

8 COMMUNITY

8.1 A language for program design

CommUnity is a language similar to Unity [19] and Interacting Processes [47] that was initially developed in [44] to show how “programs” fit into Goguen’s categorical approach to General Systems Theory. Since then, the language and the design framework have been extended to provide a formal platform for testing ideas and experimenting techniques for the architectural design of open, reactive, and reconfigurable systems.

One of the extensions that we have made to CommUnity since its original definition in [44] concerns the support for higher levels of design. At such levels of design, the architecture of the system is given in terms of components that are not necessarily programs but abstractions of programs – called *designs* – that can be *refined* into programs in later stages of the development process. Designs may also account for components of the real-world with which the software components will be interconnected. Typically, such abstractions derive from requirements that have been specified in some logic or other mathematical models of the behaviour of real-world components.

The goal of supporting abstraction is not only to address a typical stepwise approach to software *construction*, but also the definition of an architectural design layer that is close enough to the application domain for the evolution of the system to be driven directly as a reflection of the changes that occur in the domain. An important part of this evolution may consist of changes in the nature of components, with real-world components being replaced or controlled by software components, or software components being reprogrammed in another language.

The support for abstraction in CommUnity is twofold. On the one hand, designs account for what is usually called *underspecification*, i.e. they are structures that do not denote unique programs but collections of programs. On the other hand, designs can be defined over a collection of data types that do not correspond necessarily to those that will be available in the final implementation platform. Therefore, there are two refinement procedures that have to be accounted for in CommUnity. On the one hand, the removal of underspecification from designs in order to define programs over the layer of abstraction defined by the data types that have been used. On the other hand, the reification of the data types in order to bring programs into the target implementation environment.

The choice of data types determines, essentially, the nature of the elementary computations that can be performed locally by the components, which are abstracted as

operations on data elements. Such elementary computations also determine the granularity of the services that components can provide and, hence, the granularity of the interconnections that can be established at a given layer of abstraction. Nevertheless, data refinement is more concerned with the computational aspects of systems than with the coordination mechanisms that are responsible for interactions among system components. Because the support that Category Theory can provide to the specification of abstract data types is already well established and available in the literature, even through books [8,28,29,75,93], we shall not address this aspect of CommUnity in depth but, rather, concentrate on the broader architectural aspects, giving more emphasis to refinement of designs for a fixed choice of data types and omitting any discussion on data refinement.

Given this, we shall assume a fixed collection of data types. In order to remain independent of any specific language for the definition of these data types, we take them in the form of a first-order algebraic specification. That is to say, we assume a data signature $\langle S, \square \rangle$, where S is a set (of sorts) and \square is a $S^c \times S$ -indexed family of sets (of operations), to be given together with a collection \square of first-order sentences specifying the functionality of the operations.

A CommUnity design for a component over such a data type specification is of the form

```

design P is
out   out(V)
in    in(V)
prv   prv(V)
do     $\begin{array}{l} g[D(g)]: L(g), U(g) \square R(g) \\ g \square_{\text{sh}}(\square) \\ g \square_{\text{prv}}(\square) \end{array} \text{ prv } g[D(g)]: L(g), U(g) \square R(g)$ 

```

where

- V is a set (of *communication channels*). A communication channel (or, simply, channel) can be declared as *input*, *output* or *private*. Each channel v is typed with a sort $\text{sort}(v) \square S$ that reflects the nature of the data that is exchanged through it.
 - \square Input channels are used for reading data from the environment of the component. The component has no control on the values that are made available in such channels. Moreover, reading a value from an input channel does not “consume” it: the value remains available until the environment decides to replace it.
 - \square Output and private channels are controlled locally by the component, i.e. the values that, at any given moment, are available on these channels cannot be modified by the environment. Output channels allow the environment to read data produced by the component. Private channels support internal activity that does not involve the environment in any way. We use $\text{loc}(V)$ to denote the union $\text{prv}(V) \square \text{out}(V)$, i.e. the set of local channels.
 - \square In some of the earlier papers on CommUnity, we have named the elements of V *variables* or *attributes*. The change from *variables* to *channels* aims at reinforcing the idea that the elements of V correspond to means that components have to communicate rather than “store” data. This is consistent with the “black-box” view of components that we intend to model, which should hide the representation of the state of components and provide only means for it to be observed.
 - \square Channels cater for asynchronous communication between components in the sense that reading and writing into a channel are independent op-

erations: a value that is written on a channel will remain there, regardless of how many times it is read, until it is overwritten.

- \square is a set (of *action names*). The named actions can be either *private* or *shared* (for simplicity, we only declare which actions are private).
 - \square Private actions represent internal computations in the sense that their execution is uniquely under the control of the component.
 - \square Shared actions are used for synchronous interactions between the component and the environment, meaning that their execution is also under the control of the environment.
 - \square The significance of naming actions will become obvious below; the idea is to provide points of *rendez-vous* at which components can synchronise, for instance as a means of ensuring that the right values are being exchanged through the channels.
- For each action name g , the following attributes are defined:
 - \square $D(g)$ is a subset of $loc(V)$ consisting of the local channels into which executions of the action can write. This is what is sometimes called the *write frame* of g . For simplicity, we will omit the explicit reference to the write frame when $R(g)$ is a conditional multiple assignment (see below), in which case $D(g)$ can be inferred from the assignments. Given a local channel v , we will also denote by $D(v)$ the set of actions g such that $v \square D(g)$, i.e. the actions that write into v .
 - \square $L(g)$ and $U(g)$ are two conditions such that $U(g) \supseteq L(g)$. These conditions establish an interval in which the enabling condition of any guarded command that implements g must lie. The condition $L(g)$ is a lower bound for enabledness in the sense that it is implied by the enabling condition. Therefore, its negation establishes a *blocking* condition. On the other hand, $U(g)$ is an upper bound in the sense that it implies the enabling condition, therefore establishing a *progress* condition. Hence, the enabling condition is fully determined only if $L(g)$ and $U(g)$ are equivalent, in which case we write only one condition.
 - \square $R(g)$ is a condition on V and $D(g)'$ where by $D(g)'$ we denote the set of primed local channels from the write frame of g . As usual, these primed channels account for references to the values that the channels display after the execution of the action. These conditions are usually a conjunction of implications of the form $pre \supset pos$ where pre does not involve primed channels. They correspond to pre/post-condition specifications in the sense of Hoare. When $R(g)$ is such that the primed version of each local channel in the write frame of g is fully determined, we obtain a conditional multiple assignment, in which case we use the notation that is normally found in programming languages. When the write frame $D(g)$ is empty, $R(g)$ is tautological, which we denote by *skip*.

