Higher-Order Architectural Connectors

ANTÓNIA LOPES
University of Lisbon, Portugal
MICHEL WERMELINGER
New University of Lisbon, Portugal
and
JOSÉ LUIZ FIADEIRO
University of Leicester, United Kingdom

We develop a notion of higher-order connector towards supporting the systematic construction of architectural connectors for software design. A higher-order connector takes connectors as parameters and allows for services such as security protocols and fault-tolerance mechanisms to be superposed over the interactions that are handled by the connectors passed as actual arguments. The notion is first illustrated over CommUnity, a parallel program design language that we have been using for formalizing aspects of architectural design. A formal, algebraic semantics is then presented which is independent of any Architectural Description Language. Finally, we discuss how our results can impact software design methods and tools.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; D.2.11 [**Software Engineering**]: Software Architectures—*languages (interconnection); patterns*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques*

General Terms: Design, Theory, Languages

1. INTRODUCTION

Although components have always been considered to be the fundamental building blocks of software systems, the ways the components of a system interact are determinant for establishing the global system properties, that is, the properties that emerge from the way the individual components are

This research was partially supported by Fundação para a Ciência e Technologia through project POSI/32717/00 (FAST—Formal Approach to Software Architecture).
Authors' addresses: A. Lopes, Department of Informatics, Faculty of Sciences, University of Lisbon, 1749-016 Lisboa, Portugal; email: mal@di.fc.ul.pt; M. Wermelinger, Departamento de Informática, Univ. Nova de Lisboa, 2829-516 Caparica, Portugal; email: mw@di.fct.unl.pt; J. L. Fiadeiro, Department of Computer Science, University of Leicester, University Road, Leicester LE1 7RH, United Kingdom; email: jose@fiadeiro.org.

interconnected. Hence, component interactions have been promoted to first-class design entities as well, and architectural connectors have emerged as a powerful tool for supporting the design of these interactions [Perry and Wolf 1992; Shaw 1993].

Although the use of connectors is widely accepted at the conceptual level, their explicit representation at the linguistic level is not always felt to be necessary. For example, the Darwin [Magee et al. 1999] Architecture Description Language (ADL) does not include connectors. However, we feel that distinct conceptual entities should correspond to distinct linguistic entities, so that they can truly become first-class and be manipulated as such. In fact, as argued in [Mehta et al. 2000], the current level of support that ADLs provide for connector building is still far from the one awarded to components. For instance, although a considerable amount of work can be found on several aspects of connectors [Shaw et al. 1995; Allen and Garlan 1997; Bass et al. 1998; Spitznagel and Garlan 2001; Hirsch et al. 1999; Mehta et al. 2000], further steps are still necessary to achieve a systematic way of constructing new connectors from existing ones. Yet, the ability to manipulate connectors in a systematic and controlled way is essential for promoting reuse and incremental development, and to make it easier to address complex interactions.

At an architecture level of design, component interactions can be very simple (for instance a shared channel), but they can be very complex as well (e.g., database-access and networking protocols). Hence, it is very important that we have mechanisms for designing connectors in an incremental and compositional way, as well as principled ways of extending existing ones, promoting reuse. This is especially important for connectors that are used at lower levels of design because it is well known that the implementation of complex protocols is a very difficult and error prone part of system development. Furthermore, as argued in [Denker et al. 1999], modularising the different kinds of services involved in interaction protocols has other advantages. It prevents interactions from being “hard-wired” across different components and makes it easier to evolve systems (possibly at run-time), because service modules may be added only when necessary, hence preventing performance penalties when such complex interactions are not required.

In this article, we take a step towards this goal by proposing a specification mechanism that allows independent aspects such as compression, fault-tolerance, security, monitoring, etc., to be specified separately, and then composed and integrated with existing connectors. In this way, it becomes possible to benefit from the multiple combinations of different services, ideally chosen *à la carte*. We develop a notion of higher-order connector—a connector that takes a connector as a parameter—through which it is possible to describe the *superposition* of certain capabilities over the form of coordination that is handled by the connector that is passed as an actual argument. In this way, we obtain “connector stacks” that are similar in spirit to meta-object towers [Denker et al. 1999] and to network protocol stacks [O’Malley and Peterson 1992], where each stack layer handles a given communication or interaction protocol.

More concretely, we define a higher-order connector through a (formal) parameter declaration and a body connector that models the nature of the service that is superposed on instantiation of the formal parameter. For instance, the monitoring of messages in unidirectional communication can be captured by a higher-order connector with a parameter *Unidirectional-comm* that specifies the kind of connectors to which the service can be applied, and a body connector that describes how an actual parameter is adapted in order to transmit certain messages to a monitoring component.

A higher-order connector can be applied to any connector that instantiates the formal parameter, giving rise to a connector with the new capabilities. In the case of monitoring, the higher-order connector can be applied, for instance, to a connector that models asynchronous communication between a sender and a receiver. Higher-order connectors can also be applied to other high-order connectors. In this case, the result is also a higher-order connector. This later form of application of higher-order connectors can be defined as a parametric instantiation (the instantiation of a parameter with a parameterized entity) and models a noncommutative composition of high-order connectors through which their capabilities are superposed.

The idea of defining higher-order connectors as operators through which new connectors can be built from old ones was proposed by Garlan [1998], arguing that, conceptually, operations on connectors allow one to factor out common properties for reuse and to better understand the relationships between different connector types. The notation and semantics of such connector operators were recognised to be among the main issues to be dealt with and were later developed by Spitznagel and Garlan [2001] in the context of the ADL Wright.

Whereas Spitznagel and Garlan define moderately complex and specialized operations, our first attempt at systematic connector construction provided three generic and very simple connector transformations [Fiadeiro et al. 2003]. Our second approach added the notion of higher-order connector, first presented in [Wermelinger et al. 2000] in a preliminary, informal, and ADL-specific way. In this article, capitalizing on our previous work on the formal underpinning of connectors [Fiadeiro et al. 2003] and using the well-known mathematical "technology" of parameterisation [Goguen 1996] we present a formalization of higher-order connectors and their composition that is not specific to any ADL. For this purpose, we use the categorical semantics of connectors presented in [Fiadeiro et al. 2003]. Therein, we establish the semantics of architectural connectors, in the style defined by Allen and Garlan [1997], independently of specific choices of design languages and behavioural models. We also make use of the characterisation of the minimal set of features that constitutes an ADL, presented in [Fiadeiro et al. 2003]. As a result, the reader will be able to understand and verify the extent up to which his/her favourite ADL can support the higher-order mechanisms that we are going to present, and to extend it if necessary and desired according to the semantics that we propose.

The article is organized as follows: Section 2 illustrates, through an example, the key ideas of the notion of higher-order connector we wish to put forward. It shows, in a communication service that involves compression of messages, how the communication service can be separated from the compression service. In Section 3, following a categorical approach, we present a parallel program design language, inspired by Unity [Chandy and Misra 1988] and Interacting

Processes [Francez and Forman 1996], which is nearer to the abstractions used by conventional programming languages than the process calculi used by others [Magee et al. 1999; Allen and Garlan 1997; Spitznagel and Garlan 2001]. We then present the way higher-order connectors can be defined over this setting and, in Section 4, we show that the definition is not specific to the design formalism we have adopted in Section 3, as long as the chosen formalism satisfies some structural properties. In Section 5, we present a notion of composition of higher-order connectors. We finish by including a comparison with related work (Section 6) and some concluding remarks in which we discuss how our results can impact software design methods and tools.

2. MOTIVATION

Software Architecture has put forward connectors as first-class entities for modelling interactions between systems components. According to Allen and Garlan [1997], a connector is defined by a set of *roles* and a *glue* specification. Each role describes the behaviour that is expected of each of the interacting parts, that is, it determines the obligations that they have to fulfil to become instances of the roles. The glue describes how the activities of the role instances are coordinated.

For instance, asynchronous communication through a bounded channel can be represented by a connector *Async* with two roles—*sender* and *receiver*. The glue of *Async* is a bounded *buffer* with a FIFO discipline that prevents the sender from sending a new message when there is no space and prevents the receiver from reading a new message when there are no messages. (See Figure 1.)

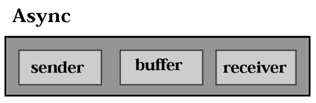

Fig. 1.

The use of a connector in the construction of a particular system consists in the instantiation of its roles with specific components of the system. The instantiation of a role with a component is possible if and only if the component fulfils the obligations the role determines. Therefore, instantiation corresponds, typically, to a form of refinement.

Let us suppose that in a given system, two components, say A and B, are connected through *Async*, A playing the role of *sender* and B playing the role of *receiver*. (See Figure 2.)

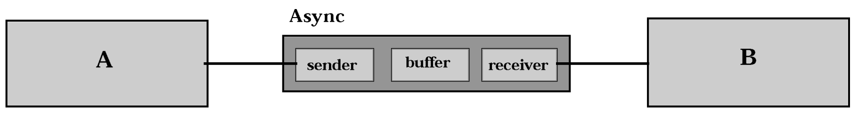

Fig. 2.

Suppose that, for some reason, the information transmitted from A to B must be compressed. Clearly, we may develop, from scratch, a new connector *C-Async* with the required functionality, possibly keeping the same roles *sender* and *receiver* but replacing the glue with a new one—*c-async*, and then replace *Async* by *C-Async* in the instantiation to *A* and *B*. (See Figure 3.)

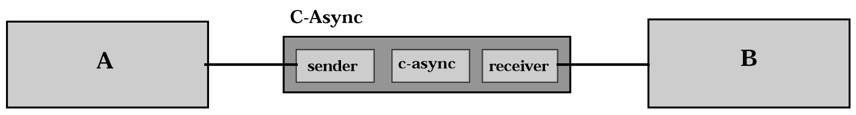

Fig. 3.

However, it would be certainly better if we could obtain the new connector by simply installing a compress/decompress service over the existing communication service as modeled through *Async*. The idea is to modify *Async* in a way that messages are compressed for transmission without intruding on the original connection, that is, without "rewiring" the connections to the buffer. Hence, in the resulting connector, the outgoing messages should be compressed before they are put into the buffer and decompressed when they are removed from the buffer, before being delivered to the receiver. It is not difficult to realise that this form of coordination of the sender and receiver activities, embodied by the glue *C-Async*, can be obtained by instantiating the sender role of *Async* with a component *comp* that compresses messages before it transmits them, and by instantiating the *receiver* role with a component *decomp* that decompresses the messages it receives. (See Figure 4.)

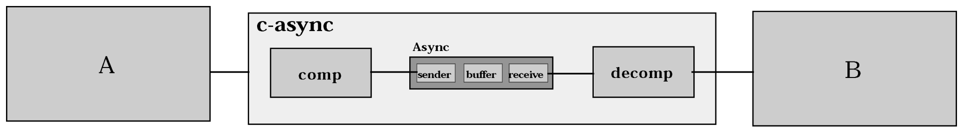

Fig. 4.

In this way, in the resulting protocol *C-Async*, with the same roles as before and *c-async* as the new glue, a message sent by *sender* is first compressed by *comp*, which then uses *Async* to transmit it to *decomp*. Finally, *decomp* decompresses the message and forwards the result to *receiver*.

It is important to realize that the procedure we described for installing the compress/decompress service over *Async* can be applied to other connectors. In fact, it is possible to give a parameterized description of the compress/decompress service such that the installation of the service over a given connector can be obtained by a suitable instantiation of the parameter.

In this article, our aim is to introduce such parameterized entities. As we shall see, these are connectors with a distinguished formal parameter part and, hence, are called higher-order connectors. In the example of the compression, this means that we shall define a higher-order connector *Compression(Uni_comm)* whose formal parameter *Uni_comm* is a connector that models a generic unidirectional communication protocol. This formal parameter can be instantiated by several different connectors, in particular by the asynchronous message passing connector that we have been discussing.

3. HIGHER-ORDER CONNECTORS IN COMMUNITY

We use the parallel program design language CommUnity in order to make the ideas put forward in the previous section more concrete, and also to motivate the general categorical semantics of higher-order connectors to be given in Section 4 as an extension to our previous work on formalizing architectural modelling [Lopes and Fiadeiro 1999; Wermelinger et al. 2000; Fiadeiro et al. 2003].

CommUnity is a Unity-like design language that was initially presented in Fiadeiro and Maibaum [1997] to show how programs fit into Goguen's categorical approach to General Systems Theory [Goguen 1973]. Since then, the language and the design framework have been extended in order to provide a formal platform for the architectural design of open, reactive and reconfigurable systems.

