Theory and Practice of Software Architecture

José Fiadeiro

LabMOL/University of Lisbon and ATX Software
PORTUGAL

Summer School and Workshop on Generic Programming
St Anne's College, Oxford, UK
August 26-30 2002

bjectives

To provide mathematical foundations to the Theory and Practice of Software Architectures

- abstracting a mathematical semantics from existing languages and models
- using it to generalise these ideas to other contexts
- explore useful generalisations of existing concepts

capitalizing on research on SA, Reconfigurable Distributed Systems and Coordination Languages and Models

Outline

Motivation and overview

CommUnity: Parallel program design and architectural design using CT

Coordination in CommUnity, characterisation in CT and examples

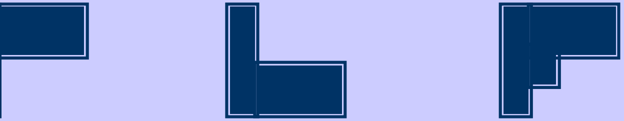
Software Architectures in Coordinated Categories

Software Evolution through Dynamic Reconfiguration

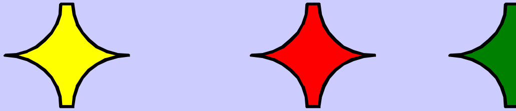
0

Motivation

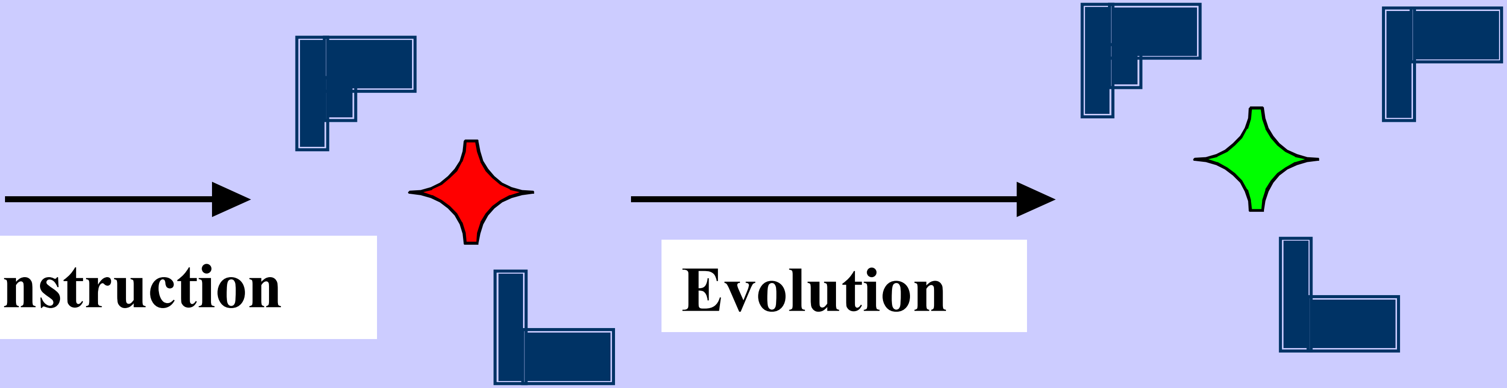
Envisaged development process



“structural objects”



“business rules”



Construction

Evolution

hy Software Architectures ?

SA addresses the gross decomposition of systems in terms of components and the connectors that define how they interact.

An attempt (the best we know…) at tackling the complexity of system development:

- Leads to "standard" ways of constructing systems – architectural styles – reflecting the structure of the application/business domain.
- Allows systems to evolve based on a black-box view of components – non-intrusive, dynamic reconfiguration – reflecting directly changes that take place in the domain.

Why Coordination ?

“Recent” languages like Linda, Gamma, Manifold, … have promoted the separation between **computation** (what is responsible for the functionality of services in basic components) and **coordination** (the mechanisms that are made available for components to interact);

“Programming by **emergence**”: local functionalities + interactions

Black-box view of components: interactions can evolve without changing the computations.

Why Category Theory ?

The mathematical tool, par excellence, for addressing "structure" and "modularity".

In Category Theory, entities are characterised in terms of the relationships they have to other entities and not in terms of their internal representation.

- The information one gets from the structure of an entity is determined from the way that entity "interacts" with the other entities.
- This is analogous, for instance, to the encapsulation mechanisms made available by Abstract Data Types and Object-Oriented Programming.

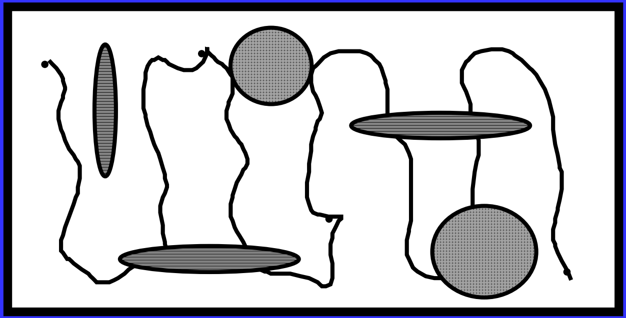
Category theory vs Set theory

→	∈
Implicit	Explicit
External	Internal
Black-box	White-box
“Social”	“Physiological”

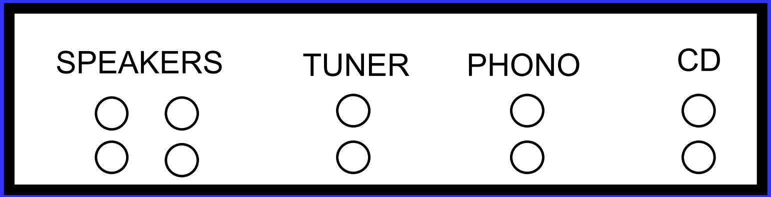
Category theory vs Set theory

Example

power amplifier in set theory



power amplifier in category theory



Set theory in Category theory

- The social life of sets;
- Characterisation of the empty set;
- Characterisation of singleton sets;
- Characterisation of the (disjoint) union;
- What makes a "social life" a category?

Uses of Category theory in Computing

- The “arrows as computations” paradigm
- The “arrows as interpretations” paradigm
  - General Systems Theory;
  - Abstract Data Types;
  - Concurrency Theory

1

Introduction to Category Theory

Graphs

A graph is a tuple

$$(G_0,G_1,src,trg)$$

where:

- **G_0** is a collection (of nodes),
- **G_1** is a collection (arrows),
- **src** maps each arrow to a node (the source of the node)
- **trg** maps each arrow to a node (the target of the node)

We usually write **f:x→y** to indicate that src(f)=x and trg(f)=y.

Between two nodes there may exist no arrows, just one in either direction, or several arrows, possibly in both directions.

Examples

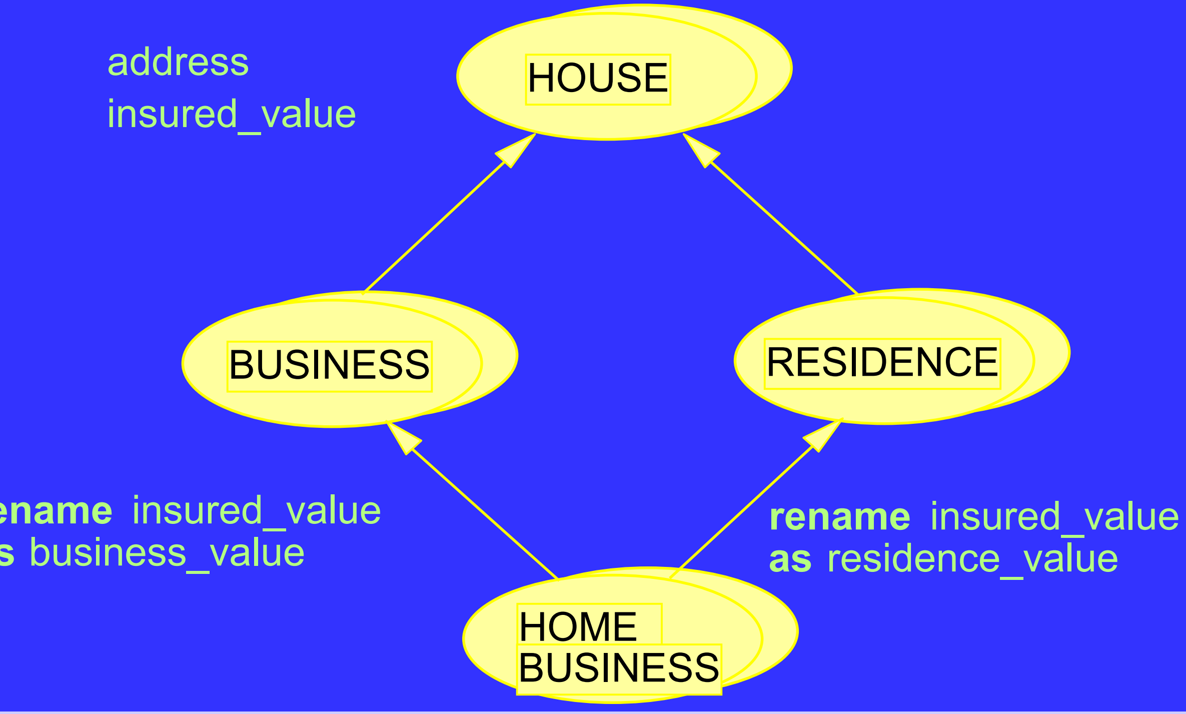
Around sets:

- The most "popular" graph is the graph whose nodes are the sets and whose arrows are the **total functions**.
- Another useful example is the graph that has exactly the same nodes (sets) but whose arrows are **partial functions**.

There are many other examples in Computing:

Class inheritance hierarchies

These are graphs whose nodes are object classes and for which the existence of an arrow between two nodes (classes) means that the source class inherits from the target class.



address
insured_value

HOUSE

BUSINESS

RESIDENCE

ename insured_value
s business_value

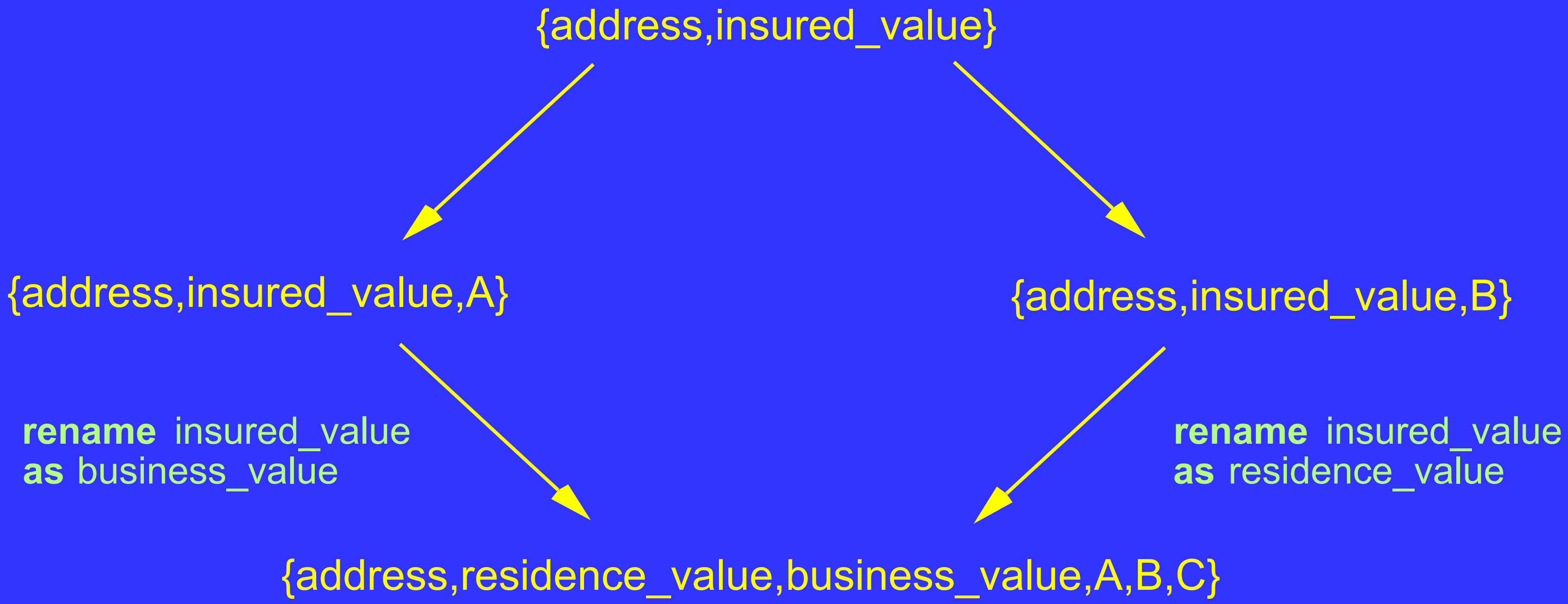
rename insured_value
as residence_value

HOME BUSINESS

In class inheritance hierarchies, there exists at most one arrow between two nodes. However, arrows can carry more information.

Class inheritance hierarchies

These are graphs whose nodes are object classes and for which the existence of an arrow between two nodes (classes) means that the source class inherits from the target class.



{address,insured_value}

{address,insured_value,A}

{address,insured_value,B}

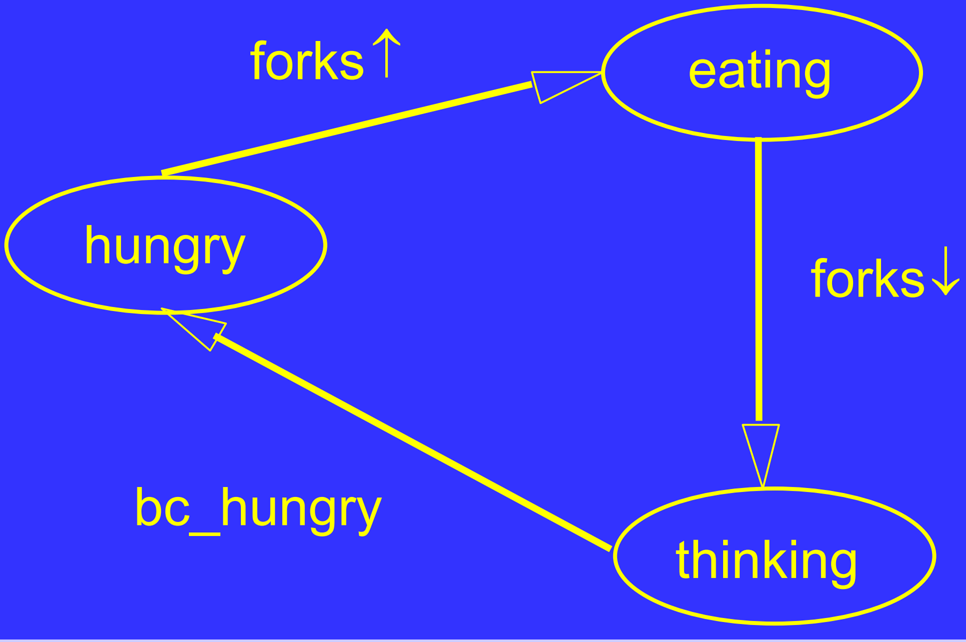
rename insured_value
as business_value

rename insured_value
as residence_value

{address,residence_value,business_value,A,B,C}

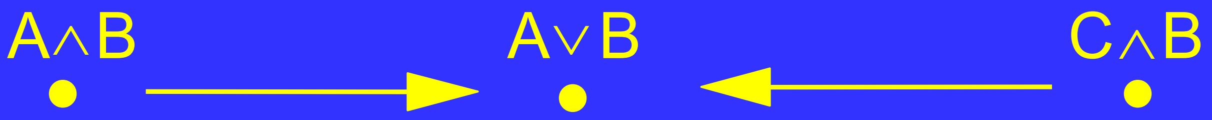
Transition systems

Every transition system constitutes a graph whose nodes are the states and whose arrows are the transitions



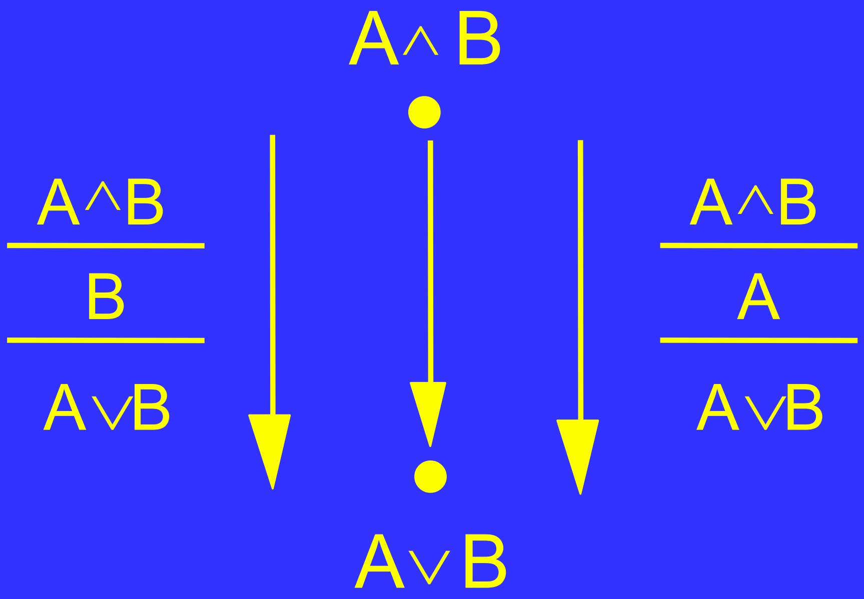
Consequence systems

One of the possible views that one can have of a "logic" is through the notion of a sentence being a consequence of, or derivable from, another sentence. This notion of consequence can be represented by a graph whose nodes are sentences and whose arrows correspond to "logical implication".



Proof systems

Every proof system constitutes a graph whose nodes are formulae and whose arrows are proofs.



$A \wedge B$

$$\frac{\dfrac{A\wedge B}{B}}{A\vee B} \qquad \frac{\dfrac{A\wedge B}{A}}{A\vee B}$$

$A \vee B$

Paths

Let G be a graph and x,y nodes of G.

A **path** from x to y of length $k>0$ is a sequence $f_k \ldots f_1$ of arrows of G (not necessarily distinct) such that

1. $src(f_1)=x$
2. $trg(f_i)=src(f_{i+1})$ for $1 \leq i \leq k-1$
3. $trg(f_k)=y$.

For every x, the path of length 0 at x (the empty path at x) from x to x is by convention the empty sequence.

Paths

The collection of paths of G of length k is denoted by G_k.

Hence,

- G_0 corresponds to the collection of nodes,
- G_1 corresponds to the collection of arrows,
- G_2 corresponds to the collection of pairs of composable arrows.

Graph Homomorphism

A homomorphism of graphs

$\varphi: G \rightarrow H$

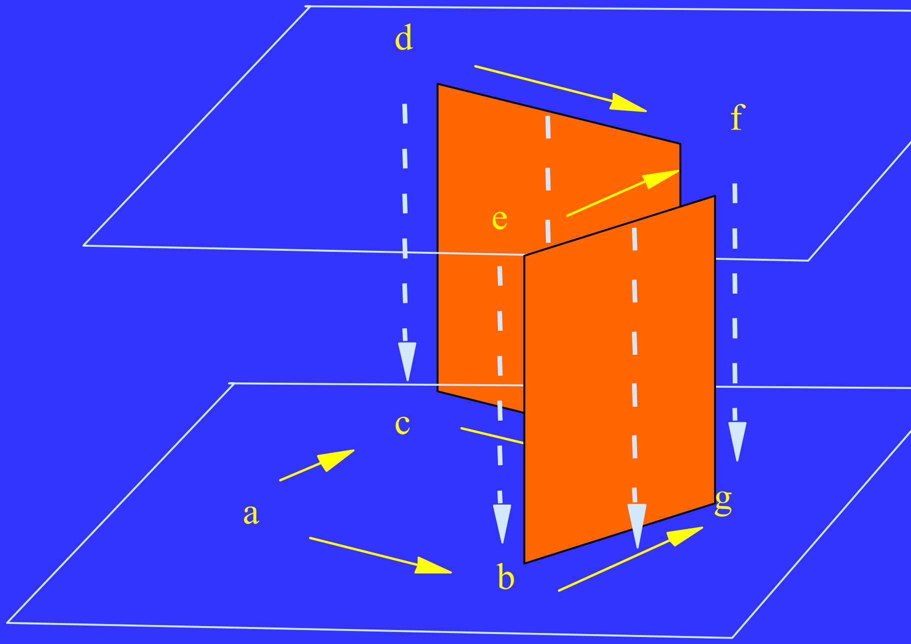
s a pair of maps

$\varphi_0: G_0 \rightarrow H_0$ and $\varphi_1: G_1 \rightarrow H_1$

uch that

for each arrow $f: x \rightarrow y$ of G ,

$\varphi_1(f): \varphi_0(x) \rightarrow \varphi_0(y)$ in H.



That is, nodes are mapped to nodes and arrows to arrows but preserving sources and targets.

Category

A category C is a triple $(G,;,id)$ where:

- G is a graph,
- $;$ is a map from G_2 into G_1
- **id** is a map from G_0 into G_1

such that

- $src(f;g)=src(f)$,
- $trg(f;g)=trg(g)$
- $(f;g);h = f;(g;h)$
- $src(id_x)=trg(id_x)=x$,
- for each $f:x\rightarrow y$ of G_1, $f;id_y = id_x;f = f$.

