Software Connectors: A Hierarchical Model

Dušan Bálek[1], František Plášil[1,2)]

[1] Charles University, Faculty of Mathematics and Physics, Department of Software Engineering,
Malostranské nám•stí 25, 118 00 Prague 1, Czech Republic
{balek, plasil}@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz

[2] Academy of Sciences of the Czech Republic, Institute of Computer Science,
Pod vodárenskou v•ñí 2, 180 00 Prague 8, Czech Republic
plasil@cs.cas.cz, http://www.cs.cas.cz

Abstract

To support rapid software evolution, it is desirable to construct software systems from reusable components. In this approach, the architecture of a system is described as a collection of components along with the interactions among these components. Whereas the main system functional blocks are components, the properties of the system also strongly depend on the character of the component interactions. This fact gave birth to the "connector" concept which is an abstraction capturing the nature of these interactions. The problem tackled in this paper is that even though the notion of connectors originates in the earliest papers on software architectures [24, 18], connectors are currently far from being a typical first class entity in the contemporary component-based systems.

The authors address the problem by (1) bringing an additional argument in favor of considering connectors as first class entities representing component interactions and by (2) introducing a connector model reflected at all the key stages of an application's development: ADL specification, deployment, and implementation.

By articulating the "deployment anomaly", the paper identifies the role connectors should play when the distribution and deployment of a component-based application is considered. Based on basic connector elements (both predefined and user-defined), a connector can be (at least) partially generated from its ADL generic description and from the information on deployment of the components it connects. This way, the underlying code of a component-based application is advantageously factored into the deployment-neutral part (comprising the "real" application functionality located in components) and deployment-sensitive part (embodied in connectors) which can be regenerated anytime the deployment of the application is modified. As a proof of the concept, a case study is provided, illustrating how the connector model can be integrated into the SOFA/DCUP component model.

1. Introduction

A few years ago, the trend to construct software systems as a collection of cooperating reusable components emerged and has become widely accepted since. Influenced by the academic research projects focused on components [9, 23, 25, 8, 1, 17, 6], several industrial systems on the market [26, 27, 15, 16, 29] advertise support of component technology.

As for *components*, there is a broad agreement on grasping them as reusable black/grey-box entities with well-defined interfaces and specified behavior. Usually, a component can have multiple interfaces; some of them to provide services to the component's clients, other to require services from the surrounding environment. Components can be nested to form hierarchies; a higher-level component can be composed of several mutually interconnected, cooperating subcomponents. Serving as tools for specifying component

This work is partially supported by the Grant Agency of the Academy of Sciences of the Czech Republic (project number A2030902) and the Grant Agency of the Czech Republic (project number 201/99/0244).

interfaces and architecture, a number *architecture description languages (ADLs)* [9, 25, 1, 17, 13] have been designed.

The notion of a connector can be found in many papers on software architectures [25, 1, 17, 12, 2]. Typically, a *connector* is an architectural element that reflects the specific features of interactions among components in a system. Even though the notion of a connector originates in the earliest papers on software architectures [24, 18], no widespread consensus on how to incorporate it into the existing program development systems and languages has been reached until present.

1.1. Connectors: overview and related work

Nowadays, it is generally accepted that understanding of a system architecture can be improved by a precise specification of the interactions among the system's components. However, there is no consensus on the form of such a specification. In the related work [9, 25, 1, 17, 13], the following three basic approaches to specifying component interactions can be identified. In compliance with the terminology coined in [2, 11], these are: (1) *implicit connections,* e.g., in the Darwin language, (2) *an enumerated set of built-in connectors,* e.g., in the UniCon language, and (3) *user defined connectors,* e.g., in the Wright language.

The Darwin language [9] is a typical representative of ADLs that use implicit connections. The connections among components are specified in terms of direct bindings of *requires* and *provides* interfaces. The semantics of a connection is defined by the underlying environment (programming language, operating system, etc.), and the communicating components should be aware of it (to communicate, Darwin components directly use ports in the underlying Regis environment).

The main drawback of the ADLs using implicit connections is the following. When specifying interactions among components in a system, it is convenient to concentrate the specification of a particular interaction into a single abstraction unit (rather than spread it over the specifications of all components involved in the interaction). Such an abstraction unit is usually called a *connector* [25, 24, 1, 17]. In addition to making system maintenance easier, loose coupling of components via connectors has also other benefits: direct support for distribution, location transparency and mobility of components in a system, support for dynamic changes in the system's connectivity, etc. The detachment of the communication mechanisms from the component specification can also increase reusability (the same component can be used in a variety of environments, each of them providing specific communication primitives).

The UniCon language [25] is a representative of ADLs where connectors are first class entities. In this language, a developer is provided with a selection of several predefined built-in connector types that correspond to the common communication primitives supported by the underlying language or operating system (such as RPC, pipe, etc.). The semantics of a particular interaction is defined by the connector type selected to reflect this interaction. The most significant drawback of a UniCon-like ADL is that the predefined set of connector types reduces the range of possible component interactions that can be expressed by the language (there is no way to capture any interaction among components that does not correspond to a predefined connector type).

User defined connectors, the most flexible approach to specifying component interactions, are employed, e.g., in the Wright language [1]. The interactions among components are fully specified by the user, i.e. the system developer. Every interaction in a system is represented by an instance of a connector type. Complex interactions can be expressed by nested connector types. To specify the protocols of component interactions, Wright uses a modified Hoare's CSP notation [5]. The main drawback of the Wright language is the absence of any guidelines as to how to realize connectors in an implementation. (In Wright, connectors exists at the specification level only, which results in the problem of how to correctly reflect the specification of a connector in its implementation.)

Recently, based on a thorough study of existing ADLs, Medvidovic et al. [12] presented a classification framework and taxonomy of software connectors. This taxonomy is an important attempt to improve the current level of understanding of what software connectors and their main building blocks are. Not addressing all the issues of designing connector types, it is focused mainly on classification (and therefore better comprehension) of connector types. In addition, the selection of basic connector types in [12] may be questionable, as not all of them seem to be at the same abstraction level (e.g., adaptor, arbitrator and distributor vs. procedure call, event, and stream).

1.2. Challenges and the goals of the paper

The common objection against considering connectors as separate first class entities is that component interactions can be expressed by dedicated “communication” components; in other words, that there is no need for another abstraction to represent component interactions. However, those opposing this view argue that there are intrinsic differences between a “communication” component and an “ordinary” component. These differences, emphasizing arguments in favor of separating these two abstraction, include: (1) Difference in lifecycle (an ordinary component encapsulates a specific functionality and, typically, can execute autonomously, while a communication component exist only to serve the interaction needs of other components [17, 7]). (2) Difference in component types’ genericity (communication component types inherently have to allow for a type parameterization in component interfaces [24]).

The first goal of the paper is to bring an additional argument in favor of considering the communication components as separate first class entities - connectors; this argument, articulated as *deployment anomaly*, is based on showing that there is an important difference between deployment of an ordinary component and of a communication component, leading to the necessity to treat these two concepts as separate first class entities.

As [12] indicates, a variety of possible interactions among components exist. The complexity of some interactions far exceeds the description abilities of current ADLs. An effective way to understand and describe a complex interaction is to factor it (recursively) into parts covering different (and possibly orthogonal) aspects of its nature. The elementary parts identified in this factorization should be simple enough to be implemented and reasoned about. Similarly, the connector type representing an interaction should be constructed as a hierarchy of internal elements reflecting the interaction’s factorization. The second goal of the paper is to capture this idea in a connector model.

The paper has the following outline. Reviewing the basic ADL concepts, Section 2 briefly introduces the SOFA component model also used in an evolving example throughout the paper. In Section 3, the deployment anomaly is introduced. The set of basic connector tasks and requirements is identified and studied in Section 4. A new connector model is proposed in Section 5. As a proof of the concept, it is integrated into the SOFA/DCUP component model in Section 6. Finally, the main achievements and future intentions are summarized in the concluding Section 7.

2. A component model and case study

2.1. SOFA/DCUP component model - the basics

In SOFA, an application is viewed as a hierarchy of software components. Analogous with the classical concept of an object as an instance of a class, a *software component* is an instance of a *component template* (*template* for short). In principle, a template is a component type defined by a pair <component frame, component architecture>. The *component frame* of a template T defines the interfaces of T's instances. These interfaces specify the sets of services either *provided* or *required*. Simply, the component frame provides a black-box view of T's instances, while the *component architecture* provides a gray-box view of each of T’s instances by describing its internal structure in the terms of its direct subcomponents and their interactions (interface ties). A component architecture can be specified as *primitive* which means that there are no subcomponents and the component frame is directly implemented in the underlying implementation language. If a component C is an instance of T which is based on a primitive architecture, we say that C is a *primitive component*, otherwise C is a *composed component.* Templates are specified in the *SOFA CDL* (Component Definition Language) [13]. The DCUP architecture is a specific architecture of SOFA components which allows for their safe updating at runtime [19].

