Programming using component technology is more effective for several reasons: it eliminates debugging of the reused parts, there are more opportunities

for visual manipulation [JBN97], and it makes it easier to arrange for a reconfiguration of an application [GK96, MDK94, IB96, BAB96].

There is a group of works ([FS96, GK96, MDK94, IB96, BAB96]) which focus on identifying a way to define interfaces of software components and to specify the structure of an application in terms of interface interconnections (including distributed applications in most cases). Essentially, all of them use some kind of configuration language (also called MIL, Module Interconnection Language), that allows interfaces of software components to be defined, and to specify the structure of an application in terms of interface interconnections. Even though some of the approaches allow for dynamic reconfiguration, as in [FS96, JBN97, GK96], in principle these works deal mainly with relations among components and do not focus on the issues related to the internal state of the components being subject to reconfiguration. A further problem is how to control the actual reconfiguration/update of a component and what kind of knowledge of the remaining part of the application is necessary. A typical approach chosen is an interactive communication via some kind of "application builder" [JBN97, FS96]; usually, an update has to be done by a qualified person aware of the structure of the component framework which is being rebuilt.

In general, updating a component at runtime inherently means disconnecting it from the other parts of the application and connecting a new version of the component back into the application. Here, two key issues arise. First, what to do if the new component does not support exactly the same interface as the old one; here connectors [BAB96] are a way to deal with the problem. Second, references among components have to be updated. These references are typically handled by a higher-level reference abstraction (e.g. CORBA reference [OMG95a], event listener [JBN97]) which can be reassigned to a target in the new component, or by the introduction of auxiliary objects which mediate access to a component (wrappers, proxies).

Even though the above-mentioned sounds promising, to paraphrase [MB96], the main obstacle in a large application of component based programming is the difficulty of mapping the proposed abstractions into concrete working computer systems. This is reflected by the fact that there are only a few significant commercial products available at the moment (e.g. [UNIF97, JBN97, KON96]). Also, to our knowledge, the only standardizing body active in this area that is recognized world wide, is the Object Management Group (OMG) [OMG97b].

1.2 Automated Software Downloading

In 1990 the Intermind [INT97] introduced a concept based on the idea of a control structure exchanged between an information publisher and subscriber. The publisher creates the control structure describing how to automate communications between the subscriber and the publisher; the subscriber uses a special program to store and process the control structure to automate the flow of information from the publisher. This type of communication has been called a *channel*, and the control structure a *channel object*. So far, several enterprises have adopted this scheme of communication. Microsoft has proposed its own specification for channel objects, Channel Definition Format (CDF) [CDF97], which is used by several software companies, e.g. [PNT97, BWEB97]. Other companies have used a similar approach, for example [INT97, NCAS97, MA96]. The standardizing body active in this area is W3C [OSD97].

To protect users from having to search the Web for the latest versions of products, these companies have taken the approach of providing the requested products in an automated way - they push the product (e.g. software package) to subscribers. Technically, the channel transmission is activated at the subscriber side, based on the time schedule provided by the publisher at the subscription moment. After the communication is initiated, the requested information is downloaded to the subscriber. As the whole process is automated from the subscriber’s point of view, it creates the impression that the requested information is *pushed* to the subscriber. In compliance with [W3C97], we prefer this type of information (software) distribution to be called a "smart pull" model. As for granularity, the typical unit of information exchange is a file (HTML documents, native code file, etc.).

1.3 Challenges, SOFA and DCUP

The purpose of this paper is to address some of the challenges of the current technologies mentioned in Sections 1.1 and 1.2. In the area of component-based programming, the key challenge we focus on is dynamic component updating even at runtime of affected applications. In automated software downloading, the key issues to be targeted are (1) considering a software component together with the entire context necessary for a dynamic update to be a unit of transmission (not just a file), and (2) adopting a "true-push" model as well, in order to allow for silent software modification (e.g. for removing minor bugs). In both technologies, the key issue we concentrate on is how to combine them with electronic business and market phenomena; in other words, the question is how these technologies and electronic commerce in software components can be integrated.

In this paper, to address these challenges, we introduce the SOFA architecture, the SOFA component model, and the SOFA component model extension called *DCUP* (*D*ynamic *C*omponent *UP*dating). The *SOFA* (*SOF*tware *A*ppliances) architecture objective is to provide a small set of well scaling orthogonal abstractions to model trading in software components over a computer network, and, at the same time, to support their instantiation into running applications where they can even be subject to updating. To reflect these objectives, these abstractions address three areas: (1) the background for electronic commerce in components, (2) the component model, (3) support for dynamic component updating in running applications (DCUP).

Principally, SOFA encompasses several software domains, e.g. the communications middleware, component management, component design, electronic commerce, and security. The main issues to be addressed by SOFA include component transmission protocol, dynamic component updating, component description, component versioning, and support for component trading, licensing, accounting, and billing.

1.4 The Goals of the Paper

Although this paper focusses particularly on DCUP, our first goal is to describe, in a concise form, the features of SOFA which were designed in order to address the challenges pointed out in Section 1.3. These features include:

- SOFA Component model and CDL (Component Description Language) which support versioning by strictly separating the component interface from the component architecture.
- SOFAnode, a small set of well scaling orthogonal abstractions, which provide a unified view of all the potential roles of the parties involved in the component electronic commerce process (component producer, retailer, end-user, etc.). These abstractions allow for a party to take several of these roles at the same time.

Our second goal is to introduce and demonstrate the key features of DCUP, which are novel with respect to the existing technologies and which significantly contribute to component updating at runtime. These feature include:

- Architectural support for component state transition during updates.
- Architectural support for versioning (Versioning is not addressed as a special case of reconfiguration; however, reconfiguration is considered a special case of versioning).
- Dynamic updates done with no human intervention at the subscriber (end-user) side.

1.5 Structure of the Paper

The paper has the following outline. The basic concepts of the SOFA architecture are described in Section 2. In Section 3, an overview of the DCUP architecture is provided; the SOFAnode - DCUP interplay is described in Section 3.5. As a proof of the concept, a simple application has been prototyped. Section 4 presents the key fragments of this application and summarizes the experiences gained while designing and debugging the prototype application. Section 5 is devoted to our future intentions. Finally, the main achievements are summarized in the concluding Section 6.

