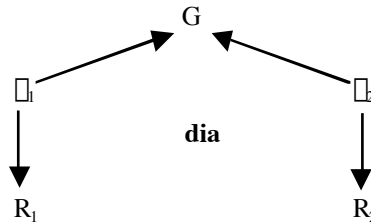


10 AN ALGEBRA OF CONNECTORS

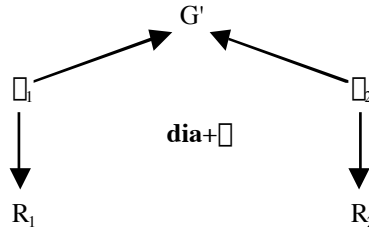
As argued in [86], the current level of support that Architectural Description Languages (ADLs) provide for connector building is still far from the one awarded to components. For instance, although considerable amounts of work can be found on several aspects of connectors [2,14,86], further steps are 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 variable), but they can be very complex as well (e.g., database-accessing 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.

It is not always possible to adapt components to work with existing connectors. Even in those cases where it is feasible, a better alternative may be to modify the connectors because, typically, there are fewer connector types than components types. Moreover, most ADLs either provide a fixed set of connectors or only allow the creation of new ones from scratch, hence requiring from the designer a deep knowledge of the particular formalism and tools at hand. 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 are, of course, among the main issues to be dealt with. Our purpose in this section is to show how typical operators can be given an ADL-independent semantics by formalising them in the categorical framework that we presented in the previous chapter. An example of such an operator was already given in section 9.4. We saw how, given a connector expressed through a configuration diagram **dia**



and a refinement $\Box:G \Box G'$ of its glue, we can construct through **dia**+ \Box another connector that has the same roles as the original one, but whose glue is now G' .



A fundamental property of this construction, given by compositionality, is that the semantics of the original connector, as expressed by the colimit of its diagram, is preserved in the sense that it is refined by the semantics of the new connector. This means that all instantiations of the new connector are refinements of instantiations of the old one. This operation supports the definition of connectors at higher levels of abstraction by delaying decisions on concrete representations of the coordination mechanisms that they offer, thus providing for the definition of specialisation hierarchies of connector types.

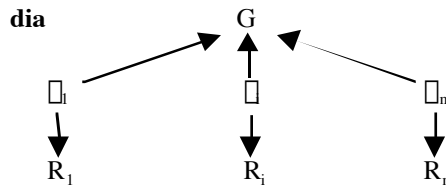
In this chapter we present three connector transformations that operate on the roles rather than the glue. Transformations that, like above, operate at the level of the glue are more sensitive in that they interfere more directly with the semantics of the connector to which they are applied. Hence, they should be restricted to engineers who have the power, and ensuing responsibilities, to change the way connectors are implemented. Operations on the roles are less critical and can be performed more liberally by users who have no access to the implementation (glue). In the last section, we present higher-order mechanisms that can be applied to connectors to obtain other connectors. Throughout the chapter, we will be working over a fixed architectural school $F = \langle \mathbf{c-DESC}, \mathbf{Conf}, \mathbf{r-DESC} \rangle$ (see 9.4.3).

10.1 Three operations on connectors

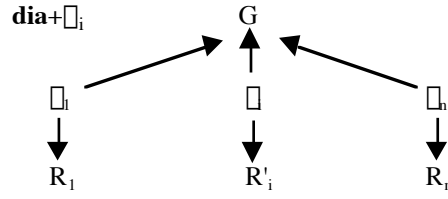
10.1.1 ROLE REFINEMENT

To tailor general-purpose connectors for a specific application, it is necessary to replace the generic roles by specialised ones that can effectively act as “formal parameters” for the application at hand. Role replacement is done in the same way as applying a connector to components: there must be a refinement morphism from the generic role to the specialised one. The old role is cancelled, and the new role morphism is the composition of the old one with the refinement morphism in the sense discussed in section 3.

Given an n -ary connector



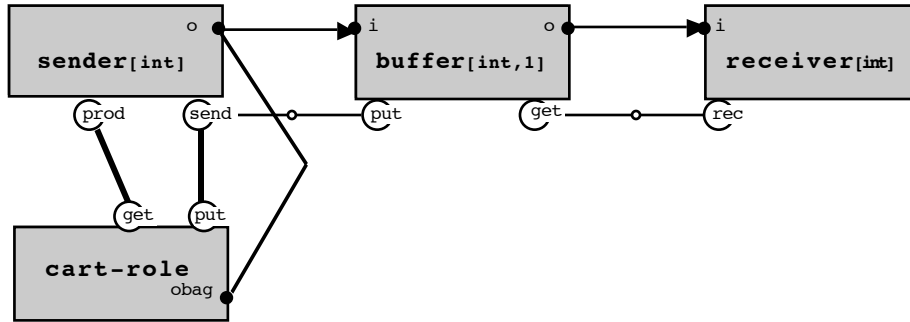
and a refinement morphism $\square_i: R_i \rightarrow R'_i$ for some $0 < i \leq n$, the role refinement operation yields the connector



Notice that this operation can be applied to both abstract and heterogeneous connectors.

This operation preserves the semantics of the connectors to which it is applied in the sense that any instantiation of the refined connector is also an instantiation of the refined one. This is because the refinement morphism that instantiates R'_i can be composed with \square_i to yield an instantiation of R_i .