3.1 Component Design

We start by presenting an example of a CommUnity design—*help*. This design models a box consisting of a button, a sensor and a light. Its purpose is to allow a patient to request help in case of medical emergency, with the transmission of the current value of the sensor (e.g., pulse). Pressing the button, which is modeled by the execution of *hreq*, turns on the light, which is modeled by channel *off* becoming false. The light is turned off when the help request is acknowledged. After the button is pressed, the current value of the sensor is read, which is modeled by the execution of the private action *read*, and made available for transmission in the output channel *data*. The private channel *rd* is used to distinguish between states in which the value in *data* is the value to be transmitted or not.

```
design   help is
in       sensor:int
out      data:int, off: bool
prv      rd: bool
do       hreq: off → off:=false
[]  prv  read: ¬rd ∧ ¬off→data:=sensor‖rd:=true
[]       hack: rd → rd:=false‖off:=true
```

A CommUnity component design is of the form

```
design  P is
out     out(V)
in      in(V)
prv     prv(V)
do      []_{g∈sh(Γ)}          g[D(g)]: L(g), U(g) → R(g)
        []_{g∈prv(Γ)}   prv   g[D(g)]: L(g), U(g) → R(g)
```

where

—V is the set of *channels*. Channels can be declared as *input*, *output* or *private*. Input channels are used for reading data from the environment of the component. The component has no control on the values that are made available in such channels. Moreover, reading a value from an input channel does not "consume" it: the value remains available until the environment decides to replace it. Output and private channels are controlled locally by the component, that is, the values that, at any given moment, are available on these channels cannot be modified by the environment. Output channels allow the environment to read data produced by the component. Private channels support internal activity that does not involve the environment in any way. We use $loc(V)$ to denote the union $prv(V) \cup out(V)$, that is, the set of local channels. Each channel v is typed with a sort $sort(v) \in S$.

—by Γ we denote the set of *action names*. The named actions can be declared either as *private* or *shared* (for simplicity, we only declare which actions are private). Private actions represent internal computations in the sense that their execution is uniquely under the control of the component. Shared actions represent possible interactions between the component and the environment, meaning that their execution is also under the control of the environment. The significance of naming actions will become obvious below; the idea is to provide points of rendezvous at which components can synchronize.

For each action name g, the following attributes are defined:

—$D(g)$ is a subset of $loc(V)$ consisting of the local channels that can be effected by executions of the action named by g. This is what is sometimes called the *write frame* of g. For simplicity, we omit the explicit reference to the write frame when $R(g)$ is a conditional multiple assignment (see below), in which case $D(g)$ can be inferred from the assignments. Given a local channel v, we will also denote by $D(v)$ the set of actions g such that $v \in D(g)$.

—$L(g)$ and $U(g)$ are two conditions such that $U(g) \supset L(g)$. These conditions establish an interval in which the enabling condition of any guarded command that implements g must lie. The condition $L(g)$ is a lower bound for enabledness in the sense that it is implied by the enabling condition. Therefore, its negation establishes a *blocking* condition. On the other hand, $U(g)$ is an upper bound in the sense that it implies the enabling condition, therefore establishing a *progress* condition. Hence, the enabling condition is fully determined only if $L(g)$ and $U(g)$ are equivalent, in which case we write only one condition.

—$R(g)$ is a condition on V and $D(g)'$ where by $D(g)'$ we denote the set of primed local channels from the write frame of g. As usual, these primed channels account for references to the values that the channels take after the execution of the action. These conditions are usually a conjunction of implications of the form $pre \supset pos$ where pre does not involve primed channels. They correspond to pre/post-condition specifications in the sense of Hoare. When $R(g)$ is such that the primed version of each local channel in the write frame of g is fully determined, we obtain a conditional multiple assignment, in which case we use the notation that is normally found in programming languages. When the write frame $D(g)$ is empty, $R(g)$ is tautological, which we denote by *skip*.

Notice that CommUnity supports several mechanisms for underspecification—actions may be underspecified in the sense that their enabling conditions may not be fully determined (subject to refinement by reducing the interval established by L and U) and their effects on the channels may also not be fully determined.

When, for every $g \in \Gamma$, $L(g)$ and $U(g)$ coincide, and the relation $R(g)$ defines a conditional multiple assignment then the design is called a *program*. The behavior of a program is as follows: At each execution step, any of the actions whose enabling condition holds of the current state can be selected. When selected, the assignments associated with the actions are executed atomically as a transaction. Private actions that are infinitely often enabled are guaranteed to be selected infinitely often (see Lopes and Fiadeiro [1999] for a model-theoretic semantics of CommUnity). A program with input channels is *open* in the sense that it needs to be connected to other components of the system to read data. We explain how such connections can be established in later sections.

As a language, CommUnity is independent of the actual data types that are used and, hence, we have assumed that there are predefined sorts and functions given by a fixed algebraic specification $\Xi = <<S, \Omega>, \Phi>$ where S is a set of sorts, Ω is a set of operations, and Φ is a set of first-order axioms specifying the functionality of the operations [Ehrig and Mahr 1985]. For the purposes of examples in this article, we consider an algebraic signature containing basic data types such as Booleans (*bool*), integers (*int*), and queues (*queue*(–, –)), with the usual operations.

Formally, CommUnity designs can be defined as follows:
A signature is a tuple $<V, \Gamma, tv, ta, D>$ *where*

—V *is an S-indexed family of mutually disjoint finite sets*,
—Γ *is a finite set,*
—$tv : V \rightarrow \{out, in, prv\}$ *is a total function,*
—$ta : \Gamma \rightarrow \{sh, prv\}$ *is a total function,*
—$D : \Gamma \rightarrow 2^{loc(V)}$ *is a total function.*

A design is a pair $<\theta, \Delta>$ *where* $\theta = <V, \Gamma, tv, ta, D>$ *is a signature and* Δ*, the body of the design, is a tuple* $<R, L, U>$ *where:*

—R *assigns to every action* $g \in \Gamma$*, a proposition over* $V \cup D(g)'$*,*
—L *and* U *assign a proposition over* V *to every action* $g \in \Gamma$.

In order to support higher levels of design, a CommUnity design may also be parameterized by an algebraic specification indicated after the name of the component (see examples in the next section). This parameter is instantiated at configuration time, that is, when a specific component needs to be included in the configuration of the system being built, or as part of the reconfiguration of an existing system.

3.2 Connectors

In CommUnity, a complex system is described as the interconnection of a number of interacting component designs by defining a configuration. A connector in CommUnity consists of a set of roles, a glue specification, and a configuration involving these designs.

Let us consider the connector *Async*[$t + K$] representing asynchronous communication of values of type t through a bounded buffer with capacity K. This connector has two roles—*sender*[t] and *receiver*[t]—defining the behavior required of the components to which the connector can be applied. For the *sender*, we require that it does not produce another message before the previous one has been processed. After producing a message, the *sender* should expect an acknowledgment to produce a new message. For the *receiver*, we simply require that it has an action that models the reception of a message.

```
design    sender[t] is               design   receiver[t] is
out       o:t                        in       i:t
prv       rd:bool                    do       rec: true,false→skip
do   prv  prod[o,rd]: ¬rd,false→
          rd:=true
[]        send[rd]: rd,false→
          rd:=false
```

In order to leave unspecified when and how many messages the sender (receiver) will send (receive), and in which situations the sender will produce a new message, the progress guards of these actions are all false. Because progress conditions establish an upper bound for enabledness, if the progress condition of an action is false then its enabling condition can be as strong as we wish. Furthermore, by including o in the write frame D(*prod*) but not including any condition on how it can be effected, we avoid committing to a particular discipline of production.

Both designs are parameterized by the algebraic specification $<<\{t\},\emptyset>, \emptyset>$ that consists just of a sort (without operations nor axioms), which we denoted simply by t. When the connector needs to be used in the configuration of a specific system, the sort t is instantiated with whatever sort is appropriate.

The glue of *Async*[$t + K$] is a bounded buffer with a FIFO discipline that prevents the sender from sending a new message when there is no space and prevents the receiver from reading a new message when no new messages have been sent. In CommUnity, this buffer can be designed as follows:

```
design    buffer [Ξ^b] is
in        i:t
out       o:t
prv       rd:bool; q:queue(K,t)
do        put: ¬full(q)→ q:=enqueue(i,q)
[] prv    next: ¬empty(q)∧¬rd → o:=head(q)||q:=tail(q)||rd:=true
[]        get: rd → rd:=false
```

This buffer can store, through the action *put*, messages of sort t, received from the environment through the input channel i, as long there is space for them. It can also make stored messages available to the environment through the output channel *o* and the action *next*. Naturally, this activity is possible only when there are messages in store and the current message in *o* has already been read by the environment, which is modeled by the action *get* and the private channel *rd*.

The type of messages as well as the capacity of the buffer are part of the parameter specification Ξ^b defined below, where we use Ξ^{nat} to represent the subspecification of Ξ that is concerned with the specification of natural numbers. We also use $t + K$ to denote Ξ^b.

spec Ξ^{b} is Ξ^{nat}+
sorts t
ops   K: ->nat

It remains to define in which way the roles and the glue are connected. The model of interaction between components in CommUnity is based on action synchronisation and the interconnection of input channels of a component with output channels of other components. Although these are common forms of interaction, CommUnity requires interaction between components—name bindings—to be made explicit in the systems configurations. Name bindings are established as relationships between the signatures of the corresponding components and are defined with the help of additional signatures (representing the interaction points) and signature maps (morphisms).

For instance, in order to establish that messages from a *sender* component are sent (to a receiver) through a bounded channel, we consider the following configuration (see Figure 5)

$$\begin{tikzcd}
 & \texttt{cable[t]} \arrow[dl] \arrow[dr] \arrow[d, phantom, "\substack{o \leftarrow \bullet \rightarrow i \\ send \rightarrow \bullet \leftarrow put}"] & \\
\texttt{sig(}\textbf{sender}\texttt{[t])} & & \texttt{sig(}\textbf{buffer}\texttt{[t+K])}
\end{tikzcd}$$

Fig. 5.

where *cable*[t] is a signature that consists of an input channel of sort t and a shared action. The names of this channel and of this action are not relevant: they are only placeholders used to define the name bindings, and hence, we used $\bullet$ for both.

In this configuration, the input channel of *cable* is mapped to the output channel *o* of the *sender* and to the input channel i of *buffer*. This establishes an I/O interconnection between the *sender* and the *buffer*. Moreover, the actions *send* of *sender* and *put* of *buffer* are mapped to the shared action of *cable*. This defines that *sender* and *buffer* must synchronize each time either of them wants to perform the corresponding action.

The signature morphisms involved in the configurations are defined as follows:

A morphism $\sigma : \theta_1 \rightarrow \theta_2$ *between signatures* $\theta_1 = {<}V_1, \Gamma_1, tv_1, ta_1, D_1{>}$ *and* $\theta_2 = {<}V_2, \Gamma_2, tv_2, ta_2, D_2{>}$ *is a pair* ${<}\sigma_{ch}, \sigma_{ac}{>}$ *where*

—$\sigma_{ch} : V_1 \rightarrow V_2$ *is a total function satisfying:*
  —$sort_2(\sigma_{ch}(v)) = sort_1(v)$ *for every* $v \in V_1$
  —$\sigma_{ch}(o) \in out(V_2)$ *for every* $o \in out(V_1)$
  —$\sigma_{ch}(i) \in out(V_2) \cup in(V_2)$ *for every* $i \in in(V_1)$
  —$\sigma_{ch}(p) \in prv(V_2)$ *for every* $p \in prv(V_1)$
—$\sigma_{ac} : \Gamma_2 \rightarrow \Gamma_1$ *is a partial mapping satisfying for every* $g \in \Gamma_2$ *s.t.* $\sigma_{ac}(g)$ *is defined:*
  —*if* $g \in sh(\Gamma_2)$ *then* $\sigma_{ac}(g) \in sh(\Gamma_1)$
  —*if* $g \in prv(\Gamma_2)$ *then* $\sigma_{ac}(g) \in prv(\Gamma_1)$
  —$\sigma_{ch}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$
  —σ_{ac} *is total on* $D_2(\sigma_{ch}(v))$ *and* $\sigma_{ac}(D_2(\sigma_{ch}(v))) \subseteq D_1(v)$ *for every* $v \in loc(V_1)$

Signature morphisms represent more than the projections that arise from name bindings as illustrated above. A morphism σ from θ_1 to θ_2 is intended to support the identification of a way in which a component with signature θ_1 is embedded in a larger system with signature θ_2. This justifies the various constructions and constraints in the definition.

The function σ_{ch} identifies for each channel of the component the corresponding channel of the system. The partial mapping σ_{ac} identifies the action of the component that is involved in each action of the system, if ever. The fact that the two mappings go in opposite directions is justified as follows: Actions of the system constitute synchronization sets of actions of the components. Because not every component is necessarily involved in every action of the system, the action mapping is partial. On the other hand, because each action of the component may participate in more than one synchronisation set, but each synchronization set cannot induce internal synchronisations within the components, the relationship between the actions of the system and the actions of every component is functional from the former to the latter.

