Superposition

Lopes, Antónia; Fiadeiro, José Luiz

Published in:
Electronic Notes in Theoretical Computer Science

DOI:
10.1016/S1571-0661(05)82562-9

Publication date:
2002

Electronic Notes in Theoretical Computer Science 70 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 15 pages

Superposition: Composition vs Refinement of Non-deterministic, Action-Based Systems

Antónia Lopes [1]

Department of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa, Portugal

José Luiz Fiadeiro [2]

ATX Software SA
Al.António Sérgio 7, 1-C, 2795-023 Linda-a-Velha, Portugal

Abstract

We show that the traditional notion of superposition as used for supporting parallel program design can subsume both composition and refinement relationships when non-deterministic behaviour of action-based systems is considered. For that purpose, we rely on a categorical formalisation of program design in the language CommUnity that we are also using for addressing architectural concerns.

1 Introduction

Superposition (or superimposition) has been proposed (and used) as a powerful mechanism for structuring the development of parallel programs and distributed systems [6,8,5,13,3]. It supports a layered approach to systems design by which we are allowed to build on (partially) developed components by augmenting them with new features while preserving the original properties.

Typically, superposition is formalised as a relationship between two units of design (programs, commands, actions, etc), which blends well with the traditional stepwise refinement method initiated by Dijkstra [7]: starting with a specification of the intended behaviour of the system, one progresses by adding detail (superposing activities, computation mechanisms, etc) until a design is reached that satisfies certain criteria. These criteria, the notions of "specification" and "unit of design", as well as the notion of "progress"

[1] Email: mal@di.fc.ul.pt
[2] Email: jose@fiadeiro.org

(superposition step) itself, vary according to the formalism that is being used to support development. Nevertheless, there is a reassuring uniformity of concepts that one normally takes as the result of a process of crystallisation that has lasted decades — one of the few Computing Science can claim...

There is, however, an interesting twist in the notion of superposition that is best captured in the way it is formalised by Francez and Forman for a language called "Interacting Processes" [8]. They capture superposition as a (generalised) parallel composition operator that, basically, introduces a rendez-vous style of synchronisation between processes. This view is not inconsistent with the previous one: it is just more "operational" in the sense that it provides a means of achieving the extension required in a refinement step by interconnecting components. In the past [10], we have actually shown how the two notions can be brought together in a mathematical framework in which the relational, transformation-based view is captured through a notion of homomorphism between units of design for which the composition operation is achieved through a colimit construction.

What is interesting in this dual view of superposition is that composition and refinement are not necessarily two sides of the same coin. For instance, in CSP [12], 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 $Q \parallel Q'$). The notion of superposition as composition also supports a process of system development, but one that aims at constructing complex systems from simpler components rather than adding detail to more abstract but, in some sense, "equivalent" representation of the whole system. In fact, both processes are inherent to Software Engineering; what one often disregards is the fact that they are not the same

This seems to suggest that there is a separation of concerns that is worth making in regard to the role of superposition in program development: the composition and the refinement views. Our purpose in this paper is to contribute to this goal by showing that the difference between these two views of superposition can be related to another fundamental, but often ignored, separation of concerns: that between non-determinism and underspecification. More concretely, we show that the composition-view can be associated with a (horizontal) process of removing non-determinism from a system by constraining the way its components interact, whereas the refinement-view can capture an orthogonal (hence, vertical) process of making specifications more precise. Finally, we show how the two processes can be related, leading to a notion of compositionality that is at the heart of what is known today as the "architectural" approach to system development. Our discussion will be conducted over a generalisation of the language that we introduced in [10]: CommUnity.

2 Component Design in CommUnity

CommUnity is a parallel program design language in the style of Unity that was initially proposed in [10] to show how programs fit into Goguens categorical approach to General Systems Theory [11]. Since its original definition, the language and the design framework have been extended to provide a formal platform for architectural design of open, reactive, reconfigurable systems [9,19,17]. One of the extensions we have made to CommUnity concerns the support for higher levels of design. The basic building blocks of the formalism are not programs but abstractions of programs called designs that can be refined into programs in later stages of the development process.