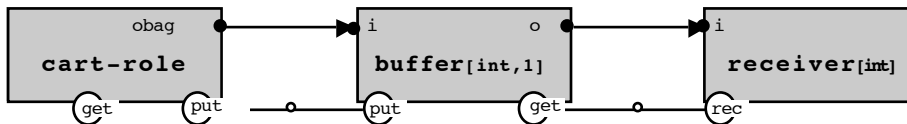
As an example of role refinement, consider the asynchronous connector shown previously. This connector is too general for our luggage distribution service because the sender and receiver roles do not impose any constraints on the admissible instances. We would like to refine these roles in order to prevent meaningless applications to our example like sending the location of the check-in as a bag to the cart.



```

design cart-role is
  out obag: int
  prv dest : -1..U - 1
  do   get[obag]: dest = -1  $\square$  dest' > -1
  []   put: dest > -1, false  $\square$  obag := 0  $\parallel$  dest := -1
    
```

with channel rd of the sender refined by the term $dest \neq -1$. The resulting connector is

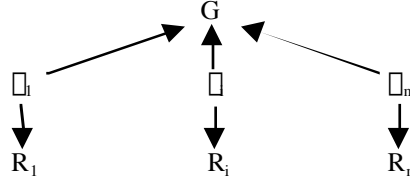


Notice that the invalid combinations are not possible because *cart-role* cannot be refined with a gate or a check-in, nor is it possible to refine *gate-role* with a cart or a check-in. Moreover, the *obag* channel of *cart-role* cannot be refined by channel *laps* of cart.

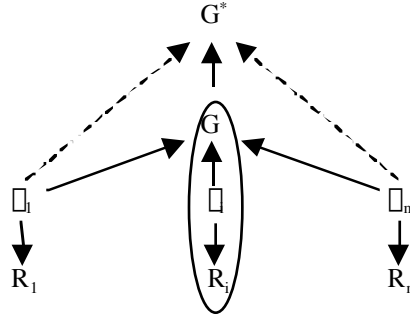
10.1.2 ROLE ENCAPSULATION

To prevent a role from being further refined, the second operation we consider, when executed repeatedly, decreases the arity of a connector by encapsulating some of its roles, making the result part of the glue. This operation requires that the glue and the encapsulated role be in the same formalism.

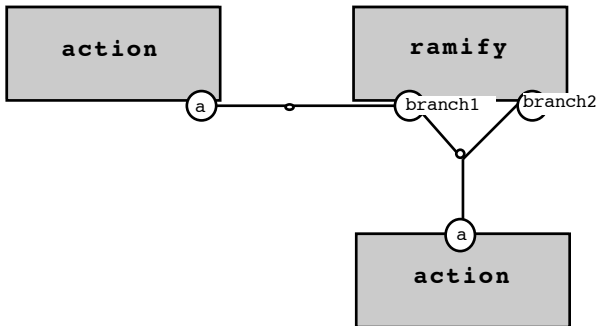
Given an n -ary connector



the encapsulation of the i -th role is performed as follows: the pushout of the i -th connection is calculated, and the other connections are changed by composing the morphisms that connect the channels to the glue with the morphism that connects the glue with the apex of the pushout, yielding a connector of arity $n-1$.

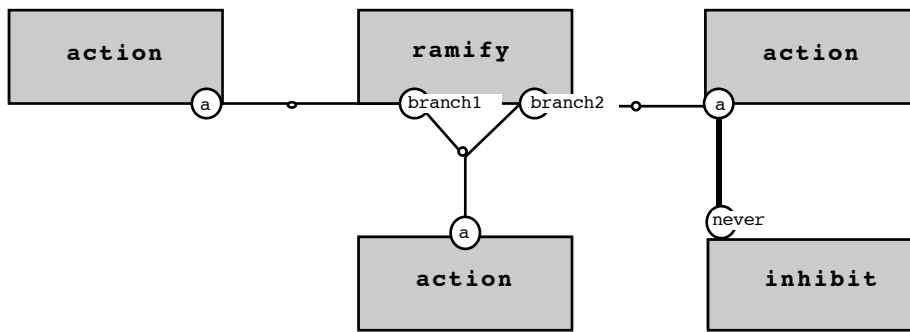


For instance, we can obtain the action subsumption connector from the action ramification one through encapsulation of the right-hand side *action* role::

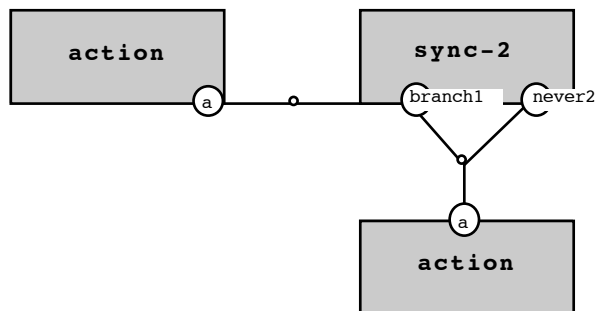


We can also obtain a synchronisation connector through refinement of one role with the *inhibit* program

10.1 Three operations on connectors



and then encapsulating it

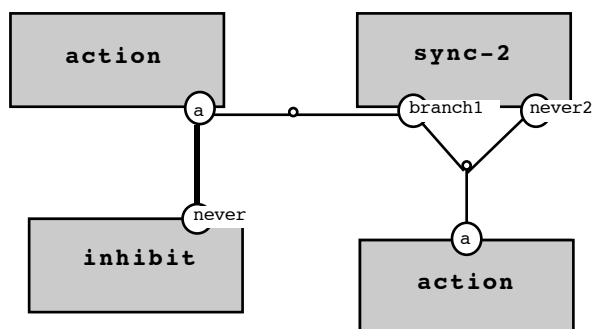


```

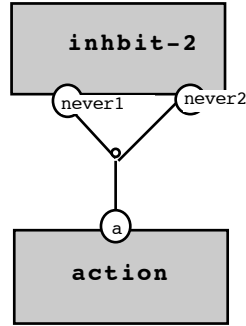
design sync-2 is
do branch1: true [] skip
    [] never2: false [] skip
  
```

Notice that the synchronisation connector obtained is not syntactically equal to the one presented in section 3.2, but it is "equivalent" because one of the ramifications of the bottom action *a* is never executed (because it is synchronized with *never2*) and thus the net effect of the connector is to synchronise both actions *a* through *branch1*.

