

9 ARCHITECTURAL DESCRIPTION

9.1 Motivation

Although components have always been considered to be the fundamental building blocks of software systems, it is in the way that the components of a system interact that the emergence of global properties of the system resides. With no interaction there is no emergence of new behaviour and, therefore, no value to the system as a whole that is not already provided through its components in isolation.

We can safely say that most of the complexity of system construction lies in the definition of the interconnections that should regulate how components interact. Designing small, encapsulated components that, through the computations that they perform locally, provide services with certain functionalities is something that can be mastered, without much difficulty, with existing methods and development techniques. Knowing how to interconnect components so that, from the interactions, the global properties that are required of the system can emerge is a totally different matter. Most of the times, it is an error prone process. What in the literature is known as the “feature interaction problem” [114] is just a symptom of this difficulty: the emergence of “strange”, “unexpected” or “undesired” behaviour from feature composition is intrinsic to the use of methods for putting together systems from individual features as basic units of functionality; while we compose features having in mind the emergence of certain properties that constitute requirements on the behaviour of the system, it is difficult to predict which other forms of behaviour will also emerge, namely ones that are not of interest and whose “negation” is normally omitted from the requirements specification because one never thought of them being possible... Hence, situations like feature interaction are not problems that need to be solved but phenomena that are intrinsic to the way we build systems and that “just” need to be controlled. For that purpose, we need first-class representations of the interconnections.

This level of complexity is aggravated by the need to evolve systems. As the world of business in general becomes more and more aggressive and competitive, for instance as a consequence of the impact of the Internet and Wireless Technologies, companies need their information systems to be easily adaptable to changes in the business rules with which they operate, most of the time in a way that does not imply interruptions to the services that they provide. Quoting directly from [46], “... the

ability to change is now more important than the ability to create e-commerce systems in the first place. Change becomes a first-class design goal and requires business and technology architecture whose components can be added, modified, replaced and reconfigured". All this means that the "complexity" of software has definitely shifted from *construction* to *evolution*, and that methods and technologies are required that address this new level of complexity and adaptability.

Software Architectures [49,90] is a "recent" topic in Software Engineering aimed at addressing the gross decomposition and organisation of systems in which, through so-called connectors, component interactions are recognised as being first-class design entities [99]. According to [2], an architectural connector (type) can be defined by a set of *roles* and a *glue* specification. For instance, a typical client-server architecture can be captured by a connector type with two roles – client and server – which describe the expected behaviour of clients and servers, and a glue that describes how the activities of the roles are coordinated (e.g. asynchronous communication between the client and the server). The roles of a connector type can be *instantiated* with specific components of the system under construction, which leads to an overall system structure consisting of components and connector instances establishing the interactions between the components.

The similarities between architectural constructions as informally described above and parameterised programming [55] are rather striking and have been developed in [58] in the context of the emerging interest in Software Architectures. The view of architectures that is captured by the principles and formalisms used in parameterised programming is reminiscent of Module Interconnection Languages and Interface Definition Languages [54]. This perspective is somewhat different from the one we motivated above in the sense that, whereas they capture functional dependencies between the modules that need to be linked to constitute a given program, we focus instead on the organisation of the *behaviour* of systems as compositions of components ruled by protocols for communication and synchronisation.

In this chapter, we show that the mathematical "technology" of parameterised programming can also be used for the formalisation of architectural connectors in the interaction sense. The mathematical framework that we propose for formalising architectural principles is not specific to any particular Architecture Description Language (ADL). In fact, it will emerge from the examples that we shall provide that, contrarily to most other formalisations of SA concepts that we have seen, Category Theory is not another semantic domain for the formalisation of the description of components and connectors (like, say, the use of CSP in [2] or first-order logic in [89]). Instead, it provides for the very semantics of "interconnection", "configuration", "instantiation" and "composition", i.e. the principles and design mechanisms that are related to the gross modularisation of complex systems. Category Theory does this at a very abstract level because what it proposes is a toolbox that can be applied to whatever formalism is chosen for modelling the behaviour of systems as long as that formalism satisfies some structural properties. It is precisely the structural properties that make a formalism suitable for supporting architectural design that we shall make our primary focus. However, we need some concrete language in which to illustrate and motivate our approach. Not surprisingly, we will use CommUnity for that purpose.

9.2 Connectors in CommUnity

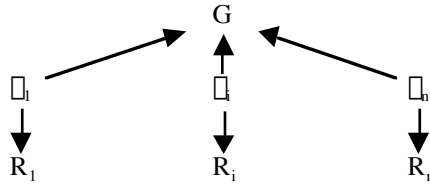
According to [2], an architectural connector (type) can be defined by a set of *roles* that can be instantiated with specific components of the system under construction, and a *glue* specification that describes how the activities of the role instances are to be coordinated. Using the mechanisms that we introduced in the previous chapter for configuration design in CommUnity, it is not difficult to come up with a formal notion of connector that has the same properties as those given in [2] for the language WRIGHT:

9.2.1 DEFINITION – architectural connector

A *connection* consists of

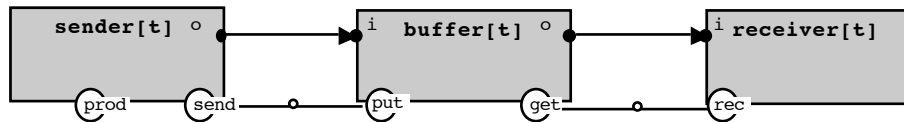
- two designs G and R , called the glue and the role of the connection, respectively;
- a signature \square and two morphisms $\square:\mathbf{dsgn}(\square) \square G, \square:\mathbf{dsgn}(\square) \square R$ connecting the glue and the role.

A *connector* is a finite set of connections with the same glue that, together, constitute a well-formed configuration (see 8.2.5).



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

For instance, asynchronous communication through a bounded channel can be modelled by a connector *ASYNC* with two connections, as depicted below using the graphical notation that we have already introduced for configurations:



The glue of *ASYNC* is the bounded buffer with FIFO discipline presented in 8.1. 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. The two roles – *sender* and *receiver* – define the behaviour required of the components to which the connector can be applied. For the *sender*, we require that no message be produced before the previous one has been processed. Its design is the one given already in section 8.1. For the *receiver*, we simply require that it have an action that models the reception of a message.

```

design receiver [t:sort] is
in i: t
do rec:[true,false] skip

```

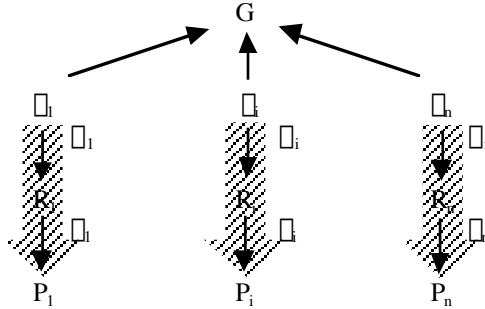
What we have described are connector *types* in the sense that they can be instantiated. More concretely, the roles of a connector type can be instantiated with specific designs. In WRIGHT [2], role instantiation has to obey a compatibility requirement expressed via the refinement relation of CSP. In CommUnity, the refinement relation is formalised through the morphisms defined in 8.3.1, leading to the following notion of instantiation:

9.2.2 DEFINITION – connector instantiation

An instantiation of a connection with role R consists of a design P together with a refinement morphism $\square: R \square P$.

An instantiation of a connector consists of an instantiation for each of its connections.