2 SOFA Architecture Overview

2.1 SOFA Component Model

As indicated in Section 1.3, SOFA is designed to provide software components and, moreover, run applications composed of these components. The abstractions introduced in this section constitute the SOFA component model.

In analogy with the classical concepts of an object as an instance of a class, we introduce a software component (*component* for short) as an instance of a component template (*template* for short). Basically a component is a framework of local implementation objects and nested component instances. In principle, a template is a triple <template_interface, template_architecture, internal objects> - see below.

Each template has associated with it a unique trademark in the form of a triple <provider_name, type_name, version_name>. Each provider has its own local type_name space and version_name space. The type_name space is, as usual, tree-like hierarchy of composed names. However, there is single (global) flat provider_name space. The specification of version_name space is provider-dependent. Given the trademark <prov_name, type_name, ver_name> of a template T, the pair <prov_name, type_name> (by convention written usually in the form `prov_name : type_name)` is called the *template interface name* of T.

Some of the template instances can be run as applications; such an instance is called a *primary component* and the corresponding template is a *primary template*. The other templates are called *secondary templates*.
A *CDL (Component Description Language)* developed as part of the SOFA project is used to specify the interface and architecture of a template. The following example illustrates the specification of two template interfaces: `AProvider: Bank` , `AProvider:Supervisor`.

```
#import "BProvider:DataStore.cdl"

template AProvider: Bank (Property: num_of_tellers) {
  provides:
    TellerInterface teller[num_of_tellers];
}

template AProvider: Supervisor {
  provides:
    SupervisorInterface supervisorAccess;
  requires:
    DataStoreInterface datastoreAccess;
}
```

Similarly to the component class in Olan [BAB96] and the component construct in Darwin [MDK94], a template interface specifies (in the `provides`, resp. `requires` clauses) the names and interfaces of the services it provides, resp. the names and interfaces of the services it requires. Thus, in the above example, the template interface `AProvider: Supervisor` (of the template type_name `Supervisor` in the `AProvider` name space) provides the `supervisorAccess` service of `SupervisorInterface`. The interface types of services both provided and required are specified using a CORBA IDL-like language which, for brevity, is not described in more detail here. Note that generic parameters (`num_of_tellers` in this example) are based on the `Property` notion.

The template_architecture of a template with a given trademark is a pair <nested components, bindings>. *Nested components* is the set of local (child) instances of some other templates in this (parent) template. Bindings define the connections among the parent template and its child components' interfaces and among the sibling components' interfaces. The following example illustrates the idea.

```
architecture AProvider: Bank version 1.1
    inst AProvider: Supervisor version 4.2  s;
    inst BProvider: DataStore version 3.1  ds;

    bind s.datastoreAccess ds.datastoreAccess;
}
```

The template architecture of the trademark `AProvider: Bank version 1.1`, contains an instance `s` of `AProvider: Supervisor version 4.2` , and an instance `ds` of `BProvider: DataStore version 3.1`.

The *internal objects* of a template T are the objects used for instantiating and implementing the "internals" of T's instances. (This is not to be understood recursively with respect to component nesting.) Usually, one of the internal objects takes the role of a *builder* with a task similar to that of a class constructor, i.e. it creates all other internal objects and the nested (child) components of T.

Based on the SOFA component model, we further define the *DCUP architecture* that is designed to allow for dynamic component updating at runtime. Similarly, the components (and templates) based on this architecture are *DCUP components* (*templates*). The DCUP architecture is described in Section 3.

2.2 SOFAnet and SOFAnode

In an electronic commerce in software components, the parties involved take various roles, e.g. those of a customer (end-user), retailer, producer, and provider. A party can take multiple roles at the same time, for example an end-user may also be a retailer. Furthermore, the parties can be engaged in several types of relationships, like trading, advertising, ordering services, payment, etc.

As a basis for modeling these roles and relationships, SOFA introduces the SOFAnet and SOFAnode concepts. *SOFAnet* is a homogeneous network of SOFAnodes. A *SOFAnode* is a tuple <In, Out, Made, Run, Template Repository>. The members of this tuple are called *SOFA parts*. The heart of the SOFAnode architecture is the Template Repository (TR); the functionality of TR includes storing, versioning, and naming of templates/components. The TR part is used by the rest of the SOFAnode, namely by the In, Out, Run, and Made parts.

TR	Template Repository. Contains all the templates available at this SOFAnode. Supports template/component versioning and naming. Not directly accessible from outside the SOFAnode.
In	Connected to the Out parts of a subset of SOFAnodes (to the In part's *outscope*). Serves as the "entry" point to the SOFAnode. Via its Out part, a provider of components wishing to push any updates or installations of templates needs to contact this In part. Moreover, it may contact an Out part in its outscope to invoke update/installation on demand (pull model). In an extended form, it may contact its outscope in order to support electronic commerce in components (e.g. support for component trading, licensing, accounting, and billing and payment).
Out	Connected to the In parts of a subset of SOFAnodes (to the Out part's *inscope*). Serves as the "output" point of the SOFAnode for transferring new templates and update requests to its inscope (in both push and pull models). In an extended form, it may contact its inscope in order to support electronic commerce in components (e.g. support for component licensing, feedback collection, accounting collection, billing and payment).
Made	A gate for newly created templates to enter SOFAnet. Provides environment for template software developers. Allows for creating out-of-SOFAnet applications from primary templates.
Run	Provides environment for launching and running of applications. Instantiates templates and passes property objects to them. It may provide other support for the running applications, e.g. access to persistent datastores.

Table 1: Basic functionality of SOFAnode parts

A *SOFAnet* is a directed graph of SOFAnodes such that if e is an edge connecting SOFAnodes A and B, e leads either from the Out part of A to the In part of B and $A \neq B$, or it leads from the Out part of B to the In part of A and $A \neq B$.

The motivation for introducing the particular parts of the SOFAnode is illustrated by the following special cases (Figure 1): "pure end-user" is modeled by a SOFAnode with Out = 0 & Made = 0 (0 denotes empty functionality); similarly, "provider and producer" is modeled by In = 0 and "retailer" by Made = 0 & Run = 0.