2.2. Case study: Banking demo

In this section, we illustrate the basic SOFA concepts on a simple example - the BankingDemo application. Consider a bank in which a number of tellers serve a potentially huge number of customers. Each customer requests a teller to perform a desired financial transaction(s) on an account(s). Certain transactions, such as an overdraft, require the teller to ask the supervisor for an approval. On demand, the construction and activity of the running Bank Demo application can be monitored via a specialized visualization tool. Each of this entities can be modeled as a component (Figure 1).

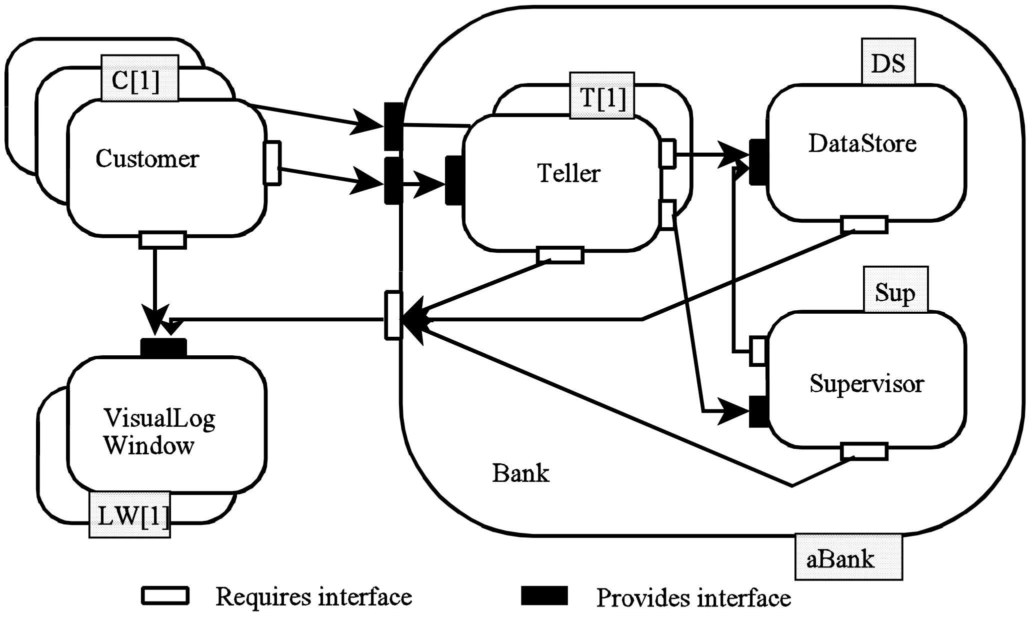

Figure 1 BankingDemo Architecture

The core of the application is an instance (named aBank) of the Bank template (Bank component for short). The Bank component internally contains an array of Teller subcomponents(T[1], T[2], ... , T[N]), the Supervisor subcomponent, and the DataStore subcomponent. As illustrated in the following CDL specification fragment, the Bank component provides NoT instances of TellerInterface, each of them being tied (delegated) to the TellerInterface of a Teller subcomponent.

The remaining part of the application is formed by the Customer and VisualLogWindow components, the latter serving for system administration purposes. The Customer components model the behavior of bank customers by requesting randomly chosen bank tellers to perform transactions on accounts. The Customer components also send event messages to those VisualLogWindow components that subscribed to receive them. The communication of the Customer components with the Bank components is based on procedure calls, while all the interaction with the VisualLogWindow components is based on event delivery. As an aside, it is assumed that the only source of active threads in the application are the Customer components.

The following piece of code illustrates the core of the Bank's CDL description. (Similar to CORBA IDL, the interface type specifications are omitted.)

```
// a frame X specifies instances of
// provides and requires interfaces of X
frame Bank {
  provides:
    TellerInterface Teller[1..NoT];
  requires:
    LogInterface LogWindow;
};
frame DataStore {
  provides:
    DataStoreInterface DataStore;
  requires:
    LogInterface LogWindow;
};
frame Supervisor {
  provides:
    SupervisorInterface Supervisor;
  requires:
    DataStoreInterface DataStore;
    LogInterface LogWindow;
```

```
};
frame Teller {
  provides:
    TellerInterface Teller;
  requires:
    DataStoreInterface DataStore;
    SupervisorInterface Supervisor;
    LogInterface LogWindow;
};
frame Customer {
  requires:
    TellerInterface BankTeller;
    LogInterface LogWindow;
};
frame VisualLogWindow {
  provides:
    LogInterface LogWindow;
};

// an architecture X version "n" defines the implementation version n of frame X
// at the first level of component nesting
architecture DataStore version "1.0" primitive;
architecture Supervisor version "1.0" primitive;
architecture Teller version "1.0" primitive;
architecture Customer version "1.0" primitive;
architecture VisualLogWindow version "1.0" primitive;

architecture Bank version "1.0" {
// an inst TF X specifies the instance (named X) of a subcomponent determined
// so far by the frame TF (the architecture of X will be determined at the
assembly
// time (Section 3.1)
  inst DataStore DS;
  inst Supervisor Sup;
  inst Teller T[1..NoT];
// subcomponent ties (bind, delegate, and subsume);
// bind = provides interface instance to requires interface instance
  bind Sup.DataStore to DS.DataStore;
  bind T[1..NoT].DataStore to DS.DataStore;
  bind T[1..NoT].Supervisor to Sup.Supervisor;
// delegate = provides to provides
  delegate Teller[1..NoT] to T[1..NoT].Teller;
// subsume = requires to requires
  subsume DS.LogWindow to LogWindow;
  subsume Sup.LogWindow to LogWindow;
  subsume T[1..NoT].LogWindow to LogWindow;
};

// architecture of the application, implements the frame System:
system architecture BankingDemo version "1.0" {
  inst Bank aBank;
  inst VisualLogWindow LW;
  inst Customer C[1..NoC];
  bind C[1..NoC].Teller to aBank.Teller[?];
  bind aBank.LogWindow to LW.LogWindow;
  bind C[1..NoC].LogWindow to LW.LogWindow;
};
```

3. The role of deployment in application/component lifecycle

3.1. Component lifecycle

The lifecycle of a component is characterized by a sequence of design time, deployment time, and run time phases (potentially repeated due to design revisions and maintenance). In a more detailed view, a design time phase is composed of the following design stages: development and provision, assembly, and distribution.

Design time

Development and provision. At this stage, a component is specified in CDL by its frame and potentially several architectures, each of them being a design version of the frame. More formally, several templates based on the same frame are developed and provided, forming a set of templates {<F,A_1>, <F,A_2>, ... ,<F,A_n>} as illustrated in Figure 2. Any template with primitive architecture (e.g. DataStore, Supervisor, and Teller in BankingDemo) has to be accompanied by its implementation in the underlying implementation language. The actual specification of an architecture A is always based on the frames of A's subcomponents (and not on the architecture of those subcomponents). For instance, in Figure 2, the frame F_{Sub2} is implemented by two different architectures: A_{Sub2} and A_{Sub3}. While A_{Sub3} is primitive, A_{Sub2} is composed of two subcomponents Sub_{21} and Sub_{22}; the employment of these subcomponents in A_{Sub2} is determined at the level of their frames F_{Sub21}, F_{Sub22}. This way, the specification of an application is factored into a hierarchy of alternating layers "frame – architecture – frame – … ". In principle, the topmost frame, F_{System}, defines the requires interfaces reflecting the system services available to all applications in the underlying system environment (referred to as "deployment dock" in Section 3.2), and the provides interfaces via which the system environment can control an application, e.g. its run time lifecycle stage.

Assembly. The goal of this design stage is to assemble the executable form of the component based on a frame F. Assembling an executable form of the $Application^i$ means reducing its development tree (Figure 2) in such a way that each frame node has only one successive architecture node. This process starts at F_{System} with choosing one particular template $<F_{System}, A_{Applicaton}i>$. If $A_{Application}i$ is not primitive, such template selection is applied recursively to all frames involved in $A_{Application}i$. This process ends by creating an *assembled tree* of $Application^i$ which represents a particular version of this application. Consequently, an executable form of the application/component is based on all the primitive architectures involved recursively in the respective assembled tree of $Application^i$.

