Superposing Connectors[*]

Michel Wermelinger
Departamento de Informática
Fac. de Ciências e Tecnologia
Universidade Nova de Lisboa
2825-114 Caparica
Portugal
mw@di.fct.unl.pt

Antónia Lopes
Departamento de Informática
Faculdade de Ciências
Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal
mal@di.fc.ul.pt

José Luiz Fiadeiro[†]
Department of
Computer Science
King's College London
Strand, London WC2R 2LS
United Kingdom
jose@fiadeiro.org

Abstract

The ability to construct architectural connectors in a systematic and controlled way has been argued to promote reuse and incremental development, e.g., as a way of superposing, à la carte*, services like security over a given communication protocol. Towards this goal, we present a notion of high-order connector, i.e., a connector that takes connectors as parameters, for superposing coordination mechanisms over the interactions that are handled by the connectors that are passed as actual arguments. The notion is developed over the language* COMMUNITY *that we have been using for formalising aspects of architectural design, and illustrated with examples inspired by the case study.*

1. Introduction

Software Architecture has put forward connectors as first-class entities to express complex relationships between system components, thus facilitating the separation of coordination from computation. However, as argued in [13], the current level of support and understanding of what the fundamental blocks of software interaction are, are still insufficient. Although there is some work on the relationships and characteristics of connectors [14, 1, 2, 9, 13], a further step is necessary to achieve the *systematic construction* of new connectors from existing ones. This would allow to compose simpler interactions into more complex ones in an easier way. For example, it is desirable to be able to add services like security, reliability, and throughtput over a given communication protocol. This would promote the incremental development of a system, and allow to obtain benefits from the multiple combinations of different services, ideally chosen *à la carte*.

As Denker and colleagues argue [5], modularising the different kinds of services has other advantages. It prevents interactions to be "hard-wired" across different components, it promotes reuse, 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.

Further reasons for principled ways of modifying connectors have been presented by Garlan [8]. His position is briefly summarised as follows. 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 might be to modify the connectors because usually there are fewer connector types than components types. Moreover, most Architecture Description Languages 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.

To our knowledge, there are only two approaches to systematic connector construction, with completely different philosophies. While our own strives at formal elegance and simplicity, providing three generic and basic operations [17], the second approach tries to achieve a middle ground, providing moderately complex and specialized operations [15]. We continue in this paper our work on the formal underpinning of connectors [6, 16], presenting another way of combining connectors through the definition of high-order

[*]This research was partially supported by Fundação para a Ciência e Tecnologia through project 32717/99 (FAST—Formal Approach to Software Architecture).

[†]On leave from the University of Lisbon with the support of Fundação para a Ciência e Tecnologia and the EPSRC.

connectors: connectors that take connectors as parameters. This allows a high-order connector to superpose certain coordination mechanisms over the interactions that are handled by the connector that is passed as an actual argument. In this way we obtain "connector stacks" that are similar in spirit to network protocol stacks, where each layer handles a given protocol, and to meta-object towers [5].

We apply our mechanism to examples inspired by the case study: a security high-order connector and a monitoring high-order connector are superposed on an asynchronous communication connector. We illustrate our proposal with a Unity-like parallel program design language which is nearer to the abstractions used by conventional programming languages than the process calculi used by others [12, 1, 15].

2. CommUnity

CommUnity programs are in the style of Unity programs [4], but they also combine elements from IP (Interacting Processes) [7]. However, CommUnity has a richer coordination model and, even more important, it requires interaction between components to be made explicit. In this way, the coordination aspects of a system can be separated from the computational aspects and externalised, making explicit the gross modularisation of the system in terms of its components and its interactions.

In this section, we present the syntax of CommUnity programs and introduce a notation for describing system configurations. Each configuration can be transformed into a single, semantically equivalent, program that represents the whole system. Finally, we discuss refinement of CommUnity programs. The formal definitions were presented in [11].