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. Although 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, data refinement is more concerned with the computational aspects of systems than with composition. Hence, we focus our attention on the refinement of designs for a fixed choice of data types: we assume a pre-defined collection of data types given in the form of a first-order algebraic specification, i.e. a data signature $< S, \Sigma >$, where S is a set (of sorts) and Ω is a $S^* \times S$-indexed family of sets (of operations), together with a collection Φ of first-order sentences specifying the functionality of the operations. An approach similar to the refinement calculus for Actions Systems [4] could be adopted to support also data refinement.

A CommUnity component design is of the form.

design P **is**		
out		out(V)
in		in(V)
prv		prv(V)
do	$[]_{g \in sh(\Gamma)}$	g[D(g)]: L(g), U(g) $\rightarrow$ R(g)
	$[]_{g \in prv(\Gamma)}$	g[D(g)]: L(g), U(g) $\rightarrow$ R(g)

V is the set of *variables* of design P. There are input variables, *output variables* and *private variables*. Input variables are read-only: the component has no control on their values. Output and private variables are controlled locally by the component, i.e. they cannot be modified by the environment. Output variables can be read by the environment but private variables cannot. We use $loc(V)$ to denote the union $prv(V) \cup out(V)$ of local variables. Each variable v is typed with a sort $sort(v) \in S$.

Γ is the set of *action names* of design P. The named actions can be declared either as *private* or *shared* (for simplicity, we only declare which actions are private). Private actions represent internal computations in the sense that their execution is uniquely under the control of P. Shared actions represent possible interactions between the component and the environment, meaning that their execution is also under the control of the environment. The significance of naming actions will become obvious in section 4; the idea is that action names, as in IP [8], provide points of *rendez-vous* at which components can synchronise. For each action name g:

- $D(g)$ consists of the local variables that can be effected by executions of the action named by g — 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 variable v, we will also denote by $D(v)$ the set of actions g such that $v \in D(g)$.
- $L(g)$ and $U(g)$ are two conditions such that $U(g) \Rightarrow L(g)$. 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.
- $R(g)$ is a condition on V and $D(g)'$ where by $D(g)'$ we denote the set of primed local variables from the write frame of g. As usual, these primed variables account for references to the values that the variables take after the execution of the action. $R(g)$ is a specification of the effects of the action g on the state of the component. These conditions are usually a conjunction of implications of the form $pre \Rightarrow pos$ where pre does not involve primed variables. They correspond to pre/post-condition specifications in the sense of Hoare. When $R(g)$ is such that the primed version of each local variable 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*.

It is important to notice that, in this way, actions may be underspecified. On the one hand, their enabling conditions may not be fully determined and, hence, they are subject to refinement by reducing the interval established by L and U. On the other hand, the effects of actions on the variables may also not be fully determined and are also subject to refinement by strengthening R.

We present below an example of a CommUnity design that models a banking account with a credit facility.

```
design account is
in am:nat
out bal:int
do   dep[bal]: true → bal'=v+bal
[]   wit[bal]: bal+CRE≥am, bal≥am →
                                    (bal≥am ⊃ bal'=bal-am) ∧
                                    (bal<am ⊃ bal'≤bal-am)
```

Deposits of any amount are always accepted and effect the balance as expected. Withdrawals are accepted whenever the balance is enough to satisfy the requested amount and are refused if the requested amount is greater than the balance plus a given credit amount. In the other situations, i.e., when $am - CRE \leq bal \leq am$, the acceptance of a withdrawal is left unspecified. Later in the development process, one may refine this design in order to model a specific policy on withdrawals as long as it complies with the limits established in this design. The effects of withdrawals in the balance are not completely specified either, leaving room for penalties to be introduced in the case of withdrawals that leave the account overdrawn.

```
design counter is
in  val:int,day:nat
out count:nat
prv d:nat
do   chg: true → d:=day ‖ count:=if(val<0,count+(day-d),count)
[]   reset: true, false → d:=day ‖ count:=0
```

Programs

When, for every $g \in \Gamma$, $L(g)$ and $U(g)$ coincide, and the relation $R(g)$ defines a conditional multiple assignment, the design is called a *program*. Notice that a program with a non-empty set of input variables is *open* in the sense that its execution is only meaningful in the context of a configuration in which these inputs have been instantiated with variables 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 [17] for a model-theoretic semantics of CommUnity.

