DOI 10.1007/s00165-003-0021-6
BCS © 2004
Formal Aspects of Computing (2004) 16: 5–18

Formal Aspects of Computing

Superposition: composition vs refinement of non-deterministic, action-based systems

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

[1]Department of Informatics, Faculty of Sciences, University of Lisbon, Lisbon, Portugal
[2]Department of Mathematics and Computer Science, University of Leicester, Leicester, UK

Abstract. The traditional notion of superposition has been used for supporting two distinct aspects of parallel program design: composition and refinement. This is because, when trace-based semantics of concurrency are considered, which is typical of most formal methods, these two relationships are modelled as inclusion between sets of behaviours. However, when forms of non-deterministic behaviour have to be considered, which is the case for component and service-based development, these two aspects do not coincide. In this paper, we show how the two roles of superposition can be separated and supported at the language and semantic levels. For this purpose, we use a categorical formalisation of program design in the language CommUnity that we are also using for addressing architectural concerns, another area in which the distinction between composition and refinement is particularly important.

Keywords: Composition, Refinement, Superposition

1. Introduction

Superposition (or superimposition) has been proposed (and used) as a powerful mechanism for structuring the development of parallel programs and distributed systems [ChM88, FrF96, BaS96, Kat93, Bos99, BoF88]. 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 [Dij76]: 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' (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.

Correspondence and offprint requests to: Antónia Lopes, Department of Informatics, Faculty of Sciences, University of Lisbon, Campo Grande, 1749-016 Lisbon, Portugal.

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' [FrF96]. 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 [FiM97], 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 the failure or ready semantics of CSP [Hoa85], parallel composition does not induce refinement (P||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||Q').

The notion of superposition as composition 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 CommUnity [FiM97].

2. Component design in CommUnity

CommUnity is a parallel program design language in the style of Unity that was initially proposed in [FiM97] to show how programs fit into Goguen's categorical approach to General Systems Theory [Gog73]. 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 [FiL97, WeF98, LoF99]. 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 predefined collection of data types given in the form of a first-order algebraic specification, i.e. a data signature $< S, \Omega >$, 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 [Bac90] could be adopted to support also data refinement.

A CommUnity component design is of the form:

```
design P  is
in              in(V)
out             out(V)
prv             prv(V)
do     []       g[D(g)]: L(g), U(g) → R(g)
     g∈sh(Γ)
       []       prv g[D(g)]: L(g), U(g) → R(g)
     g∈prv(Γ)
```

V is the set of *channels* of design P. There are *input* channels, *output* channel*s* and *private* channel*s*. Input channels are read-only: the component has no control on the values that they hold. Output and private channels are controlled locally by the component, i.e. the value that is made available on these channels cannot be modified by the environment. Output channels can be read by the environment but private ones cannot. We use $loc(V)$ to denote the union $prv(V) \cup out(V)$ of local ones. Each channel 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 [FrF96], provide points of *rendez-vous* at which components can synchronise. For each action name g :

- $D(g)$ consists of the local channels 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 channel 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 $U(g) \supset L(g)$. 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 channels from the write frame of g. As usual, these primed channels account for references to the values that the channels exhibit 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 $pre \supset pos$, 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 $skip$.

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 channels 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 bank account with a credit facility.

```
design  account  is
in      am: nat
out     bal: int
do      dep[bal]: true → bal′=am+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 affect 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 $am - CRE < bal < am$, 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.

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 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 to channels 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 [LoF99] for a model-theoretic semantics of CommUnity.

As an example of a program consider the following design:

