

Composing Connectors of Elements

Tomas Bures, Frantisek Plasil

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

Academy of Sciences of the Czech Republic
Institute of Computer Science
{bures,plasil}@cs.cas.cz, <http://www.cs.cas.cz>

Abstract

Connectors are used in component-based systems as first-class entities to abstract component interactions. To explain their responsibilities, several taxonomies have been published to date. However, most of them mix different levels of abstraction and fail to provide any guidelines that address different component interconnections through “real connectors” (employed in assembling real-life applications).

In this paper, we propose a way to compose connectors by using fine-grained elements, each of them representing only a single, well-defined functionality. We identify an experimentally proven set of connector elements, which, composed together, model four basic component interconnection types (procedure call, messaging, streaming, blackboard), and allow for connector variants as well (to reflect distribution, security, fault-tolerance, etc.).

The presented results are based on a proof-of-the-concept implementation where connectors are automatically generated (assuming description of connector structure and middleware technologies are provided). Thanks to the element composition approach, such connectors can support more middleware technologies at the same time.

Keywords: software connector, component-based programming, software architecture, middleware, component interconnections

1 Introduction and motivation

1.1 Background

The concept of “connector” was introduced in software architectures as a first-class entity representing component interactions [17]. The basic idea is that components of an application contain only the application business logic, leaving the component interaction-specific tasks to connectors. However, such characterization is too vague, since it does not strictly draw a line between component and connector responsibilities.

Different types of connectors are associated with architecture styles in [17] and analyzed as well as classified e.g. in [8,9]. There, every particular architecture is characterized by a specific pattern of components and connectors, and by specific communication styles (embodied in connectors). Thus, a style requires specific connector types. For example, in the pipe-and-filter architectural style, an application is a set of filters connected by pipes. As stream is here the inherent method of data communication, the pipe connector is used to mediate a unidirectional data stream from the output port of a filter to the input port of another filter. Interestingly, the main communication styles found in software architectures correspond to the types of interaction distinguished in different kinds of middleware – remote procedure call based middleware (e.g. CORBA [13], RMI [22]), message oriented middleware (e.g. JMS [21], CORBA Message Service [13], JORAM [12]), middleware for streaming (e.g. Helix DNA [7]), and distributed shared memory (e.g. JavaSpaces [23], relational databases).

In general, a communication style represents a basic contract among components; however, such a contract has to be more elaborated when additional requirements are imposed (e.g. security, transactions). This triggers the need to capture the details desirably not visible to components, but vital for an actual connection. This comprises the technology/middleware used to realize the connection, security issues such as encryption, quality of services, etc. These details are usually referred to as non-functional resp. extra-functional properties (NFPs). They should be considered an important part of a connector specification, since they influence the connection behavior (reflecting these properties directly in the components’ code can negatively influence the portability of the respective application across different platforms and middleware). The NFPs are addressed mainly in reflective middleware research [2,15], which actually does not consider the connectors (in terms of component-based systems), but implements a lot of their desired functionality.

To our knowledge, there are just few component models really supporting connectors in an implementation, e.g. [10] and [16]. However, these component systems do not consider middleware and do not deal with NFPs. As an aside, the term “connector” can be also found in EJB [19], to perform adaptation in order to incorporate legacy systems (capturing neither communication style nor NFPs though).

1.2 The goals and structure of the paper

As indicated in Sect. 1.1, a real problem with component models supporting connectors is that they are scarce and those existing do not benefit from the broad spectrum of functionality offered by the variety of existing middleware. Thus, a challenge is to create a connector model which would address this problem. Specifically, it should respect the choice of a particular communication style, offer a choice of NFPs, allow for automated generation of connector code to a large extent, and benefit from the features offered by the middleware on the target deployment nodes. With the aim to show that this is a realistic requirement, the goal of this paper is to present an elaboration of the connector model designed in our group [1,4] which covers most of the problem above, including connector generation and removal of the middleware-related code from components.

The goal is reflected in the structure of the paper in the following way. In Sect. 2, we focus on the basic communication styles supported by middleware and present a generic connector model able to capture these styles and also reflect NFPs. At the end of the section we present the way we generate connectors. In Sect. 3, we use the generic model to specify connector architecture for each of the communication styles with respect to a desired set of NFPs. An evaluation of our approach and related work are discussed in Sect. 4, while Sect. 5 summarizes the achievements.

2 Connectors vs. middleware

2.1 Component interactions reflected by middleware

In this section, we assume the connections among components are reflected in ADL (Architecture Description Language) via bindings of the components' interfaces (e.g. as in [18]). This assumption is mostly triggered by the practical need to employ an implementation environment/middleware based on subroutine calls. This assumption leads us to considering only the types of component interaction that are reflected in a middleware – procedure call, messaging, streaming, blackboard (see table below). Interestingly, these interaction types correspond also to the examples of connectors in [17].