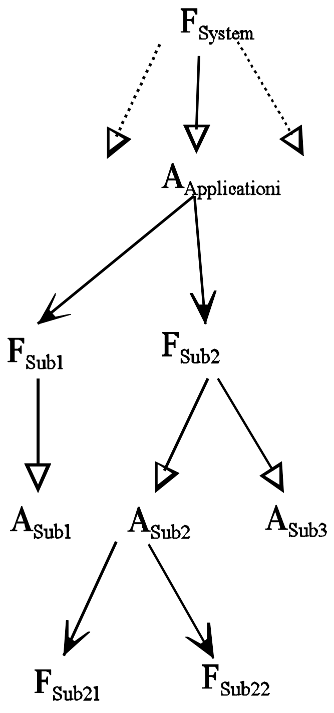

Figure 2 A hierarchy of frame/architecture

Distribution. An application composed of components can be distributed. To address distribution, the goal of this design stage is to divide the assembled tree of $Application^i$ into *deployment units*. The components forming a deployment unit are to be submitted to a single deployment dock for instantiation. In principle, a *deployment dock* serves as a component factory and controls the lifecycle of a running component. For example, a deployment dock can be based on Java Virtual Machine with a ClassLoader modified to launch components, an EJB container [27], a CORBA implementation repository, etc. As an aside, we assume that a deployment unit can contain just a single primitive component in a simple case, and a primitive component cannot be divided into more deployment units since its internals are invisible.

Deployment time

The goal of the deployment time phase is to achieve **deployment** of an $Application^i$, i.e. to associate each of the deployment units of this application with a real deployment dock (e.g., identified by its URL), submit all the components in the application to relevant deployment docks, and let these deployment docks start the application.

Run time

In a run time phase of a component's lifecycle, execution of the component code takes place (in the general case with subcomponents deployed over a network). The key executable code implementing the functionality of the component is located in those of its subcomponents the architecture of which is primitive.

3.2. Deployment anomaly

As to the distribution and deployment of a composed component, the following two approaches are to be considered: (1) A deployment unit boundary ("deployment boundary" for short) cannot cross a frame boundary. Thus, in the case of composed components, deployment boundaries can cross the component interface ties, but not the component/frame boundaries (Figure 3). Moreover, the deployment description of composed/nested components can be done advantageously on a top-down basis, following the hierarchy of components.

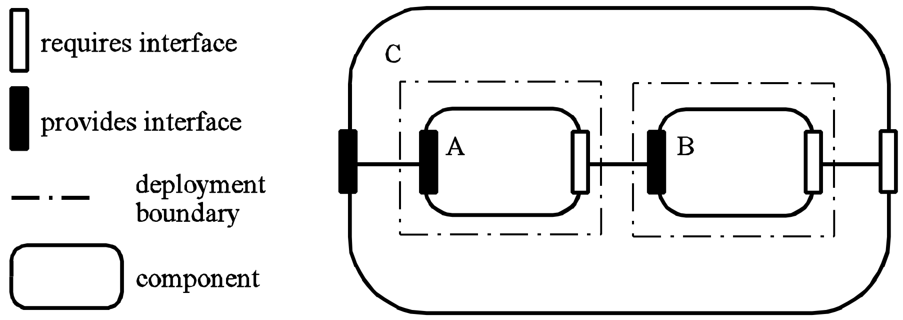

Figure 3 Deployment boundaries crossing interface ties

(2) Deployment boundaries are orthogonal to component/frame boundaries. Thus, deployment boundaries can cross a component/frame boundary (recall that a primitive component cannot be divided into more deployments units). On Figure 4, the respective alternatives are analyzed given the components C, A, B and their ties from Figure 3. In the a), b) and c) cases, the deployment boundary crosses the internal interface ties. Consider now that A is composed; in such a case, d) leads recursively to one of the alternatives a) - d) since the distribution boundary crosses A's boundary. If A is primitive, it cannot be divided into more deployment units, i.e. d) is not possible, and therefore the deployment boundaries can cross component ties only - similarly to (1). This reasoning implies that there is no difference between (1) and (2) in terms of deploying primitive components. As to composed components, the following two problems are not easy to overcome: (i) the deployment description cannot parallel the hierarchy of component nesting, and (ii) the deployment of a composed component into more deployments docks may be a complex process.

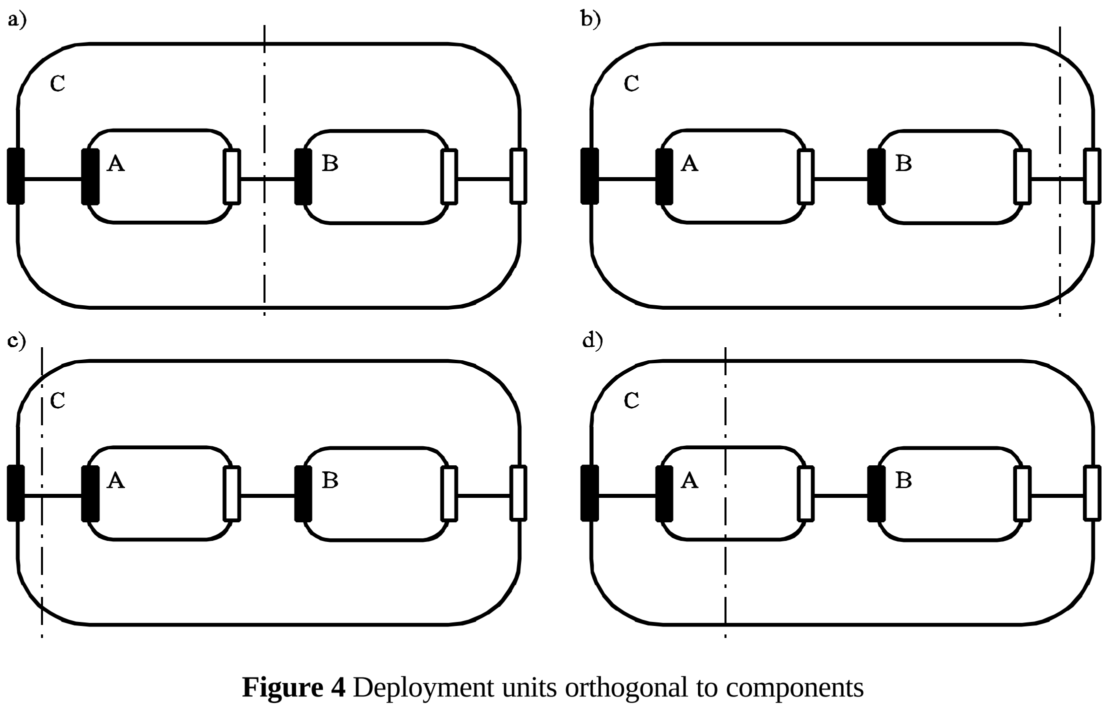

Figure 4 Deployment units orthogonal to components

Also, there is no difference between (1) and (2) in the following issue: If a distribution boundary crosses the interface tie of two components A and B, the actual deployment of A and B in general substantially influences the communication of A and B. For example, in Figure 5 the method calls on the r and q interfaces have to be modified in order to use an appropriate middleware technique of remote method calls, e.g. RMI stub and skeleton is to be employed. These modifications include changes to the internal architectures of A and B . Analogously with the inheritance anomaly concept [10, 21], we refer to this kind of a post-design modification of a component enforced by its deployment as *deployment anomaly.*

As a quick fix, one can imagine employing an ordinary component DC mediating the communication of A and B (Figure 6). In principle, however, this leads to the deployment anomaly again: (1) If a component DC was added to handle change in communication enforced by the deployment, the parent

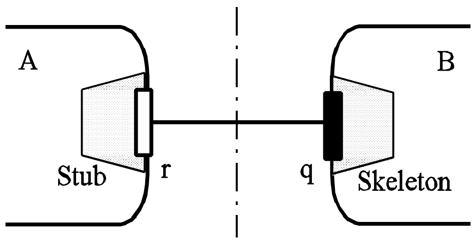

Figure 5 Modifying components' architecture internally during deployment

component of A and B would be modified by this adjustment of its architecture; (2) As it is unrealistic to imagine a primitive component spanning more deployment docks (and we assume this is not permitted), DC has to be considered a composed component; this leads to the issue of adjusting the internals of some inner components of DC, similarly to Figure 5.