Examples

SET –

objects: sets

arrows: total functions

identities: identity functions

composition: functional

GRAPH –

objects: graphs

arrows: graph homorphisms

identities: identity functions

composition: functional

PROOF –

objects: sentences

arrows: proofs

identities: empty proofs

composition: cut rule

Pre-orders

Every pre-order $\langle S,\leq\rangle$ defins a category $\mathbf{S}_{\leq}$ as follows:

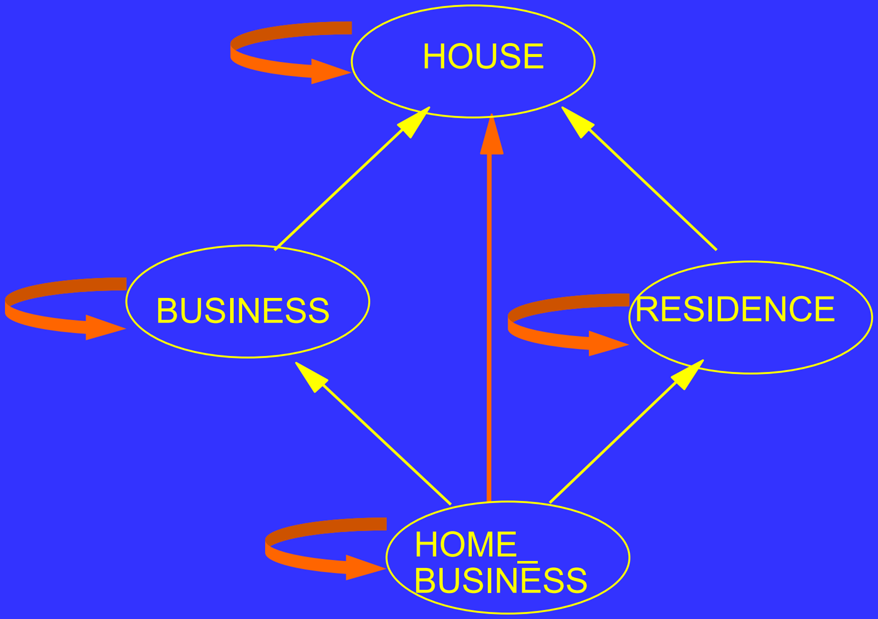
- **objects**: elements of S
- **arrows**: there is morphism $x \to y$ iff $x \leq y$;
- **identities**: reflexivity law;
- **composition**: transitive law.

LOGI –

- **objects**: sentences
- **arrows**: existence of a logical implication;

Ancestor

In Eiffel, given an inheritance graph *G* between classes, the category ***ancestor(G)*** is generated by completing the graph with the arrows that result from **reflexivity** (identities) and **transitivity** (compositions).



Category generated from a graph

Every graph *G* generates a category ***cat(G)*** as follows:

- **objects**: nodes
- **arrows**: paths
- **identities**: empty paths;
- **composition**: path concatenation.

Runs (T) - for every transition system T

- **objects**: states
- **arrows**: finite runs;

dding structure

The most typical way of building a new category is, perhaps, by adding "**structure**" to the objects of a given category (or a subset thereof).

The expression "**adding structure**" has, of course, a broad meaning...

The morphisms of the new category are then the morphisms of the old category that "**preserve**" the additional structure.

ointed sets

ET$_\perp$ -

objects: pairs $\langle A,\perp_A\rangle$ where A is a set and $\perp_A \in A$

arrows: $f{:}\langle A,\perp_A\rangle \to \langle B,\perp_B\rangle$ is $f{:}A \to B$ s.t. $f(\perp_A)=\perp_B$

identities: those of **SET**

composition: that of **SET**

proof obligations:

well-formedness of identities;

closure of composition

Processes

Pointed sets can be interpreted as **process alphabets:**

- Elements denote events;
- The designated element denotes an environment event;
- Morphisms identify sub-components of processes.

We can associate **trajectories** (full behaviours) with alphabets and their morphisms:

$$\mathbf{tra}(A)=\{\lambda:\omega\rightarrow A\}$$

$$\mathbf{tra}(f:A\rightarrow B)(\lambda)=\lambda;f:\omega\rightarrow B$$

Processes

PROC –

objects: pairs $\langle A_\perp,\Lambda\rangle$ where $A_\perp$: **SET**$_\perp$ and $\Lambda\subseteq$**tra**(A)

arrows: $f:\langle A_\perp,\Lambda\rangle\rightarrow\langle B_\perp,M\rangle$ is $f:A_\perp\rightarrow B_\perp$ s.t. **tra**$(f)(\Lambda)\subseteq M$

identities: those of **SET**$_\perp$

composition: that of **SET**$_\perp$

proof obligations:

well-formedness of identities;
closure of composition

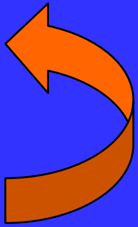
Processes

process VM **is**

alphabet co, ca, ci

behaviour

$$\Lambda ::= \bot^{\omega} \mid \bot^{*}\mathrm{co}\bot^{\omega} \mid (\bot^{*}\mathrm{co}\bot^{*}\{\mathrm{ca},\mathrm{ci}\})\Lambda$$



process RVM **is**

alphabet co, ca, ci, to

behaviour

$$\Lambda ::= \bot^{\omega} \mid \bot^{*}\mathrm{co}\bot^{\omega} \mid (\bot^{*}\mathrm{co}\bot^{*}\mathrm{ca})\Lambda \mid (\bot^{*}\mathrm{co}\bot^{*}\mathrm{to}\bot^{*}\mathrm{ci})\Lambda$$

Temporal specifications

FSET -

objects: finite sets **arrows**: total functions

Linear temporal language PROP(Σ) over a finite set Σ:

ϕ ::= **beg** | $a \in \Sigma$ | $\neg\phi$ | $\phi_1 \supset \phi_2$ | ϕ_1 **U** ϕ_2

Translation defined by $f:\Sigma \rightarrow \Sigma'$

$\underline{f}(\phi)$ ::= **beg** | $\underline{f}(a) \in \Sigma$ | $\neg\underline{f}(\phi)$ | $\underline{f}(\phi_1) \supset \underline{f}(\phi_2)$ | $\underline{f}(\phi_1)$ **U** $\underline{f}(\phi_2)$

Temporal specifications

Semantics of PROP(Σ) over $(2^{\Sigma})^{\omega}$

$\lambda ,^{i} a$ iff $a \in \lambda(i)$

$\lambda ,^{i}$ **beg** iff $i=0$,

$\lambda ,^{i} \neg\phi$ iff it is not the case that $\lambda ,^{i} \phi$

$\lambda ,^{i} \phi_1 \supset \phi_2$ iff $\lambda ,^{i} \phi_1$ implies $\lambda ,^{i} \phi_2$,

$\lambda ,^{i} \phi_1 \mathbf{U} \phi_2$ iff, for some $j>i$, $\lambda ,^{i} \phi_2$ and, for every $i<k<j$, $\lambda ,^{k} \phi_1$

λ , ϕ iff $\lambda ,^{i} \phi$ for every i

Φ , ϕ iff, for every λ, λ , Φ implies λ , ϕ

Temporal specifications

THEO-

- **objects**: theories $\langle\Sigma,\Phi\rangle$ such that Φ is closed
- **arrows**: $f:\langle\Sigma,\Phi\rangle\rightarrow\langle\Sigma',\Phi'\rangle$ is $f:\Sigma\rightarrow\Sigma'$ s.t. $f(\Phi)\subseteq\Phi'$

PRES-

- **objects**: theory presentations $\langle\Sigma,\Phi\rangle$
- **arrows**: $f:\langle\Sigma,\Phi\rangle\rightarrow\langle\Sigma',\Phi'\rangle$ is $f:\Sigma\rightarrow\Sigma'$ s.t. $f(\Phi)\subseteq c(\Phi')$

  where $c(\Phi)=\{\phi: \Phi \;, \phi\}$

Temporal specifications

Specification vending machine **is**

signature coin, cake, cigar

axioms

$\mathbf{beg} \supset \neg \mathrm{cake} \wedge \neg \mathrm{cigar} \wedge (\mathrm{coin} \vee (\neg \mathrm{cake} \wedge \neg \mathrm{cigar})\,\mathbf{W}\,\mathrm{coin})$

$\mathrm{coin} \supset (\neg \mathrm{coin})\,\mathbf{W}\,(\mathrm{cake} \vee \mathrm{cigar})$

$(\mathrm{cake} \vee \mathrm{cigar}) \supset (\neg \mathrm{cake} \wedge \neg \mathrm{cigar})\,\mathbf{W}\,\mathrm{coin}$

$\mathrm{cake} \supset \neg \mathrm{cigar}$

Temporal specifications

Specification regulated vending machine **is**

signature coin, cake, cigar, token

axioms

$\mathbf{beg} \supset \neg\text{cake} \wedge \neg\text{cigar} \wedge \neg\text{token} \wedge$
$(\text{coin} \vee (\neg\text{cake} \wedge \neg\text{cigar})\mathbf{W}\text{coin})$

$\text{coin} \supset (\neg\text{coin})\mathbf{W}(\text{cake} \vee \text{cigar})$

$\text{coin} \supset (\neg\text{cigar})\mathbf{W}\text{token}$

$(\text{cake} \vee \text{cigar}) \supset (\neg\text{cake} \wedge \neg\text{cigar})\mathbf{W}\text{coin}$

$\text{cake} \supset \neg\text{cigar}$

Temporal specifications

Specification regulator **is**

signature tri, ted, tor

axioms

$\mathbf{beg} \supset \neg \mathrm{tor}$

$\mathrm{tri} \supset (\neg \mathrm{ted})\,\mathbf{W}\,\mathrm{tor}$

What relationships can be established between vending machine, regulated vending machine and regulator ?

Functors

- the social life of categories

Given a category C

- $|C|$ denotes the collection of nodes of C
- $\mathrm{Hom}_C(x,y)$ denotes the collection of morphisms from x to y.

Let C and D be categories.

A **functor** $\Phi{:}C\rightarrow D$ is a graph homomorphism from the graph of C into the graph of D such that:

- $\Phi_1(f;g) = \Phi_1(f);\Phi_1(g)$ for each path gf in C_2
- $\Phi_1(id_x) = id_{\Phi0(x)}$ for each x in C_0.

Functors

Examples

Sign:PRES→fSET s.t. $\mathrm{Sign}(\langle\Sigma,\Phi\rangle)=\Sigma$

Alph:PROC→$\mathrm{SET}_\perp$ s.t. $\mathrm{Alph}(\langle A_\perp,\Lambda\rangle)=A_\perp$

These are examples of *forgetful functors*: they “forget” part of the structure of the source category.

Sem: PRES→PROC^{op} s.t.

- $\mathrm{Sem}(\langle\Sigma,\Phi\rangle)=\langle 2^\Sigma,\{\lambda:\omega\rightarrow 2^\Sigma \mid \lambda \models \Phi\}$
- $\mathrm{Sem}(f:\langle\Sigma,\Phi\rangle\rightarrow\langle\Sigma',\Phi'\rangle)=f^{-1}:2^{\Sigma'}\rightarrow 2^\Sigma$

Universal Constructions

Isomorphisms

Let *C* be a category and x,y objects of *C*.

A morphism f:x→y of *C* is said to be an **isomorphism** iff here is a morphism g:y→x of *C* such that:

$$f;g = id_x \text{ and } g;f = id_y.$$

In these conditions, x and y are said to be **isomorphic**.

Universal Constructions

Initial objects

An object x of a category C is said to be **initial** iff, for each object y of C, there is a unique morph. from x to y.

Two initial objects are isomorphic. Hence, we usually refer to **the** initial object of a category, if it exists.

Terminal objects

An object is terminal in a category C iff it is initial in C^{op}. That is, x is terminal in C iff, for each object y of C, there is a unique morphism from y to x.

Sums / Coproducts

Let C be a category and x,y objects of C.

The object **z** is said to be the **sum** (or **coproduct**) of x and y with injections $\mathbf{i_x : x \to z}$ and $\mathbf{i_y : y \to z}$ iff for any object v and pair $f_x : x \to v$, $f_y : y \to v$ of C there is a unique $k : z \to v$ in C such that $i_x ; k = f_x$ and $i_y ; k = f_y$.

$$\begin{tikzcd}
x \arrow[dr, "i_x"] \arrow[ddr, bend right, "f_x"'] & & y \arrow[dl, "i_y"'] \arrow[ddl, bend left, "f_y"] \\
& x+y \arrow[d, "k"] & \\
& v &
\end{tikzcd}$$

If the sum of x and y exists, it is unique up to isomorphism (denoted x+y).

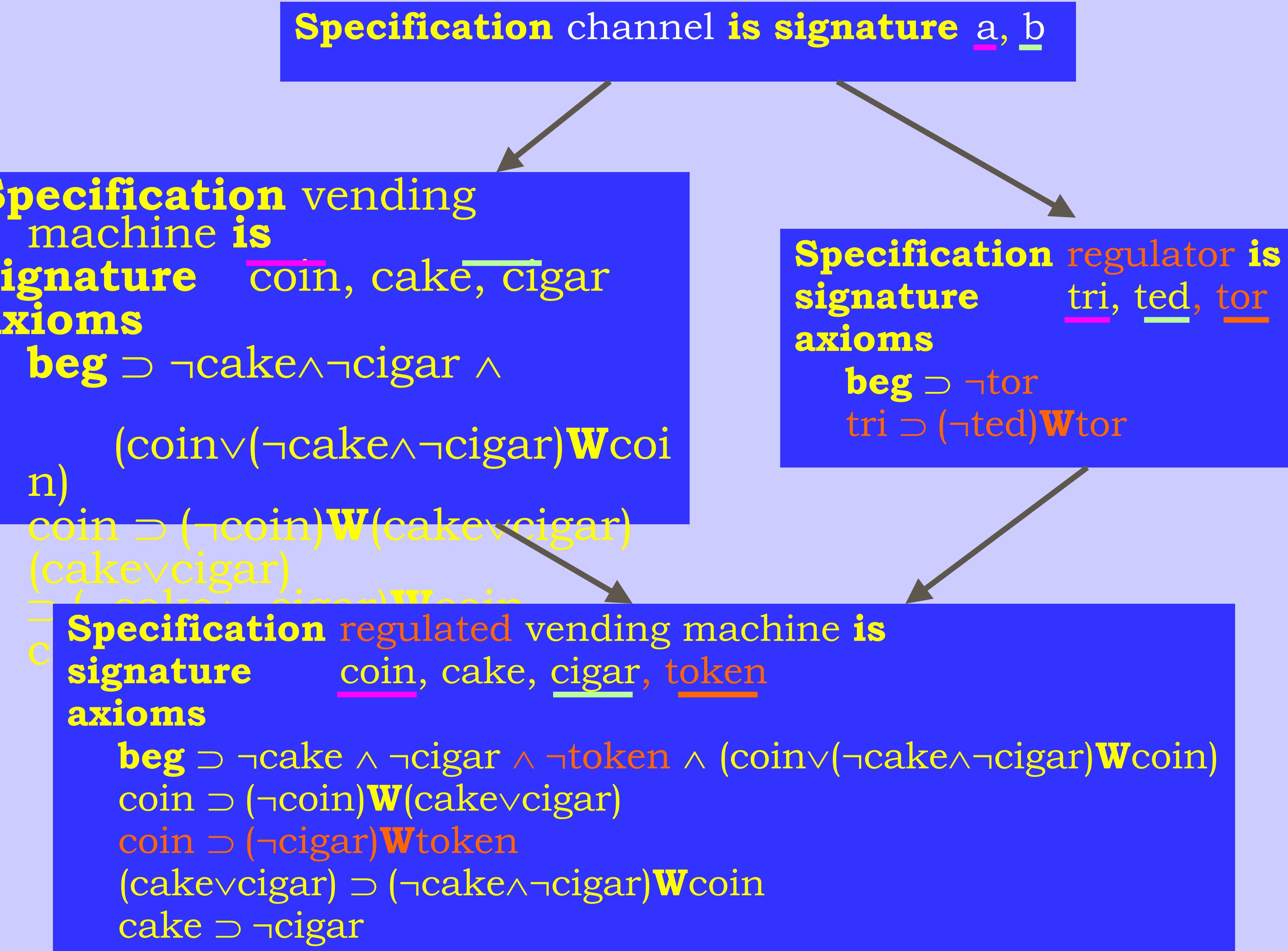
Universal Constructions

Amalgamated Sums / Pushouts

Let C be a category and $f:x\rightarrow y$, $g:x\rightarrow z$ morphisms of C. The amalgamated sum (or pushout) of f and g consists of two morphisms **$f':y\rightarrow w$** and **$g':z\rightarrow w$** such that

- $f;f' = g;g'$
- for any other $f'':y\rightarrow v$ and $g'':z\rightarrow v$ such that $f;f'' = g;g$, there is a unique morphism $k:w\rightarrow v$ in C such that $f';k = f''$ and $g';k = g''$.

$$\begin{tikzcd}
& y \arrow[dr, "f'"] \arrow[drr, bend left, "f''"] & & \\
x \arrow[ur, "f"] \arrow[dr, "g"'] & & w \arrow[r, "k"] & v \\
& z \arrow[ur, "g'"'] \arrow[urr, bend right, "g''"'] & &
\end{tikzcd}$$



Specification channel **is signature** a, b

Specification vending
machine **is**
ignature coin, cake, cigar
xioms
beg ⊃ ¬cake∧¬cigar ∧
(coin∨(¬cake∧¬cigar)**W**coin)
coin ⊃ (¬coin)**W**(cake∨cigar)
(cake∨cigar)
⊃ (¬cake∧¬cigar)**W**coin
c

Specification regulator **is**
signature tri, ted, tor
axioms
beg ⊃ ¬tor
tri ⊃ (¬ted)**W**tor

Specification regulated vending machine **is**
signature coin, cake, cigar, token
axioms
beg ⊃ ¬cake ∧ ¬cigar ∧ ¬token ∧ (coin∨(¬cake∧¬cigar)**W**coin)
coin ⊃ (¬coin)**W**(cake∨cigar)
coin ⊃ (¬cigar)**W**token
(cake∨cigar) ⊃ (¬cake∧¬cigar)**W**coin
cake ⊃ ¬cigar

Diagrams

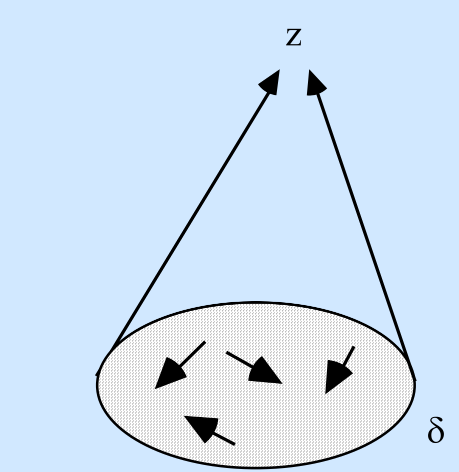
Let C be a category and I a graph. A **diagram** in C with shape I is a graph homomorphism $\delta : I \to G(C)$ where G(C) is the underlying graph of C.

- The homomorphism corresponds to a labelling of the graph I.
- A diagram in a category can be seen as a graph whose nodes are labelled with objects and the arrows are labelled with morphisms of that category.
- The diagram δ is said to **commute** iff, for every pair x,y of nodes and every pair of paths $w=u_m \ldots u_1$, $w'=v_n \ldots v_1$ from x to y in graph I, $\delta_{um} \circ \ldots \circ \delta_{u1} = \delta_{vn} \circ \ldots \circ \delta_{v1}$ holds in C.

Cocones

Let $\delta:I\to C$ be a diagram in a category C. A **cocone** with base δ s an object z of C together with a family $\{p_a:\delta_a\to z\}_{a\in I0}$ of morphisms of C, usually denoted by $p:\delta\to z$.

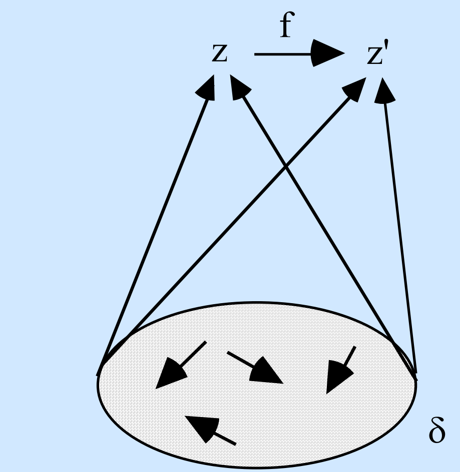
- The object z is said to be the **vertex** of the cocone, and, for each $a\in I_0$, the morphism p_a is said to be the **edge** of the cocone at point a.
- A cocone p with base $\delta:I\to C$ and vertex z is said to be **commutative** iff for every arrow $s:a\to b$ of graph I, $p_b\circ\delta_s = p_a$.



Colimits

Let $\delta: I \to C$ be a diagram in a category C.

A **colimit** of δ is a commutative cocone $p:\delta \to z$ such that, for every other commutative cocone $p':\delta \to z'$, there is a unique morphism $f:z \to z'$ such that $f \circ p = p'$, i.e. $f \circ p_a = p'_a$ for every edge.



Cocompleteness

A category is (finitely) **cocomplete** if all (finite) diagrams have colimits.

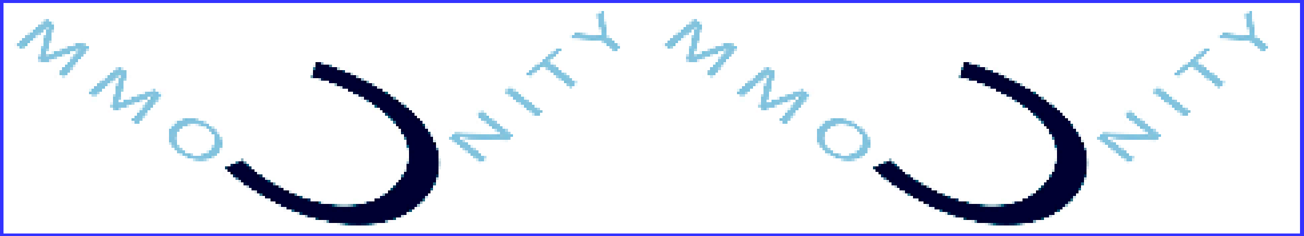
There are several results on the (finite) co completeness of categories. A commonly used one is:

A category C is finitely cocomplete

iff

it has initial object and pushouts of all pairs of morphisms with common source.