As it is not the purpose of this paper to describe SOFAnet in more detail, we provide only a short summary of SOFAnode functionality in Table 1. This section is also intended to provide background for Section 3.5, where we describe an extension of the Run part in order to support runtime updating in DCUP applications. So far, based on IDL interfaces, we have defined the basic functionality of the SOFAnode parts [PTR97], which includes a simple component transmission protocol but which does not cover the electronic commerce related operations. We intend, using the usual inheritance approach, to extend this basic functionality to deal with electronic commerce support.

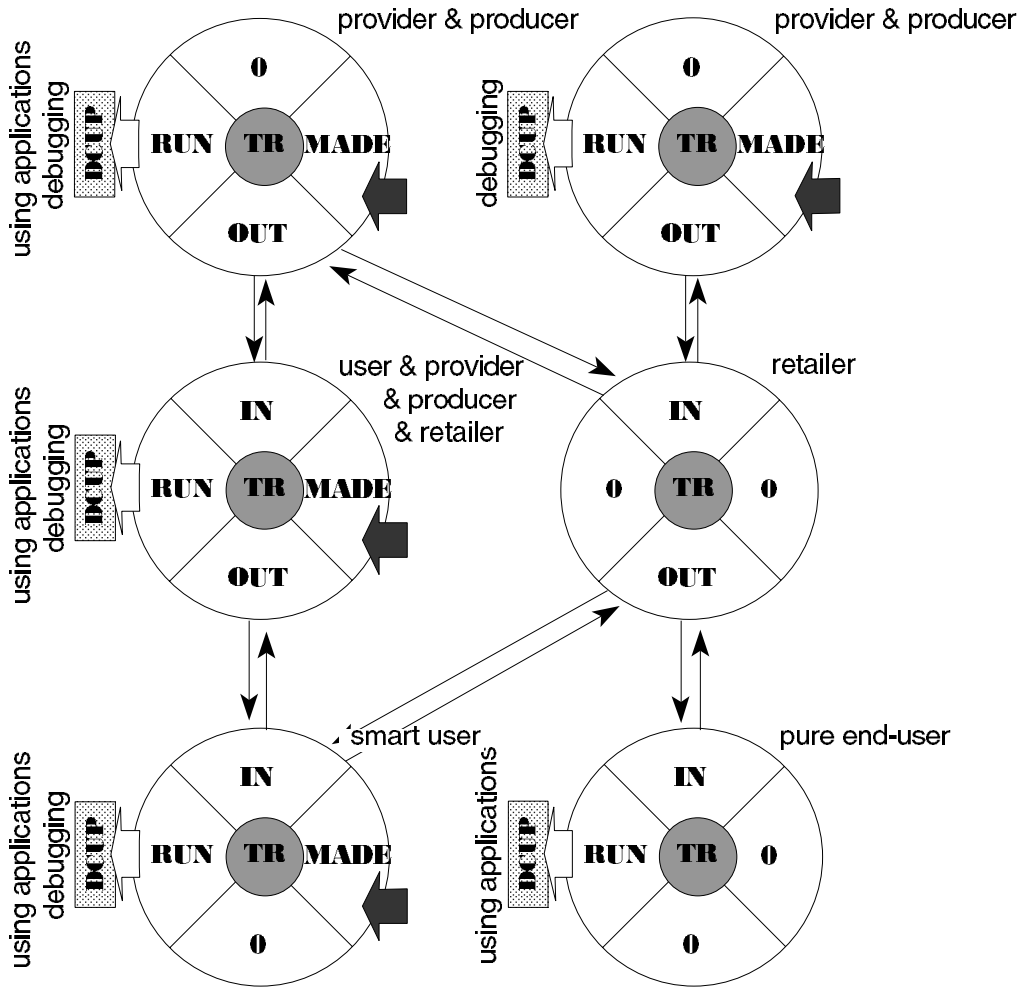

Figure 1 SOFAnet example

3 DCUP Architecture Overview

The DCUP architecture is a specific architecture of SOFA components which allows for their safe updating at runtime. It extends the SOFA component model (Section 2.1) in the following way: (1) It introduces specific implementation objects. (2) It makes the way components are interconnected more specific. (3) It presents a technique for the updating of a component inside a running application. (4) It specifies the necessary interaction between a running application and the Run part of a SOFAnode. This section describes the DCUP version for the Java environment; this version exploits some of the features specific to Java (e.g. redefinition of class loaders). We intend to design a DCUP version also for, e.g., CORBA. During the DCUP design, we were significantly inspired by the mobile agent concept, especially by the Aglet Approach [LAR96], and by our experience in designing and implementing the CORBA Object Persistent Service [KPT96].

3.1 Structure of a DCUP Component

The DCUP components are dynamically updatable; with respect to an update operation a component is divided into a *permanent part* and a *replaceable part*. Orthogonally, with respect to the nature of the operation provided, the component is divided into a *functional part* and a *control part*. The respective interfaces are called *control interface* and *functional interface.* The control interface is uniform across all DCUP components and it is used only for managing purposes (e.g. starting an update). On the other hand, the functional interface corresponds to the component interface described by the SOFA component model (Section 2.1).

The DCUP architecture introduces several specific (control) implementation objects: Component Managers, Component Builders, Updaters, Class Loaders, and Wrappers. Every DCUP component has to contain exactly one instance of Component Manager (*CManager* for short) and exactly one instance of Component Builder (*CBuilder* for short). A CManager is the heart of the component's permanent part, existing thus for the whole lifetime of the component. The key task of the CManager is to coordinate updates. On the other hand, a CBuilder (the key object of the component's replaceable part) is associated with a particular version of the component only, and it is therefore replaced together with each of the components' versions. The key task of a CBuilder is to build/terminate the replaceable part of a component (including restoring/externalizing component states whenever necessary).

In an application, a component may have an *Updater* associated with its CManager. The role of an Updater is to accept updating requests coming from the Run part of the SOFAnode and to take appropriate action in the corresponding components. The general rule for determining whether the given component has its own Updater associated with its CManager (for brevity we say *the component has an Updater)* can be described as follows: (1) Every primary component has an Updater. (2) Every component with the provider_name (Section 2.1) different to the provider_name of its parent component has an Updater. Similarly, we define the *Updater scope* of an Updater recursively: (Let C be a component and U the Updater C has): (1) Initially, the Updater scope of U contains C. (2) If C belongs to the Updater scope S, then all of C's subcomponents which do not have an Updater, belong to S.