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

When, for every $g \in \square$, $L(g)$ and $U(g)$ coincide, and the relation $R(g)$ defines a conditional multiple assignment, then the design is called a *program* and the traditional notation for guarded commands is used. Notice that a program with a non-empty set of input channels is *open* in the sense that its execution is only meaningful in the context of a configuration in which these inputs have been connected with local outputs of other components. The notion of configuration, and the execution of an

open program in a given configuration, will be discussed further below. The behaviour of a closed program is as follows. At each execution step, one of the actions whose enabling condition holds of the current state is selected, and its assignments are executed atomically. Furthermore, private actions that are infinitely often enabled are guaranteed to be selected infinitely often. See [76] for a model-theoretic semantics of CommUnity.

Designs can be parameterised by data elements (sorts and operations) indicated after the name of the component (see an example below). These parameters are instantiated at configuration time, i.e. when a specific component needs to be included in the configuration of the system being built, or as part of the reconfiguration of an existing system.

As an example, consider the following parameterised design:

```

design buffer [t:sort, bound:nat] is
in      i:t
out     o:t
prv     rd: bool, b: list(t)
do      put: |b|<bound □ b:=b.i
        [] prv next: |b|>0 □ rd □ o:=head(b) || b:=tail(b) || rd:=true
        [] get: rd □ rd:=false

```

The parameters of this design consist of the sort t of data elements that the buffer can handle and the capacity $bound$ of the buffer. The buffer itself is defined over a list with elements of t . As already discussed, we are assuming that the data type *list* is available through an algebraic specification that includes the traditional operations such as $|_$ returning the current size of the list, $head(_)$ returning the first element of the list, $tail(_)$ returning the list after the first element, and $_.i$ for appending an element to the end of the list.

This design is actually a (parameterised) program and the traditional notation of guarded commands was used accordingly. Notice in particular that the reference to the write frame of the actions was omitted: it can be inferred from the multiple assignments that they perform. As already mentioned, because we are dealing with multiple assignments, the traditional notation involving the symbol $:=$ was used instead of the logical language over channels and their primed versions. In the case above, this corresponds to:

```

R(put):   b'=b.i
R(next):  o'=head(b) □ b'=tail(b) □ rd'
R(get):   ¬rd'

```

This program models a buffer with a limited capacity and a FIFO discipline. It can store, through the action *put*, messages of sort t received from the environment through the input channel i , as long as there is space for them. The buffer can also discard stored messages, making them available to the environment through the output channel o and the action *next*. Naturally, this activity is possible only when there are messages in store and the current message in o has already been read by the environment (which is modelled by the action *get* and the private channel rd).

In order to illustrate the ability of CommUnity to support higher-level component design, we present below the design of a typical sender of messages.

```

design sender[t:sort] is
out     o:t
prv     rd: bool
do      prod[o,rd]:[]rd,false[] rd'
        [] send[rd]:[]rd,false[] ¬rd'

```

In this design, we are primarily concerned with the interaction between the sender and its environment, ignoring details of internal computations such as the production of messages. This is why the output channel o is included in the write frame of $prod$ but $R(prod)$ does not place any constraint on how it is updated. Notice that the component *sender* cannot produce another message before the previous one has been processed: after producing a message, the sender expects an acknowledgement (modelled through the execution of *send*) to produce a new message.

In order to leave unspecified when and how many messages the *sender* will send and in which situations it will produce a new message, the progress conditions of $prod$ and $send$ are false (recall that the progress condition defines the upper bound for enabledness). Furthermore, the discipline of production is also left completely unspecified: the action $prod$ includes the output channel o in its write frame but the design does not commit to any specific way of updating the values in this channel.

From a mathematical point of view, (instantiated) CommUnity designs are structures defined as follows.

8.1.1 DEFINITION – signatures and designs

A *signature* in CommUnity is a tuple $\langle V, \square, tv, ta, D \rangle$ where

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

A *design* in CommUnity is a pair $\langle \square, \square \rangle$ where $\square = \langle V, \square, tv, ta, D \rangle$ is a signature and \square , the body of the design, is a tuple $\langle R, L, U \rangle$ where:

- R assigns to every action $g \square \square$, a proposition over $V \square D(g)'$,
- L and U assign a proposition over V to every action $g \square \square$.

The reader who is familiar with parallel program design languages or earlier versions of CommUnity will have probably noticed the absence of initialisation conditions. The reason they were not included in CommUnity designs is because they are part of the configuration language of CommUnity, not the parallel program design language. That is to say, we take initialisation conditions as part of the mechanisms that relate to the building and management of configurations out of designs, not of the construction of designs themselves.

