SOFTWARE CONNECTORS AND THEIR ROLE IN COMPONENT DEPLOYMENT

Dušan Bálek[1], František Plášil[1,2]
[1] *Charles University, Faculty of Mathematics and Physics, Department of Software Engineering, Malostranské nám-stí 25, 118 00 Prague 1, Czech Republic, http://nenya.ms.mff.cuni.cz*

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

Abstract To support rapid software evolution, it is desirable to construct software systems from reusable components. In this approach, the architecture of a system is described as a collection of components along with the interactions among these components. Whereas the main system functional blocks are components, the properties of the system also strongly depend on the character of the component interactions. This fact gave birth to the "connector" concept which is an abstraction capturing the nature of these interactions. The problem tackled in this paper is that even though the notion of connectors originates in the earliest papers on software architectures [20, 15], connectors are currently far from being a typical first class entity in the contemporary component-based systems.

By articulating the "deployment anomaly", the paper identifies the role connectors should play when the distribution and deployment of a component-based application is considered. Further, we introduce a connector model reflected at all the key stages of an application's development: ADL specification, deployment, and implementation

Keywords: Software component, deployment, connector

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

1. INTRODUCTION

A few years ago, the trend to construct software systems as a collection of cooperating reusable components emerged and has become widely accepted since. Influenced by the academic research projects focused on components [8, 19, 21, 7, 1, 14, 5], several industrial systems on the market [22, 23, 12, 13, 24] advertise support of component technology. As for *components*, there is a broad agreement on grasping them as reusable black/grey-box entities with well-defined interfaces and specified behavior. Usually, a component can have multiple interfaces; some of them to provide services to the component's clients, other to require services from the surrounding environment. Components can be nested to form hierarchies; a higher-level component can be composed of several mutually interconnected, cooperating subcomponents. Serving as tools for specifying component interfaces and architecture, a number *architecture description languages (ADLs)* [8, 21, 1, 14, 17] have been designed. To describe component interactions, an ADL may encompass the connector concept: Typically, a *connector* is a first class architectural element that reflects the specific features of interactions among components in a system [21, 1, 14, 10, 3]. Even though the notion of a connector originates in the earliest papers on software architectures [20, 15], no widespread consensus on how to incorporate it into the existing application development systems and languages has been reached until present.

1.1 Connectors: overview and related work

Studying the related work [8, 21, 1, 3, 14], the following basic approaches to specifying component interactions can be identified: (1) using *implicit connections*, (2) via *connectors*, which can be either *built-in* or *user-defined.*

The Darwin language [8] is a typical representative of ADLs that use implicit connections. The connections among components are specified via direct bindings of their *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 of the underlying Regis environment).

In addition to making system maintenance easier, letting components communicate via connectors has also other significant benefits: increased

reusability (the same component can be used in a variety of environments, each of them providing specific communication primitives) direct support for distribution, location transparency and mobility of components in a system, support for dynamic changes in the system's connectivity, etc.

The UniCon language [21] is a representative of ADLs with (only) built-in connectors. A developer is provided with a selection of several predefined built-in connector types that correspond to the common communication primitives supported by the underlying language or operating system (such as RPC, pipe, etc.). However, the most significant drawback of a UniCon-like ADL is that there is no way to capture any interaction among components that does not correspond to a predefined connector type.

User-defined connectors, the most flexible approach to specifying component interactions, are employed, e.g., in the Wright language [1]. The interactions among components are fully specified by the user (system developer). Complex interactions can be expressed by nested connector types. However, the main drawback of the Wright language is the absence of any guidelines as to how to realize connectors in an implementation. (In Wright, connectors exist at the specification level only, which results in the problem of how to correctly reflect the specification of a connector in its implementation.)

Based on a thorough study of existing ADLs, Medvidovic et al. [10] presented a classification framework and taxonomy of software connectors. This taxonomy is an important attempt to improve the current level of understanding of what software connectors and their building blocks are. Not addressing all the issues of designing connector types, it is focused mainly on classification (thus better comprehension) of connector types. In addition, the selection of basic connector types in [10] may be questionable, as not all of them seem to be at the same abstraction level (e.g., adaptor, arbitrator and distributor vs. procedure call, event, stream).