```
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
```

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 channels *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 (channel *d*).

3. Superposition

The design of a bank 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 mon-reg-account  is
in     am: nat, day: nat
out    count: nat, bal: int
prv    d: nat
do     dep[bal,d,count]: true → bal′ = am + 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 *mon-reg-account* we have introduced the new input channel *day* and the new local channels *count* and *d* (superposed channels) and the new action *reset*. The input channel *day* gives the current day. The channel *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 channel *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 [FiM97] 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 channels be preserved and allows for the enabling condition of its actions to be strengthened.

Definition 1. A composition morphism of designs σ: $P_1 \rightarrow P_2$ consists of a total function σ_{ch}: $V_1 \rightarrow V_2$ and a partial mapping σ_{ac}: $\Gamma_2 \rightarrow \Gamma_1$ s.t.:

1. For every $v \in V_1$, $\mathrm{sort}_2(\sigma_{ch}(v)) = \mathrm{sort}_1(v)$; for every $o \in out(V_1)$, $\sigma_{ch}(o) \in out(V_2)$; for every $i \in in(V_1)$, $\sigma_{ch}(i) \in out(V_2) \cup in(V_2)$; for every $x \in prv(V_1)$, $\sigma_{ch}(x) \in prv(V_2)$.
2. For every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined: if $g \in sh(\Gamma_2)$ then $\sigma_{ac}(g) \in sh(\Gamma_1)$; if $g \in prv(\Gamma_2)$ then $\sigma_{ac}(g) \in prv(\Gamma_1)$.
3. For every $v \in loc(V_1)$: σ_{ac} is total on $D_2(\sigma_{ch}(v))$ and $\sigma_{ac}(D_2(\sigma_{ch}(v))) \subseteq D_1(v)$.
4. For every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined: $\sigma_{ch}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$; $\Phi \vdash (R_2(g) \supset \underline{\sigma}(R_1(\sigma_{ac}(g))))$; $\Phi \vdash (L_2(g) \supset \underline{\sigma}(L_1(\sigma_{ac}(g))))$; $\Phi \vdash (U_2(g) \supset \underline{\sigma}(U_1(\sigma_{ac}(g))))$.

where $\underline{\sigma}$ is the extension of σ to the language of expressions and conditions. Designs and composition morphisms define a category **c-DSGN**. We denote by **SIGN** the category of design signatures (constituted by the channels, actions and associated types and domains) and the signature morphisms underlying the morphisms of **c-DSGN**.

A composition morphism $\sigma : P_1 \rightarrow P_2$ 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_{ch} identifies for every channel of the component the corresponding channel of the system. The first group of constraints establishes that sorts of channels have to be preserved. Notice, however, that input channels of a component can become output channels of the system. This is because the result of interconnecting an input channel of a component with an output channel of another component in the system is an output channel 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 two groups 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 three conditions on write frames imply that actions of the system in which a component is not involved cannot have local channels of the component in their write frame. In the last group, the second 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 *mon-reg-account* can be regarded as the result of applying superposition to the design *account* is $\sigma : account \rightarrow mon\text{-}reg\text{-}account$ where

σ_{ch} is the inclusion function

$\sigma_{ac}(dep) = dep$, $\sigma_{ac}(wit) = wit$ and $\sigma_{ac}(reset)$ is undefined (because *reset* 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 in Fig. 1 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. Notice that the other designs involved in the diagram – *cable1, cable2* and *cable3* – are essentially a set of input channels and a set of shared actions. These kinds of designs are called cables. It defines:

- An i/o interconnection between channels *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 latter 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*.

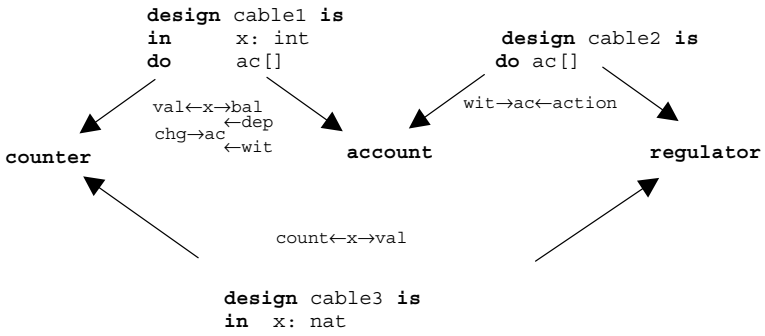

Fig. 1. An example of a configuration diagram

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 channels *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.

Notice that we are using the notion of diagram in a technical, mathematical, sense: a graph whose nodes are labelled with objects (in this case, CommUnity designs) and the edges with morphisms (in this case composition morphisms) of a given category. It is often convenient to adopt the same mathematical language in the semantics of a development approach and we will do so by looking at graphs as categorical constructions themselves: the graph over which the diagram is built is no more than a very simple category and the labelling of nodes and edges is no more than what is known as a functor. Hence, a diagram over the category **c-DSGN** is no more than a functor $\mathbf{I} \rightarrow$ **c-DSGN** where **I** is the underlying graph (the 'shape' of the diagram).

The fact that we can handle diagrams as mathematical objects is very convenient. 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:

- channels involved in each i/o-communication established by the configuration are amalgamated;
- every set $\{g_1, \ldots, g_n\}$ of actions that are synchronised through the interconnections is represented by a single action $g_1 || \ldots || g_n$ 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 *mon-reg-account* 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.

As happens in most of the languages and formalisms that are used for the configuration of software systems (and also hardware and mechanical systems), the interconnection of components in CommUnity is governed by specific rules. In CommUnity, not every diagram in the category **c-DSGN** expresses a meaningful configuration in the sense that it describes a well-configured system. For instance, diagrams in which output channels of different components are connected to one another do not make sense as configurations.

Consider that **Ch** denotes the forgetful functor from **c-DSGN** to **SET** that maps designs to their underlying sets of channels. The class of diagrams that represent correct configurations of interconnected components can be formalised as follows.