If we now refine the left-hand branch also with *inhibit*



and then encapsulate it, we obtain a connector that is equivalent to *inhibit*.



```

design inhibit-2 is
do  never1: false □ skip
    [] never2: false □ skip

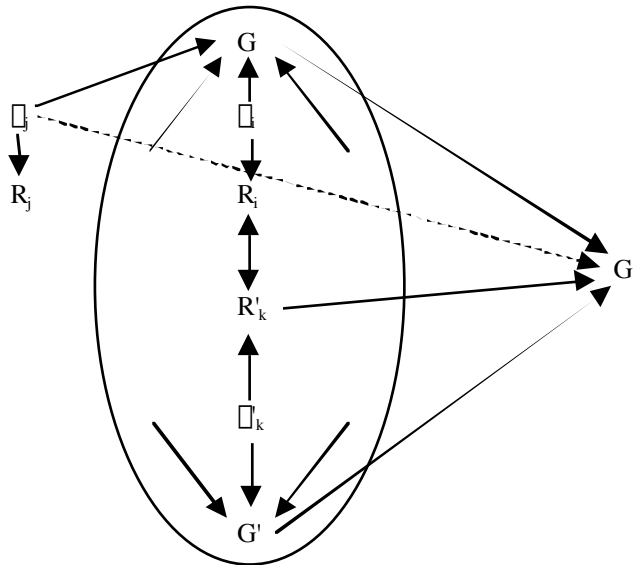
```

Hence *ramification* can be seen as a primitive connector.

10.1.3 ROLE OVERLAY

The third operation allows combining several connectors into a single one if they have some roles in common, i.e., if there is an isomorphism between those roles. The construction is as follows.

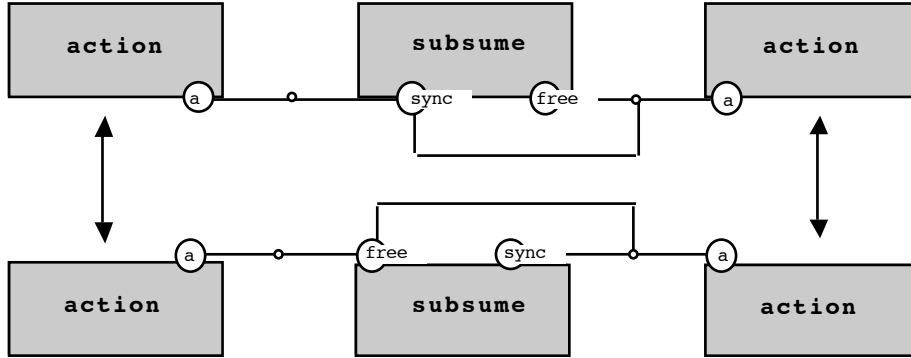
Consider an n -ary connector with roles R_i and glue G , and an m -ary connector with roles R'_k and glue G' . The glue of the new connector is calculated as follows. Consider the diagram that consists of all the connections that have isomorphic roles together with the isomorphisms, and calculate its colimit. The apex of the colimit is the new glue. Each of the pairs of connections involved gives rise to a connection of the new connector. The role of this connection is one of the roles of the old connections; because they are isomorphic, it does not matter which one is chosen (in the figure, we chose R'_k). This role is connected directly to the new glue through one of the morphisms that results from the colimit; hence, its channel is the role signature.



Each of the connections of the original connectors that is not involved in the calculation of the new glue, which means that it does not share its role with a connection of the

other connector, becomes a connection of the new connector by composing the old new glue morphism with the colimit morphism that connects the old glue to the new one. This is exemplified in the figure with the connection with role R_j .

This operation provides a second way of showing that synchronisation is not a primitive connector in our catalogue. We can obtain full synchronisation of two actions by making each one subsume the other. This is achieved by overlaying two copies of the subsumption connector in a symmetric way: the first (resp. second) role of one copy corresponds to the second (resp. first) role of the other copy. The diagram is



and its colimit makes all actions collapse into a single one. Hence the glue of the resulting connector is isomorphic to the 'sync' component. Moreover, each pair of overlaid roles results into a single one, and therefore there will be only two 'action' roles. In summary, the resulting connector is the synchronisation connector.

10.2 Higher-order connectors

As explained before, 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 *compression* as an example. The goal is to be able to adapt any connector that represents a communication protocol in order to compress data for transmission in a transparent way. In order to give a first-class description of this form of adaptation, the kind of communication protocol modelled 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 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 (see sections 8.1 and 9.2, respectively). 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 parameterised by the algebraic specification described below.

```

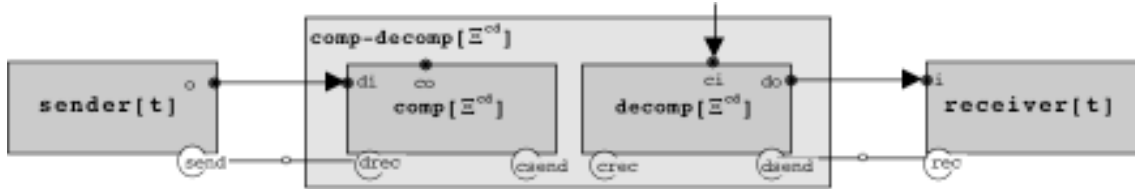
spec   []cd is []nat +
sorts  s,t
ops    comp:t->s          decomp:s->t
          size_s:s->nat      size_t:t->nat
axioms decomp(comp(x))=x, for any x:t
          size_s(comp(x)) ≤ size_t(x), for any x:t

```

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. The size of the compressed message is required not to be 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)/[]^{cd}*, is defined by

- the binary connector *Compression*/[]^{cd}



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

```

design comp[[]cd] 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[[]cd] 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