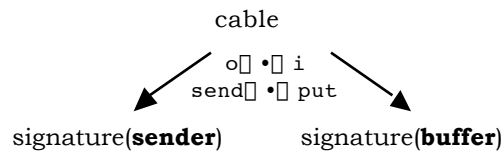
8.2 Interconnecting designs

So far, we have presented the primitives for the design of individual components, which are another variation on guarded commands, albeit with some “twists” of originality such as the use of an interval as a specification for the enabling conditions of commands. The main distinguishing features of CommUnity are those that concern design “in the large”, i.e. the ability to design large systems from simpler components.

The model of interaction between components in CommUnity is based on action synchronisation and the interconnection of input channels of a component with output channels of other components. These are standard means of interconnecting software components. What distinguishes CommUnity from other parallel program design languages is the fact that such interactions between components have to be made explicit by providing the corresponding name bindings. Indeed, parallel program design languages normally leave such interactions implicit by relying on the use of the same names in different components. In CommUnity, names are local to designs. This means that the use of the same name in different designs is treated as being purely accidental, and, hence, expresses no relationship between the components.

In CommUnity, name bindings are established as relationships between the signatures of the corresponding components, matching channels and actions of these components. These bindings are made explicit in configurations. A configuration determines a diagram containing nodes labelled with the signatures of the components that are part of the configuration. Name bindings are represented as additional nodes labelled with sets representing the actual interactions, and edges labelled with the projections that map each interaction to the corresponding component signatures.

For instance, a configuration in which the messages from a *sender* component are sent through a bounded buffer defines the following diagram:



The node labelled *cable* is the representation of the set of bindings. (The choice of name intends to suggest analogies with the use of physical cables as a means of interconnecting mechanical or electrical components). Because, as we have seen, channels and action names are typed and classified in different categories, not every pair of names is a valid binding. To express the rules that determine valid bindings, it is convenient to structure *cable* as a signature itself (just like the wires inside electrical cables are coloured). Hence, in the case above, *cable* consists of an input channel to model the medium through which data is to be transmitted between the sender and the buffer, and a shared action for the two components to synchronise in order to transmit the data. Because, as we have already mentioned, names in CommUnity are local, the identities of the shared input channel and the shared action in *cable* are not relevant: they are just placeholders for the projections to define the relevant bindings. Hence, we normally do not bother to give them explicit names, and represent them through the symbol •.

The bindings themselves are established through the labels of the edges of the diagram. In the case above, the input channel of *cable* is mapped to the output channel *o* of *sender* and to the input channel *i* of *buffer*. This establishes an i/o-interconnection between *sender* and *buffer*. On the other hand, the actions *send* of *sender* and *put* of *buffer* are mapped to the shared action of *cable*. This defines that *sender* and *buffer* must synchronise each time either of them wants to perform the corresponding action. The fact that the mappings on action names and on channels go in opposite directions will be discussed below.

The arrows that we are using to define interconnections between components are also mathematical objects: they are examples of signature morphisms.

8.2.1 DEFINITION – signature morphisms

A morphism $\square: \square_1 \rightarrow \square_2$ between signatures $\square_1 = \langle V_1, \square_1, tv_1, ta_1, D_1 \rangle$ and $\square_2 = \langle V_2, \square_2, tv_2, ta_2, D_2 \rangle$ is a pair $\langle \square_{ch}, \square_{ac} \rangle$ where

- $\square_{ch}: V_1 \rightarrow V_2$ is a total function satisfying:
 1. $sort_2(\square_{ch}(v)) = sort_1(v)$ for every $v \in V_1$
 2. $\square_{ch}(o) \in out(V_2)$ for every $o \in out(V_1)$
 3. $\square_{ch}(i) \in out(V_2) \cap in(V_2)$ for every $i \in in(V_1)$
 4. $\square_{ch}(p) \in prv(V_2)$ for every $p \in prv(V_1)$
- $\square_{ac}: \square_2 \rightarrow \square_1$ is a partial mapping satisfying for every $g \in \square_2$ s.t. $\square_{ac}(g)$ is defined:
 5. if $g \in sh(\square_2)$ then $\square_{ac}(g) \in sh(\square_1)$
 6. if $g \in prv(\square_2)$ then $\square_{ac}(g) \in prv(\square_1)$
 7. $\square_{ch}(D_1(\square_{ac}(g))) \in D_2(g)$
 8. \square_{ac} is total on $D_2(\square_{ch}(v))$ and $\square_{ac}(D_2(\square_{ch}(v))) \in D_1(v)$ for every $v \in loc(V_1)$

Signature morphisms represent more than the projections that arise from name bindings as illustrated above. A morphism \square from \square_1 to \square_2 is intended to support the identification of a way in which a component with signature \square_1 is embedded in a larger system with signature \square_2 . This justifies the various constructions and constraints in the definition.

The function \square_{ch} identifies for each channel of the component the corresponding channel of the system. The partial mapping \square_{ac} identifies the action of the component that is involved in each action of the system, if ever. The fact that the two mappings go in opposite directions is justified as follows. Actions of the system constitute synchronisation sets of actions of the components. Because not every component is necessarily involved in every action of the system, the action mapping is partial. On the other hand, because each action of the component may participate in more than one synchronisation set, but each synchronisation set cannot induce internal synchronisations within the components, the relationship between the actions of the system and the actions of every component is functional from the former to the latter. Hence, actions will be dealt with in the category **PAR** of partial functions. As seen in 7.1.12, this category is equivalent to the category that we used for modelling alphabets of processes, meaning that the intuitions that we developed on the way universal constructions capture composition can be used for CommUnity as well.

Input/output communication within the system is not modelled in the same way as action synchronisation. Synchronisation sets reflect parallel composition whereas with i/o-interconnections we wish to merge communication channels of the components. This means that, in the system, channels should be identified rather than paired. This is why mappings on channels and mappings on actions go in opposite directions. We will see that, as a result, the mathematical semantics of configuration diagrams induces fibred products of actions (synchronisation sets) and amalgamated sums of channels (equivalence classes of connected channels).

