

# SOFA/DCUP: Architecture for Component Trading and Dynamic Updating

František Plášil<sup>1,2</sup>, Dušan Bálek<sup>1,2</sup>, Radovan Janeček<sup>1,2</sup>

<sup>1</sup> Charles University  
Faculty of Mathematics and Physics  
Department of Software Engineering  
Malostranské náměstí 25, 118 00 Prague 1  
Czech Republic  
{plasil, balek, janecek}@nenya.ms.mff.cuni.cz  
<http://nenya.ms.mff.cuni.cz>

<sup>2</sup> Institute of Computer Science  
Academy of Sciences of the Czech Republic  
Pod vodárenskou věží  
180 00 Prague 8  
Czech Republic  
{plasil, balek, janecek}@uivt.cas.cz

## 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 (SOftware Appliances) architecture, the SOFA component model and its extension, DCUP (Dynamic Component UPdating), are introduced. SOFA and DCUP provide a small set of well scaling orthogonal abstractions (easily mapped to Java and CORBA) 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.*

**Keywords:** software component, dynamic updating, Java, CORBA, MIL, ADL, IDL, electronic commerce

## 1 Introduction

It is generally believed that in the very 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 components in accordance to specific requirements of a particular application and to set the necessary interconnections with other parts of the applications. This issue is also studied in connection with frameworks [11, 45]. What still remains as a particular

challenge is 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 [46]. (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 [17], and it makes it easier to arrange for a reconfiguration of an application [13, 23, 15, 4].

There is a group of works [10, 13, 23, 15, 4] 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 (or 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 [10, 17, 13], 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, resp. 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" [17, 10]; 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 [4] 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 [33], event listener [17]) 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 [26], 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., [17, 18]). Also, to our knowledge, the only standardizing body active in this area that is recognized world wide is the Object Management Group (OMG) [38].

## 1.2 Automated Software Downloading

In 1990 the InterMind [16] 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) [7], which is used by several software companies, e.g., [44, 5]. Other companies have used a similar approach, for example [16, 31, 25]. The standardizing body active in this area is W3C [39].

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 [39], 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 files, etc.).

## 1.3 Challenges

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 (autonomous) 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.

## 1.4 The Goals of the Paper

In this paper, to address the challenges articulated in Section 1.3, we introduce the *SOFA* (*SOFT*ware *Appliances*) *architecture*, the *SOFA component model*, and the SOFA component model extension called *DCUP* (*D*ynamic *C*omponent *U*pside). The abstractions defined here address three areas: (1) the background for electronic commerce in components, (2) the component model, (3) support for dynamic component updating in running applications.

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.

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 and issues mentioned above. These features include:

- SOFA Component model and CDL (Component Description Language) which support versioning by strictly separating the component interface from the component architecture (multiple versions of architecture per an interface).
- Provision of an update operation with clearly defined relationship to versioning.
- SOFAnet and SOFAnode, a small set of well scaling orthogonal abstractions providing a platform for trading with software components over a computer network.

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

- Architectural support for safe component state transition during dynamic updates (state transitions executed at well defined moments).
- Localization of updating effects (while updating of a component is in progress, the execution of the remaining parts of the running application is affected as minimally as possible).
- Dynamic updates done with no human intervention at the subscriber (end-user) side.

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 reflects the experience 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

In SOFA, an application is viewed as a hierarchy of software components. 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). In [27], our "template" would be referred to as a "component type". Basically a template is a framework which contains definitions of *implementation objects* and *nested components* (Figure 1). Every template is determined by its interface (set of services either *provided* or *required*), and by the definitions and *bindings* of implementation objects and nested components. The Darwin/Olan-like [23, 10, 13, 24, 4] *SOFA CDL* (Component Description Language) developed as part of the SOFA project is used to specify a component's interface and its architecture (in terms of nested components and their interconnections).

In the world of SOFA, every template is uniquely identified by a triple  $\langle \text{provider\_name, type\_name, version} \rangle$  called its *Template Marker (TM)*.

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*.

In a template  $t$ , a nested component  $nc$  is *defined* by embedding of a template  $t'$  into  $t$  under the name  $nc$  (for brevity we also say that  $t$  contains the  $nc$  component definition based on  $t'$ ). Naturally, creation of the  $nc$  component takes place at the moment of instantiation of  $t$ .