Communication style	Description
Procedure call	A classical client server call. The client is blocked until the request is processed by the server and result is returned. <i>Example: CORBA remote procedure call</i>
Messaging	An asynchronous message delivery from a producer to the subscribed listeners. <i>Example: CORBA event channel service</i>
Streaming	A uni- or bidirectional stream of data between a sender and (multiple) recipients. <i>Example: Unix pipe</i>

Blackboard	A communication via shared memory. An object is referenced by a key. Using this key the object may be repeatedly read, written, and deleted. <i>Example: JavaSpaces</i>
------------	--

However, the connector instances reifying a particular component communication style can vary in the way they capture NFPs, such as real-time constraints, middleware interoperability, monitoring, and security, as well as fault tolerance, transaction context modification, etc. In Appendix I, for each communication style, we list the NFPs we consider important and sensible in middleware-based connectors. The features are mostly mutually orthogonal; the few cases where they are not are clearly indicated by grey bars.

2.2 Connector model and construction

We model connectors using a notation based on [1], capturing connectors as a composition of elements (Fig. 1). Using the elements we can model connectors with different NFPs. Compared to [1], we use structured connector elements to capture fine-grained parts of middleware, such as marshaling, unmarshaling, etc. Using the notation, Fig. 1 shows a sample architecture of a connector (reflecting the procedure call communication style) where several client components can have access to a single server component. *Roles*, the black resp. white circles are in principle generic interfaces of the connector. They will be later bound to a requires resp. provides interface of a component. They are on the connector *frame* (its boundary, the solid rectangle). Having a specific, typically elementary, functionality, each of the *elements* provides a part of the connector implementation. In principle, the designer of a connector specifies instances of elements and bindings between them (specifies the connector's *architecture*).

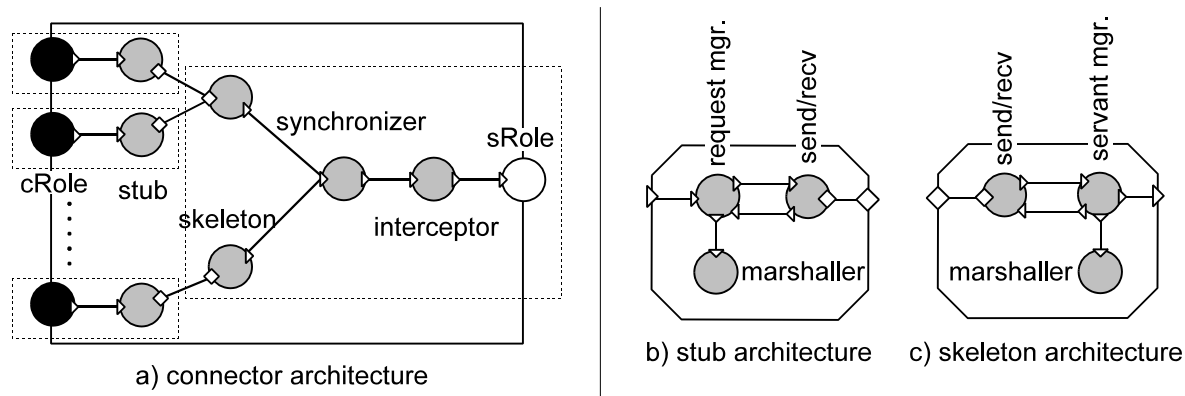


Figure 1 Example of a Procedure call connector

The use of deployment units (dotted lines) allows to express the inherently distributed nature of connectors. So, in principle, a deployment unit groups together the elements to be instantiated in a single address space. The responsibility for constituting a link between elements that crosses unit boundaries is delegated to the elements on both sides of the link. Typically, the underlying middleware is used to implement the link. Obviously, specification of deployment units has to be a part of connector architecture specification, because the boundaries have a significant impact on

the resulting architecture.