In order to define the semantics of such an instantiation, notice that, as discussed in 0, each instantiation $\square: R \square P$ of a connection can be composed with \square to define $\square; \square: \square \square \mathbf{c}\text{-sign}(P)$. Because the category of designs is coordinated over signatures (8.2.4), every such signature morphism can be lifted to a design morphism $\square; \square: \mathbf{dsgn}(\square) \square P$. 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 8.2.5, the diagram defined by the instantiation is, indeed, a configuration and, hence, has a colimit.

9.2.3 DEFINITION – semantics of connector instantiation

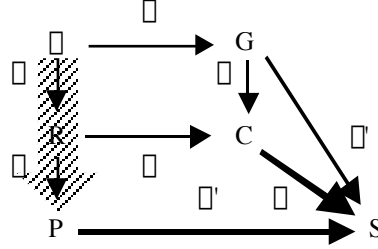
The interconnection (configuration) defined by a connector instantiation is the diagram in **c-DSGN** formed as described above by composing the role morphism of each connection with its instantiation.

The semantics of a connector instantiation is the colimit of the interconnection that it defines.

Because, as already argued, colimits in **c-DSGN** express parallel composition, this semantics agrees with the one provided in [2] for the language WRIGHT. In the next section, we shall take this analogy with WRIGHT one step further. Moreover, the

categorical formalisation makes it possible to prove that the design that results from the semantics of the instantiation is a refinement of the semantics of the connector itself.

As an example, let us consider, for simplicity, a connector with one role.

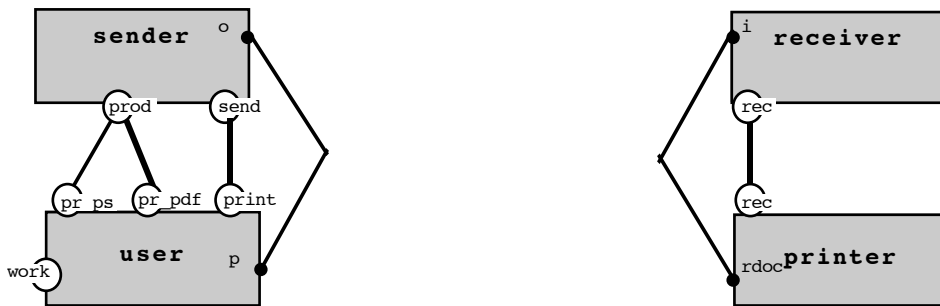


The meaning of the connector is given by the colimit of the pair $\langle \square, \square \rangle - \langle \square:R\square C, \square:G\square C \rangle$. The instantiation of the role with the component P through the refinement morphism \square is given by the colimit of $\langle \square; \square, \square \rangle - \langle \square':P\square S, \square':G\square S \rangle$. We can easily prove that there exists a refinement morphism $\square:C\square S$, which establishes the "correctness" of the instantiation mechanism. This is because all the different objects and morphisms involved can be brought into a more general category in which the universal properties of colimits guarantee the existence of the required refinement morphism. A full proof of this property can be found in [77].

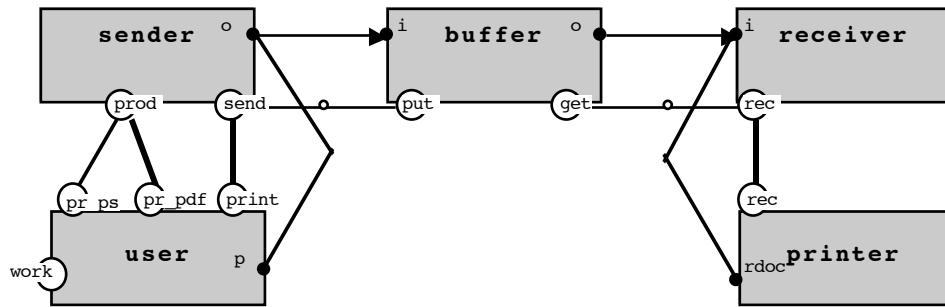
As an example, consider again the connector *ASYNCR*. We have already seen in section 0 that *sender* is refined by the design *user*. Likewise, *printer* is a refinement of *receiver* via the refinement morphism $\square:receiver\square printer$ defined by

$$\begin{aligned} \square_{ch}(i) &= rdoc \\ \square_{ac}(rec) &= rec \end{aligned}$$

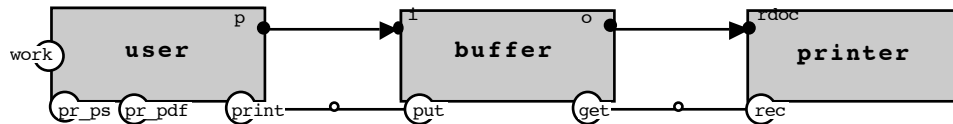
In *printer*, the reception of a message from the input channel (named *rdoc*) corresponds to downloading it into the private channel *pdoc*. This action is only enabled if the previous message has already been printed.



Therefore, we can instantiate the connector to connect the user to the printer asynchronously:



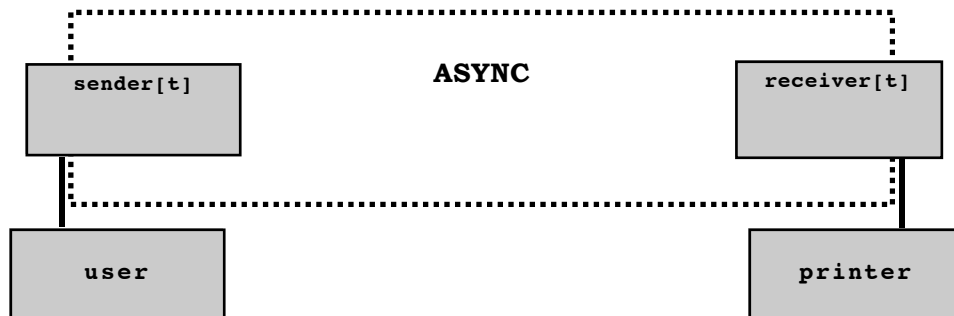
The final configuration is obtained by calculating the composition of the signature morphisms that define the two connections of *ASYN*C with the refinement morphisms \square, \square . For instance, the channel p of *user* gets connected to the input channel i of *buffer* because $\square(o)=p$ and o is connected to i of *buffer*. The resulting configuration is exactly the one we have already presented in section 8.2



In order to simplify the notation when making use of connectors, we will “hide the glue and its connections to the roles, leaving just the roles visible to suggest that they provide the “interface” of the connector:



Instantiation will be denoted as follows:



Architectural connectors are used for systematising software development by offering “standard” means for interconnecting components that can be reused from one application to another. In this sense, the “typical” glue is a program that implements a well-established pattern of behaviour (e.g. a communication protocol) that can be superposed to existing components of a system through the instantiation of the roles of the connector.

However, architectures also fulfil an important role in supporting a high-level description of the organisation of a system by identifying its main components and the way these components are interconnected. An early identification of the architectural elements intended for a system will help to manage the subsequent design phases according to the organisation that they imply, identifying opportunities for reuse or the integration of third-party components. From this point of view, it seems useful to allow for connectors to be based on glues that are not yet fully developed as programs but for which concrete commitments have already been made to determine the type of interconnection that they will ensure. For instance, at an early stage of development, one may decide on adopting a client-server architecture without committing to a specific protocol of communication between the client and the server. This is why, in the definition of connector in CommUnity, we left open the possibility for the glue not to be a program but a design in general.

However, in this more general framework, we have to account for the possible refinements of the glue. What happens if we refine the glue of a connector that has been instantiated to given components of a system? Is the resulting design a refinement of the more abstract design from which we started? More generally, how do connectors propagate through design, be it because the instances of the roles are refined or the glue is refined? One of the advantages of using Category Theory as a mathematical framework for formalising architectures is that answers to questions like these can be discussed at the right level of abstraction. Another advantage is that the questions themselves can be formulated in terms that are independent of any specific ADL and answered by characterising the classes of ADLs that satisfy the given properties. This is what we will do in later sections.