The other constraints are concerned with typing. Sorts of channels have to be preserved but, in terms of their classification, input channels of a component may become output channels of the system. This is because, in the absence of other constraints, the result of interconnecting an input channel of a component with an output channel of another component in the system is an output channel of the system. Mechanisms for internalizing communication can be applied but they are not the default in a configuration. The last two conditions on write frames implies that actions of the system in which a component is not involved cannot have local channels of the component in its write frame. That is, change within a component is completely encapsulated in the structure of actions defined for the component.

In the case of configurations involving components parameterized with data type specifications, the signature morphisms must satisfy an additional property:

A morphism $\sigma : \theta_1[\Pi_1] \rightarrow \theta_2[\Pi_2]$ *between parameterized signatures is a signature morphism* $\sigma : \theta_1 \rightarrow \theta_2$ *for which* Π_1 *is a subspecification of* Π_2*, that is,* σ *defines an inclusion of sorts, operations and theorems* [*Ehrig and Mahr* 1985].

Let us consider again the connector $Async[t + K]$ that we have been discussing. The configuration depicted below completes its definition, establishing how the roles and the glue are connected. (See Figure 6.)

$$\begin{tikzcd}
 & \texttt{cable[t]} \arrow[dl] \arrow[dr] \arrow[d, phantom, "\begin{matrix}\texttt{o}\leftarrow\bullet\rightarrow\texttt{i}\\ \texttt{send}\rightarrow\bullet\leftarrow\texttt{put}\end{matrix}"] & & \texttt{cable[t]} \arrow[dl] \arrow[dr] \arrow[d, phantom, "\begin{matrix}\texttt{o}\leftarrow\bullet\rightarrow\texttt{i}\\ \texttt{get}\rightarrow\bullet\leftarrow\texttt{rec}\end{matrix}"] & \\
\texttt{sig(}\textbf{sender}\texttt{[t])} & & \texttt{sig(}\textbf{buffer}\texttt{[t+K])} & & \texttt{sig(}\textbf{receiver}\texttt{[t])}
\end{tikzcd}$$

Fig. 6.

As explained previously, the left-hand side morphisms define that *sender* and *buffer* must synchronise on actions *send* and *put*, and establish the interconnection of the output channel *o* of *sender* with the input channel *i* of *buffer*. On the other hand, the right-hand side morphisms define that *buffer* and *receiver* must synchronise on actions *get* and *rec* and establish the interconnection of the output channel *o* of *buffer* with the input channel *i* of *receiver*.

Not every diagram of signatures makes sense as a configuration. There are restrictions on the way that we can interconnect components that are not captured by the notion of morphism alone but apply to the whole diagram. The two following rules express the restrictions on diagrams that make them well-formed configurations:

—An output channel of a component cannot be connected (directly or indirectly through input channels) with output channels of the same or other components.

—Private channels and private actions cannot be involved in the connections.

It is important to notice that the second rule establishes the configuration semantics of private actions and channels. It supports the intuitive semantics we gave in Section 2.1, namely that private channels cannot be read by the environment and that the execution of private actions is uniquely under the control of the component.

Rather than using diagrams involving signatures and signature morphisms, a "box and line" notation may be adopted instead. For instance, the asynchronous communication defined above could be described as follows: (see Figure 7.)

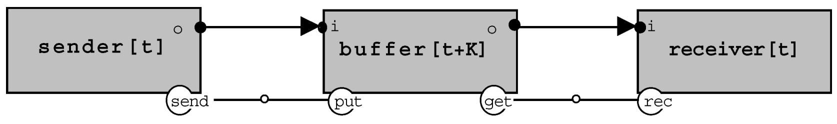

Fig. 7.

In this notation, the name bindings are still explicit but are expressed in terms of arcs that connect channels and actions directly. These configurations can be easily translated into categorical diagrams involving signatures and signature morphisms.

The semantics of connectors, and of configuration diagrams in general, relies on an extension of the notion of signature morphism that allows us to establish relationships between designs. Design morphisms capture relationships between components and the systems that they are part-of. They can be seen to provide a formalisation for a notion of superposition that is similar to those that have been used for parallel program design [Chandy and Misra 1988; Katz 1993].

A superposition morphism $\sigma : P_1 \rightarrow P_2$ of designs $P_1 = <\theta_1, \Delta_1>$ and $P_2 = <\theta_2, \Delta_2>$, consists of a signature morphism $\sigma : \theta_1 \rightarrow \theta_2$ such that, for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined:

(1) $\Phi \vdash (R_2(g) \supset \underline{\sigma}(R_1(\sigma_{ac}(g))))$;
(2) $\Phi \vdash (L_2(g) \supset \underline{\sigma}(L_1(\sigma_{ac}(g))))$;
(3) $\Phi \vdash (U_2(g) \supset \underline{\sigma}(U_1(\sigma_{ac}(g))))$;

*where Φ is the axiomatization of the data type specification, $\vdash$ denotes validity in the first-order sense, and $\underline{\sigma}$ is the extension of σ to the language of expressions and conditions. Designs and their morphisms constitute a category **c-DSGN.***

A morphism $\sigma : P_1 \rightarrow P_2$ identifies a way in which P_1 is "augmented" to become P_2 so that P_2 can be considered as having been obtained from P_1 through the superposition of additional behavior, namely the interconnection of one or more components. The conditions on the actions require that the computations performed by the system reflect the interconnections established between its components. Condition (1) reflects the fact that the effects of the actions of the components can only be preserved or made more deterministic in the system. This is because the other components in the system cannot interfere with the transformations that the actions of a given component make on its state, except possibly by removing some of the underspecification present in the component design.

Conditions (2) and (3) allow the bounds that the component design specifies for the enabling of the action to be strengthened but not weakened. Strengthening of the lower bound reflects the fact that all the components that participate in the execution of a joint action have to give their permission for the action to occur. On the other hand, it is clear that progress for a joint action can only be guaranteed when all the designs of the components involved can locally guarantee so.

Let us consider that we have established interactions between component designs $P_1, \ldots, P_n$ at the level of their signatures through a diagram $\boldsymbol{D}$. Such a diagram can be trivially lifted to a diagram $\boldsymbol{D}'$ of designs and superposition morphisms: the signature of each design is replaced by the design itself; every cable *cb* in $\boldsymbol{D}$ is replaced by ***dsgn**(cb),* the design with signature *cb* and tautological bounds for the enabledness of each action, and the least commitment as to the effects on the channels in the write frame of each action. More concretely,

***dsgn**(cb)* consists of, for each action name g in cb, the action

$$g : true, true \rightarrow true$$

Defined in this way, ***dsgn**(cb)* is a design that is "neutral" with respect to the establishment of superposition morphisms in the sense that every signature morphism $\sigma : cb \rightarrow$ ***sig**(P)* defines a superposition morphism $\sigma :$ ***dsgn**(cb)* $\rightarrow P$.

On the account of this transformation, every configuration can be transformed into a single design that represents the whole system by taking the colimit of the diagram $\boldsymbol{D}'$ in the category ***c-DSGN***. We now describe the intuitive meaning of the colimit and its construction.

Consider first the simplest case, the one in which there are no interactions. This means that any two channels of two designs are different, even if they have the same name. Therefore, the channels of the resulting design are the disjoint union of the components' channels. Concerning the actions, the parallel composition contains all possible combinations of actions that involve one at most one action from each component. This is because there is no restriction on their co-occurrence. More concretely, the actions of the resulting design are the tuples of actions of the components $a_1|\cdots|a_k$, containing at most one action of each component. In this way, the colimit provides not an interleaving but a concurrent semantics for parallel composition.

In the presence of interactions, the colimit "merges" the input channels identified with an output channel into that output channel, and each tuple $a_1|\cdots|a_k$ is retained iff, for every action a_i in the tuple, every action that is required to synchronize with a_i is also in the tuple.

With or without interactions, for each action $a_1|\cdots|a_k$, the bounds for enabledness are the conjunction of the bounds of all a_i, and its assignments are the union of the assignments of all a_i.

In the case of configurations involving parameterized designs, we need first to define the corresponding superposition morphisms.

A superposition morphism from a parameterized design $P_1(\Pi_1)$ to a parameterized design $P_2(\Pi_2)$ is a morphism of ***c-DSGN*** *from P_1 to P_2 for which Π_1 is a subspecification of* Π_2.

In this way, in the case of a configuration involving parameterized designs, the single design that represents the whole system is parameterized by the union of the data type specifications involved.

As an illustration of the transformation of a system configuration into a design that represents the whole system, we consider again the connector *Async*$[t + K]$. The lifting of the diagram presented previously, which establishes how the roles and the glue are connected at the level of their signatures, is (Figure 8):

$$\begin{tikzcd}
& \mathbf{dsgn}(\mathtt{cable[t]}) \arrow[dl] \arrow[dr, "\begin{smallmatrix}\mathtt{o}\leftarrow\bullet\rightarrow\mathtt{i}\\ \mathtt{send}\rightarrow\bullet\leftarrow\mathtt{put}\end{smallmatrix}"'] & & \mathbf{dsgn}(\mathtt{cable[t]}) \arrow[dl] \arrow[dr, "\begin{smallmatrix}\mathtt{o}\leftarrow\bullet\rightarrow\mathtt{i}\\ \mathtt{get}\rightarrow\bullet\leftarrow\mathtt{rec}\end{smallmatrix}"'] & \\
\mathbf{sender}\mathtt{[t]} & & \mathbf{buffer}\mathtt{[t+K]} & & \mathbf{receiver}\mathtt{[t]}
\end{tikzcd}$$

Fig. 8.

where ***dsgn**(cable[t])* is a design with an input channel of sort t and a shared action whose bounds for enabledness are *true*.

The colimit of this diagram returns the design given, up to an isomorphism, by the design below. This design models the parallel composition of *sender*, *receiver* and *buffer* with the restrictions defined by the given configuration diagram and defines the semantics of the connector.

```
design   async[t+K] is
out      o_s,o_b:t
prv      rd_s, rd_r:bool; q:queue(K,t)
do prv   prod[o_s,rd_s]: ¬rd_s, false→rd_s:=true
[]       send|put: ¬full(q)∧rd_s, false→q:=enqueue(o_s,q)‖rd_s:=false
[] prv   next: ¬empty(q)∧¬rd_r → o_b:=head(q)‖q:=tail(q)‖rd_r:=true
[]       rec|get: rd_r, false → rd_r:=false
```

This design provides the means for global properties of the protocol that it defines to be derived. For instance, using the logical formalism for reasoning about CommUnity designs defined in [Lopes and Fiadeiro 1999], it is possible to conclude that $async[t+K]$ has the following property expressed in a branching time temporal logic:

$$\boldsymbol{A}((send|put \wedge o_s = msg) \supset \boldsymbol{F}(o_b = msg \wedge < rec|get > true))$$

This sentence expresses that if a message *msg* is sent, eventually *msg* will be made available in the input channel of the receiver, ready to be received. In other words, at least a copy of each message is delivered. In the same way, it is possible to conclude that the correctness of the transmission/reception of data (in order message delivery) does not depend on the speed at which messages are produced and consumed.

3.3 Using Connectors in System Construction

The use of a connector in the construction of a particular system is achieved by the instantiation of its roles with specific components of the system. To model instantiation, we use a different kind of design morphism that ensures that the behavior specified by a role is satisfied by the instance. These morphisms correspond to a form of refinement and, hence, are called refinement morphisms.

