Revisiting the Categorical Approach to Systems*

Antónia Lopes[1] and José Luiz Fiadeiro[1,2]

[1] Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa, Portugal
mal@di.fc.ul.pt

[2]ATX Software SA
Alameda António Sérgio 7, 1A, 2795-023 Linda-a-Velha, Portugal
jose@fiadeiro.org

Abstract. Although the contribution of Category Theory as a mathematical platform for supporting software development, in the wake of Goguen's approach to General Systems, is now reasonably recognised, accepted and even used, the emergence of new modelling techniques and paradigms associated with the "New-Economy" suggests that the whole approach needs to be revisited. In this paper, we propose some revisions capitalising on the lessons that we have learned in using categorical techniques for formalising recent developments on Software Architectures, Coordination Technologies, and Service-Oriented Software Development in general.

1 Introduction

In the early 70's, J.Goguen proposed the use of categorical techniques in General Systems Theory for unifying a variety of notions of system behaviour and their composition techniques [11,12]. The main principles of this approach are the following:

- components are modelled as objects;
- system configurations are modelled as diagrams that depict the way in which the components of the system are interconnected;
- the behaviour of a complex system is given through the colimit of its configuration diagram.

These categorical principles have been used to formalise different mathematical models of system behaviour [13,24,25,14], their logical specifications [2,7] and their realisations as parallel programs [5].

This categorical approach has also been used as a platform for comparing different modelling approaches and for giving semantics to the gross modularisation of complex systems (e.g., [26,4]). Grounded on the principle that a design formalism can be

* This research was partially supported by Fundação para a Ciência e Tecnologia through project POSI/32717/00 (FAST).

expressed as a category, we have established an algebraic characterisation of the notions of compositionality [8] and coordination [9], and formalised architectural principles in software design [10].

More recently, in the scope of a joint research effort with ATX Software – a Portuguese IT company with a very strong R&D profile – we have been engaged in the development of "coordination technologies" for supporting the levels of agility that are required on systems by the new ways in which companies are doing business, which includes coping with the volatility of requirements and the new paradigms associated with the Internet (B2C, B2B, P2P, etc). Our research has focused on a set of primitives centred around the notion of "coordination contract" with which OO languages such as the UML can be extended to incorporate modelling techniques that have been made available in the areas of Software Architectures, Parallel Program Design and Reconfigurable Distributed Systems.

The coordination technologies that we have been developing are largely "language independent" in the sense that they can be used together with a variety of modelling and design languages. Our experience in formalising this independence through the use of categorical techniques made us realise that the general approach as outlined above is far too simplistic and restrictive. On the one hand, it does not allow one to capture the typical restrictions that apply to the way components can be interconnected (e.g. [19]). These restrictions are of crucial importance because properties such as compositionality usually depend on the class of diagrams that represent correct configurations. On the other hand, as soon as more sophisticated models of behaviour are used, e.g. those in which non-determinism needs to be explicitly modelled, different mechanisms become necessary to support vertical and horizontal structuring. Hence, we cannot expect that the same notion of morphism can be used to support "component-of" relationships as required for horizontal structuring, and "refined-by" relationships as required by vertical structuring.

In this paper, we review the categorical approach in order to support the two new aspects motivated above, including a new characterisation of the notion of compositionality. We use a simple program design formalism — CommUnity, to illustrate the applicability of our proposal. Although we could have used instead a sophisticated and powerful formalism, the simplicity of CommUnity is ideal to present what we have found to be important features of the new breed of design formalisms that need to be accommodated in the categorical approach.

2 Capturing constraints on configurations

The basic motto of the categorical approach to systems design is that morphisms can be used to express interaction between components. More concretely, when a design formalism is expressed as a category – $\boldsymbol{DESC}$, a configuration of interconnected components can be expressed as a diagram in $\boldsymbol{DESC}$. The semantics of the configuration, i.e. the description of the system that results from the interconnections, can be obtained from the diagram by taking its colimit.

In the languages and formalisms that are typically used for the configuration of software systems, the interconnection of components is usually governed by specific rules, as happens with physical components, say in hardware or mechanical systems. A simple example can be given in terms of languages with I/O communication. Typically in these languages output channels cannot be connected to each other (see for instance [21,1]). Hence, it may happen that not every diagram represents a correct configuration. For instance, if ***DESC*** is not finitely cocomplete, then not every configuration of components is "viable", in the sense that it gives rise to a system (has a semantics). Clearly, in this situation not every configuration of components is correct.

However, the correctness of a configuration diagram cannot always be reduced to the existence of a colimit. There are situations in which the colimit exists (the category may even be cocomplete) but, still, the diagram may be deemed to be incorrect according to the rules set up by the configuration language. In situations like this, the rules that establish the correctness of configurations are not internalised in the structure of the category and have to be expressed as restrictions on the diagrams that are considered to be admissible as configurations. In the situations we have to deal with in practice, such correctness criteria are reflected on properties of the category that are related to properties of the development approach, such as the separation between computation and coordination, and compositionality. A specific example will be presented in section 3.

Definition 1 (Configuration criterion).
A configuration criterion *Conf* over ***DESC*** is a set of finite diagrams in ***DESC*** s.t. if ***dia***$\in$*Conf* then ***dia*** has a colimit.

In order to illustrate the notion of configuration criterion, we use a very simple interface definition language. Further on, we will use a program design language that is built over this interface language.