Finally, we have to add that every Updater is responsible only for updating components in its scope. An Updater is not allowed to update a component that does not belong to its scope.

Wrappers are implementation objects closely related to the functional interface of a component. Basically, each of the services provided by a component has a Wrapper object associated with it (in a one-to-one relationship). The Wrapper object mediates access from the outside of the component to the service implementation and it allows for a transparent and safe update.

Classloaders are typical examples of the exploitation of a specific feature of the Java environment. Java allows for dynamic class loading into the application's runtime. Whenever an application needs to create a new instance of a class that has not been loaded yet, a classloader is asked to load it. Thus, by writing the special classloader that can contact and communicate with a template repository of a SOFAnode (via its Run part), we can ensure that the proper versions of classes (corresponding to the current version of a component) are always loaded into the runtime. Each component has its own instance of classloader.

In summary, a component is a framework divided, with respect to updating, into the permanent part and the replaceable part (Figure 2a). The permanent part contains a CManager and service wrappers of the component. The replaceable part contains a CBuilder, functional objects, and subcomponents of the component. With respect to operations provided, the component is orthogonally divided into its functional part and control part (Figure 2b). The functional part contains functional objects together with their respective wrappers, and subcomponents. The control part contains CManager and CBuilder.

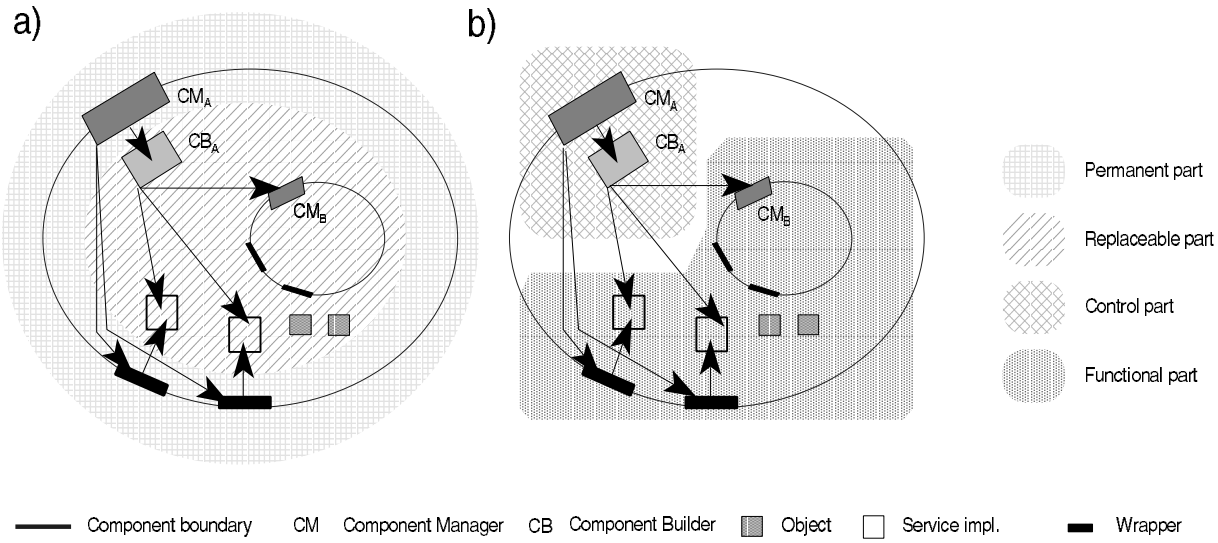

Figure 2 Structure of a component

3.2 Creating a DCUP Component

Component creation is a process that results in the appearance of a new component in the runtime of an application. As the first step, the application has to create an instance of the CManager of this component. The instantiation of the CManager object causes the creation of the rest of the component's permanent part. Afterwards, to create a replaceable part (RP) that has the real functionality of the component, the application has to invoke `createComponent()` on CManager. This method instantiates the current version of CBuilder that represents the i-th version of the component, and invokes upon it `onArrival()`.

The $CBuilder_i.onArrival()$ method creates the internal objects and the subcomponents of RP_i, initializes their state (e.g. from the externalized state of a previous run/version of the component), and sets up all internal references in RP_i (among the internal objects, subcomponents, etc.). We say that $CBuilder_i.onArrival()$ *builds* RP_i. To *terminate* RP_i, the $CBuilder_i.onLeaving()$ method is used. This method ends all execution threads in the replaceable part of the component, and may externalize the state of its "important" objects; finally it destroys all internal objects and subcomponents in RP_i.

Note: Creating the primary DCUP component C (passing all the necessary property objects to it) corresponds to launching the application determined by the C template and the property objects as the actual generic parameters.

3.3 Interconnecting DCUP Components

As mentioned in Sections 2.1 and 3.1, the functional component interface contains the `provides` and `requires` clauses. Recall that each clause contains a list of services' interfaces.

a) **Services provided.** Services provided by a component C are directly accessible from its parent component P only. An access to the component C from a higher level component has to be mediated via P (P has to

explicitly export (part of) C's interface). An access to a particular service of the component C can be obtained via a *bindToService()* call upon the C's CManager specifying the service name as a parameter. As a result of the *bindToService()* operation (*binding*), the caller gets the reference to the wrapper object WO representing the corresponding service (Section 3.1). For an example please see Section 4.3.

b) **Services required.** Basically, it is the parent component's responsibility to provide references to all the required services. In DCUP, a component C indicates its requirements via the *getRequirements()* method of its CManager (the requirements are returned as a list of services' names). The parent component P sets the references to the corresponding services by using the *provideRequirements()* method of C's CManager. Recall that references to all the services required by C have to be set before the first access to any of the services provided by C. For an example, please see Section 4.3.

3.4 Component Updating

By the *updating* of a component we mean replacing its replaceable part by a new version of this part at runtime. Thus, the *lifecycle* of a component is the sequence RP_1, RP_2, ... , RP_n, where RP_i is the i-th version of the replaceable part. Recall that each RP_i version of the replaceable part is associated with a $CBuilder_i$ (the i-th version of the Component Builder).