As an example of a program consider the design *counter*. This program

models a counter that counts how many days a value has been negative since the last time it was reset. It receives from the environment the value it has to monitoring and the current day, through the input variables *val* and *day*, respectively. Furthermore, through the execution of action *chg*, it expects to be notified each time *val* is updated. In order to count the elapsed time, it keeps in its private memory the date of the last update (variable *d*).

3 Superposition

The design of a banking account with a policy on withdrawals that uses the number of days the account has been overdrawn to decide if withdrawals are accepted or not, can be obtained from the design *account* presented before by superposing a *counter*:

```
design monregaccount is
in  am:nat, day:nat
out count:nat, bal:int
prv d:nat
do  dep[bal,d,count]: true→bal'=v+bal ∧
                                d'=day ∧
                                (bal<0 ⊃ count'=count+(day-d)) ∧
                                (bal≥0 ⊃ count'=count)
[]  wit[bal,d,count]: bal+CRE≥am ∧ count<LIM, bal≥am ∧ count<LIM →
                                d'=day ∧
                                (bal≥am ⊃ bal'=bal-am) ∧
                                (bal<am ⊃ bal'≤bal-am) ∧
                                (bal<0 ⊃ count'=count+(day-d)) ∧
                                (bal≥0 ⊃ count'=count)
[]  reset[d,count]: true, false → d'=day ∧ count'=0
```

In the design *monregaccount* we have introduced the new input variable *day* and the new local variables *count* and *d* (superposed variables) and the new action *reset*. The input variable *day* gives the current day. The variable *count* counts how many days the account has been overdrawn since the last time the counter was reset. This is achieved through the use of the auxiliary variable *d* that stores the last time the balance was changed.

It is important to notice that, in this design, the account is not only monitored but also regulated. This is reflected in the fact that the enabling condition of withdrawals is strengthened. In this design, any request for a withdrawal is refused if the number of days the account has been overdrawn exceeds a certain limit.

In [10] it was shown that several notions of superposition can be formalised in terms of morphisms of programs. Here, we extend the formalisation of the so-called *regulative superposition* for CommUnity designs. Regulative superposition requires that the functionality of the base program in terms of the assignments of its variables be preserved and allows for the enabling condition of its actions to be strengthened.

Definition 3.1 A composition morphism of designs $\sigma : P_1 \to P_2$ consists of a total function $\sigma_{var} : V_1 \to V_2$ and a partial mapping $\sigma_{ac} : \Gamma_2 \to \Gamma_1$ s.t.:

1. for every $v \in V_1, i \in in(V_1), o \in out(V_1), x \in prv(V_1)$
   $sort_2(\sigma_{var}(v)) = sort_1(v)$
   $\sigma_{var}(o) \in out(V_2)$
   $\sigma_{var}(i) \in out(V_2) \cup in(V_2)$
   $\sigma_{var}(x) \in prv(V_2)$
2. for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined
   if $g \in sh(\Gamma_2)$ then $\sigma_{ac}(g) \in sh(\Gamma_2)$
   if $g \in prv(\Gamma_2)$ then $\sigma_{ac}(g) \in prv(\Gamma_2)$
3. for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined
   $\sigma_{var}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$
   $\sigma_{ac}(D_2(\sigma_{var}(v))) \subseteq D_1(v)$ for every $v \in loc(V_1)$
   $\Phi \models (R_2(g) \Rightarrow \sigma(R_1(\sigma_{ac}(g))))$
   $\Phi \models (L_2(g) \Rightarrow \sigma(L_1(\sigma_{ac}(g))))$
   $\Phi \models (U_2(g) \Rightarrow \sigma(U_1(\sigma_{ac}(g))))$

where we have used also σ to denote the extension of the morphism to the language of expressions and conditions. Designs and composition morphisms define a category **c-DSGN**.