9.3 Examples

We now present more examples of connectors, namely some that we will need in later sections for illustrating algebraic operations on connectors. Their application will be illustrated on examples related to a case study on mobility [94]: One or more carts move continuously in the same direction on a U -units long circular track. A cart advances one unit at each step. Along the track there are stations. There is at most one station per unit. Each station corresponds to a check-in counter or to a gate. Carts take bags from check-in stations to gate stations. All bags from a given check-in go to the same gate. A cart transports at most one bag at a time. When it is empty, the cart picks a bag up from the nearest check-in. Carts must not bump into each other. Carts also keep a count of how many laps they have done, starting at some initial location.

The program that controls a cart is

```

design cart is
  in   idest: 0..U - 1, ibag: int
  out  obag, laps : int
  prv  loc: 0..U - 1, dest: -1..U - 1, initloc : int
  do   move: loc  $\neq$  dest  $\square$  loc := loc +U 1 || laps := if(loc=initloc,laps+1,laps)
  []   get: dest = -1  $\square$  obag := ibag || dest := idest
  []   put: loc = dest  $\square$  obag := 0 || dest := -1

```

where $+_U$ is addition modulo U .

Locations are represented by integers from zero to the track length minus one. Bags are represented by integers, the absence of a bag being denoted by zero. Whenever the cart is empty, its destination is an unreachable location (-1), so that the cart keeps moving until it gets a bag and a valid gate location through action *get*. When it reaches its destination, the cart unloads the bag through action *put*. Notice that, because input channels may be changed arbitrarily by the environment, the cart must copy their values to output/private channels to make sure the correct bag is unloaded at the correct gate.

A check-in counter manages a queue of bags that it loads one by one onto passing carts.

```

design check-in is
  out bag: int, dest:  $0..U - 1$ ,
  prv loc:  $0..U - 1$ , next: bool, q: list(int)
  do    new: q#[]  $\square$  next  $\square$  bag := head(q)  $\parallel$  q := tail(q)  $\parallel$  next := false
  []    put:  $\square$ next  $\square$  next := true

```

Channel *next* is used to impose sequentiality among the actions. In a configuration in which a cart is loading at a gate, the *put* action must be synchronised with a cart's *get* action and channels *bag* and *dest* must be shared with *ibag* and *idest*, respectively.

A gate keeps a queue of bags and adds each new bag to the tail.

```

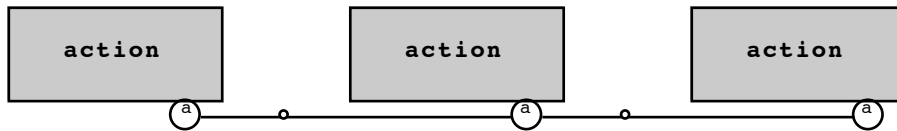
design gate is
  in    bag : int
  prv   loc:  $0..U - 1$ , q: list(int)
  do    get: q := q.bag

```

In a configuration in which a cart is unloading at a gate, action *get* of the gate must be synchronised with the cart's *put* action, and channel *bag* must be shared with *obag*.

9.3.1 SYNCHRONISATION

We begin with the connector that allows us to synchronise two actions of different components. A plain cable would suffice for this purpose, but it is not able to capture the general case of transient synchronisation [94]. Having already a connector for the simpler case makes the presentation more uniform. The glue of the synchronisation connector and the roles are the same:



with

```

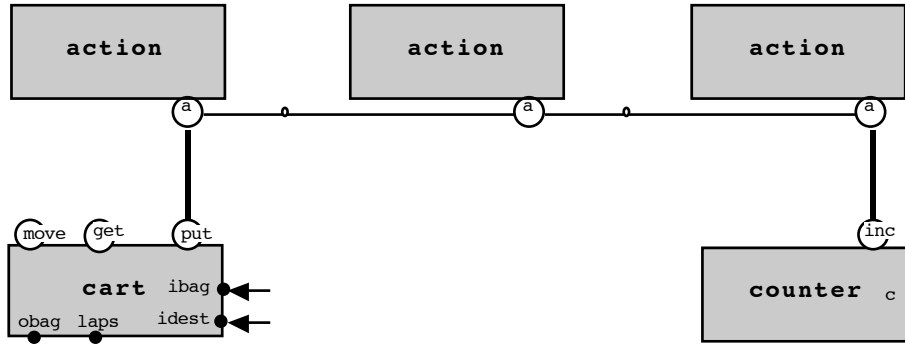
design action is
  do a: true,false  $\square$  skip

```

Notice that the action has the least deterministic specification possible: its guard is given the widest possible interval and no commitments are made on its effects. Hence, it can be refined by any action.

According to the colimit semantics of connectors, when the two roles are instantiated with particular actions a_1 and a_2 of particular components, the components have to synchronise with each other every time one of them wants to execute the corresponding action: either both execute the joint action, or none executes.

As an example of using this connector, if we wish to count how often a cart unloads, we can monitor its *put* action with a counter:

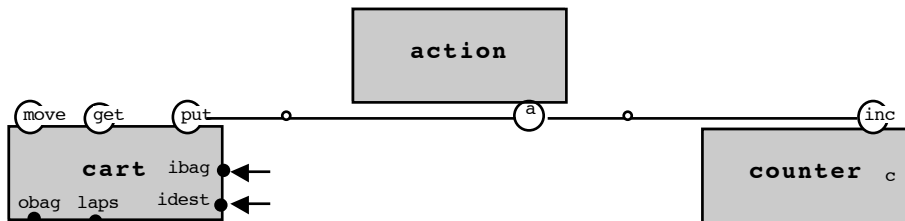


```

design counter is
  out   c:int
  do   inc: true [] c := c+1
  []    reset: true [] c := 0

```

According to what was defined in 9.2.2, the interconnection defined by this instantiation is the following configuration:



The resulting semantics is the synchronisation of *put* and *inc*. The following program captures the joint behaviour of the interconnected components:

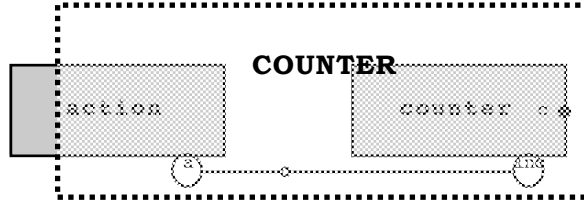
```

design monitored_cart is
  in   idest: 0..U - 1, ibag: int
  out   obag, laps, unloads : int
  prv   loc: 0..U - 1, dest: -1..U - 1, initloc : int
  do   move: loc ≠ dest [] loc := loc +U 1 || laps := if(loc=initloc,laps+1,laps)
  []   get: dest = -1 [] obag := ibag || dest := idest
  []   put|inc: loc = dest [] obag := 0 || dest := -1 || unloads := unloads+1

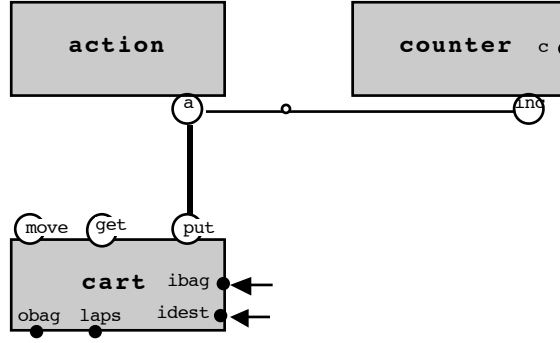
```