The updating transitions RP_i -> RP_{i+1} in a component are controlled by its CManager. To terminate an RP_i, the CManager calls $CBuilder_i.onLeaving()$, loads a new version of the CBuilder class, creates a $CBuilder_{i+1}$, and builds an RP_{i+1} by calling $CBuilder_{i+1}.onArrival()$. More specifically, the updating transitions are determined by the *updateComponent()* method of the CManager. The basic functionality of this method can be captured by the following pseudo-code:

```
public class CManager {
  .
  .
  updateComponent(String subComponentName, String builderClassFile,
                  String storeDataStoreID, String restoreDataStoreID){
    // if ComponentName is the name of a subcomponent, then delegate this call to the
    // corresponding CManager, else:
    OldBuilder.onLeaving(storeDataStoreID);
    NewBuilder = new ComponentBuilder(BuilderClassFile);
    NewBuilder.onArrival(RestoreDataStoreID);
  }
}
```

3.5 SOFAnode - DCUP Application Interplay

For the Run part, the structure of an application is transparent. The Run part does not even know the main class of the application or the builder classes of particular component versions. The only thing the Run part knows is a list of primary templates (that represent executable applications) stored in TR. Important interactions between an application and the SOFAnode are launching, Updater registering, component updating, and application terminating.

When a user starts an application via the Run user interface, the Run part asks TR for the main class of the application and executes it as a new process (in a web browser for applets, or JVM for standalone applications). The main class then creates the CManager of the main component that handles the creation of the rest of the application. During component creation, each CManager associated with a component which has an Updater (Section 3.1) has to create an instance of the component's Updater. To be able to receive update messages the Updater must register itself with the Run part. If a new version of an updatable component arrives at the In part of a SOFAnode while the component is running, the In part passes the component instance identification to the Run part. The Run part then identifies (in cooperation with TR) which of the registered Updaters have this component in their scope and asks these Updaters to invoke an update on the corresponding instance of the component (Section 3.4).

Thanks to component state externalization that is used to preserve (and potentially transform) component's state during the update process, the whole application can store its state before a user terminates it. This optional action on the application is also done under control of the Run part. Finally, it is worth remarking

that these operations on running applications are atomic and mutually exclusive. For example, it is impossible to terminate an application during an update of its internal component.

4 Banking Application - Case Study of DCUP Use

By presenting an example, this section illustrates how the basic DCUP ideas and concepts can be applied in an (almost) real-world situation. The example was implemented as a proof-of-the-concept application.

4.1 Application Description and its Decomposition into Components

Let us imagine that there is a bank in which a number of tellers serve a potentially huge number of customers. A customer specifies the desired transaction to a teller who then accomplishes the request. To perform certain transactions on accounts, such as an overdraft, tellers have to apply for a supervisor's approval. Both the teller and the supervisor need access to an account repository.

To model this hypothetical bank, we introduce the following templates: `Bank`, `Supervisor`, and `DataStore`. For the CDL description of the `Bank` template, we refer the reader to Section 2.1. In DCUP, each component is an autonomous unit of code that can eventually be updated (e.g. as the number of customers of this bank grows, we may realize that our implementation of `DataStore` does not satisfy the original requirements for accessing stored accounts and therefore `DataStore` has to be updated). The advantage of our approach is that such an update can be safely processed even at runtime of the corresponding application, and, moreover, without any administrative effort at the end-user side. A component provider only sends an update message to the corresponding SOFAnode, which passes it via its Run part to the appropriate Updater(s). Our hypothetical bank application does not need to be shut down each time an update occurs (this is particularly important for real-time applications).

In accord with the overall philosophy of DCUP, the whole Banking Application is implemented as the primary template `MainComponent`. The desired nesting and instantiation of all the mentioned components is illustrated in Figure 3.

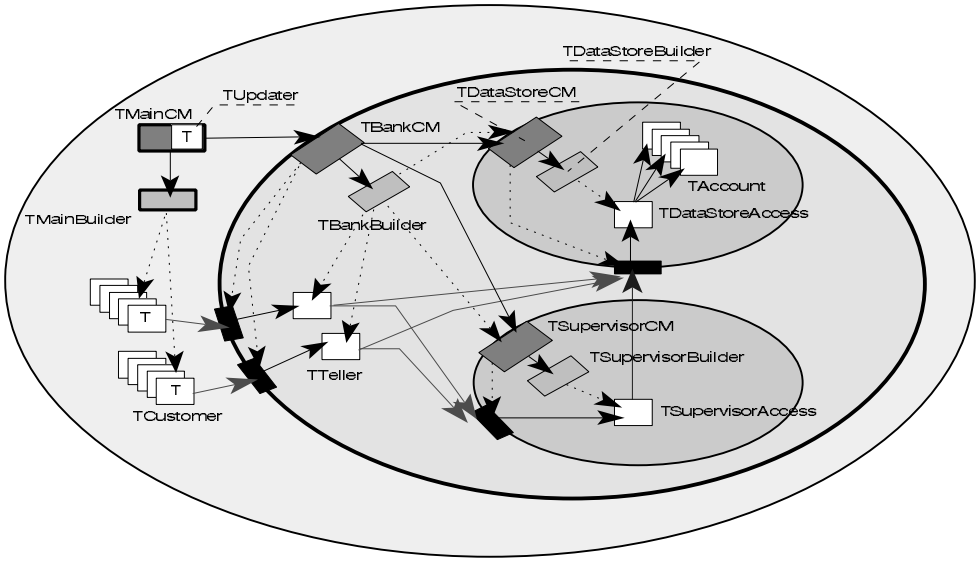

Figure 3 Structure of the application

4.2 Launching an Application

To start the Banking Application, we create *MainComponent*. This is done by the following code:

```
public static void main(String []args){
  TMainCM app = new TMainCM( ... );
  app.createComponent();
}
```

Remember that the CManager of an application's primary component (*MainComponent* in this example) is associated with an Updater. The creation of the Updater and its registration with the Run part of the SOFAnode is processed inside the CManager constructor. The *registerUpdater()* method returns the unique ID of the *MainComponent* template description (of its current version) within TR (*template descriptor*):

```
TMainCM( ... ){
  ... MainID = RunPart.registerUpdater(new TUpdater()); ...
}
```

In a similar way, all the Updaters possibly associated with some of the subcomponents in the application are registered immediately after they are created.