We consider that the interaction between a component and its environment is modelled by the simultaneous execution of actions and by exchanging data through shared channels. We will distinguish between input and output channels. We shall also assume a fixed set S of sort symbols for typing the data that can be exchanged through these channels.

An interface description θ is a triple $<O,I,\Gamma>$ where O and I are disjoint S-indexed families of sets and Γ is a finite set. The union of O and I is denoted by V.

The families O and I model, respectively, the output and the input channels, and the set Γ models the actions that a given system makes available through its public interface. For each sort s, O_s (resp. I_s) is the set of output (resp. input) channels that take data in s.

An interface morphism σ: $\theta_1 \rightarrow \theta_2$ is a pair $<\sigma_{ch},\sigma_{ac}>$ where σ_{ch}: $V_1 \rightarrow V_2$ is a total function s.t. (1) $\sigma_{ch}(V_{1_s}) \subseteq V_{2_s}$, for every $s \in S$ and (2) $\sigma_{ch}(O_1) \subseteq O_2$ and σ_{ac}: $\Gamma_2 \rightarrow \Gamma_1$ is a partial mapping.

An interface morphism σ from θ_1 to θ_2 supports the identification of a way in which a system with interface θ_1 is a component of another system, with interface θ_2. The function σ_{ch} identifies for each channel of the component the corresponding channel of the system. The partial mapping σ_{ac} identifies the action of the component that is involved in each action of the system, if ever. The typing of the channels has to be preserved and the output channels of the component have to be mapped to output channels of the system. Notice, however, that input channels of the component may be identified with output channels of the system. This is because the result of interconnecting an input of θ_1 with an output of another component of θ_2 results in an output for the whole system. This is different from most approaches in that communications are normally internalised and not made available in the interface of the resulting system. We feel that hiding communication is a decision that should be made explicitly and not as a default, certainly not as a part of a "minimal" operation of interconnection which is what we want to capture (see [25] for a categorical characterisation of hiding).

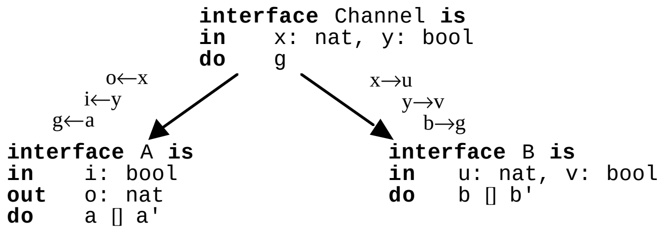
Interface descriptions and interface morphisms constitute a category which we will denote by ***INTF****.*

Interface morphisms can be used to establish synchronisation between actions of different components as well as the interconnection of input channels of one component with output channels of other components. However, they can also be used to establish the interconnection of an output channel of one component with an output channel of another component, a situation that in our formalism is ruled out.

In order to express that, in the interface description formalism we are presenting, output channels of an interface cannot be connected with output channels of the same or other interfaces we define which diagrams represent correct configurations.

A diagram $\boldsymbol{dia}: \boldsymbol{D} \rightarrow \boldsymbol{INTF}$ *is a configuration diagram iff if* $(\mu_n : \boldsymbol{dia}(n) \rightarrow \langle I,O,\Gamma \rangle)_{n \in |\boldsymbol{D}|}$ *is a colimit of* $\boldsymbol{dia}$*, then for every* $v \in O$*, there exists exactly one* n *in* $|\boldsymbol{D}|$ *s.t.* $\mu_n^{-1}(v) \cap O_{\boldsymbol{dia}(n)} \neq \emptyset$ *and, for such* n*,* $\mu_n^{-1}(v) \cap O_{\boldsymbol{dia}(n)}$ *is a singleton.*

That is to say, in a configuration diagram each output channel of the system can be traced back to one, and only one, output channel of one, and only one, component.



```
interface Channel is
in    x: nat, y: bool
do    g
```

```
interface A is
in    i: bool
out   o: nat
do    a [] a'
```

```
interface B is
in    u: nat, v: bool
do    b [] b'
```

An example of a configuration diagram is given above. This diagram defines a configuration of a system with two components whose interfaces are *A* and *B*. To make explicit the connections between the two components, we use a third interface – *Channel*. The reason we call this third interface "Channel" is that it can be viewed as a natural extension of the interconnection services that are made available through the interfaces (input and output channels and actions), i.e. a kind of “structured” or “complex” channel. Using the analogy with *hardware*, the channel and the two morphisms act as a "cable" that can be used to interconnect components.

More concretely, the diagram defines that the input channel *u* of *B* is connected with the output channel *o* of *A*, and that the input channels *i* of *A* and *v* of *B* are identified, i.e., in the resulting system, *A* and *B* input from the same source. Furthermore, the configuration establishes that *A* and *B* synchronise on actions *a* and *b*. Indeed, the colimit of the diagram depicted above returns

$$\begin{tikzcd}
\textbf{interface}\ \mathtt{A} \arrow[dr, swap, "\begin{array}{l} a\leftarrow ab \\ a'\leftarrow a'b' \\ a'\leftarrow a' \end{array}"] \arrow[dr, "\begin{array}{l} o\leftarrow o \\ i\leftarrow i \end{array}"] & & \textbf{interface}\ \mathtt{B} \arrow[dl, swap, "\begin{array}{l} u\rightarrow o \\ v\rightarrow i \end{array}"] \arrow[dl, "\begin{array}{l} ab\rightarrow b \\ a'b'\rightarrow b' \\ b'\rightarrow b' \end{array}"] \\
& \textbf{interface}\ \mathtt{AB}\ \textbf{is} &
\end{tikzcd}$$

```
interface AB is
in    i: bool
out   o: nat
do    ab [] a'b'[] a'[] b'
```

The action *ab* models the simultaneous execution of *a* in A and *b* in *B*. In the same way, action *a'b'* models the simultaneous execution of *a'* in A and *b'* in *B*. However, because *A* and *B* do not have to synchronise on actions *a'* and *b*', the interface *AB* has also two actions — *a'* and *b'* — modelling the isolated execution of *a'* in A and *b'* in B.

