Connectors in Software Architectures

Dušan Bálek

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

Abstract

Nowadays, to allow for rapid software evolution, more and more software developers are starting to construct their products from reusable software components. In this approach, the architecture of a system is described as a collection of components along with the interactions among those components. Even though the main building blocks of the system are components, the properties of the system also strongly depend on the character of the component interactions. This fact gave birth to a "connector" as an abstraction capturing the nature of these interactions. Even though the notion of connectors originates in the earliest papers on software architectures [54, 47], connectors are currently far from being a typical first class entity in contemporary component-based systems.

The main problem of current component-based systems and their respective ADLs is that they either do not capture component interactions at all, or that they only focus on application design stages. By articulating the problem of deployment anomaly, this thesis identifies a role that connectors should play when the distribution and deployment of a component-based application are considered. As none of the related component-based systems does so, a new connector model that allows a variety of (possibly complex) component interactions to be reflected at all the key stages of an application lifecycle (ADL specification, implementation, deployment, and runtime) is further introduced.

Keywords: component-based programming, architecture description language, software component, connector, deployment

Acknowledgments

I am deeply grateful to František Plášil, the supervisor of my doctoral studies at Charles University, for his support and valuable comments on this thesis. Also, Petr Tůma deserves a great deal of acknowledgment. My thanks also go to the current and recent members of our Distributed Systems Research Group for giving me inspiration, amusement, and fun. Among others, I particularly name Radovan Janeček, Radek Pospíšil, Marek Procházka, Miloslav Bešta, Stano Višovský, Vladimír Mencl, and Adam Buble. I also thank the Hlavka Foundation for partially sponsoring my participation in the ICCDS'98 conference. Last but not least, I am in debt to my parents and my wife, whose support and patience made this work possible.

Contents

Chapter 1

Introduction

1.1 Motivation

In 1998, a project was started by one of the Czech banks with the aim of creating a new information system for its dealers. One of the first design decisions was to develop this application as a collection of cooperating distributed components. As the underlying environment of the application was rather heterogeneous (PCs with various connectivity on the side of individual dealers and mostly UNIX servers on the bank side), CORBA was selected as the communication middleware. The original intention was clear: to start with a small core (a group of components providing the key functionality of the resulting application) and to later gradually extend this core with additional components. Some of the application's components would be created by the bank's IT department staff (specialists in various banking operations), some of the application's components (i.e., transaction management, security management, various data stores) would be provided by third parties. Our role, as people experienced in CORBA design, was to provide consultancy on communication middleware.

Since CORBA had been selected as the communication middleware from the very beginning, it was decided to fit the business logic of the application to specific needs of the CORBA environment as much as possible. This crucial decision caused the major problems which we faced during the project.

The first problem appeared immediately - at the design stage of the application. It was difficult to find a common language between ourselves (as CORBA specialists without a good understanding of the details of banking operations) and the bank staff (specialists in banking operations who did not understand the necessary details of CORBA communication) in order to successfully cooperate on the design of an application which comprised the needs of both worlds (the world of banking operations and the world of CORBA).

A similar problem arose when we started implementing the components of the application. Since the application's business logic was too closely tied to the communication middleware, every component contained both its operational code and CORBA related routines mixed

together. This approach became extremely inefficient for two reasons - we were unable to effectively divide our work (both teams worked on the same code with many collisions), and the resulting code was hard to maintain (Is it not the case that a small change made to a CORBA routine within a component extraneously affects the component's business operation?).

The last problem arose during deployment of the final application. The application had been developed over a period of more than two years, and the underlying environment of the application had therefore been modified in the meantime. Database and operating systems upgrades made it necessity to also upgrade the selected ORB. Unfortunately, the new version of the selected ORB significantly differed from the previous one, with only limited back compatibility. It was therefore necessary to make new modifications to the application components before they were brought into a real operation.

Upon completion of work on this project, a uniform experience had been gained: it had been proven that it is extremely useful (or even necessary) to clearly separate the business logic of an application from any concrete communication middleware. This idea became a major motivation for this work.

1.2 The SOFA/DCUP project

Work on this thesis began in 1999 as part of the SOFA/DCUP project being conducted by our research group at the Department of Software Engineering. Based on the CORBA and Java technologies, the goal of the SOFA project is to design a software environment to support component trading and dynamic updating based on provider - user (consumer) relation. In SOFA, an application is composed of dynamically downloadable and updatable components. The key issues addressed by SOFA are dynamic component downloading, dynamic component updating, component trading, licencing and billing, versioning, and security support.

The core of SOFA is implemented in the DCUP (Dynamic Components Updating) framework. DCUP is a novel architecture, which allows for dynamic component updating at runtime of related applications. The key problems of dynamic component updating addressed are making an update of a component fully transparent to the rest of the application, transition of state from the old version of a component to the new one, transition of references which cross the component boundary (in both directions), and dynamic communication with a component provider. In DCUP, these problems are addressed by a small set of abstractions with a clear separation of their functionality.

The bases of the SOFA/DCUP project were stated in 1998 by publication of our paper "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating" [48] at the ICCDS'98 conference. Since that time, more than twenty papers, Ph.D. and master theses have been published within this project extending the basic framework with a support for specifying component behavior, connectors, a versioning schema, and transaction management. Work on the SOFA/DCUP project is currently 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).

1.3 Challenges and goals of the thesis

Component-based systems and Architecture Description Languages (ADLs) have become fields of study in their own right, however their practical application is still to be demonstrated.

Reuse of components is an attractive idea, but real life has proven on many occasions that combining the business components provided by third parties in a running application can be a very demanding process. The main problem of current ADLs is that they either do not capture component interactions at all, or that they only focus on application design stages. The component interactions must however be reflected throughout the whole application lifecycle, otherwise they may become a serious obstacle in component reusability. In particular, the deployment phase of the application lifecycle has turned out to be critical in this respect.

As its initial goal, this thesis focuses on a typical lifecycle of a component-based application stating the problem of deployment anomaly. The use of connectors is discussed as a possible solution to this problem (connectors are ADL entities dedicated to reflecting and representing component interactions, clearly separating application's business logic from the underlying communication middleware). In analyzing the role of connectors throughout the whole application lifecycle, an additional argument is brought in favor of considering connectors as first-class ADL entities. An inherent part of this goal is to provide a brief survey of related component models and their respective ADLs, stressing their ability to describe component interactions.

Since none of the related connector models and their respective ADLs provides a sufficient support for describing a variety of (possibly complex) interactions with clear relations to their implementation, the second goal of the thesis is to propose a new connector model that allows for describing a variety of complex interactions at the same time with a (semi-) automatic generation of the corresponding interaction code. This encompasses identification of major requirements on a connector model design, followed by proposing the architectural model and programming framework of the connectors.

1.4 Structure of the text

The text of this thesis consists of four major parts. To provide a necessary background in the topic, a brief overview of concepts related to software components is presented in Chapter 2 together with a preview of related component models.

The problem of deployment anomaly that forms the second major part of the thesis is introduced in Chapter 3. Here the typical lifecycle of a component-based application is first described and, based upon this, the problem is articulated. The chapter concludes with a discussion of the way in which connectors (as abstractions that allow for a clear separation of the application's business logic from the communication middleware) can help to solve the problem.

The core of the thesis is formed by Chapters 4 - 7. At the beginning, basic connector tasks and requirements for a connector model's design are articulated in Chapter 4. This is followed by description of the design concepts of the SOFA/DCUP connector model, including a preview of predefined connector types together with an example of the creation process of a new connector type in Chapter 5. After that, Chapter 6 contains detailed description of the programming framework of SOFA/DCUP connectors. Changes are introduced which must be made to SOFA Component Description Language (CDL) to allow for description of connectors. The thesis then proceeds with presentation of the SOFA deployment framework and its role in connector generation. The runtime support gives a notion of how to instantiate SOFA/DCUP connectors and use them to interconnect a collection of component instances. To illustrate how the presented framework fits together, Chapter 7 ends with demonstration of a case study application.

The whole thesis concludes with Chapter 8 in which the goals achieved are first summarized and the major contributions to the topic are presented. A discussion of open problems and future intentions concludes Chapter 8 and the thesis itself.

Chapter 2

Background Information

A few years ago, the trend of constructing software systems as a collection of cooperating reusable components appeared and has become widely accepted. Apart from a number of academic research projects dealing with components [23, 52, 55, 22, 1, 46, 17], several industrial systems are now also on the market [58, 59, 38, 39, 53]. To provide the casual reader with the necessary background in the topic, this chapter presents an overview of the basic concepts related to components together with a brief survey of the existing component models (including the SOFA/DCUP component model).

2.1 Component model

In software engineering, the term component is used to denote many different things. For the purpose of this thesis, a *component* is a reusable black/grey-box entity (a piece of code) with well-defined interface and specified behavior which is intended to be combined with other components to form a software system (an application). A component can usually have multiple interfaces, some to provide services to the component's clients, others to require services from the environment. Components can be nested to form hierarchies; a higher-level component can be composed of several mutually interconnected, cooperating subcomponents.

The basic architecture of a component is defined by a *component model*. A component model specifies the structure of component interfaces, the mechanisms by which a component interacts with its environment, component internal structuring, etc. Basically, the component model provides guidelines as to how to create and implement components and how to assemble them into a larger application.

Nowadays, a number of different component models exist. Each of them usually provides a specialized language that allows for specifying a system architecture in terms of cooperating mutually interconnected components. Such a language is usually denoted as an *architecture description language* (ADL). Note that the relation between a component model and the respective ADL is usually very close; a component model is often identified by the name of its respective ADL.

A more detailed description of the key concepts of the current component models is provided in the rest of this section.

2.1.1 Multiple interfaces and composition

Interfaces are the means by which components connect. An interface provided by a component can be viewed as a set of operations implemented by the component. It corresponds to a procedural interface of a traditional library or to an object interface. In addition to being a set of operations, an interface also serves as the contract between a component and its environment which separates providers implementing the component interface from clients using this interface. (By publishing its interface, a component tells the outside world about the services it provides. By invoking operations upon the published interface, clients access the component's services as a black-box.)

Nowadays, component interfaces are usually modeled by multiple object interfaces. In such a case, a component model usually provides mechanisms of navigation among a component's interfaces and it usually defines a concept of component instance identity. This concept makes it possible to decide whether two object interfaces belong to the same component.

In addition to services *provided* by a component, the component's interfaces also specify services *required* by the component. Require interfaces declare operations that a properly connected component can invoke on its environment (other components, containers, etc.). The existence of require interfaces distinguishes components from traditional libraries and ordinary objects whose requirements are usually hidden in their implementation code. That is why connections between components are more symmetric than connections between objects. A connection between two objects O1 and O2 originates somewhere inside O1's implementation code and ends in the declared interface of O2 (Figure 1a), while a connection between two components C1 and C2 originates in a declared require interface of C1 and ends in a declared provide interface of C2 (Figure 1b). This symmetry allows for easier configuration management of systems based on component technologies.

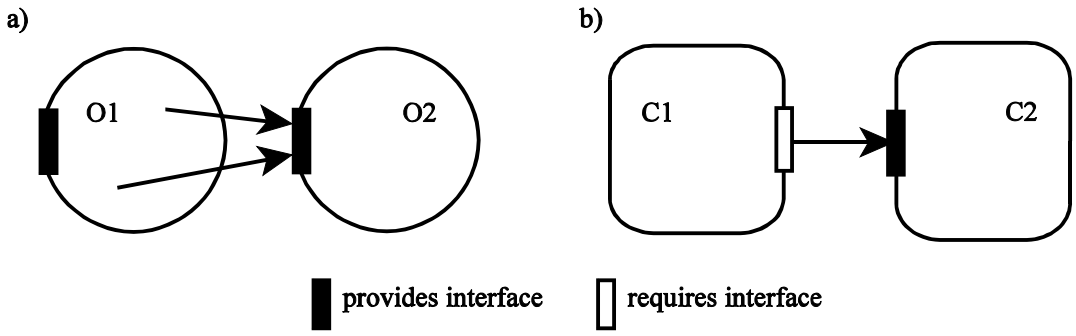

Figure 1 a) Asymmetric connections between objects (traditional libraries),
b) Symmetric connections between components

2.1.2 Component hierarchies and nesting

As mentioned in Section 2.1, using a component technology, software systems (applications) are constructed by assembling individual (reusable) components together. When constructing a

larger software system with hundreds of components that have to be connected together to form the desired system architecture, this task could be quite complicated.

To allow for abstraction in complex software systems, software engineering introduced aggregation. In component models, aggregation appears in a form of component nesting. This concept permits definition of a component whose internal architecture is formed by a set of mutually interconnected subcomponents. Such a component is usually called a *compound component.* A component without subcomponents (which is therefore directly implemented using underlying implementation language) is called a *primitive component.* As an example, compound component C with three nested primitive subcomponents S1..S3 is depicted in Figure 2.

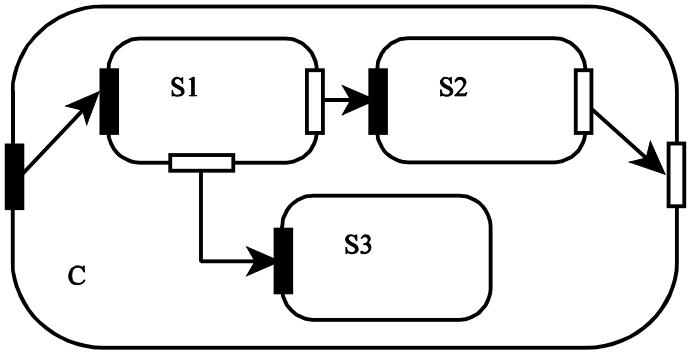

Figure 2 Component nesting

Component nesting usually preserves two typical properties of aggregation: *lifecycle control* and *hiding*. In a majority of component models which allow for component nesting, a compound component controls the lifecycle of its subcomponents (i.e. a compound component creates all of its subcomponents on its startup and it destroys them during its own destruction). With respect to hiding, a compound component hides the interfaces of its subcomponents (only those subomponents' interfaces that are exported by the compound component itself are visible to the outside world).

2.1.3 Component interconnections

Even though components are the main building blocks of the system, the properties of the system depend strongly on the character of the component interconnections. 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. Three basic approaches to specifying component interactions are employed in current component models and their respective ADLs. In compliance with the terminology coined in [6, 29], 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 [23] 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 UniCon language [55] 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 an underlying language or operating system (such as RPC, pipe, etc.). The semantics of a particular interaction is simply defined by the connector type selected to reflect this interaction.

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., 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.

2.1.4 Behavioral specifications

Since the main mechanism of component reuse is composition of existing (mostly off-the-shelf) components, a precise specification of component behavior becomes very important. Several techniques of specifying component behavior have evolved over time, scaling from a documentation written in plain English attached to a component distribution package to some sophisticated formal methods.

One of the oldest forms of component behavior specification (used in e.g. C2 SADEL [27]) is based on enhancing component interfaces with pre- and post-conditions. For a given method of a component interface, pre-conditions enforce the required values of input parameters and put some constraints on the component state before the method invocation; post-conditions make assertions on the values of output parameters and on the component state after the method invocation is completed. However, this approach does not tell anything about ordering of method invocations and it cannot specify a cooperation among multiple component interfaces, which makes this approach inconvenient for a description of component behavior.

A more convenient approach to specifying component behavior is employing various process algebras. These formal methods became domains of interest for a majority of component models originating in the academic area. Among others, the modified Hoare's CSP notation [16] used in Wright, π-calculus employed in Darwin [24], and also the behavioral protocols proposed by SOFA [49] are named in particular.

On the other hand, in the area of the software industry, the Unified Modeling Language (UML) [42] is becoming increasingly popular as a widely accepted standard for specifying software architectures. To that purpose, UML defines a set of diagrams as views describing various aspects of an architecture. The description of behavior involves two aspects - the structural description of the participants and their relationships illustrated by *collaboration diagrams*, and the description of the communication patterns presented by *sequence diagrams*. *Statechart diagrams* represent the behavior of entities capable of dynamic behavior by specifying their response to the receipt of events. *Activity diagrams* are variations of a state machine in which the states represent the performance of actions and the transitions are triggered by the completion of these actions. *Use case diagrams* show actors and use cases together with their relationships. The use cases represent functionality of a system as manifested to external interactors with the system.

2.1.5 Dynamic architectures

Evolution is a typical feature of software. Software systems constantly evolve for many reasons (new requirements on their functionality, performance, reliability, a need for error corrections, etc.). While some of the complex component-based systems require non-stop availability (it is not possible to shut them down for upgrades), a critical issue for component-

based systems design is their support for dynamic architectural changes (the term *dynamic* refers to changes during system runtime). This is because dynamic changes to architectural structure may infer with ongoing computations of the system.

There are many kinds of dynamic changes that can be supported by a component model. They scale from minor bug fixes of component bodies to heavy system reconfigurations (including changes to component interfaces and interconnections).

Concerning current component models, their support for dynamic changes is still very limited today. There are many open issues that concern maintaining system integrity during a runtime change, describing runtime changes via ADLs or similar languages, applying runtime changes, etc.

2.1.6 Deployment

A software system (an application) composed of components can be distributed. Thus before being subject to an execution, each of its components must be deployed into a container/underlying address space (*deployment dock* in general). The aim of deployment is to install a logical component topology to a physical computing environment. Typically, in software industry supported technologies, e.g., [38, 39, 59], a distribution of the system is specified by means of deployment descriptors; however, none of the experimental ADL systems, such as [46, 52, 22, 55, 1], targets the deployment issue directly.

2.2 Related component models

This section very briefly sketches the most remarkable features of related component models with an accent on component interconnections.

2.2.1 ACME

ACME [11, 12] is an architecture description language which was created by Garlan et al. at Carnegie-Mellon University. The original goal of ACME was to create a simple, generic ADL that could be used as a common interchange format for architecture design tools and/or as a foundation for developing new architectural design and analysis tools.

In ACME, core elements of architectural description are (1) implementation independent *components* representing the primary computational elements and data stores of a system with interface points called *ports*, (2) explicit *connectors* representing interactions among components with interface points called *roles*, and (3) *systems* representing configurations of components and connectors.

As for connector support, ACME connectors are user-defined hierarchically structured entities with their semantics described by protocols. Since ACME itself does not provide any direct support for semantics description, it uses the semantics description supports of other ADLs in a form of the protocol properties attached to components and/or connectors.

An example of simple client-server architecture described using the ACME notation follows.

```
System simple_client_server = {
 Component client = {Port send-request}
 Component server = {Port receive-request}
 Connector rpc = {Roles {caller, callee}}
```

```
 Attachments : {
   client.send-request to rpc.caller;
   server receive-request to rpc.callee;
 }
}
```

2.2.2 Aesop

Aesop [10, 33] is a predecessor of ACME created by Garlan at al. at Carnegie-Mellon University. The main goal of Aesop is to serve as a toolkit for rapidly building software architecture design environments specialized for domain specific architectural styles. An architectural style description includes items such as a vocabulary of design elements (components, connectors, and patterns) along with their associated semantics, global design rules, customized visualizations, and other information, if desired. From these inputs, Aesop creates a software architecture design environment that is specialized to support design in the styles that it has taken as input.

The generic style presents only those elements of software architecture that are common to all styles - implementation independent *components* with input and output *ports* as interface points, explicit *connectors* with *roles* as interface points, and *configurations* as compositions of components and connectors into systems.

Aesop allows for creation of user-defined hierarchically structured connector types based on interaction protocols (to that purpose Aesop can use style specific languages for specifying semantics). The Aesop language is not being actively developed at present; emphasis was shifted to ACME instead.

2.2.3 Aster

Aster [17, 7, 68] is a configuration-based development environment proposed by Issarny et al. at IRISA-INRIA Rennes. The aim of Aster is easing both the specification of non-functional requirements within distributed applications, and the construction of corresponding customized middleware. The proposed environment is composed of (1) the Aster language for describing a distributed application in terms of interconnection of possibly heterogenous components, together with the application's non-functional requirements upon the underlying middleware, and (2) a set of tools for constructing customized middleware. The notation used in the Aster language to define application components, their interfaces, and interconnections is based on the notation of Interface Definition Language [37] by OMG. The application's non-functional properties are formally specified in terms of the first order logic, so the software specification matching mechanism [69] can be used to retrieve the components of the middleware customized to the application needs.

Even though connectors are (conceptually) considered as basic architectural elements in Aster, their ultimate goal is to locate a middleware architecture mediating the interaction between application components in a way that satisfies their non-functional requirements. Hence, connectors are not defined explicitly in the architectural description of an architecture, so Aster can be classified as an ADL that uses implicit connections for specifying component interactions.

2.2.4 C2

C2 [46, 28, 45] is the special architectural style designed at the University of California, Irvine. According to this style, a system architecture is a hierarchical network of concurrent components linked together by connectors in accordance with a set of style rules. The basic C2 style rules are depicted in Figure 3. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. The number of components attached to a single connector is not limited. It is also possible to directly connect one connector to another.

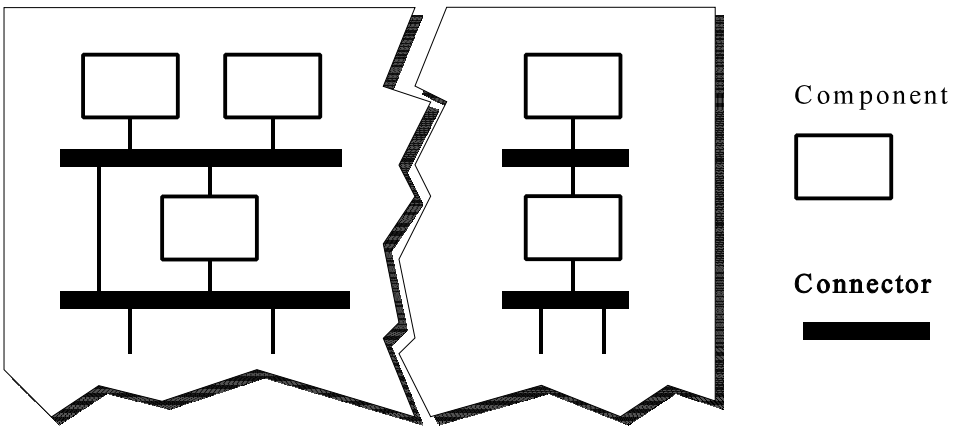

Figure 3 A C2 architecture example

In the C2 style, a communication among components is based on asynchronous exchange of messages through connectors. There are two types of messages: requests (sent explicitly upward through a C2 architecture) and notifications (broadcasted implicitly downward). Therefore, a connector's primary responsibility is the routing and broadcast of messages. A secondary responsibility is message filtering.

To define architectures built according to the C2 style, C2 SADL has been created. An example of a simple stack visualization architecture described using the C2 SADL notation follows.

```
architecture StackVisualizationArchitecture is
    components
        top_most
            StackADT;
        internal
            StackVisualization1;
            StackVisualization2;
        bottom_most
            GraphicsServer;
    connectors
        connector TopConnector is
            message_filter no_filtering
        end TopConnector;
        connector BottomConnector is
            message_filter no_filtering
```

```
        end BottomConnector;
    architectural_topology
        connector TopConnector connections
            top_ports
                StackADT;
            bottom_ports
                StackVisualization1;
                StackVisualization2;
        connector BottomConnector connections
            top_ports
                StackVisualization1;
                StackVisualization2;
            bottom_ports
                GraphicsServer;
end StackVisualizationArchitecture;
```

2.2.5 Darwin

Darwin [23, 14, 13, 24] is an architecture description language developed at Imperial College, London. It describes components as instances of component types. A *component type* specifies the component interface (in terms of *provided* and *required* services) and internal architecture. A component architecture can be either primitive (a component is directly implemented using underlaying implementation language) or composed (a component is formed by a collection of mutually interconnected subcomponents). Darwin supports the description of systems with dynamic architectures through lazy instantiation and explicit dynamic instantiation mechanisms. π-calculus is used to specify component semantics.

