# A Compositional Approach to Connector Construction*

Antónia Lopes[1], Michel Wermelinger[2,3], and José Luiz Fiadeiro[1,3]

[1] Department of Informatics, Faculty of Sciences, University of Lisbon, Campo Grande, 1749-016 Lisboa, Portugal
mal@di.fc.ul.pt
[2] Department of Informatics, Faculty of Sciences and Technology, New University of Lisbon, 2829-516 Caparica, Portugal
[3]ATX Software SA
Alameda António Sérgio 7, 1A, 2795-023 Linda-a-Velha, Portugal
mw@di.fct.unl.pt,jose@fiadeiro.org

**Abstract.** We develop a notion of higher-order connector towards supporting the systematic construction of architectural connectors for software design. The idea is that individual properties of connectors, such as security and fault-tolerance, can be designed separately as higher-order connectors. Complex connectors can then be described as suitable combinations of higher-order connectors and basic connectors. We use CommUnity, a Unity-like parallel program design language that we have been using for formalising aspects of architectural design, for illustrating our approach to the compositional construction of connectors and also to motivate the categorical semantics of higher-order connectors that we propose.

## 1 Introduction

Although components have always been considered the fundamental building blocks of software systems, the way the components of a system interact may also be determinant on the system properties. Recently, component interactions were recognised also to be first-class design entities, and architectural connectors have emerged as a powerful tool for supporting the description of these interactions. However, as argued in [11], the current level of support for connectors is still far from the one components have.

At an architectural level of design, component interactions can be very simple (for instance a shared variable) but also very complex (for instance database-acessing protocols). Hence, it is of great interest to have mechanisms for designing connectors in an incremental and compositional way as well as principled ways of extending existing ones, promoting reuse. Furthermore, as argued in [3], modularising the different kinds

* This research was partially supported by Fundação para a Ciência e Tecnologia through project POSI/32717/00 (FAST — Formal Approach to Software Architecture).

of services involved in interaction protocols 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 work 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*. More concretely, we propose a notion of higher-order connector — a connector that takes a connector as parameter — through which 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.

We follow Garlan's proposal of considering higher-order connectors as operators with which new connectors can be built up from old connectors [8,9]. 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 a 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.

This notion of higher-order connector extends our preliminary proposal presented in [12]. Furthermore, we not only develop this notion over a specific language — CommUnity, but also delineate a notion of higher-order connector which is not specific to any Architecture Description Language. Such generalisation capitalizes on our previous work [4,5,6,13] on the formal underpinning of connectors, namely on the categorical framework that establishes the semantics of architectural connectors independently of specific choices of design languages and behavioural models. Due to space limitation, there are some important details of the categorical semantics of higher-order connectors that will not be discussed.

We start by presenting the program design language CommUnity, a Unity-like [2] parallel program design language that we have been using for formalising aspects of architectural design. In this paper, we will use CommUnity to illustrate the notion of higher-order connector we wish to put forward and also to show how categorical techniques can be used to formalise a design formalism. This is important because the formal semantics we shall present for connectors and higher-order connectors relies on the fact that category theory provides a convenient framework for describing a design formalism, namely designs, configurations and relationships between designs, such as refinement.

## 2 CommUnity

CommUnity is a program design language in the style of Unity [2], that also combines elements from IP (Interacting Processes) [7]. The language is independent of the actual data types used and, hence, we assume there are pre-defined sorts and functions given by a fixed algebraic signature. For the purposes of examples, we assume this algebraic signature contains sorts *bool*, *nat*, *list*, *queue* with the usual operations; ordered pairs with projection functions *fst* (first element) and *snd* (second element); a function *if(cond,then-expr,else-expr)* with the obvious meaning.

### 2.1 Component designs

A CommUnity design is of the form

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

where

- $V$ is the set of *variables* that can be declared as *input*, *output* or *private*. Input variables are read from the environment of the component but cannot be modified by the component. Output and private variables are local to the component, i.e., they cannot be modified by the environment. We use *loc(V)* to denote the set of local variables. Output variables can be read by the environment but private variables cannot. Each variable *v* is typed with a sort *sort(v)*.
- $\Gamma$ is the set of *action names*. Actions can be declared either as *private* or *shared*. Private actions represent internal computations and their execution is uniquely under the control of the component. In contrast, shared actions represent interactions between the component and the environment and their execution is also under the control of the environment.
- For each action *g*, *D(g)* consists of the local variables that action *g* can change – its write frame. For every local variable *v*, we also denote by *D(v)* the set of actions that can change *v*.
- For each action *g*, *L(g)* and *U(g)* are two conditions such that $U(g) \supset L(g)$. They establish an interval in which the enabling condition of *g* must lie: *L(g)* is the lower bound of the interval, i.e., it is implied by the enabling condition; *U(g)* is the upper bound, i.e., it implies the enabling condition. Therefore, the negation of *L(g)* establishes a *blocking* condition, whereas *U(g)* establishes a *progress* condition. The enabling condition is fully determined when *L(g)* and *U(g)* are equivalent, in which case we write only one condition.
- For each action *g* and variable *v* in *D(g)*, *F(g,v)* is a expression that denotes a set – the set of values that can be assigned to *v*. We abbreviate deterministic assignments

of the form $v:\in\{t\}$ as $v:=t$. When an action has empty domain, we use the expression `skip` instead.

When, for every $g \in \Gamma$, $L(g)$ and $U(g)$ coincide, and $F(g,v)$ is a singleton for every local variable $v$, then the design is called a *program*. The behaviour of a program without input variables is as follows. At each execution step, one of the actions whose enabling condition holds of the current state is selected, and its assignments are executed atomically in parallel. Furthermore, private actions that are infinitely often enabled are guaranteed to be selected infinitely often.