Definition 2. Given a finite set J and a J-indexed multi-set of designs $\mathcal{D}$, a configuration diagram of a system with those components is a finite diagram **dia**:$\mathbf{I} \rightarrow$ **c-DSGN** s.t.:

1. For every $j \in J$, $j \in |\mathbf{I}|$ and $\mathbf{dia}(j) = \mathcal{D}_j$.
2. For every $f : i \to j$ in $\mathbf{I}$, either ($i = j$ and $f = id_i$) or ($j \in J$ and $i \notin J$ and $\mathbf{dia}(i)$ is a cable).
3. For every $i \in |\mathbf{I}| \backslash J$ s.t. $\mathbf{dia}(i)$ is a cable, there exist distinct j, k and morphisms $f : i \to j$ and $g : i \to k$ in $\mathbf{I}$.
4. If $\{\mu_i : \mathbf{Ch}(\mathbf{dia}(i)) \to V : i \in |\mathbf{I}|\}$ is a colimit of $\mathbf{dia};\mathbf{Ch}$ then, for every $v \in V$, there exists at most one $i \in |I|$ such that $\mu_i^{-1}(v) \cap out(V_{\mathbf{dia}(i)}) \neq \emptyset$ and, for such i, $\mu_i^{-1}(v) \cap out(V_{\mathbf{dia}(i)})$ is a singleton.

Condition 1 means that every component of a system must be involved in its configuration diagram. Condition 2 states that the elementary interconnections are established through cables. Condition 3 ensures that a configuration diagram does not include cables that are not used. Finally, condition 4 prevents the identification of output channels.

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 *mon-reg-account*. 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 an action logic whose grammar is given by

$$\phi ::= a | (\neg\phi) | (\phi \supset \phi') | [\gamma]\phi$$

where a denotes a first-order sentence in the language of design channels and γ is a set of action names. Each (modal) operator $[\gamma]$ is such that the formula $[\gamma]\phi$ expresses that, for every action in γ, ϕ holds after the action occurs. (The full syntax and semantics of this logic can be found in [LoF97].) The dual of $[\gamma]$ is the modal operator $< \gamma >$ defined by the abbreviation $< \gamma > \phi \equiv_{\text{abv}} (\neg[\gamma](\neg\phi))$. The formula $< \gamma > \phi$ expresses that there exists an action in γ that establishes ϕ. We also adopt the abbreviation of a singleton by its element, that is, $[g]\phi \equiv_{\text{abv}} [g]\phi$ and $< g > \phi \equiv_{\text{abv}} < g > \phi$. Notice, in particular, that $<g>true$ expresses that g is enabled.

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 [LoF97, Lop99]:

- $< g > true \supset L(g)$, i.e. actions can only take place when the lower bound of the enabling condition is true.
- $U(g) \supset < g > true$, i.e. actions are ready to take place when the upper bound of the enabling condition is true.
- $\underline{R(g)}$ where $\underline{R(g)}$ is the condition obtained from $R(g)$ by replacing every occurrence v' of a design channel v with the term $[g]v$ that denotes the value of the term after the occurrence of the action.

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 [Hoa85] 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. More precisely, if we consider the language of state properties (i.e. properties of design channels):

$$\varphi ::= a | (\neg\varphi) | (\varphi \supset \varphi')$$

where a denotes an atomic formula built over the set of channels and the operations of the underlying data type signature, the set of*properties* defined by

$$\phi ::= \varphi | (\varphi \supset [\gamma]\varphi')$$

are preserved by composition morphisms. These properties capture invariants (φ), effects of actions ($\phi \supset [g]\varphi'$), and restrictions to the occurrence of actions ($< g > \textbf{\textit{true}} \supset \varphi$).

The set of *co-properties* defined by

$$\psi ::= \varphi | (\varphi \supset < g > \textbf{\textit{true}})$$

are reflected by composition morphisms in the sense that, if they hold in a system, it is because they were already true in the component. Co-properties capture the ability of actions to occur in certain states – readiness ($\varphi \supset < g > \textbf{\textit{true}}$) – and also state propositions (φ). That is, state propositions can be used both as properties, guaranteeing that they will hold in any context, and as co-properties, expressing the fact that the component will reflect them. Again, refer to [LoF97] for full details.

A notion of morphism that captures refinement of CommUnity designs is the following:

Definition 3. A refinement morphism of designs $\sigma : P_1 \to P_2$ consists of a total function $\sigma_{ch} : V_1 \to V_2$ and a partial mapping $\sigma_{ac} : \Gamma_2 \to \Gamma_1$ s.t. :

1. For every $v \in V_1$, $\text{sort}_2(\sigma_{ch}(v)) = \text{sort}_1(v)$, for every $o \in out(V_1)$, $\sigma_{ch}(o) \in out(V_2)$; for every $i \in in(V_1)$, $\sigma_{ch}(i) \in in(V_2)$; for every $x \in prv(V_1)$, $\sigma_{ch}(x) \in prv(V_2)$; $\sigma_{ch} \downarrow (out(V_1) \cup in(V_1))$ is injective.
2. For every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined: if $g \in sh(\Gamma_2)$ then $\sigma_{ac}(g) \in sh(\Gamma_1)$; if $g \in prv(\Gamma_1)$ then $\sigma_{ac}(g) \in prv(\Gamma_1)$; if $g \in sh(\Gamma_2)$ then $\sigma_{ac}^{-1}(g) \neq \emptyset$.
3. For every $v \in loc(V_1)$: σ_{ac} is total on $D_2(\sigma_{ch}(v))$ and $\sigma_{ac}(D_2(\sigma_{ch}(v))) \subseteq D_1(v)$.
4. For every $g \in \Gamma_2$ s.t. $\sigma_{ac}(g)$ is defined: $\sigma_{ch}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$; $\Phi \vdash (R_2(g) \supset \underline{\sigma}(R_1(\sigma_{ac}(g))))$; $\Phi \vdash (L_2(g) \supset \underline{\sigma}(L_1(\sigma_{ac}(g))))$.
5. For every $g_1 \in \Gamma_1 : \Phi \vdash (\underline{\sigma}(U_1(g_1)) \supset \bigvee_{\sigma_{ac}(g_2)=(g_1)} U_2(g_2)$

Designs and their refinement morphisms constitute 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_{ch} identifies for each input (respectively output) channel of P_1 the corresponding input (respectively output) channel of P_2. Notice that, contrary to what happens with the composition relationship, refinement does not change the border between the system and its environment and, hence, input channels can no longer be mapped to output channels.

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 channels that are not involved in $\sigma_{ch}(V_1)$ (the new channels) introduce more detail into the description of the component.

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

The last condition in 4 and condition 5 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 following design.