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

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

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

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

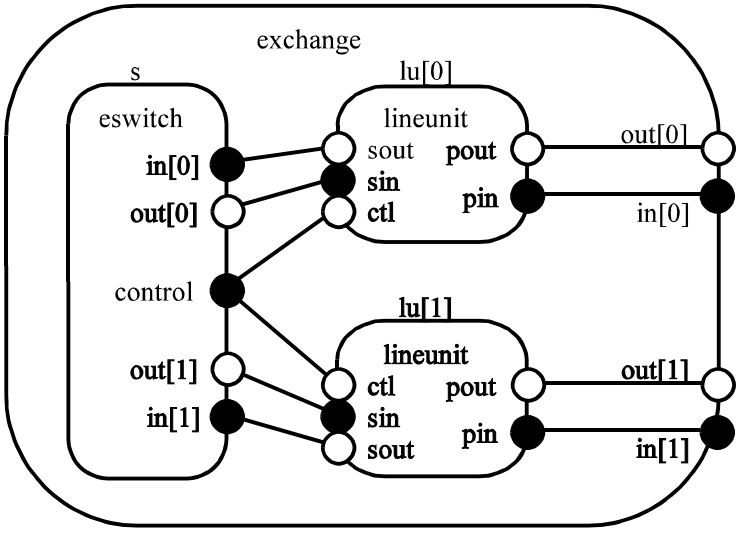

Figure 4 Darwin description of telephone exchange

Interactions among component are specified in terms of direct binding provides and requires interfaces of involved components. The semantics of a connection is defined by the underlying environment, and the communicating components should be aware of it. Darwin language is a typical representative of ADLs that use implicit connections among components.

2.2.6 Olan

Olan [4, 5] is an architecture description language created by Bellissard et al. at INRIA Rhone Alpes as a part of the SIRAC project. Olan is an ADL specially designed for distributed applications. Those applications require from the configuration language the ability to integrate heterogeneous software components in an homogeneous way, have them distributed in various ways across the networked computers according to users' connections, and adapt the communication between components according to the designers' preferences, the components distribution and the middleware characteristics.

Olan is a syntactic extension on OMG's IDL. Components are defined with an interface that describes requirements and the provisions of the component. From the point of view of the control flows, the types of provided services defined by Olan are divided into two parts - the classical services, intended to be called synchronously, and the event-based services which can be executed asynchronously in reply to a notification.

To specify interconnections among components, a set of predefined connectors are offered to the application architect (syncCall, asyncCall, RandSyncCall). Connectors are named in a unique way, each of which corresponds to a communication pattern and an implementation on top of a middleware platform.

2.2.7 Rapide

Rapide [22, 20] is an architecture description language developed at Stanford University by Luckham et al. It focuses mainly on modeling and simulation of dynamic behavior described by an architecture. The simulation is based on partially ordered event sets (posets), poset-based constraints, and event patterns to recognize posets.

The Rapide language consists of several sub-languages. *Type language* is used to describe interfaces to both hardware and software components. *Architecture definition language* defines the synchronization and communication of component interfaces in terms of patterns of events. *Constraint language* uses the time poset model as a basis to provide constructs for defining patterns which are required, or forbidden, within the context of a described system. *Executable language* uses objects, types and common expressions to the type language to provide a control structure.

An architecture described in Rapide consists of component types (*interfaces*) that define synchronous and asynchronous communication elements called *functions* (*provides*, *requires*) and *events* (*in action*, *out action*), in-line arbitrarily complex connectors (*connections*), and constraints. An example of simple component specification written in the Rapide notation follows.

```
type Application is interface
  in action Request(p: params);
  out action Results(p: params);
behavior
  (?M in String) Receive(?M) => Results(?M);
end Application;
```

2.2.8 SADL

SADL [34, 35] is an architecture description language created by Moriconi, Riemenscheider, et al. at SRI International. The SADL language can be used to specify both the structure and the semantics of an architecture. A SADL architecture denotes a logical theory in an extended first-order logic that includes numerical quantifiers (often called ω-logic).

An *architecture* is a (possibly parametrized) collection of components, connectors, and configurations. A *component* represents a computation or a data store. It has a name, a type (a subtype of type COMPONENT), and an interface, the *ports* of the component. A port has a name, a type, and is designated for *input* or *output*. A *connector* is a typed object (a subtype of type CONNECTOR) relating ports. Every connector is required to accept values of a given type on one end and to produce output values of the same type on the other. A *configuration* constrains the wiring of components and connectors into an architecture. It can contain *connections* that associate type-compatible connectors and ports, and *constraints* that are used to relate named objects or to place semantic restrictions on how they can be related in an architecture.

2.2.9 UniCon

UniCon [55, 56] is an architecture description language devised at the Carnegie Mellon University by Shaw et al. It is based on two main constructs: a system architecture is described as a set of typed *components* that interact via *connectors*. UniCon's components roughly correspond to compilation units of conventional programing languages and they are specified by notion of *interfaces*. Connectors, on the other hand, define the rules governing component interactions and do not correspond directly to compilation units. Rather, they are realized as procedure calls, linker instructions, shared data structures, scheduler policies, etc.

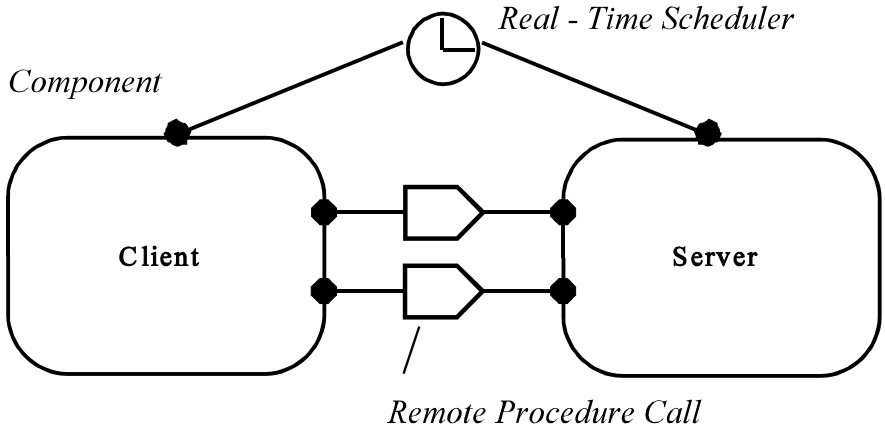

Figure 5 A UniCon architecture example

A connector in the UniCon language is specified by its *protocol*. A connector's protocol consists of the connector's type, specific set of properties, and a list of typed roles. Each *role* serves as a point through which the connector is connected to a component. UniCon is a typical representative of ADLs supporting a predefined set of built-in connector types only. The semantics of built-in connector types are defined as part of the language, and are intended to correspond to the usual interaction primitives supported by underlaying operating system or programming language. UniCon currently supports seven built-in connector types which

represent the basic classes of interactions among components: Pipe, FileIO, ProcedureCall, RemoteProcedureCall, DataAccess, RTScheduler, and PLBundler.

To illustrate the expressiveness of the UniCon language, consider the following simple example. Imagine an architecture (depicted in Figure 5) of a real-time client-server system with two schedulable tasks sharing a computing resource which interact with each other via remote procedure calls. Description of this architecture written in UniCon notation follows.

```
component Real_Time_System
  interface is
    type General
  implementation is
    uses client interface rtclient
     PRIORITY(10)
     ...
     end client
    uses server interface rtserver
     PRIORITY(9)
     ...
     end server
    establish RTM-realtime-sched with
     client.application1 as load
     server.application2 as load
     server.services as load
     ALGORITHM(rate_monotonic)
     ...
     end RTM-realtime-sched
    establish RTM-remote-proc-call with
     client.timeget as caller
     server.timeger as definer
     IDLTYPE(Mach)
     end RTM-remote_proc_call
    ...
  end implementation
end Real-Time-System
connector RTM-realtime-sched
  protocol is
    type RTScheduler
    role load is load
  end protocol
  implementation is builtin
  end implementation
end RTM-realtime-sched
...
```

2.2.10 Wright

Wright [1, 2] is an architecture description language developed at Carnegie Mellon University by Allen et al. It is based on the formal description of the abstract behavior of architectural components and connectors using the modified Hoare's CSP notation [16]. Because the

semantics of Wright specifications is formally defined, an architecture characterized in Wright provides a sound basis for reasoning about the properties of the system described.

In Wright, the description of a component has two important parts, the *interface* and the *computation*. A component interface consists of a number of *ports*, each of them defines a separate interaction in which the component will participate. The computation section of a description describes what the component actually does.

A connector represents an interaction among a collection of components. A Wright description of a connector consists of a set of *roles* and the *glue*. Each role defines the behavior of one participant in the interaction. The connector glue defines how the roles will interact with each other. Connectors in Wright are user-defined.

An example of simple client-server application described using the Wright notation follows:

```
Component Client
  Port p = request -> reply -> p ⊓ §
  Computation = internalCompute -> p.request -> p.reply -> Computation ⊓ §

Component Server
  Port p = request -> reply -> p □ §
  Computation = p.request -> internalCompute -> p.reply -> Computation □ §

Connector Link
  Role c = request -> reply -> p ⊓ §
  Role p = request -> reply -> p □ §
  Glue = c.request -> s.request -> Glue
       □ s.reply -> c.reply -> Glue
       □ §

Configuration Client-Server
  Instances
    C:Client; L:Link; S:Server
  Attachments
    C:p as L.c; S.p as L.s
End Configuration
```

2.2.11 Java component models

There is no need to introduce Java. Everybody using Internet today has heard of this platform evolved by Sun Microsystems, owing to its fantastic boom over the past few years since its creation. By its nature which is built upon bytecode interpreting, Java permits the writing of portable platform independent components that can be easily distributed on the Web. Java currently provides two component models: the basic component model called JavaBeans and its specialization for server-side components called Enterprise JavaBeans.

2.2.11.1 JavaBeans

The JavaBeans specification [58] defines a software component model for Java, so that third party software vendors can create and ship Java components that can be composed together into applications by end users. A *Java Bean* (*bean* for short) is defined as a reusable software

component that can be manipulated visually in a builder tool. Some beans may be simple GUI elements such as buttons and sliders, other beans may be sophisticated visual software components such as database viewers, or data feeds. Some beans may have no GUI appearance of their own, but may still be composed together visually using an application builder.

Individual beans vary in the functionality they support, but the typical unifying features that distinguish a Java Bean are: (1) support for *introspection* so that a builder tool can analyze how a bean works, (2) support for *customization* so that when using an application builder a user can customize the appearance and behavior of a bean, (3) support for *events* as a simple communication metaphor that can be used to connect up beans, (4) support for *properties*, both for customization and for programmatic use, and (5) support for *persistence*, so that a bean can be customized in an application builder and then have its customized state saved away and reloaded later.

2.2.11.2 Enterprise JavaBeans

The Enterprise JavaBeans (EJB) component model logically extends the JavaBeans component model to support server components. Server components (*enterprise beans* or *beans* for short) are reusable, prepackaged pieces of application functionality that are designed to run in an *application server*. They can be combined with other components to create customized application systems.

The EJB specification [59, 60, 61] defines interfaces and required behavior for both - enterprise beans and their containers. A *container* provides a deployment environment that wraps beans during their lifecycle (every bean must live within a container) and provides them with a set of basic services (namely transactions, security, and persistence). Access to a bean is handled by the *home* and *remote* interfaces. While the remote interface defines the bean's *business methods*, the home interface specifies methods for the bean's creation and destruction, as well as so-called *finder methods* - methods for querying the population of beans to find a particular bean instance. By default, both interfaces are accessible via RMI over IIOP protocol [64]. To obtain a reference to the bean's home interface, the Java Naming and Directory Interface (JNDI) [63] can be used. EJB supports distributed flat transactions defined in Java Transaction Service (JTS) [65]. Every client method invocation on a bean is supervised by the bean's container, which makes it possible to manage the transactions according to the transaction attributes that are specified in the corresponding bean's *deployment descriptor* (an XML-based document containing the bean's basic characteristics, usage of the services provided by the container, and requested references to other beans).

There are three types of enterprise beans - *session beans*, *entity beans*, and *message-driven beans*. Session beans are short-lived objects existing on behalf of a single client that do not represent any shared and/or persistent data. Depending upon its conversational state, a session bean can be *stateful* or *stateless*. On the other hand, an entity bean is usually a long-lived, transactional object representing persistent data (usually stored in a database) that can be shared by multiple clients. Entity bean persistence is managed either by the bean itself *(bean-managed persistence*), or it is driven by the container *(container-managed persistence*) in compliance with the *abstract persistence schema* defined in the bean's deployment descriptor. A message-driven bean is in fact a stateless session bean, whose execution is driven by messages delivered through Java Message Service (JMS) [62].

2.2.12 CORBA component model

The CORBA Component Model (CCM) [38, 39, 40] specified by the OMG represents the base of the forthcoming CORBA 3 platform. Unlike EJB which is only intended to be used in the Java environment, the main focus of CCM is to ease the development process of applications made of distributed components that are heterogenous (implemented using different programming languages, running on various platforms). The CCM specification defines four basic models.

The CCM *abstract model* covers specification of component interfaces and their mutual interconnections. The Interface Definition Language (IDL) has been extended for this purpose. A CORBA component can have multiple interfaces (*ports* in the CCM terminology) to either provide or require functionality to/from its clients. Those interfaces support two interaction modes: *facets* and *receptacles* can be used for synchronous method invocations, *event sources* and *sinks* can be used for asynchronous notifications. Like EJB, the CCM abstract model also defines component *homes* - instance managers serving as component factories and finders.

The base of the CCM *programming model* is formed by the Component Implementation Framework (CIF) and its associated Component Implementation Definition Language (CIDL). The main purpose of the CCM programming model is to describe component implementations and their non-functional properties/system requirements (security, persistent state, transactions, etc).

The CCM *deployment model* defines how to assemble an application composed of components, pack it into a software package, and instal the application on various sites. The deployment information is provided in a form of various descriptors (there are four kinds of descriptors defined by the CCM) using the XML vocabulary of the Open Software Description.

The CCM execution model defines containers as a runtime environment for component instances and their respective homes. Several containers can be hosted by a single container server. Containers hide the complexity of the underlying system services such as the POA, transactions, persistence, security, etc.

2.3 SOFA/DCUP component model

In SOFA [48], 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*. A component template is defined by a pair <component frame, component architecture>. A *component frame* of a template T determines component type as the set of interfaces either *provided* or *required* by every instance of T (reflecting a black-box view on the instances of T). To provide a gray-box view on the instances of T, *component architecture* is provided. It describes the internal structure of each of the instances of T in terms of its direct subcomponents and their interactions (interface ties, “wiring”). A component architecture can be specified as *primitive* which means that there are no subcomponents and the frame is directly implemented using an underlying implementation language, such as a set of Java classes, a shared library, or a binary executable file. If a component C is an instance of a template with a primitive architecture, we say that C is a *primitive component*, otherwise C is a *compound component*. To specify component templates, the *SOFA CDL* (Component Definition Language) [31] has been created. *Connectors* as entities describing component interactions in SOFA are the subject of this thesis.

The DCUP architecture is a specific architecture of SOFA components which allows for their safe updating at runtime. It extends the SOFA component model by (1) defining the

ComponentManager interface as a special interface to control the lifecycle of a component and to navigate among its interfaces, and by (2) introducing two specific implementation objects (Component Manager and Component Builder).

With respect to an update operation a component can be divided into a permanent part and a replaceable part. A Component Manager, as an implementation of the ComponentManager interface, is the heart of the component's permanent part, existing thus for the entire lifetime of the component. Its key task is to create and destroy component internals using a Component Builder, and to control the component updates. On the other hand, a Component Builder (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 other internals of the component every time an update occurs. The key task of a Component Builder is to build/destroy the replaceable part of a component including restoring/externalizing the component state whenever necessary. The ComponentManager interface and the ComponentBuilder interface are specified as follows.

```
module SOFA {
  module Component {
    interface ComponentLifecycleControl {
      void createComponent(in Storage stateStore) raises
        (ComponentLifecycleException);
      void destroyComponent(in Storage stateStore) raises
        (ComponentLifecycleException);
      void storeComponent(in Storage stateStore) raises
        (ComponentLifecycleException);
      void restoreComponent(in Storage stateStore) raises
        (ComponentLifecycleException);
      void updateComponent(in string subcomponentName,
        in SOFA::Deployment::DeploymentDescriptor newDescriptor,
        in Storage stateStore) raises (ComponentLifecycleException);
    };

    interface ComponentNavigation {
      Object lookupService(in string name) raises (NamingException);
      void provideRequirement(in string name, Object target)
        raises (NamingException);
    };

    interface ComponentManager : ComponentLifecycleControl,
      ComponentNavigation {};

    interface ComponentBuilder {
      void onArrival(in Storage stateStore) raises
        (ComponentLifecycleException);
      void onLeaving(in Storage stateStore) raises
        (ComponentLifecycleException);
      void store(in Storage stateStore) raises
        (ComponentLifecycleException);
      void restore(in Storage stateStore) raises
        (ComponentLifecycleException);
  };
};
```

Chapter 3

Deployment Anomaly

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. Note that for the rest of the thesis the SOFA component model has naturally been adopted. Even though it differs in some details, it generally follows the basic spirit of most ADL languages.

3.1.1 Design time

Development and provision. At this stage, a component is specified at the level of its template, i.e. component frame and architecture is specified in CDL. Any template with primitive architecture must be accompanied by its implementation in the underlying implementation language/environment. As a frame F can potentially be implemented by several component architectures, each of such templates can be viewed as a design version of components of type F. More formally, several templates based on the same frame can be developed and provided, forming a set of templates $\{<F,A_1>, <F,A_2>, ... <F,A_n>\}$.

The actual specification of an architecture A is always based on the frames of the subcomponents of A (and not on the architecture of those subcomponents). For instance, in Figure 6, 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 – … ". Such a hierarchy is called a *development tree* of an application.

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 6.2.1.1), and the provides interfaces via which the system environment can control an application.

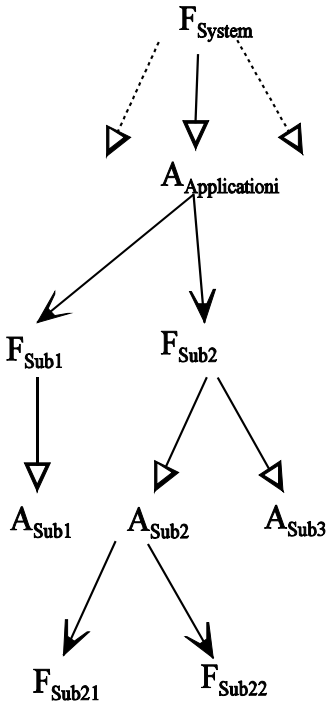

Figure 6 Development tree of an application

Assembly. An application is assembled by choosing one particular component architecture for each frame involved recursively in the topmost frame of the application. More formally, assembling an executable form of the Application^i means reducing its development tree 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 is based on all the primitive architectures involved recursively in the component architecture associated with the topmost frame.

Distribution. An application composed of components can be distributed. To reflect its future distribution, the assembled tree of an application is divided into *deployment units*. The components forming a deployment unit are to be submitted to a single deployment dock for instantiation. Since primitive components can be implemented in a variety of different ways (e.g., a set of Java classes, shared library, binary executable file) it is natural to require a primitive component not to be distributed, i.e., a single primitive component cannot be assigned into more deployment units.

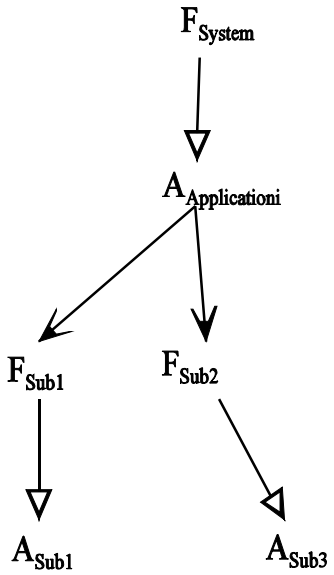

Figure 7 Assembled tree of an application

3.1.2 Deployment time

The goal of the deployment time phase is to achieve *deployment* of the application, i.e., to associate each of its deployment units with a deployment dock, submit all the application components to their relevant deployment docks, and let these deployment docks start the application.

With respect to the deployment of a composed component, the following two approaches are to be considered: (1) Deployment unit boundaries can cross the component interface ties, but not the component/frame boundaries (Figure 8). Advantageously, the deployment description of composed/nested components can be done on a top-down basis, following the hierarchy of components.

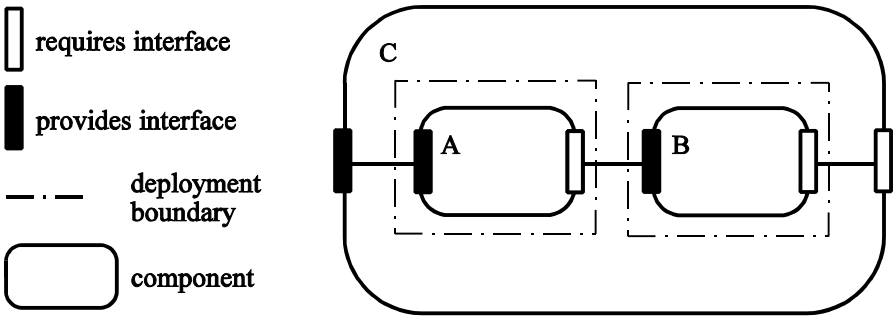

Figure 8 Deployment boundaries crossing interface ties

(2) Deployment unit boundaries are orthogonal to component/frame boundaries. Thus, deployment unit boundaries can cross a component/frame boundary. Assuming that a primitive component cannot be distributed, the respective alternatives are analyzed (see Figure 9) using the components A, B, C and their ties as shown in Figure 8. In the a), b) and c) cases, the deployment

unit 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 deployment unit boundary crosses the boundary of A. If A is primitive, it cannot be divided into more deployment units, i.e. d) is not possible, and therefore the deployment unit 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. With respect to composed components, the following two problems are not easy to overcome: (i) the deployment description must be made on a bottom-up basis and therefore it cannot parallel the hierarchy of component nesting, and (ii) the deployment of a composed component into more deployments docks may be a complex process.

3.1.3 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 which have primitive architecture.

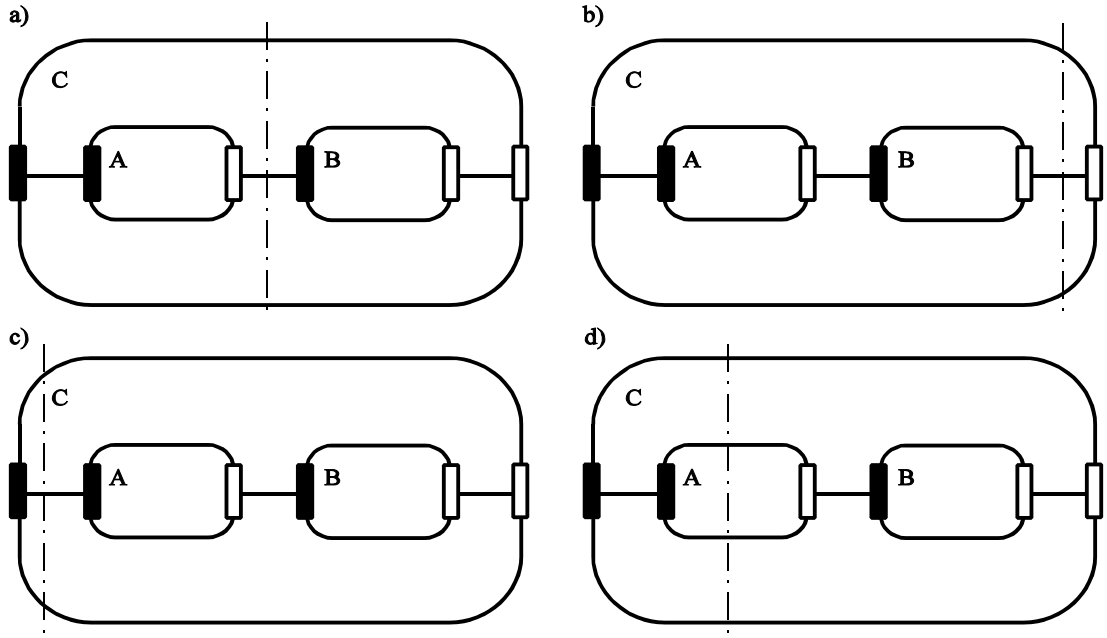

Figure 9 Deployment units orthogonal to components

3.2 Deployment anomaly

Let a distribution boundary cross the interface tie of two components A and B. Since the actual deployment of A and B in general influences the communication of A and B, it is usually necessary to adjust the selected communication mechanisms whenever the deployment of A and B changes. In the case that communication mechanisms are directly hard-coded in components, such adjustments mean modifications of component internals. Analogously with the inheritance anomaly concept [26, 50], we refer to this kind of a post-design modification of a component internals enforced by its deployment as *deployment anomaly*.