1.2 Challenges and the goals of the paper

Component-based systems and 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 the real life has proved many times that to combine the business components provided by third parties into a running application can be very demanding. The main problem of the current ADLs is that they either do not capture component interactions at all, or they focus on

application design stages only. However, the component interactions have to be reflected throughout the whole application lifecycle, otherwise they may become a serious obstacle in component reusability. In particular, the deployment phase has turned out to be critical in this respect.

The first goal of the paper is to bring an additional argument for considering connectors as first class ADL entities by analyzing their role in component deployment (deployment anomaly is articulated). The second goal of the paper is to propose a connector model that allows to describe a variety of (possibly complex) component interactions, helps system developers target the deployment anomaly, and at the same time allows to generate the corresponding interaction code (since none of the existing connector models/ADL systems provides a sufficient support for that).

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

2. COMPONENTS AND THEIR LIFECYCLE

2.1 A component model

For the purpose of this paper we adopt the following component model (event though based on [16,17] in some details, it very much follows the basic spirit of most ADL languages): An application is viewed as a hierarchy of software components. A *software component* is an instance of a *component template* (*template* for short). A template is defined by a pair <component frame, component architecture>. In principle, the *component frame* determines a component type as the set of interfaces *provided* and *required* by every instance of T (reflecting a black-box view on T's instances). Similarly, the *component architecture* reflects a gray-box view of each of T's instances by describing its internal structure in the terms of its direct subcomponents and their interactions (interface ties, "wiring"). A component architecture can be

specified as *primitive* which means that there are no subcomponents and the component frame is directly implemented in the underlying implementation language, for example as a set of Java classes, a shared library, or even 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 *composed component.*

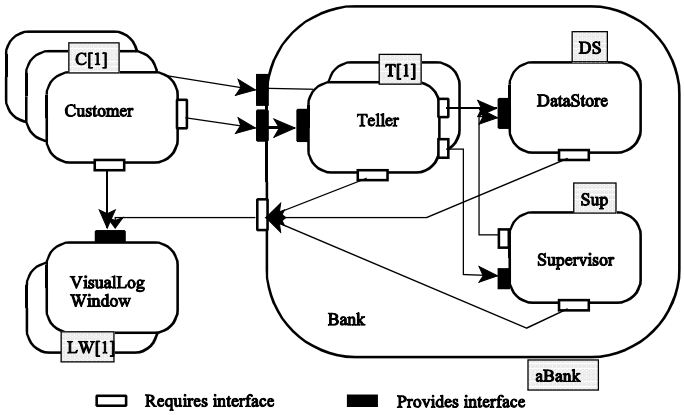

Figure 1. BankingDemo Architecture

For illustration, consider a bank where tellers serve a number of customers. Each customer requests a teller to perform a desired financial transaction(s) on an account(s). Certain transactions, such as an overdraft, require the teller to ask the supervisor for an approval. Each of these entities can be modeled as a component (Figure 1). The core of the application is the instance aBank of the Bank template (Bank component for short). The Bank component internally contains an array of Teller subcomponents (T[1], T[2], ... , T[N]), the Supervisor subcomponent, and the DataStore subcomponent. The Bank component features a number of provide interfaces, each of them being tied to a Customer component's requires interface, and internally tied (delegated) to the provides interface of a Teller subcomponent. The remaining part of the application is formed by the Customer and VisualLogWindow components, the latter serving for system administration purposes. The communication of the Customer components with the Bank component is based on procedure calls, while all the interaction with the VisualLogWindow components relies on event delivery.

2.2 Component lifecycle

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

Development and provision. The component is specified at the level of its template, i.e. component frame and component architecture is specified in an ADL; if the architecture is primitive, its implementation in an underlying programming language/environment has to be also supplied. As a frame F can be implemented by potentially several component architectures, each of such templates can be viewed as a design version of components of the type F.

Assembly. An application is assembled by choosing one particular component architecture for each frame involved recursively in the topmost frame of the 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.