As an example consider the design presented below that 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 modelled by the execution of *hreq*, turns on the light, which is modelled by variable *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 modelled by the execution of the private action *read*, and made available for transmission in the output variable *data*. The private variable *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:nat
out   data:nat, off: bool
prv   rd: bool
do    hreq: off → off:=false
[] prv read: ¬rd ∧ ¬off →data:=sensor‖ rd:=true
[]    hack: rd → rd:=false ‖ off:=true
```

## 2.2 Configurations

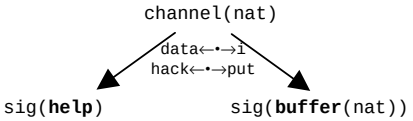
In CommUnity it is also possible to describe a system as the interconnection of a number of interacting component designs by defining a configuration. The model of interaction between components is based on action synchronisation and the interconnection of input variables of a component with output variables of other components. Although these are common forms of interaction, CommUnity requires interaction between components — name bindings — to be made explicit in 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.

Let us consider, for instance, that we want to describe a system in which the messages from the *help* component are sent (to a receiver) through a bounded channel. The design *buffer* parametrised by sort *s* described below models the bounded channel, more concretely it models a bounded buffer with a FIFO discipline.

```
design buffer[s] is
in    i:s
out   o:s
prv   rd:bool; q:queue(K,s)
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 *s*, received from the environment through the input variable *i*, as long there is space for them. It can also make stored messages available to the environment through the output variable *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 modelled by the action *get* and the private variable *rd*.

In order to establish that messages from the *help* component are sent (to a receiver) through a bounded channel, we consider the following configuration




where we use ***sig****(P)* to denote the signature of *P*, and *channel(nat)* is a signature that consists of an input variable of sort *nat* and a shared action. The names of this variable and of this action are not relevant: they are only placeholders for the name bindings.

In this configuration, the input variable of *channel* is mapped to the output variable *data* of the *help* component and to the input variable *i* of *buffer*. This establishes an I/O interconnection between *help* and *buffer*. Moreover, the actions *hack* of *help* and *put* of *buffer* are mapped to the shared action of *channel*. This defines that *help* and *buffer* must synchronise each time either of them wants to perform the corresponding action.

Signatures and signature morphisms can be formally defined as follows.

*A design signature is a tuple <V,Γ,tv,ta,D> where*
- *V is an S-indexed family of mutually disjoint finite sets,*
- *Γ is a finite set,*
- *tv: V→{out,in,prv} is a total function,*
- *ta: Γ→{sh,prv} is a total function,*
- *D: Γ→$2^{loc(V)}$ is a total function.*

*A morphism $\sigma$: $\theta_1 \to \theta_2$ is a pair $<\sigma_{var},\sigma_{ac}>$ where*
- *$\sigma_{var}$: $V_1 \to V_2$ is a total function satisfying:*
  - ***1.*** *$sort_2(\sigma_{var}(v))=sort_1(v)$ for every $v \in V_1$;*
  - ***2.*** *$\sigma_{var}(o) \in out(V_2)$ for every $o \in out(V_1)$;*
  - ***3.*** *$\sigma_{var}(i) \in out(V_2) \cup in(V_2)$ for every $i \in in(V_1)$;*
  - ***4.*** *$\sigma_{var}(p) \in prv(V_2)$ for every $p \in prv(V_1)$.*
- *$\sigma_{ac}$: $\Gamma_2 \to \Gamma_1$ is a partial mapping satisfying for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined:*
  - ***5.*** *if $g \in sh(\Gamma_2)$ then $\sigma_{ac}(g) \in sh(\Gamma_1)$;*
  - ***6.*** *if $g \in prv(\Gamma_2)$ then $\sigma_{ac}(g) \in prv(\Gamma_1)$;*

7. $\sigma_{var}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$;
8. $\sigma_{ac}(D_2(\sigma_{var}(v))) \subseteq D_1(v)$ *for every* $v \in loc(V_1)$.

*Design signatures and signature morphisms constitute a category* ***SIGN****.*

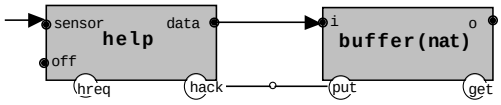
A morphism $\sigma: \theta_1 \rightarrow \theta_2$ identifies a way in which a component with signature $\theta_1$ is embedded in a larger system with signature $\theta_2$. More concretely, $\sigma_{var}$ identifies for each variable of the component the corresponding variable of the system, and $\sigma_{ac}$ identifies the action of the component that is involved in each action of the system, if ever. Notice that input variables of a component may become output variables of the system (condition 3) because we consider that the result of interconnecting an input variable of a component with an output variable of another component is an output variable of the system. Conditions 7 and 8 require that change within a component is completely encapsulated in the structure of actions defined for the component.

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

- An output variable of a component cannot be connected (directly or indirectly through input variables) with output variables of the same or other components.
- Private variables and private actions cannot be involved in the connections.

The second rule ensures that private variables cannot be read by the environment and that the execution of private actions is uniquely under the control of the component.

In the sequel, for simplicity, rather than using diagrams involving signatures and signature morphisms, we will mainly use a more user-friendly notation. For instance, the configuration presented previously could be described as follows.




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

### Configuration Semantics

The semantics of configuration diagrams 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 [2,10].