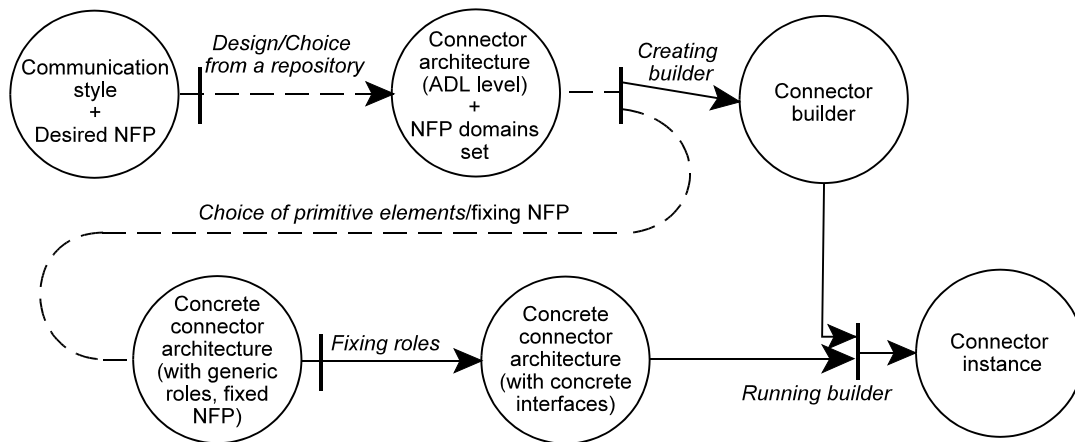


Figure 2 Connector evolution steps

The evolution of a connector comprises several activities captured on the activity diagram in Fig. 2. Based on a desired communication style and a set of envisioned NFPs, the developer designs a connector architecture by identifying the roles, elements, their links and distribution units and also identifies the potential value space of the associated NFPs. Typically, an architecture specification is written in an ADL notation, e.g.:

```

/* This is the ProcedureCall connector from Fig. 1 specified in the component      *
 * definition language of the SOFA component model (http://nenya.ms.mff.cuni.cz)  *
 * This is a fragment, full version in Appendix II                               */
connector frame ProcedureCall <ClientType, ServerType> {
  multiple role ClientRole {
    provides:
      ClientType ClientProv;
  };
  role ServerRole {
    requires:
      ServerType ServerReq;
  };
};

connector architecture SampleProcedureCall implements ProcedureCall {
  unit Client {
    inst EStub stub;
  };
  unit Server {
    inst ESkeleton skeleton;
    inst EInterceptor interceptor;
    bind skeleton.callOut to interceptor.in;
  };
  delegate ClientRole.ClientProv to Client.stub.callIn;
};
  
```

```
bind Client.stub.lineOut to Server.skeleton.lineIn;
bind Server.skeleton.lineOut to Client.stub.lineIn;
subsume Server.interceptor.out to ServerRole.ServerReq;
};
```

Now, for a specific architecture A and its set of NFP domains, two activities can take place simultaneously (for better understanding, we illustrate the process in terms of a Java implementation): (i) a connector builder is created (a Java class), serving as a factory for connectors based on A ; (ii) for each NFP domain D associated with A (Appendix I), a specific value $\text{nf}v_D \in D$ is chosen. Based on all the $\text{nf}v_D$ values chosen this way, specific connector element factories are selected (an element factory is a Java class which can later generate a concrete element class). Now, each role in A is substituted by the actual interface determined by a tied component. In addition, this substitution is “announced” to element factories which generate concrete elements (*element adaptation*). Finally, the connector builder is run to instantiate the whole connector from the adapted elements.

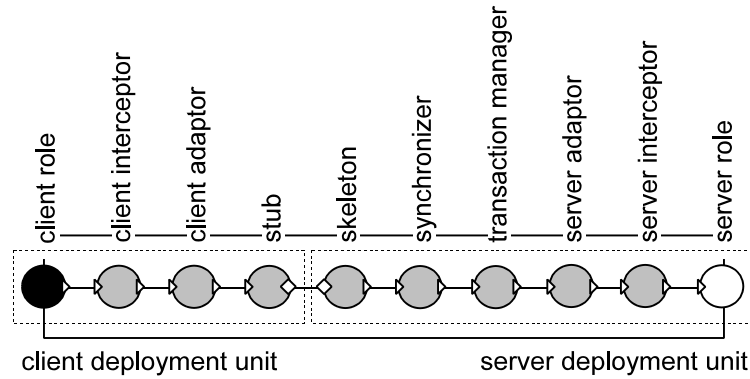
Several steps of the connector construction process can be automated. They are emphasized in Fig. 2 by solid lines (builder generation, element adaptation and assembly). The actions drawn by dashed lines have to be done manually; however, we believe that even they could be automated to a certain degree too.

3 Building real connectors

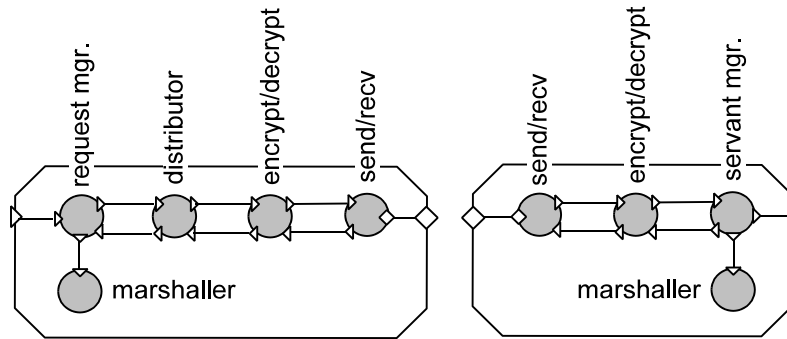
By analyzing several middleware designs and implementations [13,22,21,12,7,23], we have identified a list of NFPs which can be addressed in middleware (the list is in Appendix I). In this section, we suggest a connector architecture for each of the communication styles reflecting an appropriate spectrum of the identified NFPs. Similar to the example from Sect. 2, we map a single NFP to one or more connector elements organized in a specific pattern to achieve the desired connector functionality.