9.3.2 MONITORS

The counter used in 9.3.2 can be “wrapped” as the glue of a connector that offers a role corresponding to the action to be monitored:



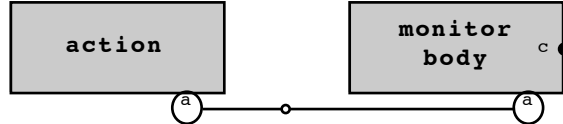
The monitored cart results from the instantiation of the role of this connector with the cart through the same refinement morphism as before, i.e. connecting *put* to *a*:



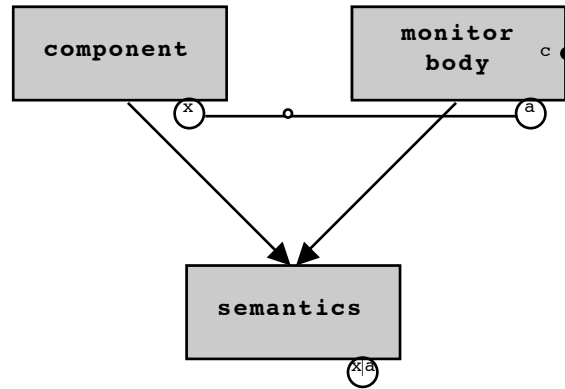
Indeed, the defined interconnection is the same for both instantiations.

As a connector, the counter enjoys a very interesting property. Because *inc* is always enabled, its synchronisation with *put* does not interfere with the behaviour of the cart. This can be witnessed in the guard of the joint action *put/inc*, which is the same as for *put*: $loc = dest$.

Hence, we can say that the counter is *monitoring* the cart or that, as a connector, the counter is a monitor. Generically, we can characterise a monitor as a connector of the form:



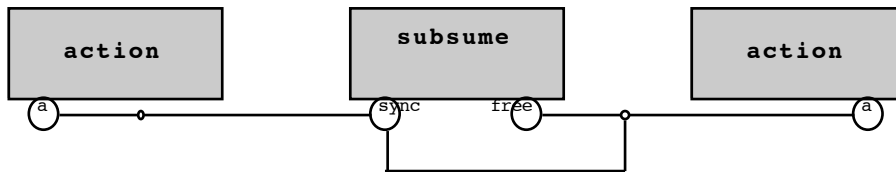
where $L(a)=U(a)=true$ and $R(a)$ is satisfiable for action a of *monitor_body*. The output channel c is used for transmitting the information that is being gathered about the execution of the action that will instantiate a . The property of non-interference with the behaviour of the monitored component can be made more precise as follows: in the semantics of the instantiation, the co-cone morphism that originates in the component is “spectative” in the sense that any model of the component can be “expanded” to a model of the interconnection. The property of being “spectative”, a term that was applied in [71] to superpositions that leave guards unchanged, has to be formalised according to the notion of model that one is adopting for the underlying design language. Because we have not detailed a specific semantics for CommUnity, we shall not expand on this issue much further. An extensive account of this property can be found in [44] for a trace-based semantics of CommUnity, which includes an analysis of the relationship between “non-interference” and the notions of “model-expansion” and “conservative extension” that have been used for first-order logic specifications of abstract data types [104,106].



9.3.3 SUBSUMPTION

Intuitively, synchronisation corresponds to an "equivalence" between the occurrence of two actions: the occurrence of each of the actions "implies" the occurrence of the other. In many circumstances, we are interested in one of the implications. For instance, to avoid a cart colliding with a cart that is right in front of it, we only need one implication: if the first one moves, so must the one in front. The other implication is not necessary. The analogy with implication also extends to the counter-positive: if the front car cannot move, for instance because it is (un)loading a bag, then neither can the rear one. We call this "one-way" synchronisation *action subsumption*.

The subsumption connector is given by the following configuration:



where the glue is now given by

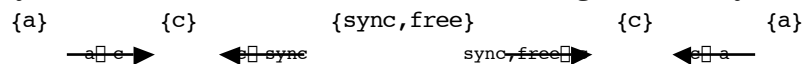
```

component subsume is
  do sync: true,false  $\square$  skip
  [] free: true,false  $\square$  skip

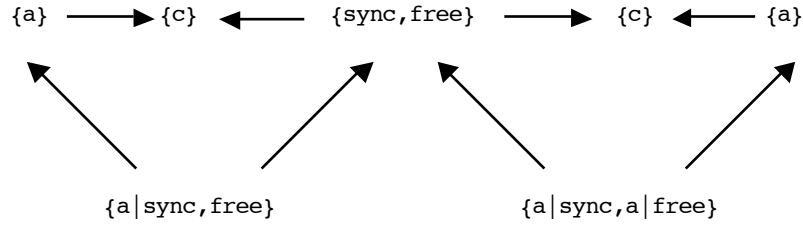
```

Notice that although the two roles are the same, the connector is not symmetric because the connections treat the two role actions differently: the right-hand one may be executed alone at any time, while the left-hand one must co-occur with the right-hand one, through action *sync*. Indeed, the semantics of the connector generates the following synchronisation sets: $\{a_1, sync, a_2\}$ and $\{free, a_2\}$ where a_1 is a renaming of the left-hand role action and a_2 is a renaming of the right-hand role action. Hence, action a_1 can only occur together with a_2 but that a_2 can occur without a_1 .

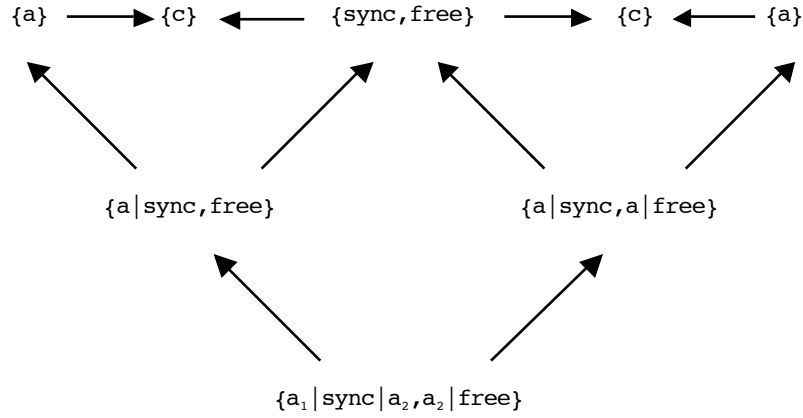
In order to understand how subsumption works, let us detail the construction of the semantics of the connector, starting with the corresponding diagram of signatures. Because only actions are involved, we will take the diagram directly over pointed sets:



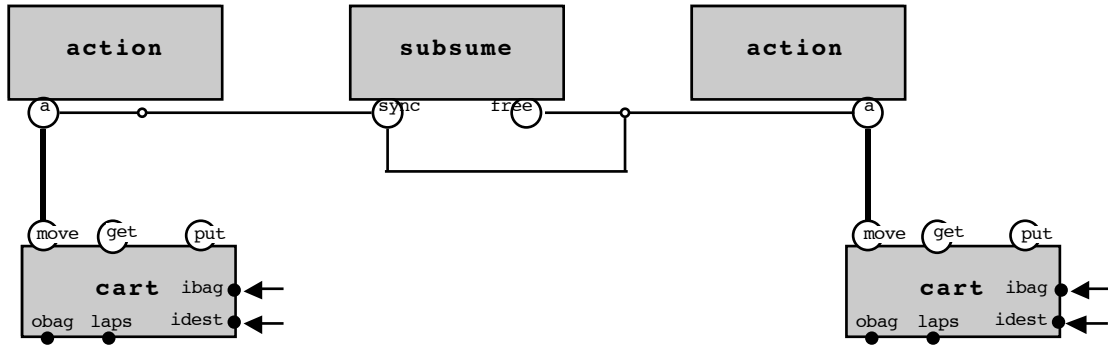
The limit of this diagram can be computed by first taking the two pullbacks:



And finally the middle pullback:



As an example of the application of this connector, consider the following instantiation.