```
design ref-account  is
in       am: nat, day: nat
out      count: nat, bal: int
prv      d: nat
do       dep: true → bal := am + bal ‖ d := day ‖
                       count := if (bal<0, count + (day - d), count)
```

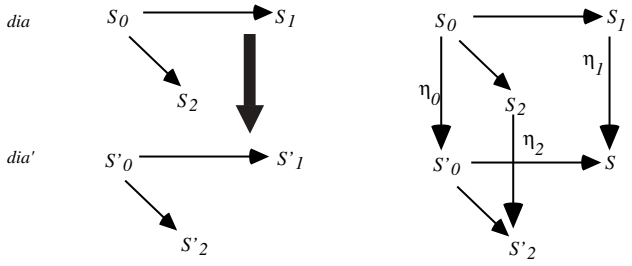

Fig. 2. Refinement of configuration diagrams

```
[]      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
```

Despite the similarities between designs *ref-account* and *mon-reg-account, ref-account* is in fact a refinement of *account*. This is because the policy on withdrawals adopted in *ref-account* 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 the design decision was also taken of charging a 10% penalty when the credit facility is used.

It is important to notice that neither *mon-reg-account* is a refinement of *account* nor *ref-account* could be obtained through regulative superposition over the 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 interval of the enabling condition of actions be preserved. Such morphisms model, to some extent, *spectative* superposition as in [FrF96], the kind of superposition also used in Unity.

In what concerns the logical properties of designs, it is easy to see that refinement morphisms preserve *both* properties and co-properties, namely those of the form $(U(g) \supset\!< g >\! true)$. That is to say, refinement morphisms translate to specification morphisms as they are normally understood in specification theory – theorem preserving mappings.

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 levels (which includes their implementation).

When component interconnections are explicitly modelled through configurations, as is the case for CommUnity, 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*.

The notion of refinement can be extended to configuration diagrams in a straightforward manner. Consider the configuration diagrams ***dia*** and ***dia'*** depicted in Fig. 2.

Given that the refinement of the components is defined by 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 morphism $\eta_0 : S_0 \to S'_0$ that makes the two squares, at the level of signatures, commute.

In this way, intuitively, we are requiring that the system architecture, seen as a collection of components and 'cables', cannot change during a refinement step. Note that the 'cables' used to interconnect the more abstract designs can be replaced by 'cables' with more capabilities. However, the interconnection ***dia'*** must be consistent with ***dia*** that is, the i/o communications and synchronisations defined by ***dia*** have to be preserved.

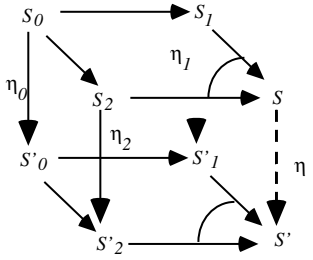

Fig. 3. Compositionality ensures the existence of the refinement morphism η

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 channel that was left 'unplugged' 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***.

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. This result can be formulated and proved as follows (see Fig. 3).

Theorem 1. Given ***dia***: $\boldsymbol{I} \to$ ***c-DSGN*** and ***dia'***: $\boldsymbol{I} \to$ ***c-DSGN***, two well-formed configuration diagrams of systems with a J-indexed multi-set of components, and an $|\mathbf{I}|$-indexed family $(\eta_i : dia(i) \to dia'(i))_{i \in |I|}$ of morphisms in ***r-DSGN*** s.t., for every morphism $f : i \to j$ in $\boldsymbol{I}$:

1. ***Sig***(***dia***(f));***r-Sig***(η_j) = ***r-Sig***(η_i);***Sig***(***dia'***(f)), where ***Sig*** and ***r-Sig*** denote the forgetful functors, respectively, from ***c-DSGN*** to ***SIGN*** and ***r-DSGN*** to ***SIGN***