3.1 Procedure call

The proposed connector architecture for the procedure call communication style is depicted in Fig. 3a. It consists of a server deployment unit and multiple client deployment units. For simplicity, only one client deployment unit is shown. The other client units (identical in principle) are connected to the server deployment unit in the way illustrated in Fig. 1.



a) connector architecture



b) stub architecture

c) skeleton architecture

Figure 3 Proposed procedure call connector architecture

In summary, the connector NFPs (listed in Appendix I) are mapped to elements as described below. In principle, distribution is mapped to the existence of stubs and skeletons, and the NFPs dependent on distribution are reflected by the existence or variants of the elements inside the stub or the skeleton – encryption by an encrypt/decrypt element, connection quality by a send/recv element, and fault-tolerance by a distributor element replicating calls to multiple server units.

In more detail, the functionality of particular elements is following:

- *Roles*. Not reflecting any NFP, roles form the connector entry/exit generic points.
- *Interceptors* reflect the *monitoring property*. In principle, they do not modify the calls they intercept. If monitoring is not desired, these elements can be omitted.
- *Adaptors* implement the *adaptation property*. They solve minor incompatibilities in the interconnected components' interfaces by modifying the mediated calls. An adaptation can take place on the client side as well as on the server side (affecting thus all clients). If no adaptation is necessary, the elements can be omitted.
- *Stub*. Together with a skeleton element, stub implements the *distribution property*. This element transfers a call to the server side and waits for a response. The element can be either primitive (i.e. directly mapped to the underlying middleware) or compound. A typical architecture of a stub is on Fig. 3b. It consists of a request manager, which, using the attached marshaller, creates a request from the incoming calls and blocks the client thread until a response is received. An encryption element reflects the *encryption property*; sender/receiver elements transport a stream

of data and also reflect the *connection quality property*. The *fault-tolerance property* is implemented by a distributor performing call replication. The stub element is needed only when distribution is required.

- *Skeleton* is a counterpart of the stub element. Again, its architecture can be primitive or compound (Fig. 3c). The elements in the compound architecture are similar to those in compound stub. The servant manager uses the attached marshaller to create a call from the received data and assigns it to a worker thread. Again, skeleton can be omitted if distribution is not required.
- *Synchronizer* reflects the *threading policy property*. It synchronizes the calls going to the server component, allowing, e.g., a thread-unaware code to work properly in a multithreaded environment.
- *Transaction mgr.* implements the *transaction property*. When appropriate, it can modify the transaction context of the mediated calls.

3.2 Messaging

The proposed connector architecture for the messaging communication style is depicted in Fig. 4a. It consists of a distributor deployment unit and several sender/recipient units. (In a fault-tolerant case, there can be multiple distributor deployment units.) The sender/recipient deployment unit allows for sending messages to other attached components (as well as for receiving messages from them). The distributor deployment unit is in the middle of this logical routing star. For simplicity, only one sender/recipient deployment unit is shown. Other sender/recipient units would be connected to the distributor deployment unit in a similar way.

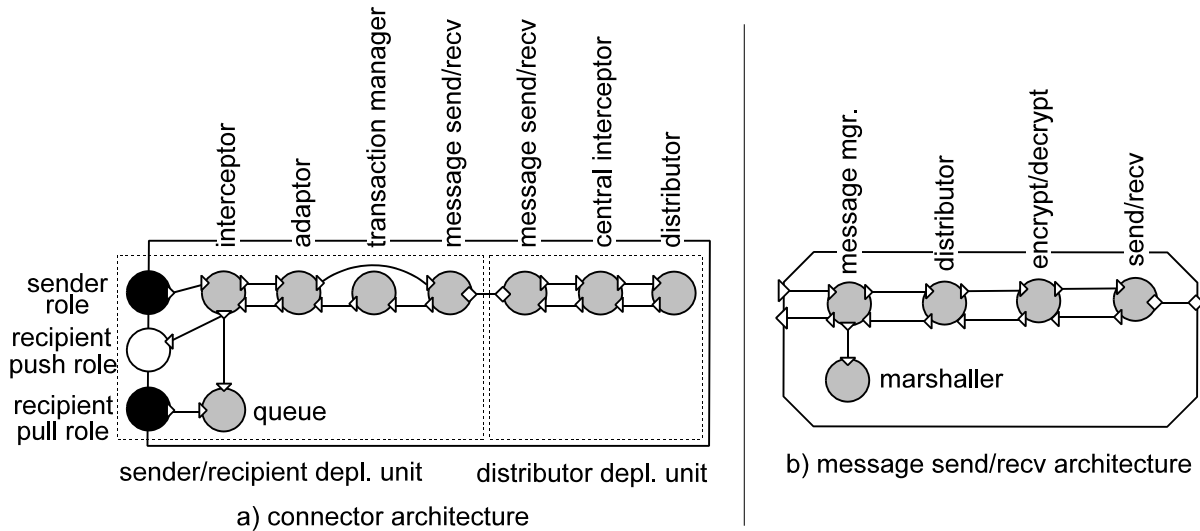


Figure 4 Proposed messaging connector architecture

The connector NFPs (listed in Appendix I) are mapped to elements as described below.