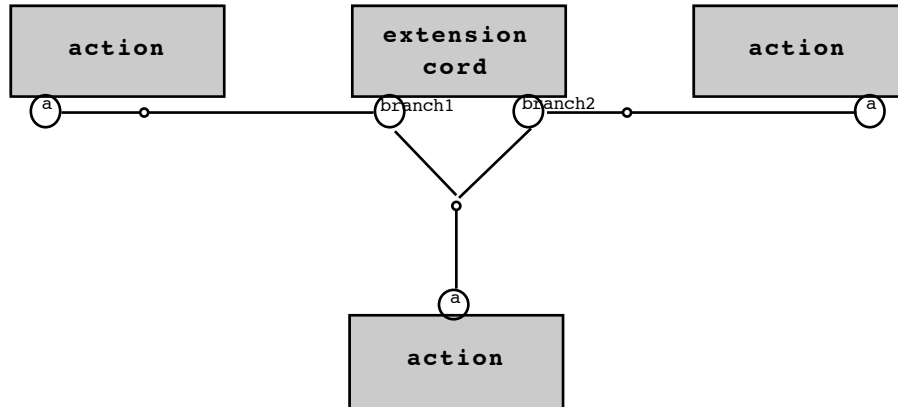


Any movement of the cart on the left implies a movement of the cart on the right. Hence, this instantiation can be used to prevent collision when the left cart is too close behind the right cart.

9.3.4 EXTENSION CORD

A generalisation of the subsumption and synchronisation connectors is to allow an action to synchronise, independently, with two actions of two different programs,

achieving an effect similar to an extension chord that one can use to connect two independent devices to the same power supply.



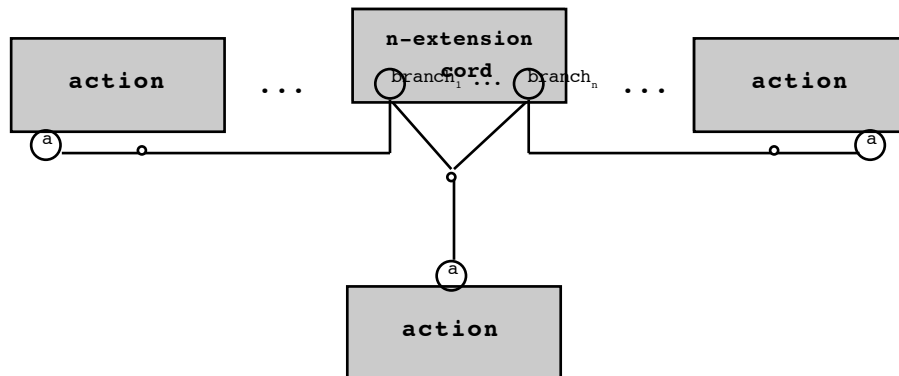
```

design extension cord is
do  branch1: true, false □ skip
      branch2: true, false □ skip

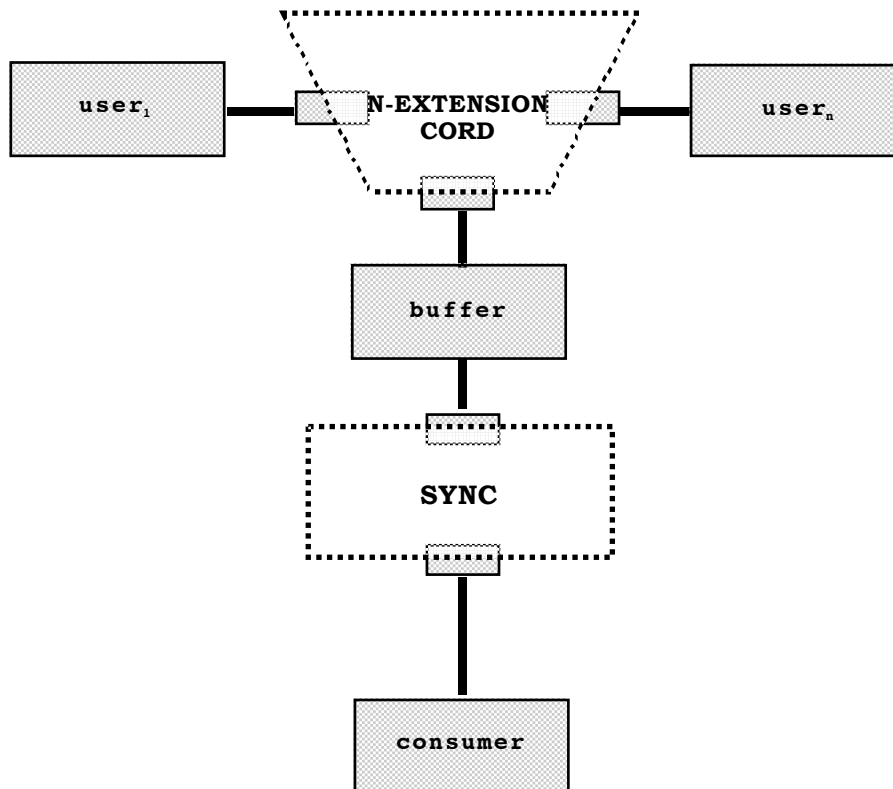
```

If we instantiate the left-hand role with an action a_1 and the right-hand role with an action a_2 , and the middle role with an action b , the semantics of the interconnection, as obtained through the colimit, is given by two synchronisation sets: $\{a_1, \text{branch}_1, b\}$ and $\{a_2, \text{branch}_2, b\}$. Notice that action b will always occur simultaneously with either a_1 or a_2 but not with both.

This connector can be generalised to any finite number n of ramifications, giving rise to an “n-extension cord”:

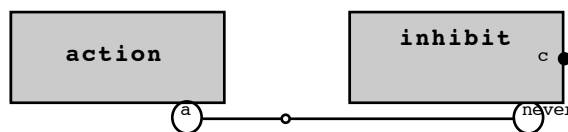


It allows for a server to be connected simultaneously, but independently, to a fixed maximum number of clients. Hence, a system of n -users and one consumer communicating through a buffer can be configured as follows:



9.3.5 INHIBITION

Another basic connector type is the one that allows us to inhibit an action by making its guard false. This is useful when, for some reason, we need to prevent an action from occurring but without having to reprogram the component. Indeed, the mechanism of superposition that we have used as a semantics for the application of architectural connectors allow us to disable an action without changing the guard directly but by just inducing this effect: it suffices to synchronise the action with one that has a false guard.



```
design inhibit is
do never: false □ skip
```

When the role is instantiated with an action with guard B , the result of the interconnection is the same action guarded by $B \sqcap \text{false}$. This connector can be generalised to arbitrary conditions with which one may strengthen the guards of given actions. The inhibitor just has to be provided with the data that is necessary to compute the condition C that will strengthen the guard, for instance through the use of input channels through which we can select the sources of the information that will disable the action.

```

design inhibit(C) is
in ...
do never: C  $\square$  skip

```

The result of instantiating the role with an action with guard B is the same action guarded by $B \sqcap C$.

9.4 An ADL-independent notion of connector

The notion of connector that we presented in 9.2 can be generalised to design formalisms other than CommUnity. In this section, we shall discuss the properties that such formalisms need to satisfy for supporting the architectural concepts and mechanisms that we have illustrated for CommUnity.