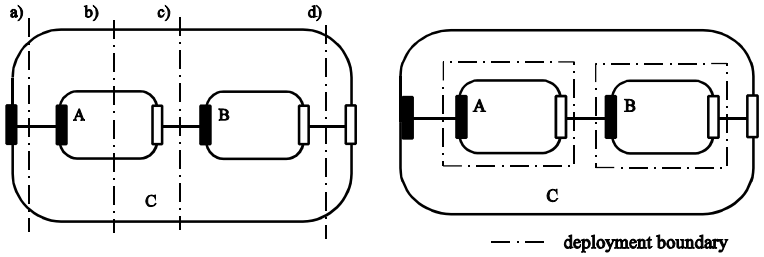

Figure 2. Deployment boundaries crossing interface ties

Distribution. To reflect its future distribution, the assembled application is divided into deployment units. Here, two approaches are to be considered: (1) Deployment unit boundaries can cross the component interface ties, but not the component/frame boundaries (Figure 2, right part). Advantageously, the deployment description of composed/nested components can be done on a top-down basis, following the hierarchy of components. (2) Deployment boundaries are orthogonal to component/frame boundaries. Thus, deployment boundaries can cross a component/frame boundary. Assuming that a primitive component cannot be distributed (see **Deployment** below), deployment boundaries can cross a component/frame boundary of compound components only (the

alternatives a), c) d) in the left part of Figure 2 are permitted, b) is not). Thus there is no difference in comparison with (1) when deploying primitive components. As to composed components, the following two problems are not easy to overcome: (i) the deployment description cannot parallel the hierarchy of component nesting, and (ii) the deployment of a composed component into more deployments docks (see below) may be a complex process.

Deployment time. The goal is to achieve *deployment* of the application, i.e. to associate each of its deployment units with a deployment dock and let these deployment docks start the application. In principle, a *deployment dock* serves as a component factory and a container which controls the lifecycle of running components. A deployment dock may be an instance of Java Virtual Machine, capable to load, instantiate, and run components written in Java, a processes capable to load dynamically linked native libraries and instantiate components into its address space, a daemon that instantiates components by starting new processes from binary executables, etc. In such settings, it is natural to require a primitive component not to be distributed.

2.3 Deployment anomaly

If a deployment unit boundary crosses the interface tie of two components A and B, the actual deployment of A and B in general substantially influences the communication of A and B. For example, in Figure 3a), the method calls on the r and q interfaces have to be modified in order to use an appropriate middleware technique of remote procedure calls (RPC), e.g. RMI stub and skeleton is to be employed. These modifications include changes to the internal architectures of A and B . Analogously with the inheritance anomaly concept [9, 18], we refer to this kind of a post-design modification of a component enforced by its deployment as *deployment anomaly*.

As a quick fix, one can imagine employing an ordinary component DC mediating the communication of A and B (Figure 3 b)). 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, DC has to be a composed component; this leads to the issue of adjusting the internals of some inner components of DC.

To illustrate the deployment anomaly on the BankingDemo example, consider the DataStore component is to be deployed in a separate deployment dock, compared to the rest of the application. Thus, inside the Bank component, all the interactions with DataStore have to be modified to be based on RPC. This post-design modification affects the Teller, Supervisor, and DataStore components.

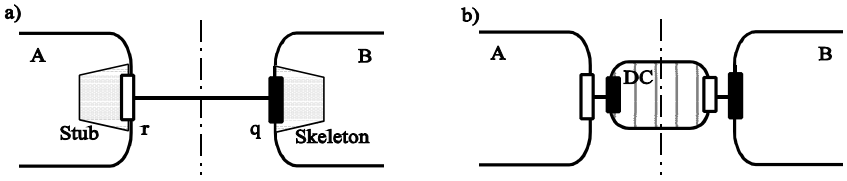

Figure 3. Deployment anomaly

2.4 Targeting the deployment anomaly: connectors

Basically, the deployment anomaly could be addressed by introducing a first class 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 the changes in communication induced by the deployment modification, a flexible parametrization system of the connector internals has to be provided. (3) To reflect inherent distribution, the deployment of a connector should not be specified explicitly, but inferred from the deployment description of the components involved in the communication the connector conveys. As a consequence, the lifecycle of a connector inherently differs from the lifecycle of a component as its underlaying code has to be supplied (e.g. semiautomatically generated) as late as its deployment is known (Section 4.3).

3. BASIC CONNECTOR TASKS

To understand the connector concept properly, it is useful to identify and analyze the basic tasks a connector should perform; here, the taxonomy of software connectors presented in [10] can be used for guidance. From the main service categories and the basic connector types of this taxonomy, we have selected the connector tasks listed below that we generally consider the key ones. In Section 5, we will show that most of the basic connector tasks can be provided through a simple hierarchical composition of a few primitive connector elements.