As an example, consider components A and B interacting via local procedure calls. Deploying those components to different address spaces, the method calls on the r and q interfaces must be modified in order to use an appropriate middleware technique of remote procedure calls, as can be seen in Figure 10.

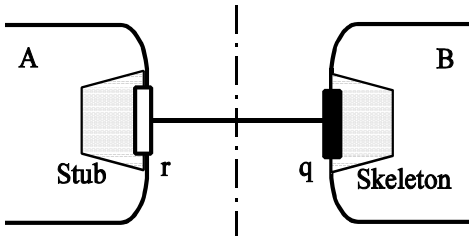

Figure 10 Modifying component internals during deployment

As a quick remedy, one can imagine employing an ordinary component DC mediating the communication of A and B (Figure 11). 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 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 it is assumed that this is not permitted), DC must be considered to be a composed component; this leads to the issue of adjusting the internals of some inner components of DC, similarly to Figure 10.

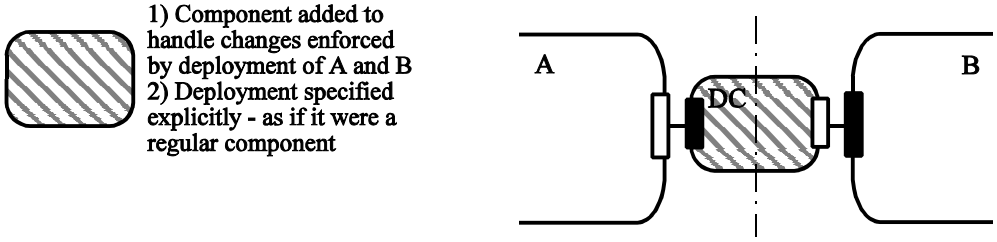

Figure 11 Adding a component to reflect change in deployment

3.3 Targeting the deployment anomaly: connectors

Basically, the deployment anomaly could be addressed by introducing a first class ADL abstraction being (a) inherently distributed and (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: (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) To absorb changes in communication induced by the particular deployment modification, a flexible parametrization system of the connector internals must be provided. (3) To reflect inherent distribution, the deployment of a connector should not be specified explicitly, but should be inferred from the deployment description of the components involved in the communication the connector conveys (Figure 12). As a consequence, the lifecycle of a connector inherently

differs from the lifecycle of a component as its underlying code has to be supplied (e.g. semiautomatically generated) as late as its deployment is known (Section 5.3.3).

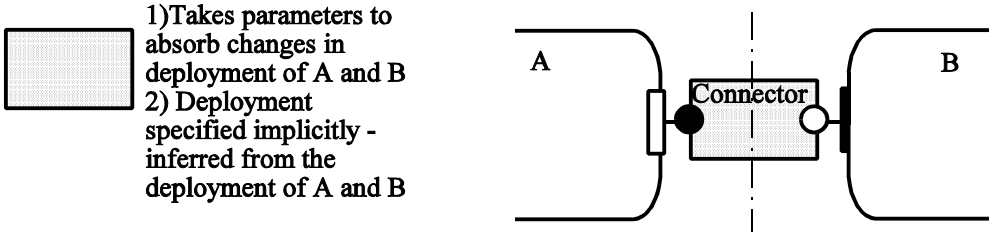

Figure 12 Connector accepting deployment modification

3.3.1 Other pros of the existence of connectors

Besides targeting the deployment anomaly problem, adopting the connector concept to a component model brings some additional benefits. Other pros of connectors' existence can be found by studying related work.

In one of the earliest papers on software architectures [54], Mary Shaw presents many arguments in favor of connectors as first-class ADL entities. According to Mary Shaw, connectors deserve a first-class status because they serve for a better understanding of system architecture.

- Connectors help to localize information about interactions of components in a system (this information is no longer spread over all communicating components and therefore it is easier to change and maintain).
- As practical systems have quite sophisticated rules about component interactions, a connector abstraction is necessary to capture all design decisions so that they can be exploited for analysis or maintenance.
- Connectors provide natural support for components with incompatible packaging (a connector that mediates a communication among components can accommodate incompatibilities of their interfaces).
- Connectors support multi-language and multi-paradigm systems in a natural way.
- Connectors can support dynamic changes in system connectivity (relations among components are not hard-coded in the code of the components).

Additional arguments in favor of using connectors can be found in [46], a shared work of Oreizy, Rosenblum and Taylor.

- Connectors naturally support mobility as they are the only architectural entities that know the network location of individual components.
- As connectors separate communication mechanisms from components, they increase the reusability of components (the same component can be used in a variety of environments with different communication primitives).

Oreizy, Rosenblum and Taylor also argue that even though special component types are sometimes used to represent particular interconnection among components instead of using a connector abstraction, there are slight differences between component and connector types that

make it preferable to have two separate abstractions. The mentioned differences are: (1) Components are usually autonomous elements encapsulating functionality while a connector exists only to serve the communication needs of others, and (2) components roughly correspond to compilation units of conventional programming languages while connectors in general do not.

An important attempt to improve the current level of understanding of the nature of software connectors and their building blocks was undertaken by Medvidovic et al., who presented a classification framework and taxonomy of software connectors [30]. This taxonomy is based on a study and deep analysis of the existing component-based systems. It distinguishes between the four service categories (*Communication, Coordination, Conversion,* and *Facilitation*) and the eight basic connector types (*Procedure call, Event, Data Access, Linkage, Communication Stream, Arbitrator, Adaptor*, and *Distributor).* Additional features of each connector type are expressed in terms of the dimensions of the connector type.

Chapter 4

Requirements of a Connector Model Design: Goals of the Thesis Revisited

On the basis of analysis of related connector models, it was realized that none of the models provides sufficient support to suppress deployment anomaly and it was therefore decided to create a new connector model aimed primarily at solving this problem. In particular, the features lacking in the related connector models can be summarized as follows.

Those component-based systems and their respective ADLs that employ a predefined set of built-in connector types (UniCon is the typical representative) inherently lack a way of describing the whole variety of interactions among components that occur in today's quite often very complex applications. Since predefined connector types are usually selected as elementary enough to directly correspond to some communication primitives of an underlying programming language and/or an execution environment, it is not possible to capture interactions at a higher level of abstraction than is represented by procedure calls and/or sending events. Moreover, none of these languages provides a connector parametrization system flexible enough to cover changes in the deployment of the application (e.g., procedure call and remote procedure call are two different connector types in UniCon).

On the other hand, those component-based systems and their respective ADLs that employ user defined connector types (Wright is the typical representative) allow for capturing a variety of (possibly very complex) interactions among components in applications. However, they do not provide any notion as to how to realize these interactions in the code of the application. Since connectors are not required to map to explicit runtime entities (and they usually do not), it is difficult to localize all the necessary modifications that must be made to the application's code whenever the deployment of the application changes.

For the purpose of solving the deployment anomaly problem, it is necessary to adopt the third approach to specifying component interactions - user-defined connector types with a clear relation to their implementation. When identifying the requirements of our novel connector model, the taxonomy of software connectors presented in [30] was proved to be very useful. The set of requirements that was collected is the subject of the rest of this chapter.

4.1 Functional requirements

Requirements that make assumptions on the functionality of a connector are denoted as *functional*. Those requirements should cover all the basic tasks that connectors are to perform.

4.1.1 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 should specify the mechanisms on which a particular interaction is based (such as procedure call, sending events, data stream, etc.). Each of these mechanisms has specific characteristics and properties. A procedure call can be local or remote. In the case of remote procedure call, various kinds of middleware can be used to implement such a call (CORBA, Java RMI, etc.). Similarly, sending events can be based on an event channel, the publisher-subscriber pattern, a centralized event queue etc. Moreover, for a selected communication technology, a connector can specify its quality of service, i.e. whether a connection is optimized for performance, reliability, security, saving network resources, etc.

4.1.2 Interface adaptation and data conversion

While building an application composed of reusable components, a system developer can encounter the need for connecting two (or more) components that were not originally designed to interoperate. Their interfaces are therefore most likely incompatible. If these interfaces are "similar enough", a possible solution is to mediate such a connection 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 [67], there is the option (and challenge) of devising a mechanism for automatic or semi-automatic generation of all necessary interface adaptors and/or data convertors.

4.1.3 Access coordination and synchronization

In principle, the ordering of method calls on a component's interface is important (the protocol concept in [36]). The appropriate ordering alternatives are usually specified as behavioral specification of the component (e.g., interface, frame and architecture protocols in SOFA [49], 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 illustrating when specification of the ordering of method calls is important, consider several clients, each of whom breaks the required ordering of calls on a server. These clients could be grouped together and synchronized in such a way that the resulting sequence of calls produced by the group complies with the requested ordering on the server. This can be achieved by designing a connector which mediates access to the server component for all of its clients and ensures (and internally implements) the necessary locking.

As another simple example, consider a server component implemented for usage in a singe-threaded environment which is actually deployed in an environment with multiple concurrent client threads. For such a component to work correctly, it is necessary to serialize all the client

threads before entering the component. Again, a connector ensuring the necessary locking can be designed.

4.1.4 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). Such an interception can be used, e.g., 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 supporting mechanism for debugging (e.g., breakpoints).

4.1.5 Dynamic component linking

Since all information concerning component interactions is concentrated in connectors, connectors decouple interacting components from each other (components are not statically bound together). Thus an obvious connector task is dynamic component linking that allows for a variety of dynamic changes to the system architecture and deployment (e.g., component migration, replacement of one component by another, and employing a load balancing mechanism).

4.2 Non-functional requirements

The following set comprises those requirements that are not directly related to the functionality of connectors but rather affect the overall design of the connector model.

4.2.1 User defined ADL entities

A connector should be a first class ADL entity at the same abstraction level as a component. To capture the whole variety of interactions among components, connector types should be user-defined. The complexity of a connector type should follow the complexity of an interaction represented by the connector type. The simplest and most common interaction patterns such as procedure calls and event passing should be represented by primitive connector types whereas more complex interactions should be represented by connector types of higher complexities. To define complex connector types, composition of simpler connector types can be used.

4.2.2 Clear parametrization system

In contrast to components, connectors should be generic entities. There are two chief reasons for this requirement. (1) As interactions of a particular nature usually occur at various places in an application, the instances of the corresponding connector type should be applicable in a variety of contexts. For example, all procedure call interactions among components within an application should be represented by connectors of the same type. Therefore, the connector type representing procedure calls must be parametrized by a signature of the procedures being called. (2) To help solve the deployment anomaly problem, connectors should be flexible enough to accommodate changes in the deployment of the application. For instance, internals of the procedure call connector type must be parametrized to adjust from local to remote procedure calls and vice

versa. To describe a connector's genericity, a clear parametrization system of connector specifications must be provided.

4.2.3 Lightweight ADL notation

According to a widely accepted opinion, use of connectors permits better understanding of architectures, as described in Section 3.3.1. On the other hand, use of explicit connectors as in the classical ADLs (e.g. UniCon, Wright) makes the architecture description of an application more difficult to read in comparison with descriptions written using the ADLs with implicit connections (e.g. Darwin, Rapide). A natural sub-goal of the thesis is to work out a lightweight notation for architecture descriptions that employ explicit connectors. Such a notation should exploit as much of the available implicit information as possible.

4.2.4 Explicit runtime entities

Besides being an explicit ADL entity, a connector should also remain as an explicit entity in an application's runtime. This is especially important keeping the deployment anomaly problem in mind. Since every connector type has its concrete representative in the implementation, it is not difficult to localize the modifications that must be performed upon the code of the application every time the application's deployment changes. No modifications should affect the application's business logic concentrated in components.

4.2.5 Mapping from ADL to runtime

Since connectors should be both ADL and runtime entities, correspondence between the architectural elements and their runtime representation must be clearly articulated. To that purpose, a set of mapping rules must be provided for each primitive connector type, thus expressing how to realize the particular connector types in code. The existence of such rules will simplify the process of adjusting connectors triggered by an application's deployment changes (since the adjustments can be processed (semi-) automatically following the mapping rules).

4.2.6 Various middleware support

When designing a set of mapping rules for a connector type, the underlying middleware, operating system, and/or implementation language support should be utilized as much as possible. The aim of connectors is not to replace the existing communication primitives but rather to build a unified facade on top of them. Thus, the mapping rules for a certain connector type can differ for various underlying platforms.

4.2.7 Clear specification of runtime behavior

With respect to runtime entities, it is necessary to specify certain moments of the runtime behavior of connectors. This includes a connector's instantiation, binding it to components, destruction, etc.The proposed behavior should be in compliance with corresponding rules of the related component model.

4.2.8 Clear relation to deployment framework

Since the mapping rules for a certain connector type can differ for various underlying platforms, the connector code must be supplied to an application as late as the deployment of the application is determined (as mentioned in Section 3.3). Thus, an important aspect of the design of a connector model is its relation to the underlying deployment framework. In particular, this encompasses two issues. (1) It must be clarified how the deployment framework provides the information on the application's deployment. This information is essential to create correct implementations of connectors. (2) It is also necessary to specify how to deploy the connectors themselves.

Chapter 5

Architectural Model for SOFA/DCUP Connectors

To reflect a variety of interactions among components in a hierarchically structured system, a connector model which supports 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 architecture description of the system. A finer granularity of description implies a larger number of components with simpler interactions, while a coarser granularity of description implies a smaller number of components with more complex interactions.

In this chapter, a connector model is proposed which is designed as follows: Every interaction among components in an 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.

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 context of the frame, 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 it is intended to be connected to a requires interface of a component (or to a requires role of another connector). Similarly, a *requires* role serves as an outlet point of the component interaction represented by the connector type instance and it is intended to be connected to a provides interface of a component (or to a provides role of another connector).

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 modified SOFA CDL notation [31] (a proposed enhancement of the SOFA CDL). Note that the `Role` interface is a generic interface (expressed by the keyword `template`) parametrized by an interface type T. The `Role` interface provides all the methods of T and a method for linking a role to other elements (in the `Linkable` interface). The `_sofa_getReference` method returns the corresponding connector's reference (Section 6.3.2.1). The Role interface is the same for both provides and requires roles.

```
template interface Linkable<T> {
  void _sofa_link(in T target) raises LinkException;
};
template interface Role<T> : T, Linkable<T> {
  SOFA::Connector::Reference getReference();
};
```

The number of roles within a connector frame denotes the *degree* of a connector type. For example, the CSProcCall connector type representing procedure call interaction between client and server entities (Section 5.4.1) is the degree of two. More complex interactions among three or more entities are typically represented by connector types of higher degrees.

Similarly to CORBA Relationship Service [41], each role within a connector frame has its *cardinality* specified. The cardinality of a role refers to the number of entities that can be simultaneously tied to a connector instance via the particular role. The cardinality of a role can be one of the four values (*1, 0..1, 1..*, 0..**) expressed by modifiers `optional` and `multiple` in the declaration of the role within a connector frame. By default, when no modifiers are used, the cardinality of the role is specified as *1,* meaning that exactly one entity can be simultaneously tied to the connector instance via this role. Using the modifier `optional`, the cardinality of the role is set to *0..1,* meaning that at most one entity can be simultaneously tied to the connector instance via this role. The role's cardinality *1..** is specified using the modifier `multiple`, denoting that at least one entity can be simultaneously tied to the connector instance via this role. Finally, using both of the modifiers, the cardinality of the role is set to *0..*,* meaning that any number of entities can be simultaneously tied to the connector instance via this role. For example, in the CSProcCall connector type, the role for connecting server entities has its cardinality set to *1* (exactly one entity at a time can play the role of server within the instance of the CSProcCall) while the role for connecting client entities has its cardinality set to *0..** (any number of clients can access the server via the single instance of the CSProcCall).

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 13a). 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; their types are, however, usually generic (both the interface type and property parameters are allowed). The functionality specification for every primitive element type is provided in plain English. The precise specification of the semantics of a primitive element type 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 parametrized 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, SOAP, etc.). For examples of simple connector architectures, we refer the reader to Sections 5.4.1 - 5.4.3.

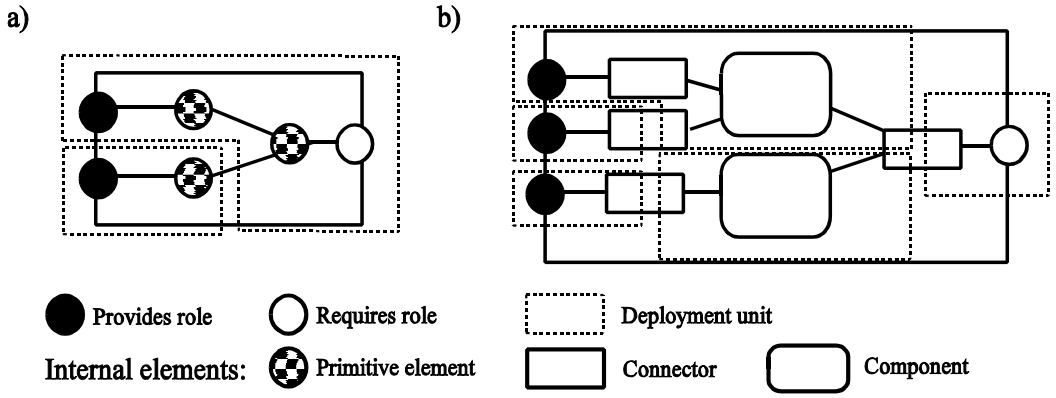

Figure 13 Connector model: a) simple architecture, b) compound architecture

The internal elements of a compound connector architecture are instances of other connector types and components (Figure 13b). 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 (Figure 14). For an example of the compound connector architecture, the reader is referred to Section 5.5.1.

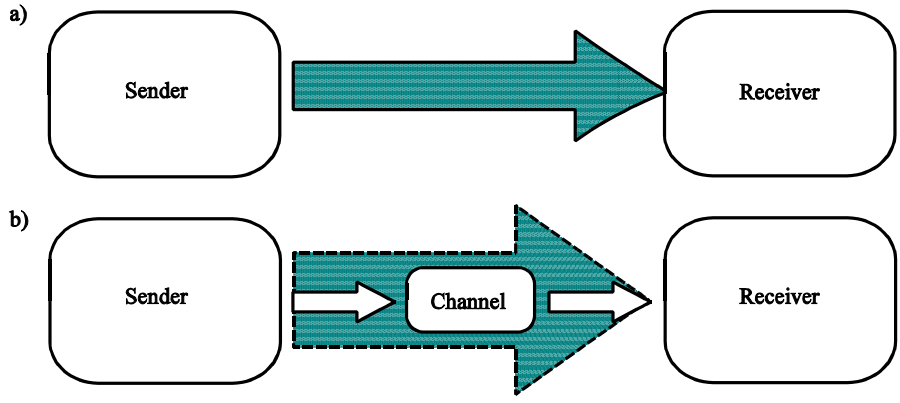

Figure 14 Decomposing an interaction

Similarly to roles, each internal element within a connector architecture has its cardinality specified.

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 provides and requires roles and their generic parameters. As to the connector architecture, specifying compound connector architecture is similar to specifying 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 element types. For each of the primitive element types, its description in plain English must be provided together with a definition of its mappings to concrete underlying environments (at least one mapping must 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 the development process of a connector type is the specification of potential distribution boundaries. This is done by dividing the connector architecture into a number of disjointed *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 consists 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 roles of the connector 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

It is natural that connectors are deployed at the same time as those components of which they represent the interactions. During the deployment phase, each of the deployment units of the connector is assigned a specific deployment dock to be deployed into. The actual deployment dock of the deployment units of the connector can be inferred from the locations of the components interconnected by the connector. However, this is true for connectors with simple architectures only. As connectors with compound architectures may contain component instances as their internal elements, the deployment of these internal components is usually specified separately.

Once the deployment of a connector is known, the implementation code of the connector is (semi-automatically) generated according to the communication primitives offered by the underlying environments of the deployment docks. 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 described in Section 6.2.2.

5.4 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*.

5.4.1 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).

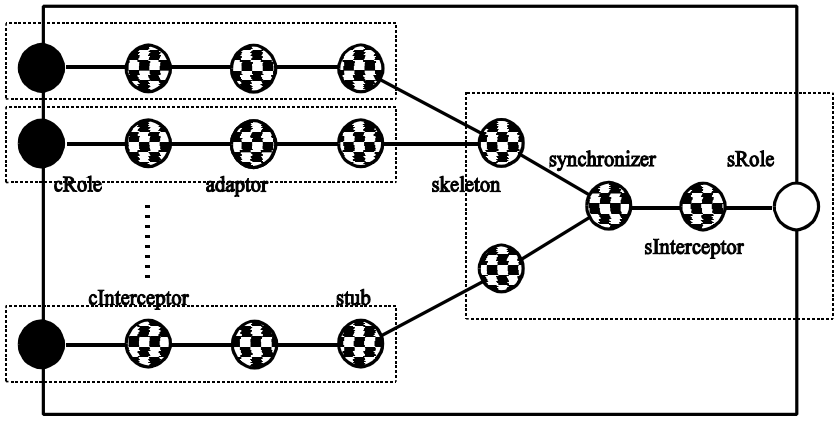

Figure 15 CSProcCall connector type architecture

5.4.1.1 Frame

The CSProcCall frame consists of a requires role cardinality of exactly one to connect a server component (`sRole`), and of a provides role cardinality of any to connect client components (`cRole`). Both roles are generic entities with interface type parameters (the exact role types are specified at the time of connector instantiation in compliance with the concrete client and server component interfaces).

```
connector frame CSProcCall<CT, ST> (Properties properties) {
provides:
  optional multiple Role<CT> cRole;
requires:
  Role<ST> sRole;
};
```

The number of client entities simultaneously connected to a CSProcCall's instance can be furthermore limited by setting the MAX_CLIENTS property of the connector.

5.4.1.2 Architecture

The CSProcCall architecture is simple. It consists of several primitive elements interconnected in the way illustrated in Figure 15.

The `cInterceptor` and `sInterceptor` instances of `TInterceptor` provide a framework for plugging in an additional connector functionality to support logging, debugging, etc. `TInterceptor` is based on a callback notification of the registered (subscribed) entities. For example, using interceptors, one can be notified of communications between clients and server, and modify these communications if the user so wishes, effectively altering the standard behavior of the connector.

```
template interface TInterceptor<T> : T, Linkable<T> {
  void linkNotificationDistributor(in T distributor) raises LinkException;
  void unlinkNotificationDistributor() raises LinkException;
};
```

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. The main task of an interface adaptor is to convert calls between two incompatible interfaces.

```
template interface TAdaptor<T1, T2> : T1, Linkable<T2> {};
```

A (`TStub`, `TSkeleton`) instance pair is used in the case when a remote invocation is needed. These primitive elements provide the standard RPC marshaling/unmarshaling functionality.

```
template interface TStub<T> : T {};
template interface TSkeleton<T> : Linkable<T> {};
```

A `synchronizer` can be included in a connector instance in the case when the server component requires client invocations to be synchronized when accessing its interface.

```
template interface TSynchronizer<T> : T, Linkable<T> {};
```

The CDL description of the CSProcCall connector type's architecture has the following form:

```
connector architecture CSProcCall {
  inst optional multiple TInterceptor<CT> cInterceptor;
  inst optional multiple TAdaptor<CT, ST> adaptor;
  inst optional multiple TStub<ST> stub;
  inst optional multiple TSkeleton<ST> skeleton;
  inst optional TSynchronizer<ST> synchronizer;
  inst optional TInterceptor<ST> sInterceptor;
  delegate cRole to cInterceptor;
```

```
  bind cInterceptor to adaptor;
  bind adaptor to stub;
  bind stub to skeleton;
  bind skeleton to synchronizer;
  bind synchronizer to sInterceptor;
  subsume sInteceptor to sRole;
};
```

5.4.1.3 Deployment

The CSProcCall connector type consists of several deployment units. There is exactly one *server deployment unit* (composed of `sRole`, `sInterceptor`, `synchronizer`, and `skeleton`) and any number of *client deployment units* (each of them composed of `cRole`, `cInterceptor`, `adaptor`, 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 must be deployed into the same deployment dock as the server component).

```
connector units CSProcCall {
  optional multiple Client {cRole, cInterceptor, adaptor, stub};
  Server {sRole, sInterceptor, synchronizer, skeleton};
};
```

5.4.2 EventDelivery

EventDelivery 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).