A morphism $\sigma : P_1 \to P_2$ identifies a way in which P_1 is "augmented" to become P_2 so that it can be considered as having been obtained from P_1 through the superposition of additional behaviour, namely the interconnection of one or more components. In other words, σ identifies P_1 as a component of P_2. The map σ_{var} identifies for every variable of the component the corresponding variable of the system. The first group of constraints also establishes that sorts of variables have to be preserved. Notice, however, that input variables of a component can become output variables of the system. This is because the result of interconnecting an input variable of a component with an output variable of another component in the system is an output variable of the system. Mechanisms for hiding communication, i.e. making it private, can be applied, but they are not the default in a configuration.

The partial mapping σ_{ac} identifies the action of the component that is involved in each action of the system, if ever. The second group of constraints states that the type of actions is preserved. The last group of conditions on actions requires that change within a component is completely encapsulated in the structure of actions defined for the component and that the computations performed by the system reflect the interconnections established between its components. The two conditions on write frames imply that actions of the system in which a component is not involved cannot have local variables of the component in their write frame. The third condition reflects the fact that the effects of the actions of the components can only be preserved or made more deterministic in the system. The last two conditions allow the bounds that the design specifies for the enabling of the action to be strengthened but not weakened. Strengthening of these 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. For instance, the *composition* morphism that captures the fact that *monregaccount* can be regarded as the result of applying superposition to the design *account* is

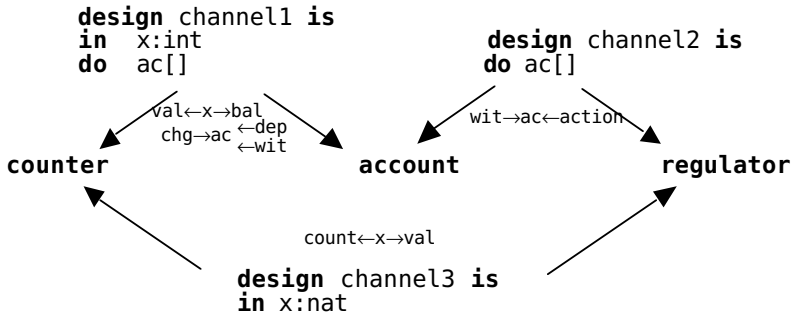
$$\sigma : account \rightarrow monregaccount$$

where σ_{var} is the inclusion function between the corresponding set of variables and $\sigma_{ac}(dep) = dep$ and $\sigma_{ac}(wit) = wit$. $\sigma_{ac}(reset)$ is undefined because this action does not involve the base design.

An interesting property of the composition morphisms we have just defined is that they can also be used to support the process of structuring complex systems by interconnecting simpler components. More concretely, configurations of complex systems can be described by diagrams in the category **c-DSGN**. In order to illustrate the way *composition* morphisms can be used for defining systems configurations consider the diagram below where *regulator* is the following design.

```
design regulator is
in   val:nat
do   action: val<LIM → skip
```

This diagram defines the configuration of a system with three components — *account*, *counter* and *regulator*. The other designs and the composition morphisms define the interactions between these components in the system.