Control and data transfer. A connector specifies the mechanisms on which possible control and/or data transfer is based (like procedure call, event handling, and data stream). Each of these mechanisms has specific characteristics and properties, e.g., a procedure call can be local or remote. As to RPC, various kinds of middleware can be used to implement it. Similarly, event handling can be based on an event channel, a centralized event queue, etc.

Interface adaptation and data conversion. When facing the need to tie two (or more) components that have not been originally designed to interoperate, a straightforward idea is to include an *adaptor* into the connector abstraction. As mentioned in [16], there is the option (and challenge) to devise a mechanism for automatic or semi-automatic generation of adaptors and/or data convertors.

Access coordination and synchronization. In principle, the ordering of method calls on a component's interface is important (the protocol concept in [11]). The permitted orderings are usually determined by a behavioral specification of the component (e.g., interface, frame and architecture protocols in SOFA [17], CSP-based glue and computation in Wright). Thus another connector task is access coordination and synchronization - enforcing compliance with the protocol of an interface (set of interfaces). As an example, consider a server component, implemented for a singe-threaded environment, to be deployed into an environment with multiple client threads. The necessary serialization of threads can be achieved by a connector mediating the clients' access to the component.

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) which might help implement various filters (with applications in cryptography, data compression, load monitoring, debugging, etc.).

4. CONNECTOR MODEL

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

In this section, we propose a connector model designed as follows: Every interaction among components in an application is represented by a *connector* which is an instance of a *connector template.* Being generic in principle, a connector template is a pair <*connector frame*, *connector architecture*> that can be parameterized by *interface type* and *property* parameters. Given a connector template T, the *connector frame* specifies the black-box view of a T's instance (thus it can be referred to as connector type), while the *connector architecture* specifies the structure of a T's instance in terms of its *internal elements* (primitive elements, component instances, and instances of other connector templates) and their interactions (thus it can be referred to as connector implementation).

4.1 Connector frame

A connector frame is represented by a set of role instances. In principle, a *role* is a generic interface of the connector intended to be tied to a component interface. In a frame, a role is specified either as a *provides* role or *requires* role. A *provides* role serves as an entry point to the component interaction represented by the connector template instance and is intended to be tied to a (single) requires interface of a component (or to a requires role of another connector). Similarly, a *requires* role servers as an outlet point of the component interaction represented by the connector template instance and is intended to be tied to a (single) provides interface of a component (or to a provides role of another connector). In general, a role is an entity of a generic interface type; the actual interface type of a role R of a template T is implicitly determined by the specific interface (of a component or another connector) tied to R at the instantiation time of T.

4.2 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 instances of *primitive elements* only (Figure 4a); some of them can be specified as optional. Primitive elements are typed (usually generic types are employed - both the interface type and property parameters are allowed). For every primitive element type, in addition to functional specification in plain English, a precise specification of its semantics is given by mappings to underlying environments. For example: "Stub and skeleton elements provide the standard marshaling and unmarshaling functionality of RPC". Each of these elements is parameterized by its remote interface type and by the underlying implementation platform (specified as a property parameter). The mappings of the stub and skeleton element types exist for each of the implementation platforms supported (CORBA, Java RMI, etc.).

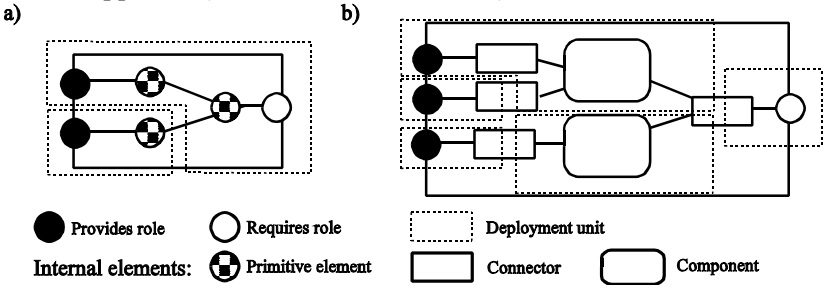

Figure 4. Connector model: a) simple architecture, b) compound architecture

The internal elements of a compound connector architecture are instances of other connector types and/or components (Figure 4b). This concept allows for creating complex connectors with hierarchically structured architectures reflecting the hierarchical nature of component interactions. For examples, we refer the reader to Sections 5.1 and 5.2.

