Communication Style Driven Connector Configurations

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. In this paper, we propose a way to compose connectors by using fine-grained elements, each of them representing a single, well-defined function. 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 to reflect distribution, security, fault-tolerance, etc. We also discuss how to generate such a connector semi-automatically. The presented results are based on a proof-of-the-concept implementation.

1 Introduction and motivation

1.1 Background

"Connector" was introduced in software architectures as a first-class entity representing component interactions [21]. The basic idea is that application components contain only the application business logic, leaving the component interaction-specific tasks to connectors. However, such a 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 [21] and analyzed as well as classified, e.g., in [11, 12]. Every architectural style is characterized by a specific pattern of components and connectors, and by specific communication styles (embodied in connectors). Thus, an architecture 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 [16], RMI [26]), message oriented middleware (e.g. JMS [25], CORBA Message Service [16], JORAM [14]), middleware for streaming (e.g. Helix DNA [10]), and distributed shared memory (e.g. JavaSpaces [27]).

In general, a communication style represents a basic contract among components; however, such a contract has to be further elaborated when additional requirements are imposed (e.g. security, transactions). This triggers the need to capture the details 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 [5,19], 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 very few component models supporting connectors in an implementation, e.g. [13] and [20]. 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 [23] to perform adaptation in order to incorporate legacy systems, but capturing neither communication style nor NFPs.

1.2 The goals and structure of the paper

As indicated in Section 1.1, a real problem with component models supporting connectors is that 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, 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 [3,6] which covers most of the problems 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 Section 2, we focus on the basic communication styles supported by middleware and present a generic connector model able to capture these styles and also to reflect NFPs. In the second part of Section 2 we present the way we generate connectors. In Section 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 Section 4, while Section 5 summarizes the contributions.

Table 1. Communication styles

Communication style	Description
Procedure call	Classic client server call. A client is blocked until its request is processed by a server and result is returned. *Example: CORBA remote procedure call*
Messaging	Asynchronous message delivery from a producer to subscribed listeners. *Example: CORBA event channel service*
Streaming	Uni- or bidirectional stream of data between a sender and (multiple) recipients. *Example: Unix pipe*
Blackboard	Communication via shared memory. An object is referenced by a key. Using this key the object may be repeatedly read, written, and deleted. *Example: JavaSpaces*

2 Connector model

2.1 Key factors determining a connector

As we assume the runtime communication among components will be mediated via a middleware, the key factors determining a connector include choice of (i) communication style, (ii) NFPs, and (iii) the actual deployment of individual components.

Communication style. Typically, middleware supports the following communication styles: procedure call, messaging, streaming, and blackboard (Table 1). Interestingly, these interaction types correspond also to the examples of connectors in [21].

NFPs. A middleware-based implementation of a communication style can vary depending upon the desired NFPs, such as real-time constraints, middleware interoperability, monitoring, and security, as well as fault tolerance, transaction context modification, etc. In the Appendix, for each communication style, we list the NFPs we consider important and sensible in middleware-based connectors. The features are orthogonal at the same level of their hierarchy.

Deployment. We expect components to be deployed in a heterogenous distributed environment (with respect to the middleware available on each of the nodes composing the environment). Therefore deployment must inherently ensure interoperability of the used middleware technologies – "agreement on middleware". In our approach, the abstraction of a node performing negotiation upon agreement on middleware, instantiation of components, and providing their run time environment is *deployment dock* [22].

2.2 Architecture level specification

At architecture level, we model connectors using a notation based on [3], capturing connectors as a composition of connector elements which can be primitive or composed. Contrary to [3], we use composed elements to capture fine-grain parts of middleware, such as marshaling, unmarshaling, etc. Using the notation, Figure 1 shows a sample architecture of a connector reflecting the procedure call