*A superposition morphism* $\sigma: P_1 \rightarrow P_2$ *consists of a signature morphism* $\sigma: \theta_1 \rightarrow \theta_2$ *s.t., for every* $g \in \Gamma_2$ *s.t.* $\sigma_{ac}(g)$ *is defined:*

1. *for every* $v \in D_1(\sigma_{ac}(g)), \Phi \models (F_2(g,\sigma_{var}(v)) \subseteq \sigma(F_1(\sigma_{ac}(g),v)))$;
2. $\Phi \models (L_2(g) \supset \sigma(L_1(\sigma_{ac}(g))))$;
3. $\Phi \models (U_2(g) \supset \sigma(U_1(\sigma_{ac}(g))))$;

*where* $\models$ *denotes validity in the first-order sense. Designs and superposition morphisms constitute the 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 behaviour. The conditions say that the effects of the actions have to be preserved or made more deterministic and that the bounds for the enabledeness of each action cannot be weakened. Strengthening of the lower bound is typical of superposition and 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 be performed. On the other hand, it is clear that progress for a joint action can only be guaranteed when all the components involved can locally guarantee so.

Any diagram ***D*** in ***SIGN*** that establishes the interactions between a given set of components can be trivially lifted to a diagram ***D'*** of designs and superposition morphisms: the signature of each design is replaced by the design itself; every channel *ch* in ***D*** is replaced by ***dsgn****(ch)*— the design with signature *ch*, tautological bounds for the enabledeness of each action and the least deterministic assignment for the variables in the write frame of each action ($v: \in sort(v)$). Defined in this way, ***dsgn****(ch)* is a design that is "neutral" with respect to the establishment of superposition morphisms in the sense that every signature morphism $\sigma: ch \rightarrow \mathbf{sig}(P)$ defines a superposition morphism $\sigma: \mathbf{dsgn}(ch) \rightarrow P$. For simplicity, we will continue to use *ch* instead of ***dsgn****(ch)* in the configuration diagrams.

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 ***D'*** in the category ***c-DSGN***. Very roughly, the colimit “merges” the input variables identified with an output variable into that output variable, it establishes the synchronisation sets and it assigns to each of these sets an action whose bounds for enabledeness are the conjunction of the local bounds of each action in the set and whose assignments are the parallel composition of the assignments performed locally by each action in the set. In the next section we will present an example of the colimit construction.

### 2.3 Refinement

CommUnity supports several mechanisms for underspecification which can be reduced or eliminated through refinement. Concretely, in CommUnity designs, actions may be underspecified in the sense that their enabling conditions may not be fully determined and subject to refinement by reducing the interval established by $L$ and $U$, and their effects on the variables may be undetermined and also subject to refinement by replacing the assignment sets by proper subsets.

Refinement of CommUnity designs can be modelled by the following morphisms.

*A refinement morphism* $\sigma: P_1 \rightarrow P_2$ *consists of a signature morphism* $\sigma: \theta_1 \rightarrow \theta_2$ *s.t.:*

**1.** $\sigma_{var}(inp(V_1)) \subseteq inp(V_2)$ *and* $\sigma_{var}{\downarrow}(out(V_1) \cup inp(V_1))$ *is injective;*

**2.** $\sigma^{-1}_{ac}(g) \neq \emptyset$, $g \in sh(\Gamma_1)$;

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

**3.** *for every* $v \in D_1(\sigma_{ac}(g)), \Phi \models (F_2(g, \sigma_{var}(v)) \subseteq \sigma(F_1(\sigma_{ac}(g), v)))$;

*4.* $\Phi \models (L_2(g) \supset \sigma(L_1(\sigma_{ac}(g))))$.

*For every* $g_1 \in \Gamma_1$:

**5.** $\Phi \models (\sigma(U_1(g_1)) \supset \bigvee_{\sigma_{ac}(g_2)=g_1} U_2(g_2))$.

*Designs and refinement morphisms constitute the category* ***r-DSGN***.

A refinement morphism supports the identification of a way in which a design $P_1$ is refined by another design $P_2$. Each variable of $P_1$ has a corresponding variable in $P_2$ and each action $g$ of $P_1$ is implemented by the set of actions $\sigma^{-1}_{ac}(g)$ in the sense that $\sigma^{-1}_{ac}(g)$ is a menu of refinements for action $g$. The actions for which $\sigma_{ac}$ is left undefined (the new actions) and the variables which are not in $\sigma_{var}(V_1)$ (the new variables) introduce more detail in the refined description of the component.

Condition 1 ensures that an input variable cannot be made local by refinement and that different variables of the interface cannot be collapsed into a single one (refinement does not alter the border between the system and its environment). Condition 2 ensures that those actions that model interaction between the design and its environment have to be implemented. Conditions 4 and 5 state that the "interval" of (allowed) non-determinism defined by the two bounds for enabledeness can only be preserved or reduced by refinement (refinement, pointing in the direction of an implementation, should reduce allowed non-determinism). The non-determinism of assignments must be preserved or decreased (condition 3).

For instance, the *help* design presented previously is a refinement of a generic sender of messages.

```
design  sender[s] is
out       o:s
prv       rd:bool
do  prv   prod: ¬rd,false→o:∈s ‖ rd:=true
 []       send: rd,false→rd:=false
```

Notice that, in this design, it is left unspecified when and how many messages the sender will send as well as the discipline of production. The underlying refinement morphism is

$\eta: sender(nat) \rightarrow help \qquad \eta(o)=data, \eta(rd)=rd, \eta(read)=prod, \eta(send)=hack$

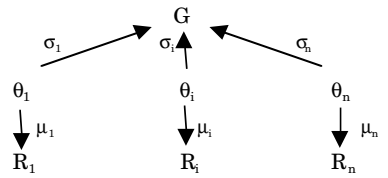
## 3 Architectural Connectors

Software Architecture has put forward connectors as first-class entities for modelling interactions between systems components. According to [1], 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, i.e., it determines the obligations that they have

to fulfill to become instances of the roles. The glue describes how the activities of the role instances are coordinated.