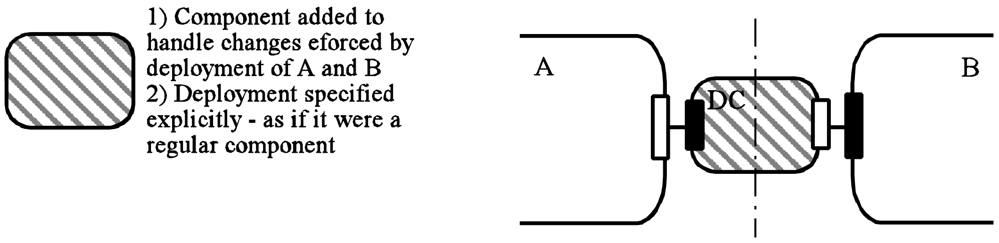
1) Component added to handle changes eforced by deployment of A and B
2) Deployment specified explicitly - as if it were a regular component
A
DC
B

Figure 6 Adding a component to reflect change in deployment

To illustrate the deployment anomaly on the BankingDemo example, consider the following deployment scenarios for the Bank component. (I) The whole Bank component is to be deployed into a single deployment dock with all of the Bank's subcomponents. Since all the interactions among the Bank's subcomponents are based on procedure calls, the procedure calls within the Bank component are local calls only (Figure 7a). (II) The DataStore component is to be deployed in a separate deployment dock. Thus, inside the Bank component, all the interactions with the DataStore component have to be modified to be based on remote procedure calls. This is the post-design modification enforced by this particular deployment, affecting the Teller, Supervisor, and DataStore components (Figure 7b). Similarly, adding the mediator components RPC1 and RPC2 to encapsulate the remote communication (Figure 7c) does not help much since it is necessary to post-modify the architecture of the parent Bank component.

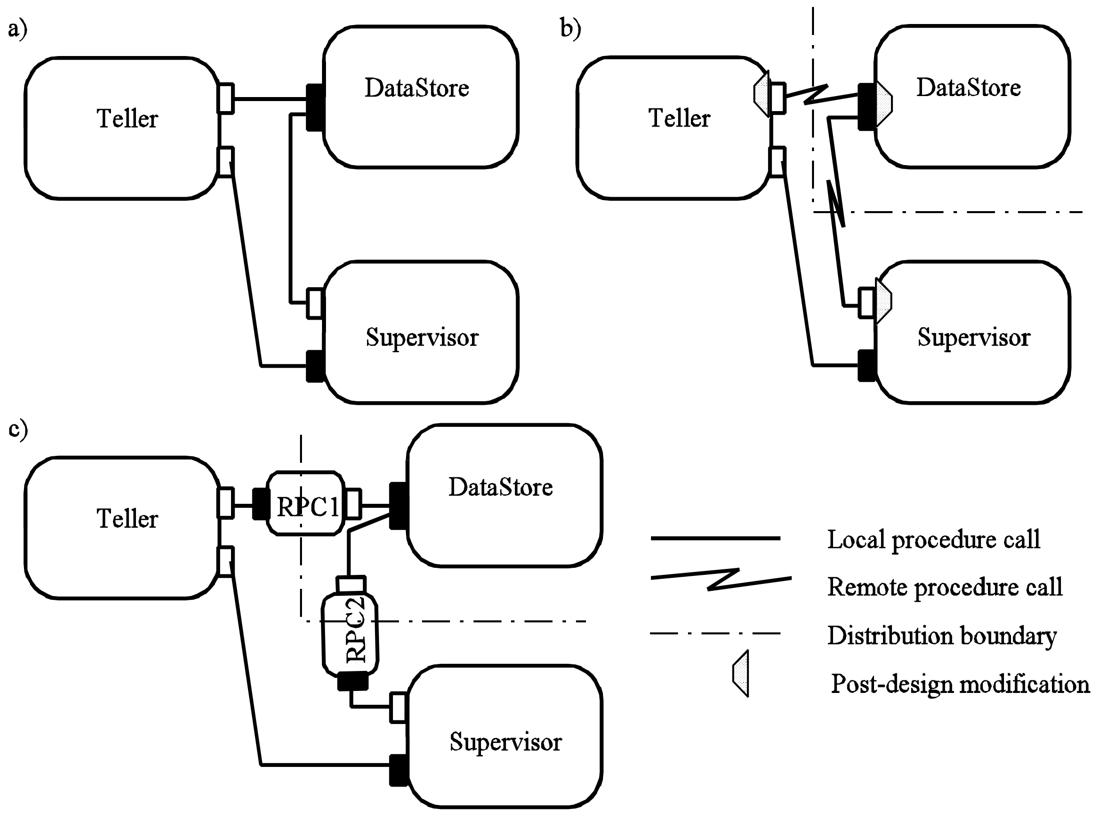

Figure 7 Deployment anomaly

3.3. Targeting the deployment anomaly: connectors

Basically, the deployment anomaly could be addressed by introducing a first class abstraction which would be: (a) inherently distributed (a boundary of distribution could cross it), (b) flexible enough to accommodate changes to the component communication enforced by a particular deployment. The connector abstraction can meet this requirement if defined accordingly: In order to address the deployment anomaly issue in full, a connector should exhibit the following properties: (1) It should be a part of the system architecture from the very beginning (being a first class entity at the same abstraction level as a component). (2) In order to absorb the changes in component communication induced by the modification of deployment, a flexible parametrization system of the connector internals has to be provided. (3) To reflect inherent distribution, the deployment of a connector should not be specified explicitly; it can be inferred from the deployment description of the components involved in the communication the connector mediates (Figure 8). As a consequence, the lifecycle of a connector inherently differs from the lifecycle of a component; moreover, its underlaying code has to be (semiautomatically) generated as late as its deployment is known - see Section 5.3.3.

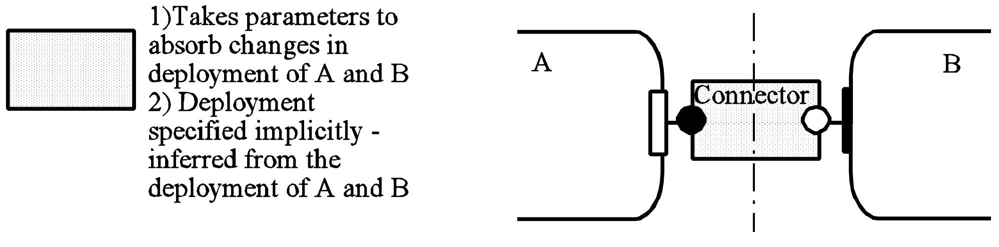
1)Takes parameters to absorb changes in deployment of A and B
2) Deployment specified implicitly - inferred from the deployment of A and B
A
Connector
B

Figure 8 Connector accepting deployment modification

4. Basic Connector Tasks

Let us consider a connector to be a first class entity representing a particular interaction among the components of an application at both the architecture description level and implementation level. To understand the connector concept properly, it is necessary to identify and analyze the basic tasks a connector should perform. In the process of such an identification, the taxonomy of software connectors presented in [12] can be used for guidance. From the main service categories and the basic connector types of this taxonomy, we have selected the following list of connector tasks that we consider the key ones: control and data transfer, interface adaptation and data conversion, access coordination and synchronization, communication intercepting, and dynamic component linking. Each of these tasks covers a different aspect of the component interactions. Being a result of many experiments with the connector design, this particular selection reflects our aim at building hierarchically structured connectors. In Section 6, we will show that most of the basic connector tasks can be provided through a simple hierarchical composition of a few primitive connector elements.

It should be emphasized that, first of all, connectors are intended to target the communication at the underlying middleware level and not to address, e.g., the classical communication protocol stack issues.

4.1. Control and data transfer

The most obvious connector task is to specify a particular interaction among components in terms of possible control and/or data transfer. A connector specifies the mechanisms on which such an interaction is based (like procedure call, event handling, and data stream). Each of these mechanisms has specific characteristics and properties. For example, a procedure call can be local or remote. As to a remote procedure call, various kinds of middleware can be used to implement it (such as CORBA, Java RMI, SOAP[31]). Similarly, event handling can be based on an event channel, the publisher-subscriber pattern, a centralized event queue etc.

4.2. Interface adaptation and data conversion

When building an application composed of reusable components, a system developer can face the need to tie two (or more) components that have not been originally designed to interoperate. Therefore, their interfaces are most likely incompatible. However, if they are "similar enough", a possible solution is to mediate such an interaction via an adaptor converting the calls between these interfaces. A straightforward idea implied by this thought is to conceptually include an *adaptor* into the connector abstraction.

As mentioned in [16], there is the option (and challenge) to devise a mechanism for automatic or semi-automatic generation of all the necessary interface adaptors and/or data convertors.

4.3. Access coordination and synchronization

In principle, the ordering of method calls on a component's interface is important (the protocol concept in [14]). The permitted orderings are usually determined by a behavioral specification of the component (e.g., interface, frame and architecture protocols in SOFA [20], CSP-based glue and computation in Wright). Thus another basic connector task is access coordination and synchronization - enforcing compliance with the protocol of an interface (set of interfaces).