Before embarking on this discussion, we need to fix a framework in which designs, configurations and relationships between designs, such as refinement, can be formally described.

9.4.1 DEFINITION – design formalism

A formalism supporting system design consists of:

- a category **c-DESC** of component descriptions in which systems of interconnected components are modelled through diagrams;
- for every set CD of component descriptions, a set $Conf(CD)$ consisting of all well-formed configurations that can be built from the components in CD . Each such configuration is a diagram in **c-DESC** that is guaranteed to have a colimit. Typically, $Conf$ is given through a set of rules that govern the interconnection of components in the formalism.
- a category **r-DESC** with the same objects as **c-DESC**, but in which morphisms model refinement, i.e. a morphism $[S] \rightarrow [S']$ in **r-DESC** expresses that S' refines S , identifying the design decisions that lead from S to S' . Because the description of a composite system is given by a colimit of a diagram in **c-DESC** and, hence, is defined up to an isomorphism in **c-DESC**, refinement morphisms must be such that descriptions that are isomorphic in **c-DESC** refine, and are refined exactly by, the same descriptions.

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

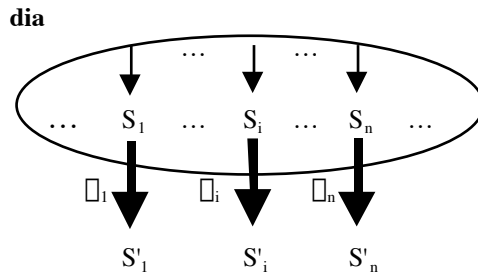
In the context of this categorical framework, we shall now discuss the properties that are necessary for supporting Software Architectures. A key property for supporting architectural design is a clear separation between the description of individual components and their interaction in the overall system organisation. In other words, the formalism must support the separation between what, in the description of a system, is responsible for its computational aspects and what is concerned with coordinating the interaction between its different components.

In the case of CommUnity, as we have seen, only signatures are involved in interconnections. The body of a component design describes its functionality and, hence, corresponds to the computational part of the design. At the more general level that we are discussing, we shall take the separation between coordination and computation to be materialised through a functor **sign: c-DESC** \rightarrow **SIGN** mapping descriptions to signatures, forgetting their computational aspects. The fact that the computational side does not play any role in the interconnection of systems can be captured by requiring **sign** to be coordinated in the sense of **Error! Reference source not found.**

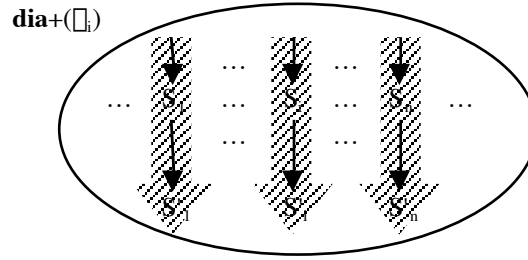
Another crucial property for supporting architectural design is in the interplay between structuring systems in architectural terms and refinement. We have already pointed out that one of the goals of Software Architectures is to support a view of the gross organisation of systems in terms of components and their interconnections that can be carried through the refinement steps that eventually lead to the implementation of all its components. Hence, it is necessary that the application of architectural connectors to abstract designs, as a means of making early decisions on the way certain components need to be coordinated, will not be jeopardised by subsequent refinements of the component designs towards their final implementations. Likewise, it is desirable that the application of a connector may be made on the basis of an abstract design of its glue as a means of determining main aspects of the required coordination without committing to the final mechanisms that will bring about that coordination.

One of the advantages of the categorical framework that we have been proposing is that it makes the formulation of these properties relatively easy, leading to a characterisation of the design formalisms that support them in terms of the structural properties that we have been discussing. For instance, we have already seen that, in the situations in which refinement morphisms map directly to signature morphisms, we may simply put together, in a diagram of signatures, the morphisms that define the interactions and the morphisms that establish the refinement of the component descriptions.

More precisely, in the situations in which there exists a forgetful functor **r-sign: r-DESC** \rightarrow **SIGN** that agrees with the coordination functor **sign** on signatures – i.e. **r-sign(S)=sign(S)** for every $S:c\text{-DESC}$ – given a well-formed configuration diagram **dia** of a system with components S_1, \dots, S_n and refinement morphisms $\square_i: S_i \rightarrow S'_i$ $i=1..n$,



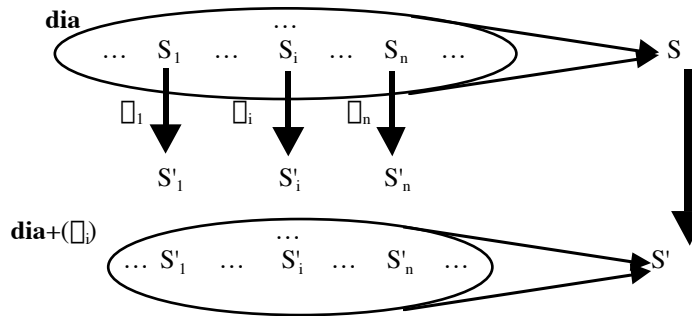
we can obtain a new diagram in **c-DESC** and, hence, a new configuration, by composing the morphisms **r-sign**(\square_i) with those in **dia** that originate in channels (signatures) and have the S'_i as targets.



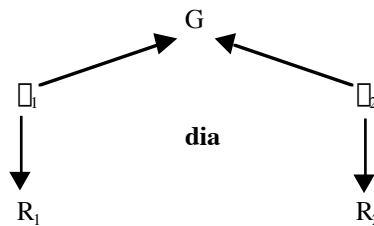
In general, it may be possible to propagate the interactions between the components of a system when their descriptions are replaced by more concrete ones even when refinement morphisms do not map to signature morphisms. This more general situation can be characterised as follows:

9.4.2 DEFINITION – compositional design formalism

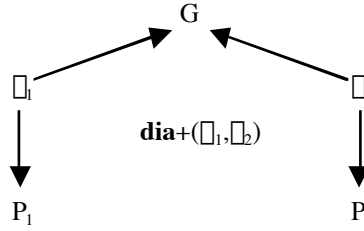
We say that a design formalism is *compositional* whenever, for every well-formed configuration \mathbf{dia} involving descriptions $\{S_1, \dots, S_n\}$ and refinements morphisms $\{\Box_i: S_i \rightarrow S'_i; i \in 1..n\}$, there is a well-formed configuration diagram $\mathbf{dia}+(\Box_i)$ that characterises the system obtained by replacing the S_i by their refinements and satisfies the following correctness criterion: the colimit of $\mathbf{dia}+(\Box_i)$ provides a refinement for the colimit of \mathbf{dia} .



When we consider the specific case of the configurations obtained by direct instantiation of an architectural connector, this property reflects the compositionality of the connector as an operation on configurations. Compositionality ensures that the semantics of the connector is preserved (refined) by any system that results from its instantiation. For instance, in the case of a binary connector



and given instantiations $\Box_1:R_1 \Box P_1$ and $\Box_2:R_2 \Box P_2$, compositionality means that the description returned by the colimit of **dia** is refined by the description returned by the colimit of **dia**+(\Box_1, \Box_2).



Likewise, compositionality guarantees that if a connector with an abstract glue G is applied to given designs, and the glue is later on refined through a morphism $\Box:G \Box G'$, the description that is obtained through the colimit of **dia**+ \Box is a refinement of the semantics of the original instantiation. In fact, we can consider the refinement of the glue to be a special case of an operation on the connector that delivers another connector – a refinement of the original one in the case at hand.