2.1. Programs

CommUnity 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 in the usual sense. For the purposes of examples, we consider an algebraic signature containing: sorts `bool` (booleans) and `int` (integers) with the usual operations; lists with empty list 'nil', operations `head`, `tail`, and `+` (concatenation); ordered pairs with projection functions `fst` (first element) and `snd` (second element); a function `if`(*cond, then-expr, else-expr*) with the obvious meaning.

The syntax of a CommUnity program is

```
prog $P$
in     $in(V)$
out    $out(V)$
prv    $prv(V)$
init   $I$
do
$\mathop{[]}_{a\in A}$   $a$: $S(a), P(a) \rightarrow$   $\mathop{\|}_{v\in D(a)}$   $v :\in E(a,v)$
```

- $in(V)$ is the set of *input* variables. They are imported from the environment of the program, i.e., they are to be connected with output variables of other components in the environment. Their values can be read but not modified by the program.

- $out(V)$ and $prv(V)$ are the sets of *output* and *private* variables, respectively. They are called *local* to the program, because the environment cannot modify them. Output variables are accessible to the environment (can be read) but private variables are not.

- I is a proposition over the local variables, defining their admissible values in the initial state.

- A is the set of *actions*. Their execution is also under the control of the environment, i.e., their execution may require synchronisation with actions of other components. In a sense, actions provide interaction points as in IP (and are also similar to the actions of Action Systems [3]).

- $S(a)$ is the *safety* guard of a, i.e., when $S(a)$ is false, a cannot be executed.

- $P(a)$ is the *progress* guard of a, i.e., when $P(a)$ holds, the system is willing to execute a. This means that the program cannot refuse to execute a if the environment requests it. In other words, $P(a)$ defines in which conditions progress via a is required. It follows that $P(a)$ is required to imply $S(a)$.

- $D(a)$ is the *domain* of a, defined as the set of local variables that action a can change—its write frame.

- For every variable v in $D(a)$, $v :\in E(a,v)$ is a nondeterministic assignment, with $E(a,v)$ a set expression: each time a is executed, v is assigned one of the values denoted by $E(a,v)$, chosen in a nondeterministic way.

When $S(a)$ and $P(a)$ coincide, we write only one guard, and we abbreviate $v :\in \{t\}$ as $v := t$. When an action has empty domain we use `skip` to denote the absence of assignments.

The behaviour of a closed program, i.e., a program with no input variables, is as follows. The program starts its execution in some state that satisfies the initial condition. At each step, one of the actions whose safety guard is true is selected and its assignments are executed simultaneously. The behaviour of an open program can only be given in the context of a configuration in which its input variables have been connected to output variables of other components. We will address this issue in Section 2.2.

We now present programs to be used in the remaining of the paper. The first program models a small box with buttons and a light. Its purpose is to allow patients and their families to request for help in a simple and quick way in case of medical emergency. Each button is assigned to a particular emergency. Pressing a button turns on the light, which will turn off after sucessful transmission of the request. To make the example compact, we consider only two different emergency cases.

```
prog Help
out   data: {1,2} × int
prv   off : bool
init  off
do    help$_1$: off → data := ⟨1, $Id$⟩ || off := false
[]    help$_2$: off → data := ⟨2, $Id$⟩ || off := false
[]    ack: ¬off → off := true
```

We assume that Id is a constant representing the patient identification. Notice that variable 'off' representing the status of the light is private because the light is only visible to the patient.

We also need a component for the assistance centre. We assume the centre can handle at most k patients to guarantee the necessary quality of service. There is one input variable 'd$_i$' and one action 'req$_i$' for the i-th patient ($i = 1, \ldots, k$). There is one request queue for each kind of emergency, and each 'req$_i$' puts the patient's request in the appropriate queue. For our purposes, the actions that take requests from the queues and process them are irrelevant.

```
prog Centre
in    d$_i$ : {1,2} × int
prv   reqs$_1$, reqs$_2$ : list(int); current$_1$, current$_2$ : int
init  reqs$_1$ = nil ∧ reqs$_2$ = nil
do    req$_i$: true
      → reqs$_1$ := if(fst(d$_i$)=1, reqs$_1$ + snd(d$_i$), reqs$_1$)
      || reqs$_2$ := if(fst(d$_i$)=2, reqs$_2$ + snd(d$_i$), reqs$_2$)
```