Using the mechanisms that we have just described for configuration design in CommUnity, it is not difficult to come up with a formal notion of connector:

– *A connection consists of*
  - *two designs G and R, called the glue and the role of the connection, respectively;*
  - *a signature $\theta$ and two superposition morphisms $\sigma$:**dsgn**($\theta$)→G, $\mu$:**dsgn**($\theta$)→R 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.*




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

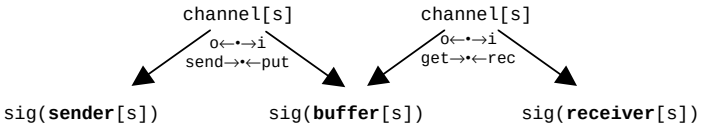
For instance, asynchronous communication of values of type *s* through a bounded channel can be modelled by a connector *Async* with two roles — *sender* and *receiver*. These roles define the behaviour required of the components to which the connector can be applied. For the *sender* we require that it does not produce another message before the previous one has been processed. After producing a message, the *sender* should expect an acknowledgement to produce a new message. For the *receiver*, we simply require that it has an action that models the reception of a message. In CommUnity, the role *sender* is modelled by the design *sender[s]* defined previously and the *receiver* can be designed as follows.

```
design receiver[s] is
in     i:s
do     rec: true,false→skip
```

In order to leave unspecified when and how many messages the receiver will receive the progress guard of *rec* is false (in this way, the enabling condition can be as strong as we wish).

The glue of *Async* is the design *buffer[s]* presented previously. It 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.

Finally, the configuration below establishes how the roles and the glue of *Async* are connected.




The left-hand side morphisms define that *sender* and *buffer* must synchronise on actions *send* and *put*, and establish the interconnection of the output variable *o* of *sender* with the input variable *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 variable *o* of *buffer* with the input variable *i* of *receiver*.

The semantics of connector *Async* is given by the colimit of its configuration diagram, which returns the design given, up to an isomorphism, by

```
design async[s] is
out      o_s, o_b : s
prv      rd_s, rd_r : bool;  q : queue(K,s)
do prv   prod : ¬rd_s,  false → o_s :∈ s ‖ 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 *Async* defines to be derived. For instance, reasoning about this design, it is possible to conclude that no messages are lost and 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.

The connectors we have described so far are connector *types* in the sense that they can be instantiated. More concretely, the roles of a connector type can be instantiated with specific designs. Role instantiation has to obey a compatibility requirement expressed via the refinement relationship and, hence, an instantiation of a connector can now be defined as follows:

- *An instantiation of a connection with role R consists of a design P together with a refinement morphism $\eta: R \rightarrow P$.*
- *An instantiation of a connector consists of an instantiation for each of its connections.*

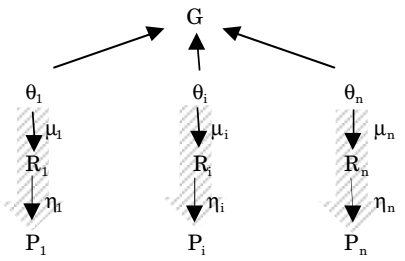



**Figure 1**

– *The semantics of a connector instantiation is the colimit of the diagram in* ***c-DSGN*** *formed as described above by composing the role morphism of each connection with its instantiation.*

For instance, the connector *Async* can be used to interconnect the *help* component with a given receiver. This can be achieved by instantiating the role *sender* of *Async* with *help*, through the refinement morphism $\eta$ defined previously.

It is important to notice that the formalisation of architectural connectors we presented is not specific to any design formalism insofar as the involved categories are not fixed. However, there are some properties that a design formalism needs to satisfy to support such architectural concepts. For instance, the semantics of a connector instantiation relies on the possibility of composition of the role morphism of each connection with its instantiation.

This property and other properties that a design formalism need to satisfy to support such architectural concepts are discussed in detail in [4]. Herein, we shall simply explain why it is possible to compose a role morphism with its instantiation in the case of CommUnity.

In CommUnity, each instantiation $\eta:R\rightarrow P$ of a connection is composable with $\mu$ in order to define $\mu;\phi:\theta\rightarrow\mathbf{sig}(P)$ because $\theta$ is according to the rules that define well-formed configurations and, hence, has no private variables, which means that at the level of signatures the refinement morphism has the same properties as a composition morphism over $\theta$. As we have already argued in section 2.2, every such signature morphism can be lifted to a design morphism $\mu;\phi:\mathbf{dsgn}(\theta)\rightarrow P$. Hence, an instantiation of a connector defines a diagram in ***c-DSGN*** that connects the role instances to the glue. Moreover, because each connection is according to the rules set for well-formed configurations as detailed in the previous section, the diagram defined by the instantiation is, indeed, a configuration and, hence, has a colimit.

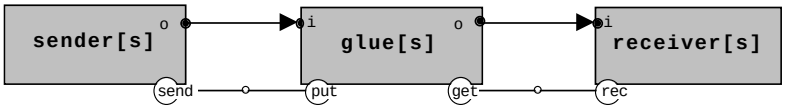
In the sequel, we will use $C+(\eta_i)$ to denote the configuration depicted in figure 1, obtained by composing the role morphism of each connection $i$ of $C$ with its instantiation $\eta_i$.

## 4 Higher-order Architectural Connectors

As explained before, it is important that connectors can be designed compositionally, by combining different interaction capabilities. In particular, it is important to have principled forms of adapting connectors to new situations, for instance in order to incorporate compression, fault-tolerance, security, monitoring, etc.

Let us consider for instance *compression*. In this case, the goal is to adapt a connector that represents a unidirectional communication protocol in order to compress data for transmission in a transparent way.

A generic unidirectional communication protocol can be modelled by the binary connector *Uni-comm[s]*




where

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

It is not difficult to realize that this form of coordination of the sender and receiver activities, embodied by the glue of the new connector, can be obtained by instantiating the sender role of *Uni-comm* with a component *comp* that compresses messages before it transmits them, and by instantiating the sender role of *Uni-comm* with a component *decomp* that decompresses the message it receives.

Such components can be designed in CommUnity as follows

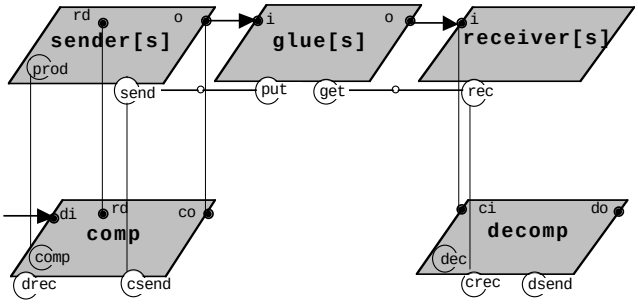
```
design comp is
in     di:t
out    co:s
prv    v:t; rd,msg:bool
do     drec: ¬msg → v:=di‖ msg:=true
[] prv comp:¬rd∧ msg → co:=comp(v)‖ rd:=true
[]     csend:rd → rd:=false‖ msg:=false