- *Roles*. The sender role servers to sending messages. Depending on the *recipient mode property*, the reception can work either in push mode, employing the push role to automatically deliver the incoming messages to the attached component via a callback interface; or in pull mode,

when the attached component polls for new messages via the pull role. If the component does not need to receive messages, the recipient role can remain unconnected.

- *Queue*. Together with the pull role, it implements the pull variant of the *recipient mode property*. Thus, the queue is present only when the message reception works in pull mode to buffer the incoming messages if necessary.
- *Interceptors* implement the *monitoring property* (similarly to the procedure call architecture).
- *Adaptor* reflects the *adaptation property* by modifying the mediated messages.
- *Transaction mgr.* implements the *transaction property*. Its presence is meaningful only if message reception operates in push mode.
- *Message sender/receiver* realize the *distribution property*. Each of them performs communication with remote nodes. It can be either primitive (directly implemented by underlying middleware) or compound (its typical architecture is on Fig. 4b). It is similar to the stub element, however the request manager is replaced by a message manager which allows the messages to be transferred in both directions. The distributor deployment unit supports implementation of the *fault-tolerance property*.
- *Distributor*. Being inherent to the communication style, it is a central part of the connector architecture. It distributes all the incoming messages to the attached recipient components. The element reflects *delivery strategy property* by implementing different policies for message routing (one recipient, all recipients, group address, etc.).

3.3 Streaming

The proposed connector architecture for the streaming communication style (Fig. 5a) consists of a number of send/rcv deployment units. There are two basic operating modes depending on the *recipient mode property*: In full duplex mode, a point-to-point component connection is realized as a pair of send/rcv deployment units communicating through their stream access elements; in half-duplex mode, each of one sender and multiple receivers is formed as a send/rcv deployment unit. For simplicity, only one such deployment unit is shown on Fig. 5.

The connector NFPs (listed in Appendix I) are mapped to the elements as described below.

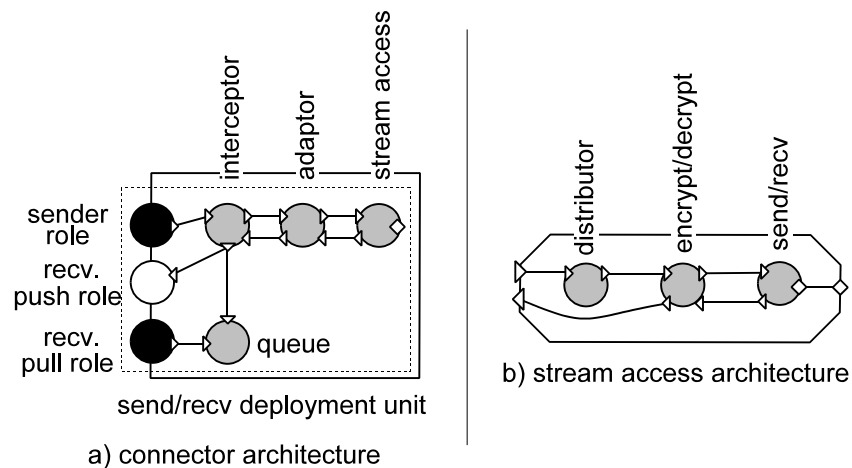


Figure 5 Proposed data stream connector architecture

- *Roles*. The streaming connector type allows for transmitting a data stream (containing both input and output streams in general). The sender role is dedicated for writing to the output stream. Depending on the *recipient mode property*, the input stream is processed either in pull or push mode. In push mode, the attached component receives the incoming data via the *recv. push role* (callback). In pull mode, the component has to poll for data.
- *Queue*. Together with the pull role, it implements the pull variant of the *recipient mode property*. It holds the incoming data to be processed.
- *Interceptor* implements the *monitoring property* (similar to the procedure call architecture).
- *Adaptor* reflects the *adaptation property* by modifying the mediated data.
- *Stream access* reflects the *distribution property* by transmitting a data stream. It can be directly implemented by underlying middleware or composed of other elements (Fig. 5b). The architecture of the latter is very similar to the architecture of the stub element (in the procedure call architecture).

3.4 Blackboard

The proposed connector architecture for the blackboard communication style consists of a storage deployment unit and multiple access deployment units (for simplicity, Fig. 6a depicts only one access deployment unit). The access deployment units mediate access to the storage deployment unit and notifications about changes in the stored data. The storage deployment unit stands in the middle of this “communication star”.