In Section 3 we present a generic connector to transmit the help requests asynchronously. For that purpose we need generic sender and receiver components for messages of any sort s.

```
prog Sender ($s$)
out   o : $s$
prv   sent : bool
init  sent
do    prod: sent, false → o :∈ $s$ || sent := false
[]    send: ¬sent, false → sent := true
```

```
prog Receiver ($s$)
in    i : $s$
do    rec: true, false → skip
```

For the 'Sender', we require that it does not produce another message before the previous one has been read. After generating a message with action 'prod', the 'Sender' expects an acknowledge—modelled by the execution of action 'send'—to produce a new message. For the 'Receiver', we simply require that it has an action that models the reception of a message.

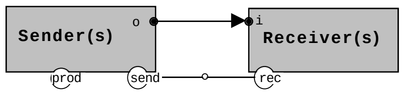
Notice that progress guards are 'false' to leave unspecified when and how many messages the sender (receiver) will send (receive)—see also Section 2.3. We choose the least deterministic assignment (o :∈ s) for the production of messages in order to avoid committing to a particular discipline of production.

2.2. Configurations

In CommUnity, 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, unlike other program design languages, CommUnity requires interaction between components (name bindings) to be made explicit. For this purpose, a very simple box-and-line notation is introduced. Interactions between components in a system are described in terms of their interfaces. In CommUnity, the interface of a program consists of the non-private variables and the actions. We consider that interfaces are represented as boxes with actions identified by circles and variables by bullets. Input variables are distinguished from output variables with an incoming arrow.

The interconnection of an input variable i of a component P with an output variable o (of the same sort as i) of a component Q is simply expressed by definining o as the source of the arrow that comes into i. The synchronisation of an action of a component with an action of another component is expressed by connecting the two actions with a line.

As a configuration example, the 'Sender' and 'Receiver' programs of the previous subsection may be interconnected to transmit messages synchronously. This is achieved by synchronising the send action with the receive action, and connecting the sender's output with the receiver's input.



Such a box-and-line configuration has a very precise mathematical semantics, given by a diagram in a category whose objects are programs and whose morphisms denote superposition. The interconnections between two components are described by programs (which we call channels) that just declare the common variables and actions of the components, without introducing any additional behaviour, thus corresponding to the recently proposed notion of duct [13]. This semantics has been presented previously [11, 10] and it is trivial to translate the box-and-line diagrams into categorical diagrams. One of the advantages of the semantics is that any categorical diagram corresponding to a box-and-line configuration can be transformed into a single program that represents the whole system, as proven in [10]. The transformation operation is given by a universal categorical construction called colimit. We now describe the intuitive meaning of the colimit and its construction.

Let us consider a box-and-line diagram involving the programs $P_1, \ldots, P_n$ (not necessarily all different) and expressing the configuration of a system. In the simplest case, there are no interactions. This means that any two variables of two programs are different, even if they have the same name. Therefore the variables of the resulting program are the disjoint union of the components' variables. Regarding actions, the parallel composition contains all possible combinations of actions from the components, since there is no restriction on their co-occurrence. More concretely, the actions of the resulting program are the tuples of actions of the components $a_1|\ldots|a_k$, for $1 \leq k \leq n$, containing at most one action of each component. In this way, the colimit provides a model for concurrent execution based on synchronisation sets in which each set is executed atomically

In the presence of interactions, the colimit "merges" the input variables identified with an output variable into that output variable, and a tuple $a_1|\ldots|a_k$ is allowed if for every action a_j in the tuple, every action that is required to synchronise with a_j is also in the tuple.

With or without interactions, the initialisation condition of the resulting program is the conjunction of all the initialisation conditions of the components, and for each action $a_1|\ldots|a_k$, its guards are the conjunction of the guards of all a_i and its assignments are the union of the assignments of all a_i.