The constraints are concerned with typing. Sorts associated with channels have to be preserved but, in terms of their classification, input channels of a component may become output channels of the system in the sense that, as a default, they should

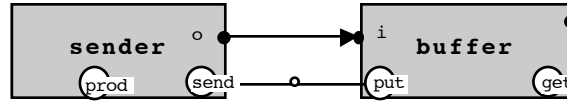
remain open for communication with other components. In most languages for parallel design, the default is to hide the communication, what in CommUnity would correspond to classify the resulting channel as being private. In our opinion, closing/hiding the channel should not be a default but a design decision that should be performed explicitly. Hence, in CommUnity, mechanisms for internalising communication can be applied but they are not the default in a configuration. The last two conditions on write frames (6 and 7) imply that actions of the system in which a component is not involved cannot have local channels of the component in its write frame. That is, change within a component is completely encapsulated in the structure of actions defined for the component.

Given the ingredients out of which signatures are assembled, the proof of the following result is pure routine and left as an exercise:

8.2.2 PROPOSITION – category of signatures

Signatures in CommUnity together with their morphisms constitute a category that we shall denote by **c-SIGN**.

The notation can be simplified and made friendlier by adopting features that are typical of languages for configurable distributed systems like [89]. For instance, the interconnection defined before can be described as follows.



The notation should be self-explaining, a good excuse for not providing a formal definition of the graphical language that we shall be using. Components are represented through boxes, their channels through bullets and their actions through circles. Interconnections, i.e. name bindings, are still represented explicitly but, instead of being depicted as a component, the cable is now represented, perhaps more intuitively, in terms of arcs that connect channels and actions directly. The direction of the arcs is from output to input channels. Configurations in this notation are easily translated into categorical diagrams by transforming the interconnections into channels and morphisms, something which, again, we shall abstain from formalising here.

So far, we have explained how interconnections between components can be established at the level of the signatures of their designs. It remains to explain how the corresponding designs are interconnected, i.e. what is the semantics of the configuration diagram once designs are taken into account. For that purpose, we need to extend the notion of morphism from signatures to designs.

8.2.3 DEFINITION/PROPOSITION – design morphisms

A morphism $\square: P_1 \square P_2$ of designs $P_1 = \langle \square_1, \square_1 \rangle$ and $P_2 = \langle \square_2, \square_2 \rangle$, consists of a signature morphism $\square: \square_1 \square \square_2$ such that, for every $g \square \square_2$ s.t. $\square_{ac}(g)$ is defined:

1. $\square \vdash \square(R_2(g) \square \square(R_1(\square_{ac}(g))))$
2. $\square \vdash (L_2(g) \square \square(L_1(\square_{ac}(g))))$
3. $\square \vdash (U_2(g) \square \square(U_1(\square_{ac}(g))))$

where \sqsubseteq is the axiomatisation of the data type specification, \vdash denotes validity in the first-order sense, and \sqsubseteq is the extension of \sqsubseteq to the language of expressions and conditions. Designs and their morphisms constitute a category **c-DSGN**. This category is concrete over **c-SIGN** through the obvious forgetful functor.

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

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

The notion of morphism that we have just defined captures what in the literature on parallel program design is called "superposition" or "superimposition" [19,47,71]. See [44] for the categorical formalisation of different notions of superposition and their algebraic properties.

The semantics of configurations is given by a categorical construction: the colimit of the underlying diagrams. As we have already explained in chapter 1, taking the colimit of a diagram collapses the configuration into an object by internalising all the interconnections, thus delivering a design for the system as a whole. Furthermore, the colimit provides a morphism \sqsubseteq_i from each component design P_i in the configuration into the new design (that of the system) – the edge of the co-cone at P_i (see 4.4.1). Each such morphism is essential for identifying the corresponding component within the system because the construction of the new design typically requires that the features of the components be renamed in order to account for the interconnections.

Again, given the nature of the "ingredients", it is not difficult to understand how colimits of designs work: because channels are handled through total functions, colimits amalgamate channels (4.3.2); and because actions are handled as partial functions in the opposite direction, i.e. in the dual category, colimits operate on actions as limits and compute fibred products (4.3.8). For instance, in the case of actions, the colimit represents every synchronisation set $\{g_1, \dots, g_n\}$ of actions of the components, as defined through the interconnections, by a single action $g_1 \parallel \dots \parallel g_n$ whose occurrence captures the joint execution of the actions in the set (recall 4.3.8 and 4.4.11). Because limits perform conjunctions of logical conditions (4.2.5), the transformations performed by a joint action are specified by the conjunction of the specifications of the local effects of each of the synchronised actions, i.e. $R(g_1 \parallel \dots \parallel g_n) = \sqsubseteq_1(R(g_1)) \sqcap \dots \sqcap \sqsubseteq_n(R(g_n))$ where the \sqsubseteq_i are the morphisms that connect the components to the system (the edges of the co-cone). The bounds on the guards of joint actions are also obtained through the conjunctions of the bounds specified by the components, i.e.

$$L(g_1 \parallel \dots \parallel g_n) = \sqcap_1(L(g_1)) \sqcap \dots \sqcap \sqcap_n(L(g_n)) \text{ and } U(g_1 \parallel \dots \parallel g_n) = \sqcap_1(U(g_1)) \sqcap \dots \sqcap \sqcap_n(U(g_n)).$$

This way of computing colimits derives from the strong algebraic properties of the category of designs. More precisely:

8.2.4 PROPOSITION

The forgetful functor **c-sign** that maps CommUnity designs to the corresponding signatures defines **c-DSGN** as a category coordinated over **c-SIGN** (7.5.4). We shall call a *cable* the discrete lift of a signature: given a signature \sqcap the corresponding cable **dsgn**(\sqcap) has \sqcap for signature and, for every action g , $R(g)$, $L(g)$ and $U(g)$ are all *true*.