A refinement morphism $\sigma : P_1 \to P_2$ *of designs* $P_1 = <\theta_1, \Delta_1>$ and $P_2 = <\theta_2, \Delta_2>$ *is a pair* $<\sigma_{ch}, \sigma_{ac}>$ *where*

—$\sigma_{ch} : V_1 \to Term(V_2)$ *is a total function mapping the channels of* P_1 *to the class of terms built from the channels of* P_2 *and the data type operations. This mapping is required to satisfy, for every* $v \in V_1, o \in out(V_1), i \in in(V_1), p \in prv(V_1)$:

—$sort_2(\sigma_{ch}(v)) = sort_1(v)$
—$\sigma_{ch}(o) \in out(V_2)$
—$\sigma_{ch}(i) \in in(V_2)$
—$\sigma_{ch}(p) \in Term(loc(V_2))$
—$\sigma_{ch} \downarrow (out(V_1) \cup in(V_1))$ *is injective*

—σ_{ac}: $\Gamma_2 \rightarrow \Gamma_1$ *is a partial mapping satisfying for every* $g \in \Gamma_2$ *s.t.* $\sigma_{ac}(g)$ *is defined*:

- —*if* $g \in sh(\Gamma_2)$, *then* $\sigma_{ac}(g) \in sh(\Gamma_1)$
- —*if* $g \in prv(\Gamma_2)$, *then* $\sigma_{ac}(g) \in prv(\Gamma_1)$
- —*if* $g \in sh(\Gamma_1)$, *then* $\sigma_{ac}^{-1}(g) \neq \emptyset$
- —$\sigma_{ch}(D_1(\sigma_{ac}(g))) \subseteq \underline{D}_2(g)$
- —σ_{ac} *is total on* $\underline{D}_2(\sigma_{ch}(v))$ *and* $\sigma_{ac}(\underline{D}_2(\sigma_{ch}(v))) \subseteq D_1(v)$ *for every* $v \in loc(V_1)$

and, furthermore,

—*for every* $g \in \Gamma_2$ *s.t.* $\sigma_{ac}(g)$ *is defined:*

(1) $\Phi \vdash (R_2(g) \supset \underline{\sigma}(R_1(\sigma_{ac}(g))))$;
(2) $\Phi \vdash (L_2(g) \supset \underline{\sigma}(L_1(\sigma_{ac}(g))))$;

—*for every* $g_1 \in \Gamma_1$,

(3) $\Phi \vdash (\underline{\sigma}(U_1(g_1)) \supset \bigvee_{\sigma_{ac}(g_2)=g_1} U_2(g_2))$

where $\underline{D}$ *is the extension of* D *to the language of expressions. Designs and their refinement morphisms constitute a category* ***r-DSGN.***

A refinement morphism is intended to support the identification of a way in which a design P_1 (its source) is refined by a more concrete design P_2 (its target).

The function σ_{ch} identifies for each input (respectively, output) channel of P_1 the corresponding input (respectively, output) channel of P_2. Notice that, contrarily to what happens with the component-of relationship, refinement does not change the border between the system and its environment and, hence, input channels can no longer be mapped to output channels. Moreover, refinement morphisms allow each private channel of P_1 to be expressed in terms of the local channels of P_2 through an expression. The evaluation of such an expression may involve some computation as captured through the use of operations from the underlying data types. Naturally, it is required that the sorts of channels be preserved.

The mapping σ_{ac} identifies for each action g of P_1, the set of actions of P_2 that implements g—given by $\sigma_{ac}^{-1}(g)$. This set is a menu of refinements for action g and can be empty for private actions. However, every action that models interaction between the component and its environment has to be implemented.

The actions for which σ_{ac} is left undefined (the new actions) and the channels that are not involved in $\sigma_{ch}(V_1)$ (the new channels) introduce more detail in the description of the component.

Conditions (2) and (3) require that the interval defined by the blocking and progress conditions of each action (in which the enabling condition of any guarded command that implements the action must lie) be preserved or reduced. This is intuitive because refinement, pointing in the direction of implementations, should reduce underspecification. This is also the reason why the effects of the actions of the more abstract design are required to be preserved or made more deterministic.

It is important to notice that, although refinement and superposition morphisms have some conditions in common, the two relationships are very different. As evidence of this notice, the fact that in CommUnity, as in other formalisms such as CSP [Hoare 1985], a design is not necessarily refined by a system of which it is a component.

The component design *help* defined previously is an example of a refinement of the design *sender*(*int*)—the result of the instantiation of t in *sender*[t] with the sort *int* of Ξ. It refines *sender*(*int*) through the refinement morphism η: *sender*(*int*) $\rightarrow$ *help* defined by

$$\eta_{ch}(o) = data, \eta_{ch}(rd) = rd$$
$$\eta_{ac}(read) = prod, \eta_{ac}(hack) = send.$$

In *help,* the production of messages to be sent is modeled by the action *read* and the messages are made available in the output channel *data*. The production of messages, that was left unspecified in *sender*, corresponds to the sensor readings.

This refinement can be represented graphically as depicted below. Notice that nonprivate channels and actions are placed on the boundary of the component and private ones inside. (See Figure 9.)

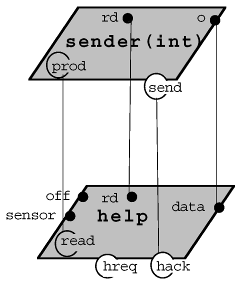

Fig. 9.

Let us suppose that *centre* is a design that models an assistance centre that refines *receiver*(*int*) through some refinement morphism κ. The connector *Async*(*int* + 10) can be used to interconnect *help* and *centre*. The resulting system—a system in which the *help* component sends the help requests to the assistance *centre* through a bounded channel with capacity for ten messages—can be represented as follows (see Figure 10):

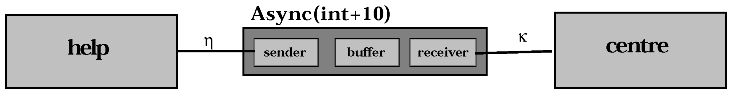

Fig. 10.

In this case, we have used the connector *Async*[$t + K$] with t instantiated with sort *int* and K instantiated with 10. In more abstract levels of design, it may be useful to use *Async*[$t + K$] for coordinating the activities of parameterized designs. In such cases, the instantiation of the connector has to be defined by refinement morphisms between parameterized designs.

A refinement morphism from a parameterized design $P_1(\Pi_1)$ to a parameterized design $P_2(\Pi_2)$ is a morphism of ***r-DSGN*** *from P_1 to P_2 for which Π_1 is a subspecification of* Π_2.

In the next section, we shall see examples of such kind of morphisms.

3.4 Adapting Connectors

As explained before, it is important to have principled ways to adapt connectors to new situations, for instance in order to incorporate features such as compression, fault-tolerance, security and monitoring, among others.

Let us consider *compression* once more as an example. In this case, the goal is to adapt a connector that represents a communication protocol in order to compress data for transmission in a transparent way. In order to be able to give a first-class description of this form of adaptation, the kind of communication protocol modeled by the adapted connector needs to be made more precise. We shall describe the *compression* adaptation mechanism only for connectors that model unidirectional communication protocols.

A generic unidirectional communication protocol can be modeled by the binary connector *Uni-comm*[*s*] (see Figure 11):

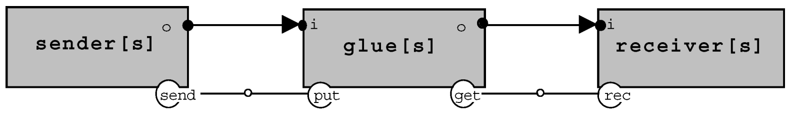

Fig. 11.

where

```
design   glue[s] is
in       i:s
out      o:s
do       put: true,false → skip
[]  prv  prod[o]: true,false → true
[]       get: true,false → skip
```

and *sender[s]* and *receiver[s]* are defined as before. Notice that this glue leaves completely unspecified the way in which messages are processed and transmitted.

Our aim is to install a compression/decompression service over *Uni-comm*. That is to say, our aim is to apply an operator to *Uni-comm* such that, in the resulting connector, a message sent by the sender is compressed before it is transmitted through *Uni-comm* and then decompressed before it is delivered to the receiver. We shall see that such an operator can be described by a higher-order connector where the compression and decompression algorithms are taken as parameters. More concretely, it is parameterized by the algebraic specification described below.

```
spec  Ξcd  is Ξnat+
sorts      s,t
ops        comp:t->s          decomp:s->t
           size_s:s->nat      size_t:t->nat
axioms     decomp(comp(x))=x, for any x:t
           size_s(comp(x)) ≤  size_t(x), for any x:t
```

Sorts *t* and *s* represent the types of original and compressed messages, respectively. The operation *comp* represents the process of compression of a single message, and *decomp* the inverse process of decompression. It is required that the size of the compressed message is not greater than the size of the original message. At configuration time, these data elements must be instantiated with specific sorts and operations.

The higher-order connector itself, which we name *Compression*(*Uni-comm*)[Ξ^{cd}], is defined by

—the binary connector *Compression*[Ξ^{cd}] (Figure 12)

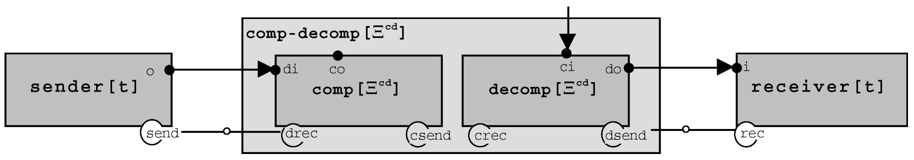

Fig. 12.

where the glue, *comp-decomp*[Ξ^{cd}], is defined in terms of a configuration with the following two components:

```
design   comp[Ξcd] is                  design   decomp[Ξcd] is
in       di:t                          in       ci:s
out      co:s                          out      do:t
prv      v:t; rd,msg:bool              prv      v:s; rd,msg:bool
do       drec: ¬msg →v:=di             do       crec: ¬msg →v:=ci
         ||msg:=true                            ||msg:=true
[] prv   comp:¬rd∧ msg →               [] prv   dec:¬rd∧ msg → do:=
         co:=comp(v)||rd:=true                  decomp(v)||rd:=true
[]       csend:rd → rd:=false          []       dsend: rd → rd:=false
         ||msg:=false                           ||msg:=false
```

Design *comp*[Ξ^{cd}] models the compression of messages of type *t* received through *di* into messages of type *s* that are then transmitted through *co*. Design *decomp*[Ξ^{cd}] models the decompression of messages of type *s* received through *ci* into messages of type *t* that are then transmitted through *do*.

—the connector *Uni-comm*[*s*]—the formal parameter;

—the refinement morphisms

η_s:*sender*[*s*] → *comp-decomp*[Ξ^{cd}] and η_r : *receiver*[*s*] → *comp-decomp*[Ξ^{cd}]

induced, respectively, by the refinement morphisms

$$\eta_s^* : sender[s] \to comp[\Xi^{cd}]$$

$$\eta_s^*(o) = co,\ \eta_s^*(rd) = rd,\ \eta_s^*(comp) = prod,\ \eta_s^*(csend) = send$$

$$\eta_r^* : receiver[s] \to decomp[\Xi^{cd}]$$

$$\eta_r^*(i) = ci,\ \eta_r^*(crec) = rec$$

Because components *comp* and *decomp* do not interact, any component refined by one of them is also refined by their composition *comp-decomp*$[\Xi^{cd}]$. The corresponding induced morphisms have only to take into account the renaming of channels and actions that takes place in composition.

Putting the two previous pictures together we get a graphical representation of the higher-order connector *Compression*(*Uni-comm*)$[\Xi^{cd}]$ (Figure 13).

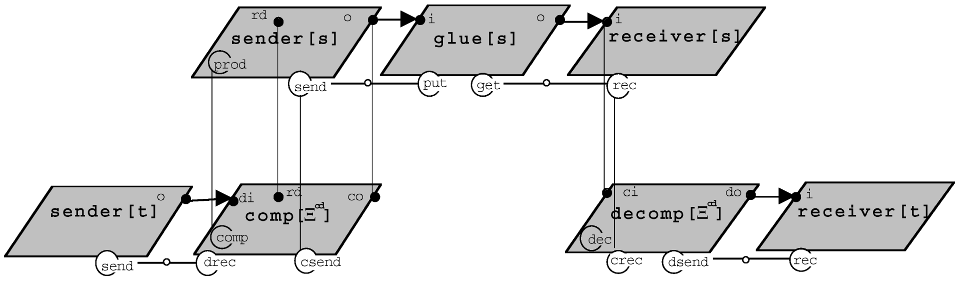

Fig. 13.

In summary, *Compression*(*Uni-comm*)$[\Xi^{cd}]$ has the formal parameter *Uni-comm*$[s]$, which restricts the actual connectors to which the service of compression/decompression can be applied—it requires that the actual connector models a unidirectional communication protocol. The connector *Compression* describes, on the one hand, that messages sent by the actual sender are transmitted to *comp* which compresses them and, on the other hand, that *decomp* decompresses the messages it receives and delivers the result to the actual receiver. Finally, the two refinement morphisms establish the instantiation of *Uni-comm*$[s]$ with *comp*[s] in the role of sender and *decomp*$[s]$ in the role of receiver. In this way, it is established that the formal parameter *Uni-comm*$[s]$ is the connector used to transmit compressed messages.

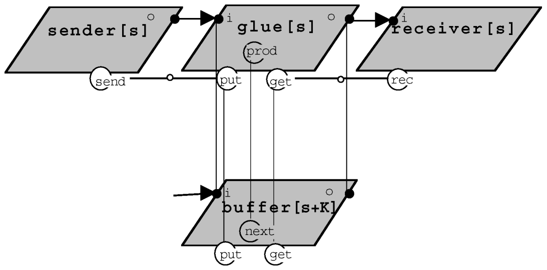

Fig. 14.

Now it remains to explain the procedure of parameter passing, that is, how the service just described can be installed over a specific connector and how the resulting connector is obtained.