9.4.3 DEFINITION – architectural school

A design formalism $F=\langle \mathbf{c-DESC}, \mathbf{Conf}, \mathbf{r-DESC} \rangle$ supports architectural design, and is called an architectural school, iff

- **c-DESC** is coordinated over a category **SIGN** through a functor **sign: c-DESC** \Box **SIGN**
- F is compositional

9.5 Adding abstraction to connectors

The mathematical framework that we presented in the previous sections provides not only an ADL-independent semantics for the principles and techniques that can be found in existing approaches to Software Architectures, but also a basis for extending the capabilities of existing ADLs. Until the end of this chapter, we will present and explore some of the avenues that this mathematical characterisation has opened, hoping that the reader will want to explore them even further, or find new ones!

As already mentioned, the purpose of the roles in a connector is to impose restrictions on the local behaviour of the components that are admissible as instances. In the approach to architectural design outlined in the previous sections, this is achieved through the notion of correct instantiation via refinement morphisms. As also seen above, roles do not play any part in the calculation of the resulting system. They are used only for defining what a correct instantiation is. This separation of concerns justifies the adoption of a more declarative formalism for the specification of roles, namely one in which it is easier to formulate the properties required of components to be admissible instances.

In this section, we are going to place ourselves in the situation in which the glues are designs, the roles are specifications, and the instantiations of the roles are,

again, designs. We are going to consider that specifications are given as a category **SPEC**, e.g. the category of theories of a logic formalised as an institution – see 6.5.11 above. We take the relationship between specifications and designs to be captured through the following elements:

- a functor **spec:SIGN** **SPEC** mapping signatures and their morphisms to specifications.

The idea behind the functor **spec** is that, just like, through **c-desc** (the left adjoint of **sign**) signatures provide the means for interconnecting designs, they should also provide means for interconnecting specifications. Hence, every signature generates a canonical specification – the specification of a cable. However, it is not necessary for **spec** to satisfy as many structural properties as **c-desc** because, for the purposes of this section, we are limiting the use of specifications to the definition of connector roles. Naturally, if we wish to address architecture building at the specification level, then we will have to require **SPEC** to satisfy the properties that we discussed in section 9.4.

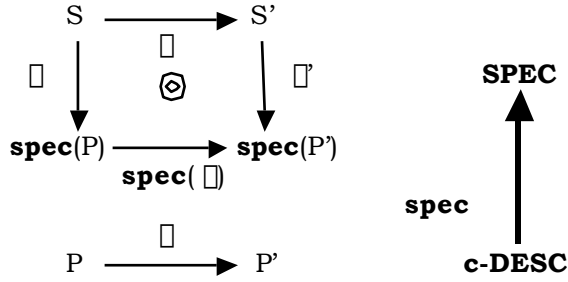
- a satisfaction relation \models between design morphisms and specification morphisms satisfying the following properties:

1. If $\square:P\square P' \models \square:S\square S'$, then $id_P \models id_S$ and $id_{P'} \models id_{S'}$.
2. If $\square_1:P_1\square P_2 \models \square_1:S_1\square S_2$ and $\square_2:P_2\square P_3 \models \square_2:S_2\square S_3$ then $\square_1;\square_2 \models \square\square_1;\square_2$
3. Let **s:I** **SPEC** be a diagram of specifications and **p:I** **c-DESC** a diagram of designs with the same shape such that, for every edge $f:i \rightarrow j$ in **I**, $\mathbf{p}_f:\mathbf{p}_i \rightarrow \mathbf{p}_j \models \mathbf{s}_f:\mathbf{s}_i \rightarrow \mathbf{s}_j$. We require that, if **p** admits a colimit $\square_i:\mathbf{p}_i \rightarrow P$, then **s** admits a colimit $\square_i:\mathbf{s}_i \rightarrow S$ such that, for every node $i:I$, $\square_i \models \square$.
4. If $id_P \models id_S$ and $\square:P\square P'$ is a refinement morphism, then $id_{P'} \models id_S$
5. For every signature \square , $id_{\mathbf{c}\text{-desc}(\square)} \models id_{\mathbf{spec}(\square)}$

The satisfaction relation is defined directly on morphisms because our ultimate goal is to address interconnections, not just components. Satisfaction of component specifications by designs is given through the identity morphisms. The properties required of the satisfaction relation address its compatibility with the categorical constructions that we use, namely composition of morphisms and colimits. The last two properties mean that refinement of component designs leaves the satisfaction relation invariant, and that the design (cable) generated by every signature satisfies the specification (cable) generated by the same signature.

We have already seen in 5.1.3 that a satisfaction relation between specifications and programs can be defined if we are able to establish a functor **r-DESC** **SPEC** that maps every design to the maximum set of properties that it “satisfies”. Intuitively, this functor should be an extension of **spec** in the sense that, for every signature \square , **spec**(\square) should be isomorphic to **spec(c-desc(\square))**, so we will call it **spec**. In this case, it makes sense to define

$$\square:P\square P' \models \square:S\square S' \text{ iff there exist } \square:S\square \mathbf{spec}(P) \text{ and } \square':S'\square \mathbf{spec}(P') \text{ s.t. } \square;\mathbf{spec}(\square)=\square';\square'$$



Notice that we get $id_p \models id_s$ iff there exists $\square : S \square \text{spec}(P)$, which is exactly how satisfaction was defined in 5.1.3. Moreover, properties 1-4 are obtained free from the fact that **spec** is a functor. Condition 5 is satisfied because **spec**(\square) has been required to be isomorphic to **spec**(**c-desc**(\square)).

Given such a setting, we generalise the notion of connector as follows:

9.5.1 DEFINITION – (generalised) architectural connector

A (*generalised*) *connection* consists of

- a design G and a specification R , called the glue and the role of the connection, respectively;
- a signature \square and two morphisms $\square : \mathbf{c-desc}(\square) \square G$, $\square : \mathbf{spec}(\square) \square R$ in **c-DESC** and **SPEC**, respectively, connecting the glue and the role via the signature (cable).

A (*generalised*) *connector* is a finite set of connections with the same glue.

An instantiation of a connection with signature \square and role morphism \square consists of a design P and a design morphism $\square : \mathbf{c-desc}(\square) \square P$ such that $\square \models \square$.

An instantiation of a connector consists of an instantiation for each of its connections. An instantiation is said to be correct if the diagram defined by the instantiation morphisms and the glue morphisms is a well-formed configuration. The colimit of this configuration defines the semantics of the instantiation, guaranteed to exist if the instantiation is correct.

Although the generalisation seems to be quite straightforward, we do not have an immediate generalisation for the semantics of connectors. This is because the glue is a design and the role is a specification, which means that a connector does not provide us with a diagram like in the homogeneous case that we studied in section 9.2. However, if we are provided with a specification for the glue, we can provide semantics for the connector at the specification level:

9.5.2 DEFINITION – complete architectural connector

A *complete connection* consists of

- a design G and a specification R , called the glue and the role of the connection, respectively;
- a signature \square and two morphisms $\square : \mathbf{c-desc}(\square) \square G$, $\square : \mathbf{spec}(\square) \square R$ in **c-DESC** and **SPEC**, respectively, connecting the glue and the role via the signature (cable);
- a specification S and a morphism $\square : \mathbf{spec}(\square) \square S$ such that $\square \models \square$. Notice that this means that the design G satisfies the specification S .

A *complete connector* is a finite set of complete connections with the same glue design and specification. Its semantics is given by the colimit, if it exists, of the **SPEC**-diagram defined by the \square_i and the \square .

Notice that we obtain the following property:

9.5.3 PROPOSITION

The semantics of the instantiation of a complete connector satisfies the semantics of the connector.