4.3 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.

Connector design. The connector is specified as a template in ADL. For each of its primitive element types, a functional specification and definition of corresponding mappings (at least one) are to be provided. Since connectors are inherently distributed entities, the connector architecture is divided into a number of disjoint *deployment units*. A deployment unit is formed by the role instances and internal elements designed to share the same deployment dock.

Connector instantiation. The connector is instantiated within an application. Since the actual interface types of the entities tied by the connector instance become known at this point, the interface type parameters of the connector's roles can be resolved. Also the actual need for some of those primitive elements specified as optional at the design time (e.g. interface adaptors) reveals. A part of the connector instance remains generic - due to the unresolved property parameters related to a future deployment of the connector.

Connector deployment and generation. Connectors are deployed at the same time as the components the interactions of which they convey. To each of the connector's deployment units, a specific deployment dock is assigned. For a connector of simple architecture, the actual deployment docks of the connector's deployment units can be inferred from the locations of the components interconnected by the connector. The deployment of potential internal components of a connector is specified in the same way as the deployment of "ordinary" components in the application.

Once the deployment of a connector is known, the connector's implementation code is (semi-automatically) generated to use the communication primitives offered by the deployment docks's underlying environments. Note that the generated code of the primitive elements either follows their mapping to the underlying programming environment, or it can be null (e.g., no need for an adaptor). Note that the connectors with primitive architectures are considered for code generation while the connectors of compound architectures are created by composition of their internal elements.

A typical scenario of the code generation of a connector is as follows: (1) Using a deployment tool, the deployment of components (the interaction of which the connector conveys) is specified. (2) Each of the selected deployment docks is then asked 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 (the

returned list can be empty). (4) All returned lists are examined by the deployment tool in order to find a match in the offered technologies. (5a) If a matching technology exists, the deployment docks are asked to generate the connector's implementation code for the technology. (5b) If no matching technology exists, the user is given the options to either change the application's deployment, or to provide the connector's implementation manually.

5. CASE STUDY: SOFA/DCUP CONNECTORS

As a proof of the concept, the connector model described in Section 4 have been integrated into the SOFA/DCUP component mode [16,17]. This section describes this integration by introducing the *SOFA/DCUP connector model*.

5.1 Predefined connector templates

To avoid specifying the frequently used connector templates repeatedly, SOFA/DCUP provides a set of predefined connector templates - *CSProcCall*, *EventDelivery*, and *DataStream*. For brevity, only the CSProcCall connector template (Figure 5a) will be described here (for details see [2]).

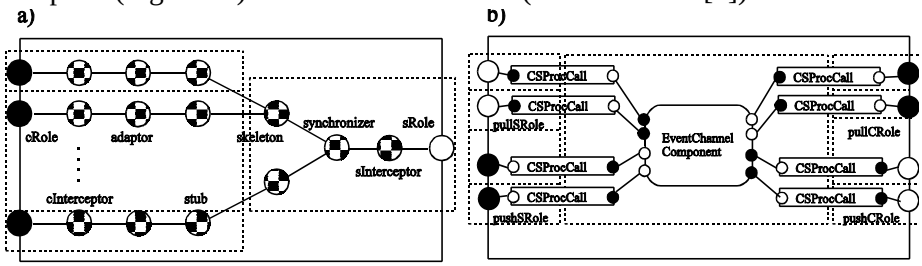

Figure 5. a) CSProcCall connector template, b) EventChannelDelivery connector template

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

The CSProcCall frame consists of a requires role to connect a server component (`sRole`), and of any number of provides roles to connect client

components (`cRole`). All of the roles are generic entities with interface type parameters.

The CSProcCall architecture is simple. It consists of several primitive elements interconnected in the way illustrated in Figure 5a). The `cInterceptor` and `sInterceptor` instances of `TInterceptor` provide a framework for plugging in an additional connector functionality to support logging, debugging, etc. An interface `adaptor` is (optionally) included in a connector instance if a particular client's interface does not match the server interface. A (`TStub`, `TSkeleton`) instance pair is used if a remote invocation is needed. These primitive elements provide the standard RPC marhalling and unmarshalling. A `synchronizer` is (optionally) included if the server component requires client invocations to be synchronized when accessing its interface.