We consider again the *Async* connector. In this case, it is not difficult to realize that we may replace the formal parameter of *Compression*(*Uni-comm*)[Ξ^{cd}] by *Async* because this connector does model a unidirectional communication protocol. More concretely, *Async* has exactly the same roles that *Uni-comm* and its glue is a refinement of *Uni-comm*'s glue (see Figure 14).

In a more general situation, the instantiation of a higher-order connector is established by a suitable fitting morphism from the formal to the actual connector. Such a morphism formulates the correspondence between the roles and glue of the formal parameter with those of the actual parameter connector. In the next section, we present and discuss these morphisms in more detail.

The construction of a new connector from the given higher-order connector and the actual parameter connector is straightforward. We only need to compose the interconnections of the *buffer* to *sender* and *receiver* with the refinements η_s and η_r that define the instantiation of *Uni-comm* with *comp* and *decomp*, respectively. For example, channel *co* of *comp* becomes connected to the input channel *i* of *buffer* because *co* corresponds to the channel *o* of *sender* which in turn is, in *Async*, connected to *i*. The resulting configuration fully defines the connector *Compression*(*Async*)[$\Xi^{cd} + K$]. Its roles are *sender* and *receiver* and its glue *c-buffer-d* [$\Xi^{cd} + K$] is defined in terms of a configuration involving *comp*, *decomp* and *buffer* as shown below. (See Figure 15.)

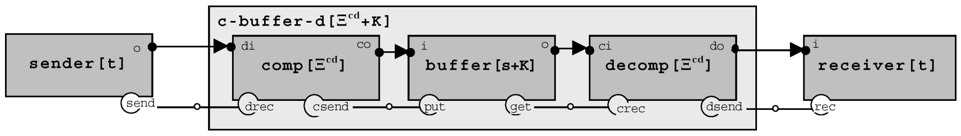

Fig. 15.

Summarizing, in this section, we have described the installation of a compression-decompression service over a unidirectional communication protocol as a parameterized entity that has connectors as parameters and result and, thus, is called a higher-order connector. Then we have explained how the higher-order connector can be instantiated with a specific connector and, finally, we showed how the resulting connector is obtained.

We end this section by presenting another example of a higher-order connector—*monitoring*. The aim is to model the adaptation of a unidirectional communication protocol in order to transmit certain kind of messages (e.g., error messages) to a monitoring component.

The kind of messages that should be transmitted to the monitoring component is taken as a parameter. More concretely, we define a higher-order connector that is parameterized by the following algebraic specification:

```
spec Ξ^m is  Ξ^bool+
sorts        s
ops          to_monitor:s->bool
```

Sort *s* represents the type of messages, and operation *to_monitor* identifies the special kind of messages that are to be monitored. We use Ξ^{bool} to represent the subspecification of Ξ that is concerned with the specification of Booleans.

In order to simplify the presentation, we shall consider that the communication with the monitoring component is achieved by synchronous message passing. However, it would be more appropriate to model *monitoring* with a higher-order connector with two formal parameters, both modelling a unidirectional communication protocol. One of them would be used for the normal transmission of messages and the other for the transmission to the monitoring component.

The higher-order connector itself, *Monitoring*(*Uni-comm*)[Ξ^m], consists of

—the connector *Monitoring*[Ξ^m] defined by Figure 16,

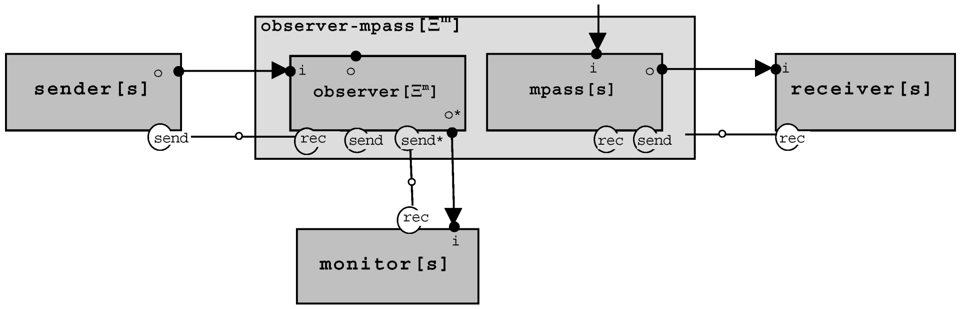

Fig. 16.

where the glue, *observer-mpass*[Ξ^m], is defined in terms of a configuration with the following two components:

```
design   observer[Ξm] is             design   mpass[s] is
in       i:s                           in     i:s
out      o,o*:s                        out    o:s
prv      v:s; rd,rd*,msg:bool          prv    rd:bool
do       rec: ¬msg→ v:=i               do     rec:¬rd → o := i
         ||msg:=true                          ||rd:=true
[] prv   obsv:¬rd∧¬rd*∧msg →           []     send:rd → rd:=false
         msg:=false||o:=v||rd:=true||
         o*:=v||rd*:=to_monitor(i)
[]       send:rd  → rd:=false
[]       send*:rd*→ rd* :=false
```

Component *observer*[Ξ^m] observes the messages to be transmitted and forwards a copy of certain transmitted messages to a third component. More precisely, it sends through *o* the messages received in *i*, and sends through *o** those messages that satisfy *to_monitor*. Component *mpass*[*s*] just transmits through *o* the messages received in *i*.

The connector has three roles—*sender*, *receiver* and *monitor*. The role *monitor*[*s*] is similar to *receiver*[*s*]:

```
design    monitor[s] is
in        i: s
do        rec: true → skip
```

Notice, however, that the progress condition of *rec* is true in *monitor* and false in *receiver*. This means that any component that acts as monitor must be always willing to read the values that are input through *i* whereas the actual receiving component may decide when and how many times it will read the values sent to it. In this way, it is ensured that the monitoring component listens to (part of) the communication between the connected components without affecting it.

—the connector *Uni-comm*[*s*]—the formal parameter;

—the refinement morphisms depicted in Figure 17.

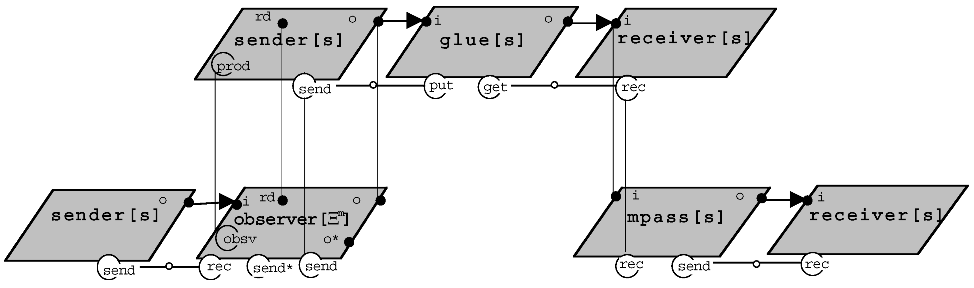

Fig. 17.

4. AN ADL-INDEPENDENT NOTION OF HIGHER-ORDER CONNECTOR

The notion of higher-order connector presented for CommUnity can be generalized to other design formalisms. In this section, based on previous work [Fiadeiro et al. 2003], we start by identifying the properties that such formalisms need to satisfy to support the architectural concepts and mechanisms that we have illustrated for CommUnity. Then, we shall present ADL-independent notions of connector and higher-order connector.

First, we need to fix a framework in which designs, configurations and relationships between designs, such as refinement, can be formally described. Our experience in formalizing notions of structure in Computing, building on previous work of J. Goguen on General Systems Theory, suggests that, as illustrated in Section 3, Category Theory provides a convenient framework for our purpose. More concretely, we shall consider that a formalism supporting system design includes:

—a category ***c-DESC*** of component designs in which systems of interconnected components are modeled through diagrams;

—for every set *CD* of component designs, a set *Conf*(*CD*) consisting of all well-formed configurations that can be built from the components in *CD*. Each such configuration is a diagram in ***c-DESC*** that is guaranteed to have a colimit. Typically, *Conf* is given through a set of rules that govern the interconnection of components in the formalism.

—a category ***r-DESC*** with the same objects as ***c-DESC***, but in which morphisms model refinement, that is, a morphism $\eta : S \to S'$ in ***r-DESC*** expresses that S' refines S, identifying the design decisions that lead from S to S'. Because the design of a composite system is given by a colimit of a diagram in ***c-DESC*** and, hence, is defined up to an isomorphism in ***c-DESC***, refinement morphisms must be such that designs that are isomorphic in ***c-DESC*** refine, and are refined exactly by, the same designs. Hence, it is required that *Isomorph*(***c-DESC***) $\subseteq$ *Isomorph*(***r-DESC***).

Summarizing, all that we require is a notion of system design, a relationship between designs that captures components of systems, another relationship that captures refinement, and criteria for determining when a diagram of interconnected components is a well-formed configuration.

4.1 Architectural Schools

In the context of this categorical framework, we shall now present the properties of a design formalism for supporting the architectural concepts that we have illustrated for CommUnity. These properties define what we call an architectural school.

The categorical properties that a formalism needs to satisfy for supporting the notion of connector and its instantiation mechanism are identified and discussed in detail in Fiadeiro et al. [2003]. We shall summarize this characterization and extend it in order to support higher-order connectors, too.

4.1.1 *Coordination.* A key property of a formalism for supporting architectural design is that it provides a clear separation between the description of the individual behavior of components and that of their interaction in the overall system organisation.

We shall take the separation between coordination and computation to be materialized through a functor ***sig: c-DESC*** $\to$ ***SIG*** mapping designs to signatures, forgetting their computational aspects. The fact that the computational side does not play any role in the interconnection of systems can be captured by the following properties of this functor:

—***sig*** *is faithful;*

—***sig*** *lifts colimits of well-formed configurations;*

—***sig*** *has discrete structures;*

together with the following property on the well-formed configuration criterion

—*given any pair of configuration diagrams* $\boldsymbol{dia}_1$, $\boldsymbol{dia}_2$ *s.t.* $\boldsymbol{dia}_1;\boldsymbol{sig} = \boldsymbol{dia}_2;\boldsymbol{sig}$, *either both are well-formed or both are ill-formed.*

The first property states that morphisms of systems cannot induce more relationships than those that can be established between their underlying signatures. The second property means that if we interconnect system components through a well-formed configuration, then any colimit of the underlying diagram of signatures establishes a signature for which a computational part exists that captures the joint behavior of the interconnected components. The third property implies that every signature θ has a realization as a system component ***desc*** (θ). In a sense, sources of morphisms in diagrams of designs are, essentially, signatures.

These three properties ensure that any interconnection of systems can be established via their signatures, legitimizing the use of signatures as channels in configuration diagrams. By requiring that any two configuration diagrams that establish the same interconnections at the level of signatures be either both well formed or both ill formed, the fourth property ensures that the criteria for well-formed configurations do not rely on the computational parts of designs.

In such situation, we say that *the formalism <**c-DESC**, Conf, **r-DESC**> is coordinated over **SIG** through the functor **sig***.

4.1.2 *Compositionality.* Another crucial property for supporting architectural design is in the interplay between structuring systems in architectural terms and refinement.

We start by noticing that, in order to support the refinement of an abstract description of a system, it must be possible to propagate the interactions between the components of the system when their designs are replaced by more concrete ones. This situation can be characterized by the existence, for every well-formed configuration ***dia*** involving designs $\{S_1, \ldots, S_k\}$ and every set of refinements morphisms $\{\eta_i : S_i \to S'_i : i \in 1..k\}$, of a well-formed configuration diagram ***dia*** $+ (\eta_i)_{i\in 1..k}$ obtained by "composing" in some way each refinement morphism η_i with the morphisms of ***dia*** whose target is S_i. The diagram ***dia*** $+ (\eta_i)_{i\in 1..k}$ describes the system obtained by replacing the designs of the components of the system (S_i) by more concrete ones (S'_i). (See Figures 18 and 19.)

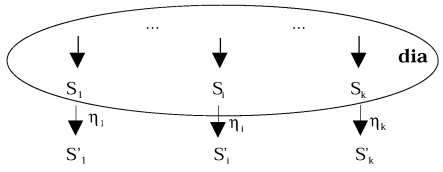

Fig. 18.

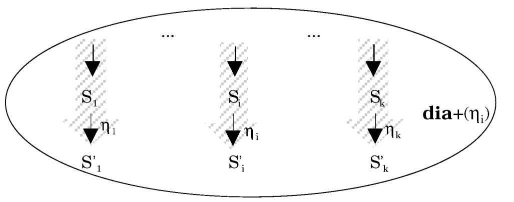

Fig. 19.

Naturally, a method of propagation of the interactions between the components of the system when their designs are replaced by more concrete ones is only significant if all decisions made previously are respected. In other words, the correctness criterion for this form of "configuration refinement" is that the colimit of ***dia*** + $(\eta_i)_{i \in 1..k}$ provides a refinement for the colimit of ***dia***. (See Figure 20.)