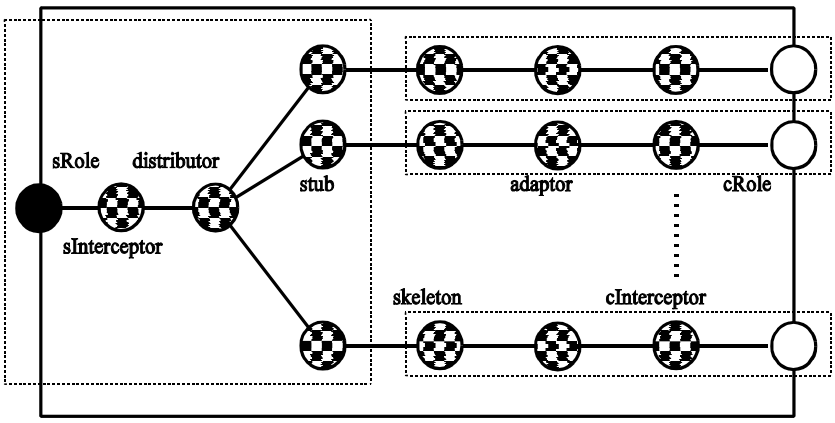

Figure 16 EventDelivery connector type architecture

5.4.2.1 Frame

The EventDelivery frame consists of a provides role cardinality of exactly one to connect an event supplier component (sRole) and a requires role cardinality of any to connect consumer components (cRole). Both 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. Note that by the nature of events, these interfaces should not have methods with output parameters or return values.

```
connector frame EventDelivery<ST, CT> (Properties properties) {
provides:
  Role<ST> sRole;
requires:
  optional multiple Role<CT> cRole;
};
```

The number of consumer components simultaneously connected to an EventDelivery's instance can be limited by setting the MAX_CONSUMERS property of the connector.

5.4.2.2 Architecture

The EventDelivery architecture is simple. It consists of several primitive elements connected as indicated in Figure 16.

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).

```
template interface TDistributor<T> : T, Linkable<T> {
  void unlink(in T target);
};
```

The CDL description of the EventDelivery connector type architecture has the following form:

```
connector architecture EventDelivery {
  inst optional TInterceptor<ST> sInterceptor;
  inst TDistributor<ST> distributor;
  inst optional multiple TStub<ST> stub;
  inst optional multiple TSkeleton<ST> skeleton;
  inst optional multiple TAdaptor<ST, CT> adaptor;
  inst optional multiple TInterceptor<CT> cInterceptor;
  delegate sRole to sInterceptor;
  bind sInterceptor to distributor;
  bind distributor to stub;
  bind stub to skeleton;
  bind skeleton to adaptor;
  bind adaptor to cInterceptor;
  subsume cInterceptor to cRole;
};
```

5.4.2.3 Deployment

The EventDelivery connector type contains several deployment units. There is an exactly one *supplier deployment unit* (composed of `sRole`, `sInterceptor`, `distributor`, and `stub`) and any number of *consumer deployment units* (each of them is composed of `cRole`, `cInterceptor`, `adaptor`, and `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 must be deployed into the same deployment dock as the supplier component, each of the consumer component units must be deployed together with the corresponding consumer component).

```
connector units EventDelivery {
  Supplier {sRole, sInterceptor, distributor, stub};
  optional multiple Consumer {cRole, cInterceptor, adaptor, skeleton};
};
```

5.4.3 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 component.

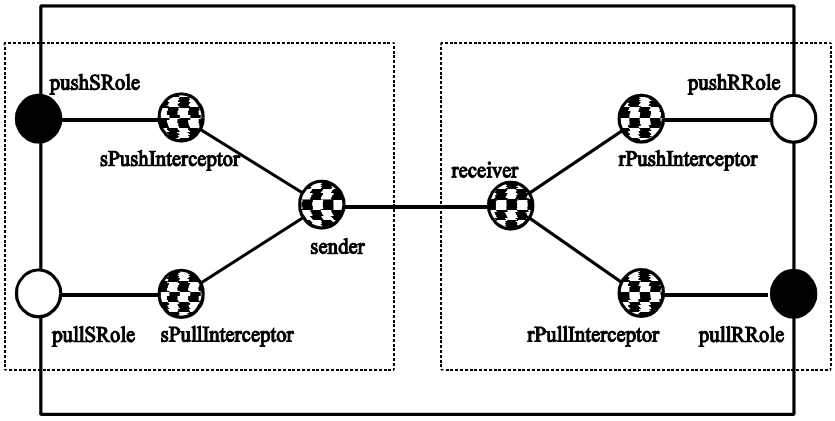

Figure 17 DataStream connector type architecture

5.4.3.1 Frame

The DataStream frame consists of two roles each of them with a cardinality of at most one to connect the sender component in the push or pull mode (`pushSRole` and `pullSRole`), and of two roles each of them with a cardinality of at most one to connect the receiver component in the push or pull mode (`pushRRole` and `pullRRrole`).

```
typedef sequence<byte> bytes;
interface InputStream {
  long read(out bytes buffer, in long len);
```

```
};
interface OutputStream {
  void write(in bytes buffer, in long off, in long len);
};

connector frame DataStream (Properties properties) {
provides:
  optional Role<OutputStream> pushSRole;
  optional Role<InputStream> pullRRole;
requires:
  optional Role<InputStream> pullSRole;
  optional Role<OutputStream> pushRRole;
};
```

The availability of communication modes can be restricted by setting SENDER_MODE and RECEIVER_MODE properties to values PUSH_ONLY or PULL_ONLY.

5.4.3.2 Architecture

The DataStream architecture is simple. It consists of several primitive elements connected as indicated on Figure 17.

The primitive elements `sender` and `receiver` form the pair of elements that specifies the mechanism used to perform the data transmission (TCP protocol, Unix pipe, etc.).

```
template interface Sender<T1, T2> : T1, Linkable<T2> {};
template interface Receiver<T1, T2> : Linkable<T1>, T2 {};
```

The CDL description of the DataStream connector type architecture has the following form:

```
 connector architecture DataStream {
  inst optional Interceptor<OutputStream> sPushInterceptor;
  inst optional Interceptor<InputStream> sPullInterceptor;
  inst Sender sender<OutputStream, InputStream>;
  inst Receiver receiver<OutputStream, InputStream>;
  inst optional Interceptor<OutputStream> rPushInterceptor;
  inst optional Interceptor<InputStream> rPullInterceptor;
  delegate pushSRole to sPushInterceptor;
  delegate pullSRole to sPullInterceptor;
  bind sPushInterceptor to sender;
  bind sPullInterceptor to sender;
  bind sender to receiver;
  bind receiver to rPushInterceptor;
  bind receiver to rPullInterceptor;
  subsume rPushInterceptor to pushRRole;
  subsume rPullInterceptor to pullRRole;
};
```

5.4.3.3 Deployment

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 must be deployed into the same deployment dock as the sender component, the receiver component unit must be deployed together with the receiver component).

```
connector units DataStream {
  Sender {pushSRole, pullSRole, sInterceptor, sender};
  Receiver {pushRRole, pullRRole, rInterceptor, receiver};
};
```

5.5 User-defined connector types

The process of creating a new connector type starts with specification of its frame and architecture. As earlier mentioned, 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.

5.5.1 EventChannelDelivery

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 modes 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.

5.5.1.1 Frame

The EventChannelDelivery connector frame consists of two roles both of them with a cardinality of any to connect supplier components in both the push and pull modes (`pushSRole` and `pullSRole`), and of two roles both of them with a cardinality of any 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.

```
interface Pusher {
  void push(in any data);
};
```

```
interface Puller {
  any pull();
};

connector frame EventChannelDelivery (Properties properties) {
provides:
  optional multiple Role<Pusher> pushSRole;
  optional multiple Role<Puller> pullCRole;
requires:
  optional multiple Role<Puller> pullSRole;
  optional multiple Role<Pusher> pushCRole;
};
```

The availability of communication modes can be restricted by setting SUPPLIERS_MODE and CONSUMERS_MODE properties to values PUSH_ONLY or PULL_ONLY. The number of supplier and consumer entities can be furthermore limited by setting the MAX_SUPPLIERS and MAX_CONSUMERS properties of the connector respectively.

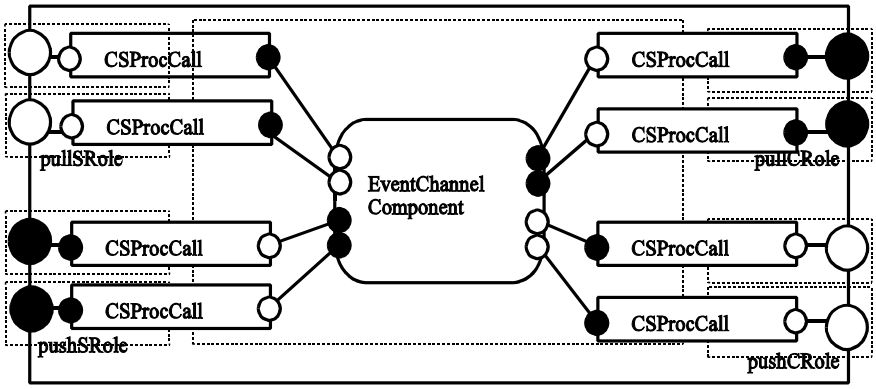

Figure 18 EventChannelDelivery connector type architecture

5.5.1.2 Architecture

The architecture of the EventChannelDelivery connector type is compound. As depicted in Figure 18, the core element of the EventChannelDelivery architecture is an instance of the `EventChannel` component. The other internal elements of EventChannelDelivery are several instances of the CSProcCall connector type to tie EventChannelDelivery's roles to the EventChannel's interfaces. The CDL specification of the EventChannelDelivery architecture follows:

```
connector architecture EventChannelDelivery {
  inst EventChannel channel;
  inst connector CSProcCall hS, lS, hC, lC;
  delegate pushSRole to channel.pushSInterface using hS;
  delegate pullCRole to channel.pullCInterface using lC;
  subsume channel.pullSInterface to pullSRole using lS;
```

```
  subsume channel.pushRInterface to pushCRole using hC;
};
```

5.5.1.3 Deployment

In the 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.

```
connector units EventChannelDelivery {
  Channel {channel};
  optional multiple pushSupplier {pushSRole, hS:Client};
  optional multiple pullSupplier {pullSRole, lS:Server};
  optional multiple pushConsumer {pushCRole, hC:Server};
  optional multiple pullConsumer {pullCRole, lC:Client};
};
```

5.6 Evaluation

This section summarizes the main features of the proposed connector model, describes key decisions that have been made during its design, and evaluates the result.

5.6.1 Connector frame and architecture

As mentioned in Section 2.1.3, studying the related art, two main approaches to specifying component interactions can be identified: (1) an enumerated set of built-in connector types employed e.g. in UniCon, and (2) user-defined connectors employed e.g. in Wright. However, both of these approaches have their own pros and cons. In the first approach, several predefined connector types with a clear relationship to the communication primitives of an underlying environment exist. However, there is no way of specifying a type of interaction that does not correspond to any of the predefined connector types. The main advantage of the later approach relies on the ability to specify all potential kinds of interaction. Such a specification, however, misses any relation to a follow-up implementation.

The approach introduced by the SOFA/DCUP connector model based on the hierarchical structuring of connectors comprises the advantages of both of the above-mentioned approaches and at the same time eliminates their known disadvantages - this approach allows for specifying a connector of any complexity with a clear relation to the implementation. A complex interaction is first of all divided into simpler pieces that cover different aspects of the interaction. These pieces are elementary enough to be easily implemented. The resulting connector is then created by composition of these elements.

The second feature of the SOFA/DCUP connector model that is worth mentioning is a clear system of parametrization of connector types. Since connectors are by their nature template-like constructs reflecting the structure of interactions they represent, absence of a clear parametrization system makes a (possibly automatic) creation of their implementations difficult or even impossible. The SOFA/DCUP connector model introduces two basic kinds of parameters

that can be applied to a connector type - interface type parameters and property parameters. Interface type parameters make connector specifications similar to template-like constructs known from C++ [57] and Eiffel [32]. These parameters usually affect the roles and certain internal elements of the connector (e.g., interface adaptors), and they are resolved at the time of the connector's instantiation within an application's architecture. The property parameters are the classical <name, value> pairs affecting mainly the connector's overall architecture and internals. They are usually resolved at the time of the application's deployment, or they can be set up for the specific run of the application.

5.6.2 Connector lifecycle

Several years ago, when the first ADLs employing explicit connectors appeared, there was a significant disagreement among researchers as to whether connectors are really necessary or not. The problem of classical ADLs is that connectors are very similar to components and therefore the question as to whether it is necessary to have two different abstractions naturally arises. Analyzing both, the typical component lifecycle and the typical connector lifecycle, it has been shown that there is also a substantial difference between the lifecycles of components and connectors. This difference results from a separation of the computation and communication-based parts of a component-based application. Whereas the computation-based part of the application represented by components is deployment neutral, the communication-based part of the application represented by connectors is deployment sensitive (parts of the connector's code can be generated only after the application's deployment has been determined).

5.6.3 Conformance to the requirements

With respect to the functionality of connectors, the following requirements are directly addressed by the architectural model of SOFA/DCUP connectors.

Requirement 4.1.1: Control and data transfer. The proposed connector model clearly specifies both transfer of control and transfer of data. The control is transferred through a connector from its provide roles (the connector's input points) along chains of internal elements to the connector's require roles (the connector's outbound points). The simple connectors are passive elements in application. Whenever an active computation is required to be performed by a connector, an internal component must be employed within the connector's architecture as the source of the connector's activity, i.e., the only sources of activity in applications are components (standalone or employed inside connectors). As for data transfer, `in`, `out`, and `inout` modifiers (known from CORBA IDL) are employed in the connector's interfaces to express the direction of data transfer.

Requirement 4.1.2: Interface adaptation and data conversion. When certain conditions require, a SOFA/DCUP connector can serve as an adaptor that allows for interconnecting components, interfaces of which are directly incompatible. To that purpose special elements (interface adaptors) can be added to the connector's internal architecture to mediate the paths from the connector's provides to requires roles. In such a case, the control and data are propagated along the path of the connector's internal elements from a connector's provides role to an adaptor in a form which is determined by the provides role's interface, then the conversion is made inside of the adaptor, and finally, control and data are propagated along the rest of the path to a connector's requires role in a form determined by the requires role's interface.

Requirement 4.1.3: Synchronization and access coordination. Similarly to an interface adaptation, the problem of synchronization and access coordination is solved inside of SOFA/DCUP connectors by adding special elements (synchronizers) to certain places on the paths from the connector's provides to requires roles. Whereas interface adaptors affect rather a data transfer within a connector, synchronizers are aimed to handle a control transfer within the connector.

Requirement 4.1.4: Communication interception. Also this requirement on the functionality of connectors is met by adding special elements (interceptors) to the certain places in a connector's architecture. While designing interceptor support, we were inspired by the concept of *portable interceptors* used in CORBA [37]. Similarly to CORBA, interceptors in SOFA/DCUP connectors represent hooks positioned in certain places of connectors' architectures that allow users to plug-in an additional custom functionality.

Now, let us look more closely at how the proposed architectural model for SOFA/DCUP connectors meets the requirements for an overall connector model design.

Requirement 4.2.1: User defined ADL entities. Connectors are user-defined ADL entities at the same abstraction level as components in SOFA/DCUP. In principal, there are two possibilities for a user to define a new connector type. The easier one is to define a connector type with compound architecture by simple composition of other existing connector types and possibly components. It is more difficult to define a connector type with primitive architecture, since mapping to an underlying environment has to be provided for all newly introduced primitive elements.

Requirement 4.2.2: Clear parametrization system. The SOFA/DCUP connectors are generic entities that can be parametrized in two ways. Interface type parameters refer to a connector's roles and its certain internal elements (e.g., adaptor). They are resolved at the time of connector's instantiation within an application's architecture. Property parameters refer to the overall architecture of the connector and its certain internal elements (e.g., subs, skeletons, synchronizers). They are usually resolved at the time of the application's deployment, or they can be set up for the concrete run of the application.

Requirement 4.2.5: Mapping from ADL to runtime. The mapping of a connector type with primitive architecture is given by mappings of its primitive elements (at least one mapping must always be provided for each newly introduced primitive element). Since a new connector type with compound architecture is created by composition of existing connector types (and possibly components) each of them having its mapping specified, the mapping of the newly created connector type can be inferred from the mappings of the nested connector types.

Chapter 6

Programming Framework for SOFA/DCUP Connectors

Integrating connectors into the SOFA/DCUP component model made it necessary to enhance the original SOFA/DCUP programming framework to support different phases of connector lifecycle. In this chapter, these enhancements are presented, thus introducing the programming framework for SOFA/DCUP connectors.

6.1 Modified CDL

As mentioned in Section 5.3, the design and instantiation times stand at the beginning of any connector's lifecycle. For both of these phases, CDL specification plays a major role. To allow for specifying new connector types and their instantiation in applications, CDL has been enhanced by the following constructs: template interface definition, connector frame definition, connector architecture definition, connector deployment units definition, named connector instance declaration, and anonymous connector instance declaration. Examples of connector frame and architecture CDL specifications can be found in Sections 5.4.1 - 5.5.1; example instantiations of connectors within an application are presented in Section 7.1.1.

6.1.1 Template interface definition

This construct allows for interface parametrization of generic connector roles and internal elements. An interface marked as `template` accepts a list of type parameters that are used as formal parameters in the interface definition body. Concrete types are substituted for those formal parameters at the connector's instantiation time. The following BNF description specifies the proposed syntax for this construct.

```
<template_interface_def> ::= <template_interface_header>
                             "{" <interface_body> "}"
```

```
<template_interface_header> ::= "template" "interface" <identifier>
                                <template_params> [<inheritance_spec>]
<template_params> ::= "<" <identifier> ("," <identifier>)* ">"
```

6.1.2 Connector frame definition

This construct allows for specification of new connector frames. Syntactically, it is very similar to the construct for specification of component frames with two exceptions. First, a connector frame accepts a list of type parameters that represents an interface parametrization of the particular connector type (Section 5.1). Second, all of the provides and requires interfaces declared within a connector frame are based on the generic `Role` interface described in Section 5.1. The following BNF description specifies the proposed syntax of the connector frame definition.

```
<conn_frame_def> ::= <conn_frame_header> "{" <conn_frame_body> "}"
<conn_frame_header> ::= "connector" "frame" <identifier>
                        [<template_params>] [<properties>]
<properties> ::= "(" <property> ("," <property>)* ")"
<property> ::= <identifier> ":" <string_literal>
<conn_frame_body> ::= [<provides_roles>][<requires_roles>]
<provides_roles> ::= "provides" ":" <role_spec>*
<requires_roles> ::= "requires" ":" <role_spec>*
<role_spec> ::= ["optional"] ["multiple"] <role_type_spec> <declarators>";"
```

6.1.3 Connector architecture definition

This construct allows for a connector architecture specification. Note that a connector architecture can be either simple or compound. Simple architecture can be composed of primitive elements only; compound architecture is composed of components and instances of other connector types. In both cases, the interface parameters of the connector type (specified as a part of the connector frame) can be used as formal parameters within the body of the connector architecture definition. The following BNF description specifies the proposed syntax for connector architecture definition.

```
<conn_architecture_def> ::= <conn_architecture_header>
                            "{" <conn_architecture_body> "}"
<conn_architecture_header> ::= "connector" "architecture" <scoped_name>
<conn_architecture_body> ::= <conn_architecture_elemet>*
<conn_architecture_element> ::= <conn_instance_spec>
                              | <component_instance_spec>
                              | <bind_def>
                              | <delegate_def>
                              | <subsume_def>
```

6.1.4 Connector deployment units definition

This construct is used to specify the deployment units of a connector type. A deployment unit is specified as enumeration of the roles and those internal elements designed to share the same deployment dock. When specifying deployment units of the connector type with a compound architecture, a notation *connector_instance_name:unit_name* can be used to refer to a particular deployment unit of a nested connector. Similarly to roles and internal elements of a connector architecture, also deployment units can be denoted as optional and/or multiple. The following BNF description specifies the proposed syntax.

```
<conn_units_def> ::= <conn_units_header> "{" <unit_def>* "}"
<conn_units_header> ::= "connector" "units" <scoped_name>
<unit_def> ::= ["optional"] ["multiple"] <identifier>
               "{" <unit_members> "}"
<unit_members> ::= <unit_member> ("," <unit_member>)* ")"
<unit_member> ::= <identifier>
                | <identifier> ":" <identifier>
```

6.1.5 Named connector instance declaration

This construct is used to create a named connector instance within a component and compound connector architecture definition bodies (remember that in SOFA/DCUP, an application is represented by a top-level component). The following BNF definition specifies the proposed syntax for named connector declaration.

```
<conn_instance_spec> ::= "inst" ["optional"] ["multiple"] "connector"
                         <scoped_name> <declarators> ";"
<bind_def> ::= "bind" <component_interface_list> "to"
               <component_interface_list> "using" <identifier> ";"
             | ...
```

Note that in a connector instance declaration, each role defined within the connector frame must be assigned with a corresponding component interface. Provides connector roles are tied to requires component interfaces in the order of their appearance within the connector frame specification, i.e. the first provides role defined in the connector frame specification is tied to the first requires component interface used in the connector instance declaration; the second provides role defined in the connector frame specification is tied to the second requires component interface used in the connector instance declaration, etc. Requires connector roles are tied to provides component interfaces in a similar fashion. The roles declared as `optional` within a connector frame can be assigned with the *null* value at the corresponding position in the connector instance declaration.

6.1.6 Anonymous connector instance declaration

This construct is used to create an anonymous instance of a connector type within component and compound connector architecture definition bodies. Note that every connector instance has its identification within the SOFA/DCUP runtime system. The term anonymous denotes that the connector instance is only declared by its type and an implicit identification (generated by the

SOFA/DCUP runtime system) is associated with it. The proposed syntax for anonymous connector instance declaration follows.

```
<bind_def> ::= ...
             | "bind" <component_interface_list> "to"
               <component_interface_list> "using" <scoped_name> ";"
```

6.2 SOFA deployment framework

Event though it is rarely targeted by other component models, deployment is an important phase in the lifecycle of many component-based applications as is described in the section on deployment anomaly (Section 3.2). This is especially true with respect to connectors that represent the deployment-sensitive part of an application. Since connectors are strongly tied to an underlying environment, their implementation code must be generated after the deployment of the application is known. The SOFA deployment framework has been designed to take this fact into account.

6.2.1 Architecture overview

The SOFA deployment framework is formed by a set of *deployment docks* running on various network locations. The main tasks of the docks are to load, instantiate, and run application components. The component code is obtained from a SOFAnode's *template repository (TR)* [94] whereas the runtime information on a component type is provided by the SOFAnode's *type information repository (TIR)* [92]. As a SOFA application could be distributed, each of its components can run within a separate deployment dock. On the other hand, a single deployment dock can simultaneously run multiple components (possibly belonging to different applications). Components are assigned to their respective deployment docks via a *deployment control tool*.

Having a component design, the SOFA deployment framework itself represents a kind of component-based application. Its building blocks (deployment docks, deployment control tools, a type information repository, etc.) are components that interact with each other using connectors. However, as the main task of the SOFA deployment framework is to serve as an underlying environment for other (ordinary) SOFA applications, the bootstrap mechanism of the SOFA deployment framework differs from the bootstrap mechanism of an ordinary SOFA application.

6.2.1.1 Deployment docks

Deployment docks are bases of the SOFA deployment framework. Primarily, they serve as component containers - they are able to load, instantiate, and run individual SOFA components of various formats (binary executables, shared libraries, Java classes, etc.). For components running within, a deployment dock represents an underlying environment (a programming language and operating system platform). However, the SOFA deployment docks are more than ordinary component containers. In addition to common resources of the underlying environments, deployment docks also provide the necessary communication mechanisms that form the bases of the application's connectors. Therefore, the secondary responsibility of deployment docks is a (semi-) automatic connector code generation. This task will be described with all the necessary details in Section 6.2.2.

One can imagine a wide scale of different deployment docks: Java Virtual Machine instances able to load, instantiate, and run components written in Java, various processes able to load dynamically linked native libraries and instantiate components within their own address spaces, various daemons that instantiate components by starting new processes from binary executables, etc. An example of a more complex deployment dock could be the dock representing a cluster of computers managing load balancing by migration of running components within this cluster.