Summarising, colimits in CommUnity capture a generalised notion of parallel composition in which the designer makes explicit what interconnections are used between components. Because the category of designs is coordinated over signatures, all interconnections can be performed through cables, i.e. they do not involve the computational part of components, only their "interfaces"– i/o communication through channels and rendez-vous through action synchronisation. We can see this operation as a generalisation of the notion of superimposition as defined in [47].

The colimit of the configuration, when it returns a closed program, can also be used for providing an operational semantics for the system thus configured: as explained in section 8.1, at each execution step, any action whose guard is true can be executed, with the guarantee that private actions that are infinitely often enabled are selected infinitely often. Because actions of the system are synchronisation sets of actions of the components, the evaluation of the guard of the chosen action can be performed in a distributed way by evaluating the guards of the component actions in the synchronisation set. According to the semantics that we have just given, the joint action will be executed iff all the local guards evaluate to *true*. The execution of the multiple assignment associated with the joint action can also be performed in a distributed way by executing each of the local assignments. What is important is that the atomicity of the execution is guaranteed, i.e. the next system step should only start when all local executions have completed, and the i/o-communications should be implemented so that every local input channel is instantiated with the correct value – that which holds of the local state before any execution starts (synchronicity).

Hence, the colimit of the configuration diagram should be seen as an abstraction of the actual distributed execution that is obtained by coordinating the local executions according to the interconnections, rather than the program that is going to be executed as a monolithic unit. The fact that the computational part, i.e. the one that is concerned with the execution of the actions on the state, can be separated from the coordination aspects is, therefore, an essential property for guaranteeing that the operational semantics is compositional on the structure of the system as given through its configuration diagram.

Not every diagram of designs reflects a meaningful configuration. For instance, it does not make sense to interconnect components by connecting two output channels. Indeed, we cannot guarantee that every diagram admits a colimit, meaning that there are diagrams that have no "semantics" as configurations.

8.2.5 DEFINITION/PROPOSITION – well-formed configurations

Let **chan** be the forgetful functor from **c-DSGN** to **SET** that maps designs to their underlying sets of channels. A *configuration* is a finite diagram $\mathbf{dia} : \mathbf{I} \rightarrow \mathbf{c-DSGN}$ together with a subset J of $|\mathbf{I}|$ (the nodes that represent the components being interconnected) such that:

1. For every $f : i \rightarrow j$ in \mathbf{I} , either $i=j$ and $f=id_i$; or $j \notin J$ and $i \notin J$ and $\mathbf{dia}(i)$ is a cable;
2. For every $i \notin |\mathbf{I}| \setminus J$ s.t. $\mathbf{dia}(i)$ is a cable, there exist distinct nodes $j, k \in |\mathbf{I}|$ with morphisms $f : i \rightarrow j$ and $g : i \rightarrow k$;
3. If $\{\square_i : \mathbf{chan}(\mathbf{dia}(i)) \rightarrow V : i \in |\mathbf{I}|\}$ is a colimit of $\mathbf{dia}; \mathbf{chan}$ then, for every $v \in V$, there exists at most one $i \in |\mathbf{I}|$ s.t. $\square_i^1(v) \in \text{out}(V_{\mathbf{dia}(i)}) \neq \emptyset$ and, for such i , $\square_i^1(v) \in \text{out}(V_{\mathbf{dia}(i)})$ is a singleton.

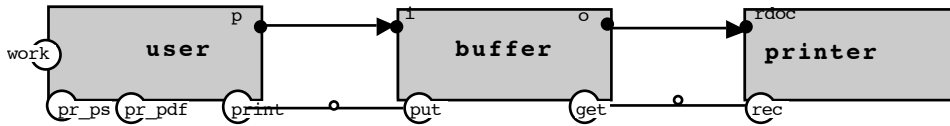
Configurations admit colimits.

We further say that a configuration is *well-formed* if for every $i \notin |\mathbf{I}| \setminus J$ s.t. $\mathbf{dia}(i)$ is a cable, $\mathbf{dia}(i)$ has neither private actions nor private channels.

Condition 1 states that the elementary interconnections are established through cables. Condition 2 ensures that a configuration diagram does not include cables that are not used. Finally, condition 3 prevents the identification of output channels. The explicit reference to the subset J of components is necessary because the distinction between nodes that are being used as channels and as components is a pragmatic, not formal one: it is possible that, in a given configuration, a node is intended to represent a component but, because it is still totally underspecified, it is the discrete lift of a signature, i.e. what we have called a channel.

Well-formed configurations are such that private actions and channels are not involved in the interconnections, i.e. they support the intuitive semantics we gave in section 8.1 according to which private channels cannot be read by the environment and that the execution of shared actions is uniquely under the control of the component.

An example of a more complex configuration is given below. It models the interconnection between a user and a printer via a buffer.



The user produces files that it stores in the private channel w . It can then convert them either to postscript or pdf formats, after which it makes them available for printing in the output channel p .

```

design user is
out   p:ps+pdf
prv   s,t: bool, w: Lowtex
do     work[w,s,t]:  $\neg t, \text{false} \sqcap t'$ 
[]      pr_ps: $\neg s \sqcap t, \text{false} \sqcap p:=\text{ps}(w) \parallel s:=\text{true}$ 
[]      pr_pdf: $\neg s \sqcap t, \text{false} \sqcap p:=\text{pdf}(w) \parallel s:=\text{true}$ 
[]      print: $\sqcap s \sqcap s:=\text{false} \parallel t:=\text{false}$ 

```

The printer copies the files it downloads from the input channel *rdoc* into the private channel *pdoc*, after which it prints them.

```

design printer is
in    rdoc:ps+pdf
prv   busy: bool, pdoc: ps+pdf
do    rec:[ ]busy [ ] pdoc:=rdoc||busy:=true
[ ] prv end_print:[ ]busy [ ] busy:=false

```

The configuration connects the user to the printer via a buffer as expected. The user "prints" by placing the file in the buffer: this is achieved through the synchronisation pair $\{print, put\}$ and the i/o-interconnection $\{p, i\}$. The printer downloads from the buffer the files that it prints: this is achieved through the synchronisation pair $\{get, rec\}$ and the i/o-interconnection $\{o, rdoc\}$.