3 Horizontal vs Vertical Structuring

In the categorical approach that we have described in the previous section, morphisms are used for modelling “horizontal structuring”, i.e. to support the process of structuring complex systems by interconnecting simpler components. As we mentioned, a morphism identifies how its source is a component of its target.

However, for supporting software development, other relationships need to be accounted for that reflect notions of refinement between different levels of abstraction, e.g. [22,23,16]. In some cases, the morphisms used for capturing interconnections between components can also be used for expressing refinement. For instance, this is the case in the algebraic specification of abstract data types [2] and in the specification formalism of reactive systems presented in [8]. However, this is not always the case.

On the one hand, interconnection morphisms may serve the purpose of expressing refinement but may be too weak to represent the refinements that can occur during software development. For instance, this is the case in the algebraic approach proposed in [20] and refined in SPECWARE [23]. In this approach, interconnection morphisms are theorem-preserving translations of the vocabulary of a source specification into the terms of a target specification, whereas refinement morphisms are interconnection morphisms from the source specification to a conservative/definitional extension of the target specification (a refinement morphism from *A* to *B* is a pair of interconnection morphisms $A \rightarrow AasB \leftarrow B$).

On the other hand, morphisms used for interconnecting components may not be suitable for expressing refinement. In particular, this happens when a system is not necessarily refined by a system of which it is a component. For instance, in CSP [15] the functionality of a system is not necessarily preserved when it is composed with other systems. A process $a.P\square b.Q$, that is ready to execute a or b, has no longer this property if it is composed with a process that is only ready to execute a. Because the notion of refinement in CSP (based on the readiness semantics) requires that readiness be preserved by refinement, $a.P\square b.Q$ is not refined by $(a.P\square b.Q)\|\, a.R$.

Because of this, the integration of both dimensions – horizontal (for structuring) and vertical (for refinement) – may require that two different categories be considered.

Definition 2 (Design Formalism).
A design formalism is a triple <***c-DESC***,*Conf*,***r-DESC***> where ***c-DESC*** and ***r-DESC*** are categories and *Conf* is a configuration criterion over ***c-DESC*** s.t. 1. |***c-DESC***| = |***r-DESC***| and 2. *Isom(**c-DESC**)⊆Isom(**r-DESC**)*.

The objects of ***c-DESC*** and ***r-DESC***, which are the same, identify the nature of the descriptions that are handled by the formalism. The morphisms of ***r-DESC*** identify the vertical structuring principles, i.e., a morphism $\eta: S\rightarrow S'$ in ***r-DESC*** expresses that S' refines S, identifying the design decisions that lead from S to S'. The morphisms of ***c-DESC*** identify the horizontal structuring principles, that is to say, diagrams in ***c-DESC*** can be used to express how a complex system is put together through the interconnection of components. The description of the system that results from the interconnection is the description returned by the colimit of the configuration diagram and, hence, it is defined up to an isomorphism. Condition 2 ensures that the notion of refinement is "congruent" with the notion of isomorphism in ***c-DESC***. That is, descriptions that are isomorphic with respect to interconnections refine, and are refined exactly by, the same descriptions. Finally, as explained in the previous section, the relation *Conf* defines the correct configurations.

In order to illustrate that different categories may need to be used to formalise the horizontal and vertical structuring principles supported by a formalism, we use a language for the design of parallel programs — CommUnity.

CommUnity, introduced in [5], is similar to Unity [3] in its computational model but has a different coordination model. More concretely, the interaction between a program and the environment in CommUnity and Unity are dual: in Unity it relies on the sharing of memory and in CommUnity in the sharing (synchronisation) of actions.

A design in CommUnity is of the following form:

```
design P is
in     I
out    O
init   Init
do  $\mathop{[\!]}_{g\in\Gamma}$  g: [L(g),U(g) → $\mathop{\|}_{v\in D(g)}$ v:∈F(g,v)]
```

The sets of input and output channels, respectively, I and O, and the set of actions Γ constitute the interface of design *P*. *Init* is its initialisation condition. For an action g,

- *D(g)* represents the set of channels that action *g* can change.
- *L(g)* and *U(g)* are two conditions s.t. $L(g) \supset U(g)$ that establish an interval in which the enabling condition of any guarded command that implements *g* must lie. *L(g)* is a lower bound for enabledness in the sense that it is implied by the enabling condition (its negation establish a *blocking* condition) and *U(g)* is an upper bound in the sense that it implies the enabling condition. When *L(g)=U(g)* the enabling condition is fully determined and we write only one condition.
- for every channel *v* in *D(g)*, *F(g,v)* is a non-deterministic assignment: each time *g* is executed, *v* is assigned one of the values denoted by *F(g,v)*.