2



arallel Program Design using CT

CT can be used as a mathematical framework in which designs, configurations and relationships between designs, such as refinement, can be formally described

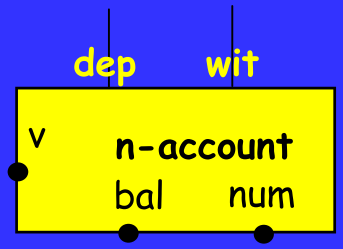
We shall illustrate this ability using a parallel program design language - COMMUNITY

COMMUNITY: Designing the components

An example

A design that models a naive bank account

```
design n-account is
out   num:nat, bal:int
in    v: nat
do    dep: true → bal:=v+bal
[]    wit: bal≥v → bal:=bal-v
```



Designing the components

Another example

The design of a VIP-account that may accept a withdrawal when the balance together with a given credit amount is greater than the equested amount.

```
design vip-account[CRE:nat] is
out  num: nat, bal:int
in   v: nat
do   dep[bal]: true → bal′=v+bal
[]   wit[bal]: bal+CRE≥v, bal≥v → bal′≤bal-v
```

Designing the components

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

Σ: an algebraic specification of the underlying data types

$D(g) \subseteq out(V) \cup prv(V)$: local vars that can be modified by *g*.

$L(g)$, $U(g)$: two conditions on V s.t. $L(g) \supset U(g)$. They define an interval in which the enabling condition of any guarded command that implements *g* must lie.

$R(g)$: a condition on V, $D(g)$ and $D(g)'$. It defines requirements over the values of variables in $D(g)$, after the execution of *g*.

perational Semantics

When, for every action *g*,

- L(*g*) and U(*g*) coincide
- R(*g*) defines a conditional multiple assignment

the design is **a program**.

Execution of a closed program (no input vars):

- at each step, one of the actions whose enabling condition holds is selected and its assignments are executed atomically
- shared actions can be selected by the environment
- private actions are internally selected in a fair way: every private action that is infinitely often enabled is selected an infinite number of times

Superposition

A structuring mechanism for the design of systems that allows to build on already designed components by “augmenting“ them while “preserving“ their properties.

Typically, the additional behaviour results from the introduction of new variables and corresponding assignments (that may use the values of the variables of the base design).

Applying Superposition

An example

Extending the design of n-account to control how many days the balance has exceed a given amount since the last reset.

```
design e-account[MAX:int] is
out    num: nat, bal:int
in     v,day:nat
out    count:int
prv    d:int
do     dep[bal,d,count]: true → bal′=v+bal ∧ d′=day ∧
                      (bal≥MAX ⊃ count′=count+(day-d)) ∧
                        (bal<MAX ⊃ count′=count)
[]     wit[bal,d,count]: bal≥v → bal′=bal-v ∧ d′=day ∧
                      (bal≥MAX ⊃ count′=count+(day-d)) ∧
                      (bal<MAX ⊃ count′=count)
[]     reset: true, false → count:=0||d′:=day
```

Characterising Superposition

The relationship between a design P_1 and a design P_2 obtained from P_1 through the superposition of additional behaviour, can be modelled as a morphism

$$\sigma : P_1 \rightarrow P_2$$

n a suitable category of designs.

Superposition Morphisms

A superposition morphism $\sigma : P_1 \rightarrow P_2$ consists of

- a total function $\sigma_{var} : V_1 \rightarrow V_2$ s.t.
  - $sort_2(\sigma_{var}(v)) = sort_1(v)$
  - $\sigma_{var}(out(V_1)) \subseteq out(V_2)$
  - $\sigma_{var}(in(V_1)) \subseteq out(V_2) \cup in(P_2)$
  - $\sigma_{var}(prv(V_1)) \subseteq prv(V_2)$

Sorts, privacy and availability
of vars are preserved
In vars may become out vars

- a partial mapping $\sigma_{ac} : \Gamma_2 \rightarrow \Gamma_1$ s.t.
  - $\sigma_{ac}(sh(\Gamma_2)) \subseteq sh(\Gamma_1)$
  - $\sigma_{ac}(prv(\Gamma_2)) \subseteq prv(\Gamma_1)$
  - $\sigma_{var}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$
  - $\sigma_{ac}(D_2(\sigma_{var}(v))) \subseteq D_1(v)$

Privacy/availability
of actions is preserved
Domains of vars are preserved

Superposition Morphisms

nd, moreover, for every g in Γ_2 s.t. $\sigma_{ac}(g)$ is defined

- $R_2(g) \supset \underline{\sigma}(R_1(\sigma_{ac}(g)))$
- $L_2(g) \supset \underline{\sigma}(L_1(\sigma_{ac}(g)))$
- $U_2(g) \supset \underline{\sigma}(U_1(\sigma_{ac}(g)))$

Effects of actions must be preserved or made more deterministic

The bounds for enabling conditions of actions can be strengthened but not weakened

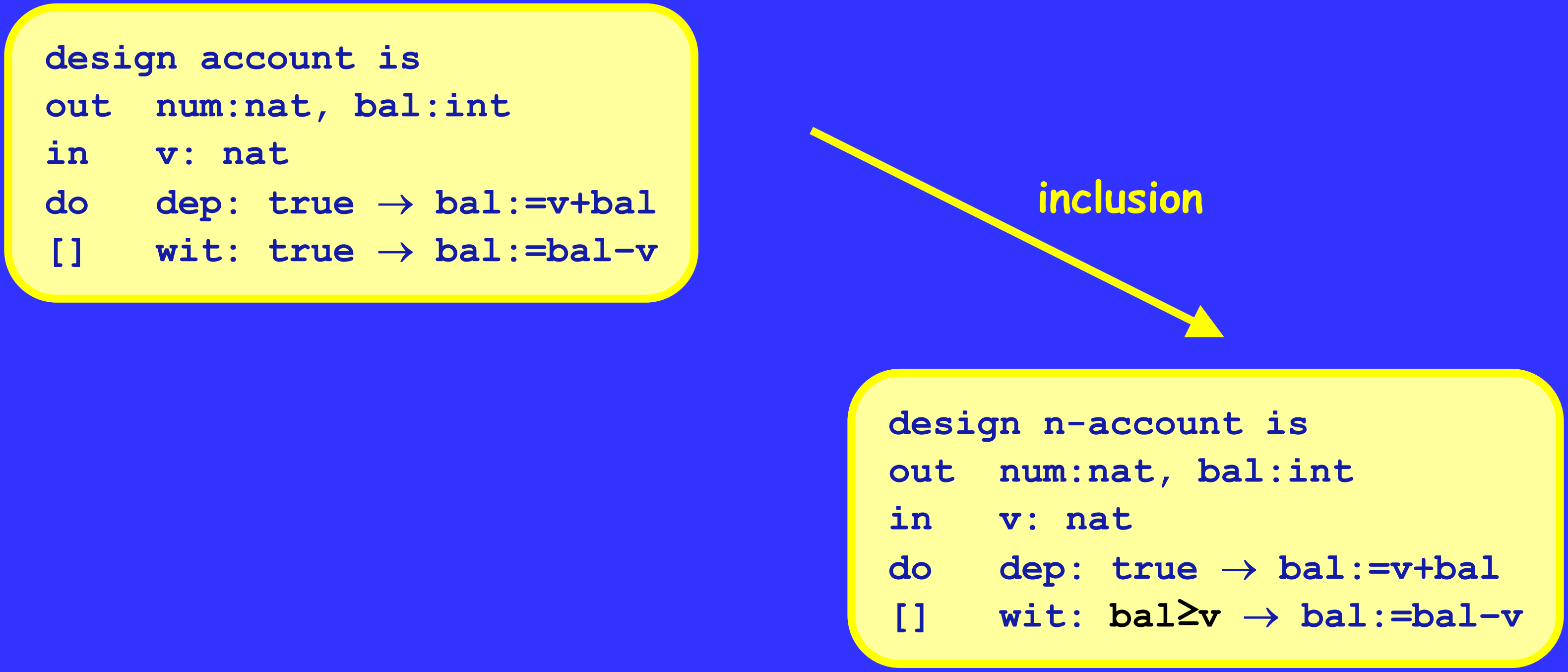
Superposition Morphisms: Examples

```
design n-account is
out    num:nat, bal:int
in     v:nat
do     dep[bal]: true → bal′=v+bal
[]     wit[bal]: bal≥v → bal′=bal-v
```

```
design e-account[MAX:int] is
out    num:nat, bal:int
in     v,day:nat
out    count:int
prv    d:int
do     dep[bal,d,count]: true → bal′=v+bal ∧ d′=day ∧
                         (bal≥MAX ⊃ count′=count+(day-d)) ∧
                         (bal<MAX ⊃ count′=count)
[]     wit[bal,d,count]: bal≥v → bal′=bal-v ∧ d′=day ∧
                         (bal≥MAX ⊃ count′=count+(day-d)) ∧
                         (bal<MAX ⊃ count′=count)
[]     reset: true, false → count:=0||d:=day
```

Superposition Morphisms: Examples

Another example



```
design account is
out   num:nat, bal:int
in    v: nat
do    dep: true → bal:=v+bal
[]    wit: true → bal:=bal-v
```

inclusion

```
design n-account is
out   num:nat, bal:int
in    v: nat
do    dep: true → bal:=v+bal
[]    wit: bal≥v → bal:=bal-v
```

Externalising the superposed behaviour

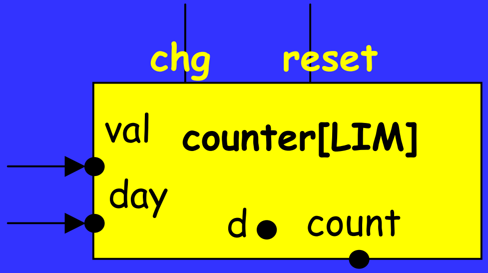
- These examples represent two typical kinds of superposition
  - monitoring
  - regulation
- The superposed behaviour can be captured by a component
  - monitor
  - regulator

Support reuse

and the new design is obtained by interconnecting the underlying design with this component.

e-account: Externalising the counter

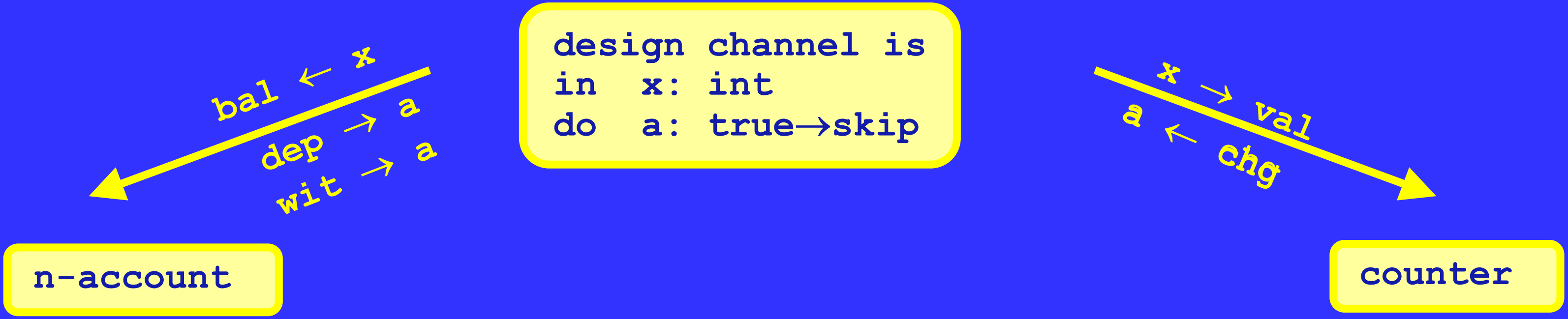
A design of a counter that counts how many days a value has exceed a iven value, since the last time it was reset



```
design counter[LIM:int] is
in     val,day:nat
out    count:int
prv    d:int
do     chg[d,count]: true → d′=day ∧
              (val≥LIM ⊃ count′=count+(day-d)) ∧
              (val<LIM ⊃ count′=count)
[]     reset: true, false → count:=0||d′:=day
```

e-account: Externalising the counter

o identify which variables and actions of the account are he subject of the monitoring expressed by the counter, we se the categorical diagram



```
design channel is
in   x: int
do   a: true→skip
```

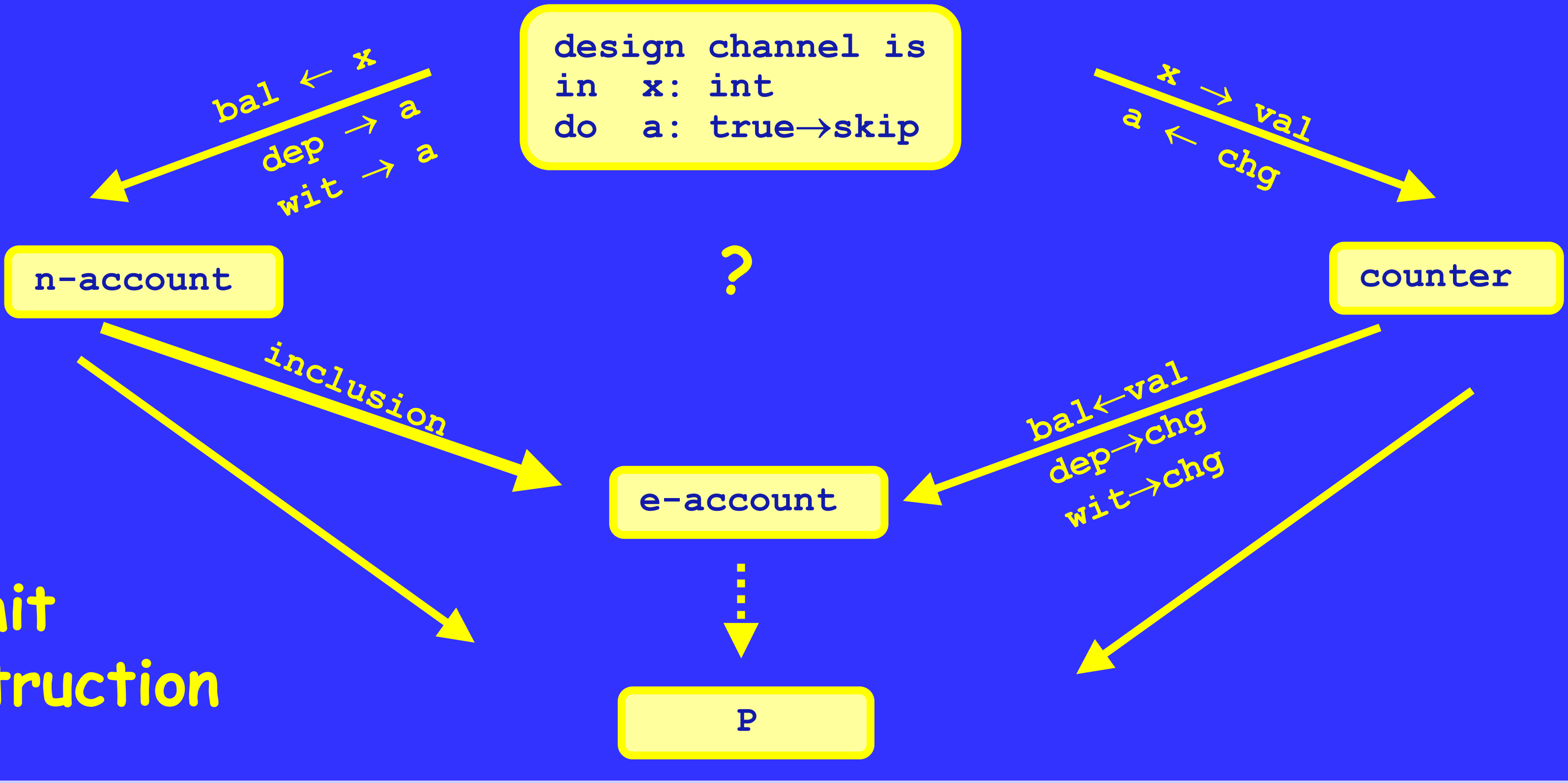
his diagram captures the configuration of a system with wo components — n-account and counter — that are nterconnected through a third design (a communication

onfigurations

- Using diagrams whose nodes are labelled by designs and whose arcs are labelled by superposition morphisms, it is possible to design large systems from simpler components.
- Interactions between components are required to be made explicit by providing the corresponding name bindings.
- Name bindings are represented as additional nodes labelled with designs and edges labelled by morphisms.

Semantics of Configurations: e-account

What's the relationship between **e-account** and the configuration?



```
design channel is
in   x: int
do   a: true→skip
```

bal ← x

dep → a

wit → a

x → val

a ← chg

n-account

?

counter

inclusion

bal←val

dep→chg

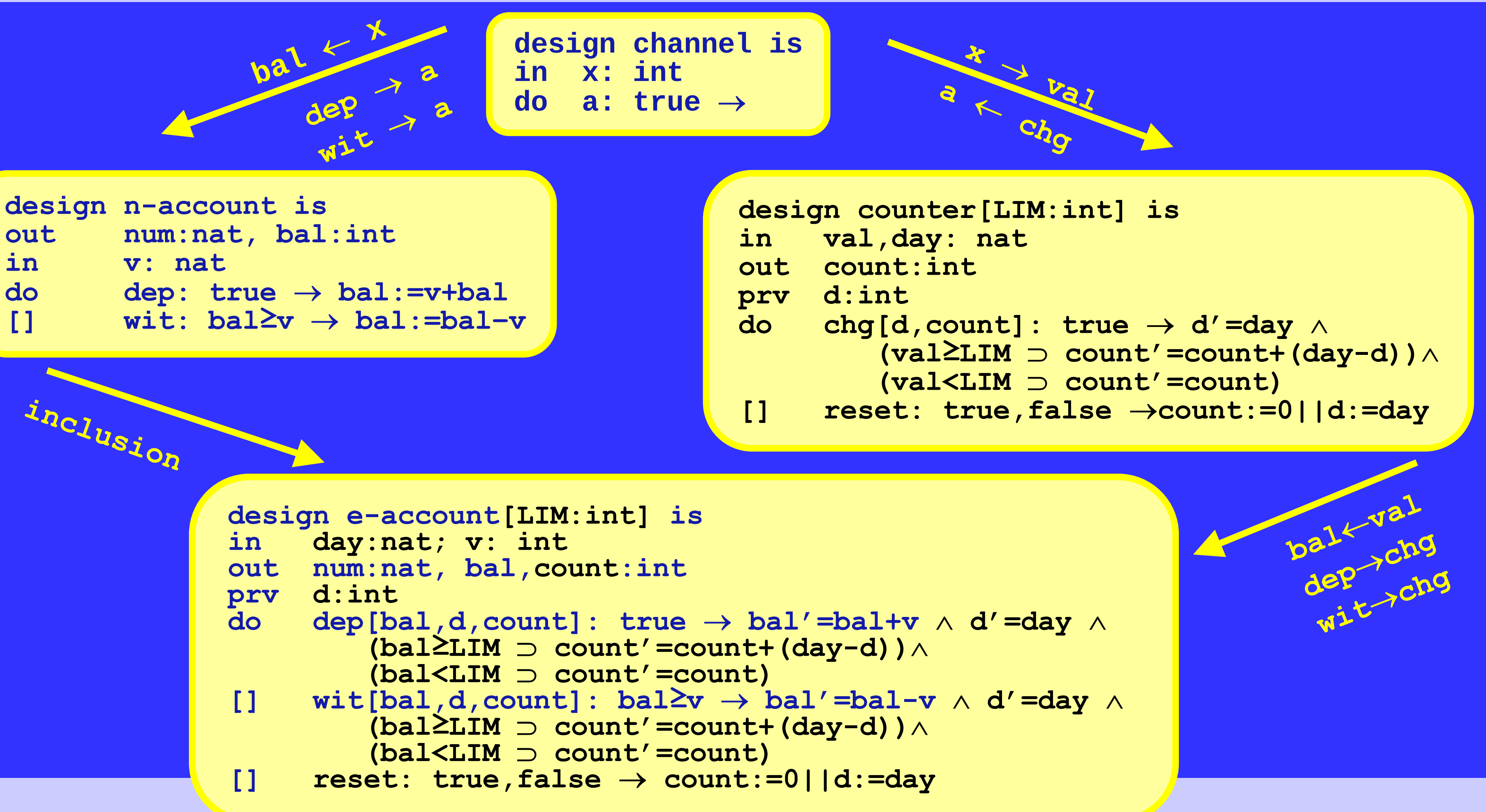
wit→chg

e-account

P

Colimit construction

Semantics of Configurations: e-account



```
design channel is
in  x: int
do  a: true →
```

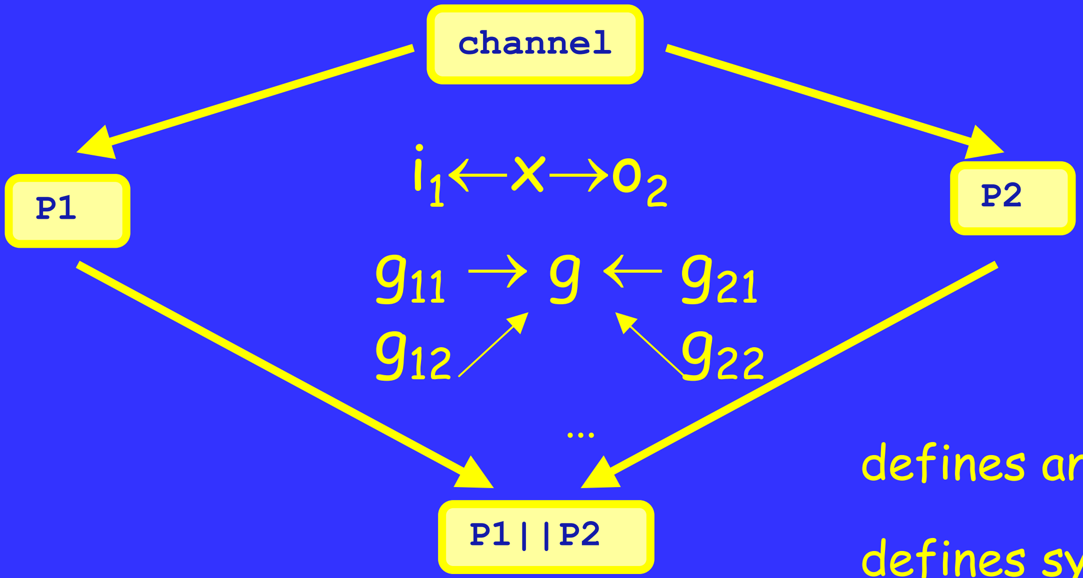
```
design n-account is
out    num:nat, bal:int
in     v: nat
do     dep: true → bal:=v+bal
[]     wit: bal≥v → bal:=bal-v
```

```
design counter[LIM:int] is
in   val,day: nat
out  count:int
prv  d:int
do   chg[d,count]: true → d'=day ∧
         (val≥LIM ⊃ count'=count+(day-d))∧
         (val<LIM ⊃ count'=count)
[]   reset: true,false →count:=0||d:=day
```

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

Semantics of Configurations

The semantics of configurations is given by a categorical construction: the colimit of the underlying diagram.