2. For every $v' \in in(V'_i)$, if $v' \notin \eta_i(V_i)$ then ***dia'***(f)(v')$\notin \eta_j(inp(V_j))$, assuming that V_i and V'_i are the set of channels of, respectively, ***dia***(i) and ***dia'***(i).
3. For every $g' \in \Gamma'_j$, if $\eta_j(g) \downarrow$ and ***dia'***$(f)(g') \downarrow$ then $\eta_i(dia'(f)(g')) \downarrow$, assuming that Γ_i and Γ'_i are the set of channels of, respectively, ***dia***(i) and ***dia'***(i).
4. For every $i \in |\boldsymbol{I}|J$, $\eta_{i_{ac}}$ is injective.

there exists a unique morphism $\eta : S \to S'$ in ***r-DESC*** s.t. ***Sig***(μ_i);***r-Sig***$(\eta) =$ ***r-Sig***(η_i); ***Sig***(μ'_i), for every $i \in |\boldsymbol{I}|$, where $(\mu_i : \mathbf{dia}(i) \to S)_{i \in |I|}$ and $(\mu_i : dia'(i) \to S')i \in |\boldsymbol{I}'|$ are colimits of, respectively, ***dia*** and ***dia'***.

Proof. From condition 1 it follows that (***r-Sig***(η_i); ***Sig***$(\mu'_i))_{i \in |\boldsymbol{I}|}$ is a candidate for being a colimit of ***dia***;***Sig*** in ***SIGN***. Because ***Sig*** preserves colimits, there exists a unique morphism η : ***Sig***$(S) \to$ ***Sig***(S') in ***SIGN*** s.t. ***Sig***(μ_i);***r-Sig***$(\eta) =$ ***r-Sig***(η_i);***Sig***(μ'_i), for every $i \in |\boldsymbol{I}|$. It remains to prove that such η also defines a morphism from S to S' in ***r-DSGN***. We shall only prove that η satisfies the conditions of refinement morphisms that do not necessarily hold for composition morphisms.
(i) η maps input channels into input channels:

We start by pointing out that the forgetful functor from ***SIGN*** to ***SET*** that forgets everything but channels preserves colimits. Let v be an input channel of S. By the colimit construction in ***SIGN***, for every $i \in |I|$ and $v_i \in \mu_i^{-1}(v)$, v_i is an input channel of ***dia***(i). Because refinement morphisms map input channels into input channels, we have that,

for every $i \in |\boldsymbol{I}|$ and $v_i \in \mu_i^{-1}(v)$, $\eta_i(v_i)$ is an input channel of ***dia'***(i) $(*)$

In order to conclude that $\eta(v)$ is an input channel of S', it is necessary to prove that for every $i \in |\boldsymbol{I}|$ and $v'_i \in \mu_i'^{-1}(\eta(v))$, v'_i is an input channel of ***dia'***(i). Assume that there exists $i \in |\boldsymbol{I}|$ and $v'_i \in \mu_i'^{-1}(\eta(v))$ such that v'_i is an output channel. Because $\eta_{i_{ch}}$ is injective and (*) holds, $v'_i \notin \eta_i(V_i)$. Because ***dia'*** is a well-founded diagram, the way colimits in ***SIGN*** work ensures that, in ***dia'***, there are a cable and morphisms establishing the i/o interconnection of v'_i with one of the input channels that is mapped into v. That is to say, there exists $k \in |\boldsymbol{I}|$ s.t. ***dia'***(k) is a cable that has an input channel x and there exist $m \in |\boldsymbol{I}|$ s.t. $\mu_m^{-1}(v)$ is not empty and morphisms

$f : k \to i$ and $g : k \to m$ in $\boldsymbol{I}$ s.t. $dia'(f)(x) = v'_i$ and $\boldsymbol{dia}'(g)(x) \in \eta_m(\mu_m^{-1}(v))$. But, then, $x \notin \eta_k(V_k)$, otherwise it would be impossible that $v'_i \notin \eta_i(V_i)$ (by condition 1). Then, by condition 2, $\boldsymbol{dia}'(g)(x) \notin \eta_m(inp(V_m))$ would have to hold, which is a contradiction given the way m was chosen.

(ii) Every shared action g of S is implemented in S', i.e., $\eta_{ac}^{-1}(g) \neq \emptyset$.