This diagram defines

- An i/o interconnection between variables *bal* of *account* and *val* of *counter* and the synchronisation of *account* and *counter* each time the first wants to perform *dep* or *wit* and the later wants to perform *chg*. In this way, it is established that the balance of the *account* is the value subject of the monitoring carried out by the *counter*. In order to ensure the counter is notified each time the balance is updated, given that *dep* and *wit* may change the balance of the account, both actions were synchronised with the action *chg*.
- An i/o interconnection between variables *val* of *regulator* and *count* of counter that establishes that the enabling condition of action of *regulator* depends on the value of the counter.
- The synchronisation of action *wit* of *account* with *action* of regulator. This

determines that requests of withdrawals in this system are refused when the number of days the account has been overdrawn reaches a certain limit.

Through a universal construction of category theory – the colimit construction – it is possible to internalise the interactions explicitly described in a configuration diagram and obtain a design for the system as a whole. Colimits in **c-DSGN** capture a generalised notion of parallel composition in which the designer makes explicit what interconnections are used between components:

- variables involved in each i/o-communication established by the configuration are amalgamated;
- every set $\{g_1, \cdots, g_n\}$ of actions that are synchronised through the interconnections is represented by a single action $g_1 \parallel \cdots \parallel g_n$ whose occurrence captures the joint execution of the actions in the set;
- the transformations performed by the joint action are specified by the conjunction of the specifications of the local effects of each of the synchronised actions, and the bounds on the enabling condition of joint actions are also obtained through the conjunction of the bounds specified by the components.

Not surprisingly, the design *monregaccount* presented before is a colimit of the diagram above. This shows that the several enhancements of the original design *account* that are underlying the superposition process can be developed independently, as autonomous components, and hence can be used (and reused) in different contexts.

4 Refinement

Composition morphisms as defined in the previous section do not capture a refinement relation. In order to make this clear, let us analyse the example that we used for illustrating composition.

We started with the design *account* in which it was left unspecified if a withdrawal is accepted or refused in the situations in which the balance is not enough to satisfy the requested amount but becomes enough if a certain credit amount is provided. However, the design *account* establishes the limits for the policies on withdrawals that can be adopted in later stages of design:

a) withdrawals cannot be refused whenever, on its own, the balance is enough to satisfy the requested amount
b) withdrawals will be refused if the requested amount is greater than the balance plus the agreed credit amount.

The superposition of additional behaviour concerning the monitoring of how many days the account has been overdrawn gave rise to the design *monregaccount*. In that design, withdrawals are refused once the number of days the account has been overdrawn reaches a certain limit. Clearly, this

policy on withdrawals does not comply with the limit a).

We can make this observation more precise by providing a formal mapping between CommUnity designs and specifications in branching temporal logic. Indeed, we can assign the following set of sentences with every action g, which capture the (informal) semantics of actions that we briefly discussed and can be found in [17]:

- $g \Rightarrow L(g)$, i.e. actions can only take place when the lower bound of the enabling condition is true.
- $U(g) \Rightarrow \mathbf{E}g$, i.e. actions can take place when the upper bound of the enabling condition is true — $\mathbf{E}$ is the branching-time operator that ensures the existence of a branch in which the condition holds.
- $g \Rightarrow \underline{R(g)}$ where $\underline{R(g)}$ is the condition obtained from $R(g)$ by replacing every occurrence v' of a variable v with the term $\mathbf{X}v$ — $\mathbf{X}$ is the linear-time operator that refers to what happens in the next state.

Whereas the conditions imposed on composition morphisms ensure that the first and third classes of properties are preserved, the second set is not. These are, precisely, properties of required non-determinism, i.e properties that require the system to be available,in certain states, for accepting the execution of certain actions.

This shows that, in the context of CommUnity, superposition is not a principle for design refinement. Given that composition morphisms were motivated as capturing the relationship that exists between systems and their components, this is hardly surprising. It is well known in languages such as CSP [12]that the parallel composition of a collection of processes does not refine, necessarily, any of the constituents. In fact, we can even prove that composition morphisms, instead of preserving, reflect non-determinism, meaning that any form of non-determinism detected in the target can be traced back to the source design [15]. A notion of morphism that captures refinement of CommUnity designs is the following:

Definition 4.1 A refinement morphism of designs $\sigma : P_1 \to P_2$ consists of a total function $\sigma_{var} : V_1 \to V_2$ and a partial mapping $\sigma_{ac} : \Gamma_2 \to \Gamma_1$ s.t.:

1. for every $v \in V_1, i \in in(V_1), o \in out(V_1), x \in prv(V_1)$
   $sort_2(\sigma_{var}(v)) = sort_1(v)$
   $\sigma_{var}(o) \in out(V_2)$
   $\sigma_{var}(i) \in in(V_2)$
   $\sigma_{var}(x) \in prv(V_2)$
   $\sigma_{var} \downarrow (out(V_1) \cup in(V_1))$ is injective
2. for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined
   if $g \in sh(\Gamma_2)$ then $\sigma_{ac}(g) \in sh(\Gamma_2)$
   if $g \in prv(\Gamma_2)$ then $\sigma_{ac}(g) \in prv(\Gamma_2)$
   if $g \in sh(\Gamma_1)$ then $\sigma_{ac}^{-1}(g) \neq \emptyset$
3. for every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined

$\sigma_{var}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$
$\sigma_{ac}(D_2(\sigma_{var}(v))) \subseteq D_1(v)$ for every $v \in loc(V_1)$
$\Phi \models (R_2(g) \Rightarrow \sigma(R_1(\sigma_{ac}(g))))$
$\Phi \models (L_2(g) \Rightarrow \sigma(L_1(\sigma_{ac}(g))))$

4. for every $g_1 \in \Gamma_1$
$\Phi \models (\sigma(U_1(g_1))) \Rightarrow \vee_{\sigma_{ac}(g_2)=g_1} U_2(g_2))$

Designs and their refinement morphisms define a category **r-DSGN**.

A refinement morphism is intended to support the identification of a way in which a design P_1 (its source) is refined by a more concrete design P_2 (its target).

The function σ_{var} identifies for each input (respectively output) variable of P_1 the corresponding input (respectively output) variable of P_2. Notice that, contrarily to what happens with the composition relationship, refinement does not change the border between the system and its environment and, hence, input variables can no longer be mapped to output variables.

The mapping σ_{ac} identifies for each action g of P_1, the set of actions of P_2 that implements g — given by $\sigma_{ac}^{-1}(g)$. This set is a menu of refinements for action g and can be empty for private actions. However, every action that models interaction between the component and its environment has to be implemented.

The actions for which σ_{ac} is left undefined (the new actions) and the variables that are not involved $\sigma_{var}(V_1)$ (the new variables) introduce more detail in the description of the component.

The two conditions on write frames require that the new actions do not modify the variables of the more abstract design.

The last condition in 3 and condition 4 require that the interval defined by the blocking and progress conditions of each action, in which the enabling condition of any guarded command that implements the action must lie, be preserved or reduced. This is intuitive because refinement, pointing in the direction of implementations, should reduce underspecification. 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. It is easy to see that the properties that we expressed above in temporal logic are preserved by refinement morphisms.

In order to illustrate refinement consider the design $refaccount$. Despite the similarities between designs $refaccount$ and $monregaccount$, $refaccount$ is in fact a refinement of $account$. This is because the policy on withdrawals adopted in $refaccount$ complies with the limits established in the design $account$. The enabling condition of action wit, which is now completely defined, lies in the interval established in the design $account$. Withdrawals are specified to be accepted if and only if either the balance is enough or the balance is enough if a certain credit amount is provided and the number of days the account has been overdrawn does not exceed a certain limit. Notice that

in this refinement step it was also taken the design decision of charging a 10% penalty when the credit facility is used.