$i_1 \leftarrow x \rightarrow o_2$

$$\begin{tikzcd}
g_{11} \arrow[r] & g & g_{21} \arrow[l] \\
g_{12} \arrow[ur] & & g_{22} \arrow[ul]
\end{tikzcd}$$

...

defines an I/O connection

defines synchronisation sets $\{g_{11}, g_{21}\}, \{g_{12}, g_{21}\}, \ldots$

Semantics of Configurations

The colimit of such design diagrams

- Amalgamates vars involved in each i/o interconnection and the result is an output var of the system design
- Represents every synchronisation set $\{g_1,g_2\}$ by a single action $g_1|g_2$ with
  - safety bound: conjunction of the safety bounds of g_1 and g_2
  - progress bound: conjuction of the progress bounds of g_1 and g_2
  - conditions on next state: conjunction of conditions of g_1 and g_2

Configurations

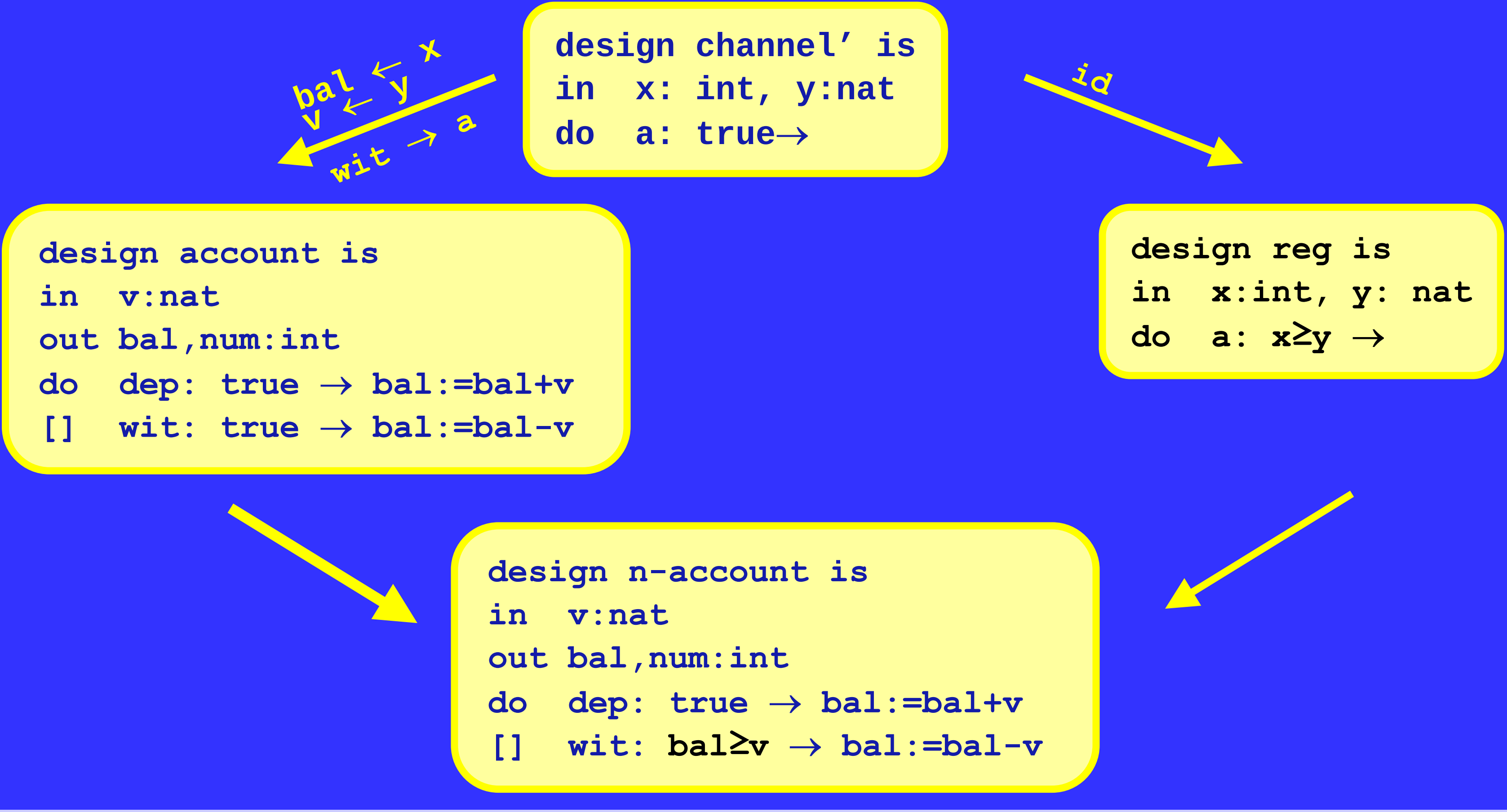
Not every diagram represents a meaningful configuration.

Restrictions on diagrams that make them well-formed configurations:

- An output variable of a component cannot be connected (directly or indirectly through input variables) with output variables of the same or other components.
- Private variables and private actions cannot be involved in the connections.

These restrictions cannot be captured by the notion of morphism because they involve the whole diagram.

-account: Externalising the regulator



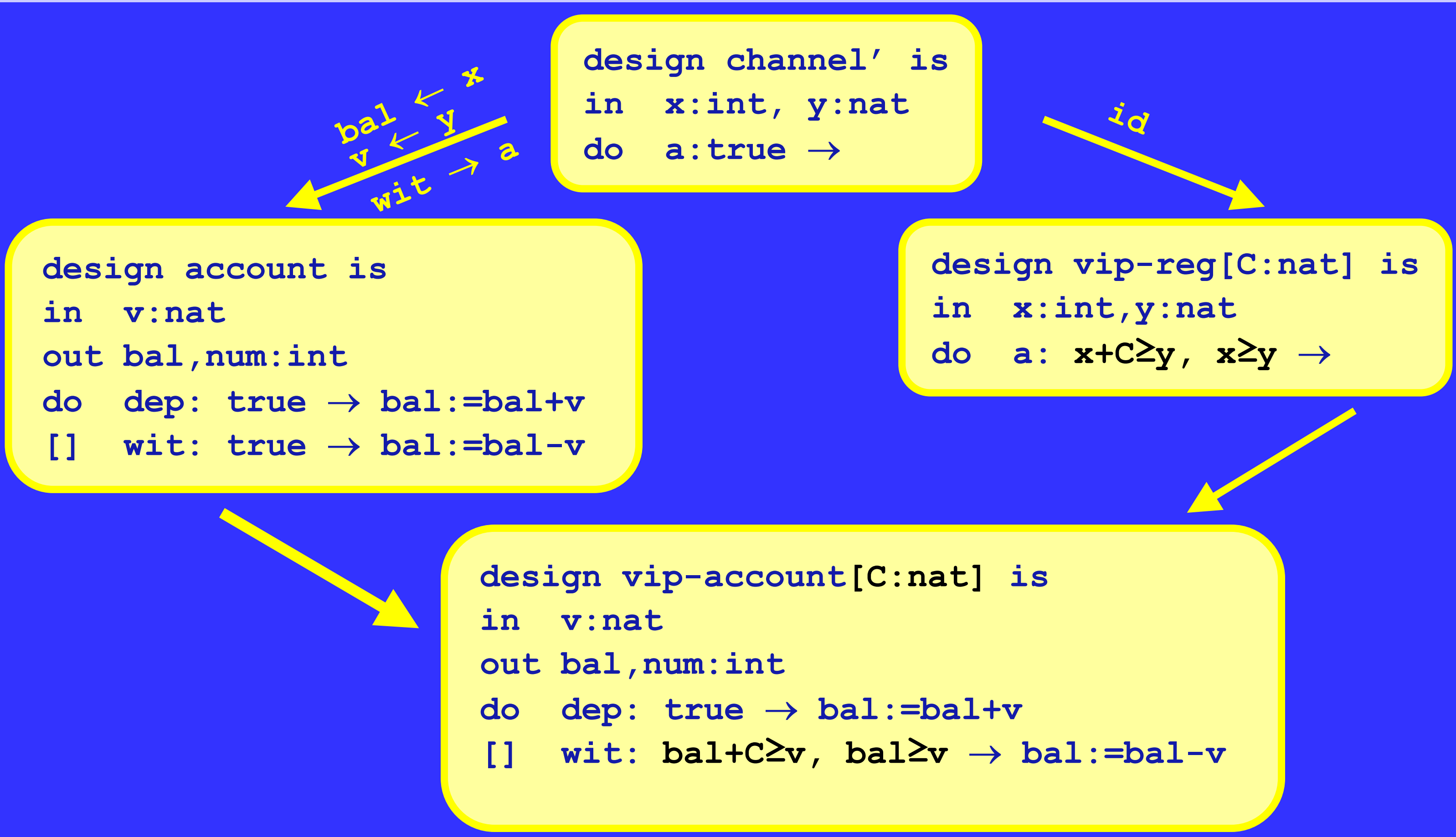
```
design channel' is
in  x: int, y:nat
do  a: true→
```

```
design account is
in  v:nat
out bal,num:int
do  dep: true → bal:=bal+v
[]  wit: true → bal:=bal-v
```

```
design reg is
in  x:int, y: nat
do  a: x≥y →
```

```
design n-account is
in  v:nat
out bal,num:int
do  dep: true → bal:=bal+v
[]  wit: bal≥v → bal:=bal-v
```

vip-account: an account with a different regulator



```
design channel' is
in   x:int, y:nat
do   a:true →
```

```
design account is
in   v:nat
out  bal,num:int
do   dep: true → bal:=bal+v
[]   wit: true → bal:=bal-v
```

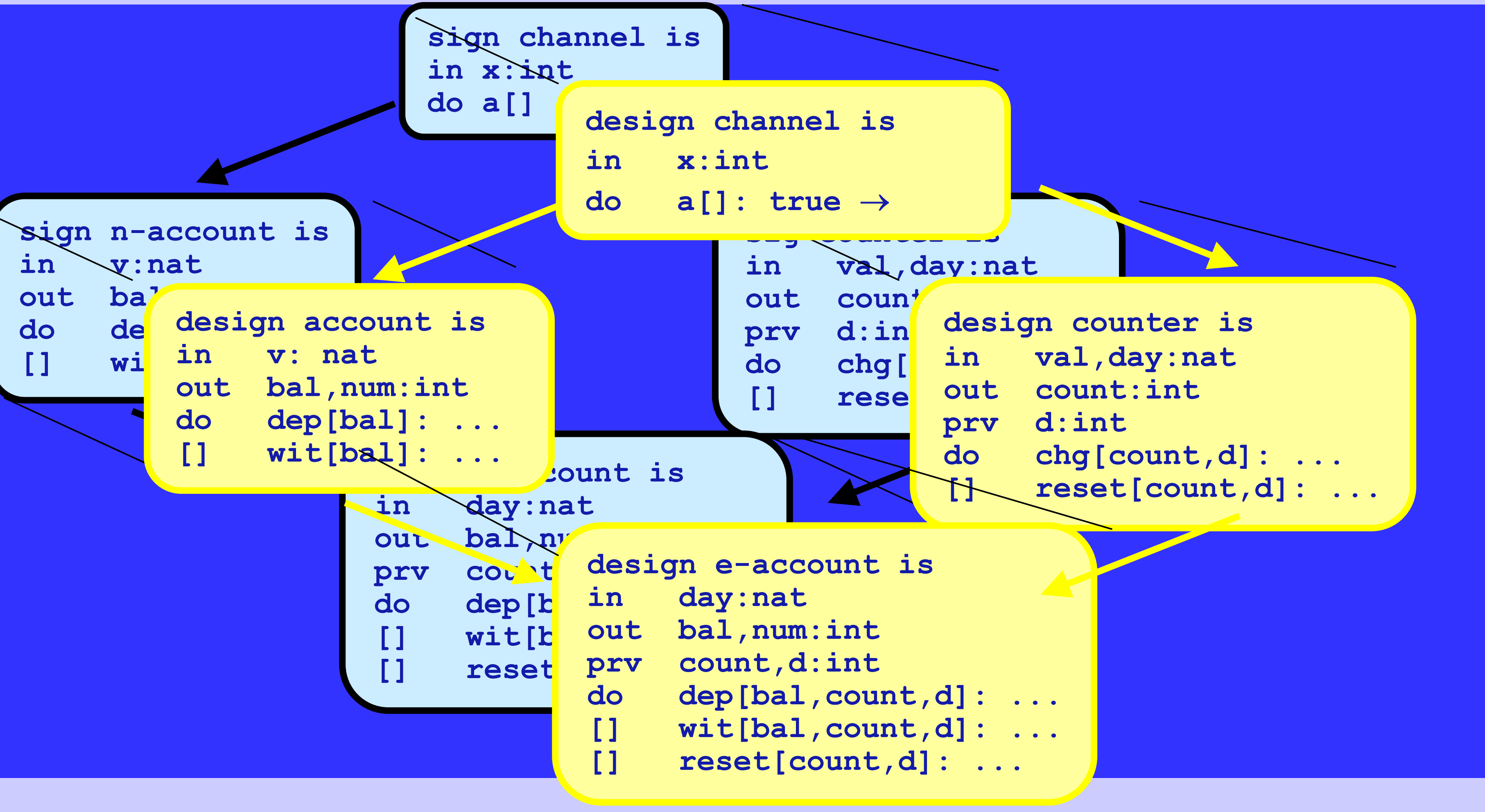
```
design vip-reg[C:nat] is
in   x:int,y:nat
do   a: x+C≥y, x≥y →
```

```
design vip-account[C:nat] is
in   v:nat
out  bal,num:int
do   dep: true → bal:=bal+v
[]   wit: bal+C≥v, bal≥v → bal:=bal-v
```

Separation of Coordination and Computation

The computational aspects do not play any role in the interconnection of systems components.

Separation of Coordination and Computation



```
sign channel is
in x:int
do a[]
```

```
design channel is
in   x:int
do   a[]: true →
```

```
sign n-account is
in    v:nat
out   bal
do    de
[]    wi
```

```
design account is
in    v: nat
out   bal,num:int
do    dep[bal]: ...
[]    wit[bal]: ...
```

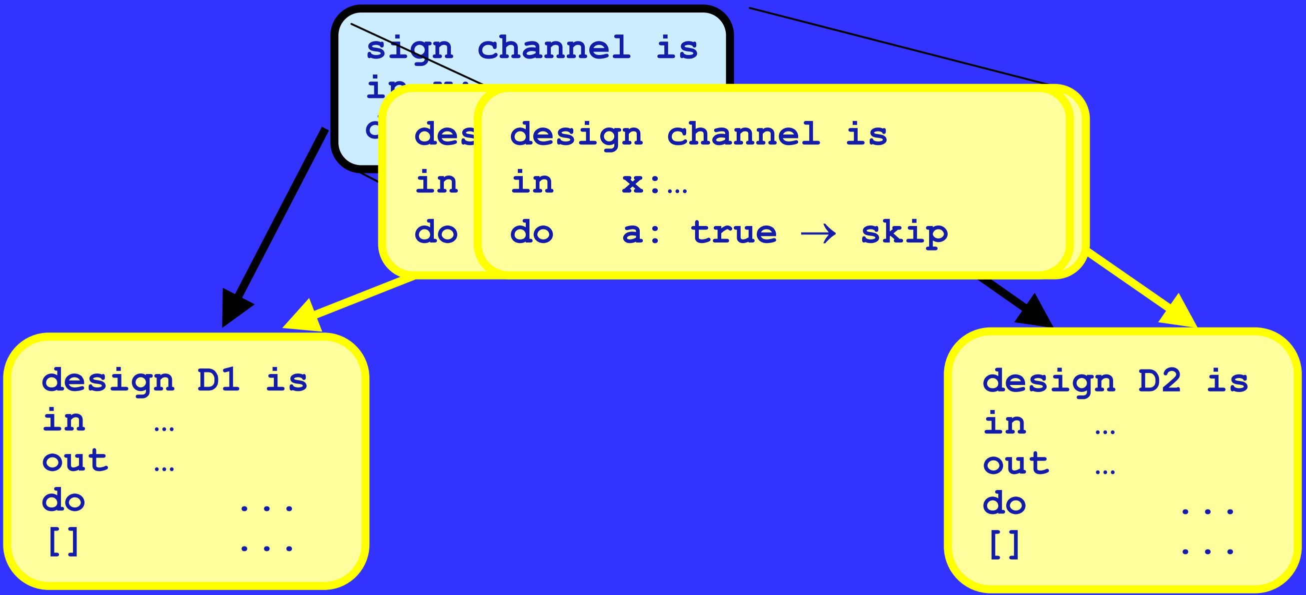
```
in    val,day:nat
out   count
prv   d:in
do    chg[
[]    rese
```

```
design counter is
in    val,day:nat
out   count:int
prv   d:int
do    chg[count,d]: ...
[]    reset[count,d]: ...
```

```
count is
in    day:nat
out   bal,nu
prv   cou t
do    dep[b
[]    wit[b
[]    reset
```

```
design e-account is
in    day:nat
out   bal,num:int
prv   count,d:int
do    dep[bal,count,d]: ...
[]    wit[bal,count,d]: ...
[]    reset[count,d]: ...
```

Separation of Coordination and Computation



```
sign channel is
```

```
des
in
do
```

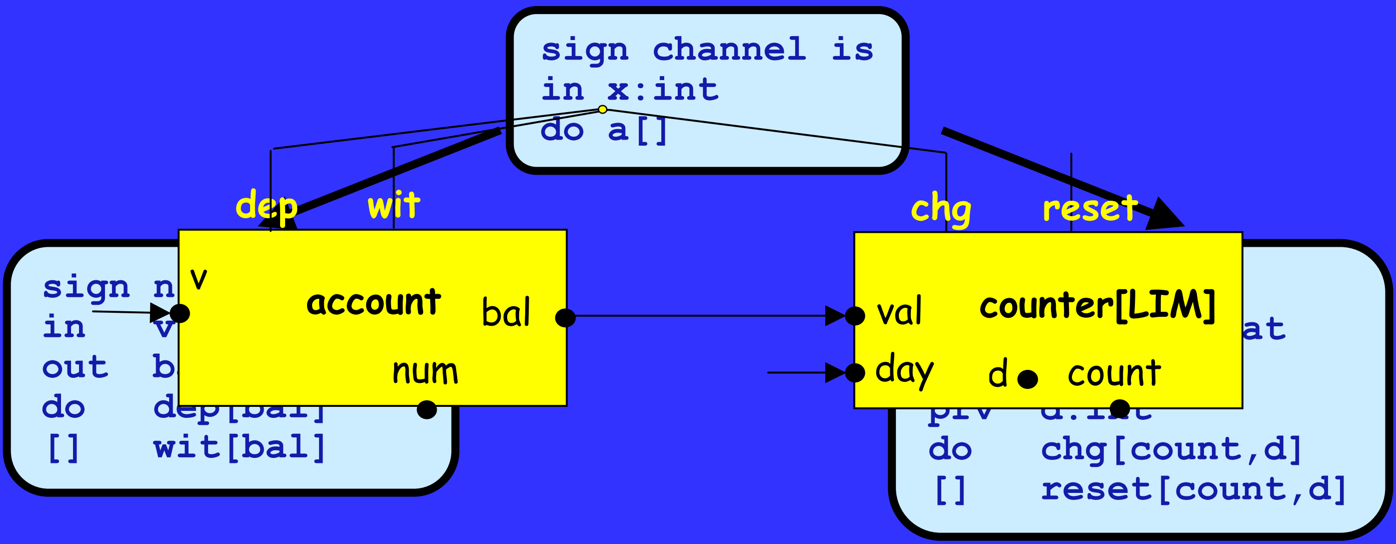
```
design channel is
in    x:…
do    a: true → skip
```

```
design D1 is
in    …
out   …
do          . . .
[]          . . .
```

```
design D2 is
in    …
out   …
do          . . .
[]          . . .
```

Separation of Coordination and Computation

Rather than using signatures and signature morphisms, a more user-friendly notation may be adopted



Separation of Coordination and Computation

What is the mathematics of this?

Externalise signatures/interfaces from designs through a functor **sig:DES→SIG** in a way that

- **sig** is **faithful**;
- **sig lifts colimits** of well-formed configurations;
- **sig** has **discrete structures**;
- given any pair of configuration diagrams **dia**$_1$, **dia**$_2$ s.t. **dia**$_1$**;sig=dia**$_2$**;sig**, either both are well-formed or both are ill-formed.

What does it mean?