From the SOFA point of view, all deployment docks can be seen as components implementing the following frame:

```
module SOFA {
  module Deployment {
    frame DeploymentDock {
      provides:
        ComponentFactory componentFactory;
        CCGNegotiation connectorGenerator;
      requires:
        SOFA::TIR::Repository typeInformationRepository;
        SOFA::TemplateRepository templateRepository;
    };
  };
};
```

To make both essential parts of the SOFA deployment dock's functionality accessible (component container and connector code generator), two basic interfaces are provided by each SOFA deployment dock - the `ComponentFactory` and `CCGNegotioation` interfaces. While the `CCGNegotiation` interface will be specified later on in the section on connector code generation (Section 6.2.2.1), the `ComponentFactory` interface is specified as follows:

```
module SOFA {
  module Deployment {

    struct TechnologyProperty {
      string name;
      string value;
    };
    typedef sequence <TechnologyProperty> TechnologyProperties;

    struct TechnologyDescriptor {
      string id;
      TechnologyProperties props;
    };
    typedef sequence <TechnologyDescriptor> TechnologyDescriptors;

    exception DeploymentException {
      string reason;
    };

    interface ComponentFactory {
      TechnologyDescriptors describeUnderlyingEnvironment();
```

```
      void install (in SOFA::TIR::ComponentInstanceDef comp, in string url)
        raises (DeploymentException);
      ComponentManager instantiate(in DeploymentDescriptor dd)
        raises (DeploymentException);
    };
  };
};
```

The semantics of the provided methods is following:

`TechnologyDescriptors describeUnderlyingEnvironment()`
This method is used to obtain information about a dock's underlying environment (which is typically used to decide whether the deployment dock is able to run a component of a particular format or not).

`void install(...)`
The purpose of this method is to install a particular component type to the deployment dock.

`ComponentManager instantiate(...)`
This method makes an instance of a particular component type within the deployment dock and returns a reference to its `ComponentManager` interface (see Section 2.3).

As the SOFA deployment framework is a special case of component-based application with deployment docks serving as components, the deployment dock's `ComponentFactory` and `CCGNegotiation` interfaces are accessible via the CSProcCall connector instances. Thus it is possible to use a variety of protocols (CORBA IIOP, Java RMI, SOAP, etc.) to make calls upon them, and so it is possible to use a variety of lookup methods (CORBA Naming Service, JNDI, etc.) to find them on a network. However, because of bootstrapping reasons, those connectors are not instantiated in an ordinary (semi-) automated way, but they are created by *deployment dock finders* from a manually supplied code.

6.2.1.2 Deployment control tools

To control the deployment process of SOFA applications, dedicated tools are used. Instead of relying on one particular deployment control tool, SOFA, preserving its openness, rather specifies a minimal set of requirements that can be met by a variety of deployment control tools. In this way a wide range of tools can be used in the SOFA environment and users (*application deployers*) are provided with a selection of deployment control tools which best fit the needs of the applications being deployed.

Concerning the selection of a deployment control tool, some applications could be shipped together with specialized tools created to control their deployment (the deployment process of the application is hard-coded in such a deployment control tool). For instance, CORBA component model [38, 39, 40] acquires this way of deployment control employing the `Assembly` objects. On the other hand, a set of generic deployment control tools could exist at the same time, able to deploy any SOFA application. Some of these tools could interpret various scripting languages (e.g. OpenCCM [25] uses the CORBA script to control application deployment), others could interact directly with the user via a graphical interface.

6.2.1.3 Application deployment process

The aim of the application deployment process is to assign the logical component topology of the application to a physical computing environment. The deployment process is initiated by the application deployer who (using a selected deployment control tool) assigns concrete deployment docks to each of the application's deployment units. The information is collected in deployment descriptors - XML-based documents containing all important information on the application's deployment.

After that, the implementation code of all the application's components must be made available to the respective deployment docks. In SOFA, deployment docks are able to obtain the runtime information on a selected component type from a SOFAnode's *type information repository* and its implementation code from a SOFAnode's *template repository*. It is the responsibility of the deployer to store the necessary information and implementation code in these repositories.

The last step of the application deployment process is to let the selected deployment docks generate an implementation code of the application's connectors.

6.2.2 Connector code generation

The task of connector code generation follows a simple scenario: (1) Using a deployment control 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 control 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 implementation code for this technology. (5b) If no matching technology exists, the user is given an option either to modify the application's deployment, or to provide the connector implementation manually.

As can be seen from the proposed scenario, the SOFA deployment framework has to provide the following functionality: (1) To be able to negotiate on connector code generation, deployment control tools and deployment docks must support the unified negotiation protocol. (2) To allow for connector code generation, each deployment dock must be supplied with a connector code generator.

Note that only connectors with primitive architectures are concerned when talking about connector code generation. Connectors with compound architectures are created by composition of their internal elements (instances of other connector types and components).

6.2.2.1 Negotiation protocol

The protocol for negotiating connector code generation reflects the scenario presented in the previous section. The negotiation process is driven by the deployment control tool that represents an interface to the user controlling the application deployment. The deployment control tool is the only active participant in the negotiation; other participants - deployment docks - passively perform the actions requested by the deployment control tool. The negotiation protocol specifies

(1) the `CCGNegotiation` interface that must be implemented by each deployment dock, and (2) the requested order and semantic description of the actions that must be performed by the deployment control tool.

The `CCGNegotiation` interface is specified as follows:

```
module SOFA {
  module Deployment {

    struct ElementTechnologyDesciptor {
      string elementName;
      TechnologyDescriptors technologies;
    };
    typedef sequence <ElementTechnologyDescriptor>
      ConnectorTechnologyDescriptor;

    struct ElementTechnologySelection {
      string elementName;
      string selectedTechnologyID;
    };
    typedef sequence <ElementTechnologySelection>
      ConnectorTechnologySelection;

    exception GenerationException {};

    interface CCGNegotiation {
      ConnectorTechnologyDescriptor canGenerate(
        in SOFA::TIR::ConnectorInstanceDef conn, in string unitName);
      SOFA::Connector::DeploymentDescriptor generate(
        in SOFA::TIR::ConnectorInstanceDef conn, in string unitName,
        in ConnectorTechnologySelection selectedTechnology)
        raises GenerationException;
      void useExternalCodeLocation(in SOFA::TIR::ConnectorInstanceDef conn,
        in string unitName, in ConnectorTechnologyDescriptor technology,
        in string url);
    };
  };
};
```

Deploying a component, a deployment control tool is required to perform the following sequence of actions to negotiate a code generation of every connector that is tied to the component being deployed. For each of these connectors, the method `canGenerate` is called first on the selected deployment dock passing the description of the connector as a parameter in the form of a reference to the corresponding `SOFA::TIR::ConnectorInstanceDef` object. A `ConnectorTechnologyDescriptor` is returned to the deployment control tool as the result of this action. It consists of a sequence of `ElementTechnologyDescriptors`, each of them describing a single internal element of the connector - the available technologies offered by the deployment dock (and its underlying environment) on which the generated implementation could be based. Each of the available technologies is described by its `TechnologyDescriptor` which encapsulates an identification together with a set of technology properties (name - value pairs). The properties are used to find matches on technologies.

As the second action determined by the negotiation protocol, the deployment control tool either asks the deployment dock to generate on its own the part of a connector that is going to be deployed in the deployment dock (by calling `generate`), or it provides the deployment dock with a manually created connector code by supplying a locator to an external storage from where the connector code is prepared to be loaded (calling `useExternalCodeLocation`).

Note that the user interface of a deployment control tool is not part of the SOFA specification and therefore it could be proprietary to every deployment control tool vendor.

6.2.2.2 Extensible connector generator (ECG)

To allow for connector code generation, each deployment dock is supplied with a connector code generator. In general, deployment dock vendors are free to provide their own solutions that produce connector codes compatible with the proposed SOFA/DCUP connector model. Nevertheless, we provide the Extensible Connector Generator (ECG) that can be used in deployment docks as the default implementation of a connector code generator directly implementing the `CCGNegotiation` interface.

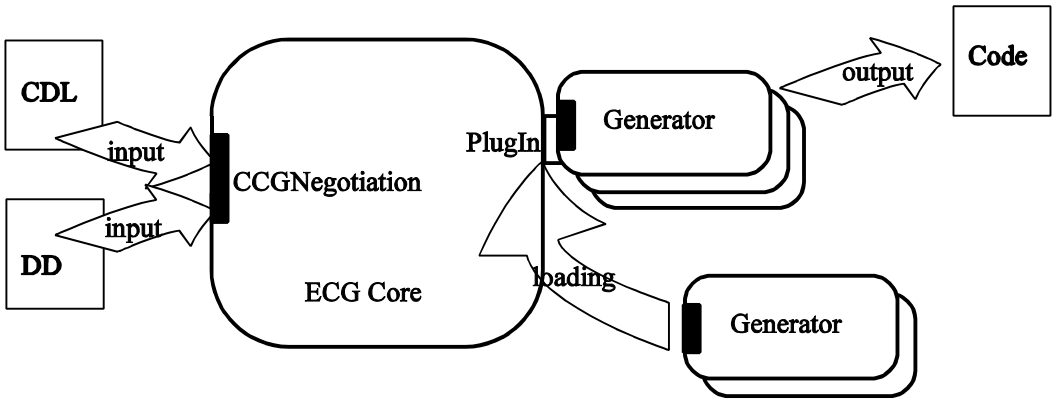

Figure 19 Extensible Connector Generator

The internal architecture of the ECG (Figure 19) reflects the main requirement imposed on its design - extensibility. The ECG architecture should be easily extensible to allow for processing a variety of connector types (even those that were not known at the time of the ECG creation).To respond to this requirement, the ECG has a simple core part that dynamically loads separated modules as connector type-specialized code generators (each connector type has its own generator). Because of its dynamic nature, Java is used to implement the ECG.

When generating connector implementations, the core part of the ECG first contacts the SOFA Type Information Repository for information related to the connectors. For each connector instance, according to its type, the corresponding generator module is located and asked to produce the necessary code. Such a generator module reflects the mapping rules of the corresponding connector type to a concrete underlying environment.

Generator modules are loaded into the ECG's runtime as separate plug-ins on demand. When the ECG's core detects a new connector type used in the application, it will look for the generator module of that type. First, it will look for a class named *<typename>CodeGenerator*. For instance, the generator module of the CSProcCall connector type is called *CSProcCallCodeGenerator*. If the correct generator module does not conform to this naming schema, it must be registered in the ECG's configuration file *generators.cfg*. If the core of the

ECG cannot find the correct generator module for a given connector type, an exception is raised. It is still possible to code the connector manually.

Every plug-in module to the ECG should implement the interface *PlugIn*. This interface states the contract for interaction between a plug-in module and the ECG's core.

```
module SOFA {
  module Deployment {
    module ECG {

      struct RoleInfo {
        string roleName;
        SOFA::TIR::InterfaceDef interfaceType;
      };
      sequence <RoleInfo> RoleInfos;

      interface PlugIn {
        void setRoleInfos(in RoleInfos infos);
        void setTechnologyPropertyValue(in string name, in string value);
        void generate(in SOFA::TIR::ConnectorInstanceDef conn,
          in string unitName)
          raises(SOFA::Deployment::GenerationException);
      };
    };
  };
};
```

6.2.3 Deployment dock case studies

As mentioned in Section 6.2.1.1, a variety of different deployment dock implementations can potentially exist in the SOFA deployment framework environment. Two of them have been selected as the simple concrete proof-of-the-concept implementations.

6.2.3.1 Simple Java

Simple Java is a deployment dock's proof-of-the-concept implementation able to load, instantiate, and run SOFA/DCUP components written in Java. The Simple Java deployment dock architecture is depicted on Figure 20. The key element of the Simple Java container part is the single instance of ComponentFactory. To load the components' code from a SOFAnode Template Repository, the ComponentFactory cooperates with a SOFAClassLoader. The SOFAClassLoader is also used to load the code of connectors previously generated by the instance of the Extensible Connector Generator (described in Section 6.2.2.2). Note that the whole Simple Java deployment dock together with all loaded components runs on behalf of a single Java Virtual Machine.

To generate the implementation code of connectors that are going to be deployed in it, Simple Java uses an instance of the Extensible Connector Generator (ECG). By default, the Simple Java deployment dock provides an implementation of those plug-in modules to ECG that arrange the generation of predefined connector types instances (instances of the CSProcCall, EventDelivery, and DataStream connector types). As the whole Simple Java deployment dock is written in Java,

all these plug-in modules produce a connector code that uses Java as the underlying environment (the generated connector code has a form of Java classes).

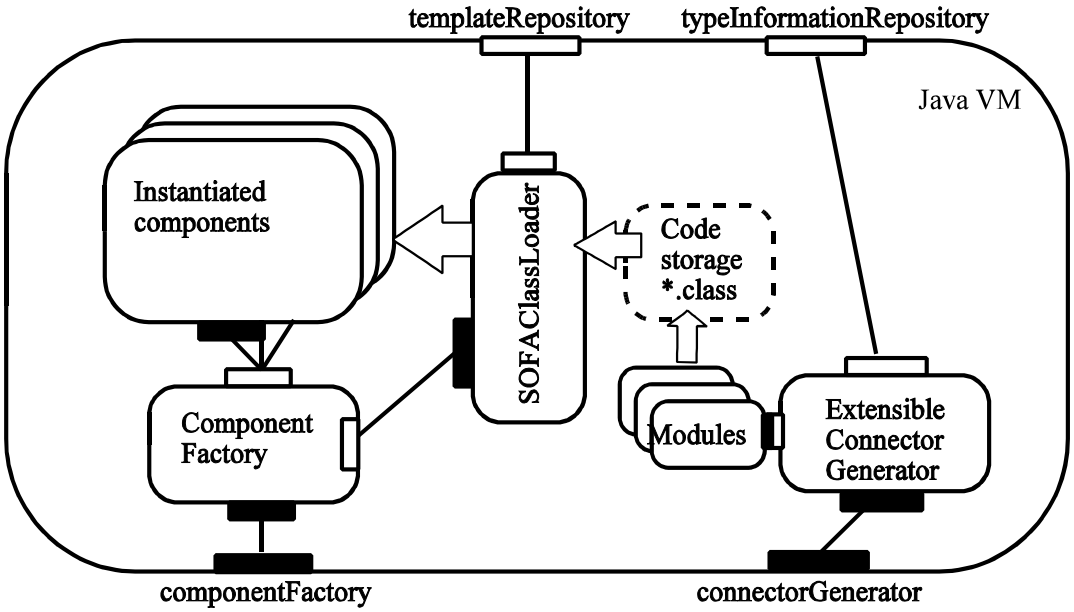

Figure 20 Simple Java deployment dock's architecture

6.2.3.2 Native Activation Daemon

The Native Activation Daemon is a deployment dock's proof-of-the-concept implementation which is able to load and run native binary executables as SOFA/DCUP components. Single component instance running within this deployment dock is represented by the single child process of the Native Activation Daemon.

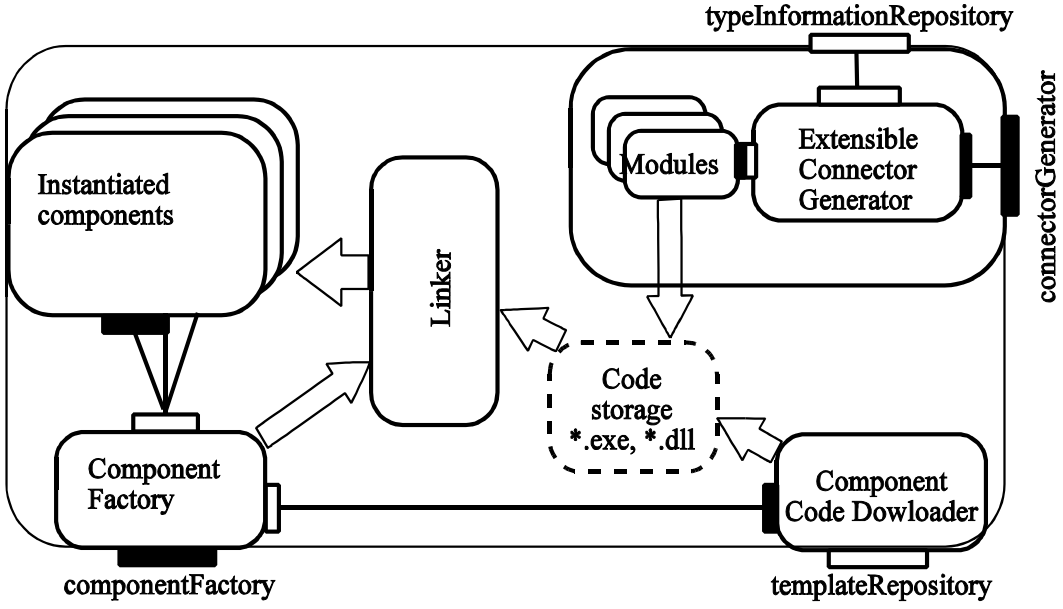

Figure 21 Native Activation Daemon deployment dock's architecture

The Native Activation Daemon architecture is depicted in Figure 6.2.3.2. It is implemented by three processes: the Component Code Downloader (used to load components' code from a SOFAnode's Template Repository), the Extensible Connector Generator (used to generate connectors), and the ComponentFactory (used to instantiate components). When instantiating a component, the ComponentFactory cooperates with a linker to put together the component (in the form of a binary executable) with generated connectors (in the form of shared dynamically linked libraries).

6.3 Runtime support for SOFA/DCUP connectors

Embedding of the deployment dock and connector concepts into the SOFA component model has an apparent reflection in two necessary modifications of the SOFA runtime. Both of them concern startup of an application mainly: (1) Each of the application components must be instantiated within the particular deployment dock in correspondence to the application's deployment descriptor. (2) Connectors must be instantiated while connecting the application components. The following piece of code illustrates instantiation of two components within their corresponding deployment docks as well as making a connection between their interfaces. Actions that happen beyond this code are described in the rest of this section.

```
ComponentFactory factory = FactoryFinder.getFactory(dock1URL);
ComponentManager cmA = factory.instantiate(componentADeploymentDescriptor);
factory = FactoryBinder.getFactory(dock2URL);
ComponentManager cmB = factory.instantiate(componentBDeploymentDescriptor);

Object service = cmA.lookupService("ServiceName");
cmB.provideRequirement("ServiceName", service);
```

6.3.1 Instantiating a SOFA component

According to the SOFA component model specification, a SOFA component instantiation consists of three steps: instantiating the component's frame, providing references to all its required services, and instantiating the component's internal architecture. When instantiating a SOFA component, a connection to its target deployment dock's `ComponentFactory` interface must be obtained first of all (Figure 22). As this interface is accessible via CSProcCall connector instance, various RPC mechanisms can be used to invoke its methods.

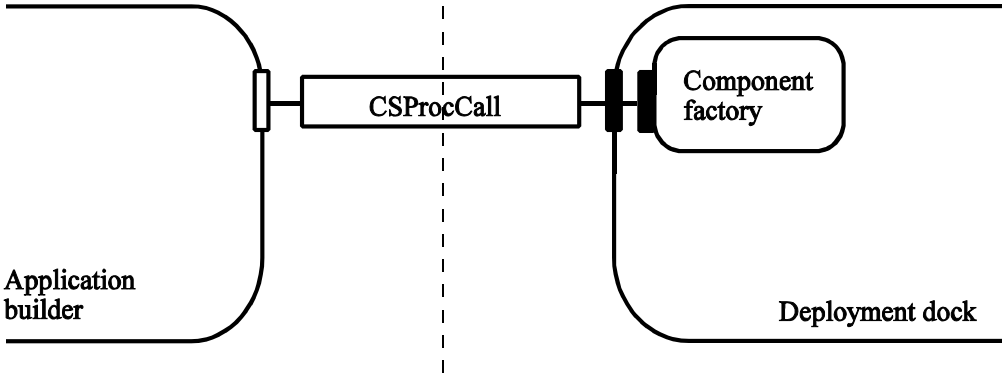

Figure 22 Obtaining a reference to a deployment dock's ComponentFactory

Afterwards, the component frame is created by calling factory method `instantiate` passing the component's deployment descriptor as a parameter. As the result of this operation, the reference to the `ComponentManager` interface (see Section 2.3) of the newly created component frame is obtained. Remember that the frame of every SOFA component contains a single instance of this interface to provide methods for managing the component's lifecycle and navigating among all the component's interfaces (Section 2.3). The `ComponentManager` interface of the newly created component frame is accessible via the CSProcCall connector instance (Figure 23).

After providing references to all services required by the component, the internal architecture of the component is created by invoking `createComponent()` on the `ComponentManager` interface.

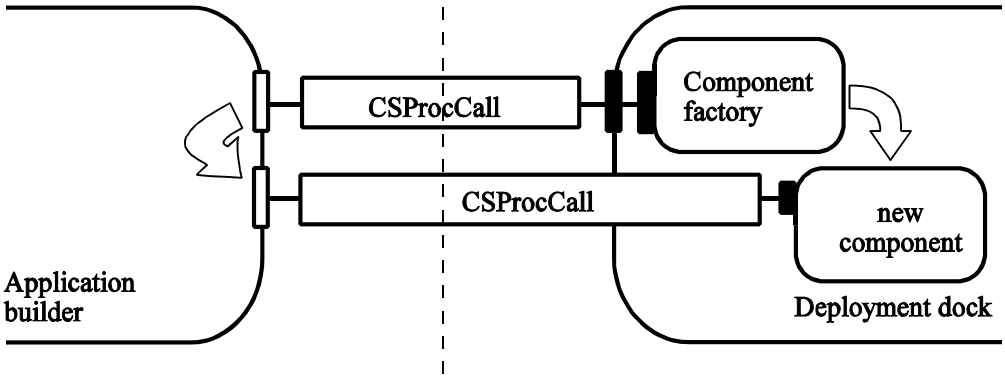

Figure 23 Instantiating a component's frame

6.3.2 Interconnecting SOFA components

With a component frame already instantiated within a deployment dock, access to any of its interfaces can be obtained by invoking one of the navigation methods provided by the corresponding `ComponentManager` interface.

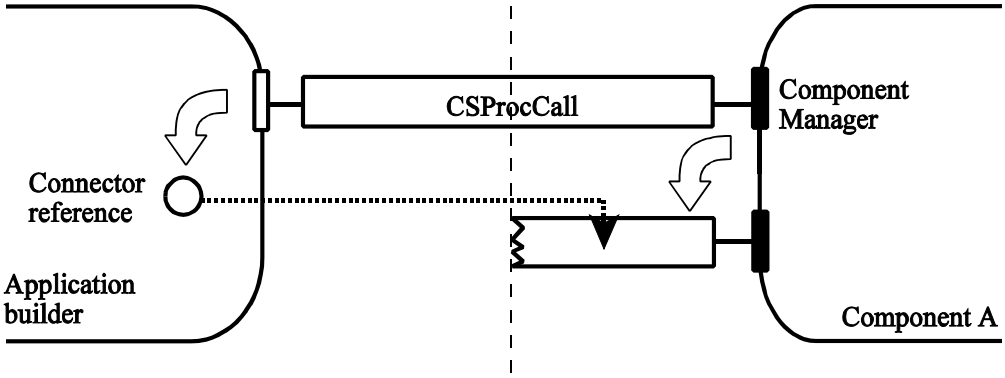

Figure 24 Component interface lookup

To obtain a reference to an interface provided by the component, the `lookupService` method is invoked on the `ComponentManager` interface. This method finds the interface being looked up and gets the connector instance mediating an access to it (note that each component interface

is accessible via a connector instance only and that only one connector instance can be tied to a particular component interface at a time). If the corresponding connector does not yet exist, it is created (respectively, to be more precise, the deployment unit of the connector instance that is tied to the interface being looked up is created). Finally, a reference to this connector instance is returned (Figure 24).

To provide a reference to a service required by a component, the `provideRequirement` method is invoked on the `ComponentManager` interface passing a connector reference as a parameter. This method finds the corresponding requires component interface and gets the connector instance tied to it. If the corresponding connector does not yet exist, it is created (respectively, to be more precise, the deployment unit of the connector instance that is tied to the requires component interface is created). Finally, a reference to the connector which is mediating access to the requested service provided by the component interface is passed to the part of the connector tied to the requires component interface and the requested connection is established (Figure 25).