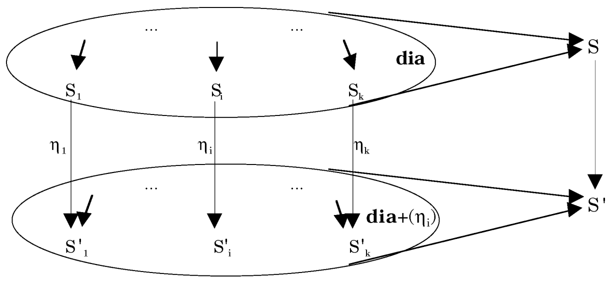

Fig. 20.

As explained in [Fiadeiro et al. 2003], a formalism supports the notion of connector if it is coordinated and has a correct method of propagation of the interactions between the components of the system when their designs are replaced by more concrete ones.

To characterize the formalisms that support the notion of higher-order connector we have illustrated for CommUnity, it is necessary to know exactly which is the notion of configuration refinement of the formalism, that is, in which situations a configuration is considered a refinement of another configuration. Given that configurations are made of components and interconnections, it is natural that a design formalism supports a notion of configuration refinement *CR* that, in addition to an operator + for the refinement of components, also allows the refinement of interconnections.

We require *CR* to be correct, that is,

*For every **dia** and **dia'** such that **dia'** refines **dia** according to CR, the colimit of **dia'** provides a refinement for the colimit of **dia**.*

Furthermore, the configurations of the form ***dia*** + (η_i) must be considered, according to *CR*, refinements of ***dia***. In this situation, we shall say that *the formalism is compositional with respect to CR.*

*In summary, a formalism F = <**c-DESC**, Conf, **r-DESC**> is called an architectural school over a functor **sig: c-DESC→SIG** and a configuration refinement notion CR if*

*—F is coordinated over **SIG** through **sig**;*
—F is compositional with respect to CR.

Besides CommUnity, other formalisms define architectural schools. In [Fiadeiro et al. 2003], we show this is the case of the coordination language Gamma [Banâtre and Le Métayer 1993]. Also the concurrency models that are formalized in Sassone et al. [1993] using categorical techniques satisfy the properties that we have laid down for architectural schools.

4.2 Connectors

Consider given an architectural school $F = \langle$***c-DESC***, ***Conf***, ***r-DESC***$\rangle$ over ***sig: c-DESC***$\rightarrow$***SIG*** and CR. The generalization of the notion of connector presented for CommUnity in Section 3.2 is straightforward.

—*A connection consists of*
  —*two designs G and R, called the glue and the role of the connection, respectively;*
  —*a signature θ and two morphisms σ:* ***desc*** $(\theta) \rightarrow G$, μ: ***desc*** $(\theta) \rightarrow R$ *in* ***c-DESC*** *connecting the glue and the role.*

—*A connector is a finite set of connections with the same glue that, together, constitute a well-formed configuration.* (*See Figure* 21.)

$$\begin{tikzcd}
 & G & \\
\theta_1 \arrow[ur, "\sigma_1"] \arrow[d, "\mu_1"] & \theta_i \arrow[u, "\sigma_i"] \arrow[d, "\mu_i"] & \theta_n \arrow[ul, "\sigma_n"'] \arrow[d, "\mu_n"] \\
R_1 & R_i & R_n
\end{tikzcd}$$

Fig. 21.

—*The semantics of a connector is the colimit of the diagram formed by its connections.*

A connector can be applied to specific components of a system under construction, establishing the intended interactions between them, by instantiating the roles of the connector with those components. Role instantiation has to obey a compatibility requirement, which is expressed via the refinement morphisms of ***r-DESC***.

An instantiation of a connector is defined as follows:

—*An instantiation of a connection with role R consists of a design P together with a refinement morphism $\eta : R \rightarrow P$ in* ***r-DESC***.

$$\begin{tikzcd}
 & G & \\
\theta_1 \arrow[ur] \arrow[d, "\mu_1"] & \theta_i \arrow[u] \arrow[d, "\mu_i"] & \theta_n \arrow[ul] \arrow[d, "\mu_n"] \\
R_1 \arrow[d, "\eta_1"] & R_i \arrow[d, "\eta_i"] & R_n \arrow[d, "\eta_n"] \\
P_1 & P_i & P_n
\end{tikzcd}$$

Fig. 22.

—*An instantiation of a connector consists of an instantiation for each of its connections such that the diagram in* ***c-DESC*** *connecting the role instances to the glue, obtained by composing the role morphism of each connection with its* *instantiation (given by* $<\sigma_i, \mu_i> + \eta_i$*), constitutes a well-formed configuration. (See Figure* 22.*)*

—*The semantics of a connector instantiation is the colimit of the diagram in* ***c-DESC*** *formed as described above.*

The compositionality of the design formalism ensures that the system that results from a instantiation of a connector C refines the semantics of C. In this way, the properties of connectors can be understood independently of specific contexts in which they are used.

Notice that, as illustrated for CommUnity, refinement morphisms are defined between designs, not just signatures, that is, they can take into account more or less complex behavioral properties. Hence, our notion of instantiation and, therefore, parameter checking is not just "syntatic". In fact, it is general enough to offer whoever is defining the architectural school the possibility of capturing the semantic conditions that are associated with whatever is considered to be the "right" notion of instantiation.

4.3 Higher-Order Connectors

The notion of higher-order connector, as a connector that takes one connector as parameter and delivers another as a result, can be defined as follows:

—*A higher-order connector (hoc) consists of*

  —*a connector pC, called the formal parameter of the hoc; its roles, glue and connections are called, respectively, the parametric roles, the parametric glue and the parametric connections of the hoc;*

  —*a connector C—its roles and glue are also called the roles and the glue of the hoc;*

  —*an instantiation of the formal parameter connector with the glue of the hoc, that is, a refinement morphism* η_i *from each of the parametric roles to the glue, such that the diagram in* ***c-DESC*** *obtained by composing the role morphism of each parametric connection with its instantiation (Figure* 23*)*

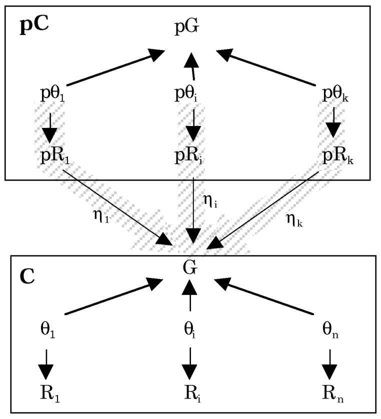

Fig. 23.

constitutes a well-formed configuration.

—*The semantics of a higher-order connector is the connector depicted below. Its roles are the roles of C and its glue is G', a design returned by the colimit of the configuration* $pC + (\eta_i)$ (*Figure* 24).

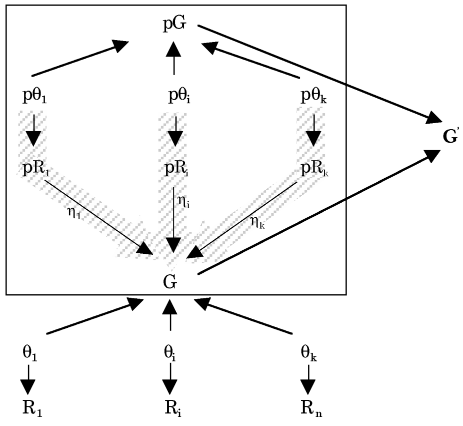

Fig. 24.

For simplicity, we have imposed one single parameter to the higher-order connector. However, the definition can be extended to the case of several parameters in a straightforward way.

Intuitively, the instantiation of the formal parameter of a higher-order connector can be regarded as the replacement of a connector (the formal parameter *pC*) that was instantiated to given components of a system (the glue of the hoc) by another connector (the actual parameter). In addition, the type of interconnection that *pC* ensures must be preserved. In other words, the design that results from the replacement must be a refinement of the design from which we started.

Like for connectors, the instantiation of the formal parameter of a higher-order connector is established via a fitting morphism from the formal to the actual parameter. These morphisms, on the one hand, formulate the correspondence between roles and glue of the formal with those of the actual parameter and, on the other hand, capture conditions under which the "functionality" of the formal parameter is preserved.

In order to be able to use, in the design of a given system, a connector C in place of a connector C', it is obvious that the two connectors must have the same number of roles. Furthermore, C' has to admit to be instantiated with the same components than C. That is to say, every restriction on the components to which C' can be applied must also be a restriction imposed by C. In this way, fitting morphisms must require that each of the roles of C' is refined by the corresponding role of C.

For instance, we cannot replace a connector *Sync* by a *Monitoring* at once. We first have to provide the component that will play the role *monitor* in the system and then encapsulate this component by making it part of the glue (see

Fiadeiro et al. [2003]). The resulting connector has only two roles, which are exactly the same of *Sync*, and hence, it can be used as a *Sync* connector.

As shown in Section 3, namely with the connector *Uni-comm*, connectors may be based on glues that are not fully developed as designs (may be underspecified) and, nevertheless, the concrete commitments that have already been made determine to some extent the type of interconnection that the connector will ensure. The type of interconnection is clearly preserved if we simply consider a more concrete glue, that is, if we refine the glue. Hence, fitting morphisms must allow for arbitrary refinements of the glue.

Having this in mind, we arrive at the following notion of fitting morphism:

—*A fitting morphism ϕ from a connection $<\sigma_1 : \textbf{desc}\ (\theta_1) \to G_1, \mu_1 : \textbf{desc}\ (\theta_1) \to R_1>$ to a connection $<\sigma_2 : \textbf{desc}\ (\theta_2) \to G_2, \mu_2 : \textbf{desc}\ (\theta_2) \to R_2>$ consists of a pair $<\phi_G : G_1 \to G_2, \phi_R : R_2 \to R_1>$ of refinement morphisms in* ***r-DESC*** *such that the interconnection $<\sigma_1, \mu_1> + \phi_G$ of R_1 with G_2 is, in accordance with the configuration refinement CR, refined by the interconnection $<\sigma_2, \mu_2> + \phi_R$.* (*See Figure* 25.)

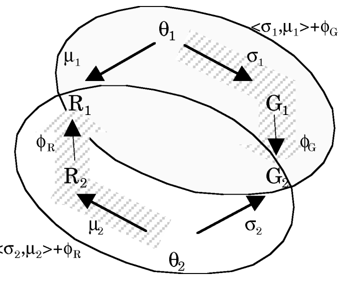

Fig. 25.

—*A fitting morphism ϕ from a connector C_1 to a connector C_2 with the same number of connections consists of a fitting morphism ϕ from each of C_1's connections to each of C_2's connections, all with the same glue refinement ϕ_G.*

If there exists a fitting morphism from a connector C_1 to a connectorC_2, then we may replace each occurrence of the connector C_1 in an architectural description of a system by an occurrence of C_2. The compositionality of the design formalism with respect to the configuration refinement *CR* ensures that every coordination decision made previously is preserved.

Based on fitting morphisms between connectors, we define an instantiation of a higher-order connector.

—*An instantiation of a higher-order connector with formal parameter pC* (*Figure* 26) *consists of a connector C^A (the actual parameter) together with a fitting morphism $\phi : pC \to C^A$, such that the diagram in* ***c-DESC*** *obtained by composing the role morphisms of each actual connection with the corresponding fitting component and then with the role instantiation* (*Figure* 27) *constitutes a well-formed configuration.*

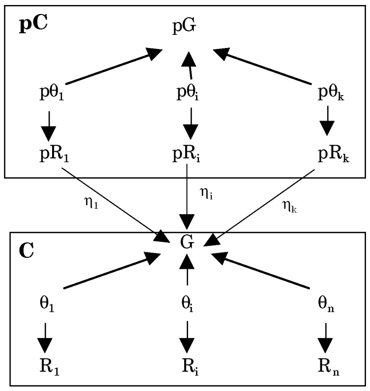

Fig. 26.

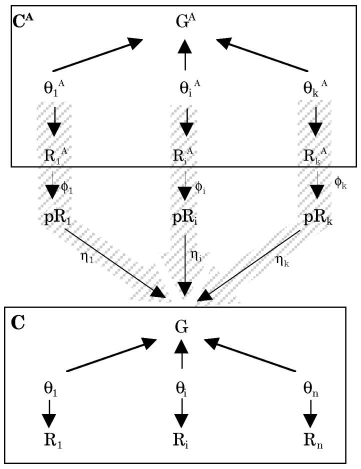

Fig. 27.