design decomp is
in     ci:s
out    do:t
prv    v:s; rd,msg:bool
do     crec: ¬msg → v:=ci‖ msg:=true
[] prv dec:¬rd∧msg → do:=decomp(v)‖ rd:=true
[]     dsend: rd → rd:=false‖ msg:=false
```

and the instantiation of *Uni-comm* with *comp* and *decomp* can be described by the refinement morphisms

$\eta^*_s$*:sender(s)→comp* $\eta^*_s(o)=co, \eta^*_s(rd)=rd, \eta^*_s(comp)=prod, \eta^*_s(csend)=send$

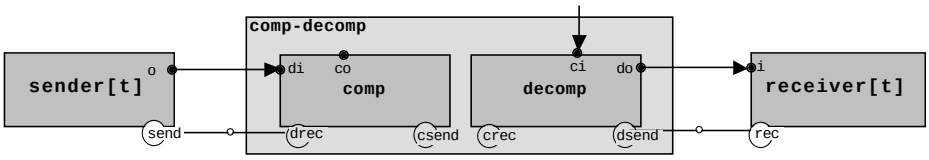
$\eta^*_r$*:receiver(s)→decomp* $\eta^*_r(i)=ci, \eta^*_r(crec)=rec$

In this way, the glue of the new connector is given by




where hairlines were used to represent the refinement relationship.

It remains to define that, in the new connector, the messages sent by the sender are received by the *comp* component and that *decomp* delivers the decompressed messages to the receiver. The connector *Compression* depicted below models this form of coordination.



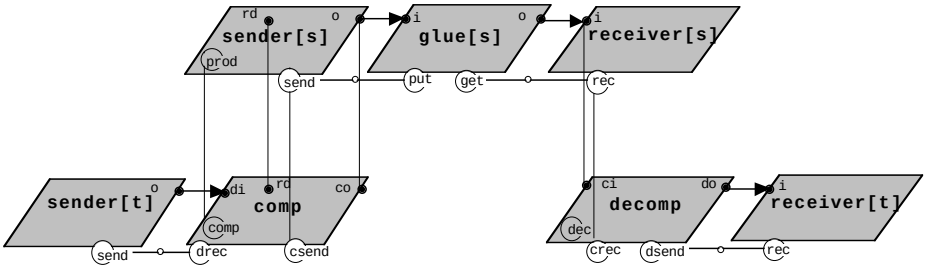

Notice that the glue, *comp-decomp*, is defined in terms of a configuration where *comp* and *decomp* do not interact.

In summary, the procedure we described for installing the compress/decompress service over *Uni-comm*, that models a generic unidirectional communication protocol, is described by

- the connector *Compression;*
- the refinement morphisms $\eta_s$*:sender(s)*$\rightarrow$*comp-decomp* and $\eta_r$*:receiver(s)* $\rightarrow$ *comp-decomp* induced, respectively, by $\eta^*_s$ and $\eta^*_r$ (because *comp* and *decomp* do not interact, any component refined by one of them is also refined by their composition).

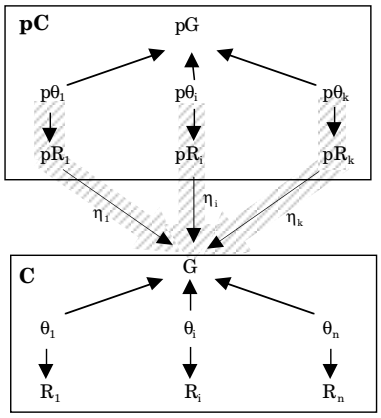
It is not difficult to realise that this procedure can be applied to more specific connectors provided they model unidirectional communication protocols. In fact, we may regard *Uni-comm* as a formal parameter of this description, that can be instantiated with different connectors. The connector *Compression* can be regarded as the body of the definition, that models the nature of the service that is superposed on instantiation of the formal parameter. Finally, the refinement morphisms establish a relationship between the parameter and the body connectors.

Putting the three previous pictures together we get a graphical representation of this parameterised entity—*Compression(Uni-comm)*, and is an example of what we call a higher-order connector.



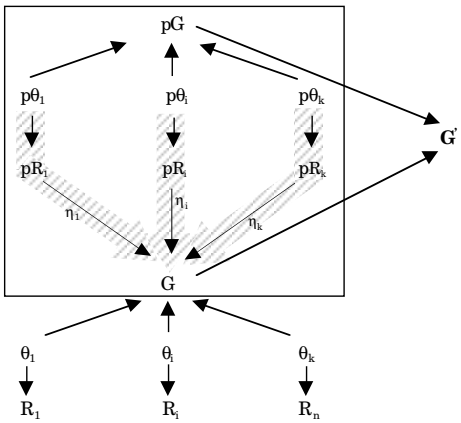

Higher-order connectors can be formally defined as follows.

– *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, i.e., a refinement morphism $\eta_i$ from each of the parametric roles to the glue, such that the diagram in* ***c-DSGN*** *obtained by composing the role morphism of each parametric connection with its instantiation*



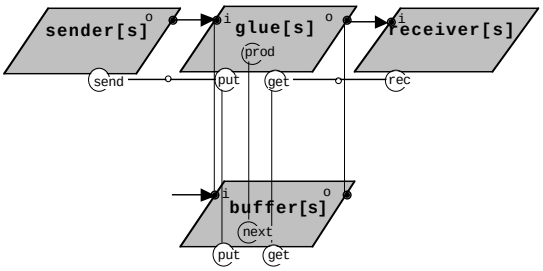

*constitutes a well-formed configuration.*

– *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+($\eta_i$), obtained by composing the role morphism of each parametric connection with its instantiation.*

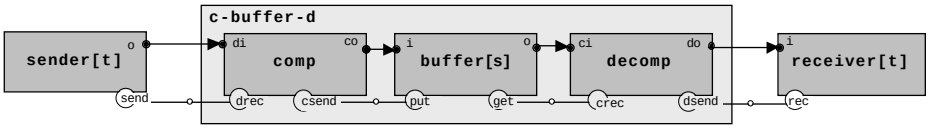