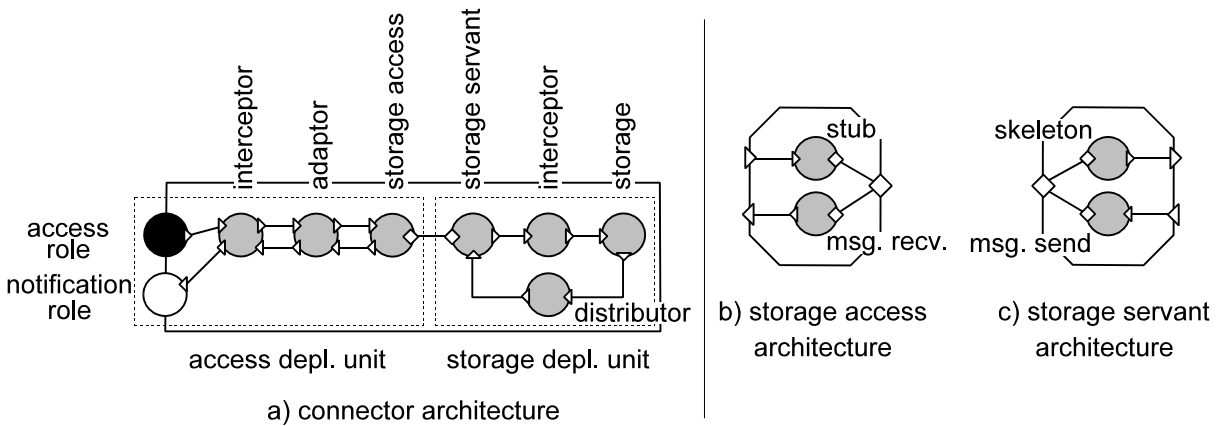


Figure 6 Proposed blackboard connector architecture

The connector NFPs (listed in Appendix I) are mapped to elements as described below.

- *Roles*. The access role is used for retrieving and storing data; the notification role embodies a callback interface through which the attached component is notified about a change in the posted data.
- *Interceptors* reflect the *monitoring property*.
- *Adaptor* reflects the *adaptation property*.
- *Storage* contains all the stored data. It may be realized in a simple case as a hash-table, or, in a more advanced way, as a relational database. This element, inherent to the communication style, can also implement the *locking property*.

- *Distributor* is used to distribute notifications among the connected clients. Being inherent to the communication style, it does not reflect any NFP.
- *Storage access*. Together with a storage servant, it implements the *distribution property*. The element realizes the client side of a remote access to the storage node. Basically, it has two responsibilities: (i) to mediate operations on the storage servant (store, fetch, etc.), and (ii) to pass notifications from the storage servant to the client. The element is either primitive (i.e. directly implemented) or compound. The architecture of the compound storage access element is modeled as a stub element for client->storage calls and a message send/recv. element for storage->client notifications (Fig. 6b). Both of these elements are either primitive or compound (their architectures would be similar to those in the procedure call and messaging connector architectures).
- *Storage servant*. Together with its counterpart (storage access), it implements the *distribution property*. The element is either primitive or compound (Fig. 6c).

4 Evaluation and related work

To our knowledge, there is no related work addressing all of the following issues in a single component model and/or in its implementation. 1) Reflecting the component interaction types which are supported by existing middleware, 2) providing the option of choosing from a set of NFPs, and 3) at least a partial generation of a connector with respect to the middleware available on target deployment nodes.

In addressing the first issue, we have identified four basic communication styles that are directly supported by middleware (i.e. procedure call, messaging, streaming, blackboard). These styles correspond to the connector types mentioned in software architectures in [17]. Medvidovic et al. in [9] go further and propose additional connector types (adaptor, linkage, etc.); in our view, being at a lower abstraction level, these extra connector types are a potential functional part of the basic four connector types (e.g., adaptation is a feature of each of our connector types).

To address the second issue, we have chosen the approach of reflecting a specific NFP as a set of reusable connector elements. Following the idea of capturing all the communication related functionality in a connector (leaving the component code free of any middleware dependent fragments), we have managed to compose the key connector types in such a way that NFPs are realized via connector elements and a change to a NFP implies only a replacement of few connector elements, leaving the component code untouched. Here, our approach is similar to reflective middleware [2,3,6,15], which is also composed of smaller parts; here, however, middleware-dependent code is inherently present in a component, making it less portable. Our work is also related to [5], which proposes a way to unify the view on NFPs influencing quality of service in real-time CORBA. It does not consider different communication styles, connectors as the communication mediators, and relies on having the source code of both the application and the middleware available.

In addressing the third issue (automatic generation), we have automated the connector builder generation, element adaptation, and connector assembly; however we plan to automate to a certain degree the design process of a connector architecture, including a supporting tool for connector element choice. The idea of automated middleware communication-related code generation is

employed in ProActive [11], where stubs and skeletons are generated at run-time. However, ProActive is bound only to Java, does not consider other communication styles than RPC, and does not address NFPs.