```


Design $comp[\square^{cd}]$ models the compression of messages of type t received through di into messages of type s that are then transmitted through co . Design $decomp[\square^{cd}]$ 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

$$\square_s.sender[s] \square comp-decomp[\square^{cd}] \text{ and } \square_r.receiver[s] \square comp-decomp[\square^{cd}]$$

induced, respectively, by

$$\square_s.sender[s] \square comp[\square^{cd}]$$

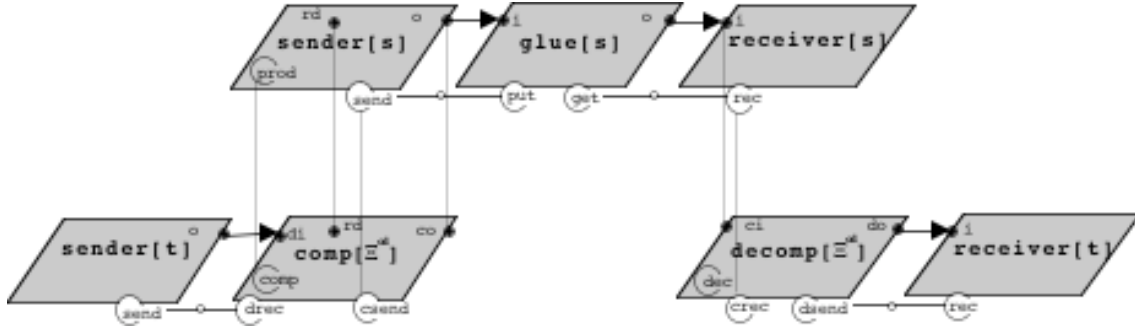
$$\square_s(o)=co, \square_s(rd)=rd, \square_s(comp)=prod, \square_s(csend)=send$$

$$\square_r.receiver[s] \square decomp[\square^{cd}]$$

$$\square_r(i)=ci, \square_r(crec)=rec$$

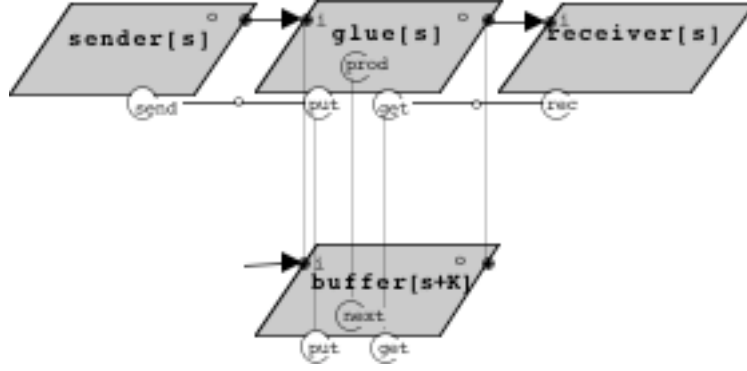
Because components $comp$ and $decomp$ do not interact, any component refined by one of them is also refined by their composition $comp-decomp[\square^{cd}]$. The corresponding induced morphisms have only to take into account the renaming of variables and actions that takes place in composition.

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



In summary, $Compression(Uni-comm)[\square^{cd}]$ 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, the formal parameter $Uni-comm[s]$ is the connector used to transmit compressed messages.

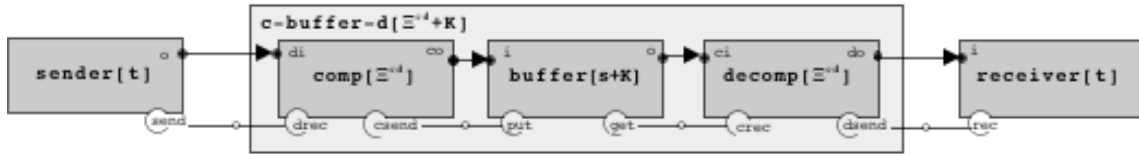
Now it remains to explain the procedure of parameter passing, i.e., how the service just described can be installed over a specific connector and how the resulting connector is obtained.



We consider again the *Async* connector. In this case, it is not difficult to realize that we may replace the formal parameter of $Compression(Uni-comm)[\square^{cd}]$ 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.

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. We will present and discuss these morphisms in more detail further down.

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 \square_s and \square_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 $Compression(Async)[\square^{cd}+K]$. Its roles are *sender* and *receiver* and its glue *c-buffer-d* $[\square^{cd}+K]$ is defined in terms of a configuration involving *comp*, *decomp* and *buffer* as shown below.



Summarising, we have described the installation of a compression-decompression service over a unidirectional communication protocol as a parameterised 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.

As another example of a higher-order connector, consider the adaptation of unidirectional communication protocols in order to transmit certain kind of messages (for instance, error messages) to a *monitoring* component. The nature of the messages that should be transmitted to the monitoring component is taken as a parameter. More concretely, we define a higher-order connector that is parameterised by the following algebraic specification:

```

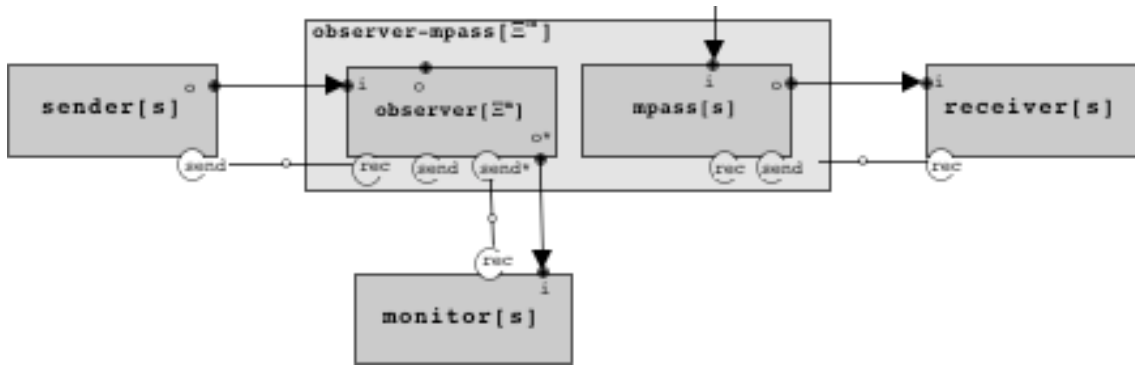
spec  $\square^m$  is  $\square^{bool+}$ 
sorts  $s$ 
ops  $to\_monitor:s \rightarrow bool$ 

```

Sort s represents the type of messages, and operation $to_monitor$ identifies the special kind of messages that are to be monitored. We use \square^{bool} to represent the sub-specification of \square that is concerned with the specification of booleans.