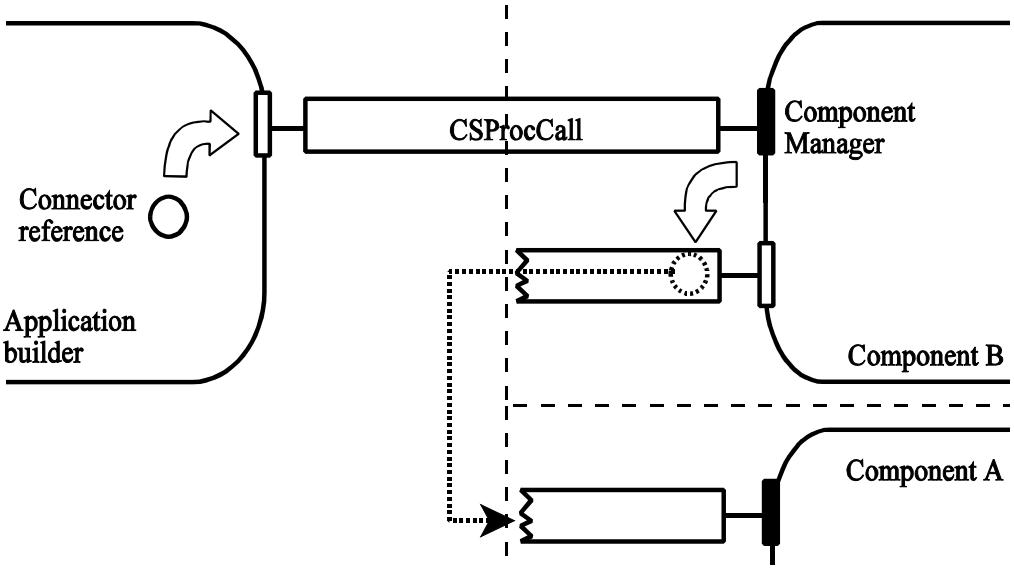

Figure 25 Providing reference to a service required by a component

6.3.2.1 Connector reference

As can be seen from the previous section, connector references play one of the key parts in establishing a connection between two or more SOFA components. Similarly to CORBA Interoperable Object Reference (IOR), it is represented by a data structure the content of which is opaque to the user (its content is handled only by the SOFA runtime). Most of the time, a connector reference resides inside a connector instance which it represents and it is used by the SOFA runtime for the purpose of establishing new connections only.

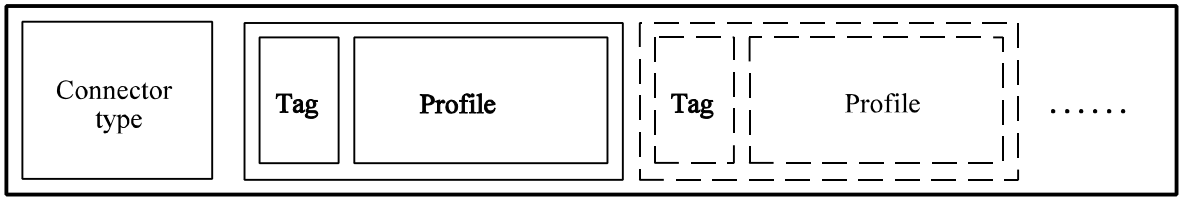

Figure 26 Internal structure of a connector reference

A connector reference can be obtained from the connector instance by calling the `_sofa_getReference` method on any of its roles. Its internal structure is depicted in Figure 26. It consists of a connector type identification followed be a sequence of tagged profiles. These profiles are specific to a particular connector type and only instances of that type know how to use them in the connection establishing process. The following text represents the connector reference specification written in SOFA CDL.

```
module SOFA {
  module Connector {
    valuetype TaggedProfile {
      string tag;
    };

    struct Reference {
      string connectorType;
      sequence <TaggedProfile> profiles;
    };
  };
};
```

6.3.2.2 Connector reference case studies

As mentioned in the previous section, connector reference internals differ from connector type to connector type which they represent (tagged profiles contained in a connector reference are specific to the particular connector type). To provide the reader with examples, the connector references of two SOFA predefined connector types have been selected.

The first case study presents the internal structure of CSProcCall's connector reference. All CSProcCall connector references contain string `"SOFA::CSProcCall"` as `connectorType`. Tagged profiles contained in the reference represent various possibilities of how to access and make calls upon the server interface. The `LocalProfile` denotes a way to contact the server interface for clients residing within the same address space. It contains string `"Local"` as the profile's tag followed by a local reference to the target interface. The `CORBAProfile` denotes a way for clients to contact the server interface via CORBA. It contains string `"CORBA"` as the profile's tag followed by CORBA IOR of the corresponding server's skeleton. The `RMIProfile` denotes a way for clients to connect the server interface via the Java RMI mechanism. It contains the string `"RMI"` as the profile's tag followed by URL of the server's skeleton. If another

mechanism to contact the server's interface appears in the future, a new profile will be created in a similar fashion.

```
module SOFA {
  module Connector {
    module CSProcCall {

      valuetype LocalProfile : SOFA::Connector::TaggedProfile {
        Object target;
      };

      valuetype CORBAProfile : SOFA::Connector::TaggedProfile {
        string IOR;
      };

      valuetype RMIProfile : SOFA::Connector::TaggedProfile {
        string URL;
      };
    };
  };
};
```

The second case study presents the internal structure of the DataStream connector reference. All DataStream connector references contain string `"SOFA::DataStream"` as `connectorType`. Tagged profiles contained in the reference represent various ways of establishing a point-to-point connection between sender and receiver entities to transmit data. The `LocalProfile` denotes a way of establishing a data stream connection for a sender residing within the same address space as the corresponding receiver. It contains string `"Local"` as the profile's tag followed by a reference to a buffer used for data transmission. The `PipeProfile` denotes a way for senders and receivers residing on the same host which supports the pipe communication. It contains string `"Pipe"` as the profile's tag followed by the open pipe's file descriptor. The `TCPIPProfile` denotes a way for data transmission via the TCP/IP protocol. It contains the string `"TCPIP"` as the profile's tag followed by the sender's host name and the listener's port number.

```
module SOFA {
  module Connector {
    module DataStream {

      valuetype LocalProfile : SOFA::Connector::TaggedProfile {
        Object buffer;
      };

      valuetype PipeProfile : SOFA::Connector::TaggedProfile {
        FileDescriptor fd;
      };

      valuetype TCPIPProfile : SOFA::Connector::TaggedProfile {
        string hostName;
```

```
        int portNumber;
      };
    };
  };
};
```

6.3.3 Accessing SOFA components from non-SOFA clients

When accessing the functionality of a SOFA component, the usual scenario is to gain access to the component's `ComponentManager` interface and then to use this interface to lookup the target interface providing the requested functionality. As a procedural interface, the `ComponentManager` interface of every SOFA component is accessible via the CSProcCall connector instance and therefore various middleware technologies (CORBA, Java RMI, etc.) can potentially be used to access its functionality. However, one of the typical features of procedural interfaces is that references to other interfaces can be passed as arguments and/or return values during their method calls (the ComponentManager `lookupService` method is one example). The problem is how to pass these references.

By default, a reference to a component interface is passed in a form corresponding to the SOFA connector reference. As described in Section 6.3.2.1, such a reference encapsulates all possible identifications of the target interface to allow for accessing it in a variety of ways. This approach ensures high flexibility in connection establishing (for example, invoking the ComponentManager `lookupService` method via Java RMI can result in obtaining a CORBA reference to the requested interface). On the other hand, it sometimes brings unwanted overhead caused by wrapping the existing middleware technologies. Furthermore, it also prevents non-SOFA clients from accessing a SOFA component's functionality. For example, although pure CORBA client is able (under certain circumstances) to invoke the ComponentManager `lookupService` method, it is not able to understand the SOFA connector reference obtained as the return value. Therefore, a dual mechanism of passing component interface references is needed.

The dual mechanism of passing interface references is based on the following simple rule for mapping the stub and skeleton internal elements of CSProcCall connectors. Having an interface method `M` that passes interface reference as its argument and/or return value, it is represented by the pair of methods (`M` and `_sofa_M`) in the generated stub and skeleton internal elements. `M` passes interface references in a form specific to a concrete middleware technology employed, while `_sofa_M` passes interface references in the form of SOFA connector references. For example, consider the above-mentioned lookupService method. Its CDL specification is as follows:

```
Object lookupService(in string serviceName);
```

In the generated CORBA skeleton internal element, the `lookupService` method is represented by the following method pair.

```
org::omg::CORBA::Object lookupService(in string serviceName);
SOFA::Connector::Reference _sofa_lookupService(in string serviceName);
```

By calling `lookupService`, both SOFA enabled and non-SOFA enabled clients can obtain a CORBA reference to the requested interface. The `_sofa_lookupService` method permits SOFA enabled clients to obtain a SOFA connector reference containing all known identifications of the target interface.

Similarly, the generated Java RMI skeleton internal element contains the following method pair to represent the `lookupService` method.

```
java.rmi.Remote lookupService(java.lang.String serviceName);
SOFA.Connector.Reference _sofa_lookupService(java.lang.String serviceName);
```

By calling `lookupService`, both SOFA enabled and non-SOFA enabled clients can obtain a Java RMI reference to the requested interface. The `_sofa_lookupService` method serves for SOFA enabled clients to obtain a SOFA connector reference containing all known identifications of the target interface.

6.4 Evaluation

This section summarizes the main features of the proposed programming framework, describes key decisions that have been made during its design, and evaluates the result.

6.4.1 SOFA CDL modifications

When the decision to integrate connectors into the SOFA/DCUP component model was made, the prime intention was to keep the SOFA/DCUP programming framework simple. This applies especially to the proposed modifications of the SOFA CDL. Current ADLs that employ connectors to specify component interactions (UniCon, Wright, etc.) usually provide constructs to explicitly instantiate a connector of the given type and to bind a particular component interface to the corresponding connector role. However, this notation makes the application architecture descriptions more complicated and difficult to read compared to the architecture descriptions written in ADLs that use implicit connections. For instance, consider two simple components (Client and Server) interacting via procedure calls. Using the UniCon notation, this can be expressed by the following code:

```
uses aClient interface Client
  ...
end aClient
uses aServer interface Server
  ...
end aServer
establish RTM-remote-proc-call with
  aClient.iface as caller
  aServer.iface as definer
  ...
end RTM-remote_proc_call
```

The same architecture is expressed using the Wright notation by the following code:

```
Component Client
  Port p = request -> reply -> p ⊓ §
  Computation = internalCompute -> p.request -> p.reply -> Computation ⊓ §

Component Server
  Port p = request -> reply -> p □ §
  Computation = p.request -> internalCompute -> p.reply -> Computation □ §

Connector Link
  Role c = request -> reply -> p ⊓ §
  Role p = request -> reply -> p □ §
  Glue = c.request -> s.request -> Glue
       □ s.reply -> c.reply -> Glue
       □ §

Configuration Client-Server
  Instances
    C:Client; L:Link; S:Server
  Attachments
    C:p as L.c; S.p as L.s
End Configuration
```

On the other hand, the modified SOFA CDL's main concern is to make the information concerning which component interfaces are bound together immediately visible. A connector used to represent a particular connection is usually specified using the anonymous connector declaration only (remember that correspondence between component interfaces and connector roles is determined implicitly as described in Section 6.1.5):

```
inst Client aClient;
inst Server aServer;

bind aClient.iface to aServer.iface using CSProcCall;
```

The second unusual feature of the modified SOFA CDL that is worth mentioning is genericity of connector roles. Usually, ADLs employ connectors with roles that are strictly typed. This approach puts some additional requirements on types of component interfaces tied to these roles (a component interface has to conform to a certain role type usually by inheriting some base interface). Besides a necessity of frequent typecasting, this approach also has a negative impact on component reusability.

On the other hand, our approach tries to adapt connector roles to component interfaces by generating roles on demand from generic templates. Increased component reusability is, however, paid for by an increase in the amount of generated code (which is a common drawback of all template constructs).

6.4.2 SOFA deployment framework

Since none of the well-known component models originating in the academic area [23, 55, 1] directly targets deployment issues, examples of deployment framework must be found in the area of software industry. Among the recent projects dealing with deployment of software

components, the J2EE platform by Sun Microsystems with its Enterprise Java Beans [59] and Object Management Group's CORBA Component Model [38, 39, 40] are probably today's most frequently discussed standards. Both of them introduce *component containers* as architectural elements designed to load, instantiate and run software components, and to provide them with the necessary services as the underlying runtime environment. Among other services, both EJB's and CORBA CCM's component containers provide a running component with access to an Object Request Broker. This allows the outside world to communicate with the running component by remotely invoking its methods. However, particular containers support particular ORBs only thus restricting the communication abilities of the contained components. For instance, a component running inside a CORBA component container has access to CORBA ORB and therefore the only protocol the component is able to use for communication with other components is IIOP, the native protocol of CORBA. On the other hand an Enterprise Java Bean running inside an EJB component container has access to Java RMI ORB only and therefore all its communication with the outside world is based on Java RMI (supporting thus IIOP and/or the RMI native protocol of remote communication). No other kind of communication (data streams, etc.) is supported by these component containers.

The SOFA/DCUP deployment dock concept enriches an ordinary component container with an extensible connector code generator. This approach allows deployment docks to provide components running inside with a wide selection of technologies that can be used for communication with the outside world. Moreover, a new technology can be dynamically added to the set of technologies maintained by the particular deployment dock by simply extending the deployment dock's connector generator with a new plug-in module. This has a positive impact on component availability and reusability.

6.4.3 Runtime support for SOFA/DCUP connectors

The main issue of the SOFA/DCUP connectors' runtime support relies on passing component interface references. As the main feature of the SOFA/DCUP deployment framework is making SOFA/DCUP components accessible simultaneously using various communication mechanisms and middlewares, the runtime should not restrict the selection of the mechanisms used to communicate with the component.

There are several possibilities for solving the problem of reference passing, each of which has its pros and cons. The first solution is based on the existence of "*multi-references*" created by wrapping the existing references valid in various middlewares into a new composite data structure. The main advantage of this method is its flexibility - each client is provided with a selection of underlying middleware used to communicate with a component by selecting the part of a multi-reference representing the component reference valid within the selected middleware. This mechanism does not, however, allow the underlying middlewares to be fully exploited (since middleware specific reference passing mechanisms are replaced by the general mechanism). Moreover, this method also prevents certain clients (those that do not understand the format of the multi-references) from using services provided by the components.

The second possible solution relies on employing a kind of hand-shaking mechanism to agree on a middleware used to communicate with a component while establishing a connection to it. This approach has the advantage of full use of the selected middleware's features paid for by the overhead while establishing connections. Similarly to the first solution presented, this approach also has the disadvantage of preventing certain clients (those that do not support the hand-shaking mechanism) from using services provided by the components.

The third solution (selected to be used in the SOFA/DCUP runtime) is based on the existence of dual access methods (a general one that employs "multi-references" to pass interface references and a middleware specific one that uses middleware specific techniques of reference passing). When using this approach, a client is provided with a selection of proper reference passing modes and thus it is the client's responsibility to solve the trade-off between flexibility and communication overhead (providing a unified facade on top of the existing middleware platforms inherently requires a heavy copying of data, as described in [19]). Moreover, in contrast to previously mentioned solutions, services provided by SOFA components can be used by all (even non-SOFA enabled) clients.

6.4.4 Conformance to the requirements

The following requirements of a connector's model design are directly addressed by the proposed programming framework for SOFA/DCUP connectors.

Requirement 4.2.3: Lightweight ADL notation. Unlike the classic ADLs employing explicit connectors (e.g. UniCon, Wright), the proposed extension to SOFA CDL uses simpler notation to express component interactions which makes the information concerning what component interfaces are bound together immediately visible. A connector used to represent a particular connection can be specified using the anonymous connector declaration in which the correspondence between component interfaces and connector roles is determined implicitly.

Requirement 4.2.4: Explicit runtime entities. Even though the SOFA/DCUP connectors are explicit user defined ADL entities capable of representing a variety of interactions among components, unlike other ADLs with user-defined connectors (e.g., Wright), SOFA/DCUP preserves its connectors to the application runtime. For each connector used within an application, there is a clear relation between its ADL description and runtime representation. This feature of our connector model allows users to concentrate the interaction code (that is inherently sensitive to the application's deployment changes) within the explicit entities - connectors. This suppresses the deployment anomaly problem since all modifications due to deployment changes are localized to connectors and the application's business logic concentrated in components is not affected.

Requirement 4.2.5: Mapping from ADL to runtime. The implementation code for a particular connector is created from the ADL description by applying the corresponding mapping rules. This can be done either manually (a typical case for newly created and rarely used connector types) or automatically (in the case of the most common and repeatedly used connector types). To allow for an automatic creation, a plug-in to the proposed Extensible Connector Generator must be available for the corresponding connector type. The automatic connector generation further simplifies the task of modifying connectors whenever the application's deployment changes.

Requirement 4.2.6: Various middleware support. To utilize an underlying execution environment support as much as possible, several mappings for a particular connector type can exist, each of them optimized for a different underlying platform. To that purpose, also the proposed deployment framework introduces deployment docks (as abstractions upon concrete execution environments) equipped with generators capable of creating the connectors' code which best fits the corresponding environment.

Requirement 4.2.7: Clear specification of runtime behavior. The proposed programming framework for SOFA/DCUP connectors precisely describes a mechanism of instantiating connectors within an application. A connector's creation is described step-by-step from creation

of all of its deployment units, binding them to the respective connector interfaces, to the process of interconnecting these units. Although it is not explicitly mentioned in the thesis, a connector instance is destroyed at the time of the destruction of components tied to it. More precisely, connectors are destroyed in per deployment unit manner, i.e., destruction of a component causes destruction of the connectors' parts (bounded by connectors' deployment units) directly tied to the component.

Requirement 4.2.8: Clear relation to deployment framework. To fit to an underlying execution environment, the connectors' implementation code is supplied to an application at the time of the application's deployment. The proposed deployment framework helps the application's deployers to comply with this task. (1) Deployment docks provide information on technologies that are available for automatic generation of a certain connector type. (2) Deployment docks are able to automatically generate an implementation code for certain connector types. (3) The application's deployer always has an option to supply his/her own implementation code for any connector type. As for deployment of connectors themselves, in contrast to components deployment of which is specified explicitly using deployment descriptors, deployment of connectors is determined implicitly from the deployment of components tied to them.

Requirement 4.1.5: Dynamic component linking. The proposed programming framework also provides an answer to the question of how SOFA/DCUP connectors handle dynamic component linking. Since components are decoupled by connectors that mediate all interactions among them, connectors represent a natural place for applying changes in the components' connectivity without letting the components know about those changes. For instance, by adding a special interceptor to an instance of CSProcCall connector type, incoming procedure calls can be easily redirected to another server component. Even a simple load balancing mechanism can be implemented in this way.

Chapter 7

Proof-of-the-Concept Application: Banking Demo

This chapter consists of a case study that illustrates the basic mechanisms used to design, deploy, and run distributed, component-based applications in the SOFA/DCUP environment. The Banking demo represents a “quick start” to writing, deploying, and running SOFA/DCUP applications and serves as a simple proof-of-the-concept implementation.

The scenario for this “quick start” application involves a bank in which a number of tellers serve a potentially huge number of customers. Each customer requests a teller to perform desired transaction(s) on proper account(s). Certain transactions, such as an overdraft, require tellers to apply for a supervisor's approval. On demand, construction and activity of the running application can be monitored using specialized visualization tools.

7.1 Application design

Every SOFA/DCUP application’s lifecyle starts with the design time. A user creating a SOFA/DCUP application usually follows these steps: 1) The application’s CDL specification is created. 2) The application’s glue code is generated using the SOFA CDL compiler and all components with primitive architectures are provided with their implementation in the underlying language (for its simplicity, Java has been selected for the demonstration purposes within this text). 3) The executable form of the application is assembled by choosing one particular component architecture for each frame involved in the application. 4) Finally, the assembled application is divided into deployment units.

7.1.1 CDL specification

The first step in creating a SOFA/DCUP application is to specify the application’s desired architecture in terms of mutually interconnected components using SOFA Component Description Language (CDL). To model our bank, several mutually interconnected components have been introduced as depicted in Figure 27.

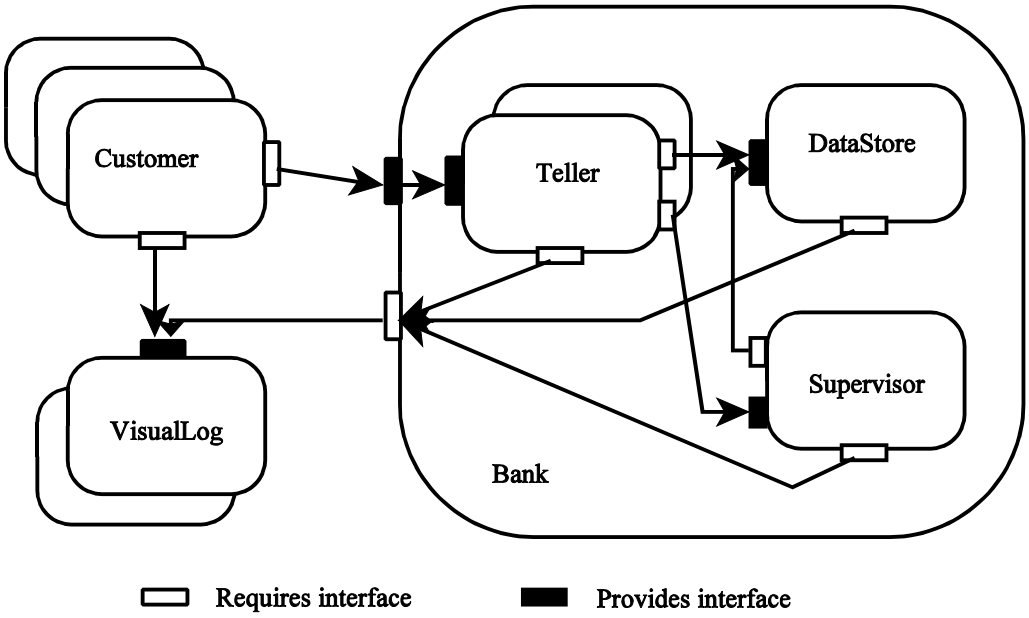

Figure 27 BankingDemo Architecture

The core of the application is the Bank component, which is the only compound component used in this application (other components are primitive - they are supposed to be directly implemented using underlaying implementation language). The Bank component internally contains NoT instances of the Teller subcomponents, the Supervisor subcomponent, and the DataStore subcomponent. The Bank component provides NoT Teller interfaces, each of them is tied (delegated) to an interface provided by the corresponding Teller subcomponent. The common requirement of all subcomponents - access to the Log interface - is tied (subsumed) to the corresponding require interface of the Bank component. All interactions among the Bank subcomponents are based on procedure calls; instances of the CSProcCall connector type are used to model these interactions. The following piece of code illustrates the core of the Bank's CDL description.

```
frame DataStore {
  provides:
    DataStoreInterface DataStore;
  requires:
    LogInterface Log;
};
architecture CUNI DataStore version "1.0" primitive;

frame Supervisor {
  provides:
    SupervisorInterface Supervisor;
  requires:
    DataStoreInterface DataStore;
    LogInterface Log;
```

```
};
architecture CUNI Supervisor version "1.0" primitive;
frame Teller {
  provides:
    TellerInterface Teller;
  requires:
    DataStoreInterface DataStore;
    SupervisorInterface Supervisor;
    LogInterface Log;
};
architecture CUNI Teller version "1.0" primitive;

frame Bank {
  provides:
    TellerInterface Teller[1..NoT];
  requires:
    LogInterface Log;
};

architecture CUNI Bank version "1.0" {
  inst DataStore aDataStore;
  inst Supervisor aSupervisor;
  inst Teller aTeller[1..NoT];
  bind aSupervisor.DataStore to aDataStore.DataStore using CSProcCall;
  bind aTeller[1..NoT].DataStore to aDataStore.DataStore using CSProcCall;
  bind aTeller[1..NoT].Supervisor to aSupervisor.Supervisor using
CSProcCall;
  delegate Teller[1..NoT] to aTeller[1..NoT].Teller using CSProcCall;
  subsume aDataStore.Log to Log using CSProcCall;
  subsume aSupervisor.Log to Log using CSProcCall;
  subsume aTeller[1..NoT].Log to Log using CSProcCall;
};
```

The Customer components model the behavior of bank customers by requesting randomly chosen bank tellers to perform transactions on accounts. Customers also send messages to all logging tools that are interested in receiving them. The Customer components are the only source of active threads in the application.

```
frame Customer {
  requires:
    TellerInterface BankTeller;
    LogInterface Log
};
architecture CUNI Customer version "1.0" primitive;
```

The VisualLog components serve as specialized logging tools monitoring the application's activity. Their architectures are primitive.

```
frame VisualLog {
  provides:
```

```
    LogInterface Log;
};
architecture CUNI VisualLog version "1.0" primitive;
```

The whole system consists of NoC instances of the Customer component, the Bank component, and NoL instances of the VisualLog component. Each Customer interacts with the Bank using procedure calls; all interactions with VisualLogs are based on event passing. The CDL description follows. For the complete Banking demo's CDL description see Appendix B.

```
system CUNI BankingDemo version "1.0" {
  inst Bank aBank;
  inst VisualLog aLog[1..NoL];
  inst Customer aCustomer[1..NoC];
  bind aCustomer[1..NoC].Teller to aBank.Teller[?] using CSProcCall;
  bind aBank.Log to aLog[1..NoL].Log using EventDelivery;
  bind aCustomer[1..NoC].Log to aLog[1..NoL].Log using EventDelivery;
};
```

7.1.2 Glue code generation and implementation of primitive components

Once the CDL specification of the application has been created, the next step is to generate the application's "glue code" using the SOFA CDL compiler. For a primitive component type, the generated "glue code" includes skeletons of the component implementation and the component builder as illustrated on the example of bank tellers.

```
// TellerImpl.java
package SOFA.demos.bankdemo;

public class TellerImpl implements TellerInterface {
  private transient DataStoreInterface DataStore;
  private transient SupervisorInterface Supervisor;
  private transient LogInterface Log;

  public TellerImpl (DataStoreInterface _DataStore, SupervisorInterface
_Supervisor, LogInterface _Log {
    DataStore = _DataStore;
    Supervisor = _Supervisor;
    Log = _Log;
  }
  public String createAccount (float init) {}
  public boolean deleteAccount (String number) {}
  public boolean deposit (String number, float amount) {}
  public boolean withdraw (String number, float amount) {}
  public float balance (String number) {}
}

// TellerBuilder_1_0.java
package SOFA.demos.bankdemo;
...
```

```
public class TellerBuilder_1_0 implements ComponentBuilder {
  private TellerImpl _impl;
  private RegisterInterface _reg;
  private DeploymentDescriptor _dd;

  public TellerBuilder_1_0 (RegisterInterface __reg,
DeploymentDescriptor __dd) {
    _reg = __reg;
    _dd = __dd;
  }
  public void onArrival (Storage stateStore) throws
ComponentLifecycleException {
    try {
      SupervisorInterface _Supervisor = (SupervisorInterface)
_reg.lookupRequirement ("Supervisor");
      DataStoreInterface _DataStore = (DataStoreInterface)
_reg.lookupRequirement ("DataStore");
      LogInterface _Log = (LogInterface) _reg.lookupRequirement
("DataStore");
      _impl = new TellerImpl (_DataStore, _Supervisor, _Log);
      _reg.registerServiceImpl("Teller", _impl);
      restore(stateStore);
    }
    catch(Exception e){
      throw new ComponentLifecycleException(e.toString());
    }
  }
  public void onLeaving(Storage stateStore) throws
ComponentLifecycleException {
    store(stateStore);
  }
  public void store(Storage stateStore) throws
ComponentLifecycleException {}
  public void restore(Storage stateStore) throws
ComponentLifecycleException {}
}
```

The generated glue code is used as a basis for creating a primitive component's implementation. The component's business logic must be added by supplying an implementation to all business methods provided by the component's interfaces. When necessary, a custom mechanism for handling the component's persistent state can be added by implementing the store and restore methods of the component builder.