Formally,

A design is a pair $<\theta,\Delta>$ where θ is an interface and Δ, the body of the design, is a 5-tuple <Init,D,F,L,U> where:

- *Init is a proposition over the output channels;*
- *D assigns to every action a set of output channels;*
- *F assigns to every action g a non-deterministic command (when $D(g)=\emptyset$, the only available command is the empty one which we denote by **skip**);*
- *L and U assign to every action g a proposition.*

An example of a CommUnity design is the following bank account, where *CRED* is a given negative number, modelling the credit limit.

```
design BankAccount is
in    amount: nat
out   bal: int
init  bal=0
do    deposit: [true → bal:=bal+amount]
   [] withdraw: [bal-amount≥CRED,bal-amount≥0→bal:=bal-amount]
```

This design has only two actions modelling the deposit and the withdrawal of a given amount. This amount is transmitted to the component via the input channel *amount*. Deposits are always available for clients. Withdrawals are definitely refused if the requested amount is over the credit limit *(bal-amount<CRED)* and are definitely accepted when the balance is sufficient to cover the requested amount, i.e. the credit facility is not necessary *(bal-amount≥0)*. In all the other situations, i.e. when the credit facility is necessary *(0>bal-amount≥CRED)*, it was left unspecified whether withdrawals are accepted or refused. This form of underspecification (also called allowed non-determinism), as we will see, can be restricted during a refinement step.

In the previous section we have seen how the interconnection of components can be established at the level of interfaces. The corresponding configurations of designs are expressed in the following category of designs.

A design morphism $\sigma: P_1 \rightarrow P_2$ is an interface morphism from θ_1 to θ_2 s.t.:

1. *For all $g \in \Gamma_2$, if $\sigma_{ac}(g)$ is defined then $\sigma_{ch}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$;*
2. *For all $v \in O_1$, $\sigma_{ac}(D_2(\sigma_{ch}(v))) \subseteq D_1(v)$;*
3. *For all $g_2 \in \Gamma_2$, if $\sigma_{ac}(g_2)$ is defined then, for all $v_1 \in D_1(\sigma_{ac}(g_2))$,*
   $\models (F_2(g_2,\sigma_{ch}(v_1)) \subseteq \underline{\sigma}(F_1(\sigma_{ac}(g_2),v_1)));$

4. *$\models (I_2 \supset \underline{\sigma}(I_1))$;*
5. *For every $g_2 \in \Gamma_2$, if $\sigma_{ac}(g)$ is defined then $\models (L_2(g_2) \supset \underline{\sigma}(L_1(\sigma_{ac}(g_2))))$;*
6. *For every $g_2 \in \Gamma_2$, if $\sigma_{ac}(g)$ is defined then $\models (U_2(g_2) \supset \underline{\sigma}(U_1(\sigma_{ac}(g_2))))$.*

*Designs and design morphisms constitute a category **c-DSGN**.*

A design morphism $\sigma{:}P_1 \rightarrow P_2$ identifies P_1 as a component of P_2. Conditions 1 and 2 mean that the domains of channels are preserved and that an action of the system that does not involve an action of the component cannot transmit in any channel of the component. Conditions 3 and 4 correspond to the preservation of the functionality of the component design: (3) the effects of the actions can only be preserved or made more deterministic and (4) initialisation conditions are preserved. Conditions 5 and 6 allow the bounds that the design specifies for enabling condition of the action to be strengthened but not weakened. Strengthening of the two bounds 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. In other words, the enabling condition of a joint action is given by the conjunction of the enabling conditions of the corresponding actions in the involved components.

The correctness of a configuration of interconnected designs depends uniquely on the correctness of the underlying interconnection of interfaces (as defined in the previous section). Let ***Int*** be the forgetful functor from ***c-DSGN*** to ***INTF***.

*A diagram **dia**: **I**$\rightarrow$**c-DSGN** is a configuration diagram iff **dia;Int**: **I**$\rightarrow$**INTF** is an interface configuration diagram.*

It is interesting to notice that the category ***c-DSGN*** is not finitely cocomplete, namely O/O connections lead to design diagrams that may have no colimit. However, the existence of colimits is ensured for correct configurations.

In order to illustrate how a system design can be constructed from the design of its components, let us consider that every deposit of an amount that exceeds a given threshold must be signaled to the environment. This can be achieved by coupling the *BankAccount* with an *Observer* design.

```
design Observer is
in     v: nat
out    sg: bool
init   ¬sg
do     obsv&detect: [¬sg ∧ v>NUMBER → sg:=true]
   []  obsv: [¬sg ∧ v≤NUMBER → skip]
   []  reset: [sg → sg:=false]
```

The *Observer* design can be used to "observe" the execution of a given action, namely to detect if *v>NUMBER* holds when that action is executed. The design is initialised so as to be ready to execute *obsv&detect* or *obsv* depending on whether *v>NUMBER* holds or not. The assignment of *obsv&detect* sets the output channel *sg* to true, thus signaling to the environment the execution of the "observed" action under the condition *v>NUMBER*. The environment may acknowledge the reception of the signal through the execution of the action *reset*.

The diagram below gives the configuration of the required system. This diagram defines that the action that is "observed" by *Observer* is *deposit* and identifies the input channel *v* of the *Observer* with input channel *amount* of the *BankAcount*. That is to say, in this system the *Observer* signals the deposits of *amount*s greater than *NUMBER*.

$$\begin{tikzcd} & \texttt{Channel} \arrow[dl, "\begin{array}{r}\text{amount}\leftarrow\text{x}\\ \text{g}\leftarrow\text{deposit}\end{array}"'] \arrow[dr, "\begin{array}{l}\text{x}\rightarrow\text{v}\\ \text{obsv}\rightarrow\text{g}\\ \text{obsv\&detect}\rightarrow\text{g}\end{array}"] & \\ \texttt{BankAccount} & & \texttt{Observer} \end{tikzcd}$$

It remains to define the notion of refinement supported by CommUnity.