To simplify the presentation, consider that the communication with the monitoring component is achieved by synchronous message passing. In the more general case, *monitoring* would be modelled through a higher-order connector with two formal parameters, both standing for 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 simpler higher-order connector that we propose, $Monitoring(Uni-comm)[\square^m]$, consists of

- the connector $Monitoring[\square^m]$ defined by



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

```

design observer $[\square^m]$  is
in  $i:s$ 
out  $o,o*:s$ 
prv  $v:s; rd,rd*,msg:bool$ 
do  $rec:\square msg \square v:=i || msg:=true$ 
     $[] \text{ obsv:} \square rd \square rd* \square msg \square \text{ msg}:=false || o:=v || rd:=true || o*:=v$ 
     $|| rd*:=to\_monitor(i)$ 
     $[] \text{ send:} rd \square rd:=false$ 
     $[] \text{ send*} : rd* \square rd*:=false$ 

design mpass $[s]$  is
in  $i:s$ 
out  $o:s$ 
prv  $rd:bool$ 
do  $rec:\square rd \square o:=i || rd:=true$ 
     $[] \text{ send:} rd \square rd:=false$ 

```

Component $observer[\square^m]$ 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]*:

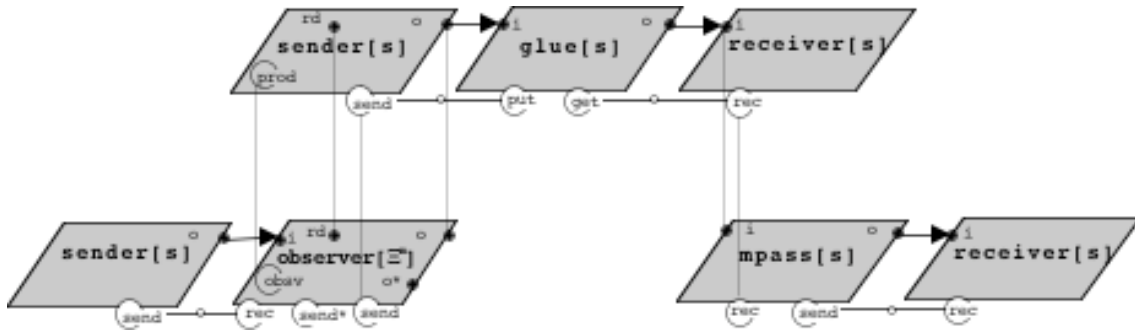
```

design monitor[s] is
in      i: s
do      rec: true  $\square$  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 below.

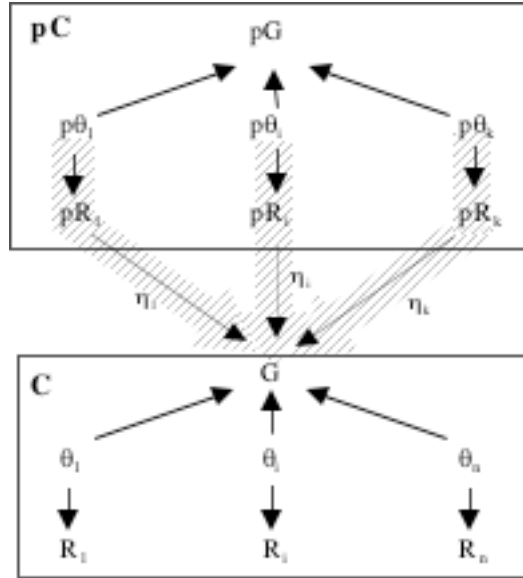


The notion of higher-order connector, as a connector that takes one connector as parameter and delivers another as a result, can be defined as follows.

10.2.1 DEFINITION – higher-order connector

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

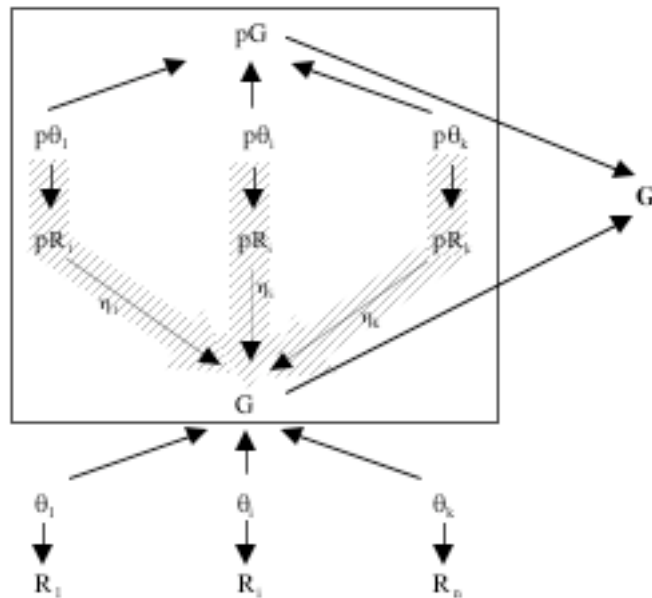
- a connector *pC*, called the formal parameter of the hoc; its roles, glue and connections are called, respectively, the parametric roles, the parametric glue and the parametric connections of the hoc;
- a connector *C* – its roles and glue are also called the roles and the glue of the hoc;
- an instantiation of the formal parameter connector with the glue of the hoc, i.e., a refinement morphism \sqsubseteq_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