As an example, consider a server component implemented for a singe-threaded environment. If the component is to be deployed into an environment with multiple concurrent client threads, it is necessary to serialize all the client threads before entering the component. This can be achieved by designing a connector mediating the clients' access to this component and ensuring (and internally implementing) the necessary synchronization.

4.4. Communication intercepting

Since connectors mediate all interactions among components in a system, they provide a natural framework for intercepting component communication(without the participating components being aware of it). For example, such an interception can be used to implement various filters (with applications in cryptography, data compression, etc.), to implement mechanisms for monitoring the load of a particular component in the system, and to implement a support mechanism for debugging.

4.5. Dynamic component linking

Since most of the information related to component interactions is concentrated in connectors, the connectors decouple the interacting components from each other; this allows for easy dynamic modifications of components' ties. Thus, a potential connector task is dynamic component linking in support of a variety of dynamic changes to the system architecture and deployment (such as component migration and replacement).

5. Connector Model

To reflect a variety of interactions among components in a hierarchically structured system, a connector model supporting the creation of a connector by a hierarchical composition of its internal elements is a natural choice. This complies with the observation that the complexity of interactions among components depends on the granularity of the system's architecture description. A finer granularity of description implies a larger number of components with simpler interactions, while a coarser granularity implies a smaller number of components with more complex interactions.

In this section, we propose a connector model designed as follows: Every interaction among components in a system (application) is represented by a *connector* which is an instance of a *connector type*. Being generic (parameterized) in principle, a connector type is a pair <*connector frame*, *connector architecture*>. A generic parameter of a connector type can be (1) an *interface type parameter* or (2) a *property parameter*. Given a connector type, the *connector frame* specifies the black-box view of a connector type instance, while the *connector architecture* specifies the structure of a connector type instance in terms of its *internal elements* (primitive elements, component instances, and instances of other connector types) and their interactions. A more detailed description of the key concepts of the connector model is provided in the rest of this section.

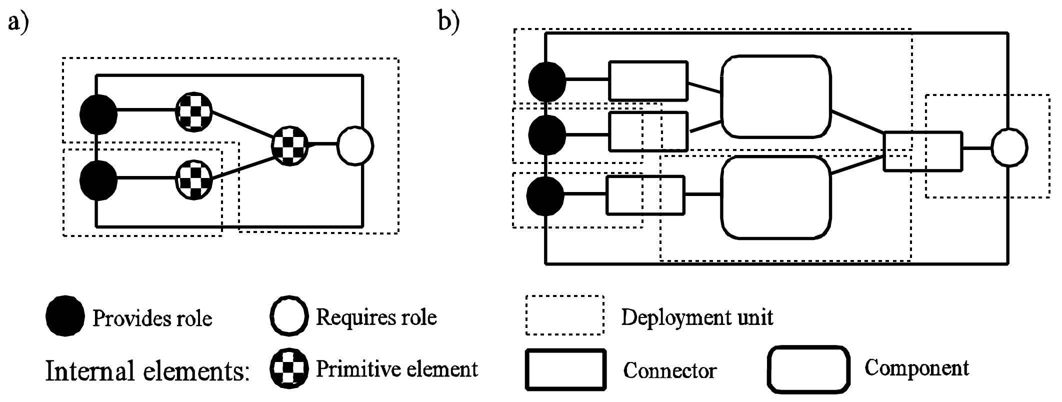

Figure 9 Connector model: a) simple architecture, b) compound architecture

5.1. Connector frame

A connector frame is represented by a set of named roles. In principle, a *role* is a generic interface of the connector intended to be tied to a component interface. In the frame's context, a role is either in the *provides* role or the *requires* role position. A *provides* role serves as an entry point to the component interaction represented by the connector type instance and is intended to be connected to a requires interface of a component (or to a requires role of another connector). Similarly, a *requires* role servers as an outlet point of the component interaction represented by the connector type instance and is intended to be connected to a provides interface of a component (or to a provides role of another connector). In general, roles are entities with generic interface type parameters; a concrete interface type of the role is specified as an actual parameter at the instantiation time of the connector type. The following fragment of ADL specification illustrates the role specification written in the SOFA CDL notation [13] (a proposed enhancement to the SOFA CDL). Note that the `Role` interface is a generic interface (expressed by the keyword `template`) parameterized by an interface type T. The `Role` interface provides the methods common to all roles (defined in the `RoleBase` interface), all the methods of T, and a method for linking a role to other elements (in the `Linkable` interface). The Role interface is the same for both provides and requires roles.

```
template interface Linkable {
  void link(in Object target) raises LinkException;
};
interface RoleBase : Linkable {...};
template interface Role<T> : RoleBase, T {};
```

5.2. Connector architecture

Depending on the internal elements employed, a connector architecture can be *simple* or *compound*. The internal elements of a simple connector architecture are *primitive elements* only (Figure 9a). Being the basic building blocks, primitive elements are directly implemented by the underlying environment (determined by the programming language and operating system used, etc.). Primitive elements are typed; however, their types are usually generic (both the interface type and property parameters are allowed). For every primitive element type, its functionality specification written in plain English is provided. The precise specification of a primitive element type's semantics is given by its mappings to the concrete underlying environments. As an example, consider the stub and skeleton primitive element types. Their specification written in plain English might read: "stub and skeleton provide the standard marshaling and unmarshaling functionality known from the remote procedure call mechanism". Each of these elements is parameterized by its remote interface type and by the underlying implementation platform (specified as a property parameter). The mappings of the stub and skeleton element types' exist for each of the implementation platforms supported (CORBA, Java RMI, etc.). For examples of simple connector architectures, we refer the reader to Sections 6.1.1.1 - 6.1.1.3.

The internal elements of a compound connector architecture are instances of other connector types and/or components (Figure 9b). This concept allows for creating complex connectors with hierarchically structured architectures reflecting the hierarchical nature of component interactions. As an example of a hierarchically structured interaction, consider a pair of components, sender and receiver, interacting by exchanging events. At a finer granularity of description, an event channel component can mediate the event exchange, i.e., the original interaction can be hierarchically decomposed into simpler interactions of the sender component with the event channel component and of the event channel component with the receiver component. For an example of the compound connector architecture supporting event handling, we refer the reader to Section 6.1.2.

5.3. Connector lifecycle

The connector lifecycle substantially differs form the component lifecycle. It can be viewed as a sequence of the design time, instantiation time, deployment and generation time, and runtime phases.

5.3.1. Connector design

To define a connector type, it is necessary to specify its frame and architecture. The frame specification involves specifying all the provides and requires roles and their generic parameters. As to the connector architecture, specifying a compound connector architecture is similar to specifying a compound component architecture - the connector internals are described in terms of its nested component and connector instances and their interconnections. A simple connector architecture is based on primitive elements types. For each of the primitive element types, its description in plain English has to be provided together with a definition of its mappings to concrete underlying environments (at least one mapping has to be provided). Moreover, the connector internal architecture is to be described in terms of its primitive elements instances and their interconnections. Note that most of the primitive elements present in the connector architecture are usually generic (employing both interface type and property parameters).

Since connectors are inherently distributed entities, the last step of a connector type's development process is the specification of potential distribution boundaries. This is done by dividing the connector architecture into a number of disjoint *deployment units*. A deployment unit is formed by the roles and those internal elements designed to share the same deployment dock.

5.3.2. Connector instantiation

The second stage of the connector lifecycle consist in instantiating the connector types within an application. For every connector instance, since the actual interface types of the components connected by the connector instance are known at this stage, the interface type parameters of the connector's roles can be resolved. Also the need for certain primitive elements (such as interface adaptors) to be present within the connector architecture arises. Nevertheless, a part of the connector instance remains generic - due to the unresolved property parameters related to a future deployment of the connector.

5.3.3. Connector deployment and generation

Its natural that connectors are deployed at the same time as the components the interactions of which they represent. During the deployment phase, each of the connector's deployment units is assigned a specific deployment dock to be deployed into. The actual deployment dock of the connector's deployment units can be inferred from the locations of the components interconnected by the connector.

Once the deployment of a connector is known, the connector's implementation code is (semi-automatically) generated according to the communication primitives offered by the deployment docks's underlying environments. Note that the generated code of the primitive elements either follows their mapping to the underlying programming environment, or it can be null (e.g., there is no need for an adaptor if the interfaces of the connected components match).