For example, the colimit of the 'Sender'/'Receiver' configuration is

```
prog Sender-Receiver (s)
out  o : s
prv  sent : bool
init sent
do   prod: sent, false → o:∈ s || sent := false
[]   send|rec: ¬sent, false → sent := true
```

2.3. Refinement

A key factor for architectural description is a notion of refinement that can be used to support abstraction. In particular, refinement is necessary when one wants to conduct the description of the architecture of a system at the level of the coordination that needs to be established between its components, leaving computational concerns for later stages of design.

A program R refines program P if there is a mapping satisfying the following conditions:

1. Every variable of P is mapped to a variable of R of the same sort and kind (input, output, private). The mapping is injective for input and output variables.

2. Every action a of P is mapped to a set $\{b_1, \ldots, b_n\}$ of actions of R with $n > 0$ (non-empty set).

3. Each b_i may strengthen the safety guard of a.

4. Each b_i must contain the assignments of a, possibly making them more deterministic.

5. The progress guard of a must imply the disjunction of the progress guards of the b_i.

6. Any action of R that is not image of an action of P may not assign to a variable that is image of a variable of P.

7. The initialisation condition of R is not weaker than that of P.

Condition 1 ensures that refinement does not alter the border between the program and its environment, in particular different variables of the interface cannot be collapsed into a single one. Condition 2 ensures that actions, because they model interaction between the program and its environment, have to be implemented. The set $\{b_1, \ldots, b_n\}$ is a menu of refinements for a. Condition 3 preserves the safety properties of P. Conditions 4 and 7 preserve the functionality of P. Condition 5 states that progress guards can be weakened but not strengthened. Indeed, because progress guards represent a requirement on the availability of an action for execution, refinement has to preserve that availability under the conditions established by the progress guard of the abstract program P. Naturally, the circumstances under which this availability is guaranteed can be widened, which corresponds to the weakening of the progress guard. Finally, condition 6 preserves locality of changes by stating that the new actions of R (i.e., those that do not involve P) cannot modify any of the variables of P.

As an example, the following mapping shows in which way 'Help' is a refinement of 'Sender($\{1,2\} \times$ int)'.

$$\begin{tikzcd}
\text{Sender}(\{1,2\} \times \text{int}) \arrow[r, squiggly, "\text{o} \mapsto \text{data, sent} \mapsto \text{off}", "\substack{\text{prod} \mapsto \{\text{help}_1,\, \text{help}_2\} \\ \text{send} \mapsto \text{ack}}"'] & \text{Help}
\end{tikzcd}$$

It should be noted how each 'help$_i$' action makes the assignment deterministic and (trivially) weakens the false progress guard.

Likewise, the 'Centre' program can be seen as a special case of a 'Receiver'. In this case, there are k different refinement mappings, one for each patient connected to the centre. For example, for patient 4 the refinement mapping is:

$$\text{Receiver}(\{1,2\} \times \text{int}) \xrightarrow[\text{rec}\mapsto\text{req}_4]{\text{i}\mapsto\text{d}_4} \text{Centre}$$

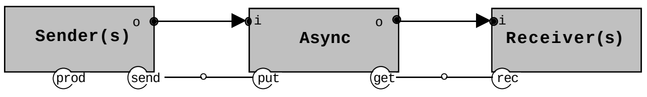
3. Connectors

Although components have always been considered the fundamental building blocks of software systems, the way the components of a system interact may be also determinant on the system properties. 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.

According to [1], an n-ary connector consists of n roles and one glue stating the interaction between the roles. The use of a connector in the construction of a particular system is realised by the instantiation of its roles with specific components of the system. Therefore the roles act as "formal parameters", restricting which components may be linked together through the connector.

In our framework, a connector is represented by a star-shaped configuration, with the glue in the center linked to each role. An n-ary connector is applied to n components by defining which component refines which role in which way.