Now it remains to explain the procedure of parameter passing, i.e., how a higher-order connector can be applied to a specific connector and how the resulting connector is obtained.

Let us suppose that we want to install the compression service over the *Async* connector. In this case, it is not difficult to realize that we may replace the formal parameter of *Compression(Uni-comm*) 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.




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 $\eta_s$ and $\eta_r$ that define the instantiation of *Uni-comm* with *comp* and *decomp*, respectively. For example, variable *co* of *comp* becomes connected to the input variable *i* of *buffer* because *co* corresponds to the variable *o* of *sender* which in turn is, in *Async*, connected to *i*. The resulting configuration fully defines the connector *Compression(Async)*. Its roles are *sender* and *receiver* and its glue — *c-buffer-d*, is defined in terms of a configuration involving *comp*, *decomp* and *buffer* as shown below.



In this example, the formal and the actual connectors have exactly the same roles and the instantiation of the higher-order connector is established merely by a refinement morphism from the glue of the former to the glue of the later. However, the instantiation of a higher-order connector can be defined in a more general way, through the definition of a notion of fitting morphism between connectors. The instantiation of a higher-order connector is then established by a fitting morphism from the formal to the actual connector.

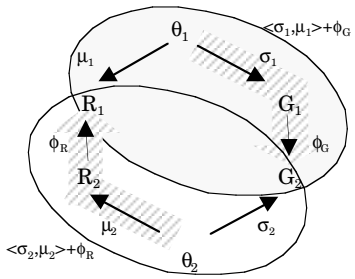
We shall now discuss the suitable notion of fitting morphism between connectors. Because the formal parameter of a higher-order connector defines the kind of connectors to which the higher-order connector can be applied, its instantiation has to obey a compatibility requirement. Intuitively, a fitting morphism from a connector $C_1$ to a connector $C_2$ must express that it is possible to use, in the design of a given system, $C_2$ in place of $C_1$ in the sense that the functionality of the system is preserved.

We first notice that, for this be possible, the two connectors must have the same number of roles. Furthermore, $C_2$ has to admit to be instantiated with the same components than $C_1$. That is to say, every restriction on the components to which $C_2$ can be applied must also be a restriction imposed by $C_1$. In this way, fitting morphisms must establish a correspondence between the roles of $C_1$ and $C_2$ and must require that each of the roles of $C_2$ is refined by the corresponding role of $C_1$.