The design of the system that results from the colimit of the configuration diagram contains two channels that account for the two i/o-interconnections $\{p, i\}$ and $\{o, rdoc\}$, together with the private channels of the components. At the level of its actions, it generates the following shared actions (synchronisation sets):

$\{print, put\}$, $\{get, rec\}$ – these are required by the interconnections

$\{work\}$, $\{pr_ps\}$, $\{pr_pdf\}$, $\{work, get, rec\}$, $\{pr_ps, get, rec\}$, $\{pr_pdf, get, rec\}$ – these reflect the concurrent executions that respect the interconnections.

No other shared actions are possible because of the synchronisation requirements imposed on the components.

8.3 Refining designs

The notion of morphism defined in the previous section does not capture a refinement relation in the sense that it does not ensure that any implementation of the target provides an implementation for the source. For instance, it is easy to see that morphisms do not preserve the interval assigned to the guard of each action. Given that the aim of the defined morphisms was to capture the relationship that exists between systems and their components, this is hardly surprising. The same holds in languages such as CSP [68]: in the failure or ready semantics, parallel composition does not induce refinement – $P \parallel Q$ is not necessarily a refinement of P – and refinement cannot always be expressed as the result of a parallel composition – P may refine Q and, yet, there may not exist a Q' such that P is $Q \parallel Q'$.

Because refinement is an important dimension in structuring software development, it is only natural that we investigate ways of supporting it in a categorical setting. This would be especially useful for analysing the way refinement and composition can work together. A notion of morphism can indeed be defined that captures a refinement relation for CommUnity designs.

8.3.1 DEFINITION/PROPOSITION – refinement morphisms

A refinement morphism $\square: P_1 \square P_2$ of designs $P_1 = \langle \square_1, \square_1 \rangle$ and $P_2 = \langle \square_2, \square_2 \rangle$ is a pair $\langle \square_{ch}, \square_{ac} \rangle$ satisfying:

- $\sqsubseteq_{ch}: V_1 \rightarrow \text{Term}(V_2)$ is a total function mapping the channels of P_1 to the class of terms built from the channels of P_2 and the data type operations. This mapping is required to satisfy, for every $v \in V_1$, $o \in \text{out}(V_1)$, $i \in \text{in}(V_1)$, $p \in \text{prv}(V_1)$:
 1. $\text{sort}_2(\sqsubseteq_{ch}(v)) = \text{sort}_1(v)$
 2. $\sqsubseteq_{ch}(o) \in \text{out}(V_2)$
 3. $\sqsubseteq_{ch}(i) \in \text{in}(V_2)$
 4. $\sqsubseteq_{ch}(p) \in \text{prv}(V_2)$
 5. $\sqsubseteq_{ch}(\text{out}(V_1) \cap \text{in}(V_1))$ is injective
- $\sqsubseteq_{ac}: \Sigma_2 \rightarrow \Sigma_1$ is a partial mapping satisfying for every $g \in \Sigma_2$ s.t. $\sqsubseteq_{ac}(g)$ is defined:
 6. if $g \in \text{sh}(\Sigma_2)$ then $\sqsubseteq_{ac}(g) \in \text{sh}(\Sigma_1)$
 7. if $g \in \text{prv}(\Sigma_2)$ then $\sqsubseteq_{ac}(g) \in \text{prv}(\Sigma_1)$
 8. if $g \in \text{sh}(\Sigma_1)$ then $\sqsubseteq_{ac}^{-1}(g) \neq \emptyset$
 9. $\sqsubseteq_{ch}(D_1(\sqsubseteq_{ac}(g))) \in D_2(g)$
 10. \sqsubseteq_{ac} is total on $D_2(\sqsubseteq_{ch}(v))$ and $\sqsubseteq_{ac}(D_2(\sqsubseteq_{ch}(v))) \in D_1(v)$ for every $v \in \text{loc}(V_1)$
- for every $g \in \Sigma_2$ s.t. $\sqsubseteq_{ac}(g)$ is defined:
 11. $F \vdash (R_2(g) \dots \sqsubseteq (R_1(\sqsubseteq_{ac}(g))))$
 12. $F \vdash (L_2(g) \dots \sqsubseteq (L_1(\sqsubseteq_{ac}(g))))$
- for every $g_1 \in \Sigma_1$,
 13. $\sqsubseteq \vdash (\sqsubseteq(U_1(g_1))) \sqsubseteq \sqsubseteq_{\sqsubseteq_{ac}(g_2)=g_1} U_2(g_2)$

We denote by \sqsubseteq the axiomatisation of the data type specification, and by \vdash the validity relation of first-order logic; \sqsubseteq is the extension of \sqsubseteq to the language of expressions and conditions; and \underline{D} is the extension of D to the language of expressions.

Designs and their refinement morphisms constitute a category **r-DSGN**. This category is concrete over **c-SIGN** through the functor **r-sign** that, like **c-sign**, projects designs to their signatures.