constitutes a well-formed configuration.

10.2.2 DEFINITION – semantics of a higher-order connector

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 $pC+(\square)_i$.



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.

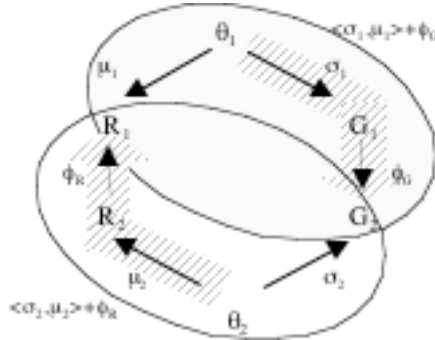
In order to be able to use, in the design of a given system, a connector C in place of a connector C' , it is obvious that the two connectors must have the same number of roles. Furthermore, C' must be able to be instantiated with the same components as C . That is to say, every restriction on the components 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 .

As shown through the connector *Uni-comm*, connectors may be based on glues that are not fully developed as designs (may be underspecified). Nevertheless, the concrete commitments that they make may 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 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:

10.2.3 DEFINITION – fitting morphism

A fitting morphism \square from a connection $\langle \square_1 : \text{desc}(\square_1) \square G_1, \square_1 : \text{desc}(\square_1) \square R_1 \rangle$ to a connection $\langle \square_2 : \text{desc}(\square_2) \square G_2, \square_2 : \text{desc}(\square_2) \square R_2 \rangle$ consists of a pair $\langle \square_G : G_1 \square G_2, \square_R : R_2 \square R_1 \rangle$ of refinement morphisms in *r-DESC* s.t. the interconnection $\langle \square_1, \square_1 \rangle + \square_G$ of R_1 with G_2 is, according to the configuration refinement CR , refined by the interconnection $\langle \square_2, \square_2 \rangle + \square_R$.



A fitting morphism \square from a connector C_1 to a connector C_2 with the same number of connections consists of a fitting morphism \square from each of C_1 's connections to each of C_2 's connections, all with the same glue refinement \square_G .

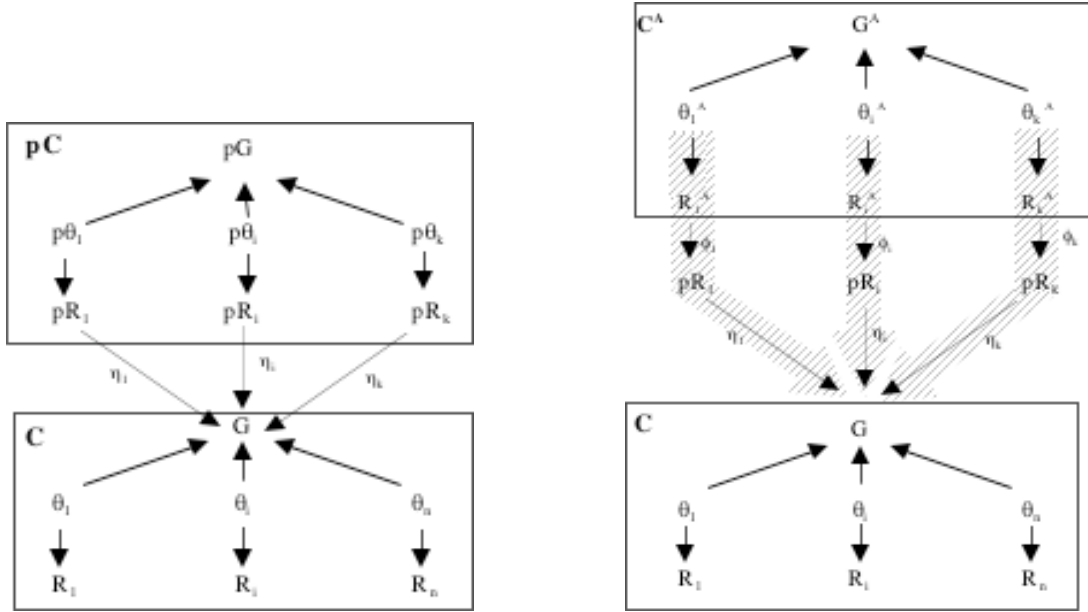
If there exists a fitting morphism from a connector C_1 to a connector C_2 , then we may replace each occurrence of the connector C_1 in an architectural description of a sys-