We start by pointing out that the forgetful functor from ***SIGN*** to $\boldsymbol{PFN}^{\boldsymbol{op}}$ that forgets everything but actions preserves colimits. Let g be a shared action of S. By the colimit construction, there exists $i \in |\boldsymbol{I}|$ s.t. $\mu_i(g) \downarrow$. Furthermore, $\mu_i(g)$ is a shared action. Because η_i is a refinement morphism, there exists a shared action g'_i in ***dia***'(i) s.t. $\eta_i(g'_i) = g_i$. In order to conclude that $\eta_{ac}^{-1}(g) \neq \emptyset$ it is necessary to prove that there exists an action g' in S' s.t. $\eta_{ac}(g') = g$ and $\mu'_i(g') = g'_i$. We shall prove a stronger result that will be useful later on.

Proposition. Let g be an action of S and let I_g be $\{i \in |\boldsymbol{I}| : \mu_i(g) \downarrow\}$. For every I_g-indexed set of actions $G = \{g'_i \in \Gamma'_i\}$ s.t. $\eta_i(g') = \mu_i(g'_i)$, and every g'_i in G, there exists $g' \in \Gamma'$ s.t., $\mu'_i(g') = g'_i$, for every $i \in I_g$, $\mu'_i(g') \uparrow$, for every $i \notin I_g$ and $\eta(g') = g$.

Proof. By the limit construction in ***PFN***, we have that for every $i, j \in I_g$ and morphism $f : i \to j$ in $\boldsymbol{I}$, ***dia***(f)$(\mu_j(g)) = \mu_i(g)$. By condition 1 of the theorem, $\eta_i(dia'(f)(g'_j)) = dia(f)(\eta_j(g'_j)) = dia(f)(\mu_j(g)) = \mu_i(g) = \eta_i(g'_i)$. By condition 4 of the theorem, and given that in well-formed configurations the sources of morphisms that are not identities are cables, we have that $\eta_{i_{ac}}$ is injective. Therefore, $\boldsymbol{dia}'(f)(g'_j) = g'_i$. Given that this holds for every $i, j \in I_g$ and morphism $f : i \to j$ in $\boldsymbol{I}$, the result follows immediatly.

(iii) For every $g \in \Gamma$, $\Phi \vdash (\underline{\eta}(U(g)) \supset \bigvee_{\eta_{ac}(g')=(g)} U(g'))$.

By the colimit construction in ***c-DSGN***, we have that

$$U(g) = \bigwedge_{i \in |\boldsymbol{I}| \text{ s.t. } \mu_i(g)\downarrow} \underline{\mu_i}(U_i(\mu_i(g))).$$

Let I_g be the set $\{i \in |I| : \mu_i(g) \downarrow\}$. For every $i \in I_g$, because η_i is a refinement morphism,

$$\Phi \vdash \underline{\eta_i}(U_i(\mu_i(g))) \supset \bigvee_{\eta_i(g'_i)=\mu_i(g)} U'_i(g'_i).$$

For every $i \in I_g$, let g'_i be s.t. $\eta_i(g'_i) = \mu_i(g)$ and $\Phi \vdash \underline{\eta i}(U_i(\mu_i(g))) \supset U'_i(g'_i)$. Then,

$$\Phi \vdash \underline{\eta}(U(g)) \supset \bigwedge_{i \in I_g} \underline{\mu'_i}(U'_i(g'_i)).$$

From the auxiliary result proved in (ii), it follows that there exists $g' \in \Gamma'$ s.t., for every $i \in I_g$, $\mu'_i(g') = g'_i$, for every $i \notin I_g$, $\mu'_i(g') \uparrow$ and $\eta(g') = g$. We have

$$U'(g') = \bigwedge_{i \in |\boldsymbol{I}| \text{ s.t. } \mu'_i(g)\downarrow} \underline{\mu'_i}(U'_i(\mu'_i(g)))$$

and, therefore,

$$\Phi \vdash \underline{\eta}(U(g)) \supset U'(g').$$

6. Application to software architectures

In the previous sections, we showed that composition and refinement are two ways of constructing complex designs from simpler ones 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.

In order to back this claim, consider the categorical semantics proposed in [FiL97] for the notion of architectural connector proposed by [AlG97] for supporting the design of interactions between components. An architectural connector in this sense 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.

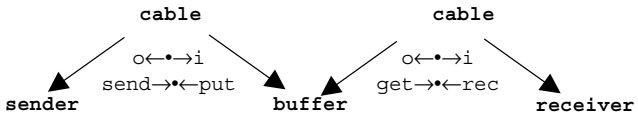

Fig. 4. *Async* connector

For instance, asynchronous communication through a bounded channel can be modelled by a connector *Async* with two roles – *sender* and *receiver* – and a glue that is responsible for establishing the required communication protocol – bounded buffer with FIFO discipline that prevents the *sender* from sending a new message when there is no space and prevents the*receiver* from reading a new message when there are no messages. The design of this glue is given below.