A refinement morphism σ: $P_1 \rightarrow P_2$ is an interface morphism from θ_1 to θ_2 s.t. conditions 1 to 5 of the definition of design morphism hold and

6. $\sigma_{ch}(I_1) \subseteq I_2$;
7. σ_{ch} *is injective;*
8. *For every* $g \in \Gamma_2$, $\sigma_{ac}^{-1}(g) \neq \emptyset$;
9. *For every* $g_1 \in \Gamma_1$, $\models (\underline{\sigma}(U_1(g_1)) \supset \bigvee_{\sigma_{ac}(g_2)=g_1} U_2(g_2))$.

A refinement morphism σ from P_1 to P_2 supports the identification of a way in which P_1 is refined by P_2. Each channel of P_1 has a corresponding channel in P_2 and each action *g* of P_1 is implemented by the set of actions $\sigma_{ac}^{-1}(g)$ in the sense that $\sigma_{ac}^{-1}(g)$ is a menu of refinements for action *g*. The actions for which σ_{ac} is left undefined (the new actions) and the channels that are not in $\sigma_{ch}(V_1)$ (the new channels) introduce more detail in the description of the design.

Conditions 6 to 8 express that refinement cannot alter the border between the system and its environment. More precisely, input channels cannot be made local by refinement (6), different channels cannot be collapsed into a single one (7) and every action has to be implemented (8). Condition 9 states that conditions *U(g)* can be weakened but not strengthened. Because condition 5 allows *L* to be strengthened, the interval defined by the conditions *L* and *U* of each action, in which the enabling condition of any guarded command that implements the action must lie, is required to be preserved or reduced during refinement. Finally, conditions 3 and 4 state that the effects of the actions of the more abstract design, as well as the initialisation condition, are required to be preserved or made more deterministic.

Preservation of required properties and reduction of allowed non-determinism are intrinsic to any notion of refinement, and justify the conditions that we have imposed on refinement morphisms. Clearly, the morphisms that we used for modelling interconnections do not satisfy these properties.

In order to illustrate refinement, consider another design of a bank account — *BankAccount2*. In this design, the history of the client's use of the credit facility is taken into account for deciding if a withdrawal is accepted or not. It is not difficult to see that

$$\sigma_{ch}\ \begin{array}{l} bal \rightarrow bal \\ amount \rightarrow amt \end{array} \qquad\qquad \sigma_{ac}\ \begin{array}{l} deposit \rightarrow deposit \\ n\_withd \rightarrow withdraw \\ s\_withd \rightarrow withdraw \\ reset \rightarrow \end{array}$$

define a refinement morphism from *BankAccount* to *BankAccount2*.

```
design BankAccount2 is
in    amt: nat
out   bal, count: int
init  bal=0∧count=0
do    deposit: [true → bal:=bal+amt]
   []  n_withd: [bal-amt≥0→bal:=bal-amt]
   []  s_withd:[bal-amt≥CRED∧count≤MAX→bal:=bal-amt||count:=count+1]
   []  reset: [true,false→count:=0]
```

The conditions under which withdrawals are available to clients, that was left unspecified in *BankAccount*, is completely defined in *BankAccount2*. The action *withdraw* of *BankAccount* is implemented in *BankAccount2* by *n_withd* and *s_withd*, where *n* and *s* stand for normal and special, respectively. Special withdrawals are counted and, as soon as their number reaches a given maximum, they are no longer accepted. Special withdrawals are accepted again if this restriction is lifted, which is modelled by the execution of the new action *reset*. Notice that *BankAccount2* also contains allowed non-determinism — the upper bound of enabledness of *reset* is false and, hence, it was not made precise in which situations the restriction is lifted.

4 Compositionality

Compositionality 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). Compositionality is usually described as the property according to which the refinement of a composite system can be obtained by composing refinements of its components. However, when component interconnections are explicitly modelled through configurations, the refinement of the interconnections has to be taken into account. In these situations, the compositionality of a formalism can only be investigated w.r.t. a given notion of refinement of interconnections. Such notion, together with the refinement principles defined by ***r-DESC***, establishes the refinement of configurations. Configurations and their refinement principles can be themselves organised in a category as follows.

Definition 3 (Configuration Refinement Criterion).
A configuration refinement criterion for a design formalism <***c-DESC***,*Conf*,***r-DESC***> is any category ***CONF*** that satisfies the following conditions:

1. The objects of ***CONF*** are the finite diagrams in ***c-DESC***;
2. If η: ***dia***→***dia'*** is a morphism in ***CONF*** and ***I*** and ***I'*** are the shapes of, respectively, ***dia*** and ***dia'***, then η is an |***I***|-indexed family of morphisms η_i: ***dia****(i)*→***dia'****(i)* in ***r-DESC***.

Because diagrams are functors, it may seem that diagram morphisms should be merely natural transformations (as they are, for instance, in SPECWARE). However, this would be very restrictive because it would enforce that the shape of diagrams that define configurations be preserved during refinement.

Once we have fixed such a configuration refinement criterion ***CONF***, compositionality can be formulated as the property according to which we can pick arbitrary refinements of the components of a system *Sys*, and interconnect these more concrete descriptions with arbitrary refinements of the connections used in *Sys*, and obtain a system that still refines *Sys*.