It is important to notice that neither *monregaccount* is a refinement of account nor *refaccount* could be obtained through regulative superposition o v erthe design *account*. However, it is easy to see that there are regulative superposition morphisms that are also refinement morphisms. In particular, this is the case of the subclass of regulative superposition morphisms that require that the in terval of the enabling condition of actions be preserved. Such morphisms model, to some extent, spectative superposition as in [8], the kind of superposition also used in Unity .

```
design refaccount is
in  am:nat, day:nat
out count:nat, bal:int
prv d:nat
do  dep: true→ bal:=v+bal ‖ d:=day ‖
                        count:=if(bal<0,count+(day-d),count)
 []  wit: (bal+CRE≥am ∧ count<LIM) ∨ bal≥am →
                 bal:=if(bal≥am,bal-am,(bal-am)*1.1) ‖ d:=day ‖
                 count:=if(bal<0,count+(day-d),count)
[]   reset: true, false → d:=day ‖ count:=0
```

5 Compositionality

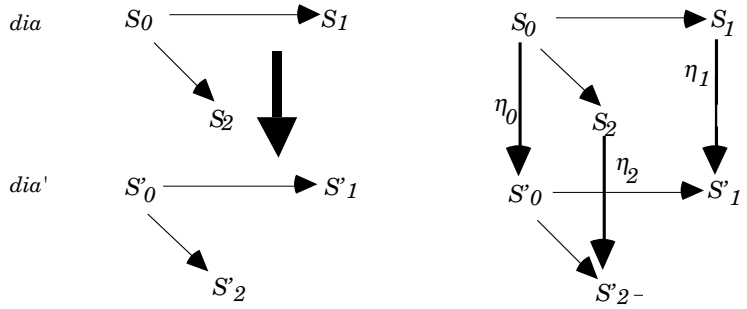
Refinement and composition morphisms, though different as justified above, can be related by a compositionality property . Compositionality is a key issue in the design of complex systems because it makes it possible to reason about a system using the descriptions of their components at any level of abstraction, without having to know how these descriptions are refined in the lower lev els (which includes their implementation).

When component interconnections are explicitly modelled through configurations, as is the case of CommUnity , compositionality can be formulated as the property according to which we can pick arbitrary refinements of the components of a system Sys, and in terconnect these more concrete descriptions with arbitrary refinements of the connections used in Sys, and obtain a system that still refines Sys. The notion of refinement can be extended to configuration diagrams in a straightforward manner. Consider the follo wing configuration diagrams dia and dia'.

Given that the refinement of the components is defined b y the refinement morphisms $\eta_1 : S_1 \to S_1'$ and $\eta_2 : S_2 \to S_2'$, the diagram dia' is a refinement of dia if there exists a refinement morphisms $\eta_0 : S_0 \to S_0'$ that makes the two squares, at the level of signatures, commute.

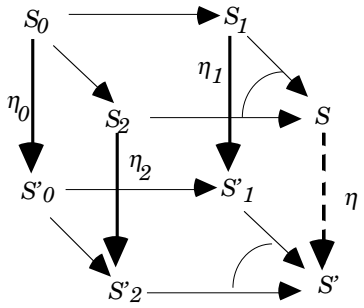
In this wa y ,in tutively, we are requiring that the system architecture, seen as a collection of components and "cables" , cannot change during a refinement step. Howev er, the "cables" used to in terconnect the more abstract designs can be replaced b y "cables" with more capabilities. Howev er, the interconnection dia' must be consistent with dia, i.e., the i/o communications and

synchronisations defined by *dia* have to be preserved.



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

- The diagram dia' cannot establish the instantiation of any input variable that was left "unpluged" in *dia*. That is to say, the input variables of the composition are preserved by refinement.
- The diagram dia' cannot establish the synchronisation of actions that were defined as being independent in *dia*.



In these situations, it is ensured that there exists a refinement morphism $\eta : S \to S'$. Furthermore, this morphism is unique if we require the preservation of the design decisions that lead from each S_i to S'_i. The proof of this result can be found in [14].

6 Conclusions

In this paper, we have shown how the concept of superposition, as used in [6,8,5,13,3], can subsume both composition and refinement of action-based designs. By adopting a categorical formalisation of designs for the language CommUnity, it becomes clear that composition and refinement are two ways of constructing complex designs from simpler ones that that are different because they are required to preserve different properties.