Separation of Coordination and Computation

sig is faithful:

- **sig** is injective on morphisms;
  - This means that morphisms of designs cannot induce more relationships than those that can be established between their underlying signatures

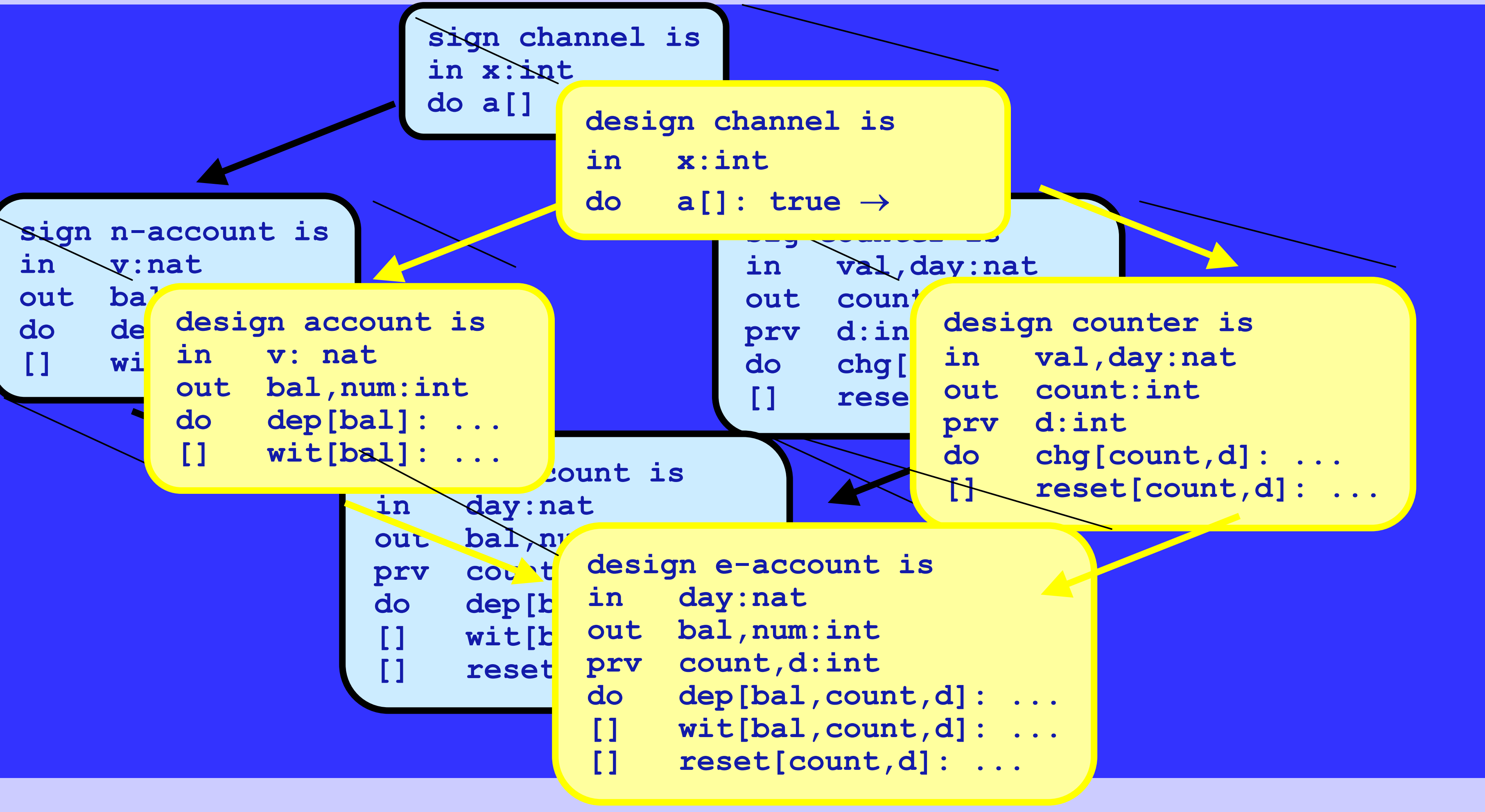
Separation of Coordination and Computation

sig lifts colimits of well-formed configurations;

Given any well-formed configuration expressed as a diagram **dia:I→DES** of designs and colimit $(\mathbf{sig}(S_i)\to\theta)_{i:I}$ of the underlying diagram of signatures, i.e. of $(\mathbf{dia};\mathbf{sig})$, there exists a colimit $(S_i\to S)_{i:I}$ of the diagram **dia** of designs whose signature part is the given colimit of signatures, i.e. $\mathbf{sig}(S_i\to S)=(\mathbf{sig}(S_i)\to\theta)$

This means that if we interconnect system components through a well-formed configuration, then any colimit of the underlying diagram of signatures establishes a signature for which a computational part exists that captures the joint behaviour of the interconnected components.

Separation of Coordination and Computation



```
sign channel is
in x:int
do a[]
```

```
design channel is
in   x:int
do   a[]: true →
```

```
sign n-account is
in   v:nat
out  bal
do   de
[]   wi
```

```
design account is
in   v: nat
out  bal,num:int
do   dep[bal]: ...
[]   wit[bal]: ...
```

```
in   val,day:nat
out  count
prv  d:in
do   chg[
[]   rese
```

```
design counter is
in   val,day:nat
out  count:int
prv  d:int
do   chg[count,d]: ...
[]   reset[count,d]: ...
```

```
count is
in   day:nat
out  bal,nu
prv  count
do   dep[b
[]   wit[b
[]   reset
```

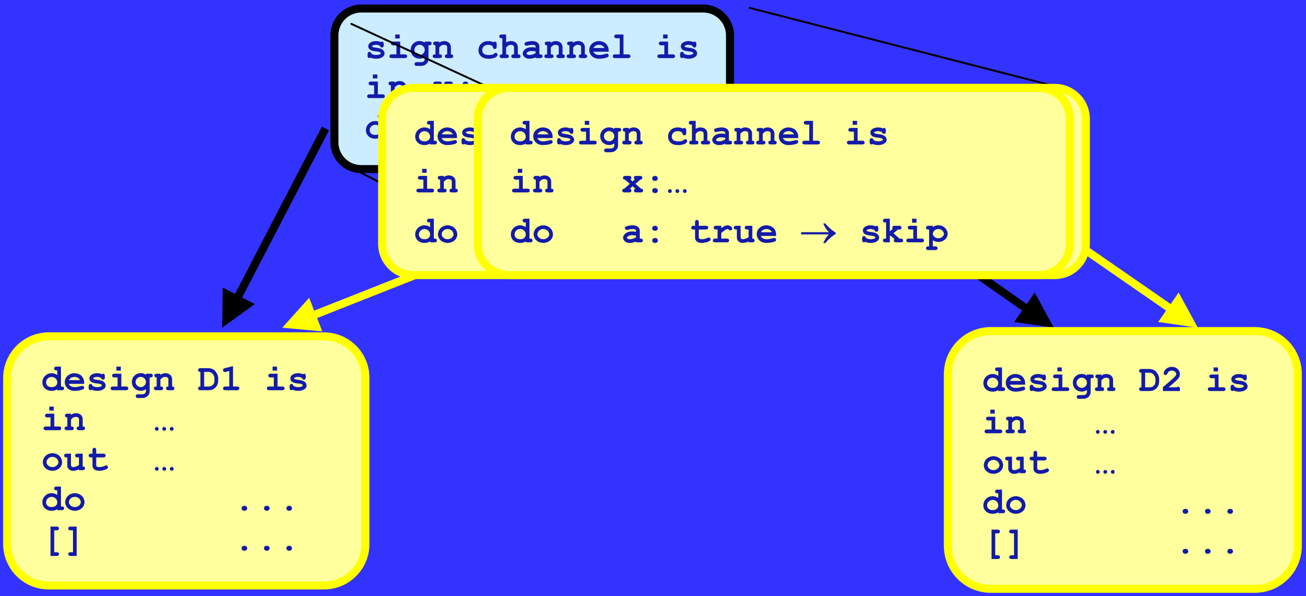
```
design e-account is
in   day:nat
out  bal,num:int
prv  count,d:int
do   dep[bal,count,d]: ...
[]   wit[bal,count,d]: ...
[]   reset[count,d]: ...
```

Separation of Coordination and Computation

sig has discrete structures;

- For every signature θ:**SIG**, there exists a design $d(\theta)$:**DES** such that, for every signature morphism $f:\theta\rightarrow$**sig**(S), there is a morphism $g:d(\theta)\rightarrow S$ in **DES** such that **sig**$(g)=f$.
- That is, every signature θ has a “realisation” (a discrete lift) as a design $d(\theta)$ in the sense that, using θ to interconnect a component S, which is achieved through a morphism $f:\theta\rightarrow$**sig**(S), is tantamount to using $d(\theta)$ through any $g:d(\theta)\rightarrow S$ s.t. **sig**$(g)=f$.
- Because **sig** is faithful, there is only one such *g*, which means that *f* and *g* are, essentially, the same. That is, sources of morphisms in diagrams of designs are, essentially, signatures.

Separation of Coordination and Computation



```
sign channel is
in
```

```
des
in
do
```

```
design channel is
in    x:…
do    a: true → skip
```

```
design D1 is
in    …
out   …
do          . . .
[]          . . .
```

```
design D2 is
in    …
out   …
do          . . .
[]          . . .
```

Separation of Coordination and Computation

given any pair of configuration diagrams **dia**$_1$, **dia**$_2$ s.t. **dia**$_1$**;sig=dia**$_2$**;sig**, either both are well-formed or both are ill-formed.

- This ensures that the criteria for well-formed configurations do not rely on the computational parts of descriptions.

Separation of Coordination and Computation

Categories **DES** for which there is a functor **sig:DES→SIG** atisfying the four given properties are said to be **oordinated over SIG**.

Which categories are coordinated?

- Processes over their alphabets;
- Theories over their signatures;
- All topological categories;
- …

rom simple to complex interaction protocols

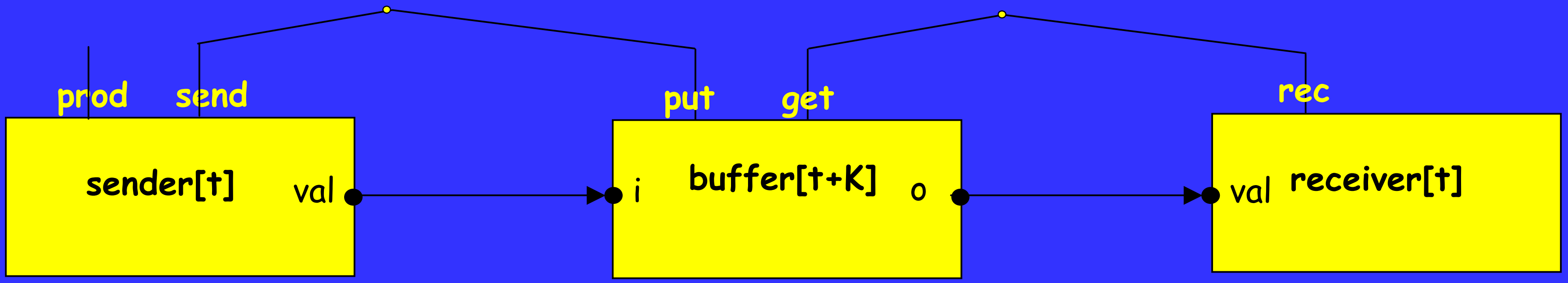
The configuration diagrams presented so far express **simple** and **static** interactions between component

- action synchronisation
- the interconnection of input variables of a component with output variables of other components

More complex interaction protocols can also be described by configurations...

Configurations: more examples

A generic sender and receiver of messages communicating asynchronously, through a bounded channel

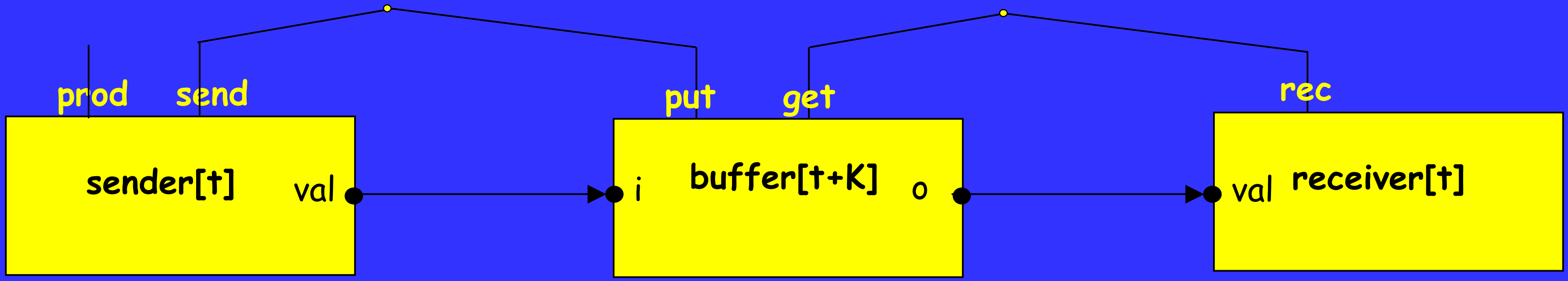


```
design sender[t] is
out   val:t
prv   rd:bool
do    prod[val,rd]:¬rd,false→rd’
[]    send[rd]:rd,false → ¬rd’
```

```
design receiver[t] is
in  val:t
do  rec:true,false→
```

Configurations: more examples

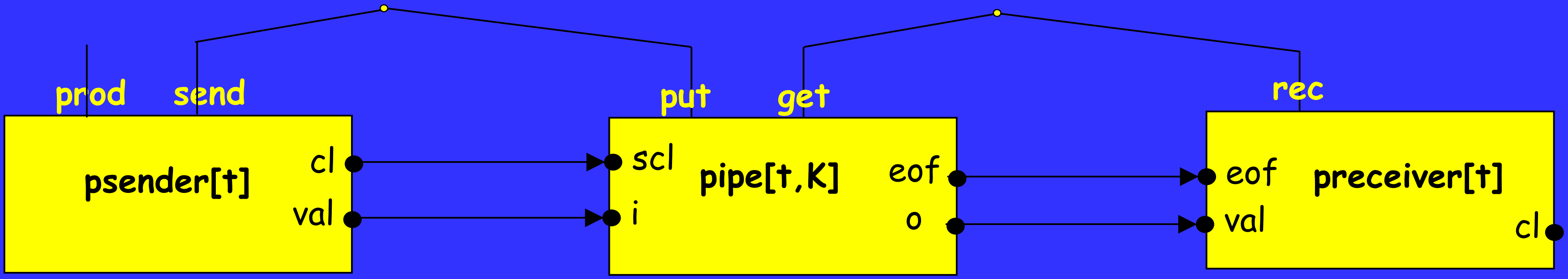
A generic sender and receiver of messages communicating asynchronously, through a bounded channel



```
design buffer[t,K:nat] is
in     i:t
out    o:t
prv    b:queue(K,t);rd:bool
do     put:¬full(q)→q:=enqueue(i,q)
[]prv  next:¬empty(q)∧¬rd →o:=head(q)||q:=tail(q)||rd:=true
[]     get:rd → rd:=false
```

Configurations: more examples

A generic sender and receiver of messages communicating through a pipe

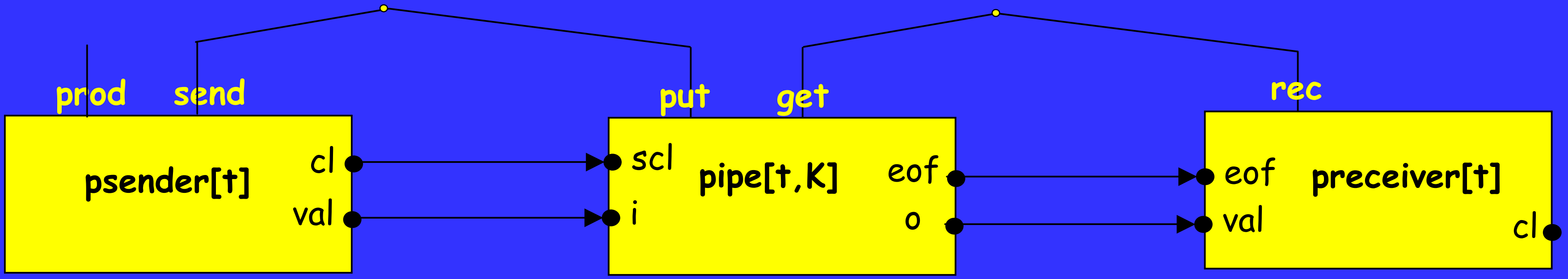


```
design psender[t] is
out    val:t, cl:bool
prv    rd:bool
do     prod[val,rd]:¬rd∧¬cl,
                 false→rd′
[]     send[rd]:rd,false → ¬rd′
[]prv close[cl]:¬rd∧¬cl,false→cl′
```

```
design preceiver[t] is
in     val:t, eof:bool
out    cl:bool
do     rec:¬eof∧¬cl,false→
[]prv  close:¬cl,¬cl∧eof →cl′
```

Configurations: more examples

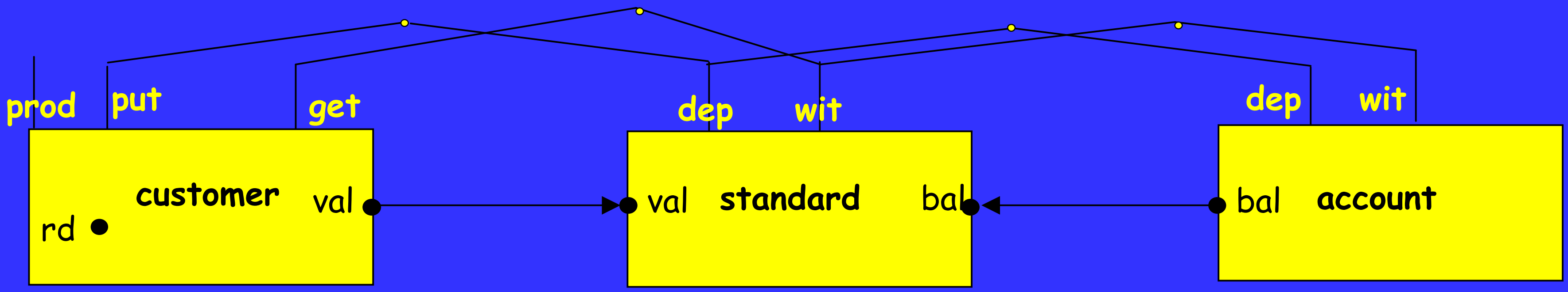
A generic sender and receiver of messages communicating through a pipe



```
design pipe[t,K:nat] is
in     i:t,scl:bool
out    o:t,eof:bool
prv    b:queue(K,t);rd:bool
do     put:¬full(q)→q:=enqueue(i,q)
[]prv  next:¬empty(q)∧¬rd →o:=head(q)||q:=tail(q)||rd:=true
[]     get:rd → rd:=false
[]prv  signal:scl∧empty(q)∧¬rd→eof:=true
```

nteraction protocols or Coordination Contracts

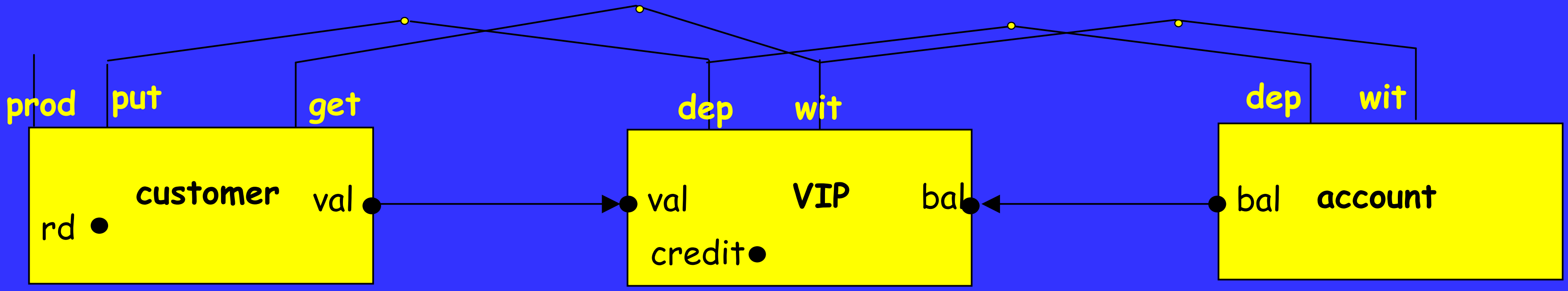
Customers may be subject to the standard rules for withdrawing money



```
design standard is
in  val: nat; bal: int
do  dep: true →
[]  wit: bal ≥ val →
```

Interaction protocols or Coordination Contracts

Customers may subscribe VIP-contracts that allow them to overdraw up to some limit as long as the average balance is greater than 1000.



```
design VIP is
in  val: nat; bal: int
prv credit:nat
do  dep: true →
[]  wit: bal+credit ≥ val →
```

Refinement

The refinement relationship between two designs can also be modelled as a morphism in a suitable category of designs.

A refinement morphism

$$\sigma : P_1 \rightarrow P_2$$

s intended to support the identification of a way in which a design P_1 is refined by P_2.