Prototype implementation: As a proof of the concept, a prototype implementation of a connector generator for SOFA component model [14] is available [18], implementing three of the proposed four communication styles (procedure call, messaging, and datastream). Designed as an open system employing plugins for an easy modification, the connector generator allows, e.g., to switch transparently between RMI and CORBA (Java IDL [20]), as well as combine these middleware technologies.

5 Summary

In this paper, we presented a way to model and generate “real connectors” employing existing middleware. We have elaborated the connector model initially proposed in [1] to reflect the commonly used communication styles, as well as non- and extra-functional properties. In addition to separating the communication-related code from the functional code of the components, the model allowed us to partially generate connectors automatically to respect (i) the middleware available on the target nodes determined by component deployment, and (ii) the desired communication style and NFPs. Our further intentions include an elaboration of automatic connector generation, including a tool supporting the connector design process.

Acknowledgments

We would like to give special credit to Lubomir Bulej, a coauthor of [4]. Special thanks go to Petr Tuma, Vladimir Mencl and other colleagues in our group for their valuable comments. Also, Petr Hnetynka deserves special credit for incorporating the implementation of the connector generator into the SOFA framework. This work was partially supported by the Grant Agency of the Czech Republic (project numbers 102/03/0672 and 201/03/0911) and the OSMOSE/ITEA project.

References

- [1] Balek, D., Plasil, F.: Software Connectors and Their Role in Component Deployment, In Proceedings of DAIS'01, Krakow, Kluwer, September 2001
- [2] Blair, G. S., et al.: A Principled Approach to Supporting Adaptation in Distributed Mobile Environments, International Symposium on Software Engineering for Parallel and Distributed Systems, Limerick, Ireland, June 2000
- [3] Blair, G., Blair, L., Issarny, V., Tuma, P., Zarras, A.: The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms, Proceedings of Middleware 2000, IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, Hudson River Valley (NY), USA. Springer Verlag, LNCS, April 2000
- [4] Bulej, L., Bures, T.: A Connector Model Suitable for Automatic Generation of Connectors. Tech. Report No. 2003/1, Dep. of SW Engineering, Charles University, Prague, 2003
- [5] Cross, J.K., Schmidt, D. C.: Quality Connectors. Meta-Programming Techniques for Distributed Real-time and Embedded Systems, the 7th IEEE Workshop on Object-oriented Real-time Dependable Systems, San Diego,

January 2000

- [6] Dumant, B., Horn, F., Dang Tran, F., Stefani, J.-B.: Jonathan: an Open Distributed Processing Environment in Java, 1998
- [7] Helix Community: Helix DNA, <https://www.helixcommunity.org/>
- [8] Medvidovic, N., Mehta, N. R.: Distilling Software Architecture Primitives form Architectural Styles. TR UCSCSE 2002-509
- [9] Medvidovic, N., Taylor, R. N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, Vol. 26, No. 1, January 2000
- [10] 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, 1997
- [11] ObjectWeb Consortium: ProActive manual version 1.0.1, January 2003
- [12] ObjectWeb Consortium: JORAM: Java Open Reliable Asynchronous Messaging, <http://www.objectweb.org/joram/index.html>
- [13] OMG formal/02-12-06: The Common Object Request Broker Architecture: Core Specification, v3.0, December 2002
- [14] Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, Proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998
- [15] Putman, J., Hybertson, D.: Interaction Framework for Interoperability and Behavioral Analyses, ECOOP Workshop on Object Interoperability, 2000
- [16] Shaw, M., DeLine, R., Zalesnik, G.: Abstractions and Implementations for Architectural Connections. Proceedings of the 3rd International Conference on Configurable Distributed Systems, May 1996
- [17] Shaw, M., Garlan, D.: Software Architecture, Prentice Hall, 1996
- [18] The SOFA Project, <http://sofa.debian-sf.objectweb.org/>
- [19] Sun Microsystems, Inc.: Enterprise JavaBeans Specification 2.0, Final Release, August 2001
- [20] Sun Microsystems, Inc.: Java IDL, <http://java.sun.com/j2se/1.4.1/docs/guide/idl/index.html>
- [21] Sun Microsystems, Inc.: Java Message Service, April 2002
- [22] Sun Microsystems, Inc.: Java Remote Method Invocation Specification – Java 2 SDK, v1.4.1, 2002
- [23] Sun Microsystems, Inc.: JavaSpaces Service Specification, April 2002

Appendix I

Procedure Call		
Feature name	Comment	
distribution	The connection may be either in one address space or span across several address spaces and/or computer nodes.	
distributed	encryption	Encryption can be employed to provide security even on insecure lines.
	connection quality	It may be necessary to assure some quality of connection (e.g. maximal latency, throughput etc.)
	fault-tolerance	The connector can support replication to make the server fault-tolerant.
threading policy	The calls may be serialized (single-threaded) or left unchanged.	
adaptation	Both the calls and their parameters may be modified in order to allow incompatible component interfaces to cooperate.	
monitoring	The calls and their parameters may be monitored to allow for profiling and other statistics (usage, throughput, etc.)	
transactions	This feature specifies how to handle the transactional context (e.g. propagate the clients' transaction at the callee side)	