An illustration of an abstract architectural school can be given in terms of a first-order extension¹² of the linear temporal logic that we studied in 3.5. As an example, we present below the specifications of a typical sender and receiver of messages through a pipe.

```

specification pipe_sender[t:sort] is
signature      o:t, eof:bool, send
axioms         eof    G(□send □ eof)

specification pipe_receiver[t:sort] is
signature      cl, eof:bool, rec
axioms         cl    G(□rec □ cl)
                  ((eof    Geof) □ (eof □ □cl))    (□recUcl)

```

The specification *pipe_sender* accounts, through *send*, for the transmission of data of sort *t* through a channel *o*. The end of data transmission is signalled through channel *eof*: the axiom requires that *eof* be stable (remains true once it becomes true) and transmission of messages to cease once *eof* becomes true.

The specification *pipe_receiver* accounts, through *rec*, for the reception of data through the channel *i*. The other means of interaction with the environment is concerned with the closure of communication. Through channel *eof*, a Boolean can be received that indicates if transmission along *i* has ceased. Closure of communication is signalled in the channel *cl*. The first axiom requires that *cl* be stable and the reception of messages to cease once *cl* becomes true. The second axiom expresses that, if the information received through *eof* is stable, the receiver is obliged to close the communication as soon as it is informed that there will be no more data. However, the receiver may decide to close the communication before that.

It remains to capture the relationship between specifications and designs in CommUnity.

First we notice that every design signature \square can be mapped into a temporal signature by forgetting the different classes of channels and actions, as well as the write frames of actions. Then, we notice that part of the semantics of CommUnity designs can be encoded in LTL:

- for every action *g*, the negation of $L(g)$ is a blocking condition for its occurrence ($g \square L(g)$);
- for every local channel *v*, $D(v)$ consists of the set of actions that can modify it –

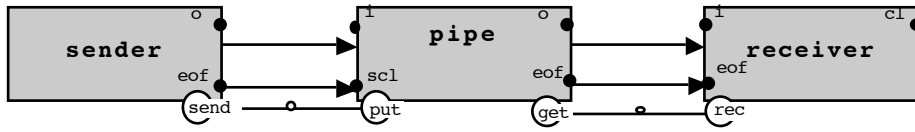
$$g \square D(v) \quad g \quad (Xv=v)$$

¹² The extension is straightforward: the reader is encouraged to formalise it as an exercise, for which [38,39,66] can be consulted.

- for every action g , the condition $R(g)$ holds in every state in which g is executed – $\langle g \rangle \Box [R(g)]$ where \Box is a translation that replaces any primed variable v' by the term (Xv) ;
- private actions that are infinitely often enabled are guaranteed to be selected infinitely often – $(GFU(g) \rightarrow GFg)$

This encoding extends to refinement morphisms, establishing a functor **r-DESC** \square **THEO**_{FOLTL}, from which we can define a satisfaction relation. Notice that the fact that refinement of CommUnity designs is contravariant on the upper bound of actions is crucial: refinement morphisms are liveness preserving but interconnection morphisms are not!

In order to illustrate the generalised notion of connector in this setting, we present below the connector *cpipe*,



where *pipe* is the design presented below.

```

design pipe [t:sort, bound:t] is
in   i:t, scl:bool
out  o:t, eof:bool
prv  rd: bool, b: list(t)
do    put: true  $\square$  b:=b.i
[] prv next: |b|>0  $\square$  rd  $\square$  o:=head(b)  $\square$  b:=tail(b)  $\square$  rd:=true
[]    get: rd  $\square$  rd:=false
[] prv sig: scl  $\square$  |b|=0  $\square$  eof:=true

```

This design, which is the glue of the connector, models a buffer with unlimited capacity and a FIFO discipline. It signals the end of data to the consumer of messages as soon as the buffer gets empty and the sender of messages has already informed, through the input channel *scl*, that it will not send anymore messages.

The two roles – the specifications *sender* and *receiver* introduced before – define the behaviour required of the components to which the connector *cpipe* can be applied. It is interesting to notice that, due to the fact that LTL is more abstract than CommUnity, we were able to abstract away completely the production of messages in the role *sender*. In the design of *sender* presented in 8.1 we had to consider an action modelling the production of messages.

We can now generalise these constructions even further by letting the connections use different specification formalisms and the instantiations to be performed over components designed in different design formalisms. In this way, it will be possible to support the reuse of third-party components, namely legacy systems, as well as the integration of non-software components in systems, thus highlighting the role of architectures in promoting a structured and incremental approach to system construction and evolution.

However, to be able to make sense of the interconnections, we have to admit that all the design formalisms are coordinated over the same category of signatures. That is to say, we assume that the integration of heterogeneous components is made at the level of the coordination mechanisms, independently of the way each component brings about its computations. Hence, we will assume given

- a family $\{\mathbf{DSGN}_d\}_{d:D}$ of categories of designs, all of which are coordinated over the same category \mathbf{SIGN} via a family of functors $\{\mathbf{dsgn}_d\}_{d:D}$,
- a family $\{\mathbf{SPEC}_c\}_{c:C}$ of categories of specifications together with a family $\{\mathbf{spec}_c, \mathbf{SPEC}_c \sqsubseteq \mathbf{SIGN}\}_{c:C}$ of functors,
- a family $\{\models_s\}_{s:S}$ of satisfaction relations, each of which relates a design formalism $d(s)$ and a specification formalism $c(s)$. We do not require S to be the cartesian product $D \sqsubseteq C$, i.e. there may be pairs of design and specification formalisms for which no satisfaction relation is provided.

Given such a setting, we generalise the notion of connector as follows:

9.5.4 DEFINITION – (heterogeneous) architectural connector

A *heterogeneous connection* consists of

- a design formalism \mathbf{DSGN}_d and a specification formalism \mathbf{SPEC}_c ;
- a design $G:\mathbf{DSGN}_d$ and a specification $R:\mathbf{SPEC}_c$, called the glue and the role of the connection, respectively;
- a signature $\square:\mathbf{SIGN}$ and two morphisms $\square:\mathbf{dsgn}_d(\square) \rightarrow G, \square:\mathbf{spec}_c(\square) \rightarrow R$ in \mathbf{DSGN}_d and \mathbf{SPEC}_c , respectively, connecting the glue and the role via the signature (cable).

An heterogeneous connector is a finite set of connections with the same glue.

An instantiation of a heterogeneous connection with specification formalism \mathbf{SPEC}_c , signature \square and role morphism \square consists of

- a design formalism \mathbf{DSGN}'_d and a satisfaction relation $\models_{\langle d', c \rangle}$ between \mathbf{DSGN}'_d and \mathbf{SPEC}_c such that $\langle d', c \rangle \sqsubseteq \square$;
- a design P and a design morphism $\square:\mathbf{dsgn}_d'(\square) \rightarrow P$ such that $\square \models_\square \square$.

An instantiation of a connector consists of an instantiation for each of its connections.

Given that in the context of heterogeneous connectors we have to deal with several design formalisms, providing a semantics for the resulting configurations requires a homogeneous formalism to which all the design formalisms can be mapped. Clearly, because we want the heterogeneity of formalisms to be carried through to the implementations in order to be able to support the integration of legacy code, third-party components and even non-software components, this common formalism cannot be at the same level as that of designs, and the mapping cannot be a simple translation. What seems to make more sense is to choose a behaviour model that can be used to provide a common semantics to all the design formalisms so that the integration is not performed at the level of the descriptions but of the behaviours that are generated from the descriptions. Indeed, when we talk about the integration of heterogeneous components, our goal is to coordinate their individual behaviours. An architecture should provide precisely the mechanisms through which this coordination is effected.