Specific to the SOFA component model is the concept of *updating*: In a template  $t$ , an  $nc$  component definition based on  $t'$  can be modified (*updated*) by basing it on another template  $t''$  which is compatible to  $t'$ . The updating operation preserves binding inside of  $t$ . There

may be several models of template *compatibility*. At present we assume that  $t''$  is compatible to  $t'$  if  $t''$  supports at least the same interface as  $t'$ .

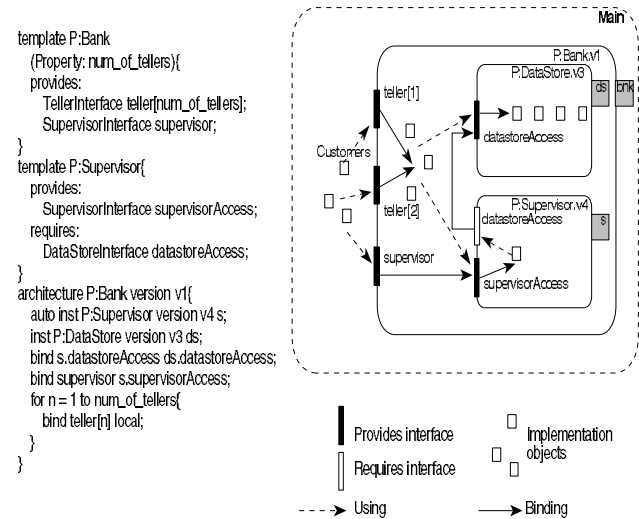


Figure 1 Example of a SOFA template

To reflect the challenge of autonomous updating (Section 1.3) two modes of component definition are introduced: *subordinate* and *autonomous* (note the *auto* clause in Figure 1). Basically, an autonomous component definition is together with all its nested subordinate components (transitively) a unit of updating in the following sense: a) If  $nc$  is a subordinate component definition, an update of  $nc$  modifies the version of its parent template, transitively until a parent template  $r$  embedded in autonomous mode is found (the version of  $r$  is also modified);  $r$  is an *updating root*. b) If  $nc$  is the target of an update operation, it is in this operation identified by its updating root  $r$  and the name sequence of the nested (subordinate) component definitions reflecting the path from the  $r$  to  $nc$ . By definition, all primary templates are considered to be autonomous component definitions.

### 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. The base of a *SOFAnode* is composed by its *In*, *Out*, *Made*, *Run*, and *Template Repository (TR) parts*. More precisely, 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  $A \neq B$ .

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. The motivation for introducing the particular parts of the SOFAnode is illustrated by the following special cases (Figure 2): a "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.

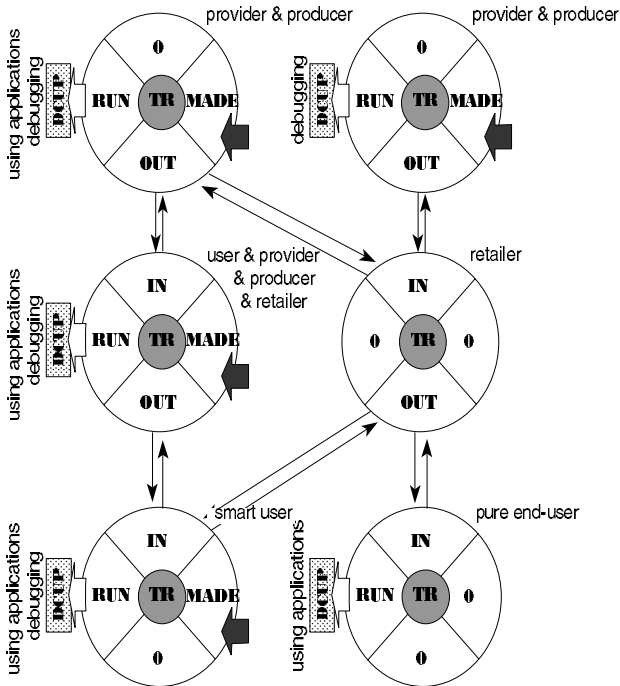


Figure 2 SOFAnet example

### 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 ClassLoaders). 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 [21], and by our experience in designing and implementing the CORBA Persistent Object Service [19, 20, 49].