The configuration



defines a generic asynchronous message passing connector, with

```
prog Async
in   i : $s$
out  o : $s$
prv  ready: bool
init ready
do   put: ready $\to$ o := i || ready := false
[]   get: $\neg$ready $\to$ ready := true
```

The semantics of the connector, given by the diagram's colimit, is the parallel composition of 'Sender', 'Receiver' and 'Async' with the restrictions defined by the configuration diagram, showing exactly the order in which transmission actions occur:

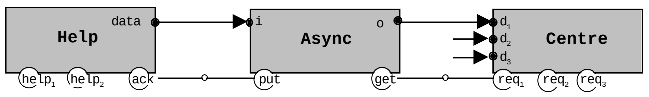
```
prog Async-Msg
out  o, i : $s$
prv  ready, sent : bool
init ready $\wedge$ sent
do   prod: sent, false $\to$ o :$\in$ $s$ || sent := false
[]   put|send: ready $\wedge$ $\neg$sent, false
          $\to$ i := o || ready := false || sent := true
[]   get|rec: $\neg$ready, false $\to$ ready := true
```

Notice that, contrary to the synchronous communication configuration shown in Section 2, actions 'send' and 'rec' are no longer synchronised.

We may use this connector to establish asynchronous communication between the 'Help' and 'Centre' components because 'Help' refines 'Sender' and 'Centre' refines 'Receiver', as described in Section 2. We use the following notation to describe connector instantiation in a compact way:

$$\begin{tikzcd} \boxed{\text{Help}} & \boxed{\text{Sender}(\{1,2\} \times \text{int})} \arrow[l, squiggly, "\text{o}\mapsto\text{data, sent}\mapsto\text{off}"', "\text{prod}\mapsto\{\text{help}_1\text{, help}_2\}\ \text{send}\mapsto\text{ack}"] \arrow[d, dotted, dash, "\text{Async}"] \\ \boxed{\text{Centre}} & \boxed{\text{Receiver}(\{1,2\} \times \text{int})} \arrow[l, squiggly, "\text{i}\mapsto\text{d}_j"', "\text{rec}\mapsto\text{req}_j"] \end{tikzcd}$$

In our categorical framework, the configuration of the resulting system can be constructed automatically by composing the interconnections with the refinements. For example, if the centre has at most 3 patients, the resulting configuration for the first patient (i.e., $j = 1$ in the previous figure) is



The connections for 'Help' are those shown because variable 'data' corresponds to variable 'o' of the 'Sender' which was connected to variable 'i' of 'Async', and likewise for actions: 'ack' corresponds to 'send' which was synchronised with 'put'. Similarly for the connections for 'Centre'.

4. High-order Connectors

As the previous section shows, a connector can be seen as a function (with roles acting as parameters) that takes components as arguments and returns a configuration (the instantiated connector). Taking this analogy further, a connector that allows connectors as parameters and result should be called a high-order connector, as suggested by Garlan [8]. To simplify the explanation, in the remaining of the paper we only deal with high-order connectors that take a single connector as parameter.

The intuition for our definition of high-order connector (hoc) is as follows. In the same way that roles restrict what components a connector can be applied to, the hoc needs a "template" of what connectors it can be applied to. We take as template just the roles because, by letting the glue unspecified, the hoc can be applied to any connector specifying any interaction between those roles.

Thus the parameters of a hoc are two sets of roles. The first set are the "normal" roles, to be instatiated with components. The second set, which we call *parametric roles*, restrict the roles of the connector that will be taken as argument. After applying the hoc to a connector whose roles match (are refined by) the parametric roles, the result will be a new connector, whose roles are given by the first set. Whereas the "normal" roles are connected to the hoc's glue, we impose that the glue must refine each parametric role. The intuitive reason is that the connector to be passed as argument to the hoc will become "embedded" into the resulting connector. Thus, the hoc must state in which way the embedding will be done, and this is specified through the particular refinement mappings from the parametric roles to the glue.