Concerning composed components, since they are created by composition of their nested subcomponents, the CDL compiler completely generates their default implementation. In rare cases only, the generated builders need to be extended with a custom code. To illustrate a code used in generated builders to create and compose subcomponents, the onArrival method of BankBuilder can serve as an example.

```
// BankBuilder_1_0.java
package SOFA.demos.bankdemo;
...
public class BankBuilder_1_0 implements ComponentBuilderInterface {
  ...
  public void onArrival (Storage stateStore) throws
ComponentLifecycleException {
    try {
      ComponentFactory _factory = FactoryFinder.getFactory
(_dd.getSubComponentLocation("aDataStore"));
      aDataStore = _factory.instantiate
(_dd.getSubComponentDeploymentDescriptor("aDataStore"));
      _reg.registerSubcomponent("aDataStore", aDataStore);
      _factory = FactoryFinder.getFactory
(_dd.getSubComponentLocation("aSupervisor"));
      aSupervisor = factory.instantiate
(_dd.getSubComponentDeploymentDescriptor("aSupervisor"));
      _reg.registerSubcomponent("aSupervisor", aSupervisor);
      for (int _i = 0; _i < aTeller.length; _i++) {
        _factory = FactoryFinder.getFactory
(_dd.getSubComponentLocation("aTeller"+_i));
        aTeller[_i] = factory.instantiate
(_dd.getSubComponentDeploymentDescriptor("aTeller"+_i));
        _reg.registerSubcomponent("aTeller"+_i, aTeller[_i]);
      }
      Object _DataStore = aDataStore.lookupService("DataStore");
      aSupervisor.provideRequirement("DataStore", _DataStore);
      for (int _i = 0; _i < aTeller.length; _i++)
        aTeller[_i].provideRequirement("DataStore", _DataStore);
      Object _Supervisor = aSupervisor.lookupService("Supervisor");
      for (int _i = 0; _i < aTeller.length; _i++)
        aTeller[_i].provideRequirement("Supervisor", _Supervisor);
      for (int _i = 0; _i < aTeller.length; _i++)
        _reg.delegateService ("aTeller"+_i, aTeller[_i].lookupService
("Teller"));
      Object _Log = _reg.subsumeService("Log");
      aDataStore.provideRequirement("Log", _Log);
      aSupervisor.provideRequirement("Log", _Log);
      for (int _i = 0; _i < aTeller.length; _i++)
        aTeller[_i].provideRequirement("Log", aLog);
      aDataStore.createComponent((stateStore == null) ? null :
stateStore.getStorageForSC("aDataStore"));
      aSupervisor.createComponent((stateStore == null) ? null :
stateStore.getStorageForSC("aSupervisor"));
      for (int _i = 0; _i < aTeller.length; _i++)
        aTeller[_i].createComponent((stateStore == null) ? null :
stateStore.getStorageForSC("aTeller"+_i));
    }
    catch(Exception e){
      throw new ComponentLifecycleException (e.toString());
    }
  }
  ...
}
```

7.1.3 Application assembly

Once various implementations for all the application's frames are prepared (remember that multiple architectures can implement a single component frame in SOFA), the executable form of the application is created by selecting one particular implementation version (identified by its builder) for each frame contained in the application. As a result, the application's *assembly descriptor* is created. Since assembly descriptors serve as bases for deployment descriptor forms (and thus for deployment descriptors themselves), they use the XML notation required by deployment descriptors in SOFA/DCUP. An example assembly descriptor of the Banking demo application follows.

```
<sofa_system name="BankingDemo">
  <tm provider="CUNI" type="BankingDemo" version="1.0"/>
  <builder_class> SOFA.demos.bankdemo.BankingDemoBuilder_1_0
</builder_class>
  <property name="NoC" value="" />
  <property name="NoL" value="" />
  <sofa_component name="aBank">
    <tm provider="CUNI" type="Bank" version="1.0"/>
    <builder_class> SOFA.demos.bankdemo.BankBuilder_1_0 </builder_class>
    <property name="NoT" value="" />
    ...
    <sofa_component name="aDataStore">
      <tm provider="CUNI" type="DataStore" version="1.0"/>
      <builder_class> SOFA.demos.bankdemo.DataStoreBuilder_1_0
</builder_class>
      ...
    </sofa_component>
    <sofa_component name="aSupervisor">
      <tm provider="CUNI" type="Supervisor" version="1.0"/>
      <builder_class> SOFA.demos.bankdemo.SupervisorBuilder_1_0
</builder_class>
      ...
    </sofa_component>
    <sofa_component name="aTeller*">
      <tm provider="CUNI" type="Teller" version="1.0"/>
      <builder_class> SOFA.demos.bankdemo.TellerBuilder_1_0
</builder_class>
      ...
    </sofa_component>
  </sofa_component>
  <sofa_component name="aLog*">
    <tm provider="CUNI" type="VisualLog" version="1.0"/>
    <builder_class> SOFA.demos.bankdemo.VisualLogBuilder_1_0
</builder_class>
    ...
  </sofa_component>
  <sofa_component name="aCustomer*">
    <tm provider="CUNI" type="Customer" version="1.0"/>
```

```
    <builder_class> SOFA.demos.bankdemo.CustomerBuilder_1_0
</builder_class>
    ...
  </sofa_component>
</sofa_system>
```

7.1.4 Decomposition into deployment units

To finish the design of the application, possible distribution boundaries within the application must be determined - the application needs to be decomposed into deployment units. In SOFA/DCUP, this decomposition is done on a top-down basis following the hierarchical structure of the assembled application with distribution boundaries crossing the component ties only. As a result, the *deployment descriptor form* is created by extending the application's assembly descriptor with information on the application's deployment units.

An example of decomposition of the Banking demo application into deployment units is shown in Figure 28. In this case, the bank, its customers, and logging tools are planned to possibly run separately. Within the Bank component, only DataStore can run externally. All Tellers and the Supervisor are required to share the same deployment dock with the Bank component itself.

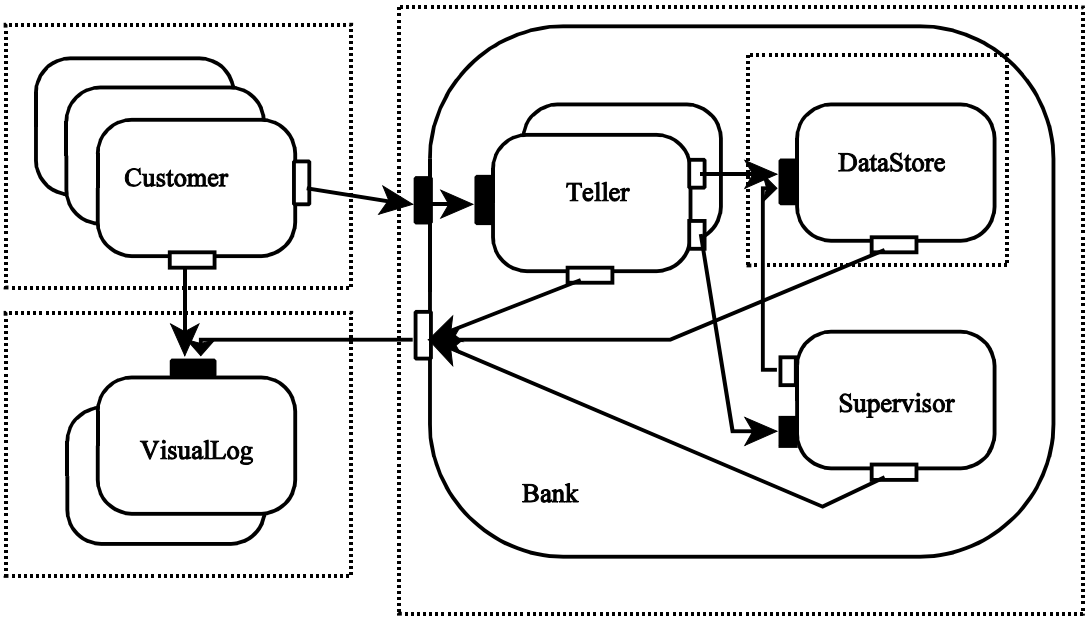

Figure 28 Decomposition into deployment units

The presented decomposition of the Banking demo application into deployment units is reflected by the following deployment descriptor form.

```
<sofa_system name="BankingDemo">
  ...
  <property name="NoC" value="3" />
  <property name="NoL" value="1" />
  ...
```

```
  <unit name="BankUnit">
    <location></location>
    <sofa_component name="aBank">
      ...
      <unit name="DataStoreUnit">
        <location></location>
        <sofa_component name="aDataStore">
          ...
        </sofa_component>
      </unit>
      <sofa_component name="aSupervisor">
        ...
      </sofa_component>
      <sofa_component name="aTeller*">
        ...
      </sofa_component>
      ...
    </sofa_component>
  </unit>
  <unit name="LogUnit">
    <location></location>
    <sofa_component name="aLog1">
      ...
    </sofa_component>
  </unit>
  <unit name="CustomersUnit">
    <location></location>
    <sofa_component name="aCustomer*">
      ...
    </sofa_component>
  </unit>
</sofa_system>
```

7.2 Example deployment

Before running a SOFA/DCUP application it is necessary to deploy its logical component topology to a physical computing environment. The deployment process starts with assigning concrete deployment docks to each of the application's deployment units. The second step of this process is to let the selected deployment docks generate an implementation code of their respective connectors (or to provide a manually created custom code). As a result, the application's deployment descriptor (an XML-based document containing all information necessary to run the application) is created by filling the application's deployment descriptor form.

7.2.1 Assigning deployment docks

To assign a concrete deployment dock to a particular deployment unit, there are two things which must be done. (1) A reference to the dock's ComponentFactory interface must be filled to the respective *location* tag in the application's deployment descriptor form. (2) The

implementation code of all components residing within the deployment unit must be made available to the selected deployment dock.

For simplicity, to illustrate one of the possible Banking demo's deployments, let the whole Banking demo application be deployed locally (to the local deployment dock) with the exception of the Log deployment unit deployed to the deployment dock running on host nenya.ms.mff.cuni.cz that is accessible via RMI. Assume that both deployment docks are SimpleJava deployment docks (section 6.2.3.1) with SOFAClassLoaders configured to load classes from the same template repository to which the Banking demo application's implementation code was previously stored. The filled deployment descriptor form reflecting this situation follows.

```
<sofa_system name="BankingDemo">
  ...
  <unit name="BankUnit">
    <location>sofa:local</location>
      ...
      <unit name="DataStoreUnit">
        <location>sofa:local</location>
        ...
      </unit>
      ...
  </unit>
  <unit name="LogUnit">
    <location>sofa:rmi//nenya.ms.mff.cuni.cz/ComponentFactory</location>
    ...
  </unit>
  <unit name="CustomersUnit">
    <location>sofa:local</location>
    ...
  </unit>
</sofa_system>
```

7.2.2 Connector code generation

To generate the implementation code for all connector instances within the application, a close cooperation between a deployer and the underlying deployment docks is necessary. To illustrate it, let us focus on the connector instance representing an interaction of the Bank and VisualLog components. This interaction is represented by the EventDelivery connector type instance where the Bank component is the event supplier entity and the VisualLog component is the consumer entity. Since both components are going to be deployed to different deployment docks, both docks have to be contacted to generate their respective parts of the resulting connector.

After agreeing on the technologies that will be used to implement the connector, the local deployment dock (to which the Bank component is going to be deployed) must be asked to generate the part of the connector corresponding to its supplier deployment unit. In particular, the code for the supplier role, distributor internal element, and stub internal element must be provided. After successful generation, the deployment dock returns a part of the deployment descriptor identifying classes generated to represent the requested connector's elements. If certain elements could not be generated, or a deployer wants to change and/or add some internal

element's implementation (e.g., interceptor), he or she must implement such an element and change the respective information in the deployment descriptor.

```
...
<sofa_component name="aBank">
  ...
  <sofa_connector interface="Log" type="EventDelivery">
    <suprole_class> SOFA.demos.bankdemo.impl.LogSupRole </suprole_class>
    <dist_class> SOFA.demos.bankdemo.impl.LogDistributor </dist_class>
    <st_class protocol="RMI"> SOFA.demos.bankdemo.impl.LogRMIStub
</st_class>
    <st_class protocol="CORBA"> SOFA.demos.bankdemo.impl.LogCORBAStub
</st_class>
  </sofa_connector>
  ...
</sofa_component>
...
```

After that, the deployment dock running on the host nenya.ms.mff.cuni.cz (to which the VisualLog component is going to be deployed) must be asked to generate the part of the connector corresponding to its consumer deployment unit. In particular, the code for the consumer role, and the skeleton internal element must be provided.

```
...
<sofa_component name="aLog1">
  ...
  <sofa_connector interface="Log" type="EventDelivery">
    <consrole_class> SOFA.demos.bankdemo.impl.LogSRole </consrole_class>
    <sk_class protocol="RMI"> SOFA.demos.bankdemo.impl.LogRMISkeleton
</sk_class>
    <sk_class protocol="CORBA"> SOFA.demos.bankdemo.impl.LogCORBASkeleton
</sk_class>
  </sofa_connector>
  ...
</sofa_component>
...
```

For the complete deployment descriptor of the Banking demo application, readers are refered to Appendix C.

7.3 Example run

To run the Banking demo application, the locally running deployment dock is asked to instantiate its main component (note that the whole application is represented as a single top-level component in SOFA) passing the application's deployment descriptor as parameter.

```
narya:$simple_java instantiate /home/balek/demo/BankingDemo.dd
```

Chapter 8

Conclusion

8.1 Summary

The thesis starts with a brief summary of the state-of-the-art in the area of component-based technologies emphasizing issues related to component interactions. A casual reader is provided with the necessary background to the topic together with a comprehensive list of references to the related work.

The deployment anomaly is then articulated as the necessity for a post-design modification of components caused by their deployment. This is a serious obstacle in using component-based software technologies in real-life applications. In a practical setting, the deployment of a component-based software system can be efficiently carried out by system staff members, experts in the underlying system environment (typically in the brands of middleware to be employed). To realize the necessary deployment modifications, these people would have to study the business logic details of the components subject to the deployment. This is inherently inefficient, if not even impossible, since some of the components may be of a third-party origin. A symmetrical inefficiency would be to ask the business logic designers to deal with the local networking/middleware details. For these reasons it is very desirable to separate the business and communication part of the component-based application. As presented in the thesis this issue can be addressed well by the connector concept.

As far as we know, none of the related ADL languages/systems, such as [46, 52, 22, 55, 1], targets the deployment issue directly nor combines it with connectors. A novel connector model is therefore proposed, which not only allows the expressing and representing of a variety of possible interactions among components in an application at all key stages of the application lifecycle, but in particular allows component distribution to be reflected. In this connector model, connectors of simple architectures (procedure call, data stream, event handling, etc.) are composed entirely of primitive elements, while connectors of compound architectures are hierarchically composed of instances of other connectors and components. Elementary enough to be implemented and reasoned about and even (semi-) automatically generated, primitive

elements can be easily composed into simple hierarchies to fulfill most of the basic connector tasks identified/analyzed in Section 4.

Finally, a programming framework for connectors is introduced. Starting with instruments that allow for a connector's architecture description, the necessary changes made to SOFA Component Description Language (CDL) were introduced. Also the SOFA deployment framework and its role in connector generation was presented followed by the description of the runtime support that gives users a notion of how to instantiate SOFA/DCUP connectors and use them to interconnect a collection of component instances. The main features of the proposed framework are illustrated by a case study given at the end of the thesis.

8.2 Meeting the goals of the thesis

In Chapter 3, the typical lifecycle of a component-based application is described and analyzed. Based upon this analysis, the necessity of post-design modifications of the application's components due to their deployment is articulated as a serious obstacle to the wide dissemination of component-based technologies. Thus, we meet the first chief goal of the thesis, which is to introduce and analyze the deployment anomaly problem.

The second chief goal of the thesis was to create a novel connector model satisfying the requirements stated in Chapter 4 of the thesis. This section summarizes, how the proposed SOFA/DCUP connector model addresses these requirements. For more detailed evaluation, we refer readers to Sections 5.6 and 6.4.

Requirement 4.1.1: Control and data transfer. The proposed connector model clearly specifies both - transfer of control and transfer of data. The control is transferred through a connector from its provides roles (connector's input points) along chains of internal elements to the connector's requires roles (connector's outbound points). As for data transfer, `in`, `out`, and `inout` modifiers are employed in the connector's interfaces to express the direction of data transfer.

Requirement 4.1.2: Interface adaptation and data conversion. To that purpose, special elements (interface adaptors) can be added to the connector's internal architecture to mediate the paths from the connector's provides to requires roles.

Requirement 4.1.3: Synchronization and access coordination. Similarly to an interface adaptation, the problem of synchronization and access coordination is solved inside of SOFA/DCUP connectors by adding special elements (synchronizers) to the certain places on the paths from the connector's provides to requires roles.

Requirement 4.1.4: Communication interception. Inspired by CORBA portable interceptors, this problem is solved by adding interceptors (hooks that allow users to plug-in an additional custom functionality) to certain places in a connector's architecture.

Requirement 4.1.5: Dynamic component linking. Since components are decoupled by connectors in SOFA/DCUP, connectors represent a natural place for applying changes in the components' connectivity without letting the components know about these changes. Simple interceptors can be added to connectors to redirect control and data flow within a connector and even to implement a simple load balancing mechanism.

Requirement 4.2.1: User defined ADL entities. Connectors are user-defined ADL entities at the same abstraction level as components in SOFA/DCUP. A user can define both - primitive connector types and compound connector types.

Requirement 4.2.2: Clear parametrization system. The SOFA/DCUP connectors are generic entities that can be parametrized by interface type parameters referring to a connector's roles and

its certain internal elements (e.g. adaptors) and/or property parameters referring to the overall connector's architecture and its certain internal elements (e.g. subs, skeletons, synchronizers).

Requirement 4.2.3: Lightweight ADL notation. The SOFA CDL uses a simple notation to express component interactions that makes the information concerning which component interfaces are bound together immediately visible. Anonymous connector declarations allow the correspondence between component interfaces and connector roles to be determined implicitly.

Requirement 4.2.4: Explicit runtime entities. The SOFA/DCUP connectors are explicit runtime entities concentrating all interaction codes. For each connector used within an application, there is a clear relation between its ADL description and runtime representation. This suppresses the deployment anomaly problem since all modifications due to deployment changes are localized to connectors and the application's business logic concentrated to components is not affected.

Requirement 4.2.5: Mapping from ADL to runtime. For each connector type in SOFA/DCUP, rules for its mapping from ADL to runtime exist. The mapping of a connector type with primitive architecture is given by mappings of its primitive elements, while the mapping of a compound connector type can be inferred from the mappings of its nested connector types. The implementation code for a particular connector is created by applying the corresponding mapping rules to its ADL description. This can be done either manually (a typical case for newly created and rarely used connector types) or automatically (in the case of the most common and repeatedly used connector types).

Requirement 4.2.6: Various middleware support. To utilize an underlying execution environment support as much as possible, several mappings for a particular connector type can exist, each of them optimized for a different underlying platform. Also the proposed deployment framework was adjusted to that purpose (deployment docks are equipped with generators able to create the connectors' code which best fits the corresponding execution environment).

Requirement 4.2.7: Clear specification of runtime behavior. The proposed programming framework describes a mechanism of instantiating connectors within an application in a step-by-step manner from creation of all of the connector's deployment units, binding them to the respective connector interfaces, to the process of interconnecting these units. As for connectors' destruction, although it is not explicitly mentioned in the thesis, a connector instance is destroyed at the time of the destruction of components tied to it.

Requirement 4.2.8: Clear relation to deployment framework. When supplying implementation code of connectors, an application's deployer cooperates closely with the deployment framework. First, deployment docks provide information about technologies that are available for an automatic generation of certain connector types. Then deployment docks either generate an implementation code for the selected connector types or allow deployers to supply the implementation code manually. As for deployment of connectors themselves, deployment of connectors is determined implicitly from the deployment of components tied to them.

8.3 Contributions

In summary, there are three main contributions of this thesis: (1) The thesis presents a new connector model that allows for specifying an interaction of an arbitrary complexity with a clear relation to its implementation thus overcoming the main drawbacks of UniCon and Wright.

(2) The thesis aims at clear articulation of the differences between components and connectors. To summarize, in the presented SOFA component and connector models, the key difference between a component and connector is in (a) distribution (a primitive connector can be

distributed, while a primitive component cannot) and (b) in the lifecycle (parts of the connector can be generated only after all deployment docks are determined).

(3) Providing a relation between an architectural specification and the application's deployment, the proposed connector model addresses the problem of deployment anomaly. To address the problem, a connector helps in (a) separation of concerns (by separating the business and communication part of a component-based application), and in (b) reusability - if the primitive elements are designed properly, they can be reused in many of the typical component communication patterns. The important trick supporting the reusability is that the primitive elements are very generic (work almost for "any interface"), and modification of the communication pattern for the actual interfaces can be done in an automatized way, i.e., it can be generated.