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	Serves as an "entry" point to the SOFAnode. It handles the incoming requests (from other SOFAnodes) for installation of new templates and updates of exiting ones. It supports both push and pull models for template downloading. It can be extended to support electronic commerce in components (e.g., support for component trading, licensing, accounting, and billing and payment).
Out	Serves as the "output" point of the SOFAnode for transferring templates and update requests to other SOFAnodes (in both push and pull models). It can be extended 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 developers.
Run	Provides environment for launching and running of applications by instantiating of templates. It may provide other support for the running applications, e.g., access to persistent datastores.

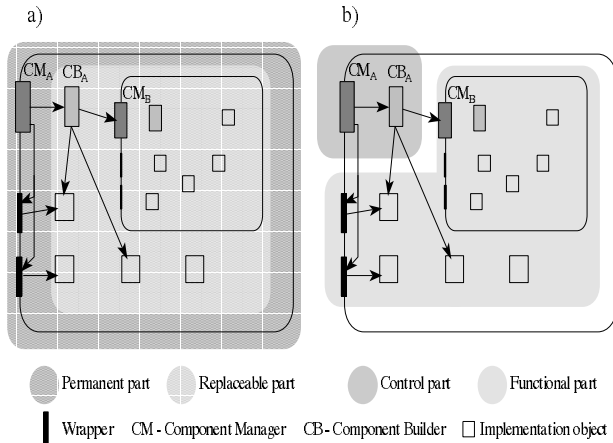
Table 1 Basic functionality of SOFAnode parts

#### 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* (Figure 3a). Orthogonally, with respect to the nature of the operation provided, the component is divided into a *functional part* and a *control part* (Figure 3b). 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, ClassLoaders, 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).



**Figure 3 Structure of a component**

In an application, a component *C* has an *Updater* associated with its *CManager* (there is one-to-one correspondence), if *C* is an instantiation of a component definition which is an updating root (Section 2.1). The role of an *Updater* is to accept updating requests coming from the *Run* part of the *SOFA*node and to forward them to the corresponding *CManager*.

*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.

In Java, whenever an application needs to create a new instance of a class that has not been loaded yet to the runtime, a *ClassLoader* is asked to load it. Particularly, DCUP takes the advantage of (a) the option to define "smart" *ClassLoaders* with a specific behavior (a DCUP *ClassLoader* can load dynamically determined code from *SOFA*node's *TR*); (b) the fact that a class *C* in JVM (Java Virtual Machine) is identified by its name prefixed by the identification of the class loader used for loading of *C*. In a DCUP application, each version of the replaceable part of a component is loaded via its own *ClassLoader* (one-to-one relationship). This eliminates class name clashes in coexisting components.

### 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), 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<sub>i</sub>.onArrival()* method creates the internal objects and the subcomponents of *RP<sub>i</sub>*, 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<sub>i</sub>* (among the internal objects, subcomponents, etc.). We say that *CBuilder<sub>i</sub>.onArrival()* builds *RP<sub>i</sub>*. To terminate *RP<sub>i</sub>*, the *CBuilder<sub>i</sub>.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<sub>i</sub>*.

### 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).

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*.

### 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  $RP_1, RP_2, \dots, RP_n$ , 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  $RP_i \rightarrow RP_{i+1}$  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,
                   TTemplateM newTMarker,
                   TStorage StateStore) {
        // if ComponentName is the name of a
        // subcomponent, then delegate this call
        // to the corresponding CManager, else:
        OldBuilder.onLeaving(StateStore);
        NewBuilder = ClassLoader.newBuilder();
        NewBuilder.onArrival(StateStore);
    }
}
```

### 3.5 Run part - DCUP Application Interplay

Important interactions between an application and the SOFANode are launching, Updater registering, component updating, and application terminating. These operations are atomic and mutually exclusive. For example, it is impossible to terminate an application during an update of its internal component.

The Run part of a SOFANode keeps the list of the primary templates (stored in TR) which can be launched via this SOFANode as applications. To start an application, the Run part asks TR for the application's main class and executes it as a new process (e.g., web browser for applets, 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. To be able to receive update messages, each of the Updaters in the application 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 (in cooperation with TR). The Run part forwards the updating information to the corresponding Updaters.

An additional role of the Run part is controlling component state externalization that is used to preserve (and potentially transform) a component's state during the update process. Using this feature, the whole application can store its state before a user terminates it.

## 4 Case Study of DCUP Use

By presenting an example (also implemented as a proof-of-the-concept), this section illustrates how the basic DCUP ideas and concepts can be applied in an (almost) real-world situation. Consider the following scenario: 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. A customer can ask supervisor as well. Both the teller and the supervisor need access to an account repository. As the number of customers of this bank grows, we may realize that the current implementation of account repository does not satisfy the original requirements. We would like to achieve updating of our real-time application without shutting it down.

To model this hypothetical bank, we introduce the following templates: Bank, Supervisor, and DataStore. 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 1.

### 4.1 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  $createComponent()$  method creates a new instance of ClassLoader that is able to obtain classes of corresponding component version from SOFANode. 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 1, TMainBuilder creates the Bank component and a number of the TCustomer objects. The  $onArrival()$  method might look like the following:

```
// in TMainBuilder class
public void onArrival(TStorage StateStore){
    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); ...

```

## 4.2 Creating other components

The creation of all other subcomponents follows the same scheme as the creation of MainComponent (except 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 like the following:

```

// in the TBankBuilder class (instantiated as a
BankCB)
public void onArrival(TStorage StateStore){
    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. 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. Every Teller is provided with references to the services provided by the DataStore and Supervisor components; this ends onArrival() of BankCB:

```

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

```

## 4.3 Updating

The Updater associated with MainCM waits for an update message from the SOFAnode Run part. After it receives an update message, it further calls updateComponent() upon its CManager. Thus, for

example, the DataStore component could be updated as follows:

```

MyCM.updateComponent("Bank.DataStore",
    newTMarker, StateStore);

```

The first string parameter denotes the DataStore component within the Bank component. The second parameter represents the Template Marker of the new component version, and the last one represents storage used for externalizing the DataStore component's state. Note that updating can be done without the tellers and customers being aware of it.

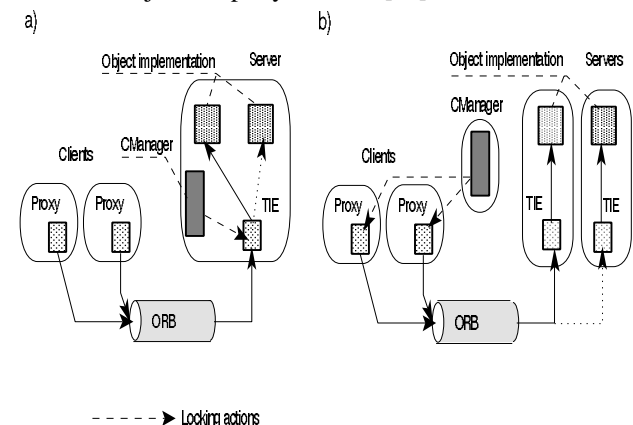
## 5 Current State and Future Intentions

So far we have finished a prototype implementation of SOFAnode. Also a CDL compiler used for generating DCUP components' source code skeletons is available.

The prototype includes implementation of a Template Repository with storing, versioning, and runtime support facilities, a Run part with its user interface to provide basic support for launching applications, and In and Out parts (including user interfaces). Based upon TCP/IP, the communication protocol among the In and Out parts is currently limited to the transferring of components. Business layers have not been integrated to SOFAnode yet. Details of our prototype implementation of SOFAnode can be found in [43].

Our future intentions include enhancing support for state externalization, component versioning, elaboration of the In-Out communication protocol, employing transactional service in updating process, elaborating of electronic commerce support, and implementing the DCUP architecture on the CORBA platform.

To support externalization of a component state during updates, the key issue 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, similar to the CORBA Object Property Service [34].



**Figure 4 Porting DCUP to CORBA environment**

As for versioning, three issues are to be elaborated: (a) capturing the version concept based on an n-tuple of (component) properties; (b) defining a set of operations

reflecting changes in a version as a consequences of property modification; (c) finding "reasonable" (component) compatibility rules.

Furthermore, our In-Out communication protocol is not mature enough to reflect all of the SOFA needs and is to be enhanced in a CDF (Channel Definition Format) [7] way to allow for transferring of other data types (in addition to templates), for scheduling of updates, etc.