A typical scenario of the code generation of a connector is as follows: (1) Using a deployment tool, the deployment of components (the interaction of which the connector represents) is specified. (2) Each of the selected deployment docks is then asked whether it is able to automatically generate the implementation code of those internal elements of the connector that are intended to be deployed in it. (3) The deployment dock replies the list of technologies offered by its underlying environment on which the generated implementation could be based. If the connector implementation cannot be automatically generated by the deployment dock, the returned list is empty. (4) All these lists are searched by the deployment tool in order to find a match in the offered technologies. (5a) If a matching technology exists, the deployment docks are asked to generate the connector's implementation code for this technology. (5b) If no matching technology exists, the user is given an option either to change the application's deployment, or to provide the connector's implementation manually. For an example of the connector deployment and implementation code generation, we refer the reader to Section 6.2.

6. Case Study

As a proof of the concept, the connector model described in Section 5 have been integrated into the SOFA/DCUP component model. This section describes this integration thus introducing the SOFA/DCUP connector model.

6.1. SOFA/DCUP connectors

Similar to SOFA components, a *connector* in SOFA is an instance of a *connector template*. A connector template is in principle a connector type. Like a component template, a connector template CT is a pair <connector frame, connector architecture>; connector templates are specified in an enhancement of SOFA CDL. The connector frame provides the black-box view of an instance of CT; it defines the interface by indicating its *provides* and *requires* roles. The gray-box view of an instance of CT is provided by the connector architecture.

6.1.1. Predefined connector types

To avoid specifying the frequently used connector types repeatedly, SOFA/DCUP provides a set of predefined connector types: *CSProcCall*, *EventDelivery*, and *DataStream*.

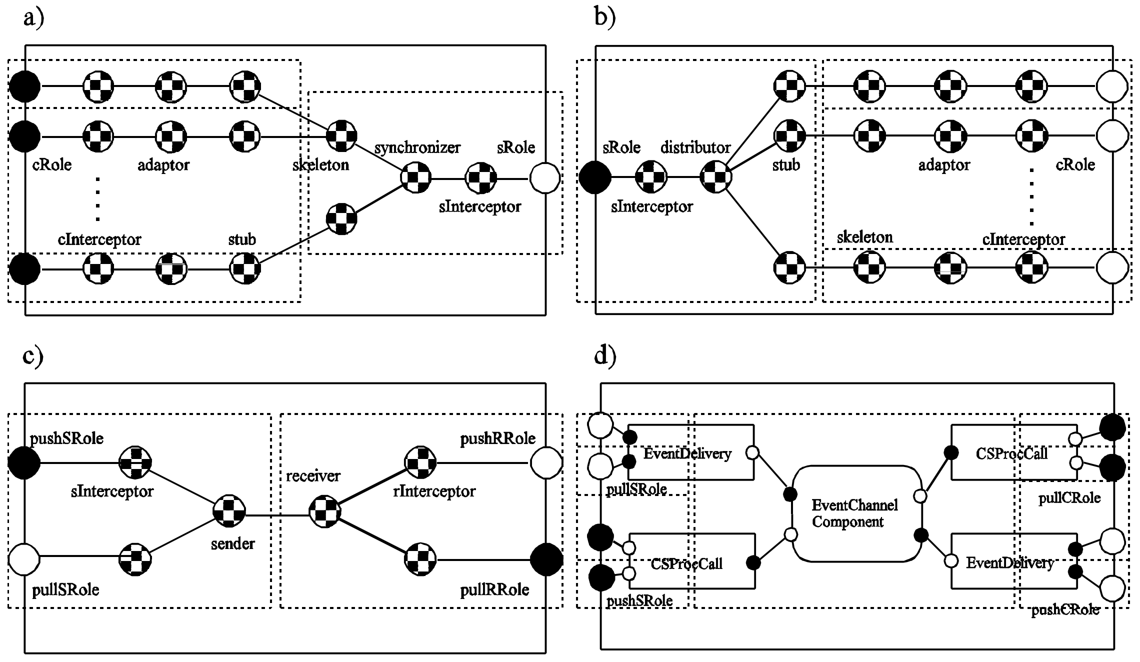

Figure 10 a) CSProcCall connector type, b) EventDelivery connector type, c) DataStream connector type, d) EventChannelDelivery connector type

6.1.1.1 CSProcCall

CSProcCall is the predefined connector type representing the (possibly remote) procedure call interaction semantics. The interaction is based on the existence of multiple *caller* entities (client components) invoking operations on a *definer* entity (server component).

The CSProcCall frame consists of a single requires role connecting a server component (`sRole`), and of multiple provides roles connecting client components (`cRole[1]` - `cRole[N]`). All of the roles are generic entities with interface type parameters.

The CSProcCall architecture is simple. It consists of several primitive elements interconnected in the way illustrated on Figure 10a). The `cInterceptor` and `sInterceptor` instances of `TInterceptor` provide a framework for plugging in an additional connector to support logging, debugging, etc. `TInterceptor` is based on a callback notification of the registered (subscribed) entities. An interface `adaptor` can be included in a connector instance in the case when a particular client's interface does not match the server interface. A (`TStub`, `TSkeleton`) instance pair is used in the case when a remote invocation is needed. These primitive elements provide the standard RPC marhalling/unmarshalling functionality. A `synchronizer` can be included into a connector instance in the case when the server component requires client invocations to be synchronized when accessing its interface.

The CSProcCall connector type consists of several deployment units. There is a single *server deployment unit* (composed of `sRole`, `sInterceptor`, `synchronizer`, and of all server `skeleton` instances) and multiple *client deployment units* (each of them composed of `cRole`, `cInterceptor`, `cAdaptor`, and a `stub`). There is one client deployment unit per connected client component. The actual deployment of each unit is inferred from the deployment of the connected components (e.g., the server deployment unit has to be deployed into the same deployment dock as the server component).

The following fragment of ADL source text illustrates the main parts of the CSProcCall CDL specification. For brevity, the CDL specifications of other connector types are omitted in this paper.

```
connector frame CSProcCall<T[1..NoC], ST> (Properties properties){
provides: Role<T[1..NoC]> CRole[1..NoC];
requires: Role<ST> SRole;
};

template interface TInterceptor<T> : T, Linkable {
  void linkNotificationDistributor(in T distributor) raises LinkException;
};
template interface TAdaptor<T1, T2> : T1, Linkable {};
template interface TStub<T> : T {};
template interface TSkeleton<T> : Linkable {};
template interface TSynchronizer<T> : T, Linkable {};

connector architecture CSProcCall {
  inst TInterceptor<T[1..NoC]> cInterceptor[1..NoC];
  inst TAdaptor<T[1..NoC], ST> adaptor[1..NoC];
  inst TStub<ST> stub[1..NoC];
  inst TSkeleton<ST> skeleton[1..NoS];
  inst TSynchronizer<ST> synchronizer;
  inst TInterceptor<ST> sInterceptor;
  delegate CRole[1..NoC] to cInterceptor[1..NoC];
  bind cInterceptor[1..NoC] to adaptor[1..NoC];
  bind adaptor[1..NoC] to stub[1..NoC];
  bind stub[1..NoC] to skeleton[1..NoS];
  bind skeleton[1..NoS] to synchronizer;
  bind synchronizer to sInterceptor;
  subsume sInterceptor to SRole;
};
```

6.1.1.2 EventDelivery

Event Delivery is the predefined connector type designed for (possibly distributed) event-based communication reflecting the publisher-subscriber design pattern. The interaction assumes the existence of a single event *supplier* entity (supplier component) invoking operations on the subscribed *consumer* entities (consumers components).

The EventDelivery frame consists of a single provides role to connect an event supplier component (SRole) and multiple requires roles to connect consumer components (`cRole[1] - cRole[N]`). All of these roles are generic entities with interface parameters. Based on concrete consumer and supplier components interfaces, the exact role types are specified at the time of a connector instantiation.

The EventDelivery architecture is simple. It consists of several primitive elements connected as indicated on Figure 10b). The `distributor` primitive element collects references to the subscribed event consumers and distributes each of the supplied events by calling the appropriate method on the registered consumer interfaces (sequentially or in parallel).

The EventDelivery connector type contains several deployment units. There is a single *supplier deployment unit* (composed of `sRole`, `sInterceptor`, `distributor`, and all the `stubs`) and multiple *consumer deployment units* (each of them is composed of `cRole`, `cInterceptor`, `adaptor`, and a `skeleton`). There is one consumer deployment unit per connected consumer component. The concrete deployment of each deployment unit is inferred from the deployment of the connected components (the supplier deployment unit has to be deployed into the same deployment dock as the supplier component, each of the consumer component units has to be deployed together with the corresponding consumer component).