A refinement morphism identifies a way in which a design P_1 (its source) is refined by a more concrete design P_2 (its target). The function \sqsubseteq_{ch} identifies, for each input (resp. output) channel of P_1 , the corresponding input (resp. output) channel of P_2 . Notice that, contrarily to what happens with the component-of relationship as captured through design morphisms (8.2.3), refinement does not change the border between the system and its environment and, hence, input channels can no longer be mapped to output channels (3). This is also why the mapping is required to be injective on input and output channels (5): identifying channels is a configuration operation to be achieved through interconnections, not a refinement step.

As for design morphisms, refinement morphisms are required to preserve the sorts of channels (1). As discussed at the beginning of this chapter, data refinement is a dimension that, for simplicity, we are deliberately ignoring in the book.

The mapping \sqsubseteq_{ac} identifies for each action g of P_1 , the set $\sqsubseteq_{ac}^{-1}(g)$ of actions of P_2 that implements g . This set is a "menu" of refinements that is made available for implementing action g ; different choices can be made at different states to take advantage of the structures available at the more concrete design level. This menu can be empty for private actions, i.e. one may choose not to implement the private actions of the more abstract design: because private actions do not intervene in interconnections, what is important is that the overall behaviour of the component as made observable through shared actions and output channels be implemented. This is also

why every shared action has to be implemented (8); again, such actions model interaction between the component and its environment, and refinement should not interfere with the border between them.

The actions for which \square_{ac} is left undefined (the new actions) and the channels which are not involved in $\square_{ch}(V_1)$ (the new channels) introduce more detail in the description of the component. As for the "old actions", the interval defined by their blocking and progress conditions (in which the enabling condition of any implementation must lie) must be preserved or reduced (12 and 13). This is intuitive because refinement, pointing in the direction of implementations, should reduce underspecification. Hence, the lower bound cannot be weakened (12) and, contrarily to design morphisms, the upper bound cannot be strengthened (13). This is also the reason why the effects of the actions of the more abstract design are required to be preserved or made more deterministic (11).

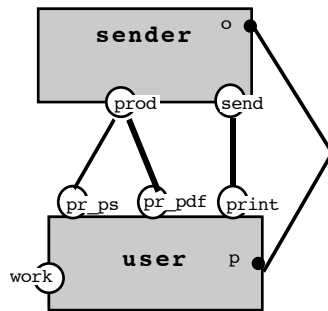
Notice that the forgetful functors **r-sign** and **c-sign** are essentially the same; they are only formally different because their sources are not the same category. Indeed, the only difference between design and refinement morphisms at the level of signatures is on the additional properties that refinement morphisms need to satisfy: 3, 5 and 8.

As an example, it is easy to see that *sender* is refined by *user* via the refinement morphism $\square: \text{sender} \sqsubseteq \text{user}$ defined by

$$\begin{aligned} \square_{ch}(o) &= p, \square_{ch}(rd) = (s) \\ \square_{ac}(pr_ps) &= \square_{ac}(pr_pdf) = prod, \square_{ac}(print) = send \end{aligned}$$

In *user*, the production of messages (to be sent) is modelled by any of the actions *pr_ps* and *pr_pdf*; the messages are made available in the output channel *p*. Notice that the production of messages, that was left unspecified in *sender*, is completely defined in *user*: it corresponds to the conversion of the files stored in *w* to ps or pdf formats.

In the simplified graphical notation that we have been using, refinement is represented through bold (thick) lines:



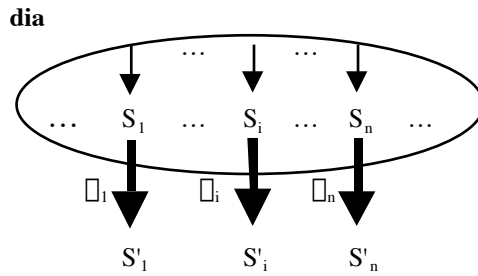
Summarising what we have built so far, we have two categories **c-DSGN** (8.2.3) and **r-DSGN** (8.3.1), both over the same notion of object – CommUnity design – but with different notions of morphism, i.e. capturing different aspects of their social lives: one tells us about their ability to relate with other designs at the same level of abstraction, and the other about the way they can be made more “concrete” by reducing the amount of underspecification. Furthermore, **c-DSGN** is coordinated over **c-SIGN** through the functor **c-sign** (8.2.4). As an exercise, the reader is invited to extend the study of the structural properties of **r-DSGN**. We are now interested in the way interconnection relates to refinement.

The first important property relates to the requirement that refinement should not be based on the specificities of each particular design as far its ability to be interconnected to other designs is concerned. In other words, refinement morphisms should be such that designs that are isomorphic in **c-DSGN** refine, and are refined exactly by, the same designs. This is, indeed, the case:

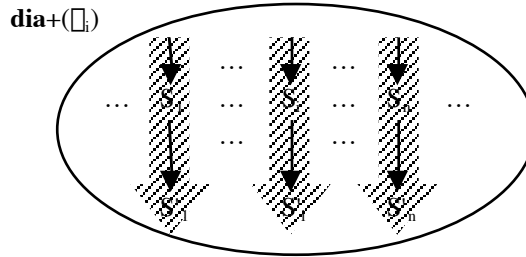
8.3.2 PROPOSITION

Every isomorphism in **c-DSGN** defines an isomorphism in **r-DSGN**.