Refinement morphisms

A refinement morphism $\sigma : P_1 \rightarrow P_2$ consists of

- a total function $\sigma_{var} : V_1 \rightarrow \mathbf{Term}(V_2)$ s.t.
  - $sort_2(\sigma_{var}(v)) = sort_1(v)$
  - $\sigma_{var}(out(V_1)) \subseteq out(V_2)$
  - $\sigma_{var}(in(V_1)) \subseteq in(V_2)$
  - $\sigma_{var}(prv(V_1)) \subseteq Term(loc(V_2))$
- a partial mapping $\sigma_{ac} : \Gamma_2 \rightarrow \Gamma_1$ s.t.
  - $\sigma_{ac}(sh(\Gamma_2)) \subseteq sh(\Gamma_1)$
  - $\sigma_{ac}(prv(\Gamma_2)) \subseteq prv(\Gamma_1)$
  - $\sigma_{ac}^{-1}(g) \neq \emptyset,\ g \in sh(\Gamma_1)$
  - $\sigma_{var}(D_1(\sigma_{ac}(g))) \subseteq \underline{D}_2(g)$
  - $\sigma_{ac}(\underline{D}_2(\sigma_{var}(v))) \subseteq D_1(v),\ v \in loc(V_1)$

Sorts are preserved as well as the border between the component and its environment

Domains of vars are preserved
Every action that models interaction has to be implemented

Refinement morphisms

nd, moreover, for every g in Γ_2 s.t. $\sigma_{ac}(g)$ is defined

- $R_2(g) \supset \underline{\sigma}(R_1(\sigma_{ac}(g)))$
- $L_2(g) \supset \underline{\sigma}(L_1(\sigma_{ac}(g)))$

Effects of actions must be preserved or made more deterministic.
The interval defined by the safety and progress bounds of each action must be preserved or reduced

nd for every g_1 in Γ_1

- $\underline{\sigma}(U_1(g_1)) \supset \bigvee\{g_2 : \underline{\sigma}(g_2)=g_1\}\, U_2(g_2)$

Refinement of vip-account

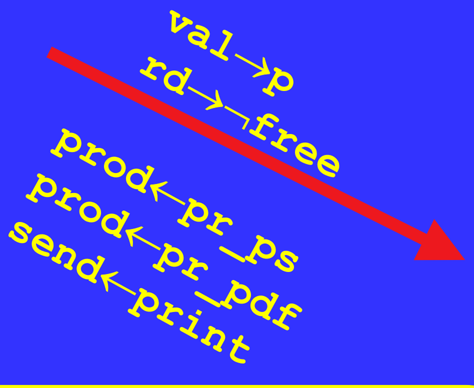
```
design vip-account[CRE:nat] is
out   num:nat; bal:int
in    v: nat
do    dep[bal]: true → bal′=v+bal
[]    wit[bal]: bal+CRE≥v, bal≥v → bal′=bal-v
```

inclusion

```
design vip-account2[CRE:nat] is
out   num:nat; bal:int
in    v,day,vip:nat
prv   d,sum,count:int
do    dep[bal,d,count,sum]: true → bal′=v+bal ∧ d′=day ∧
                count′=count+(day-d) ∧
                sum′=sum+bal*(day-d)
[]    wit[bal,d,count,sum]: bal≥v ∨ (bal+CRE≥v∧sum/count>vip)→
                bal′=bal-v ∧ d′=day ∧
                count′=count+(day-d) ∧
                sum′=sum+bal*(day-d)
[]    reset: true, false → count:=0||sum:=0||d:=day
```

vorduser - a refinement of sender

```
design sender(ps+pdf) is
out val:ps+pdf
prv rd:bool
do  prod[val,rd]:¬rd,false→rd′
[]  send[rd]:rd,false → ¬rd′
```

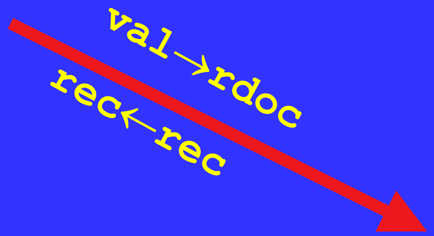


val→p
rd→¬free
prod←pr_ps
prod←pr_pdf
send←print

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

printer: a refinement of receiver

```
design receiver(ps+pdf) is
in   val:ps+pdf
do   rec[]:true,false→
```



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

Structuring systems vs Refinement

It is essential that

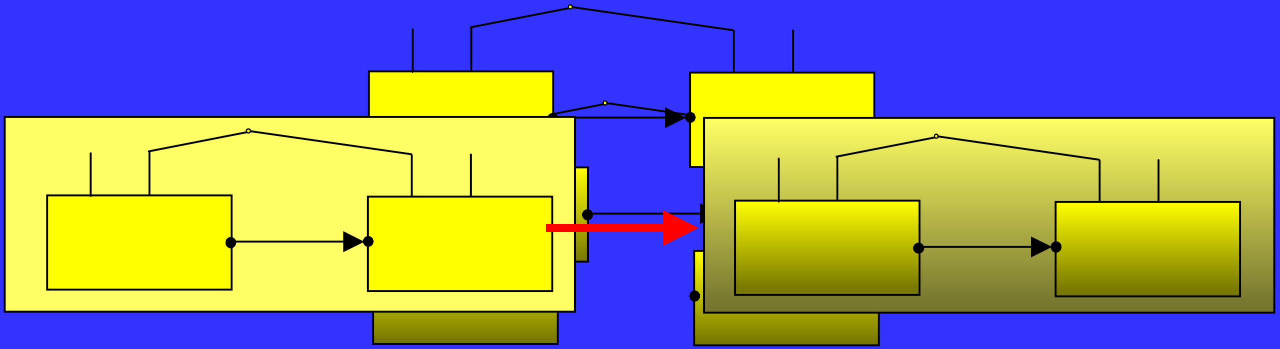
the gross modularisation of a system
in terms of
components and their interconnections

be “respected” when component designs are refined into
more concrete ones

Compositionality

Structuring systems vs Refinement

If the descriptions of the components of a system are refined into more concrete ones

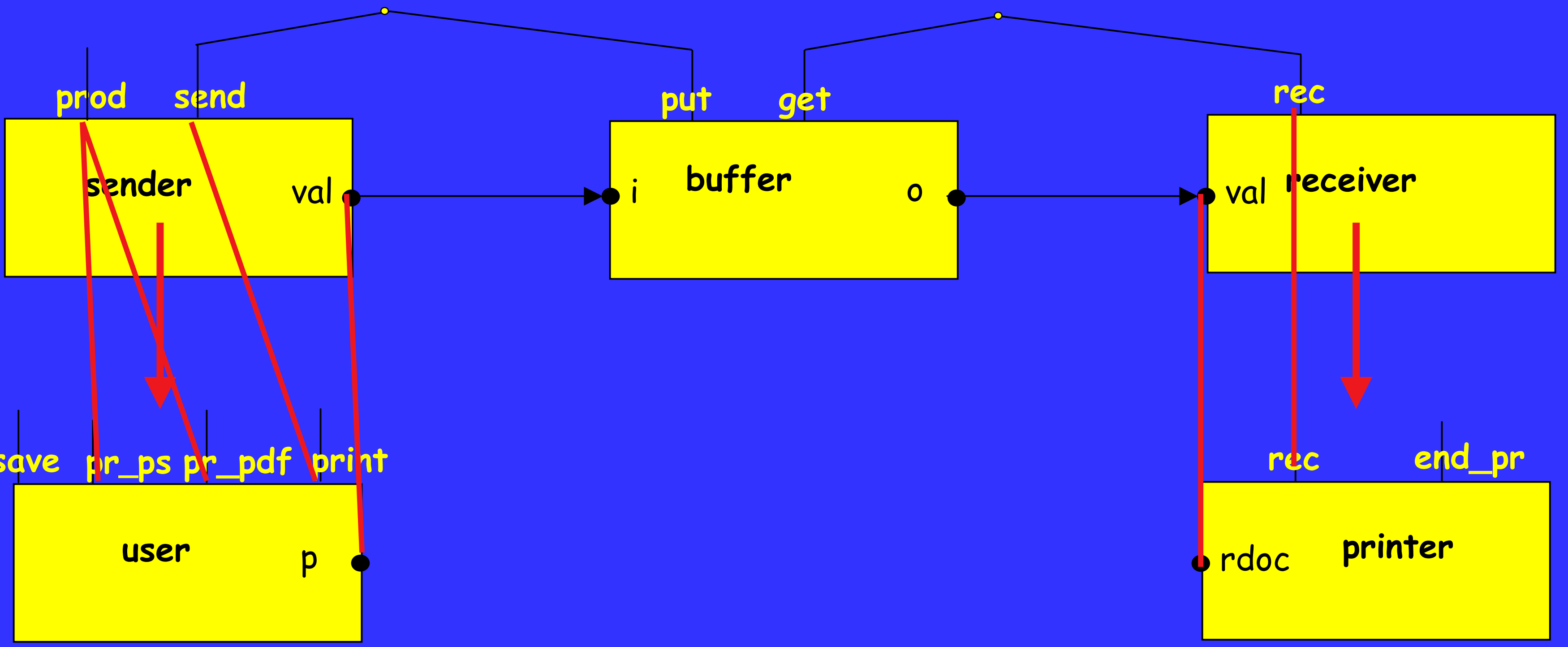


1. It is possible to propagate the interactions defined previously

2. The resulting description of the system refines the previous one

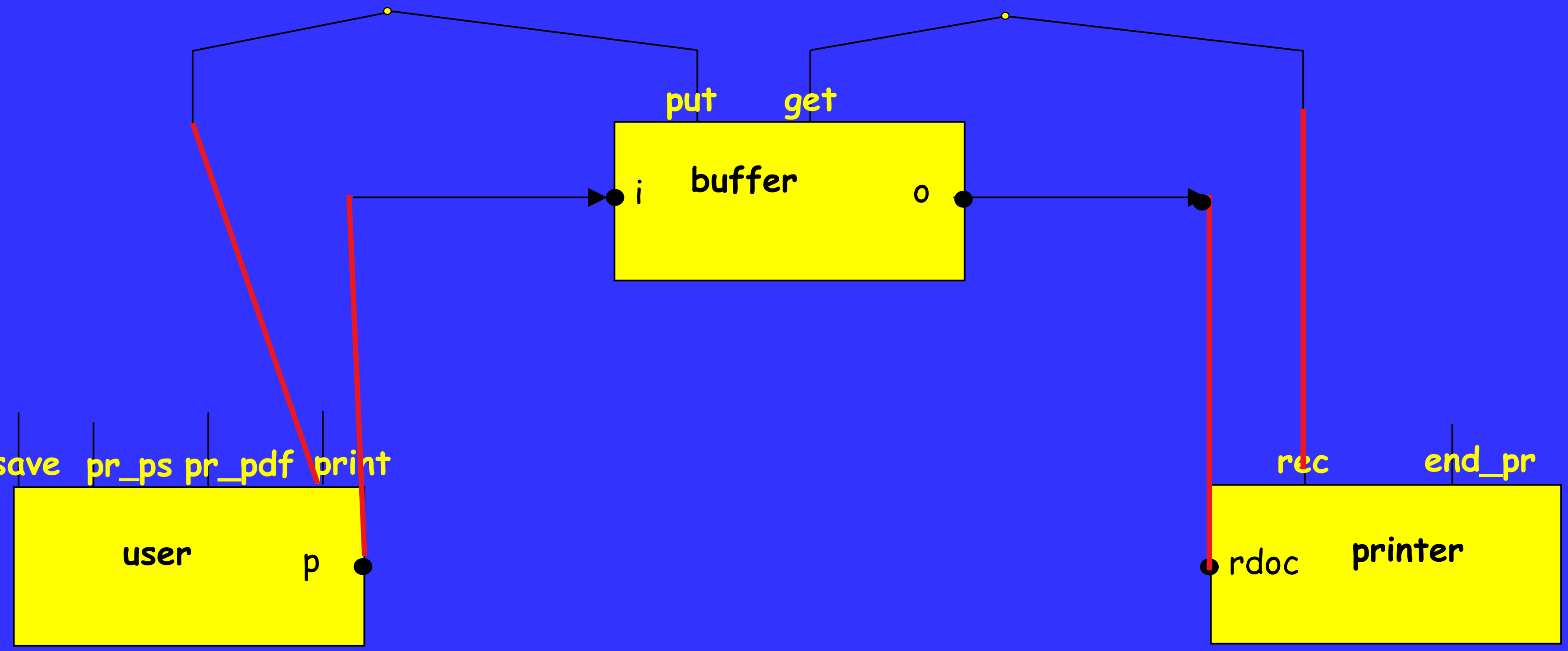
Structuring systems vs Refinement

Example

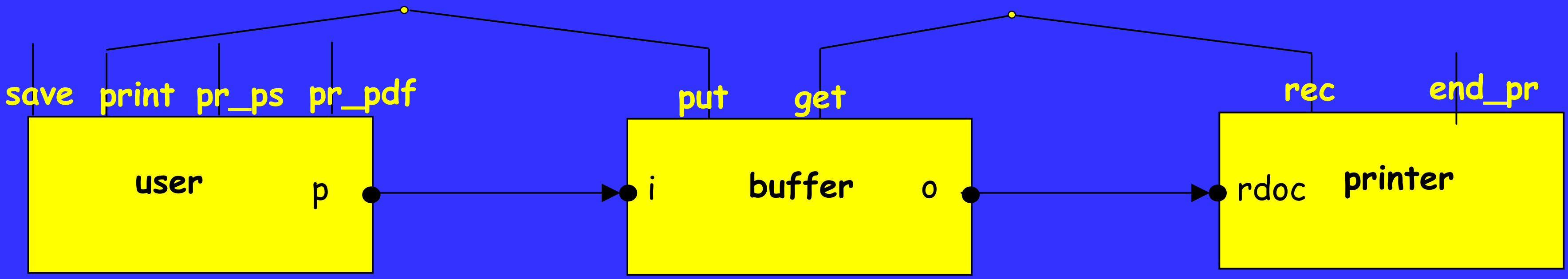

prod
send
sender
val
put
get
buffer
i
o
rec
val
receiver
save
pr_ps
pr_pdf
print
user
p
rec
end_pr
rdoc
printer

tructuring systems vs Refinement

xample



Structuring systems vs Refinement



ompositionality ensures that properties inferred from the more bstract description hold also for the more concrete (refined) one

g: **in order message delivery** does not depend on the speed at which nessages are produced and consumed

ystematizing Configurations

We have seen that

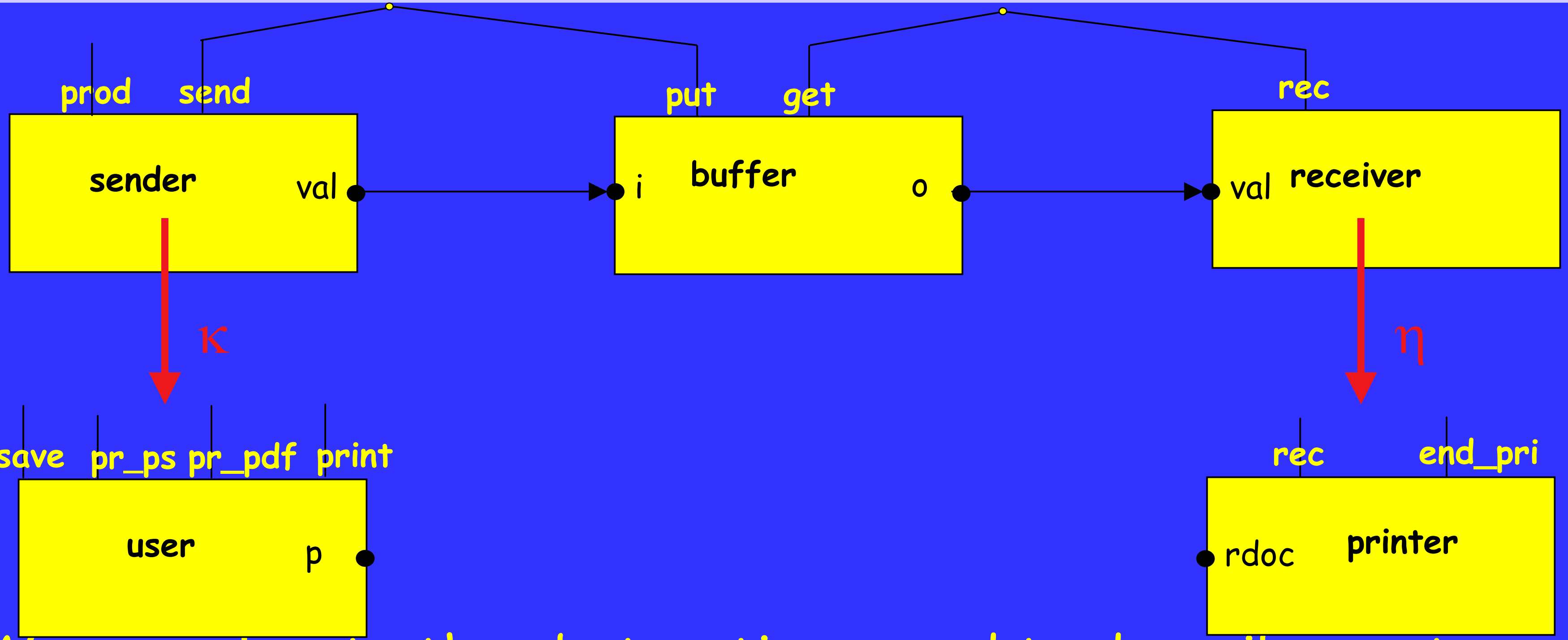
- Complex interaction protocols can be described by configurations, independently of the concrete components they will be applied to; they can be used in different contexts

  Connector Types

- The use of such interaction protocols in a given configuration corresponds to defining the way in which the generic participating components are refined by the concrete components

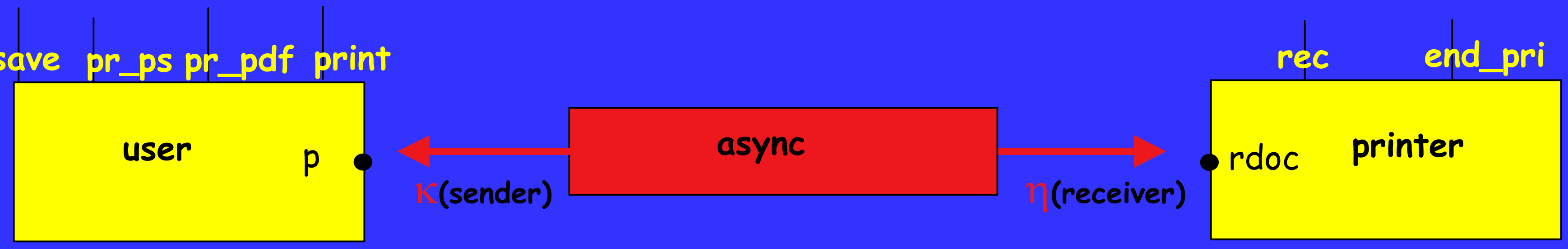
  Instantiation of Connectors

ystematizing Configurations



Ve may elevate the abstractions used to describe systems configurations...

Systematizing Configurations



. and define them in terms of computational components and connectors

3

Software Architectures

rchitectural Connectors

Interaction protocols can be described as **Connectors**

A connector consists of a configuration involving a **Glue** (design) and one or more **Roles** (designs):

- The roles describe the behaviour required of the components so that they can participate in the interaction (instantiate the roles);
- The glue describes how the activities of these components are coordinated in the intended protocol.

The **application of a connector** to given components of a system is defined by the instantiation of its roles. Role instantiation is modelled through refinement morphisms.

pplying CT to Software Architecture

he notions we presented for CommUnity can be generalised to other esign formalisms provided that they be presented by

- a category ***c-DESC*** of component descriptions in which configurations of systems of interconnected components are modelled through diagrams;
- a set *Conf(CD)* for every set of component descriptions *CD*, defining the well-formed configurations over CD;
- a category ***r-DESC*** with the same objects as ***c-DESC***, but in which morphisms model refinement

nd

define **an architectural school** in the following sense:

Architectural Schools

Coordination

Separation between coordination and computation materialised through a unctor

sig: c-DESC→SIG

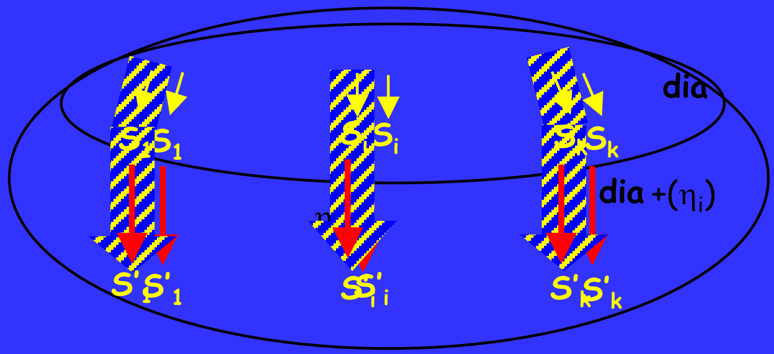
hat

- is faithful;
- lifts colimits of well-formed configurations;
- has discrete structures;
- given any pair of config. diagrams $\mathbf{dia}_1$, $\mathbf{dia}_2$ s.t. $\mathbf{dia}_1;\mathbf{sig}=\mathbf{dia}_2;\mathbf{sig}$, either both are well-formed or both are ill-formed.