We have seen that 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 in some extent the type of interconnection that the connector will ensure. The type of interconnection is clearly preserved if we simply consider a less unspecified glue, i.e., 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 $\phi$ from a connection $<\sigma_1$: **dsgn**$(\theta_1)\rightarrow G_1,\mu_1$:**dsgn**$(\theta_1)\rightarrow R_1>$ to a connection $<\sigma_2$:**dsgn**$(\theta_2)\rightarrow G_2,\mu_2$:**dsgn**$(\theta_2)\rightarrow R_2>$ consists of a pair $<\phi_G$: $G_1\rightarrow G_2,\phi_R$:$R_2\rightarrow R_1>$ of refinement morphisms s.t. the interconnection $<\sigma_1,\mu_1>+\phi_G$ of $R_1$ with $G_2$ is refined by the interconnection $<\sigma_2,\mu_2>+\phi_R$ (that is to say, there exists a refinement morphism from the colimit $<\sigma_1,\mu_1>+\phi_G$ of to the colimit of $<\sigma_2,\mu_2>+\phi_R$).*




– *A fitting morphism $\phi$ from a connector $C_1$ to a connector $C_2$ with the same number of connections consists of a fitting morphism $\phi$ from each of $C_1$'s connections to each of $C_2$'s connections, all with the same glue refinement $\phi_G$.*

Based on these fitting morphisms between connectors, we may finally define the instantiation of a higher-order connector.

– *An instantiation of a higher-order connector with formal parameter pC (figure 2) consists of a connector $C^A$ (the actual parameter) together with a fitting morphism $\phi: pC \rightarrow C^A$, such that the diagram in* ***c-DSGN*** *obtained by composing the role morphisms of each actual connection with the corresponding fitting component and then with the role instantiation (figure 3) constitutes a well-formed configuration.*

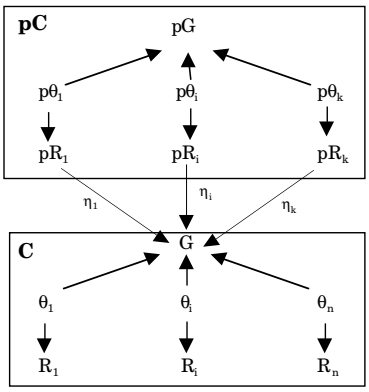



**Figure 2**

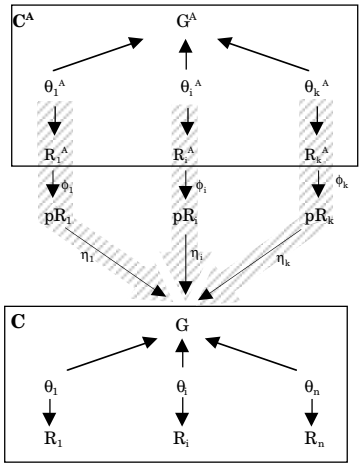



**Figure 3**

– *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 $C^A+(\phi_i;\eta_i)$.*

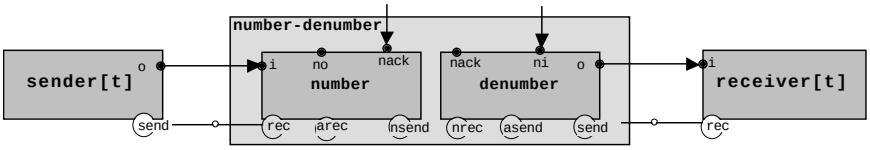
For simplicity, we defined higher-order connectors with one parameter only. However, the definition can be extended to the case of several parameters in a straightforward way. We end this section by presenting an example of a higher-order connector with two parameters that models a fault-tolerance service. Specifically, we aim to describe a way of combining two unidirectional communication protocols in order to obtain a unidirectional communication protocol that provides reliable and in order message delivery, in the presence of messages loss and duplication faults.

The idea behind this fault-tolerance service is that messages must be numbered before they are transmitted and then must be de-numbered before they are delivered to the receiver. Moreover, the messages that arrive out-of-order must be ignored (not transmitted to the receiver) and in response to a message an acknowledgment with its number must be transmitted.

This service can be modelled by a higher-order connector with two parameters. One represents the connector used to transmit the messages (after being numbered) and the other represents the connector used to transmit the acknowlegment messages (which in fact will be natural numbers) in the opposite direction. Clearly, each of these connectors should model a unidirectional communication protocol.

The higher-order connector itself, *Ft(Uni-comm[t],Uni-comm[t*nat])*, consists of

– the connector *Numbering* defined by




where the glue, *Number-Denumber*, is defined in terms of a configuration with the following two components:

```
design number is
in    i:t,nack:nat
out   no:t*nat
prv   b: list(t), current:t+null, k:nat, rd: bool
do    rec: ¬full(b)→b:=b.i
[] prv take:¬empty(b)∧current=null→current:=hd(b)‖b:=tl(b)‖k:=k+1
[] prv nprod: ¬rd∧current≠null → rd:=true‖no:=<current,k>
[]    nsend: rd → rd:=false
[]    arec: true → current:=if(nack=k,null,current)

design denumber is
in    ni:t*nat
out   nack:nat, o:t
prv   b: list(t), n,k:nat, rd_a,rd_s: bool
do    nrec: ¬rd_a∧¬full(b)→ b:=if(snd(ni)=k+1,b.fst(ni),b)
                          ‖ k:=if(snd(ni)=k+1,k+1,k)‖ n:=snd(ni)
[] prv aprod: ¬rd_a → nack:=n ‖ rd_a:=true
[]    asend: rd_a → rd_a:=false
[]    prod: ¬empty(b)∧¬rd_s → o:=hd(b)‖b:=tl(b)‖rd_s:=true
[]    send: rd_s → rd_s:=false
```

Component *number* numbers the messages to be transmitted, sending repeatedly the same message until an acknowledgement with its number is received. Meanwhile, messages to be transmitted are stored in a buffer. Component *denumber* receives numbered messages through *ni* and transmits through *nack* an acknowlegment with the message number. Meanwhile, it stores the messages that arrive in order and delivers them to the receiver.

– the connectors *Uni-comm[t*nat]* and *Uni-comm[nat]* — the two formal parameters;

– the refinement morphisms

$\eta_s$*:sender(t*nat)→number-denumber,* $\eta_r$*:receiver(t*nat)→number-denumber*
$\kappa_s$*:sender(nat)→number-denumber,* $\kappa_r$*:receiver(nat)→number-denumber*

induced respectively by

$\eta^*_s: sender(t*nat) \rightarrow number$
$\quad \eta^*_s(o)=no, \eta^*_s(rd)=rd, \eta^*_s(nprod)=prod, \eta^*_s(nsend)=send$
$\eta^*_r: receiver(t*nat) \rightarrow denumber$
$\quad \eta^*_r(i)=ni, \eta^*_r(nrec)=rec$
$\kappa^*_s: sender(nat) \rightarrow denumber$
$\quad \kappa^*_s(o)=nack, \kappa^*_s(rd)=rd_a, \kappa^*_s(aprod)=prod, \kappa^*_s(asend)=send$
$\kappa^*_r: receiver(nat) \rightarrow number$

$$\kappa^*_r(i)=nack, \kappa^*_r(arec)=rec$$

It is important to notice that the two parameters of the higher-order connector *Ft(Uni-comm[t],Uni-comm[t*nat])* can be instantiated with two different connectors but they can also be both instantiated with the same connector, provided this connector encompasses both capabilities — transmission of numbered messages in one direction and transmission of natural number in the other direction.

## 5 Conclusions

In this paper we proposed a notion of higher-order connector and showed that this abstraction indeed facilitates the separation of concerns in the development of complex connectors and their compositional construction. In particular, we have shown how a compression service and fault-tolerance service can be modelled separately as higher-order connectors. Furthermore, we continued our previous work on the use of Category Theory to formalise key notions of software architecture. Building on the categorical semantics for the notion of architectural connector, we formalised higher-order connectors and established their categorical semantics.

Our definitions agree with Garlan's original proposal [8] 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, Garlan and Spitznagel [9] propose that a connector transformation be modelled 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 formalise these ideas in the context of a particular ADL, namely Wright, relying on the specific language and semantics of CSP.

We showed that a transformation of a connector can be modelled by a parameterised 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 analysed. Although we have used CommUnity to motivate and illustrate our ideas, by adopting a categorical framework for formalising these ideas, we achieved independence relatively to the ADL.

Due to space limitation, we did not address the composition of higher-order connectors but the categorical framework leads naturally to a notion of composition of higher-order connectors that is useful for combining orthogonal properties. Such composition is naturally defined by considering a more general form of instantiation — *parameterised instantiation*, more specifically the instantiation of hocs with hocs. This composition supports the combination of different kinds of functionality, modelled separately by different higher-order connectors, giving rise also to a higher-order connector. In this way, it is possible to analyse the properties that such compositions exhibit, namely to investigate whether undesirable properties emerge and desirable properties are preserved.

As mentioned before, 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. Work reported in [15] addresses 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.

## References


1. R.Allen and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM* TOSEM, 6(3):213-249, July 1997.
2. K.Chandy and J.Misra, *Parallel Program Design - A Foundation*, Addison-Wesley 1988.
3. G.Denker, J.Meseguer and C.Talcott, "Rewriting semantics of meta-objects and composable distributed services", Internal report, Computer Science Laboratory, SRI International, 1999.
4. J.L.Fiadeiro, A.Lopes and M.Wermelinger, "A Mathematical Semantics for Architectural Connectors". Submitted for publication (available at http://www.fiadeiro.org/jose/papers)
5. J.L.Fiadeiro and A.Lopes, "Algebraic Semantics of Coordination, or what is in a signature?", in *AMAST'98*, A.Haeberer (ed), LNCS 1548, Springer-Verlag 1999.
6. J.L.Fiadeiro and A.Lopes, "Semantics of Architectural Connectors", in TAPSOFT'97, LNCS 1214, Springer-Verlag 1997, 505-519.
7. N.Francez and I.Forman, Interacting Processes, Addison-Wesley 1996.
8. D.Garlan, "Higher-Order Connectors", Presented at the Workshop on Compositional Software Architectures, Monterey, CA, January 6-7, 1998.
9. D.Garlan and B.Spitznagel, "Toward compositional construction of complex connectors", Proceedings of the Eighth International Symposium on the Foundations of Software Engineering (FSE-8), November 2000.
10. S.Katz, "A Superimposition Control Construct for Distributed Systems", *ACM TOPLAS* 15(2):337-356, 1993.
11. N.Mehta, N.Medvidovic and S.Phadke, "Towards a taxonomy of software connectors", Proc. of 22nd International Conference on Software Engineering, ACM Press, 2000, 178-187.
12. M.Wermelinger, A.Lopes and J.L.Fiadeiro, "Superposing Connectors", in *Proc. 10th International Workshop on Software Specification and Design*, IEEE Computer Society Press 2000, 87-94. (also available at http://ctp.di.fct.unl.pt/~mw/proj/fast/index.html)
13. M.Wermelinger and J. L. Fiadeiro, "Connectors for mobile programs", IEEE Trans. on Software Eng., 24(5):331--341, May 1998.
14. M.Wermelinger and J.L.Fiadeiro, "Algebraic Software Architecture Reconfiguration", in *Software Engineering – ESEC/FSE'99*, LNCS 1687, pp. 393-409, Springer-Verlag 1999.
15. M.Wermelinger, A.Lopes and J.L.Fiadeiro, "A Graph Based Architectural (Re)configuration Language", Proc. ESEC/FSE'01, ACM Press, 2001. In print..