A transactional service is to be employed 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 in order to preserve template integrities in TR if network failure occurs during a template update.

As for electronic commerce, 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 [35]), licensing, accounting, and billing. Several approaches come to mind ranging from activities of standardizing bodies (OSM [40], OMG Electronic Commerce Domain TF [36]; unfortunately, none of these activities has yet reached a mature specification), over TINA/TANGRAM [9], to a "proprietary" approach.

From the beginning, the 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 [6]). 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).

## 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 the 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 components is intended to be achieved (in particular, support for component trading, licensing, accounting, and billing). (2) The SOFA component model, in comparison with similar approaches (e.g., [23, 4]), clearly (a) separates component interface definitions from the description of component architectures, (b) reflects component versioning, and (c) defines an update operation with simple semantics which helps limit the updating effect to a subtree of the component hierarchy. (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.

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, Ratingen, for valuable remarks on the DCUP framework architecture. The authors' appreciation goes also to their colleagues Petr Tůma, for fruitful feedback in many discussions, and to Nguyen Duy Hoa, Radek Pospíšil, and Marek Procházka for taking part in the prototype implementation. Finally, the authors are grateful to Anneliese Schauert and Peter G. Anderson for proofreading the text.

## References

- [1] B. Agnew, Ch. Hofmeister, J. Purtilo: Planning for Change: A Reconfiguration Language for Distributed Systems. In Proceedings of the Second International Conference on Configurable Distributed Systems, 1994
- [2] Robert J. Allen: A Formal Approach to Software Architecture. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1997
- [3] R. Allen, D. Garlan: Specifying Dynamism in Software Architectures. In Proceedings of the Foundations of Component-Based Systems Workshop, 1997