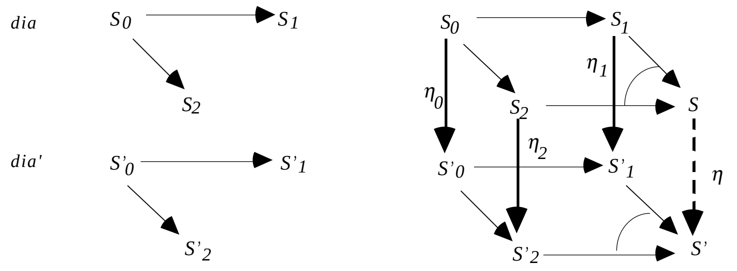


Figure 1

More precisely, if ***dia*** and ***dia'*** define the configuration of, respectively, *Sys* and *Sys'*, and η: ***dia***$\rightarrow$***dia'*** is a morphism in ***CONF***, then η defines, in a unique way, a refinement morphism in ***r-DESC*** from *Sys* to *Sys'* whose design decisions are compatible with the design decisions that were taken in the refinement of the components (see figure 1). That is to say, there exists a unique refinement morphism η in ***r-DESC*** from *Sys* to *Sys'* s.t. η is compatible with each of the morphisms $(\eta_i$: ***dia****(i)*$\rightarrow$***dia'****(i)*$)_{i\in|\mathbf{I}|}$.

Once again, it is not possible to define the meaning of "compatibility of design decisions" independently of the context. As happens with the notion of refinement of interconnections, the meaning of "compatibility of design decisions" must be fixed *a priori*.

The compatibility of η and η_i depends on the morphisms that identify ***dia****(i)* and ***dia'****(i)* as components of, respectively, *Sys* and *Sys'*. In this way, we consider that the compatibility of design decisions is abstracted as a 4-ary relation between *Mor(**r-DESC**)*, *Mor(**c-DESC**)*, *Mor(**r-DESC**)* and *Mor(**c-DESC**).*

Definition 4 (Criterion of Compatibility of Design Decisions). A criterion of compatibility of design decisions for a design formalism <***c-DESC***,*Conf*, ***r-DESC***> is a 4-ary relation *Crt* between *Mor(**r-DESC**)*, *Mor(**c-DESC**)*, *Mor(**r-DESC**)* and *Mor(**c-DESC**)* such that

1. If $Crt(\eta_i,\mu_i,\eta,\mu'_i)$ then $dom(\eta_i)=dom(\mu_i)$, $dom(\mu'_i)=cod(\eta_i)$, $dom(\eta)= cod(\mu_i)$ and $cod(\eta)=cod(\mu'_i)$;
2. If $Crt(\eta_i,\mu_i,\eta,\mu'_i)$ and κ: $cod(\eta)\rightarrow S''$ is a morphism in ***c-DESC*** and also in

r-DESC, then $Crt(\eta_i,\mu_i,(\eta;\kappa),(\mu'_i;\kappa))$;

3. If ***dia***: ***I***$\rightarrow$***c-DESC*** is a configuration diagram, $(\mu_i$: ***dia***$(i)\rightarrow S)_{i\in|\boldsymbol{I}|}$ is a colimit of ***dia*** and $Crt(\eta_i,\mu_i,\eta,\mu'_i)$, $Crt(\eta_i,\mu_i,\eta',\mu'_i)$ for every $i\in|\boldsymbol{I}|$, then $\eta=\eta'$.

The idea is that $Crt(\eta_i\text{: } S_i\rightarrow S'_i,\mu_i\text{: } S_i\rightarrow S,\eta\text{: } S\rightarrow S',\mu'_i\text{: } S'_i\rightarrow S')$ expresses that the design decisions defined by η agree with the decisions defined by η_i, taking into account the morphisms that identify S_i as a component of S and S'_i as a component of S' (see Fig. 2).

$$\begin{tikzcd}
S_i \arrow[r, "\mu_i"] \arrow[d, "\eta_i"'] & S \arrow[d, dashed, "\eta"'] \\
S'_i \arrow[r, "\mu'_i"] & S'
\end{tikzcd}$$

Figure 2

Condition 1 states the obvious restrictions on the domains and codomains of the four morphisms. Condition 2 ensures that the criterion is consistent w.r.t. the composition of morphisms that are simultaneously refinement and interconnection morphisms. Condition 3 means that if the refinement of the components of a system *Sys* is given by $(\eta_i\text{: } S_i\rightarrow S'_i)_{i\in|\boldsymbol{I}|}$, and *Sys'* is a system having $(S'_i)_{i\in|\boldsymbol{I}|}$ as components, then there exists a unique way in which *Sys'* may be a refinement of *Sys*.

Definition 5 (Compositional Formalism).
Let *Fd*=<***c-DESC***,*Conf*,***r-DESC***> be a design formalism, ***CONF*** a configuration refinement criterion for *Fd* and *Crt* a criterion of compatibility of design decisions for *Fd*. *Fd* is compositional w.r.t. ***CONF*** and *Crt* iff, for every morphism η: ***dia***$\rightarrow$***dia'*** in ***CONF***, there exists a unique morphism $\eta\text{: } S\rightarrow S'$ in ***r-DESC*** s.t. $Crt(\eta_i,\mu_i,\eta,\mu'_i)$, for every $i\in|\boldsymbol{I}|$, where $(\mu_i$: ***dia***$(i)\rightarrow S)_{i\in|\boldsymbol{I}|}$ and $(\mu'_i$: ***dia'***$(i)\rightarrow S')_{i\in|\boldsymbol{I'}|}$ are colimits of, respectively, ***dia*** and ***dia'***.