Architectural Schools

Refinement and Compositionality

f the descriptions of the components of a system are refined into more oncrete ones

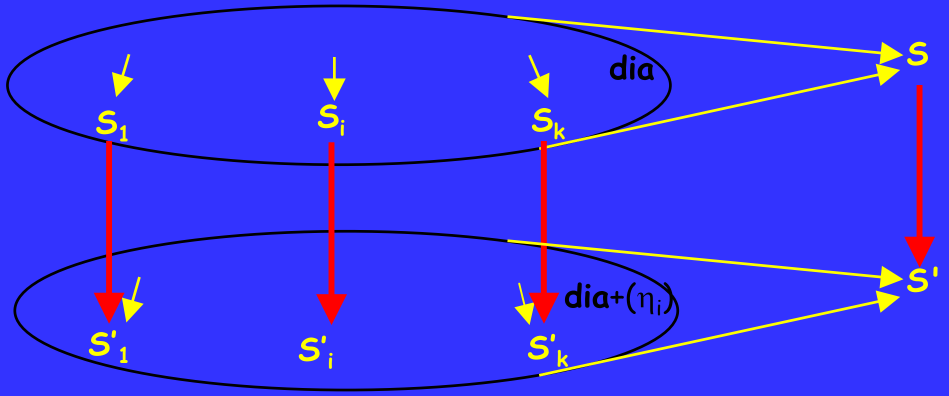


. It is possible to propagate the interactions defined previously

Architectural Schools

Refinement and Compositionality

f the descriptions of the components of a system are refined into more oncrete ones



. It is possible to propagate the interactions defined previously

2. The resulting description of the system is a refinement of the original one

Connectors

A connector is a well-formed configuration of the form

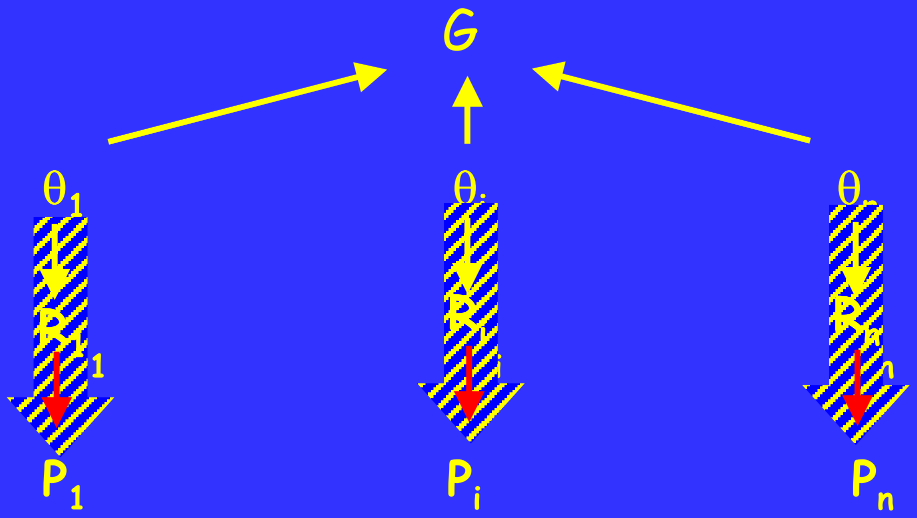
$$\begin{tikzcd}
 & & G & & \\
\theta_1 \arrow[urr] \arrow[d] & & \theta_i \arrow[u] \arrow[d] & & \theta_n \arrow[ull] \arrow[d] \\
\mathbf{R}_1 & & \mathbf{R}_i & & \mathbf{R}_n
\end{tikzcd}$$

G is the glue and **R**'s are the roles

Its semantics is given by the colimit of this diagram

Connectors - Instantiation

An instantiation of a connector consists of, for each of its roles R, a design P together with a refinement morphism ϕ:R→P



The semantics of a connector instantiation is the colimit of the diagram

Generalisations

This categorical framework provides

- an ADL-independent semantics for existing principles and techniques of SA
- a basis for extending the capabilities of existing ADLs.

Examples:

- Heterogeneous connectors
- Higher-order connectors

Heterogeneous Connectors

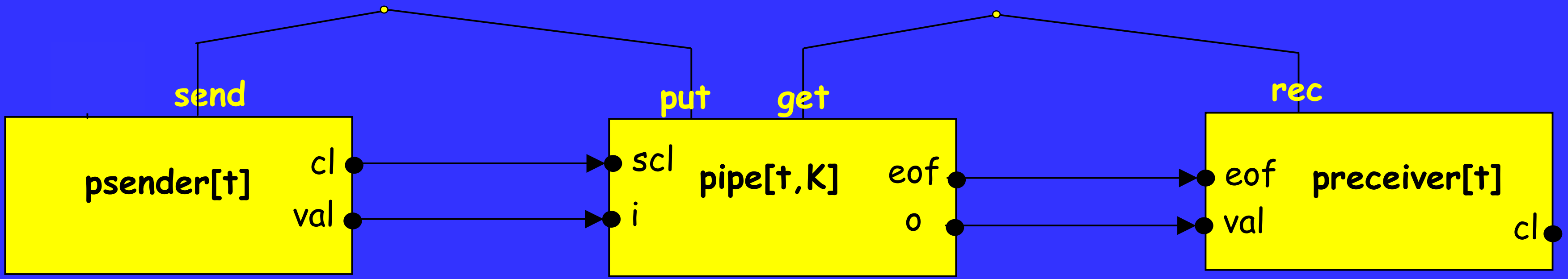
As defined previously, in connectors

- Roles are only used for defining which are the components admissible as instances.
- Correct instantiation defined by refinement morphisms

This justifies the adoption of a more declarative formalism for the specification of roles, giving rise to **Heterogeneous Connectors**

Heterogeneous Connectors

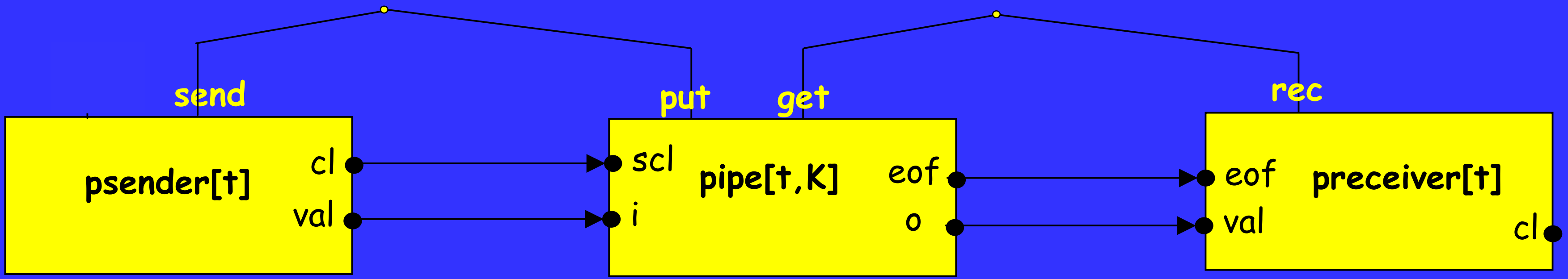
The pipe connector again...



```
spec psender[t] is
out        val:t, cl:bool
actions    send
axioms    cl⊃G(¬send∧cl)
```

Heterogeneous Connectors

The pipe connector again…



```
spec preceiver[t] is
in      val:t, eof:bool
out     cl:bool
actions rec
axioms  cl⊃G(¬rec∧cl)
       ((eof⊃Geof)∧(eof∧¬cl))⊃(¬recUcl)
```

Specifications

Specification

```
spec S is
in      in(V)
out     out(V)
actions Γ
axioms  Φ
```

- V: set of vars
- Γ: set of actions
- Φ: a set of propositions of linear temporal logic

A specification morphism σ:$S_1 \rightarrow S_2$ consists of

- a total function σ_{var}:$V_1 \rightarrow V_2$
- a partial mapping σ_{ac}:$\Gamma_2 \rightarrow \Gamma_1$ s.t.
  1. $\sigma_{var}(out(V_1)) \subseteq out(V_2)$
  2. . Φ_2 , $\underline{\sigma}$ (Φ_1)

Colimits in this category join the axioms of the component specs

Specifications

This category of specifications is also coordinated over a category of signatures, i.e., these signatures provide the means for interconnecting specifications.

Signatures of the form θ=<V,Γ> can be mapped into **specifications** as well as into **designs** and, hence, the interconnection of a role specification with a glue design is given by a pair of morphisms of the form

$$\begin{tikzcd} G & \boxed{\theta} \arrow[l] \arrow[r] & R \end{tikzcd}$$

Heterogeneous Connectors

For the instantiation of roles, we need a satisfaction relation , between design morphisms and specification morphisms

An instantiation of a connector consists of, for each of its roles, a design P together with a design morphism ϕ:dsgn(θ)$\rightarrow$P s.t.

$$\begin{tikzcd}
G & \mathrm{dsgn}(\theta) \arrow[l] \arrow[d] & \mathrm{spec}(\theta) \arrow[r] & R \\
& P & , &
\end{tikzcd}$$

eterogeneous Connectors

or CommUnity designs and LTL specifications

the satisfaction relation , between design morphisms and specification morphisms is based on a notion refinement between specifications and designs

- Part of the semantics of CommUnity designs can be encoded in LTL – Properties(P)
- P refines S iff there exists a signature morphism $\eta:\theta_S \to \theta_P$ s.t.

  Properties(P), η(Axioms(S))

$\pi:P \to P'$, $\sigma:S \to S'$ iff there exists refinements $\eta:\theta_S \to \theta_P$ and $\eta':\theta_{S'} \to \theta_{P'}$ s.t. at the signature level, commutes.

$$\begin{tikzcd}
P \arrow[r] \arrow[d] & P' \arrow[d] \\
S \arrow[r] & S'
\end{tikzcd}$$

Heterogeneous Connectors

Properties(P)

- $(g \supset L(g))$ for every $g \in \Gamma$
- $\bigvee_{g \in D(v)} g \vee (\mathbf{X}v=v))$ for every $v \in loc(V)$
- $(g \supset \tau(R(g))$ for every $g \in \Gamma$, where τ is a translation that replaces every primed variable v' by the term $(\mathbf{X}v)$
- $(\mathbf{GF}U(g) \supset \mathbf{GF}g)$ for every $g \in prv(\Gamma)$

Heterogeneous Connectors

The semantics of a heterogeneous connector

$$\begin{tikzcd}
 & & \text{properties}(G) & & \\
 & & G & & \\
\text{dsgn}(\theta_1) \arrow[urr] & & \text{dsgn}(\theta_i) \arrow[u] & & \text{dsgn}(\theta_n) \arrow[ull] \\
\text{spec}(\theta_1) \arrow[uuurr] \arrow[d] & & \text{spec}(\theta_i) \arrow[uuu] \arrow[d] & & \text{spec}(\theta_n) \arrow[uuull] \arrow[d] \\
R_1 & & R_i & & R_n
\end{tikzcd}$$

is given by the colimit of this specification diagram.

Higher-Order Connectors

Current level of support and understanding of connectors is still insufficient, far from the one components have

Need further steps for a systematic construction of new connectors from existing ones

- Promote reuse
- Promote incremental and compositional development
- Make it easier to address complex interactions

Higher-Order Connectors

A specification mechanism that allows

- independent aspects of interaction protocols to be specified separately

  e.g., compression, fault-tolerance,

  security, monitoring

- composed and integrated in existing connectors

A connector that takes a connector as a parameter describing the capabilities that must be superposed over the instantiation of the parameter

Higher-Order Connectors

Higher-Order Connector =

connector (body) + connector (formal parameter)

- The **body** models the nature of the service that is superposed on instantiation of the formal parameter
- The **formal parameter** describes the kind of connector to which that service can be applied

Example: Monitoring of messages in a unidirectional communication

Using a Higher-Order Connector

- A hoc can be applied to any connector that instantiates its formal parameter, giving rise to a connector with the new capabilities

igher-Order Connectors: An example

Installing a compress/decompress service over a unidirectional communication protocol:

- modify **Uni-comm** in a way that messages are compressed for transmission without intruding over the original connection
- the outgoing messages should be compressed before they are put into the buffer and decompressed when they are removed from the buffer, before being delivered to the receiver

Higher-Order Connectors: Example

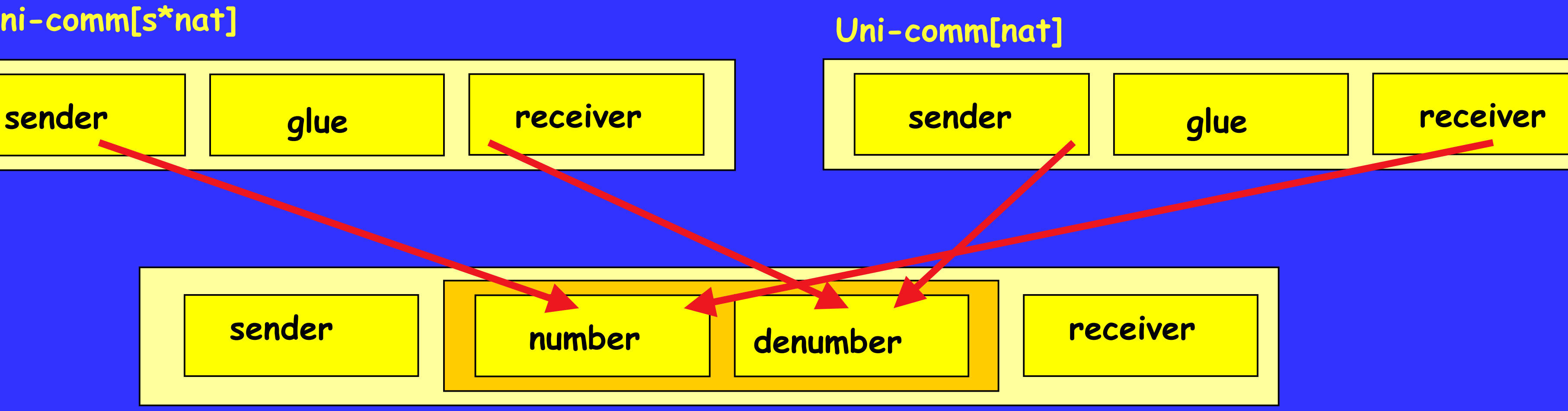
FT service that provides in-order message delivery in the presence of message-loss and duplication faults:

- numbers the messages sent by the sender; sends each numbered message until the corresponding ack is received; keeps pending messages in a queue
- sends acks for every received message; ignores the received (numbered) messages out of order and transmits the others to the receiver (not numbered anymore)

modelled by a HOC with two connector parameters:

- transmission of numbered messages
- transmission of acks (in the opposite direction)

Higher-Order Connectors



ni-comm[s*nat]

sender glue receiver

Uni-comm[nat]

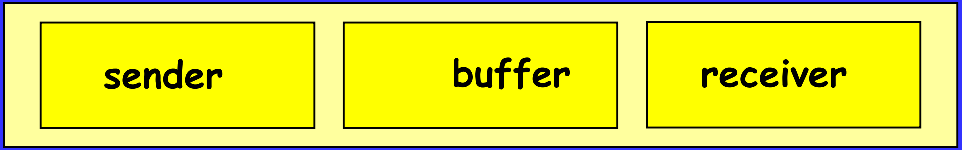
sender glue receiver

sender number denumber receiver

- **number**: sends repeatedly a numbered message until the corresponding ack is received and keeps pending messages in a queue
- **denumber**: sends acks for every received message, ignores the messages out of order and transmits the other to the receiver

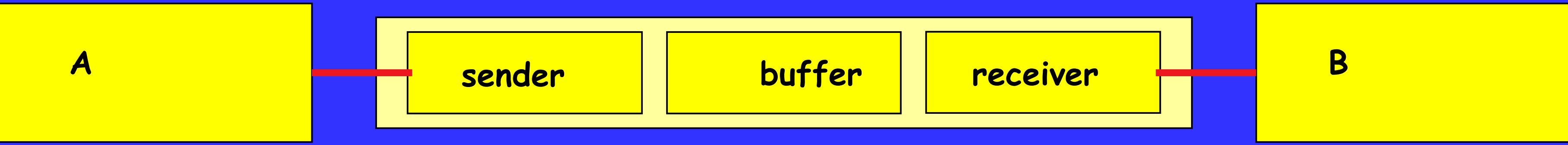
An example

Asynchronous communication through a bounded channel can be represented by a connector ***Async***



with two roles —sender and receiver. The glue is a bounded buffer with a FIFO discipline.

Components A and B connected through ***Async***

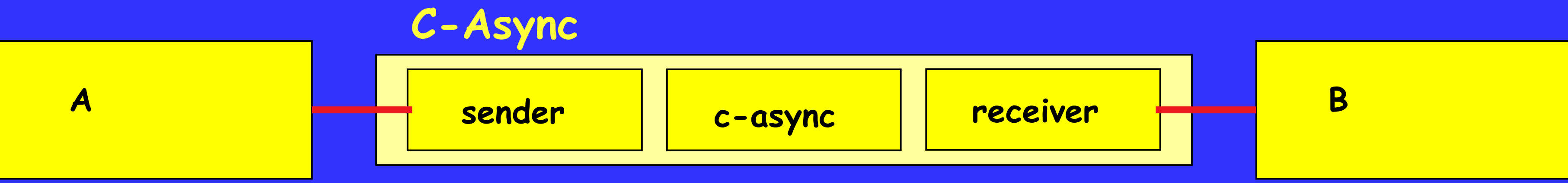


An example

Suppose that the information transmitted from A to B must be compressed.

Two alternatives:

- develop from scratch a new connector **C-Async** with the same roles but a new glue
- obtain a new connector **C-Async** by installing a compress/decompress service over **Async**



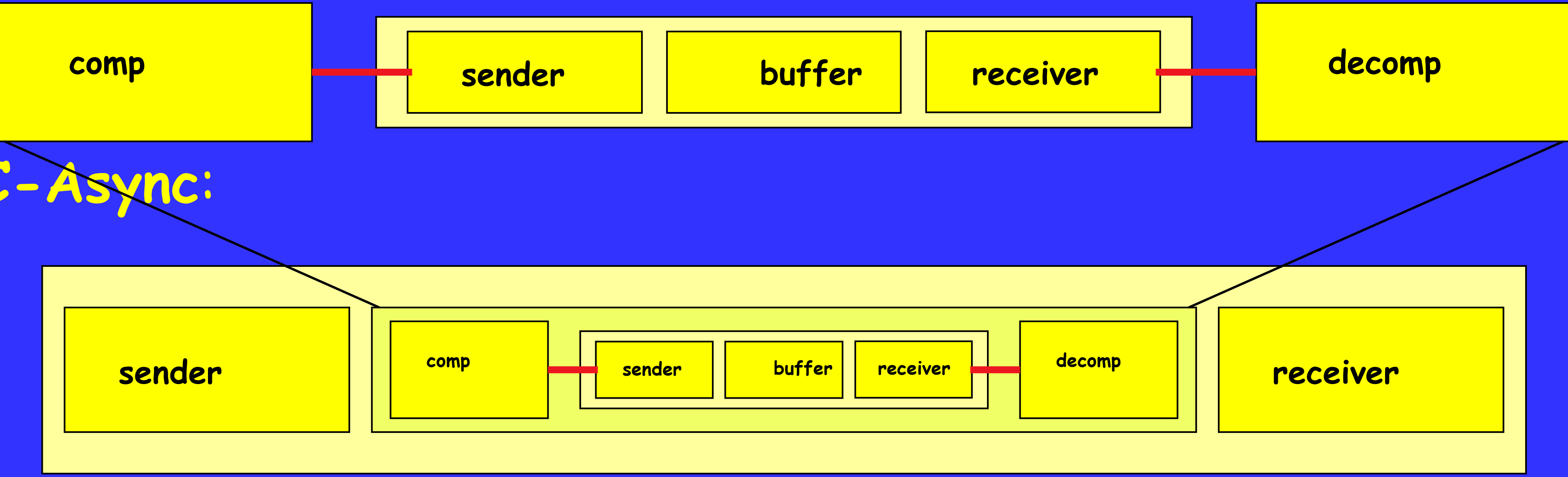
An example

Installing a compress/decompress service over **Async**:

- modify **Async** in a way that messages are compressed for transmission without intruding over the original connection
- the outgoing messages should be compressed before they are put into the buffer and decompressed when they are removed from the buffer, before being delivered to the receiver

An example

This form of coordination can be obtained by instantiating **Async** with a component **comp** in the role of sender and **decomp** in the role of receiver



C-Async:

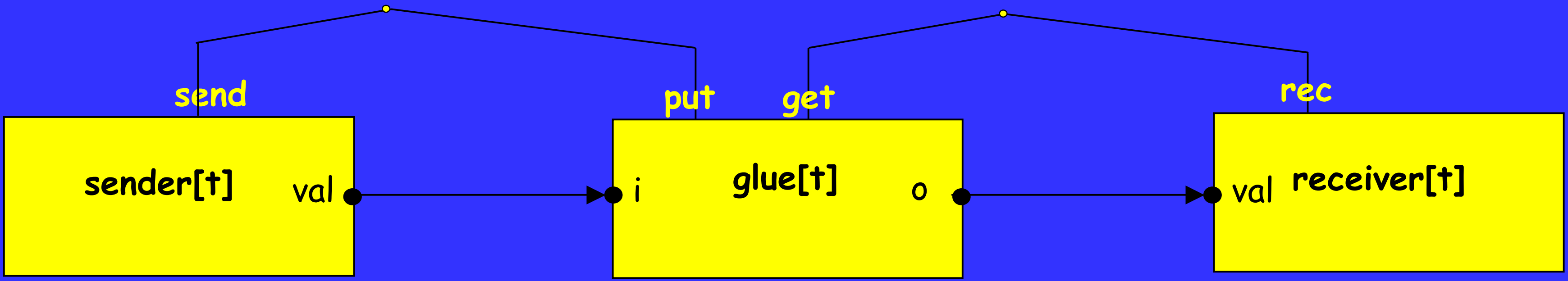
An example

The procedure for installing the compress/ decompress service can be applied to other connectors

The service itself can be modelled as a higher-order connector **Compression** and the installation of the service over a given connector can be obtained by a suitable instantiation of its parameter

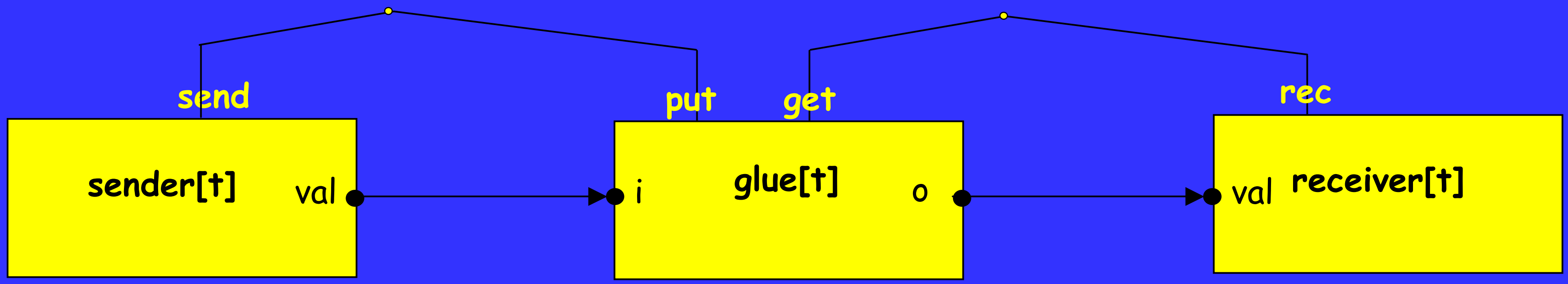
The Compression Hoc: formal parameter

- The **formal parameter** is the connector **Uni-comm[t]** modelling a generic unidirectional communication protocol