6.1.1.3 DataStream

DataStream is the predefined connector type representing the point-to-point transfer of a large amount of (possibly raw) data from a sender to a receiver components.

The DataStream frame consists of two provides roles connecting the sender component in the push or pull mode (`pushSRole` and `pullSRole`), and of two requires roles connecting the receiver component in the push or pull mode (`pushRRole` and `pullRRrole`).

The DataStream architecture consists of several primitive elements connected as indicated on Figure 10c). The primitive elements `sender` and `receiver` form the pair of elements that specifies the mechanism used to perform the data transmission (TCP/UDP protocol, Unix pipe, etc.).

The DataStream connector type consists of two independent deployment units. There is a *sender deployment unit* (composed of `pushSRole`, `pullSRole`, sInterceptor, and `sender`) and a *receiver deployment unit* (composed of `pushRRole`, `pullRRole`, rInterceptor, and `receiver`). The concrete deployment of each unit is inferred from the deployment of the connected components (the sender deployment unit has to be deployed into the same deployment dock as the sender component, the consumer component unit has to be deployed together with the receiver component).

6.1.2. Constructing new connector types

The process of creating a new connector type starts with specification of its frame and architecture. Recall that primitive elements (forming simple architectures), and instances of other connector types together with instances of any component types (forming compound architectures) can be used as building blocks for the connector architecture. When introducing a new primitive element type, its semantic description written in plain English and the mappings to the underlying environments should be provided. This can be illustrated on the example of EventChannelDelivery, a new connector type reflecting event-based communication via an event channel.

The EventChannelDelivery connector type should reflect the supplier-consumer pattern in such a way that multiple suppliers send data asynchronously to multiple consumers through an event channel. Similar to the CORBA Event Service, this connector type will provide both the pull and push communication model for suppliers and consumers. In the push mode, the supplier objects control the flow of data by pushing it to consumers. In the pull mode, the consumer objects control the flow of data by pulling it from the supplier. The EventChannelDelivery connector insulates the suppliers and consumers from having to know which mode is being used by other components connected to the channel. This means that a pull supplier can provide data to a push consumer and a push supplier can provide data to a pull consumer.

The EventChannelDelivery connector frame consists of multiple roles that allow to connect supplier components in both the push and pull modes (*PushSRole* and *PullSRole*) and multiple roles that allow to connect consumer components in both the push and pull modes (*PushCRole* and *PullCRole*). All of the connector’s roles are generic entities with interface parameters.

The architecture of the EventChannelDelivery connector type is compound. As depicted on Figure 10d), the core element of the EventChannelDelivery architecture is an instance of the `EventChannel` component. The other internal elements of EventChannelDelivery are: two instances of the CSProcCall connector type (one to allow multiple suppliers to push data to the event channel, the second to allow multiple consumers to pull for data from the event channel), and two instances of the EventDelivery connector type (one to pull for data from multiple suppliers, and the second to push data to multiple consumers).

As a next step, the architecture is to be divided into deployment units. This division is determined by the deployment units of its internal elements. The *channel deployment unit* groups together the EventChannel component with those deployment units of the internal connector instances that are tied to this component. The modification of the remaining deployment units is obvious.

6.2. Using SOFA/DCUP connectors

To demonstrate how the SOFA/DCUP connectors can be used, consider the BankingDemo application introduced in Section 2.2. The following fragment of its CDL specification illustrates how its architecture description is to be modified if connectors are used.

```
architecture Bank version "1.0" {
  inst DataStore DS;
  inst Supervisor Sup;
  inst Teller T[1..NoT];
  bind Sup.DataStore to DS.DataStore using CSProcCall;
  bind T[1..NoT].DataStore to DS.DataStore using CSProcCall;
  bind T[1..NoT].Supervisor to Sup.Supervisor using CSProcCall;
  delegate Teller[1..NoT] to T[1..NoT].Teller using CSProcCall;
  subsume DS.LogWindow to LogWindow using CSProcCall;
  subsume Sup.LogWindow to LogWindow using CSProcCall;
  subsume T[1..NoT].LogWindow to LogWindow using CSProcCall;
};

system BankingDemo version "1.0" {
  inst Bank aBank;
  inst VisualLogWindow LW;
  inst Customer C[1..NoC];
  bind C[1..NoC].Teller to aBank.Teller[?] using CSProcCall;
  bind aBank.LogWindow to LW.LogWindow using EventDelivery;
  bind C[1..NoC].LogWindow to LW.LogWindow using EventDelivery;
};
```

As a result of assembling the application, its *assembly descriptor* is created. In principle, the assembly descriptor is a text representation of the application's assembled tree - Section 3.1.

During the distribution design stage, the application is decomposed into deployment units in order to group together those components that should share the same deployment dock. An assembly descriptor enriched by the specification of deployment units is called a *deployment descriptor form*. It is a key document used in the process of the application's deployment. The deployment descriptor form of the BankingDemo application could be:

```
deployment BankingDemo architecture "1.0" {
units:
  BankUnit {
    location: <URL>;
    components:
      aBank {
        architecture "1.0";
        units:
          StaffUnit {
            location: <URL>;
            components:
              Sup {architecture "1.0";};
              T[*] {architecture "1.0";};
          };
          DataUnit {
            location: <URL>;
            components:
              DS {architecture: "1.0";};
          };
      };
      C[*] {architecture: "1.0";};
  };
  AdminUnit {
    location: <URL>;
    components:
      LW[*] {architecture: "1.0";};
  };
};
```

For simplicity, assume that the whole BankingDemo application is written in Java and we have two SimpleJava deployment docks running in our network. The SimpleJava deployment dock is formed by a single Java Virtual Machine with a ClassLoader modified to load, instantiate, and run SOFA components.

When deploying the BankingDemo application, the target deployment dock has to be specified for each of its deployment units (by filling the concrete deployment docks' URLs in the "location" tags of the deployment descriptor form). The deployment can be modified before any of the application's executions starts.

Examples of deployment scenarios for the BankingDemo application were presented in Section 3.2 (for sake of brevity, we focus on the Bank component's deployment). The first scenario assumes the whole Bank component to be deployed into a single deployment dock with all of the Bank's subcomponents. This can be expressed by the following fragment of deployment descriptor:

```
aBank {
  architecture "1.0";
  units:
    StaffUnit {
      location: sofa://nenya.ms.mff.cuni.cz/simple_java_dock;
      components:
        Sup { architecture "1.0";};
        T[*] {architecture "1.0";};
    };
    DataUnit {
      location: sofa://nenya.ms.mff.cuni.cz/simple_java_dock;
      components:
        DS {architecture: "1.0";};
    };
};
```

Let us focus on the Sup and DS subcomponents connected by a CSProcCall connector instance. Since the DS component shares the address space with the Sup component and their interfaces match, a simple local procedure call connector (with no stub, skeleton, and adaptor) is generated in cooperation with the target deployment dock to mediate their interaction (Figure 11a).

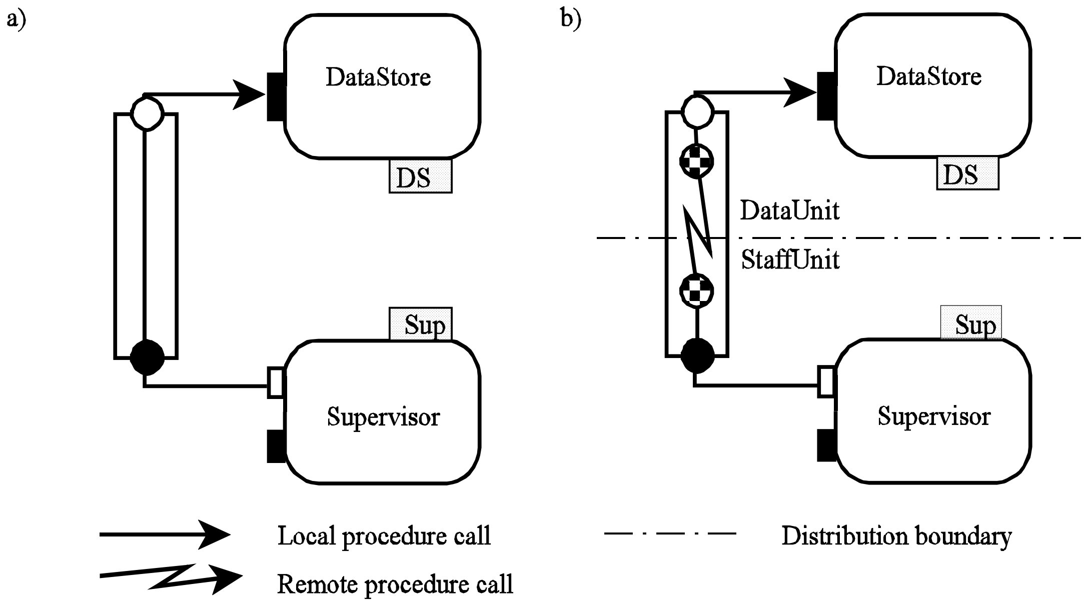

Figure 11 Example Bank deployments

Now, consider the second deployment scenario which assumes the DS component being deployed in a separate deployment dock. This can be expressed by the following fragment of deployment descriptor:

```
aBank {
  architecture "1.0";
  units:
    StaffUnit {
      location: sofa://nenya.ms.mff.cuni.cz/java_dock;
      components:
        Sup { architecture "1.0";};
        T[*] {architecture "1.0";};
    };
    DataUnit {
      location: sofa://bilbo.ms.mff.cuni.cz/java_dock;
      components:
        DS {architecture: "1.0";};
    };
};
```