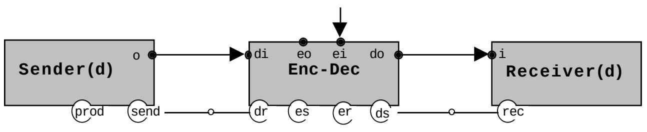
A concrete way to see this is to consider the 'Async' connector with role 'Sender' refined by 'Help' and 'Receiver' by 'Centre'. Adding a new service given by a hoc H can be seen as substituting the 'Async' connector by a new connector which is the combination of 'Async' with H. In that new connector, the asynchronous communication is not used directly by 'Help' and 'Centre' anymore, but by the service superposed on it, implemented by the glue G of H. Therefore, 'Async', which is instantiated with 'Help' and 'Centre', must become instantiated with G and that is done as for any connector application: we must show how G refines 'Sender' and 'Receiver'. Therefore the parametric roles of H must also be refined by G.

To sum up, an $\langle n, m\rangle$-hoc is defined by two star-shaped diagrams:

- the first is a configuration diagram, connecting the n roles to the glue, and

- the second is a refinement diagram, showing how the glue refines each of the m parametric roles.

As an example, we define a hoc to provide an encryption and decryption service on top of any unidirectional communication protocol involving a sender and a receiver. The configuration diagram is



with

```
prog Enc-Dec
in   di: d, ei: e
out  do: d, eo: e
prv  dec, enc: bool
init dec ∧ enc
do   dr: dec → eo := encrypt(di) || dec := false
[]   es: ¬dec → dec := true
[]   er: enc → do := decrypt(ei) || enc := false
[]   ds: ¬enc → enc := true
```

This hoc receives a decrypted message (sort d) through variable 'di' (decrypted input) and action 'dr' (decrypted receive), encrypts it, and passes it to the underlying communication protocol via action 'es' (encrypted send). No assumptions are made on this protocol: it might even transform the message! Whatever message the protocol delivers at variable 'ei' (encrypted input) is decrypted and delivered by the hoc through variable 'do' (decrypted output) and action 'ds' (decrypted send) to the receiver. The refinement diagram is therefore

$$\begin{tikzcd}
\boxed{\text{Sender}(e)} \arrow[r, rightsquigarrow, "{\begin{matrix}\text{o}\mapsto\text{eo}\\ \text{sent}\mapsto\text{dec}\end{matrix}}", "{\begin{matrix}\text{prod}\mapsto\text{dr}\\ \text{send}\mapsto\text{es}\end{matrix}}"'] & \boxed{\text{Enc-Dec}} & \boxed{\text{Receiver}(e)} \arrow[l, rightsquigarrow, "{\text{ei}\mapsfrom\text{i}}"', "{\text{er}\mapsfrom\text{rec}}"]
\end{tikzcd}$$

showing which variables and actions of the glue use the protocol to transmit encrypted messages (sort e).

Given a hoc with n roles and m parametric roles and an m-ary connector, the former is applied to the latter in the following steps, resulting in an n-ary connector:

1. Each role of the connector must be refined by one of the parametric roles of the hoc.

2. Since refinement mappings can be composed [10], the glue of the hoc refines each role of the connector. In other words, we are instantiating each role with the hoc's glue.

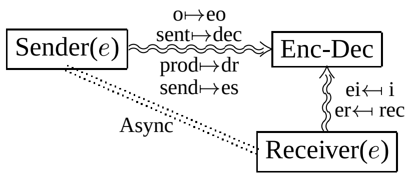
3. The configuration diagram equivalent to the previous instantiation diagram is obtained.

4. The colimit of the configuration is substituted for the hoc's glue, and the connections to the hoc's roles are changed accordingly.

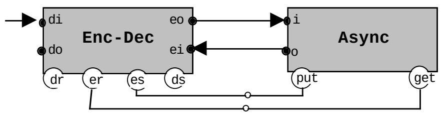
For example, the 'Enc-Dec' hoc is applied to the 'Async' connector for asynchronous communication of messages of sort e, resulting in a secure binary connector to asynchronously transmit messages of sort d:

1.

$$\begin{tikzcd}
\boxed{\text{Sender}(e)} \arrow[r, rightsquigarrow, "{\begin{matrix}\text{o}\mapsto\text{eo}\\ \text{sent}\mapsto\text{dec}\end{matrix}}", "{\begin{matrix}\text{prod}\mapsto\text{dr}\\ \text{send}\mapsto\text{es}\end{matrix}}"'] & \boxed{\text{Enc-Dec}} & \boxed{\text{Receiver}(e)} \arrow[l, rightsquigarrow, "{\text{ei}\mapsfrom\text{i}}"', "{\text{er}\mapsfrom\text{rec}}"] \\
\boxed{\text{Sender}(e)} \arrow[u, rightsquigarrow] \arrow[rr, dotted, no head, "\text{Async}"] & & \boxed{\text{Receiver}(e)} \arrow[u, rightsquigarrow]
\end{tikzcd}$$

2. 