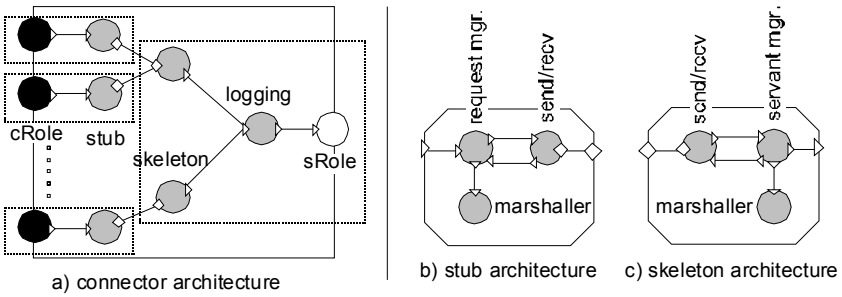

Fig. 1. Example of a Procedure call connector

communication style, where several client components can have access to a single server component. *Roles* (circles) are in principle generic interfaces of the connector. They will be bound later to a requires (white circle) resp. provides (black circle) interface of a component. They are on the connector *frame* (its boundary, solid rectangle). Having a specific functionality, each of the *elements* provides a part of the connector implementation. In principle, the *architecture* of a connector is determined by its frame, its internal elements, and bindings between these elements' *ports* (symbols ▷ resp. ◇; by convention ▷ denotes a local and ◇ remote binding).

Deployment units (dotted lines) define the distribution boundaries of a connector. In principle, a deployment unit groups together the elements to be instantiated in a single address space. The responsibility for establishing a link between elements that cross unit boundaries is split between the elements on both sides of the link.

The following is an example of an architecture level connector specification in our notation:

```
/* This is the ProcedureCall connector from Figure 1
   specified in the component definition language of
   the SOFA component model (http://nenya.ms.mff.cuni.cz).
   Full version of this fragment is in [7]*/
connector frame ProcedureCall <ClientType, ServerType> {
 multiple role ClientRole {
  provides: ClientType ClientProv;
 };
 role ServerRole {
  requires: ServerType ServerReq;
 };
};

connector architecture RemoteProcedureCallImpl
implements ProcedureCall {
 unit Client {
  inst EStub stub;
 };
 unit Server {
  inst ESkeleton skeleton;
  inst EInterceptor logging;
  bind skeleton.callOut to logging.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.logging.out to ServerRole.ServerReq;
};
```

2.3 Connector configuration

A realization of a particular connector instance is fully determined by a *connector configuration*, which is a connector architecture with implementation assigned to all of its internal elements. As the rest of this section deals with the selection of the right connector configuration, we first formalize the concept of connector configuration in order to capture all the important relationships of the related abstractions.

Let *Arch* be the set of available connector architectures and *Impl* the set of available element implementations. Combining architectures with element implementations we get a set *Conf* of connector configurations, where a $c \in Conf$ determines a connector implementation. Moreover, an $i \in Impl$ is associated with its type denoted *Type(i)* (e.g. 'EInterceptor'), the function *Elements: Arch -> P({<*e_{name}*,*e_{type}*>})* returns a set of the elements contained in $a \in Arch$; here *P(X)* denotes the powerset of *X*. For the example above, *Elements(a)* is the set {<'skeleton', 'ESkeleton'>, <'logging', 'EInterceptor'>,...}. This way:

$Conf=\{<a,s,f> \mid a \in Arch \ \& \ s \in P(Impl) \ \& \ f:Elements(a) \rightarrow s$ *is a surjective mapping such that if* $f(<e_{name},e_{type}>)=i$, *then* $Type(i)=e_{type}\}$

Informally, *f* is a function that maps the elements of the architecture *a* to their implementations (while the type of an element and its implementation is the same).

Formalization of connectors with nested elements would be done in a similar way, except that the assignment of element implementations would be recursively applied to the nested elements.

2.4 Connector construction and instantiation

A connector is constructed and instantiated in the following three steps:

A connector configuration $<a,s,f>$ is selected with respect to the desired communication style, NFPs, and deployment.

1. The generic connector roles are anchored to the specific component interfaces causing the generic interfaces in the elements in *s* to be anchored as well (element adaptation).
2. The adapted element implementations are instantiated at runtime and bound together according to *a*.

Described in [6], the steps 2 and 3 are rather technical. The step 1 is more challenging since it involves both choosing the "right" architecture *a* and finding the "right" elements fitting into *a* . In our approach we traverse the whole set *Conf* looking for such a configuration that would comply with the desired communication

style, NFPs, and deployment. Given a configuration $c \in Conf$, these three factors are reflected in the following consistency rules:

a. *Communication style consistency*. The configuration *c* implements the desired communication style.
b. *NFP consistency*. The configuration *c* complies with all desired NFPs.
c. *Environment consistency*. All element implementations used in the configuration *c* are compatible with the target platform (all necessary libraries are present, etc.).

In general, the set *Conf* may contain configurations in which the elements cannot inherently cooperate (e.g., a configuration for the procedure call communication style featuring a CORBA-based stub, but RMI-based skeletons). To identify such configurations, we introduce:

d. *Cooperation consistency*. The element implementations in *c* can cooperate (they can agree on a common middleware protocol, encryption protocol, etc.).