Since the DataStore component does not share the address space with the Supervisor component (they are deployed into two different SimpleJava deployment docks), a cross-address space communication is needed. SimpleJava deployment docks provide two mechanisms for making remote procedure calls - Java RMI and CORBA. Since both target deployment docks are of the same nature, any of these mechanism can be used to generate the stub and skeleton internal elements included in the resulting connector (Figure 11b).

7. Evaluation and Conclusion

In this paper, we have elaborated the idea of considering connectors as first class entities representing the interactions among components in an application. To meet the first goal of the paper stated in Section 1.2 (bringing an additional argument in favor of connectors as first class entities), we have focused on the lifecycle of an application, emphasizing especially its deployment phase. (Note that in software industry supported technologies, e.g., [15, 16, 27], a deployment of the system is typically specified by means of deployment descriptors, however, none of them employs connectors. Similarly, none of the experimental ADL systems, such as [17, 23, 8, 25, 1], targets the deployment issue directly nor combines it with connectors). In Section 3.2, we have shown that in a distributed environment, deploying architectures composed solely of components leads to the deployment anomaly (post design modifications of the application's components are enforced by deployment modifications). To avoid the deployment anomaly, it is necessary to factor the application into the deployment-neutral part (comprising the "real" application functionality located in components) and deployment-sensitive part, which can be regenerated anytime the deployment of the application is modified. The deployment sensitive part of the application is located in dedicated entities (connectors) with lifecycle different from the lifecyle of the components.

The second goal of the paper was to propose a novel connector model allowing to express and represent a variety of possible interactions among components in an application at all key stages of the application lifecycle. Addressing this goal, we have first identified and articulated a set of requirements on such a connector model (Section 4). To meet the requirements, we have specified the model based on hierarchical structuring of a connector: Every connector type representing a (possibly) complex interaction among components is created (recursively) as a hierarchy of internal elements. Connector types with simple architectures (procedure call, data stream, event handling, etc.) are composed entirely of primitive elements, while connector types with compound architectures are hierarchically composed of instances of other connector types (connectors), and instances of component types (components). Elementary enough to be implemented and reasoned about, primitive elements can be easily composed into simple hierarchies to fulfill most of the basic connector tasks identified and analyzed in Section 4.

Possibly the closest to our work are the connector models employed in UniCon and Wright. A key contribution of our connector model it that it addresses both the specification and implementation of connectors of arbitrary complexity. As mentioned in Section 1, recent connector models only allow for either specifying connector types of arbitrary complexity on an abstract level with no direct relation to potential implementation and deployment (Wright), or for implementing a small set of simple connector types (UniCon).

Having finished a pilot implementation of our connector model, our current work is focused on finding techniques for automatic generation (or at least semi-automatic generation) of primitive elements, including interface adaptors, stubs and skeletons for remote communication, etc. We believe this can be done by defining a mapping of every primitive element type to the primitives of the underlying programming environment. Since primitive elements are mostly generic entities, the mapping will be used to generate a concrete element for each set of its actual parameters. Another future intention is to apply behavioral protocols [20] in connector specification to express the interplay of its internal elements.

References

[1] Allen, R. J.: A Formal Approach to Software Architecture. Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1997.

[2] Bishop, J., Faria, R.: Connectors in Configuration Programming Languages: are They Necessary? Proceedings of the 3rd International Conference on Configurable Distributed Systems, 1996.

[3] Blair, G. S., Coulson, G., Robin, P., Papathomas, M.: An Architecture for Next Generation Middleware. In Proceedings of Middleware'98, Lancaster UK, 1998.

[4] Ducasse, S., Richner, T.: Executable Connectors: Towards Reusable Design Elements. In Proceedings of ESEC/FSE'97, Lecture Notes in Computer Science no. 1301, Springer-Verlag, 1997.

[5] Hoare, C. A. R.: Communicating Sequential Processes, Prentice-Hall, 1985.

[6] Issarny, V., Bidan, C., Saridakis, T.: Achieving Middleware Customization in a Configuration-Based Development Environment: Experience with the Aster Prototype. In Proceedings of ICCDS '98, 1998, http://www.irisa.fr/solidor/work/aster.html.

[7] Jackson, M.: Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices, ACM Press/Addison-Wesley, 1995.

[8] Luckham,D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., Mann, W.: Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering}, 21(4), 1995.

[9] Magee, J., Dulay, N., Kramer, J.: Regis: A Constructive Development Environment for Distributed Programs. In Distributed Systems Engineering Journal, 1(5), 1994.

[10] Matsuoka, S., Yonezawa, A.: Analysis of Inheritance Anomaly in Object-Oriented Concurent Programming Languages. Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993.

[11] Medvidovic, N., Taylor, R. N.: A Framework for Classifying and Comparing Architecture Description Languages. In Proceedings of the 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, 1997.

[12] Mehta N. R., Medvidovic, N.,Phadke S.: Towards a Taxonomy of Software Connectors. In Proceedings of the 22th International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, 2000.

[13] Mencl, V.: Component Definition Language, Master thesis, Charles University, Prague, 1998.

[14] Nierstrasz, O.: Regular Types for Active Objects, In Proceedings of the OOPSLA '93, ACM Press, 1993, pp. 1–15.

[15] OMG orbos/99-04-16, CORBA Component Model. Volume 1, 1999.

[16] OMG orbos/99-04-17, CORBA Component Model, Volume 2, 1999.

[17] Oreizy, P., Rosenblum, D. S., Taylor, R. N.: On the Role of Connectors in Modeling and Implementing Software Architectures, Technical Report UCI-ICS-98-04, University of California, Irvine, 1998.

[18] Perry, D.E., Wolf, A. L.: Foundations for the Study of Software Architecture. ACM Software Engineering Notes, vol. 17, no. 4, 1992.

[19] Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP Architecture for Component Trading and Dynamic Updating. In Proceedings of ICCDS '98, Annapolis, IEEE CS, 1998, pp. 43–52.

[20] Plasil, F., Besta, M., Visnovsky, S.: Bounding Component Behavior via Protocols. In Proceedings of TOOLS USA '99, Santa Barbara, USA, 1999.

[21] Plasil, F., Mikusik, D.: Inheriting Synchronization Protocols via Sound Enrichment Rules. In Proceedings of Joint Modular Programming Languages Conference, Springer LNCS 1204, March 1997.

[22] Plasil, F., Stal, M.: An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM. Software Concepts & Tools (vol. 19, no. 1), Springer 1998.

[23] Purtilo, J. M.: The Polylith Software Bus. ACM Transactions on Programming Languages and Systems, 16(1), 1994.

[24] Shaw, M.: Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In D.A. Lamb (ed) Studies of Software Design, Proceedings of a 1993 Workshop, Lecture Notes in Computer Science no. 1078, Springer-Verlag 1996.

[25] Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., Zalesnik, G.: Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, Vol. 21, No. 4, April 1995, pp. 314–335.

[26] Sun Microsystems: JavaBeans 1.0 Specification. http://java.sun.com/beans/docs/spec.html.

[27] Sun Microsystems: Enterprise JavaBeans 1.1 Specification. http://java.sun.com/products/ejb/docs.html.

[28] Szyperski, C.: Component Software, Beyond Object-Oriented Programming. Addison-Wesley, 1997.

[29] Rogerson, D.: Inside COM. Microsoft Press 1997.

[30] Yellin, D. M., Strom, R. E.: Interfaces, Protocols, and the Semi-Automatic Construction Of Software Adaptors. In Proceedings of the OOPSLA ‘94, ACM Press, 1994, pp. 176–190.

[31] W3C: Simple Object Access Protocol (SOAP) 1.1. http://www.w3c.org/TR/SOAP.