The notion of compositionality had been investigated and characterised in [8] for the situation in which the interconnection and refinement morphisms coincide. More concretely, when

- ***c-DESC***=***r-DESC***;
- a refinement morphism η from the diagram ***dia***: ***I***$\rightarrow$ ***c-DESC*** to the diagram ***dia'***: ***I'***$\rightarrow$***c-DESC*** exists iff ***I*** is a subcategory of ***I'*** and, in this case, η is an $|\boldsymbol{I}|$-indexed family $(\eta_i$: ***dia***$(i)\rightarrow$***dia'***$(i))_{i\in|\boldsymbol{I}|}$ of morphisms such that ***dia***$(f);\eta_j$= η_i;***dia'***(f), for every morphism $f\text{: } i\rightarrow j$ in ***I***;
- the criterion of compatibility of design decisions *Crt* is defined by $Crt(\eta_i,\mu_i,\eta,\mu'_i)$ iff $\mu_i;\eta=\eta_i;\mu'_i$;

compositionality is just a consequence of the universal property of colimits.

Figure 3 shows an example with the interconnection of two components through a channel. In this case, if $\sigma_i;\eta_i=\eta_0;\sigma'_i$, for $i=1,2$, then the universal property of colimits ensures that there exists a unique morphism η from S to S' such that $\mu_i;\eta=\eta_i;\mu'_i$, for *i=1,2*. In this case, notice that the interconnection of the two components defined by $<\sigma_1,\sigma_2>$ and the refinement morphisms $<\eta_1,\eta_2>$ define an interconnection of the more concrete descriptions S'_1 and S'_2, that is given by $<\sigma_1;\eta_1,\sigma_2;\eta_2>$. This new configuration is, by definition, a refinement of the initial one (where η_0 is the identity).

$$\begin{tikzcd}
S_0 \arrow[rr, "\sigma_1"] \arrow[dr, "\sigma_2"] \arrow[dd, "\eta_0"'] & & S_1 \arrow[dr, "\mu_1"] \arrow[dd, "\eta_1"] & \\
& S_2 \arrow[rr, "\mu_2"] \arrow[dd, "\eta_2"] & & S \arrow[dd, dashed, "\eta"] \\
S'_0 \arrow[rr, "\sigma'_1"] \arrow[dr, "\sigma'_2"'] & & S'_1 \arrow[dr, "\mu'_1"] & \\
& S'_2 \arrow[rr, "\mu'_2"'] & & S'
\end{tikzcd}$$

Figure 3

$$\begin{tikzcd}
Int(P_i) \arrow[r, "Int(\mu_i)"] \arrow[d, "r\text{-}Int(\eta_i)"'] & Int(P) \arrow[d, dashed, "r\text{-}Int(\eta)"] \\
Int(P'_i) \arrow[r, "Int(\mu'_i)"'] & Int(P')
\end{tikzcd}$$

Figure 4

For illustrating the characterisation of compositionality we have proposed, we use CommUnity again. We define informally in which conditions a configuration diagram ***dia': I'→c-DSGN*** is a refinement of another configuration diagram ***dia: I→c-DSGN***, given that the refinement of the components is defined by $(\eta_i$: ***dia(i)→dia'(i)****)*.

Basically, the morphisms of ***CONF***^{p} define that the system architecture, seen as a collection of components and "cables", cannot change during a refinement step (***I=I'***). However, the "cables" used to interconnect the more abstract designs can be replaced by "cables" with more capabilities. More precisely, the replacement of an interconnection ***dia*** by ***dia'*** must satisfy the following conditions:

– The interconnection ***dia'*** must be consistent with ***dia***, i.e., the i/o communications and synchronisations defined by ***dia*** have to be preserved.

– The diagram ***dia'*** cannot establish the instantiation of any input channel that was left “unpluged” in ***dia***. That is to say, the input channels of the composition are preserved by refinement.

– The diagram ***dia'*** cannot establish the synchronisation of actions that were defined as being independent in ***dia***.

For the compatibility of design decisions Crt^p we simply consider the commutability of the diagram in ***INTF*** depicted in Figure 4.

The design formalism CommUnity is compositional w.r.t. ***CONF***^{p} *and* Crt^p*.*

The formal definitions as well as the proof of this result can be found in [18].

It is important to notice that the adoption of different configuration refinement criteria gives rise to different notions of compositionality. For instance, the criterion ***CONF***^{p} we have defined for CommUnity prevents the addition of new components to the system (*superposition*) during a refinement step, even if they are just "observers", i.e., they do not have effects in the behaviour of the rest of the system. However, it is also possible to prove the compositionality of CommUnity w.r.t. a configuration refinement criterion that allows the addition of new components to the system. For simplicity, we have decided not to present this other criterion (which is more difficult to express because it requires further conditions over the new components and the new "cables").

5 Conclusions

In this paper, we proposed a revision of Goguen's categorical approach to systems design in order to make it applicable to a larger number of design formalisms. The new aspects of the revised approach are the ability to represent the specific rules that may govern the interconnection of components and the ability to support the separation between horizontal and vertical structuring principles. Furthermore, we proposed a characterisation of compositionality of refinement with respect to composition in this revised categorical framework. The explicit description of component interconnections leads us to a definition of compositionality that depends on a notion of refinement of interconnections and on the meaning of "compatibility of design decisions".