—*The semantics of a higher-order connector instantiation is the connector with the same roles as C and its glue is a design returned by the colimit of the configuration $C^A + (\phi_i;\eta_i)$.*

5. COMPOSITION OF HIGHER-ORDER CONNECTORS

Higher-order connectors facilitate the separation of concerns in the development of complex connectors and their compositional construction. For instance, we have seen that compression and monitoring can be modeled separately as higher-order connectors. Although we have not shown it, it is not very difficult to realize that compression can be applied to a connector that models a unidirectional communication protocol and then monitoring can be applied to the resulting connector.

An important feature of our notion of higher-order connector is that different kinds of functionality, modeled separately by different higher-order connectors, can be combined, giving rise also to a higher-order connector. In this way, it is possible to analyze the properties that such compositions exhibit, namely to investigate whether undesirable properties emerge and desirable properties are preserved.

The key idea for composition of hocs is the instantiation of a hoc with a hoc. In this section, we shall present this more general form of instantiation—*parameterized instantiation*. So, for instance, *Monitoring(Uni-comm)* can be instantiated with *Compress(Uni-comm)*, giving rise to the hoc *Monitoring&Compress(Uni-comm)*, which corresponds to a form of composition of *Monitoring* and *Compress* in which the messages are first observed, and possibly transmitted to the monitoring component, then are compressed and finally are transmitted via *Uni-comm*.

The definition of parameterized instantiation of a hoc is similar to the definition of the standard instantiation.

—*A parameterized instantiation of a higher-order connector HC with formal parameter pC consists of a higher-order connector HC' together with a fitting morphism $\phi : pC \to Con'$* (*Figure* 28)*, where Con' is the connector that gives the semantics of HC', such that it is possible to extend, in a unique way, the instantiation of pC' with G' to an instantiation of pC' with the colimit of the diagram $C' + (\phi_i; \eta_i)_{i\in 1..k}$* (*Figure* 29)*, which connects the glues of HC' and HC.*

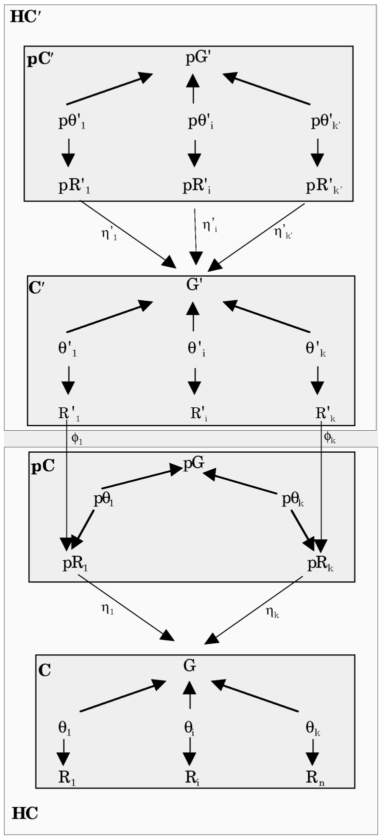

Fig. 28.

In Figure 29, G^{new} *is the colimit of the diagram* $C' + (\phi_i; \eta_i)_{i \in 1..k}$ *and we have used dotted lines for the refinement morphisms whose existence we are requiring.*

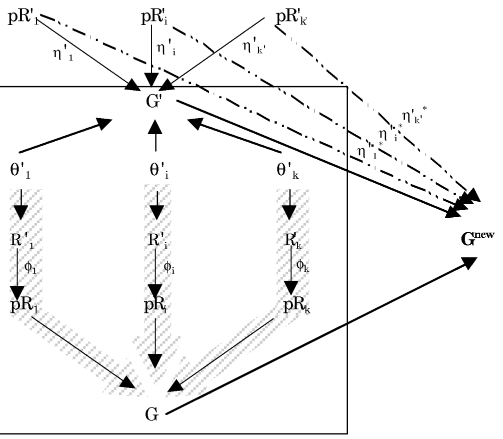

Fig. 29.

—*The semantics of a parameterized instantiation is the higher-order connector depicted in Figure* 30.

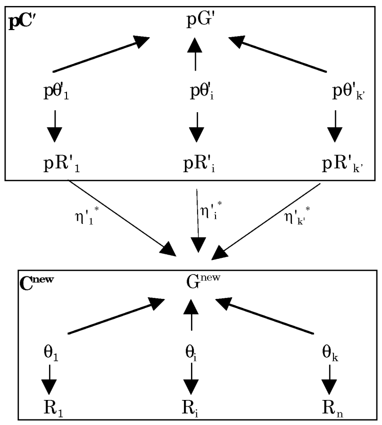

Fig. 30.

For instance, consider that we want to combine the service of compression of messages with monitoring. If we consider the parameterized instantiation of *Monitoring*(*Uni-comm*)[Ξ^m] with *Compress*(*Uni-comm*)[Ξ^{cd}] defined by the fitting morphism (Figure 31)

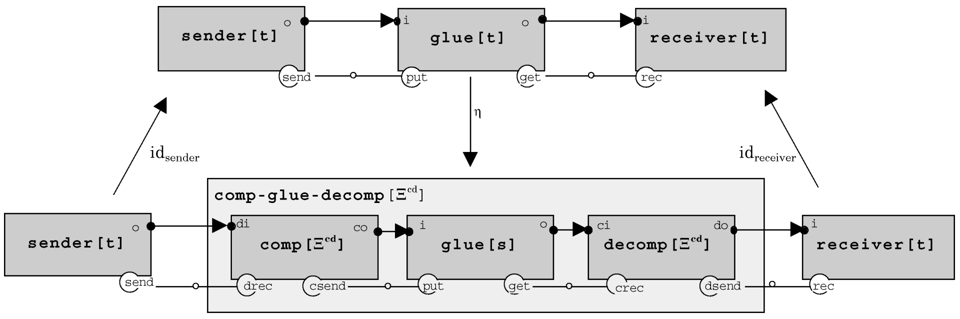

Fig. 31.

where the refinement morphism

$$\eta : glue[t] \rightarrow comp\text{-}glue\text{-}decomp[\Xi^{cd}]$$

is defined by

$$\eta(i) = \sigma_c(di), \eta(o) = \sigma_d(do),$$
$$\eta\left(\sigma_c^{-1}(drec)\right) = put, \eta\left(\sigma_d^{-1}(dec)\right) = prod, \eta\left(\sigma_d^{-1}(dsend)\right) = get$$

where $\sigma_c : comp \rightarrow comp\text{-}glue\text{-}decomp[\Xi^{cd}]$ and $\sigma_d : decomp \rightarrow comp\text{-}glue\text{-}decomp[\Xi^{cd}]$ are the morphisms in ***c-DSGN*** returned by the colimit of the diagram (Figure 32)

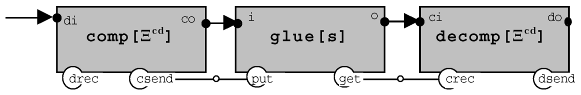

Fig. 32.

This composition gives rise to the hoc *Monitoring&Compress*(*Uni-comm*)$[\Xi^{cd} + \Xi^m]$ that is constituted by

—the connector *Monitoring&Compress*$[\Xi^{cd} + \Xi^m]$ defined in Figure 33;

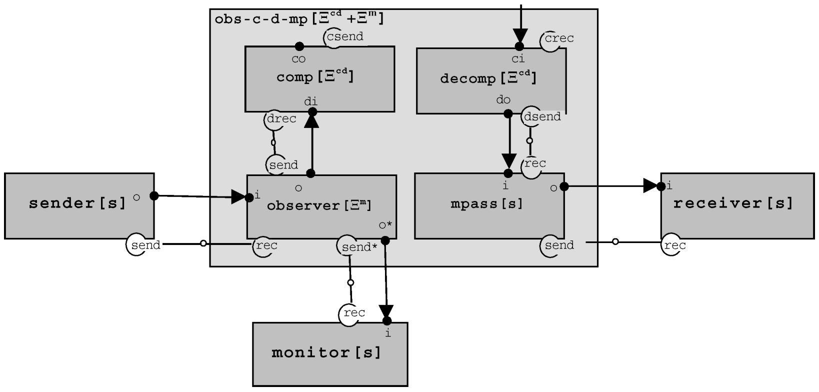

Fig. 33.

—the connector *Uni-comm*[*s*]—the formal parameter;

—the refinement morphisms

$$\eta_s : sender[s] \to obs\text{-}c\text{-}d\text{-}mpass[\Xi^{cd} + \Xi^m] \text{ and}$$
$$\eta_r : receiver[s] \to obs\text{-}c\text{-}d\text{-}mpass[\Xi^{cd} + \Xi^m]$$

obtained by composing, at the level of signatures, the morphisms $sender[s] \to comp[\Xi^{cd}]$ and $receiver[s] \to decomp[\Xi^m]$ of $Compression(Uni\text{-}comm)[\Xi^{cd}]$ with the morphisms, respectively, $comp[\Xi^{cd}] \to obs\text{-}c\text{-}d\text{-}mpass[\Xi^{cd} + \Xi^m]$ and $decomp[\Xi^{cd}] \to obs\text{-}c\text{-}d\text{-}mpass[\Xi^{cd} + \Xi^m]$ which are given by the colimit construction (for the sake of space, we omit the proof of the fact that these signatures morphisms do define refinement morphisms).

It is not difficult to realize this hoc works as described before: first messages are observed and possibly transmitted to the monitoring component, then are compressed and finally are transmitted via *Uni-comm*.

6. RELATED WORK

Our definitions agree with Garlan's original proposal [Garlan 1998] of a hoc as an operator over connectors for supporting connector construction through incremental transformation, hence allowing one to define more complex interactions in a more systematic way. More concretely, Spitznagel and Garlan [2001] propose that a connector transformation be modeled as a function—from one or more connectors to a new connector—defined in terms of its inputs, preconditions on its application and postconditions on its result. They formalize these ideas in the context of a particular ADL, namely Wright, relying on the specific language and semantics of CSP.

6.1 Adaptation of Connectors vs. Adaptation of Components

As explained, hocs may be used to represent connector adaptation and, in particular, the installation of additional services, such as security or fault-tolerance. For the same purpose, a radically different approach is to apply component adaptation, for instance by using wrappers or packaging (e.g., Katz [1993], Bosch [1999], and Denker et al. [1999]).

There are several reasons to point out in favor of our approach. For instance, as argued in Garlan [1998], it is not always possible to adapt components to work with the existing connectors. Even in those cases where it is feasible, a better alternative is to modify the connectors because usually there are fewer connector types than components types. Despite the obvious methodological differences, the two approaches bring about equivalent transformations in the sense that we explain below.

Consider again the compression example. As explained before, the instantiation of the hoc *Compression* with connector *Async* gives rise to the following configuration (Figure 34):

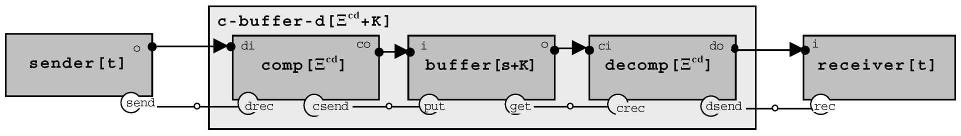

Fig. 34.

This configuration is equivalent to the one below (Figure 35). This configuration represents a system where the compression service is installed by adding (different) wrappers to the sender and receiver. The connector *Async* is used once again to connect the two components.

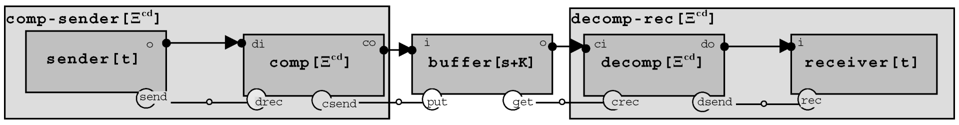

Fig. 35.

The two configurations (Figures 34 and 35) are equivalent in the sense that, from a categorical point of view, they correspond to two different ways of calculating the colimit of the same diagram and, hence, they give rise to equivalent designs (i.e., designs considered isomorphic in ***c-DESC***).

6.2 Integrating Extrafunctional Requirements in Connectors

The notion of architectural connector in the style defined by Allen and Garlan is also the basis for a completely different approach to the specification of extrafunctional properties of software architectures descriptions, such as security and fault-tolerance. Issarny and colleagues [1998] propose an extension of architectural connectors with a set of first-order formulas specifying the extrafunctional properties offered by the connector. This extension relies on the specific language and semantics of CSP and is based on the fact that the behavior of the protocol can be formally defined in terms of the predicates according to Hoare's logic. These predicates essentially characterize the coordination actions that are carried out by the protocol. This approach is mainly tailored to analyze whether a given architecture has some desired extrafunctional properties whereas our approach is geared towards system construction.