3. Composing the interconnections of the 'Async' glue to its roles, with the refinements from the roles to the hoc glue, we get

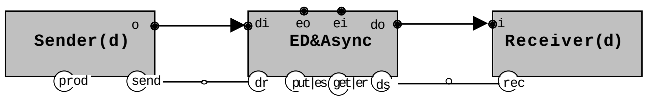


For example, variable 'eo' of 'Enc-Dec' is connected to the input variable 'i' of 'Async' because 'eo' corresponds to variable 'o' of 'Sender(e)' which in turn is connected to 'i'.

4. The program resulting from the previous configuration is

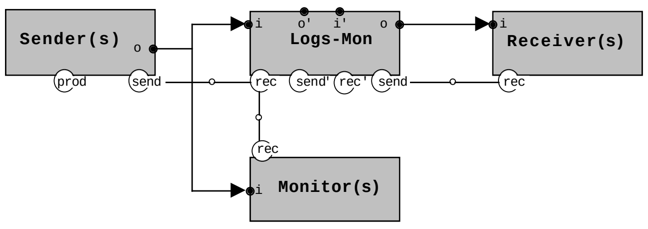
```
prog ED&Async
in     di : d
out    do: d; ei, eo : e
prv    dec, enc, ready : bool
init   dec ∧ enc ∧ ready
do     dr: dec → eo := encrypt(di) || dec := false
[]     put|es: ready∧ ¬dec
       → ei := eo || ready := false || dec := true
[]     get|er: ¬ready ∧ enc
       → do := decrypt(ei) || enc := false
       || ready := true
[]     ds: ¬enc → enc := true
```

and substituting it for 'Enc-Dec' results in the connector

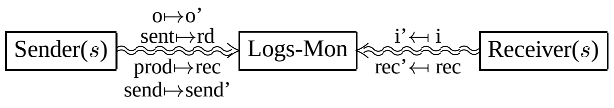


which can now be applied to 'Help' and 'Centre'.

As a second example, to log all help requests issued by a patient, we define a monitoring hoc, which forwards a copy of the transmitted messages to a monitor component. The configuration diagram is



and the refinement diagram is



with

```
prog Logs-Mon
in     i, i': s
out    o, o': s
prv    rd, rd' : bool
init   rd ∧ rd'
do     rec: rd → o':= i || rd := false
[]     send': ¬rd → rd := true
[]     rec': rd' → o:= i' || rd' := false
[]     send: ¬rd' → rd' := true
```

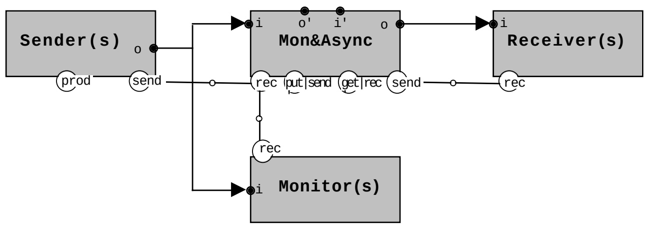
```
prog Monitor
in   i : s
do   rec: true → skip
```

The 'Logs-Mon' program is similar to 'Enc-Dec' in the sense that it "replicates" the variables and actions to use the underlying communication protocol.

Notice the difference between the 'Monitor' and 'Receiver' roles. The progress guard of the former is true, which implies that any component that acts as monitor must be always willing to read the values that are input through 'i'. However, the progress guard of 'Receiver' is false, which leaves it to the actual receiving component to decide when and how many times it reads the values sent to it.