The *createComponent()* method creates a new instance of classloader, passing MainID to it. Whenever a classloader needs to load a class, it contacts TR and passes the classname and the template descriptor to it. With the aid of this information, TR returns the corresponding version of the requested classfile. Using the newly created classloader, *createComponent()* instantiates the TMainBuilder class and calls its method *onArrival()*, which then builds the whole framework of the MainComponent. Thus, following Figure 3, TMainBuilder creates the *Bank* component and a number of the TCustomer objects. The *onArrival()* method might look as follows:

```
// in TMainBuilder class
public void onArrival(String dataStoreID){
  BankCM = new TBankCM();
  ParentCM.registerSubcomponent("Bank", BankCM);
  BankCM.createComponent();
  for(int i = 0; i < NumOfCustomers; Customers[i++] = new TCustomer());
}
```

A substantial part of the subcomponent initialization (e.g. the setting of the template descriptor) is performed within the *ParentCM.registerSubcomponent()* method.

Once a TCustomer is created, it binds itself to a (randomly chosen) Teller and uses its services:

```
CI = (TellerInterface) ParentCM.bindToService("Teller3");
accNumber = CI.createAccount(1500);
CI.deposit(accNumber, 2000); CI.withdraw(accNumber, 1000); ...
```

4.3 Creating other components

The creation of all other subcomponents follows the same scheme as the creation of *MainComponent* (except for the fact that the *MainComponent* obtains all necessary initial information from the Run part while other components are initialized by their parent components). Thus, *BankCM.createComponent()* creates its own classloader, instantiates the builder BankCB, calls the builder's *onArrival()* method, etc. The *BankCB.onArrival()* might look as follows:

```
// in the TBankBuilder class (instantiated as a BankCB)
public void onArrival(String dataStoreID){
  SupervisorCM = new TSupervisorCM();
  ParentCM.registerSubcomponent("Supervisor", SupervisorCM);
  DataStoreCM = new TDataStoreCM();
  ParentCM.registerSubcomponent("DataStore", DataStoreCM);
  SupervisorCM.createComponent();
  DataStoreCM.createComponent();
```

After that, BankCB asks its subcomponents for their upward reference requirements by calling *getRequirements()* upon their CManagers. Such a call returns a Requirements object that encapsulates the references requested. In our example, BankCB provides the *Supervisor* component with the reference to the DataStoreAccess service by calling *provideRequirements()* on the Supervisor's CManager:

```
  Requirements Req = SupervisorCM.getRequirements();
  DataStoreInterface DSI = DataStoreCM.bindToService("DataStoreAccess");
  Req.supply(DSI);
  SupervisorCM.provideRequirements(Req);
```

After the implementation objects of the services provided are created (`Tellers` in this example), `BankCB` has to register these objects under the appropriate service names. This registration allows for the binding to the services from the outside of the component. Further, every `Teller` is provided with references to the services provided by the DataStore and Supervisor components; this ends *`onArrival()`* of `BankCB`:

```
  SupervisorInterface IS = SupervisorCM.bindToService("SupervisorAccess");
  for(int i = 0; i < NumOfTellers; i++){
    Tellers[i] = new TTeller();
    ParentCM.registerServiceImplementationOjbect("Teller"+i; Tellers[i]);
    Tellers[i].setSupervisor(SI);
    Tellers[i].setDataStoreReference(DSI);
  }
}
```

4.4 Updating

The Updater associated with `MainCM` waits for an update message from the SOFAnode Run part. After it receives an update message, it calls *`updateComponent()`* upon its CManager. Thus, for example, the DataStore component could be updated as follows:

```
MyCM.updateComponent("Bank.DataStore", newMainID,
                          "/Repository/CityBank", "/Repository/CityBank");
```

The first string parameter denotes the DataStore component within the Bank component. The second parameter represents the new template descriptor, and the last two parameters are the store, resp. restore locations used for externalizing the DataStore components' state. Note that updating can be done without the tellers and customers being aware of it.

4.5 Experience gained from prototype implementation

For the externalization of components' states during updates, a simple stream-based mechanism was employed. Every CBuilder was associated with a dedicated directory in the local file system. A CBuilder externalized the state of all the "important" objects in the component into a file in this directory. To each subcomponent, a subdirectory was assigned (recursively). As an aside, we intend to identify a way to automatize externalization of a component state during updates. One of the issues here is to specify the "persistent state" of the objects involved (the subset of attributes which is worth externalizing). A remedy here might be a property-like identification of these attributes, similarly to the CORBA Property Object Service [OMG95b]. For debugging purposes, we took advantage of the clear Updater separation from the source of update messages.

5 Future Intentions

As for SOFAnode, one key step to be taken in the near future is to spell out (e.g. in a form of extensible IDL interfaces) a support in the In and Out parts for component trading (in the sense of CORBA Trading Service [OMG96a]), licensing, accounting, and billing. Several approaches come to mind ranging from activities of standardizing bodies (OSM [OSM97], OMG Electronic Commerce Domain TF [OMG96b]); unfortunately, none of these activities has reached a mature specification at present), over TINA/TANGRAM [EFS97], to a "proprietary" approach.

In DCUP a transactional service is to be incorporated to handle two situations. First, dynamic updates of nested components should be subject to transactions in order to recover from an update failure at a lower level of the component hierarchy. Second, the component downloading process should also be subject to transaction - to preserve template integrities in TR if network failure occurs during a template update. Further, as mentioned in Section 4.5, we intend to find a way to specify the state of a component which is to be externalized in an update process; an extension to the CDL language might be a way to tackle this problem.

From the beginning, DCUP architecture has been devised in a way that allows seamless porting to an existing distributed environment. Thus, one of the future steps will be porting DCUP to the CORBA environment. For this, we have identified two potential approaches (Figure 4): (a) To regard the whole CORBA server process as one component. In this server, its CORBA objects implement the services of the component. The CManager, CBuilder, and (potentially) Updater abstractions are specialized CORBA objects running within the same server. Thus, this idea is based on employing a delegation-based approach for associating an object implementation with a particular interface (e.g. the TIE-approach in IONA's Orbix [ORBIXa, ORBIXb]). Therefore, a wrapper's functionality can be implemented by a TIE object (Figure 4a). (b) To distribute every single component into a separate CORBA server process. In this case, the role of wrappers is played by client side proxies, which are controlled by the CManager captured in a standalone CORBA server process (Figure 4b).