```
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
```

This design is actually a program parameterised by the sort of messages that the buffer can store and its capacity. Through the action *put,* messages of sort *t* received from the environment (the sender) through the input channel *i* can be stored 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*. 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, more precisely, by the receiver. The two roles of this connector, *sender* and *receiver,* are modelled through the following designs:

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

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

The design *sender*[*t*] models a typical sender of messages. In this description, we are primarily concerned with the interaction between the sender and its environment, ignoring details of internal computations such as the production of messages. Notice that a sender cannot produce another message before the previous one has been processed. After producing a message, it expects an acknowledgement (modelled through the execution of *send*) to produce a new message. In order to leave unspecified when and how many messages a *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 was also left completely unspecified: the action *prod* includes the channel *o* in its write frame but the design does not commit to any specific way of changing the value of this channel. For the *receiver,* we simply require that it has an action that models the reception of a message.

The roles *sender* and *receiver* are connected to the glue (buffer) as depicted in Fig. 4.

Notice that we did not give explicit names to the actions and channels of the design *cable* used for the interconnection. These can be represented through the symbol • because the interconnection does not rely on global/implicit naming but, precisely, on associations (bindings) that need to be made explicit as above.

What we have described are connector *types* in the sense that they can be instantiated. More concretely, the roles of a connector type can be instantiated with specific designs. In WRIGHT [AlG97] role instantiation has to obey a compatibility requirement expressed via the refinement relation of CSP. In CommUnity, we use refinement morphisms for instantiation.

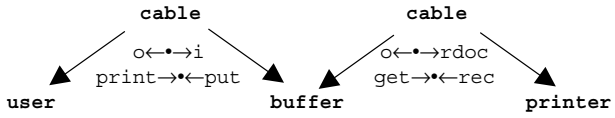

Fig. 5. The result of the instantiation of *Async* connector with designs *printer* and *user*

Indeed, 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 accept 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 and reflect the fact that the components engage in the given communication protocol. For instance, consider the following design of a user that produces files that it stores in a private channel *w* and can then convert them either to PostScript or pdf formats, after which it makes them available (for printing) in an output channel *p*.