6.3 Behavioral Analysis

Many existing ADLs have some associated technique or tool to analyze the resulting behavior of the designed system. For example, ADLs that use process calculi to specify components and connectors typically use some model-checking tool: Darwin uses LTSA [Magee et al. 1999] and Wright uses FDR [Allen and Garlan 1997], just to mention two better-known examples. Our current support for behavioral analysis is twofold.

On the one hand, the CommUnity Workbench [Wermelinger and Oliveira 2002] is being developed (in Java) as a proof of concept of the theoretical framework, hiding the underlying "mathematical machinery" from the user. Currently, the tool provides a graphical integrated development environment to write CommUnity programs (with fixed data types), define connectors, draw an architecture, calculate automatically its colimit, and run it. The workbench prevents the creation of ill-formed configurations (like binding output channels with each other) and gives great flexibility in testing CommUnity programs (like channel initialization, choice of which actions and channels to trace, and verification of invariants during execution). Higher-order connectors and some advanced features of CommUnity are not yet supported.

On the other hand, as mentioned in the end of Section 3.2, we defined a logic formalism for expressing the properties of CommUnity designs, namely the cooperation properties of a design in regard to its environment. We also developed a proof method for reasoning about CommUnity designs in a compositional way, that is, that allows us to reason about a system described through a configuration diagram, without requiring the calculation of the colimit design. This method can be also applied to connectors and hocs (given that they are defined by configuration diagrams) and allow us to reason about their capabilities. In future work, we would like to include an implementation of the logic formalism, possibly using an existing theorem prover, into the CommUnity Workbench.

6.4 Implementation

Given that hocs are parameterized entities, programming features that support parametrization can be very useful for their implementation. For instance, the implementation of network protocol stacks through the composition of modular protocol elements was shown to be naturally supported by the concept of module offered by SML [Biagioni et al. 1994]. In this work, protocols (connectors) are taken simply as collections of types, values and functions, which is a very restrictive implementation of the notion of connector.

7. CONCLUDING REMARKS

In this article, we continued our previous work on providing formal support to the definition and use of architectural techniques for software development. Building on the categorical semantics for the notion of architectural connector that, since 1995, we have developed in several papers (summarized in Fiadeiro et al. [2003]), we formalized a notion of higher-order connector that can be used for defining new connectors from existing ones by superposing aspects like security, fault-tolerance, etc. We showed that a transformation of a connector can be modeled by a parameterized entity that is essentially constituted by two connectors. One of these connectors is the formal parameter that defines the kind of connectors the transformation can be applied to. The other connector—the body of the hoc—concerns the transformation itself. Owing to the formal semantics of hocs, the transformation can be understood and analyzed.

For simplicity, we defined hocs with one parameter only, but the extension to several parameters is straightforward. However, this fact limited the kind of examples we have used throughout the artilce, namely it prevented us from showing that hocs can be used to model operators that represent more than an adaptation of connectors. For instance, it is necessary to consider an n-ary hoc in order to model an operation on n connectors so that the output of the first connector goes into the input of the second connector, etc. Another operation that requires more parameters is the aggregation of n connectors [Spitznagel and Garlan 2001], a combination of n connectors with a controller that determines which connector is active at a time. Such an operation can be useful for systems with transient interactions, for example, due to mobility of the components.

Although we first used CommUnity to illustrate the concept, we then presented a generalisation of these ideas which is applicable to any language that supports architectural design in a sense that was made precise in Section 4.1. These are criteria that we have checked to be met not only by CommUnity, but also by formalisms such as CSP which support well-known ADLs like Wright [Allen and Garlan 1997], as well as concurrency models like the ones that support, for instance, the Darwin approach [Magee et al. 1999], and specification logics like the one that is used, for instance, in Moriconi and Qian [1994]. Architectural approaches that are based on coordination languages and models [Gelernter and Carriero 1992] also comply with these criteria as shown in Fiadeiro et al. [2003] for Gamma [Banâtre and Le Métayer 1993].

Indeed, the use of Category Theory in the article is not an end in itself but, rather, a means of characterizing the proposed concepts and techniques in a way that, on the one hand, is independent of the way it can be offered to users in specific ADLs and, on the other hand, is amenable to formal analysis. As a result, we are able to compare how different ADLs support these notions and suggest ways in which they can be extended by incorporating constructions developed for other ADLs. For instance, the categorical framework led naturally to a notion of composition of higher-order connectors that turned out to be useful for combining orthogonal properties.

This level of formality also has important practical consequences. For instance, by enabling formal relationships (functors) to be established with logics for specification and verification (see Fiadeiro et al. [2003] for details and examples), the proposed semantics can be used to support the automatic derivation of properties that can be used to test the consistency and correctness of designs. As illustrated in Section 6, the categorical semantics of hocs can also be used to clarify the relationship between different methodological approaches, for instance adaptation of connectors vs adaptation of components.

Another important contribution of the proposed formalization is the possibility of using graph rewriting techniques for specifying runtime architectural changes. Previous work [Wermelinger et al. 2001] addressed the support that is required for an architectural-driven process of reconfiguration in which connectors, as well as components, can be replaced, added or deleted. The individual specification of independent aspects such as compression and fault-tolerance as higher-order connectors makes it easier to evolve systems at run-time. Through run-time reconfiguration of the system architecture, namely through the replacement of connectors, such services may be added only when necessary, hence preventing performance penalties when such complex interactions are not required.

Still on the more practical side of our work, we are now in the process of transposing these results to a "proof-of-concept"—the Coordination Development Environment [Gouveia et al. 2001]—that we have built with ATX Software for rigorously testing and validating the proposed approach and show how it can be realized in Java-based development platforms. This environment is based on a micro-architecture that transposes, into Java, the separation between Computation and Coordination that we formalized with the notion of architectural school [Andrade et al. 2000]. It has been used by us and ATX Software as a means of early and quick prototyping, to assess the practicality of our coordination-based approach to architectures, namely its scalability, and progressively gather methodological awareness that can lead to a systematic support to architectural construction. The idea now is to support the construction of new coordination contracts through the higher-order mechanisms that we have characterized and continue the process of maturing the technology in all its aspects.

We hope that these concluding paragraphs make clear how the results that we exposed in the article can impact software design methods and tools. Together, they constitute the foundations of the work that we have been developing towards our long term goal: to make available a rich toolbox that can assist software architects in systematizing and controlling the way they design, deploy and evolve architectures.

ACKNOWLEDGMENTS

We would like to thank Luís Andrade from ATX Software SA for the opportunity to discuss and validate many of these ideas. We would also like to thank the referees for their challenging requests.

REFERENCES

ALLEN, R. AND GARLAN, D. 1997. A formal basis for architectural connectors. *ACM Trans. Softw. Eng. Meth. 6*, 3 (July), 213–249.

ANDRADE, L., FIADEIRO, J. L., GOUVEIA, J., LOPES, A. AND WERMELINGER, M. 2000. Patterns for coordination. In *Proceedings of COORDINATION'00*, G. Catalin-Roman and A. Porto, Eds. Lecture Notes in Computer Science, vol. 1906. Springer-Verlag, New York, 317–322.

BANÂTRE, J. P. AND LE MÉTAYER, D. 1993. Programming by multiset transformation. *Commun. ACM 16*, 1, 55–77.

BASS, L., CLEMENTS, P., AND KASMAN, R. 1998. *Software Architecture in Practice*. Addison-Wesley, Reading, Mass.

BIAGIONI, E., HARPER, R., LEE, P., AND MILNES, B. 1994. Signatures for a network protocol stack: A system application of standard ML. In *Proceedings of the ACM Conference on LISP and Functional Programming*. ACM, New York, pp. 55–64.

BOSCH, J. 1999. Superimposition: A component adaptation technique. *Inf. Softw. Tech.*

CHANDY, K. AND MISRA, J. 1988. *Parallel Program Design—A Foundation*. Addison-Wesley, Reading, Mass.

DENKER, G., MESEGUER, J., AND TALCOTT, C. 1999. Rewriting semantics of meta-objects and composable distributed services. Internal report, Computer Science Laboratory, SRI International.

EHRIG, H. AND MAHR, B. 1985. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. Springer-Verlag, New York.

FIADEIRO, J. L., LOPES, A., AND WERMELINGER, M. 2003. A mathematical semantics for architectural connectors. In *Generic Programming*, R. Backhouse and J. Gibbons, Eds. Lecture Notes in Computer Science. Springer-Verlag, in print.

FIADEIRO, J. L. AND MAIBAUM, T. 1997. Categorical semantics of parallel program design. *Sci. Comput. Prog. 28*, 111–138.

FRANCEZ, N. AND FORMAN, I. 1996. *Interacting Processes*. Addison-Wesley, Reading, Mass.

GARLAN, D. 1998. Higher-order connectors. Position paper for the Workshop on Compositional Software Architectures, Jan.

GELERNTER, D. AND CARRIERO, N. 1992. Coordination languages and their significance. *Commun. ACM 35*, 2, 97–107.

GOGUEN, J. 1973. Categorical foundations for general systems theory. In *Advances in Cybernetics and Systems Research*, F. Pichler and R. Trappl, Eds. Transcripta Books, pp. 121–130.

GOGUEN, J. 1996. Parametrised programming and software architecture. In *Symposium on Software Reusability*. IEEE Computer Society Press, Los Alamitos, Calif.

GOUVEIA, J., KOUTSOUKOS, G., ANDRADE, L., AND FIADEIRO, J. L. 2001. Tool support for coordination-based software evolution. In *Technology of Object-Oriented Languages and Systems—TOOLS 38*, W. Pree, Ed. IEEE Computer Society Press, Los Alamitos, Calif., pp. 184–196.

HIRSCH, D., UCHITEL, S., AND YANKELEVICH, D. 1999. Towards a periodic table of connectors. In *Proceedings of Simposio en Tecnología de Software* (Buenos Aires, Argentina).

HOARE, C. A. R. 1985. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs., N.J.

ISSARNY, V., BIDAN, C., AND SARIDAKIS, T. 1998. Characterizing coordination architectures according to their non-functional execution properties. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences* (Jan.). IEEE Computer Society Press, Los Alamitos, Calif., pp. 275–283.

KATZ, S. 1993. A superimposition control construct for distributed systems. *ACM Trans. Prog. Lang. Syst. 15*, 2, 337–356.

LOPES, A. AND FIADEIRO, J. L. 1999. Using explicit state to describe architectures. In *FASE'99*, E. Astesiano, Ed. Lecture Notes in Computer Science, vol. 1577. Springer-Verlag, New York, pp. 144–160.

MAGEE, J., KRAMER, J., AND GIANNAKOPOULOU, D. 1999. Behaviour analysis of software architectures. In *Software Architecture*, Kluwer Academic Publishers, pp. 35–50.

MEHTA, N., MEDVIDOVIC, N., AND PHADKE, S. 2000. Towards a taxonomy of software connectors. In *Proceedings of the 22nd International Conference on Software Engineering*. ACM, New York, pp. 178–187.

MORICONI, M. AND QIAN, X. 1994. Correctness and composition of software architectures. In *Proceedings of the 2nd Symposium on the Foundations of Software Engineering*. ACM, New York, pp. 164–174.

O'MALLEY, S. W. AND PETERSON, L. L. 1992. A dynamic network architecture. *ACM Trans. Comput. Syst. 10*, 2, 110–143.

PERRY, D. AND WOLF, A. 1992. Foundations for the study of software architectures. *ACM SIGSOFT Softw. Eng. Notes 17*, 4, 40–52.

SASSONE, V., NIELSEN, M., AND WINSKEL, G. 1993. A classification of models for concurrency. In *CONCUR'93*, E. Best, Ed. Lecture Notes in Computer Science, vol. 715. Springer-Verlag, New York, pp. 82–96.

SHAW, M. 1993. Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status. In *Proceedings of the Workshop on Studies of Software Design* (May).

SHAW, M., DELINE, R., KLEIN, D. V., ROSS, T. L., YOUNG, D. M., AND ZELESNIK, G. 1995. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng. 21*, 4 (Apr.), 314–335.

SPITZNAGEL, B. AND GARLAN, D. 2001. A compositional approach for constructing connectors. In *The Working IEEE/IFIP Conference on Software Architecture (WICSA'01).* Royal Netherlands Academy of Arts and Sciences Amsterdam, The Netherlands.

WERMELINGER, M., LOPES, A. AND FIADEIRO, J. L. 2000. Superposing connectors. In *Proceedings of the 10th International Workshop on Software Specification and Design*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 87–94.

WERMELINGER, M., LOPES, A., AND FIADEIRO, J. L. 2001. A graph based architectural (re)configuration language. In *Proceedings of ESEC/FSE'01*. ACM, New York, pp. 21–32.

WERMELINGER, M. AND OLIVEIRA, C. 2002. The CommUnity workbench. In *Proceedings of the 24th International Conference on Software Engineering* (May), ACM, New York, p. 713.

Received November 2001; revised October 2002; accepted May 2003