Another crucial property is in the ability to refine a complex system from refinements of its individual components. Consider a well-formed configuration **dia** of a system with components S_1, \dots, S_n and refinement morphisms $\square_i: S_i \rightarrow S'_i$, $i \in 1..n$,



By composing the morphisms \square_i with those in **dia** that originate in cables (designs of the form **dsgn**(\square) where \square is a signature) and have the S_i as targets, we obtain a new diagram in **c-DSGN** that we denote **dia**+(\square_i)

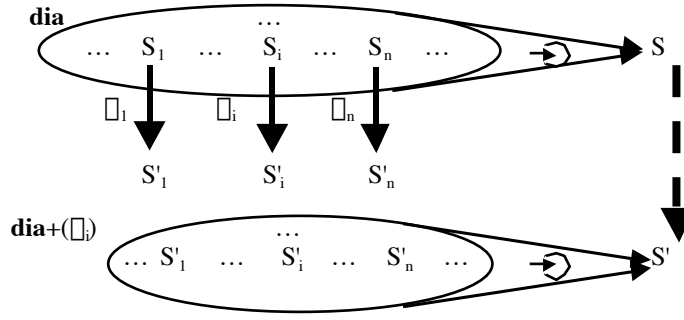


This composition is possible because, **c-DSGN** being coordinated over **c-SIGN**, any morphism $\square: \mathbf{dsgn}(\square) \rightarrow S_i$ "is" the signature morphism **c-sign**(\square): $\square \rightarrow \mathbf{c-sign}(S_i)$; given now a refinement morphism $\square_i: S_i \rightarrow S'_i$, we can compose **c-sign**(\square) with **r-sign**(\square_i) to obtain a signature morphism: $\square \rightarrow \mathbf{c-sign}(S'_i)$ that can be lifted back to **c-DSGN** as a morphism **dsgn**(\square) $\rightarrow S'_i$.

The two diagrams satisfy the following important property:

8.3.3 PROPOSITION

In the circumstances laid out above, if $p: \mathbf{dia} \rightarrow S$ and $p': \mathbf{dia}+(\square_i) \rightarrow S'$ are colimits, there is a unique refinement morphism $S \rightarrow S'$ that is also a morphism $p \rightarrow p'$.

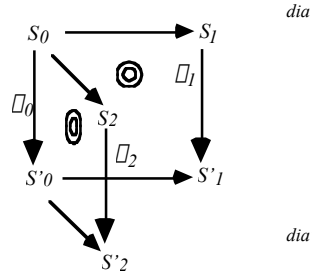


This property is another form of *compositionality*: it states that refinement of the whole can be obtained from refinements of the parts. Compositionality, as already discussed in 6.2.4, is a key issue in the design of complex systems because it makes it possible to reason about a system using the descriptions of their components at any level of abstraction, without having to know how these descriptions are refined in the lower levels (which includes their implementation).

This form of compositionality can be formulated more precisely in CommUnity by extending the notion of refinement to configurations much in the same way as we extended the notion of realisation to configurations of specifications in 6.2.4.

8.3.4 DEFINITION – refinement of configurations

Given two configurations $\mathbf{dia}:I \sqsubseteq \mathbf{c-DSGN}$ and $\mathbf{dia}':I \sqsubseteq \mathbf{c-DSGN}$, a *refinement of \mathbf{dia} over \mathbf{dia}'* is an $|I|$ -indexed family $(\Box_i: \mathbf{dia}(i) \sqsubseteq \mathbf{dia}'(i))_{i \in |I|}$ of morphisms in $\mathbf{r-DSGN}$ s.t., for every $f: i \sqsubseteq j$ in I , $\mathbf{c-sign}(\mathbf{dia}(f)); \mathbf{r-sign}(\Box_i) = \mathbf{r-sign}(\Box_j); \mathbf{c-sign}(\mathbf{dia}'(f))$.



In order to ensure compositionality, i.e., that the colimit S of \mathbf{dia} is refined by the colimit S' of \mathbf{dia}' , it is necessary to further require that:

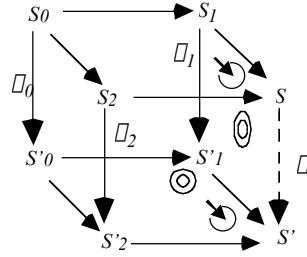
- The diagram \mathbf{dia}' cannot establish the instantiation of any input channel that was left “unplugged” in \mathbf{dia} . That is to say, the input channels of the composition are preserved by refinement.
- The diagram \mathbf{dia}' cannot establish the synchronisation of actions that were defined as being independent in \mathbf{dia} .

8.3.5 PROPOSITION – compositionality

Given two well-formed configurations $\mathbf{dia}:I \sqsubseteq \mathbf{c-DSGN}$ and $\mathbf{dia}':I \sqsubseteq \mathbf{c-DSGN}$ over a set J of components, and a refinement $(\Box_i: \mathbf{dia}(i) \sqsubseteq \mathbf{dia}'(i))_{i \in |I|}$ of \mathbf{dia} over \mathbf{dia}' such that

1. for every $v' \in \text{in}(V'_j)$, if $v' \in \Pi_i(V_j)$ then $\mathbf{dia}'(f)(v') \in \Pi_j(\text{in}(V_j))$
2. for every $g' \in \Pi'_j$, if $\Pi_j(g)$ and $\mathbf{dia}'(f)(g')$ are defined then $\Pi_i(\mathbf{dia}'(f)(g'))$ is also defined
3. for every $i \in |I| \setminus J$, $\Pi_{i_{ac}}$ is injective

there is a unique morphism $\Pi: S \rightarrow S'$ in $\mathbf{r-DSGN}$ s.t $\mathbf{c-sign}(\Pi_i); \mathbf{rsign}(\Pi) = \mathbf{r-sign}(\Pi_i); \mathbf{c-sign}(\Pi'_i)$ for every $i \in |I|$, where $(\Pi_i: \mathbf{dia}(i) \rightarrow S)_{i \in |I|}$ and $(\Pi_i: \mathbf{dia}'(i) \rightarrow S')_{i \in |I|}$ are colimits of \mathbf{dia} and \mathbf{dia}' , respectively.



An outline of the proof of this result can be found in [77].