```
design   user  is
out      p: ps + pdf
prv      rd, new: bool, w: MSWord
do       work[w,new]: ¬ rd ∧ ¬ new, false → new'
[]       pr_ps: new ∧ ¬ rd, false → p := ps(w)||rd := true
[]       pr_pdf: new ∧ ¬ rd, false → p := pdf(w)||rd := true
[]       print: rd → rd := false||new := false
```

Through the following design, we can model a printer that copies the files it downloads from an input channel *rdoc* into a 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 user can be connected to the printer by instantiating the roles of the architectural connector that we have just defined. Indeed, *sender* is refined by *user* via the refinement morphism $\eta : sender \rightarrow user$ defined by

$\eta_{ch}(o) = p, \eta_{ch}(rd) = rd$

$\eta_{ac}(pr\_ps) = \eta_{ac}(pr\_pdf) = prod, \eta_{ac}(print) = send$

In *user,* the production of messages (to be sent) is modelled by any of the actions pr_ps and pr_pdf and the messages are made available in the output channel *p*. Notice which the production of messages, which 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 format.

Likewise, *printer* is a refinement of *receiver* via the refinement morphism $\kappa : receiver \rightarrow printer$ defined by

$\kappa_{ch}(i) = rdoc, \ \ \kappa_{ac}(rec) = rec$

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

Because both superposition and refinement morphisms coincide on the signatures, they can be composed so that, as a result of the instantiation, the *user* and the *printer* are connected to the *buffer* as depicted in Fig. 5.

Architectural connectors are used for systematising software development by offering 'standard' means for interconnecting components that can be reused from one application to another. In this sense, the 'typical' glue is a program that implements a well-established pattern of behaviour that can be superposed to existing components of a system through the instantiation of the roles of the connector. However, architectures also fulfil an important role in supporting a high-level description of the organisation of a system by identifying its main components and the way these components are interconnected. An early identification of the architectural elements intended for a system will help to manage the subsequent design phases according to the organisation that they imply, identifying opportunities for reuse or the integration of third-party components. From this point of view, it is useful to allow for connectors to be based on glues that are not yet fully developed as programs but for which concrete commitments have already been made to determine the type of interconnection that they will ensure.

For instance, at an early stage of development, one may decide on adopting a client-server architecture without committing to a specific protocol of communication between the client and the server.

In such a more general framework, we have to account for the possible refinements of the glue. The results that we presented in Section 5 under compositionality ensure that if we refine the glue of a connector that has been instantiated to given components of a system, the resulting design is a refinement of the more abstract design from which we started. More generally, it ensures that connectors propagate through design, be it because the instances of the roles are refined or the glue is refined.

7. Conclusions

By adopting a categorical formalisation of designs for the language CommUnity, we have made clear that composition and refinement are two different ways of constructing complex designs from simpler ones: they are required to preserve different properties which, in categorical terms, means that they are supported by different notions of morphism, leading to different categories of designs. This is more than an interesting (if ever) observation because it has an important impact on the way we can support incremental design of software systems as shown through a formalisation of architectural constructs.

On the other hand, the paper also shows that at the core of the distinction between these two concerns (composition and refinement) is the difference between underspecification and non-determinism. Because, in the trace-based semantics that are typically used in formal methods for concurrency, both notions are modelled as inclusion of behaviours, the notion of superposition, the mechanism that supports stepwise design of parallel programs, has been used in [ChM88, FrF96, BaS96, Kat93, Bos99] for expressing both composition and refinement. Our work around CommUnity shows how this separation can be supported at the language and semantic levels of architectural modelling.

Further work is going on which addresses yet another dimension that is becoming prevalent in the age of the Internet and wireless technologies: distribution and mobility. We are currently extending 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 [LFW02]. This work has provided more evidence of the need for distinguishing between composition and refinement.

References

[AlG97] Allen R, Garlan D (1997) A formal basis for architectural connectors. ACM TOSEM 6(3):213–249
[BoF88] Bougé L, Francez N (1988) A compositional approach to superposition. In: Proceedings of the 14th ACM POPL, pp 240–249
[Bos99] Bosch J (1999) Superimposition: a component adaptation technique. Inf Software Technol
[Bac90] Back RJ (1990) Refinement calculus II: parallel and reactive programs. Stepwise Refinement of Distributed Systems, LNCS 430, Springer, Berlin Heidelberg New York, pp 67–93
[BaS96] Back RJ, Sere K (1996) Superposition refinement of reactive systems. Formal Aspects Comput 8(3):324–346
[ChM88] Chandy K, Misra J (1988) Parallel program design: a foundation. Addison-Wesley
[Dij76] Dijkstra E (1976) A discipline of programming. Prentice-Hall International
[FrF96] Francez N, Forman I (1996) Interacting processes. Addison-Wesley
[FiL97] Fiadeiro JL, Lopes A (1997) Semantics of architectural connectors. In: Proceedings of TAPSOFT'97, LNCS 1214. Springer, Berlin Heidelberg New York, pp 505–519
[FiM97] Fiadeiro JL, Maibaum T (1997) Categorical semantics of parallel program design. Sci Comput Programming 28:111–138
[Gog73] Goguen J (1973) Categorical foundations for general systems theory. In: Advances in Cybernetics and Systems Research. Transcripta Books, pp 121–130
[Hoa85] Hoare CAR (1985) Communicating sequential processes. Prentice-Hall International
[Kat93] Katz S (1993) A Superimposition control construct for distributed systems. ACM TOPLAS 15(2):337–356
[Lop99] Lopes A (1999) Non-determinism and compositionality in the specification of reactive systems. PhD Thesis, Universidade de Lisboa
[LoF97] Lopes A, Fiadeiro JL (1997) Preservation and reflection in specification. In: Proceedings of AMAST'97 LNCS 1349. Springer, Berlin Heidelberg New York, pp 380–394
[LFW02] Lopes A, Fiadeiro JL, Wermelinger M (2002) Architectural primitives for distribution and mobility. In: Proceedings of 10th symposium on the foundations of software engineering, ACM Press, pp 41–50
[LoF99] Lopes A, Fiadeiro JL (1999) Using explicit state to describe architectures. In: Proceedings of fundamental approaches to software engineering, LNCS 1577. Springer, Berlin Heidelberg New York, pp 144–160
[ShG96] Shaw M, Garlan D (1996) Software architecture: perspectives on an emerging discipline. Prentice-Hall
[WeF98] Wermelinger M, Fiadeiro JL (1998) Connectors for mobile programs. IEEE Trans Software Eng 24(5):331–341

Received November 2002
Accepted in revised form June 2003 by B. T. Denvir and E. Boiten