2.5 Choosing the right configuration

The four consistency rules above allow to automate the choice of an appropriate connector configuration by employing constraint programming techniques (e.g. backtracking). To employ the consistency rules in such a technique, we formulate the rules as logical predicates and illustrate their construction bellow.

NFP consistency: It checks whether, for a particular connector configuration $c \in Conf$, a NFP *p* (e.g. *distribution*) has a specific value *v* (e.g. 'local', 'CORBA', 'RMI', 'SunRPC').

First of all, we associate every element implementation $i \in Impl$ with a predicate to check whether the value of an arbitrary NFP *p* is *v*. For example, considering an RMI implementation of a stub element, such predicate can take the following form (in Prolog notation):

```
nfp_mapping_distrib_RMIStubImpl('distribution','RMI',
 ConConfig):-con_arch_name(ConConfig,'RMIStubImpl').
```

Now, to check the same NFP at the architecture level, we associate every architecture $a \in Arch$ with a "higher-level" predicate referring to all predicates that check *p* at element level. Technically, we write a generic predicate which will be employed whenever a configuration with architecture *a* is used (the predicate takes configuration as a parameter). For example, for the architecture from Figure 1a such predicate would be:

```
nfp_mapping_distrib_RPCImpl('distribution',NFPValue,
 ConConfig):-
 (con_arch_name(ConConfig,'RemoteProcCallImpl'),
  con_elem(ConConfig,'Stub'),
  nfp_mapping('distribution',NFPValue,StubElemConfig)).
```

Finally, to get a single predicate checking an arbitrary NFP of any configuration, we compose all the previously defined predicates (for architectures and elements) using disjunction:

```
nfp_mapping(NFPName,NFPVal,ConConfig):-
 nfp_mapping_distrib_RPCImpl(NFPName,NFPVal,ConConfig) ;
 nfp_mapping_distrib_RMIStubImpl(NFPName,NFPVal,ConConfig) ;
 ... .
```

The whole trick is that a term in the disjunction chain fails if it cannot "say anything about a given configuration".

Communication style consistency: Each of the communication styles we consider (Table 1) is reflected in a very specific connector frame (there is no frame reflecting two communication styles). Therefore, a communication style consistency check results in a simple test whether a particular architecture really `implements` the frame corresponding to the desired communication style.

Environment consistency: In order to identify the element implementations which would not run properly on a target platform, we associate each element implementation with a set of environment requirements that have to be met by the target platform. For the time being, we use a simple set of key-value pairs (such as "SunRCP"=>"2") for the requirement specification. Every target deployment node declares its "environment provisions" in form of key-value pairs as well. Environment consistency is then guaranteed if, for each element implementation used in a configuration, its set of environment requirements is a subset of the environment provisions of the deployment node where the element is to be instantiated. This approach is very simplistic though, not allowing to express complex requirements. (We plan to design a more sophisticated requirement specification technique in the future.)

Cooperation consistency: Cooperation consistency guarantees that all the element implementations in a configuration are able to cooperate (i.e. "they speak the same protocol"). Cooperation consistency is verified in two steps::

1. Every element implementation $i \in Impl$ is associated with a *transition predicate*, which expresses a relation of interfaces among the ports of i. This describes "how an interface changes" when mediated by a particular element implementation.
2. For every binding in the architecture of a given configuration, we construct a *binding predicate*, which captures the fact that two bound elements must use subtype compatible [18] interfaces to communicate with each other.

Finally, all the *transition predicates* and *binding predicates* are tied together using conjunction to form a composed predicate able to check cooperation consistency for a given configuration.