Streaming		
Feature name	Comment	
distribution	The data may be exchanged within only one address space or across several address spaces and computers.	
distributed	encryption	Encryption can be employed to provide security even on insecure lines.
	connection quality	It may be necessary to assure some quality of connection (e.g. maximal latency, throughput, etc.)
	fault-tolerance	Allows for groups of replicas instead of single recipients making the application fault tolerant.
adaptation	The transmitted stream may be modified in order to allow incompatible components to cooperate.	
monitoring	The transmitted messages may be monitored allowing for profiling and other statistics (usage, throughput, etc.)	
duplexity	The connector may be either unidirectional (half-duplex) or bidirectional (full-duplex)	
half-dup.	multicast	If the connector is half-duplex, the stream can have more recipients, allowing for e.g. audio and video broadcasting.
recipient pull/push mode	Every recipient can work either in pull or push mode. In push mode the received data are immediately given to recipient (the recipient "receive" method is invoked). In pull mode the recipient actively polls for incoming data.	

Messaging		
Feature name	Comment	
distribution	The messages may be exchanged within only one address space or across several address spaces and computers.	
distributed	encryption	Encryption can be employed to provide security even on insecure lines.
	connection quality	It may be necessary to assure some quality of connection (e.g. maximal latency, etc.)
	fault-tolerance	Allows to groups of replicas instead of single recipient making the application fault tolerant.
adaptation	The transmitted messages may be modified in order to allow incompatible components to cooperate.	
monitoring	The transmitted messages may be monitored allowing for profiling and other statistics (usage, throughput, etc.)	
transactions	This feature specifies how to handle the transactional context (e.g. requires, requires new, etc.)	
delivery strategy	This feature controls to whom the message should be delivered. Possible values may be: exactly one, at least one, all.	
recipient pull/push mode	Every recipient can work either in pull or push mode. In push mode every new message is immediately given to recipient (the recipient "accept message" method is invoked). In pull mode the recipient actively polls the incoming queue for new messages.	

Blackboard		
Feature name	Comment	
distribution	The data may be shared pro components residing only in one address space or by components spanned across networks.	
distributed	encryption	Encryption can be employed to provide security even on insecure lines.
	connection quality	It may be necessary to assure some quality of connection (e.g. maximal latency, throughput, etc.)
adaptation	The accessed values may be transparently modified in order to allow incompatible components to cooperate.	
monitoring	The accessed data may be monitored allowing for profiling and other statistics (usage, throughput, etc.)	
locking	An attached component may obtain a lock onto a set of keys. The other components accessing the same data are temporarily blocked.	

Appendix II

```
/* This is the ProcedureCall connector from Fig. 1 specified in the component      *
 * definition language of the SOFA component model (http://nenya.ms.mff.cuni.cz)  */
connector frame ProcedureCall <ClientType, ServerType> {
  multiple role ClientRole {
    provides:
      ClientType ClientProv;
  };
  role ServerRole {
    requires:
      ServerType ServerReq;
  };
};

connector architecture SampleProcedureCall implements ProcedureCall {
  unit Client {
    inst EStub stub;
  };
  unit Server {
    inst ESkeleton skeleton;
    inst EInterceptor interceptor;
    bind skeleton.callOut to interceptor.in;
  };
  delegate ClientRole.ClientProv to Client.stub.callIn;
  bind Client.stub.lineOut to Server.skeleton.lineIn;
  bind Server.skeleton.lineOut to Client.stub.lineIn;
  subsume Server.interceptor.out to ServerRole.ServerReq;
};

element frame EStub <T> {
  provides: T callIn;
  requires remote: lineOut;
};

element frame ESkeleton <T> {
  requires: T callOut;
  provides remote: lineIn;
};

element architecture ReflectiveStub implements EStub {
  inst EReqManager reqManager;
  inst EMarshaller marshaller;
  inst ESendRecv sendRecv;
  bind reqManager.requestOut to marshaller.requestIn;
  bind marshaller.requestOut to reqManager.requestIn;
  bind marshaller.dataOut to sendRecv.dataIn;
  bind sendRecv.dataOut to marshaller.dataIn;
  subsume sendRecv.lineOut to lineOut;
  delegate callIn to reqManager.callIn;
};

element architecture ReflectiveSkeleton implements ESkeleton {
  inst ESendRecv sendRecv;
  inst EMarshaller marshaller;
  inst EThrManager thrManager;
  bind sendRecv.dataOut to marshaller.dataIn;
  bind marshaller.dataOut to sendRecv.dataIn;
  bind marshaller.requestOut to thrManager.requestIn;
  bind thrManager.requestOut to thrManager.requestIn;
  delegate lineIn to sendRecv.lineIn;
  delegate thrManager.callOut to callOut;
};
```