tem by an occurrence of C_2 . The compositionality of the design formalism w.r.t. 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.

10.2.4 DEFINITION – instantiation of a higher-order connector

An instantiation of a higher-order connector with formal parameter pC consists of a connector C^A (the actual parameter) together with a fitting morphism $\llbracket pC \rrbracket C^A$ such that a well-formed configuration is obtained by first composing the role morphisms of each actual connection with the corresponding fitting component, and then with the role instantiation.



The semantics of a higher-order connector instantiation is the connector with the same roles as C and whose glue is a design returned by the colimit of the configuration $C^A + (\llbracket \cdot \rrbracket; \llbracket \cdot \rrbracket_i)$.

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 modelled 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, modelled separately by different higher-order connectors, can be combined, 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.

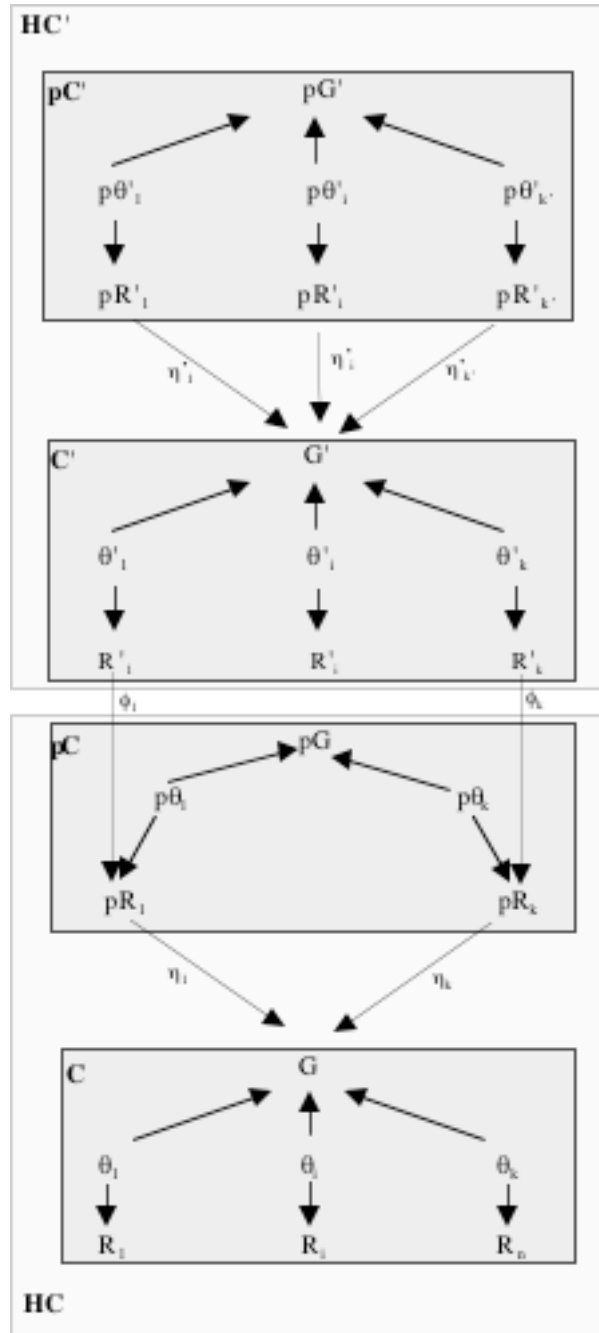
The key idea for composition of hocs is the instantiation of a hoc with a hoc. In this section we shall present this more general form of instantiation — *parameterised instantiation*. So, for instance, *Monitoring(Uni-comm)* can be instantiated with *Compress(Uni-comm)*, giving rise to the hoc *Monitoring&Compress(Uni-comm)* that 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*.

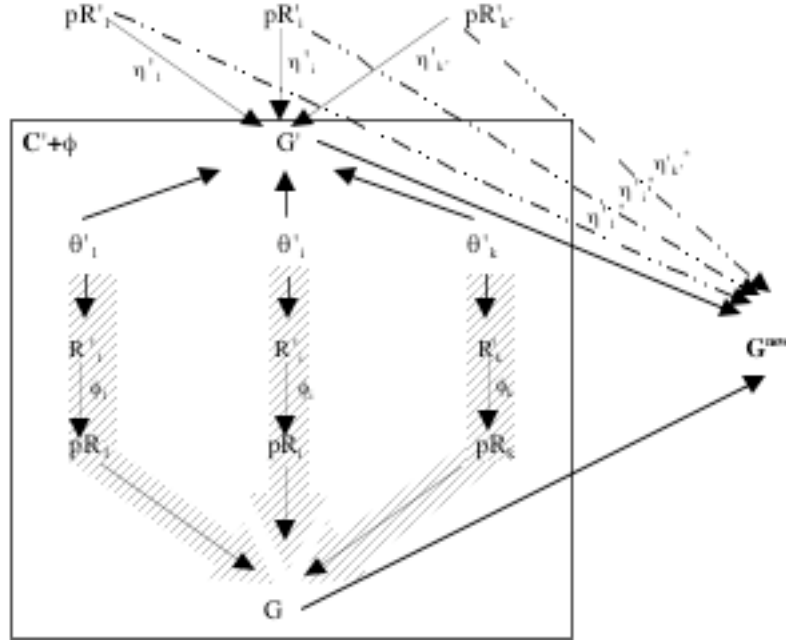
Parameterised instantiation is similar to standard instantiation:

10.2.5 DEFINITION – parameterised instantiation

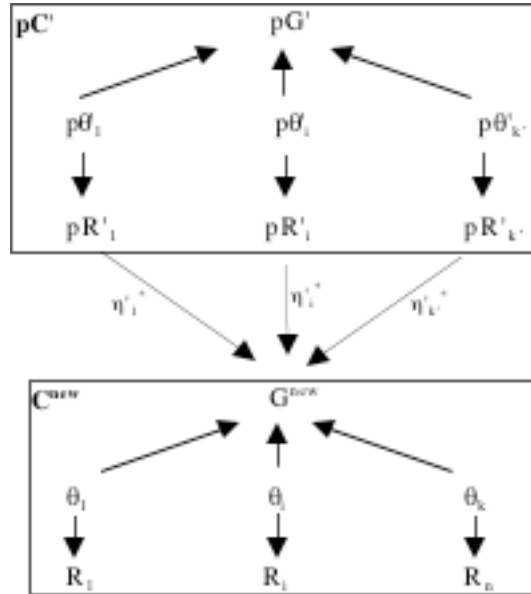
A parameterised instantiation of a higher-order connector HC with formal parameter pC consists of a higher-order connector HC' together with a fitting morphism $\llbracket pC \rrbracket Con'$, where Con' is the semantics of HC' , such that it is possible to extend, in a unique way, the instantiation of pC' with G' to an instantiation of pC' with the colimit of $C' + (\llbracket \cdot \rrbracket_i)_{i \in 1..k}$ that connects the glues of HC' and HC .



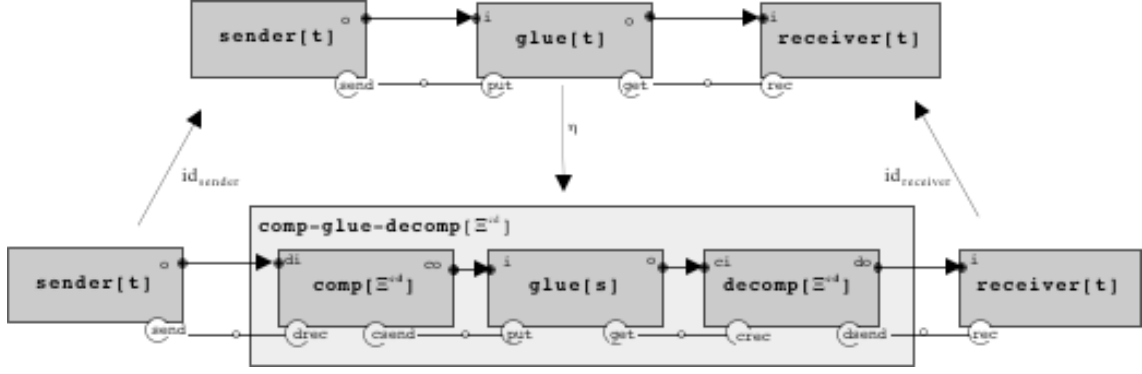