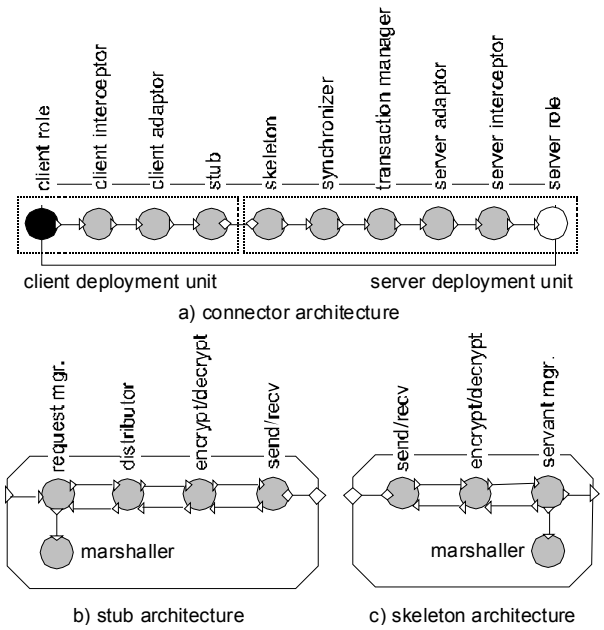

Fig. 2. Proposed procedure call connector architecture

3 Building real connectors

By analyzing several middleware designs and implementations [16, 26, 25, 14, 10, 27], we have identified a list of NFPs which can be addressed in middleware (see Appendix). In this section, we suggest a connector architecture for Procedure call and Messaging communication styles reflecting a relevant spectrum of the identified NFPs. The other two communication styles (Streaming and Blackboard) are omitted due to space constraints. Their detailed description can be found in [7]. Similar to the example from Section 2, we reflect a single NFP in 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 Figure 2a. 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 Figure 1.

In summary, the connector NFPs (listed in Appendix) 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 (thus affecting all clients). If no adaptation is necessary, the elements can be omitted.
- *Stub*. Together with a skeleton element, the 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 in Figure 2b. It consists of a request manager, which, using the attached marshaller, creates a request from the incoming call 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 the counterpart of the stub element. Again, its architecture can be primitive or compound (Figure 2c). The elements in the compound architecture are similar to those in the 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 Figure 3. 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 sending messages to attached components (as well as for receiving messages from them). The distributor deployment unit is in the middle of this logical star topology. 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.

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

- *Roles*. The sender role serves for 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.

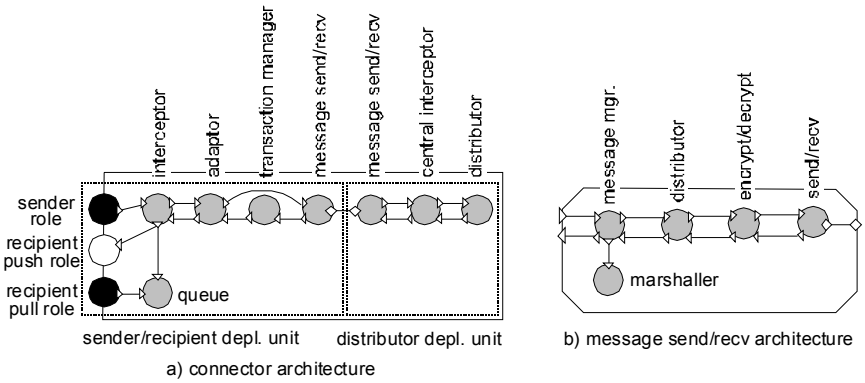

Fig. 3. Proposed messaging connector architecture

- *Queue*. Together with the pull role, this 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. They can be either primitive (directly implemented by underlying middleware) or compound (its typical architecture is on Figure 3b). 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 the *delivery strategy property* by implementing different policies for message routing (one recipient, all recipients, group address, etc.).

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) generation of a connector with respect to the middleware available on target deployment nodes.

1. *Component interactions.* 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 [21]. Medvidovic et al. in [12] go further and propose

additional connector types (adaptor, linkage, etc.); in our view, being at a lower abstraction level, these extra connector types are potential functional parts of the four basic connector types (e.g., adaptation may be associated with any of the communication styles in our approach).

2. *NFPs*. We have chosen an 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. Our approach is similar to reflective middleware [5, 4, 9, 19], which is also composed of smaller parts; however, middleware-dependent code is inherently present in components, making them less portable. Our work is also related to [8] and [2]. The former 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 communication mediators, and relies on having the source code of both the application and the middleware available. The latter describes how to incorporate connectors into the ArchJava language [1]. It allows to reflect NFPs in connectors too, but at the cost that the whole connector code has to be completely coded by hand.
3. *Automatic generation.* We have automated the process of element selection and adaptation, and connector instantiation; however we plan to automate, to a certain degree, the design process of a connector architecture. The idea of generating middleware-related code in an automatic way is employed in ProActive [15], where stubs and skeletons are generated at run-time. However, ProActive is bound only to Java not considering other communication styles than RPC and not addressing NFPs.