- [4] L. Bellissard, S. B. Atallah, F. Boyer, M. Riveill: Distributed Application Configuration. In Proceedings of the Sixteenth International Conference on Distributed Computing Systems, 1996
- [5] BackWeb: <http://www.backweb.com>
- [6] S. Baker: CORBA Distributed Objects Using Orbix, Addison-Wesley, 1997
- [7] Channel Definition Format, Microsoft, 1997, <http://www.microsoft.com/standards/cdf-f.htm>
- [8] A. Carzaniga, G. Picco, G. Vigna: Designing Distributed Application with Mobile Code Paradigms. In Proceedings of the Nineteenth International Conference on Software Engineering, 1997
- [9] 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 the Third International Symposium on Autonomous Decentralized Systems, 1997
- [10] H. Fossa, M. Sloman: Implementing Interactive Configuration Management for Distributed Systems. In Proceedings of the Third International Conference on Configurable Distributed Systems, 1996
- [11] M. Fayad, D. Schmidt, R. Johnson (Eds.): Object-Oriented Application Frameworks, Addison-Wesley, 1998 (in print)
- [12] D. Garlan, R. Monroe, D. Wile: ACME: An Architecture Description Interchange Language. In Proceedings of CASCON '97, 1997
- [13] K. M. Goudarzi, J. Kramer: Maintaining Node Consistency in the Face of Dynamic Change. In Proceedings of the Third International Conference on Configurable Distributed Systems, 1996
- [14] C. Hofmeister, J. Atlee, J. Purtilo: Writing Distributed Programs in Polyolith. CS-TR-2575, Dept. of Computer Science, University of Maryland, 1990
- [15] V. Issarny, Ch. Bidan: Aster: A Framework for Sound Customization of Distributed Runtime Systems. In Proceedings of the Sixteenth International Conference on Distributed Computing Systems, 1996
- [16] Intermind: <http://www.intermind.com>
- [17] JavaBeans 1.0 Specification, <http://splash.javasoft.com/beans/spec.html>
- [18] KONA, <http://kona.lotus.com>
- [19] J. Kleindienst, F. Plasil, P. Tuma: Lessons Learned from Implementing the CORBA Persistent Object Service. In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA '96, 1996
- [20] J. Kleindienst, F. Plasil, P. Tuma: What We Are Missing in the Persistent Object Service. Presented at the Workshop on Objects in Large Distributed and Persistent Software Systems, OOPSLA '96, 1996, <http://nenya.ms.mff.cuni.cz>
- [21] D. Lange, D. Chang: Programming Mobile Agents in Java. White Paper, <http://www.trl.ibm.co.jp/aglets>
- [22] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, W. Mann: Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering, 21(4), 1995
- [23] J. Magee, N. Dulay, J. Kramer: Regis: A Constructive Development Environment for Distributed Programs. In Distributed Systems Engineering Journal, 1(5), 1994
- [24] J. Magee, A. Tseng, J. Kramer: Composing Distributed Objects in CORBA. In Proceedings of the Third International Symposium on Autonomous Decentralized Systems, 1997
- [25] Marimba: The Castanet System, <http://www.marimba.com>
- [26] V. Marangozov, L. Bellissard: Component-Based Programming of Distributed Applications. Presented at the Third CaberNet Radicals Workshop, 1996, <http://www.twente.research.ec.org/cabernet/research/radicals/1996/papers/comp-marangozov.html>
- [27] Neno Medvidovic: A Classification and Comparison Framework for Software Architecture Description Languages. TR UCI-ICS-97-02, Dept. of Information and Computer Science, University of California, Irvine, 1996
- [28] Nenad Medvidovic, Peyman Oreizy, Richard N. Taylor: Reuse of Off-the-Shelf Components in C2-Style Architectures. In Proceedings of the Symposium on Software Reusability, 1997
- [29] R. Mili, A. Mili, R. Mittermeir: Storing and retrieving Software Components: A Refinement Based System. IEEE Transactions on Software Engineering, 23(7), 1997
- [30] M. Mira da Silva, A. Rodrigues da Silva: Insisting on Persistent Mobile Agent Systems. In Proceedings of the First International Workshop on Mobile Agents, 1997
- [31] Netscape Netcaster, <http://www.netscape.com/comprod/products/communicator/netcaster.html>
- [32] O. Nierstrasz, D. Tschritzis (eds.): Object-Oriented Software Composition. Prentice Hall, 1995
- [33] Common Object Request Broker Architecture and Specification, Revision 2.0. OMG 97-2-25, 1995
- [34] Object Property Service. OMG 95-06-01, 1995
- [35] Merged Trading Object Service subm. OMG 96-05-06, 1996
- [36] Electronic Commerce DTF Reference Model. OMG 96-09-03, 1996
- [37] Persistent State Service, version 2.0. RFP 97-5-16, 1997
- [38] CORBA Component Model. RFP 96-6-12, 1997
- [39] The Open Software Description Format (OSD). NOTE-OSD. W3C 1997, <http://www.w3.org/TR/NOTE-OSD.HTML>
- [40] Open Service Model, <http://osm-www.informatik.uni-hamburg.de/osm-www/info/index.html>
- [41] Peyman Oreizy: Issues in the Runtime Modification of Software Architectures. UCI-ICS-TR-96-35, Dept. of Information and Computer Science, University of California, Irvine, 1996
- [42] Peyman Oreizy, Nenad Medvidovic, Richard N. Taylor: Architecture-Based Runtime Software Evolution. To appear in the Proceedings of the International Conference on Software Engineering, 1998
- [43] F. Plasil, D. Balek, R. Janecek, R. Pospisil, M. Prochazka: SOFAnet and SOFAnode, Basic Functionality. TR 97/12, Dept. of Software Engineering, Charles University, Prague, 1997
- [44] Pointcast, <http://www.pointcast.com>
- [45] W. Pree: Framework Patterns. SIGS Books & Multimedia, 1996
- [46] J. M. Purtilo: The Polyolith Software Bus. ACM Transactions on Programming Languages and Systems, 16(1), 1994
- [47] M. Shaw, D. Garlan: Formulations and Formalisms in Software Architectures. In LNCS 1000, Springer, 1995
- [48] C. Szyperski: Component Software, Beyond Object-Oriented Programming. Addison-Wesley, 1997
- [49] P. Tuma: Persistence in CORBA. PhD. Thesis, Dept. of Software Engineering, Charles University, Prague, 1997, <http://nenya.ms.mff.cuni.cz>
- [50] A. M. Zaremski, J. M. Wing: Specification Matching of Software Components. ACM Transactions on Software Engineering and Methodology, 6(4), 1997