Notice also that the monitoring hoc has three roles, but only two parametric roles, namely for the underlying communication protocol on which monitoring will be superposed. Furthermore, since monitoring does not change the underlying communication, the parametric roles are a subset of the hoc's roles.

We can now apply the monitoring hoc to the 'Async' connector. Applying the same construction steps as before, the resulting connector is



with glue

```
prog Mon&Async
in   i : $s$
out  o, o', i': $s$
prv  rd, rd', ready : bool
init rd ∧ rd' ∧ ready
do   rec: rd → o':= i || rd := false
[]   put|send': ready ∧ ¬rd
       → rd := true || i' := o' || ready := false
[]   get|rec': ¬ready ∧ rd'
       → o:= i' || rd' := false || ready := true
[]   send: ¬rd' → rd' := true
```

5. Concluding Remarks

To fully become first-class entities, connectors must be amenable to systematic analysis and manipulation like components. Continuing our previous work on categorical foundations for connectors [6, 16, 17], we presented a definition of high-order connectors and how they can be used to incrementally construct complex interactions. Although we have illustrated our approach with a UNITY-like language, the framework is applicable to any language for which notions of configuration and refinement can be categorically expressed, subject to the conditions presented in [10].

A high-order connector is constituted by a glue, specifying the interaction, a set of roles, restricting the components which the connector will link together, and a set of parametric roles, restricting which connectors the hoc may take as argument. A hoc is defined by two diagrams, one showing the interconnections between glue and roles, the other showing how the parametric roles are refined by the glue. This in turn will determine how the glue of the connector given as argument will be connected to the glue of the hoc. Through the categorical operation of colimit it is possible to combine the two glues into a single one, thus resulting a new connector, with the roles of the hoc, and in which the service (i.e., glue) provided by the hoc has been superposed on the service (i.e., glue) given as argument. In this way it is possible to build complex interactions. For the case study, we have combined an encryption/decryption hoc and a monitoring hoc with an asynchronous message passing connector.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, July 1997.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[3] M. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, 1996.

[4] K. M. Chandy and J. Misra. *Parallel Program Design—A Foundation*. Addison-Wesley, 1988.

[5] G. Denker, J. Meseguer, and C. Talcott. Rewriting semantics of meta-objects and composable distributed services. Internal report, Computer Science Laboratory, SRI International, 1999.

[6] J. L. Fiadeiro and A. Lopes. Semantics of architectural connectors. In *Proc. of TAPSOFT'97*, LNCS 1214, pages 505–519. Springer-Verlag, 1997.

[7] N. Francez and I. Forman. *Interacting Processes*. Addison-Wesley, 1996.

[8] D. Garlan. Higher-order connectors. Position paper for the Workshop on Compositional Software Architectures, Jan. 1998.

[9] D. Hirsch, S. Uchitel, and D. Yankelevich. Towards a periodic table of connectors. In *Proc. of Simposio en Tecnología de Software*, Buenos Aires, 1999.

[10] A. Lopes. *Não-determinismo e Composicionalidade na Especificação de Sistemas Reactivos*. PhD thesis, Universidade de Lisboa, Jan. 1999.

[11] A. Lopes and J. L. Fiadeiro. Using explicit state to describe architectures. In *Proc. of FASE*, LNCS 1577, pages 144–160. Springer-Verlag, 1999.

[12] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Software Architecture*, pages 35–50. Kluwer, 1999.

[13] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *Proc. of ICSE*, 2000. To appear.

[14] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. on Software Eng.*, 21(4):314–335, Apr. 1995.

[15] B. Spitznagel and D. Garlan. Toward compositional construction of complex connectors. Unpublished.

[16] M. Wermelinger and J. L. Fiadeiro. Connectors for mobile programs. *IEEE Trans. on Software Eng.*, 24(5):331–341, May 1998.

[17] M. Wermelinger and J. L. Fiadeiro. Towards an algebra of architectural connectors: a case study on synchronization for mobility. In *Proc. 9th IWSSD*, pages 135–142. IEEE Computer Society Press, 1998.