Prototype implementation: As a proof of the concept, a prototype implementation of a connector generator for the SOFA component model [17] is available, implementing three of the four considered communication styles (procedure call, messaging, and datastream) [22]. Designed as an open system employing plugins for easy modification it consists of two parts. The first one, written in Prolog, takes care of selecting a particular architecture and element implementations with respect to given communication style and NFPs (as described in Section 2) and produces a connector configuration. The second part, written in Java, takes the produced connector configuration and adapts its element implementations to the actual interfaces and, later on, instantiates a connector.

5 Conclusion and future intentions

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 [3] 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 future intentions include addressing the open issue of finding a metric allowing to select the "best" configuration when more than one configuration, meeting the desired factors, is found .

Acknowledgments

We would like to give special credit to Lubomir Bulej, a coauthor of [6]. Special thanks go to Petr Tuma, Jiri Adamek and other colleagues in our group for their valuable comments. Also, Petr Hnetynka deserves special credit for incorporating the implementation of connector generator into the SOFA framework. This work was partially supported by the Grant Agency of the Czech Republic (project number 201/03/0911); the results will be employed in the OSMOSE/ITEA project.

References

[1] J. Aldrich, C. Chambers, D. Notkin, "ArchJava: Connecting Software Architecture to Implementation", in Proceedings of ICSE 2002, May 2002

[2] J. Aldrich, V. Sazawal, D. Notkin, C. Chambers, "Language Support for Connector Abstractions", in Proceedings of ECOOP 2003, Darmstadt, Germany, July 2003

[3] D. Balek, F. Plasil, "Software Connectors and Their Role in Component Deployment", in Proceedings of DAIS'01, Krakow, Kluwer, Sept. 2001

[4] G. Blair, et al., "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, Hudson River Valley (NY), USA. Springer Verlag, LNCS, April 2000

[5] G. Blair, 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

[6] L. Bulej, T. Bures, "A Connector Model Suitable for Automatic Generation of Connectors", Tech. Report No. 2003/1, Dep. of SW Engineering, Charles University, Prague, 2003

[7] T. Bures, F. Plasil, "Composing connectors of elements", Tech. Report No. 2003/3, Dep. of SW Engineering, Charles University, Prague, 2003

[8] J. K. Cross, D. C. Schmidt, "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

[9] B. Dumant, F. Horn, F. Dang Tran, J.-B. Stefani, "Jonathan: an Open Distributed Processing Environment in Java", 1998

[10] Helix Community, "Helix DNA", www.helixcommunity.org

[11] N. Medvidovic, N. R. Mehta, "Distilling Software Architecture Primitives form Architectural Styles", TR UCSCSE 2002-509

[12] N. Medvidovic, N. R. Mehta, S. Phadke, "Towards a Taxonomy of Software Connectors", in Proceedings of the International Conference on Software Engineering, Limerick, Ireland, June 2000

[13] N. Medvidovic, P. Oreizy, R. N. Taylor, "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

[14] ObjectWeb Consortium, “JORAM: Java Open Reliable Asynchronous Messaging”, www.objectweb.org/joram
[15] ObjectWeb Consortium, “ProActive manual version 1.0.1”, January 2003
[16] OMG formal/02-12-06, “The Common Object Request Broker Architecture: Core Specification, v3.0”, December 2002
[17] F. Plasil, D. Balek, R. Janecek, “SOFA/DCUP: Architecture for Component Trading and Dynamic Updating”, in Proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998
[18] F. Plasil, S. Visnovsky: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov. 2002
[19] J. Putman, D. Hybertson, “Interaction Framework for Interoperability and Behavioral Analyses”, ECOOP Workshop on Object Interoperability, 2000
[20] M. Shaw, R. DeLine, G. Zalesnik, “Abstractions and Implementations for Architectural Connections”, in Proceedings of the 3rd International Conference on Configurable Distributed Systems, May 1996
[21] M. Shaw, D. Garlan, Software Architecture, Prentice Hall, 1996
[22] The SOFA Project, http://sofa.debian-sf.objectweb.org/
[23] Sun Microsystems, Inc., “ Enterprise JavaBeans Specification 2.0, Final Release”, August 2001
[24] Sun Microsystems, Inc., “Java IDL”, http://java.sun.com/j2se/1.4.1/docs/guide/idl/index.html
[25] Sun Microsystems, Inc., “Java Message Service”, April 2002
[26] Sun Microsystems, Inc., “Java Remote Method Invocation Specification – Java 2 SDK, v1.4.1”, 2002
[27] Sun Microsystems, Inc., “JavaSpaces Service Specification”, April 2002

Appendix

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)

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.

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.

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.
Distributed	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.