The crucial distinction between these two views is related to the separation

that exists between non-determinism and underspecification. This separation of concerns is too often ignored or overlooked by formal methods but is crucial when architectural concerns are at stake. Let us consider, for instance, the categorical semantics of architectural connectors [9]. An architectural connector type, a notion proposed by [1] for supporting the design of interactions between components, is defined by a set of *roles*, that can be instantiated with specific components of the system under construction, and a *glue* specification that describes how the activities of the role instances are to be coordinated. Given that the roles of a connector are abstractions of the components that can use the communication protocol modelled by the connector, it is not difficult to realise that the compatibility requirement that role instantiation has to obey addresses the preservation of a class of properties which are not the same that are required to be preserved during the superposition of the functionalities specified by the glue over the involved components, so these components engage in the given communication protocol.

Furtherwork is going on which addresses the integration of a distribution and mobility dimension in CommUnity and the use of superposition in order to support the externalisation of the mechanisms that are responsible for managing the distribution topology of systems [16]. This work has provided more evidence for the need of distinguishing between composition and refinement.

References

[1] Allen, R., and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM*, 6(3):213-249,1997.

[2] Bougé, L., and N.Francez, "A compositional approach to superposition", *Proc. of 14th ACM POPL*, pages 240-249, 1988.

[3] Bosch, J., "Superimposition – A Component Adaptation Technique", *Information and Software Technolgy* 1999.

[4] Back, R.J., "Refinement calculus II: Parallel and reactive programs", in J. deBakker,W. deRoever and G.Rozenberg (eds), *Stepwise Refinement of Distributed Systems*, LNCS 430, pages 67-93, Springer-Verlag,1990.

[5] Back, R.J., and K.Sere, "Superposition Refinement of Reactive Systems", *Formal Aspects of Computing*, 8(3):324-346, 1996.

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

[7] Dijkstra, E., *A Discipline of Programming* Prentice-Hall International 1976.

[8] Francez, F., and I.Forman, *Interacting Processes*, Addison-Wesley 1996.

[9] Fiadeiro, J.L., and A.Lopes, "Semantics of Architectural Connectors", *in Proceedings of TAPSOFT'97*, LNCS 1214, pages 505-519, Springer-Verlag, 1997.

[10] Fiadeiro, J.L., and T.Maibaum, "Categorical Semantics of Parallel Program Design", *Science of Computer Programming*, 28:111-138,1997.

[11] Goguen, J., "Categorical Foundations for General Systems Theory", in F.Pichler and R.Trappl (eds), *Advanes in Cybernetics and Systems Research*, Transcripta Books 1973,121-130.

[12] Hoare, C.A.R., *Communicating Sequential Processes* Prentice-Hall International 1985.

[13] Katz, K., "A Superimposition Control Construct for Distributed Systems", *ACM TOPLAS* 15(2):337-356, 1993.

[14] Lopes, A., "Non-determinism and compositionality in the specification of reactive systems", PhD thesis, Universidade de Lisboa, January 1999.

[15] Lopes, A.,and J.L.Fiadeiro, "Preservation and Reflection in Specification", in M.Johnson (ed), *Proceedings of AMAST'97*, LNCS 1349, pages 380- 394, Springer-Verlag, 1997.

[16] Lopes, A., J.L.Fiadeiro and M.Wermelinger, "Architectural Primitives for Distribution and Mobility", *10th Symposium on Foundations of Software Engineering*, in print.

[17] Lopes, A., J.L.Fiadeiro, "Using explicit state do describe architectures", *Proceedingsof Fundamental Approaches to Software Engineering*, LNCS 1577, pages 144-160, Springer-Verlag, 1999.

[18] Shaw, M. and D.Garlan, *Software Architecture Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[19] Wermelinger, M. and J.L.Fiadeiro, "Connectors for Mobile Programs", *IEEE Transactionson Software Engineering* 24(5), 1998, 331-341.