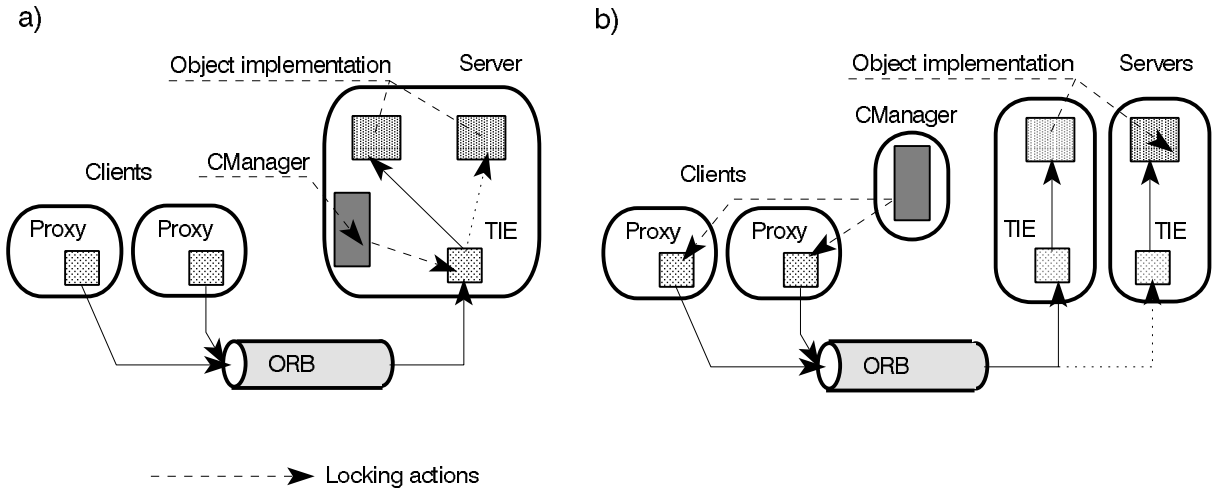

Figure 4 Porting DCUP to CORBA environment

6 Conclusion

This paper presents the principal ideas of the SOFA Architecture and, in particular, gives a comprehensive survey of the SOFA component model extension called DCUP Architecture. The following are the key features of SOFA.

(1) SOFA provides a sound basis for electronic commerce with software components. The business parties are represented by SOFAnodes and are connected into a SOFAnet network. A SOFAnode is a small set of orthogonal abstractions which reflect the roles the parties can take, such as end-user, provider, and retailer. By extending the basic functionality of these abstractions, support for electronic commerce in software component is intended to be achieved (support for component trading, licensing, accounting, and billing in particular). (2) The SOFA component model, in comparison with similar approaches (e.g. [MDK94, BAB96]), clearly (a) separates component interface definition from the description of component structure and bindings, and (b) has a description that reflects component versioning. (3) As for component downloading, SOFA supports both the pull and push models. The push model is intended particularly for achieving "silent" corrections of minor implementation errors.

The DCUP architecture gives the component providers the possibility of updating their components even at runtime without any manual intervention on the end-user side; again, DCUP is based on a small set of orthogonal abstractions - CManager, CBuilder, Wrapper, and Updater. DCUP functionality can be summarized as follows:

(1) A component update request handled in the In part of the associated SOFAnode is forwarded to the Run part to check if an application using this component is running. If so, the corresponding Updater is contacted. The Updater scope abstraction reflects the fact that subtrees of components could have been supplied by separate providers (Section 3.1). (2) Transitions among a component's versions are coordinated by the cooperation of the CManager and CBuilder abstractions. In principle, $CBuilder_i$ is associated with the run of the component's i-th version, while CManager persists over the whole lifecycle of the component (Section 3.4). (3) Transparency of a component's updating with respect to the rest of the application is achieved by limiting the updating effect only to a subtree of the tree-like hierarchy of the components comprising the application. In a component, the use of its services by the outside of the component is mediated via automatically created wrappers, as a result of binding operations (Section 3.3).

As DCUP wrappers resemble proxies, and as a name-based binding is used for accessing component services, it is easy to map a distributed, e.g. CORBA resp. RMI - based [RMI96], application onto the DCUP abstractions. For example, a component can be a CORBA resp. RMI server; the corresponding CManager can ensure the transition from an old to a new version of the server (Section 5).

With respect to updating, the DCUP architecture scales well – mainly for two reasons: First, components, being units of updating as well, can be nested. Thus the part of the application which is to be subject to an update is scalable. In general, the updating granularity can scale anything from minor implementation changes to a major reconfiguration. Second, the DCUP architecture can easily be applied in a distributed (CORBA/RMI) environment in a way which treats a component as a unit of distribution (e.g. a separate CORBA server); thus, updating in DCUP scales well also with respect to distribution of the application.

Acknowledgments

The authors of this paper would like to express their thanks to Volker Tschammer and Gerd Schuerman from GMD Fokus, Berlin for inspiring suggestions on the relation of SOFA to electronic commerce, and to Stefan Tilkov from MLC Systeme, Rattingen for valuable remarks on the DCUP framework architecture. The authors' appreciation goes also to their colleagues Petr Tuma, for fruitful feedback in many discussions, and Nguyen Duy Hoa, for taking part in the prototype DCUP implementation. Finally, the authors are grateful to Anneliese Schauert for proofreading the text.

References

[ATS96] M. Mira da Silva, M. Atkinson: Combining Mobile Agents with Persistent Systems: Opportunities and Challenges, In Proceedings of 2nd ECOOP Workshop on Mobile Object Systems, Linz, July 1996

[BEC96] P. Becker: An Embeddable and Extendable Language for Large-Scale Programming on the Internet, In Proceedings of the 16th ICDCS, 1996

[BAB96] L. Bellissard, S. B. Atallah, F. Boyer, M. Riveill: Distributed Application Configuration, In Proceedings of ICDCS '96, IEEE CS Press 1996

[BWEB97] BackWeb: http://www.backweb.com

[CACM97] CACM Theme Issue on Object-Oriented Application Frameworks. CACM October 97, Vol. 40, No. 9

[CDF97] Channel Definition Format, Microsoft, 1997, http://www.microsoft.com/standards/cdf-f.htm

[CH93] D. J. Chen, S. K. Huang: Interface for Reusable Software Components, In Journal of OO Programming Languages, January 1993