There is an 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 following fragment of source text illustrates the main parts of the CSProcCall specification using the modified SOFA CDL notation.

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

5.2 User-defined connector templates

The process of creating a new connector template can be illustrated on the example of *EventChannelDelivery*, a connector template reflecting event-based communication via an event channel. Similarly to the CORBA Event Service, this connector template allows multiple suppliers to send data asynchronously to multiple consumers in both the push and pull modes.

The EventChannelDelivery frame consists of a number of roles to connect supplier components in the push and pull modes (`pushSRole` and `pullSRole`), and of a number of roles to connect consumer components in the push and pull modes (`pushCRole` and `pullCRole`). All of the roles are generic entities with interface type parameters.

The EventChannelDelivery architecture is compound. As depicted in Figure 5b), the core element of the EventChannelDelivery architecture is an instance of the `EventChannel` component. The other internal elements of EventChannelDelivery are instances of the CSProcCall connector template to tie EventChannelDelivery's roles to the EventChannel's interfaces.

The division into deployment units is illustrated in Figure 5b). It should be emphasized that while the deployment of the internal CSProcCall connectors is partially determined by the EventChannelDelivery's roles, the deployment of the EventChannel component (and related parts of CSPracCall connectors) has to be stated explicitly as with "ordinary" components.

5.3 Using SOFA/DCUP connectors

To demonstrate how the SOFA/DCUP connectors can be used, consider the DataStore and Supervisor components from the banking application introduced in Section 2.1. The following fragment of CDL specification illustrates their interconnection using the CSProcCall connector instance.

```
inst DataStore DS;
inst Supervisor Sup;
bind Sup.dsi to DS.dsi using CSProcCall;
```

Since the actual interfaces of the DataStore and Supervisor components are known at this point, the interface type parameters of the conveying connector are resolved. Assuming that the actual interfaces match, the interface adaptor

(as an optional element of the CSProcCall architecture) will be omitted. However, the rest of the connector architecture still remains generic due to the unresolved property parameters related to future deployments of the application.

Consider a deployment scenario which assumes that the DataStore and Supervisor components are to be deployed into separate deployment docks. Since both components do not share an address space, a cross-address space communication is needed. The stub and skeleton internal elements are therefore generated and included in the resulting connector.

6. EVALUATION AND CONCLUSION

As the first goal of the paper, we articulated the deployment anomaly as the necessity for a post-design modification of components caused by their particular deployment. This is a serious obstacle in using component-based real-life applications. In a practical setting, the deployment of a component - based application can be efficiently done 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. This issue can be addressed by the connector concept presented in the paper .

None of the current ADL languages/systems, such as [14, 19, 7, 21, 1], targets the deployment issue directly nor combines it with connectors). The second goal of the paper was therefore to propose a novel connector model allowing not only to express and represent a variety of possible interactions among components in an application at all key stages of the application lifecycle, but in particular to reflect component distribution.

In summary, in the presented component model, the **key difference between a component and connector** is in (1) distribution (a primitive connector can be distributed, while primitive component cannot) and (2) in the

lifecycle (parts of the connector can be generated only after all component deployment has been determined). In addressing the deployment anomaly, a **connector helps** in (3) separation of concerns (by separating the business and communication part of a component-based application), and in (4) 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"); the modification of the communication pattern for the actual interfaces is done in an automatized way, i.e. it can be generated.

Having finished a pilot implementation of our connector model, we currently focus on finding techniques for at least semi-automatic generation of primitive elements, including interface adaptors, stubs and skeletons for remote communication, etc. We believe this can be done by defining a mapping of every primitive element type to the underlying programming environment. Another future intention is to apply behavioral protocols [17] in connector specification to express the interplay of its internal elements.

REFERENCES

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

[2] Balek, D., Plasil, F.: A Hierarchical Model of Software Connectors, Tech. Report No. 2000/2, Department of SW Engineering, Charles University, Prague, 2000.

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

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

[5] 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.

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

[7] 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.

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

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

[10] 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.

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

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

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

[14] 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.

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

[16] 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.

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

[18] Plasil, F., Mikusik, D.: Inheriting Synchronization Protocols via Sound Enrichment Rules. In Proceedings of JMPLC, Springer LNCS 1204, March 1997.

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

[20] 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.

[21] 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.

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

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

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

[25] 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.