In the figure below, G^{new} is the colimit of $C' + (\Box_i \Box_j)_{i \in 1..k}$. We have used dotted lines for the refinement morphisms whose existence is required by the construction.



The semantics of a parameterised instantiation is the higher-order connector depicted below:



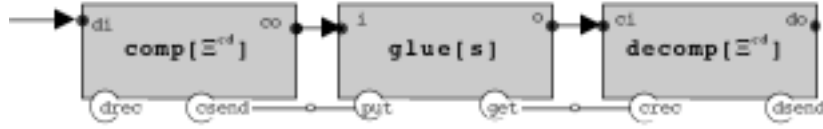
For instance, consider that we want to combine the service of compression of messages with monitoring. We take the parameterised instantiation of $Monitoring(Uni-comm)/[\Box^m]$ with $Compress(Uni-comm)/[\Box^{cd}]$ defined by the fitting morphism



and the refinement morphism $\square: \text{glue}[t] \square \text{comp-glue-decomp}[\square^{cd}]$ defined by

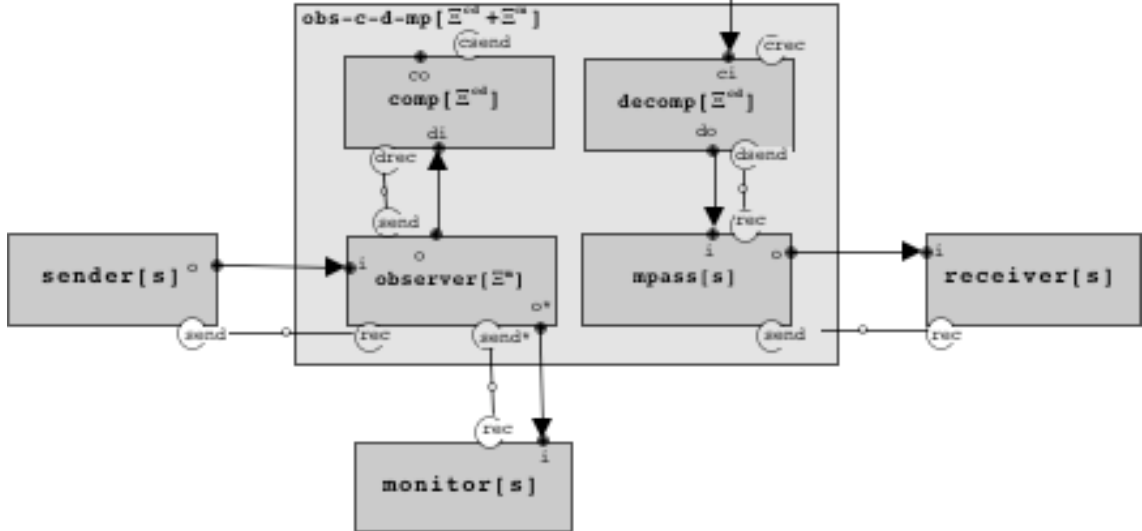
$$\begin{aligned} \square(i) &= \square_c(di), \quad \square(o) = \square_d(do), \\ \square(\square_c^{-1}(drec)) &= \text{put}, \quad \square(\square_d^{-1}(dec)) = \text{prod}, \quad \square(\square_d^{-1}(dsend)) = \text{get} \end{aligned}$$

where $\square_c: \text{comp} \square \text{comp-glue-decomp}[\square^{cd}]$ and $\square_d: \text{decomp} \square \text{comp-glue-decomp}[\square^{cd}]$ are the morphisms in **c-DSGN** returned by the colimit of the diagram



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

- the connector *Monitoring&Compress* $[\square^{cd} + \square^m]$ defined by



- the connector *Uni-comm* $[s]$ — the formal parameter;
- the refinement morphisms $\square_s: \text{sender}[s] \square \text{obs-c-d-mpass}[\square^{cd} + \square^m]$ and $\square_r: \text{receiver}[s] \square \text{obs-c-d-mpass}[\square^{cd} + \square^m]$ obtained by composing, at the level of signatures, the morphisms $\text{sender}[s] \square \text{comp}[\square^{cd}]$ and $\text{receiver}[s] \square \text{decomp}[\square^m]$ of *Compression(Uni-comm)* $[\square^{cd}]$ with the morphisms $\text{comp}[\square^{cd}] \square \text{obs-c-d-mpass}[\square^{cd} + \square^m]$ and $\text{decomp}[\square^{cd}] \square \text{obs-c-d-mpass}[\square^{cd} + \square^m]$, respectively. These are given by the colimit construction.

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