[CO97] D. D. Corkill: Countdown to Success: Dynamic Objects, GBB, and RADARSAT-1, Communications of the ACM, May 1997 / Vol. 40, No. 5

[CPV97] A. Carzaniga, G. Picco, G. Vigna: Designing Distributed Application with Mobile Code Paradigms, In Proceedings of 19th International Conference on Software Engineering, Boston, 1997

[DER96] L. Deri: Droplets: Breaking Monolithic Applications Apart, IBM Research Division, Zurich, April 1996

[EFS97] K.-P.Eckert, M. Festini, P. Schoo, G. Schurmann: TANGRAM: Development of Object-Oriented Frameworks for TINA-C-Based Multimedia Telecommunication Applications, In Proceedings of Third ISADS '97, Berlin, Germany, Apr. 9-11, 1997

[FS96] H. Fossa, M. Sloman: Implementing Interactive Configuration Management for Distributed Systems. In Proceedings of 3rd International Conference on Configurable Distributed Systems (ICCDS), Annapolis, Maryland, May 1996

[FSJ98] M. Fayad, D. Schmidt, R. Johnson (Eds.): Object-Oriented Application Frameworks, Addison-Wesley, 1998 (in print)

[GK96] K. M. Goudarzi, J. Kramer: Maintaining Node Consistency in the Face of Dynamic Change, In Proceedings of 3rd International Conference on Configurable Distributed Systems (ICCDS), Annapolis, Maryland, May 1996

[GV97] C. Ghezzi, G. Vigna: Mobile Code Paradigms and Technologies: A Case Study, In Proceedings of 1st International Workshop on Mobile Agents, MA '97, Berlin, April 1997

[HAP90] C. Hofmeister, J. Atlee, J. Purtilo: Writing Distributed Programs in Polylith, November 1990

[IB96] V. Issarny, Ch. Bidan: Aster: A Framework for Sound Customization of Distributed Runtime Systems, In proceedings of ICDCS '96, IEEE CS Press 1996

[INT97] Intermind: http://www.intermind.com

[JBG97] JavaBeans "Glasgow" specification, http://splash.javasoft.com/beans/glasgow.html

[JBN97] JavaBeans 1.0 Specification, http://splash.javasoft.com/beans/spec.html

[KON96] KONA, http://kona.lotus.com

[KPT96] J. Kleindienst, F. Plasil, P. Tuma: Lessons Learned from Implementing the CORBA Persistent Object Service, In Proceeding of OOPSLA '96, IEEE CS Press 1996

[KPT96a] J. Kleindienst, F. Plasil, P. Tuma: What We Are Missing in the Persistent Object Service, Presented at the OOPSLA '96 Workshop on Objects in Large Distributed and Persistent Software Systems; available at http://nenya.ms.mff.cuni.cz

[LAR96] D. Lange, Y. Aridor: Agent Transfer Protocol, White Paper, http://www.trl.ibm.co.jp/aglets

[LCH96] D. Lange, D. Chang: Programming Mobile Agents in Java, White Paper, http://www.trl.ibm.co.jp/aglets

[MA96] Marimba: The Castanet System, http://www.marimba.com

[MB96] V. Marangozov, L. Bellisard: Component-Based Programming of Distributed Applications, Presented at 3rd CaberNet Radicals Workshop, http://www.twente.research.ec.org/cabernet/research/radicals/1996/papers/comp-marangozov.html

[MDK94] J. Magee, N. Dulay, J. Kramer: Regis: A Constructive Development Environment for Distributed Programs, In Distributed Systems Engineering Journal, September 1994

[MR97] M. Mira da Silva, A. Rodrigues da Silva: Insisting on Persistent Mobile Agent Systems, In Proceedings of 1st International Workshop on Mobile Agents, MA '97, Berlin, April 1997

[MTK97] J. Magee, A. Tseng, J. Kramer: Composing Distributed Objects in CORBA. In Proceedings of the Third International Symposium on Autonomous Decentralized Systems (ISADS), Berlin, April 1997

[NCAS97] Netscape Netcaster: http://www.netscape.com/comprod/products/communicator/netcaster.html

[NT95] O. Nierstrasz, D. Tsichritzis (eds.) : Object-Oriented Software Composition, Prentice Hall, 1995

[OH97] R. Orfali, D. Harkey: Client/Server Programming with JAVA and CORBA, John Wiley & Sons, USA, 1997

[OMG95a] Common Object Request Broker Architecture and Specification Revision 2.0, OMG 97-2-25, 1995

[OMG95b] Object Property Service, OMG TC Document 95-6-1, June 1995

[OMG96a] Merged Trading Object Service submission, OMG 96-05-06, 1996

[OMG96b] Electronic Commerce DTF Reference Model, OMG 96-09-03, 1996

[OMG97a] Persistent State Service, version 2.0, Request For Proposal 97-5-16, May 1997

[OMG97b] CORBA Component Model, Request For Proposal 96-6-12, June 1997

[ORBIXa] Orbix, Programmer's Guide, IONA Technologies Ltd. Dublin, 1994
[ORBIXb] Orbix, Advanced Programmer's Guide, IONA Technologies Ltd. Dublin, 1994
[OSD97] The Open Softwre Desription Format (OSD). NOTE-OSD. W3C 1997, http://www.w3.org/TR/NOTE-OSd
[OSM97] Open Service Model, http://osm-www.informatik.uni-hamburg.de/osm-www/info/index.html
[PTR97] F. Plasil, et al.: SOFAnet and SOFAnode, Basic Functionality, TR 97/12, Dept. of Software Engineering, CharlesUniversity, Prague, 1997
[PNT97] Pointcast: http://www.pointcast.com
[PREE96] W. Pree: Framework Patterns, SIGS Books & Multimedia, 1996
[PT97] P.Tůma: Persistence in CORBA, PhD. Thesis, Dept. of Software Engineering, Charles University, Prague, 1997, available at http://nenya.ms.mff.cuni.cz
[PUR94] J. M. Purtilo: The Polylith Software Bus, ACM Transactions on Programming Languages and Systems, January 1994
[RMI96] RMI - Remote Method Invocation, http://java.sun.com:80/products/jdk/1.1/docs/guide/rmi, 1996
[UNIF97] UNIFACE Seven, Compuware Corporation, 1997, http://www.compuware.com