8.4 Future work

Our future intentions can be divided into three groups according to a time horizon of their completion. The first of the *short term* issues can be identified as finding techniques for an automatic generation of certain internal elements (such as interface adaptors and/or synchronizers). Having finished work on an automatic generation of stubs and skeletons for a remote communication, we turned our attention to interface adaptors. Inspired by the work of Yellin and Storm [67], we believe that dependent on a level of "incompatibility" of respective interfaces a necessary adaptor can be in certain cases automatically or at least semiautomatically generated. Similarly, we believe that a necessary synchronizer can be automatically generated in certain cases from behavioral protocols of respective component interfaces.

The task of automatic generation of synchronizers is closely related to the second of the short term issues that can be articulated as using behavioral protocols for specifying control flows within a connector. The aim is to enhance a behavior specification of connector's internal elements (currently written in plain English) with a more formal description method that will allow for an automatic proving the correctness of a system architecture (all constraints on using the application components expressed by the behavioral protocols are met) and for automatic generation of possible synchronizers.

In the *mid term* horizon, the problem of passing various execution contexts (e.g. security and transactional contexts) must be solved. The low level support for execution context passing has been already implemented. To allow for safe dynamic component updates, the DCUP architecture requires a distributed thread identification to be passed along each method invocation. To this purpose, a distributed thread identification is passed via connectors as an extra method parameter added. Using a similar mechanism, other execution contexts can be passed via connectors. The issue that has to be solved is a higher level support for execution contexts passing (e.g. mapping of execution contexts among various middleware platforms using different security and transactional models).

In the *long term* horizon, the main issue is alignment of SOFA with the mainstream initiative of OMG - Model Driven Architecture (MDA) concept [44]. The main idea behind MDA is very similar to the idea that led us to employing connectors in SOFA - when designing an application, the application business logic should be designed independently of the concrete middleware that is used later on for implementation of the application's components. Moreover, the middleware specific part of the application's code should be (at least partially) generated.

The MDA concept introduces the following approach to an application creation. As the first step, the Platform Independent Model (PIM) of the application's business logic is created using

Unified Modeling Language (UML) and/or Meta Object Facility (MOF) [43]. Then, the target application's middleware platform is selected and the application's PIM is translated into the Platform Specific Model (PSM). Finally, the application skeleton is generated by the technology specific generator from the application's PSM.

As the whole MDA framework is currently in the process of its creation, there has been no implementation of this concept so far. However, looking closer at both the SOFA and MDA technologies, we can find that SOFA already possesses some of the MDA features and therefore it would be possible to align SOFA with the MDA framework in the near future. For instance, one can easily realize that an application's architecture description written in SOFA CDL represents a platform independent model of the application. Thus, since MDA assumes UML and MOF to be used as widely accepted standards for modeling purposes, creation of a MOF-based meta-model for SOFA applications may be the first step towards the alignment. There are two advantages of this approach: (1) a variety of existing UML based tools can be used to design an application's architecture in SOFA, (2) and some of the exiting standard MOF based meta-data repositories can be used to store and access an application's meta-data in SOFA replacing thus a proprietary SOFA TIR On the other hand, the MDA framework could take advantage of clear separation of an application's business logic (centralized to components) from an underlying middleware platform (represented by connectors) implemented in SOFA.

References

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

[2] Allen, R. J., Garlan, D.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology, 1997.

[3] Balek, D., Plasil, F.: Software Connectors and their Role in Component Deployment, Proceedings of DAIS'2001, Krakow, Poland, 2001.

[4] Balter, R., Bellissard, L., Boyer, F., Riveill, M., Vion-Dury, J-Y.: Architecturing and Configuring Distributed Applications with Olan. In Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District, 15-18 September 1998.

[5] Bellissard, L., Ben Atallah, S., Boyer, F., Riveill, M.: Distributed Application Configuration. In Proceedings of 16th International Conference on Distributed Computing Systems, pp. 579-585, IEEE Computer Society, Hong-Kong, May 1996.

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

[7] Blair, G., Blair, L., Issarny, V., Tuma, P., Zarras, A.: The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms. In Proceedings of Middleware 2000 - IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing. April 2000, Hudson River Valley (NY), USA. Springer Verlag, LNCS.

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

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

[10] Garlan, D., Allen, R., Ockerbloom, J.: Exploiting Style in Architectural Design Environments. In Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering, December 1994.

[11] Garlan, D., Monroe, R. T., Wile, D.: Acme: An Architecture Description Interchange Language. In Proceedings of CASCON '97, November 1997.

[12] Garlan, D., Monroe, R. T., Wile, D.: Acme: Architectural Description of Component-Based Systems. Foundations of Component-Based Systems, Gary T. Leavens and Murali Sitaraman (eds), Cambridge University Press, 2000 pp. 47-68.

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

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

[15] Hnetynka, P. : Managing Type Information in an Evolving Environment. Master thesis, Charles University, Prague, 2000.

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

[17] 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.

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

[19] Kalibera, T.: SOFA Support in C++ Environments. Master thesis, Charles University, Prague, 2001.

[20] Kenney, J. J.: Executable Formal Models of Distributed Transaction Systems based on Event Processing. PhD thesis, Stanford University, December 1995.

[21] Leavens, G. T., Sitaraman, M. (eds.): Foundation of Component-Based Systems, Cambridge University Press, 2000.

[22] 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.

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

[24] Magee, J., Kramer, J.: Dynamic Structure in Software Architectures. In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 3-14, 1996.

[25] Marvie, R., Merle, P., Geib, J.-M.: Towards a Dynamic CORBA Component Platform. 2nd International Symposium on Distributed Object Applications (DOA 2000), Antwerp, Belgium, 2000.

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

[27] Medvidovic, N.: Architecture-Based Specification-Time Software Evolution. PhD Dissertation, Dept. Information Computer Science, UCI, November 1998.

[28] Medvidovic, N., Oreizy, P., Taylor, R. N.: Reuse of Off-the-Shelf Components in C2-Style Architectures. In Proceedings of the 1997 International Conference on Software Engineering (ICSE'97), Boston, MA, May 17-23, 1997.

[29] 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.

[30] 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.

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

[32] Meyer, B: Eiffel: the language. Prentice Hall, 1992.

[33] Monroe, R. T., Garlan, D.: Style Based Reuse for Software Architectures. In Proceedings of the 1996 International Conference on Software Reuse, April 1996.

[34] Moriconi, M., Qian, X., Riemenschneider, R. A.: Correct Architecture Refinement. IEEE Transactions on SW Engineering, 21(4), 1995.

[35] Moriconi, M., Riemenschneider, R. A.: Introduction to SADL 1.0 : A Language for Specifying Software Architecture Hierarchies. Technical Report SRI-CSL-97-01, SRI System Design Laboratory, 1997

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

[37] OMG formal/01-12-35: The Common Object Request Broker: Architecture and Specification, v 2.6, December 2001.

[38] OMG orbos/99-07-01: CORBA Component Model - Volume 1. 1999.

[39] OMG orbos/99-07-02: CORBA Component Model - Volume 2. 1999.

[40] OMG orbos/99-07-03: CORBA Component Model - Volume 3. 1999.

[41] OMG formal/97-12-16: CORBA Services - Relationship Service. 1997.

[42] OMG formal/01-09-67: Unified Modeling Language, v1.4, 2001.

[43] OMG formal/00-04-03: Meta Object Facility (MOF) v 1.3, 2000.

[44] OMG ormsc/01-07-01: Model Driven Architecture (MDA). 2001.

[45] Oreizy, P., Gorlick, M. M., Taylor, R. N, Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., Wolf, A. L.: An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems, Vol. 14, No. 3, May/June 1999.

[46] 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.

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

[48] 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.

[49] Plasil, F., Besta, M., Visnovsky, S.: Bounding Component Behavior via Protocols. Accepted for presentation at TOOLS USA ‘99, Santa Barbara, USA, 1999.

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

[51] 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.

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

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

[54] 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.

[55] 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.

[56] Shaw, M., DeLine, R., Zelesnik, G.: Abstractions and Implementations for Architectural Connections. Proceedings of the 3rd International Conference on Configurable Distributed Systems, May 1996.

[57] Stroustrup, B.: The C++ Programming Language, Second Edition. Addison-Wesley, 1995.

[58] Sun Microsystems, Inc.: JavaBeans Specification 1.0.1, July 1997.

[59] Sun Microsystems, Inc.: Enterprise Java Beans Specification 1.0, March 1998.

[60] Sun Microsystems, Inc.: Enterprise Java Beans Specification 1.1, Public Release, August 10, 1999.

[61] Sun Microsystems, Inc.: Enterprise JavaBeans Specification 2.0, Final Release, August 14, 2001.

[62] Sun Microsystems, Inc.: Java Message Service API Specification 1.02, November 9, 1999.

[63] Sun Microsystems, Inc.: Java Naming and Directory Interface Application Programming Interface (JNDI API), JNDI 1.2/Java 2 Platform, Standard Edition, v 1.3, July 14, 1999.

[64] Sun Microsystems, Inc.: Java Remote Method Invocation Specification - Java 2 SDK, v 1.3.0, December 1999.

[65] Sun Microsystems, Inc.: Java Transaction Service (JTS) 1.0, December 1, 1999.

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

[67] 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.

[68] Zarras, A.: Systematic Customization of Middleware. PhD thesis, University of Rennes, France, March 2000.

[69] Zaremski, A. M., Wing, J. M.: Specification matching of software components. In Proceedings of the ACM SIGSOFT'95 Symposium on Foundations of Software Engineering, 1995.

Appendix A: Programming Framework Interfaces

```
module SOFA {
  module Deployment {

    struct TechnologyProperty {
      string name;
      string value;
    };
    typedef sequence <TechnologyProperty> TechnologyProperties;

    struct TechnologyDescriptor {
      string id;
      TechnologyProperties props;
    };
    typedef sequence <TechnologyDescriptor> TechnologyDescriptors;

    exception DeploymentException {
      string reason;
    };

    interface ComponentFactory {
      TechnologyDescriptors describeUnderlyingEnvironment();
      void install (in SOFA::TIR::ComponentInstanceDef comp, in string url)
        raises (DeploymentException);
      ComponentManager instantiate(in DeploymentDescriptor dd)
        raises (DeploymentException);
    };

    struct ElementTechnologyDesciptor {
      string elementName;
      TechnologyDescriptors technologies;
    };
    typedef sequence <ElementTechnologyDescriptor>
      ConnectorTechnologyDescriptor;

    struct ElementTechnologySelection {
```

```
    string elementName;
    string selectedTechnologyID;
  };
  typedef sequence <ElementTechnologySelection>
    ConnectorTechnologySelection;

  exception GenerationException {};

  interface CCGNegotiation {
    ConnectorTechnologyDescriptor canGenerate(
      in SOFA::TIR::ConnectorInstanceDef conn, in string unitName);
    SOFA::Connector::DeploymentDescriptor generate(
      in SOFA::TIR::ConnectorInstanceDef conn, in string unitName,
      in ConnectorTechnologySelection selectedTechnology)
      raises GenerationException;
    void useExternalCodeLocation(in SOFA::TIR::ConnectorInstanceDef conn,
      in string unitName, in ConnectorTechnologyDescriptor technology,
      in string url);
  };

  frame DeploymentDock {
    provides:
      ComponentFactory componentFactory;
      CCGNegotiation connectorGenerator;
    requires:
      SOFA::TIR::Repository typeInformationRepository;
      SOFA::TemplateRepository templateRepository;
  };

  module ECG {

    struct RoleInfo {
      string roleName;
      SOFA::TIR::InterfaceDef interfaceType;
    };
    sequence <RoleInfo> RoleInfos;

    interface PlugIn {
      void setRoleInfos(in RoleInfos infos);
      void setTechnologyPropertyValue(in string name, in string value);
      void generate(in SOFA::TIR::ConnectorInstanceDef conn,
        in string unitName)raises(SOFA::Deployment::GenerationException);
    };
  };
};

module Connector {

  valuetype TaggedProfile {
    string tag;
  };
```

```
    struct Reference {
      string connectorType;
      sequence <TaggedProfile> profiles;
    };

    module CSProcCall {

      valuetype LocalProfile : SOFA::Connector::TaggedProfile {
        Object target;
      };

      valuetype CORBAProfile : SOFA::Connector::TaggedProfile {
        string IOR;
      };

      valuetype RMIProfile : SOFA::Connector::TaggedProfile {
        string URL;
      };
    };

    module DataStream {

      valuetype LocalProfile : SOFA::Connector::TaggedProfile {
        Object buffer;
      };

      valuetype PipeProfile : SOFA::Connector::TaggedProfile {
        FileDescriptor fd;
      };

      valuetype TCPIPProfile : SOFA::Connector::TaggedProfile {
        string hostName;
        int portNumber;
      };
    };
  };
};
```

Appendix B: Banking Demo CDL Description

```
module demos {
   module bankdemo {

     interface AccountInterface {
       void setBalance(in float balance);
       float getBalance();
       string getHistory();
     };

     interface LogInterface {
       void logAction(in string action);
     };

     interface DataStoreInterface {
       AccountInterface load(in string number);
       void save(in string number);
       AccountInterface create(in string number);
       void delete(in string number);
     };

     interface SupervisorInterface {
       boolean canWithdraw(in string number);
       boolean canDelete(in string number);
     };

     interface TellerInterface {
       string createAccount(in float init);
       void deleteAccount(in string number);
       void deposit(in string number, in float amount);
       void withdraw(in string number, in float amount);
       float balance(in string number);
     };

     frame Bank {
       provides:
```

```
          TellerInterface Teller[1..NoT];
        requires:
          LogInterface Log;
      };

      frame VisualLog {
        provides:
          LogInterface Log;
      };

      frame Customer {
        requires:
          TellerInterface BankTeller;
          LogInterface Log;
      };

      frame DataStore {
        provides:
          DataStoreInterface DataStore;
        requires:
          LogInterface Log;
      };

      frame Supervisor {
        provides:
          SupervisorInterface Supervisor;
        requires:
          DataStoreInterface DataStore;
          LogInterface Log;
      };

      frame Teller {
        provides:
          TellerInterface Teller;
        requires:
          DataStoreInterface DataStore;
          SupervisorInterface Supervisor;
          LogInterface Log;
      };
    };
  };
};

architecture CUNI ::SOFA::demos::bankdemo::VisualLog version "1.0"
primitive;
architecture CUNI ::SOFA::demos::bankdemo::Customer version "1.0"
primitive;
architecture CUNI ::SOFA::demos::bankdemo::DataStore version "1.0"
primitive;
architecture CUNI ::SOFA::demos::bankdemo::Supervisor version "1.0"
primitive;
architecture CUNI ::SOFA::demos::bankdemo::Teller version "1.0" primitive;
```

```
architecture CUNI ::SOFA::demos::bankdemo::Bank version "1.0" {
  inst ::SOFA::demos::bankdemo::DataStore aDataStore;
  inst ::SOFA::demos::bankdemo::Supervisor aSupervisor;
  inst ::SOFA::demos::bankdemo::Teller aTeller[1..NoT];

  bind aSupervisor.DataStore to aDataStore.DataStore using CSProcCall;
  bind aTeller[1..NoT].DataStore to aDataStore.DataStore using CSProcCall;
  bind aTeller[1..NoT].Supervisor to aSupervisor.Supervisor using
CSProcCall;

  delegate Teller[1..NoT] to aTeller[1..NoT].Teller using CSProcCall;

  subsume aDataStore.Log to Log using CSProcCall;
  subsume aSupervisor.Log to Log using CSProcCall;
  subsume aTeller[1..NoT].Log to Log using CSProcCall;
};

system CUNI ::SOFA::demos::bankdemo::BankingDemo version "1.0" {
  inst ::SOFA::demos::bankdemo::Bank aBank;
  inst ::SOFA::demos::bankdemo::VisualLog aLog;
  inst ::SOFA::demos::bankdemo::Customer aCustomer[1..NoC];

  bind aCustomer[1..NoC].Teller to aBank.Teller[?] using CSProcCall;
  bind aBank.Log to aLog.Log using EventDelivery;
  bind aCustomer[1..NoC].Log to aLog.Log using EventDelivery;
};
```

Appendix C: Banking Demo Deployment Descriptor

```
<sofa_system name="BankingDemo">
  <tm provider="CUNI" type="BankingDemo" version="1.0"/>
  <builder_class> SOFA.demos.bankdemo.BankingDemoBuilder_1_0
</builder_class>
  <property name="NoC" value="3" />
  <property name="NoL" value="1" />
  <sofa_connector interface="DCUPComponentManager" type="CSProcCall">
    <st_class protocol="RMI"> SOFA.DCUP.impl.DCUPComponentManagerRMIStub
</st_class>
    <crole_class> SOFA.DCUP.impl.DCUPComponentManagerCRole </crole_class>
  </sofa_connector>
  <unit name="BankUnit">
    <location>sofa:local</location>
    <sofa_component name="aBank">
      <tm provider="CUNI" type="Bank" version="1.0"/>
      <builder_class> SOFA.demos.bankdemo.BankBuilder_1_0 </builder_class>
      <property name ="NoT" value="2" />
      <sofa_connector interface="DCUPComponentManager" type="CSProcCall">
        <srole_class> SOFA.DCUP.impl.DCUPComponentManagerSRole
</srole_class>
        <sk_class protocol="RMI">
SOFA.DCUP.impl.DCUPComponentManagerRMISkeleton </sk_class>
      </sofa_connector>
      <sofa_connector interface="Teller*" type="CSProcCall">
        <srole_class> SOFA.demos.bankdemo.impl.TellerSRole </srole_class>
      </sofa_connector>
      <sofa_connector interface="_delegate_Teller*" type="CSProcCall">
        <crole_class> SOFA.demos.bankdemo.impl.TellerCRole </crole_class>
      </sofa_connector>
      <sofa_connector interface="Log" type="EventDelivery">
        <suprole_class> SOFA.demos.bankdemo.impl.LogSupRole
</suprole_class>
        <dist_class> SOFA.demos.bankdemo.impl.LogDistributor </dist_class>
        <st_class protocol="RMI"> SOFA.demos.bankdemo.impl.LogRMIStub
</st_class>
```

```
        <st_class protocol="CORBA"> SOFA.demos.bankdemo.impl.LogCORBAStub
</st_class>
      </sofa_connector>
      <sofa_connector interface="_subsume_Log" type="CSProcCall">
        <srole_class> SOFA.demos.bankdemo.impl.LogSRole </srole_class>
      </sofa_connector>
      <unit name="DataStoreUnit">
        <location>sofa:local</location>
        <sofa_component name="aDataStore">
          <tm provider="CUNI" type="DataStore" version="1.0"/>
          <builder_class> SOFA.demos.bankdemo.DataStoreBuilder_1_0
</builder_class>
          <sofa_connector interface="DCUPComponentManager"
type="CSProcCall">
            <srole_class> SOFA.DCUP.impl.DCUPComponentManagerSRole
</srole_class>
          </sofa_connector>
          <sofa_connector interface="DataStore" type="CSProcCall">
            <srole_class> SOFA.demos.bankdemo.impl.DataStoreSRole
</srole_class>
          </sofa_connector>
          <sofa_connector interface="Account*" type="CSProcCall">
            <srole_class> SOFA.demos.bankdemo.impl.AccountSRole
</srole_class>
          </sofa_connector>
          <sofa_connector interface="Log" type="CSProcCall">
            <crole_class> SOFA.demos.bankdemo.impl.LogCRole </crole_class>
          </sofa_connector>
        </sofa_component>
      </unit>
      <sofa_component name="aSupervisor">
        <tm provider="CUNI" type="Supervisor" version="1.0"/>
        <builder_class> SOFA.demos.bankdemo.SupervisorBuilder_1_0
</builder_class>
        <sofa_connector interface="DCUPComponentManager" type="CSProcCall">
          <srole_class> SOFA.DCUP.impl.DCUPComponentManagerSRole
</srole_class>
        </sofa_connector>
        <sofa_connector interface="Supervisor" type="CSProcCall">
          <srole_class> SOFA.demos.bankdemo.impl.SupervisorSRole
</srole_class>
        </sofa_connector>
        <sofa_connector interface="Log" type="CSProcCall">
          <crole_class> SOFA.demos.bankdemo.impl,LogCRole </crole_class>
        </sofa_connector>
        <sofa_connector interface="DataStore" type="CSProcCall">
          <crole_class> SOFA.demos.bankdemo.impl.DataStoreCRole
</crole_class>
        </sofa_connector>
        <sofa_connector interface="Account*" type="CSProcCall">
          <crole_class> SOFA.demos.bankdemo.impl.AccountCRole
</crole_class>
```

```
        </sofa_connector>
      </sofa_component>
      <sofa_component name="aTeller*">
       <tm provider="CUNI" type="Teller" version="1.0"/>
        <builder_class> SOFA.demos.bankdemo.TellerBuilder_1_0
</builder_class>
        <sofa_connector interface="DCUPComponentManager" type="CSProcCall">
          <srole_class> SOFA.DCUP.impl.DCUPComponentManagerSRole
</srole_class>
        </sofa_connector>
        <sofa_connector interface="Teller" type="CSProcCall">
          <srole_class> SOFA.demos.bankdemo.impl.TellerSRole </srole_class>
        </sofa_connector>
        <sofa_connector interface="Log" type="CSProcCall">
          <crole_class> SOFA.demos.bankdemo.impl.LogCRole </crole_class>
        </sofa_connector>
        <sofa_connector interface="DataStore" type="CSProcCall">
          <crole_class> SOFA.demos.bankdemo.impl.DataStoreCRole
</crole_class>
        </sofa_connector>
        <sofa_connector interface="Account*" type="CSProcCall">
          <crole_class> SOFA.demos.bankdemo.impl.AccountCRole
</crole_class>
        </sofa_connector>
        <sofa_connector interface="Supervisor" type="CSProcCall">
          <crole_class> SOFA.demos.bankdemo.impl.SupervisorCRole
</crole_class>
        </sofa_connector>
      </sofa_component>
    </sofa_component>
  </unit>
  <unit name="LogUnit">
    <location>sofa:rmi//nenya.ms.mff.cuni.cz/ComponentFactory</location>
    <sofa_component name="aLog1">
      <tm provider="CUNI" name="VisualLog" version="1.0"/>
      <builder_class> SOFA.demos.bankdemo.VisualLogBuilder_1_0
</builder_class>
      <sofa_connector interface="DCUPComponentManager" type="CSProcCall">
        <srole_class> SOFA.DCUP.impl.DCUPComponentManagerSRole
</srole_class>
        <sk_class protocol="RMI">
SOFA.DCUP.impl.DCUPComponentManagerRMISkeleton </sk_class>
      </sofa_connector>
      <sofa_connector interface="Log" type="EventDelivery">
        <consrole_class> SOFA.demos.bankdemo.impl.LogSRole
</consrole_class>
        <sk_class protocol="RMI"> SOFA.demos.bankdemo.impl.LogRMISkeleton
</sk_class>
        <sk_class protocol="CORBA">
SOFA.demos.bankdemo.impl.LogCORBASkeleton </sk_class>
      </sofa_connector>
    </sofa_component>
```

```
  </unit>
  <unit name="CustomersUnit">
    <location>sofa:local</location>
    <sofa_component name="aCustomer*">
      <tm provider="CUNI" name="Customer" version="1.0"/>
      <builder_class> SOFA.demos.bankdemo.CustomerBuilder_1_0
</builder_class>
      <sofa_connector interface="DCUPComponentManager" type="CSProcCall">
        <srole_class> SOFA.DCUP.impl.DCUPComponentManagerSRole
</srole_class>
        <sk_class protocol="RMI">
SOFA.DCUP.impl.DCUPComponentManagerRMISkeleton </sk_class>
      </sofa_connector>
      <sofa_connector interface="Log" type="EventDelivery">
        <suprole_class> SOFA.demos.bankdemo.impl.LogSupRole
</suprole_class>
        <dist_class> SOFA.demos.bankdemo.impl.LogDistributor </dist_class>
        <st_class protocol="RMI"> SOFA.demos.bankdemo.impl.LogRMIStub
</st_class>
        <st_class protocol="CORBA"> SOFA.demos.bankdemo.impl.LogCORBAStub
</st_class>
      </sofa_connector>
      <sofa_connector interface="Teller*" type="CSProcCall">
        <crole_class> SOFA.demos.bankdemo.impl.TellerCRole </crole_class>
      </sofa_connector>
    </sofa_component>
  </unit>
</sofa_system>
```