The work reported in this paper was motivated by our experience in using the categorical platform to express sophisticated formalisms, namely formalisms for architectural design [17]. For instance, we have perceived that the property according to which a framework for system design supports the separation between computation and coordination (which is a good measure of the ability of a formalism to cope with the complexity of systems) is, in general, false if *ad hoc* interconnections are permitted. In a similar way, interconnections of designs can be synthesised from interconnections of interfaces in coordinated frameworks only if interconnections are severely restricted. From this experience, the need for expressing the correctness of components configurations emerged as reported in the paper.

Further work is going on which explores the impact of this revision on the use of the categorical platform to map and relate different formalisms. As showed in [6], the categorical platform can be used to formalise the integration of different formalisms. This integration is important because it supports the use of multiple formalisms in software development and promotes reuse.

References

1. R.Alur and T.Henzinger, "Reactive Modules", in *Proc. LICS'96*, 207-218.
2. R.Burstall and J.Goguen, "Putting Theories Together to Make Specifications", in *Proc. 5th IJCAI*,1045-1058, 1977.
3. K.Chandy and J.Misra, *Parallel Program Design - A Foundation*, Addison-Wesley, 1988.
4. T.Dimitrakos, "Parametrising (algebraic) Specification on Diagrams", *Proc. 13th Int.Conference on Automated Software Engineering*, 1998.

5. J.L.Fiadeiro and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming*, 28:111-138, 1997.
6. J.L.Fiadeiro and T.Maibaum, "Interconnecting Formalisms: supporting modularity, reuse and incrementality", in G.E.Kaiser (ed), *Proc. 3rd Symp. on Foundations of Software Engineering*, 72-80, ACM Press, 1995.
7. J.L.Fiadeiro and T.Maibaum, "Temporal Theories as Modularisation Units for Concurrent System Specification", in *Formal Aspects of Computing* 4(3), 1992, 239-272.
8. J.L.Fiadeiro, "On the Emergence of Properties in Component-Based Systems", in M.Wirsing and M.Nivat (eds), *AMAST'96*, LNCS 1101, Springer-Verlag 1996, 421-443.
9. J.L.Fiadeiro and A.Lopes, "Algebraic Semantics of Coordination", in A.Haeberer (ed), *AMAST'98*, LNCS 1548, Springer-Verlag.
10. J.L.Fiadeiro and A.Lopes, "Semantics of Architectural Connectors", in M.Bidoit and M.Dauchet (eds), *TAPSOFT'97*, LNCS 1214, 505-519, Springer-Verlag, 1997.
11. J.Goguen, "Categorical Foundations for General Systems Theory", in F.Pichler and R.Trappl (eds), *Advances in Cybernetics and Systems Research*, 121-130, Transcripta Books, 1973.
12. J.Goguen and S.Ginalli, "A Categorical Approach to General Systems Theory", in G.Klir(ed), *Applied General Systems Research*, Plenum 1978, 257-270.
13. J.Goguen, "Sheaf Semantics for Concurrent Interacting Objects", *Mathematical Structures in Computer Science* 2, 1992.
14. M.Große-Rhode, "Algebra Transformations Systems and their Composition", in E.Astesiano (ed), *FASE'98*, LNCS 1382, 107-122, Springer-Verlag, 1998.
15. C.A.Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
16. K.Lano and A.Sanchez, "Design of Reactive Control Systems for Event-Driven Operations", in J.Fitzgerald, C.Jones and P.Lucas (eds), *Formal Methods Europe 1997*, LNCS 1313, 142-161, Springer-Verlag, 1997.
17. A.Lopes and J.L.Fiadeiro, "Using Explicit State to Describe Architectures", in J.Finance (ed), *FASE'99*, LNCS 1577, 144-160, Springer-Verlag, 1999.
18. A.Lopes, *Non-determinism and Compositionality in the Specification of Reactive Systems*, PhD Thesis, University of Lisbon, 1999.
19. J.Magee and J.Kramer, "Dynamic Structures in Software Architecture", in *4th Symposium on Foundations of Software Engineering*, ACM Press 1996, 3-14.
20. T.Maibaum, P.Veloso and M.Sadler, "A Theory of Abstract Data Types for Program Development: Bridging the Gap?", in H.Ehrig, C.Floyd, M.Nivat and J.Thatcher (eds) *TAPSOFT'85*, LNCS 186, 1985, 214-230.
21. Z.Manna and A.Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.
22. D.Smith, "Constructing Specification Morphisms", *Journal of Symbolic Computation* 15 (5-6), 571-606, 1993.
23. Y.Srinivas and R.Jüllig, "Specware™:Formal Support for Composing Software", in B.Möller (ed) *Mathematics of Program Construction*, LNCS 947, 399-422, Springer-Verlag, 1995.
24. G.Winskel, "A Compositional Proof System on a Category of Labelled Transition Systems", *Information and Computation* 87:2-57, 1990.
25. G.Winskel and M.Nielsen, "Models for Concurrency", *Handbook of Logic in Computer Science*, S.Abramsky, D.Gabbay and T.Maibaum (eds), Vol.4, 1-148, Oxford University Press 1995.
26. V.Wiels and S. Easterbrook, "Management of Evolving Specifications using Category Theory", *Proc. 13th Int.Conference on Automated Software Engineering*, 1998.