The Compression Hoc: formal parameter

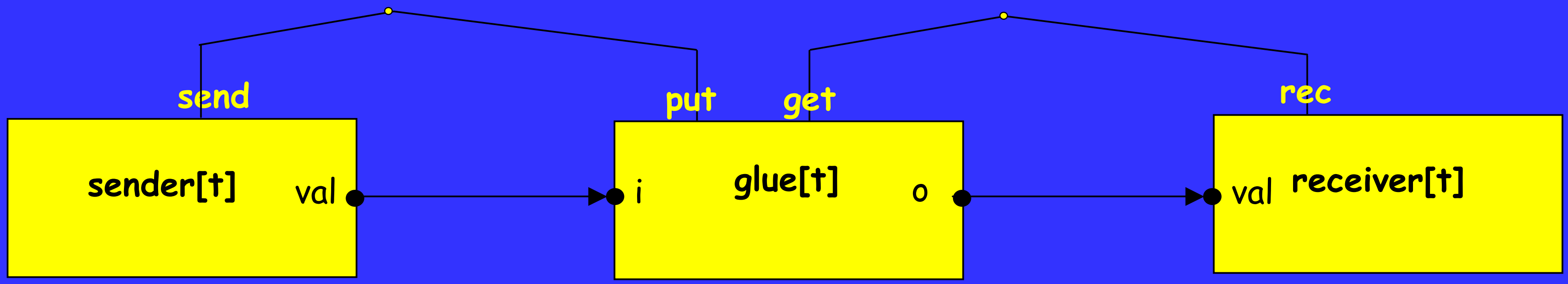
..The **formal parameter** is the connector **Uni-comm[t]** modelling a generic unidirectional communication protocol



```
design sender[t] is
out    val:t
prv    rd:bool
do prv prod:¬rd,false→rd:=true||val:∈t
[]     send:rd,false → rd:=false
```

The Compression Hoc: formal parameter

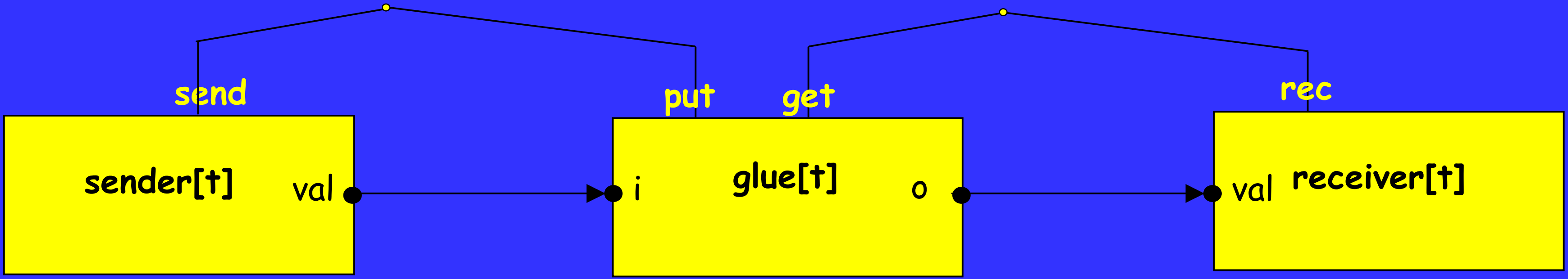
The **formal parameter** is the connector **Uni-comm[t]** modelling a generic unidirectional communication protocol



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

The Compression Hoc: formal parameter

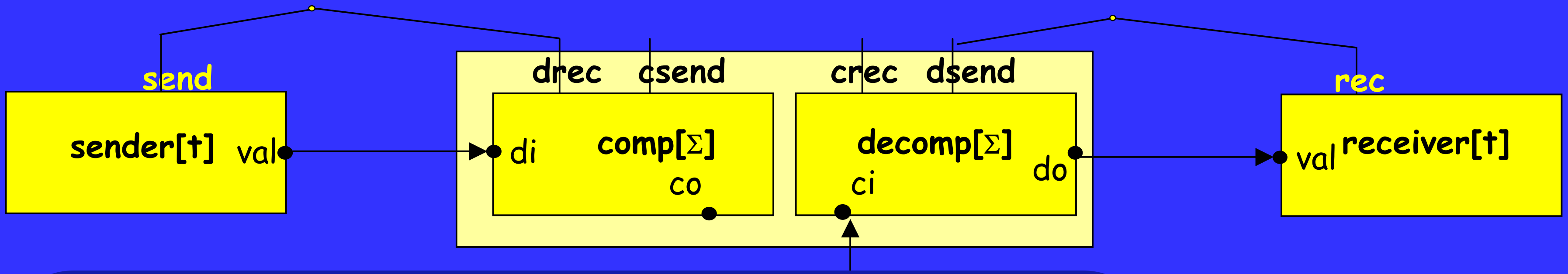
- The **formal parameter** is the connector **Uni-comm[t]** modelling a generic unidirectional communication protocol



```
design glue[t] is
in     i:t
out    o:t
do     put:true,false→skip
[]prv  prod: true,false→o:∈t
[]     get: true,false→skip
```

The Compression Hoc: body connector

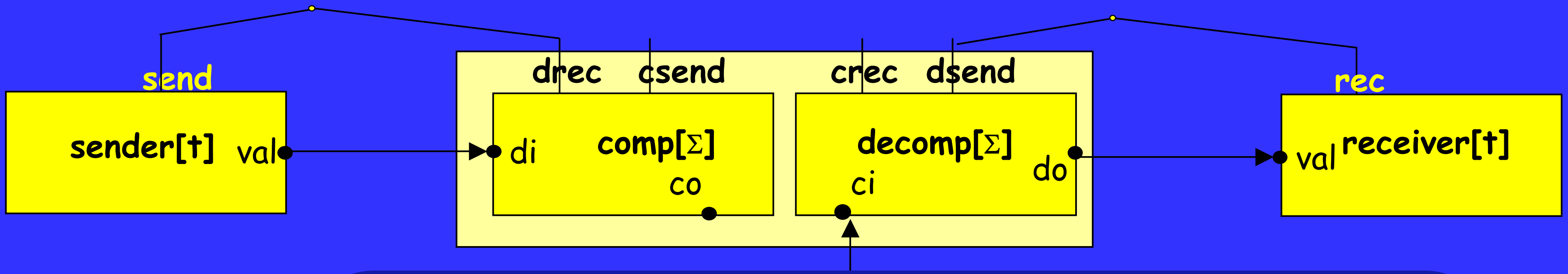
2. The body connector is Compression[Σ]



```
design comp[Σ] is
in     di:t
out    co:s
prv    v:t; rd,msg:bool
do     drec: ¬msg →v:=di||msg:=true
[] prv comp:¬rd∧msg→co:=comp(v)||rd:=true
[]     csend:rd →rd:=false||msg:=false
```

The Compression Hoc: body connector

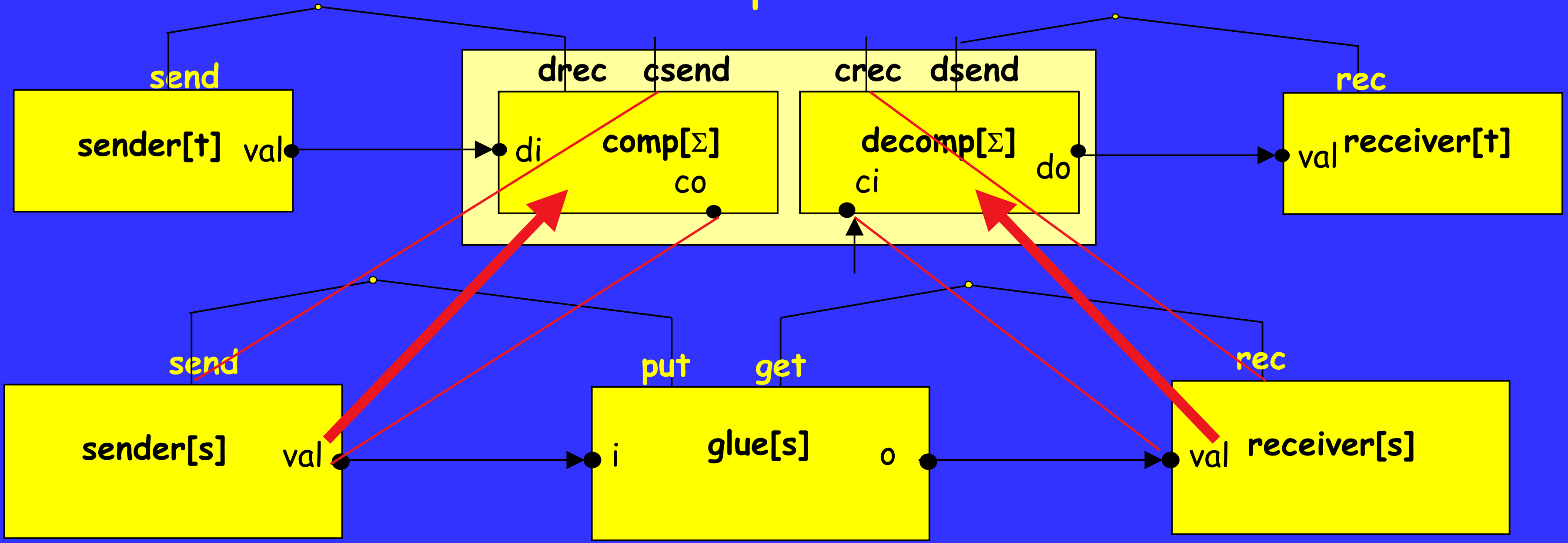
2. The body connector is Compression[Σ]



```
design decomp[Σ] is
in     ci:s
out    do:t
prv    v:s; rd,msg:bool
do     crec: ¬msg →v:=ci||msg:=true
[] prv dec:¬rd∧ msg→do:=decomp(v)||rd:=true
[]     dsend: rd →rd:=false||msg:=false
```

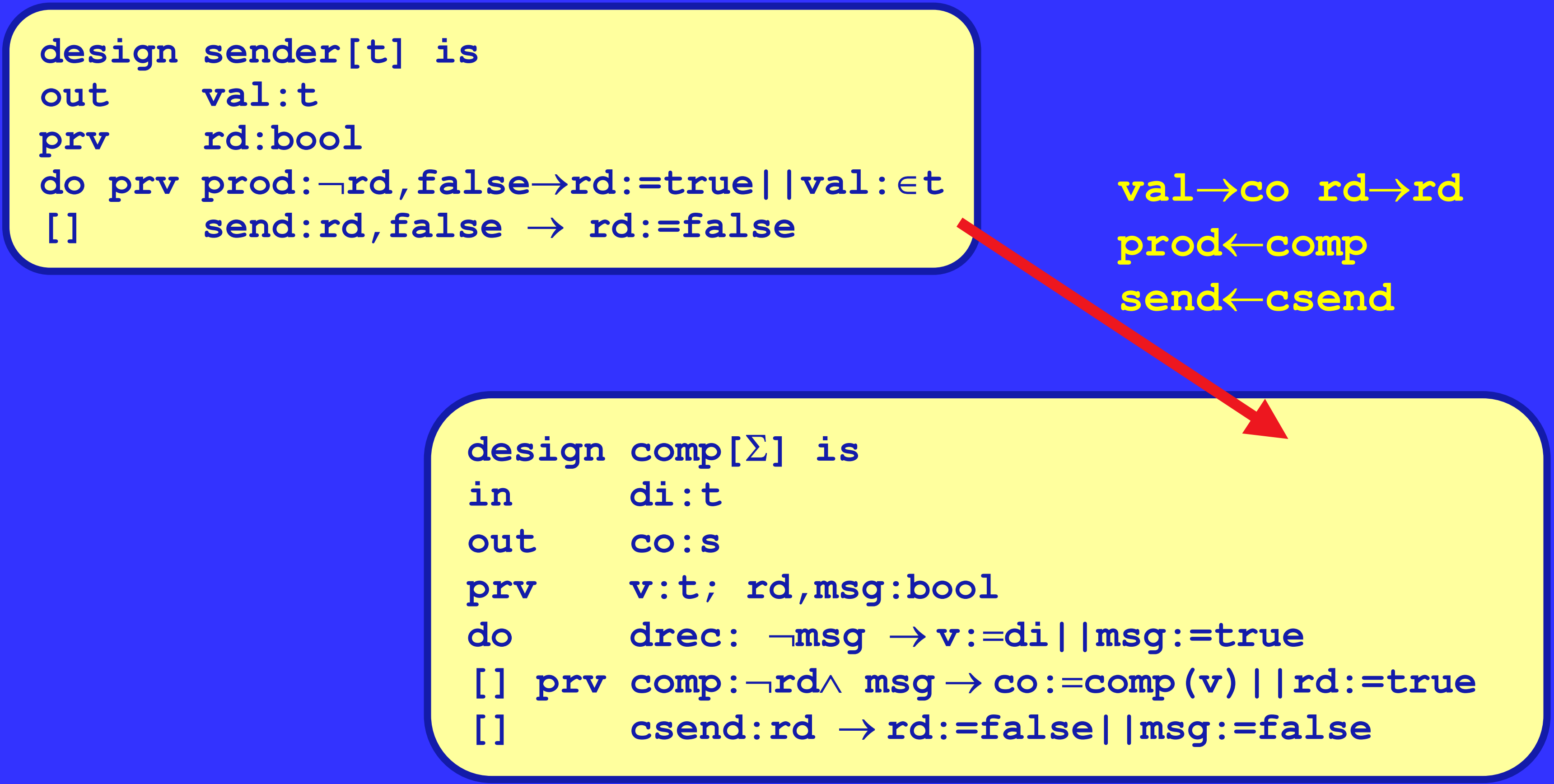
The Compression Hoc: relating the parameter and the body connector

3. The refinement relationships



establishing the instantiation of **Uni-comm[s]** with comp and decomp

The Compression hoc in Community



```
design sender[t] is
out    val:t
prv    rd:bool
do prv prod:¬rd,false→rd:=true||val:∈t
[]     send:rd,false → rd:=false
```

val→co rd→rd
prod←comp
send←csend

```
design comp[Σ] is
in     di:t
out    co:s
prv    v:t; rd,msg:bool
do     drec: ¬msg →v:=di||msg:=true
[] prv comp:¬rd∧ msg→co:=comp(v)||rd:=true
[]     csend:rd →rd:=false||msg:=false
```

Categorical Semantics of HOCs

A hoc consists of

formal parameter:

pC

$$\begin{tikzcd}
 & & pG & & \\
p\theta_1 \arrow[urr] \arrow[d] & & p\theta_i \arrow[u] \arrow[d] & & p\theta_k \arrow[ull] \arrow[d] \\
pR_1 & & pR_i & & pR_k
\end{tikzcd}$$

Categorical Semantics of HOCs

A hoc consists of

$$\begin{tikzcd}
 & pG & \\
p\theta_1 \arrow[ur] \arrow[d] & p\theta_i \arrow[u] \arrow[d] & p\theta_k \arrow[ul] \arrow[d] \\
pR_1 & pR_i & pR_k
\end{tikzcd}$$

pC

body connector:

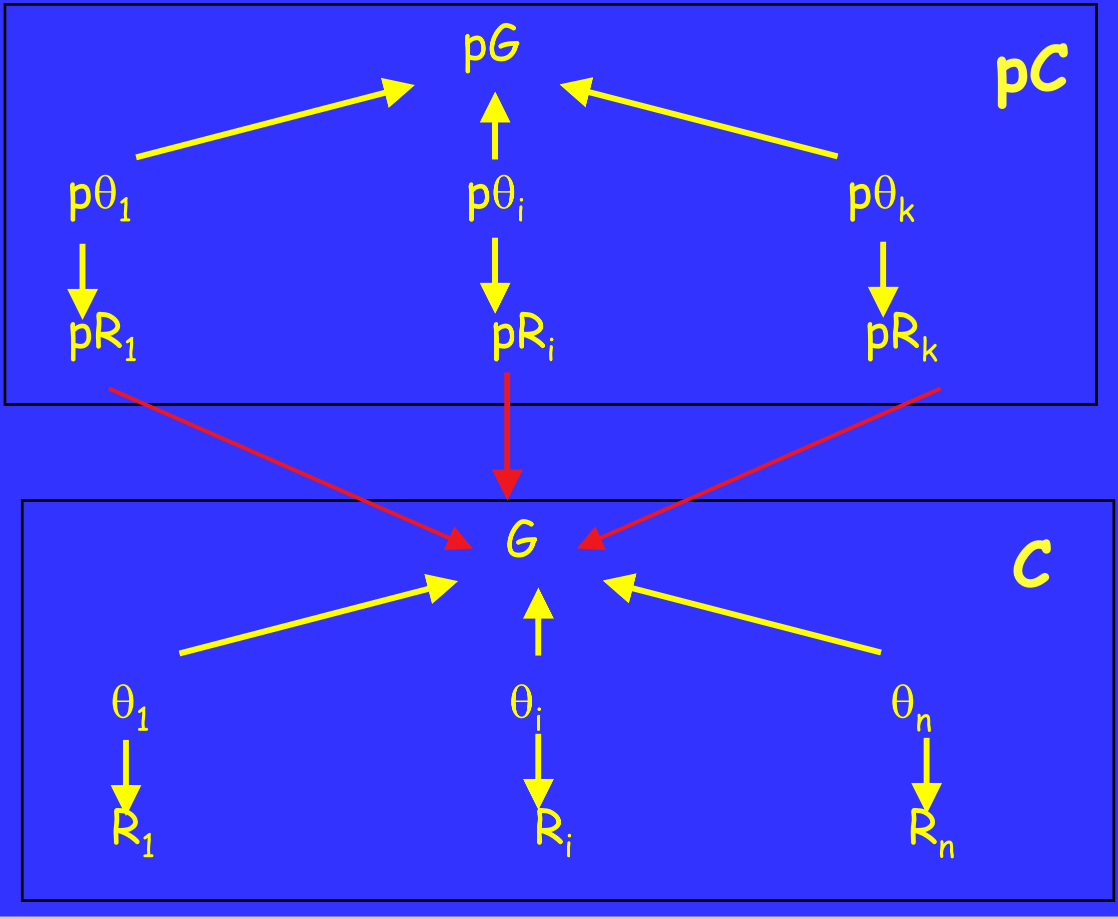
$$\begin{tikzcd}
 & G & \\
\theta_1 \arrow[ur] \arrow[d] & \theta_i \arrow[u] \arrow[d] & \theta_n \arrow[ul] \arrow[d] \\
R_1 & R_i & R_n
\end{tikzcd}$$

C

Categorical Semantics of HOCs

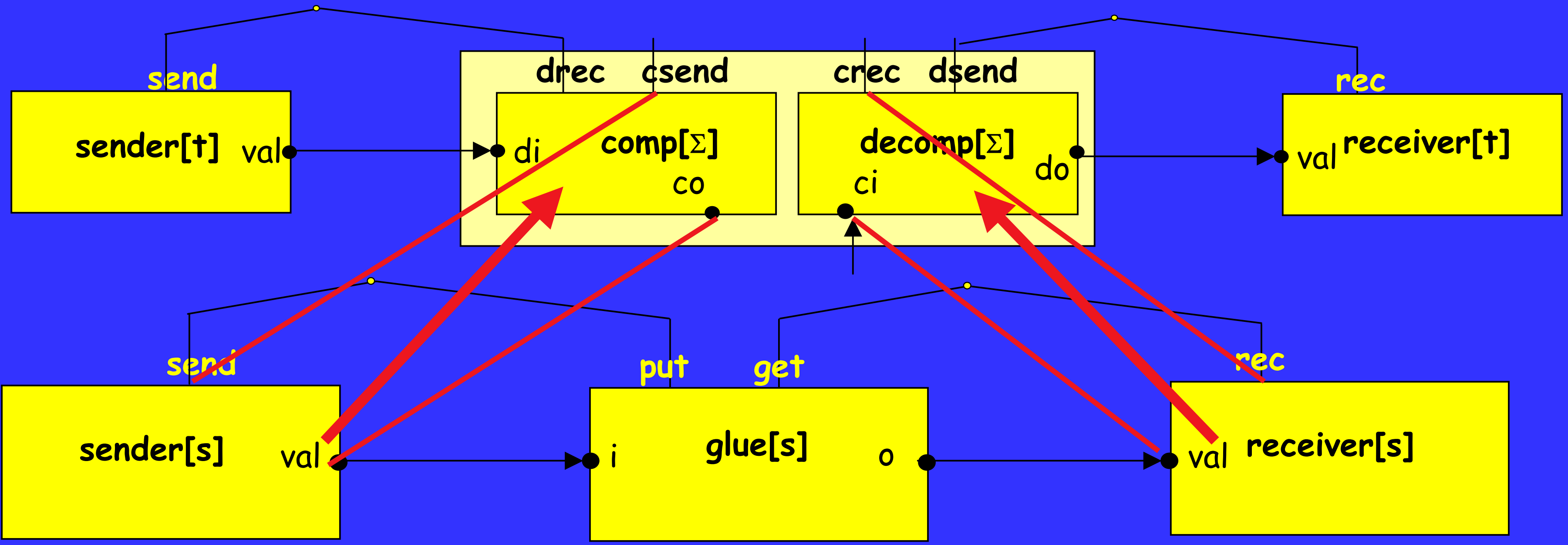
A hoc consists of

efinement morphisms:

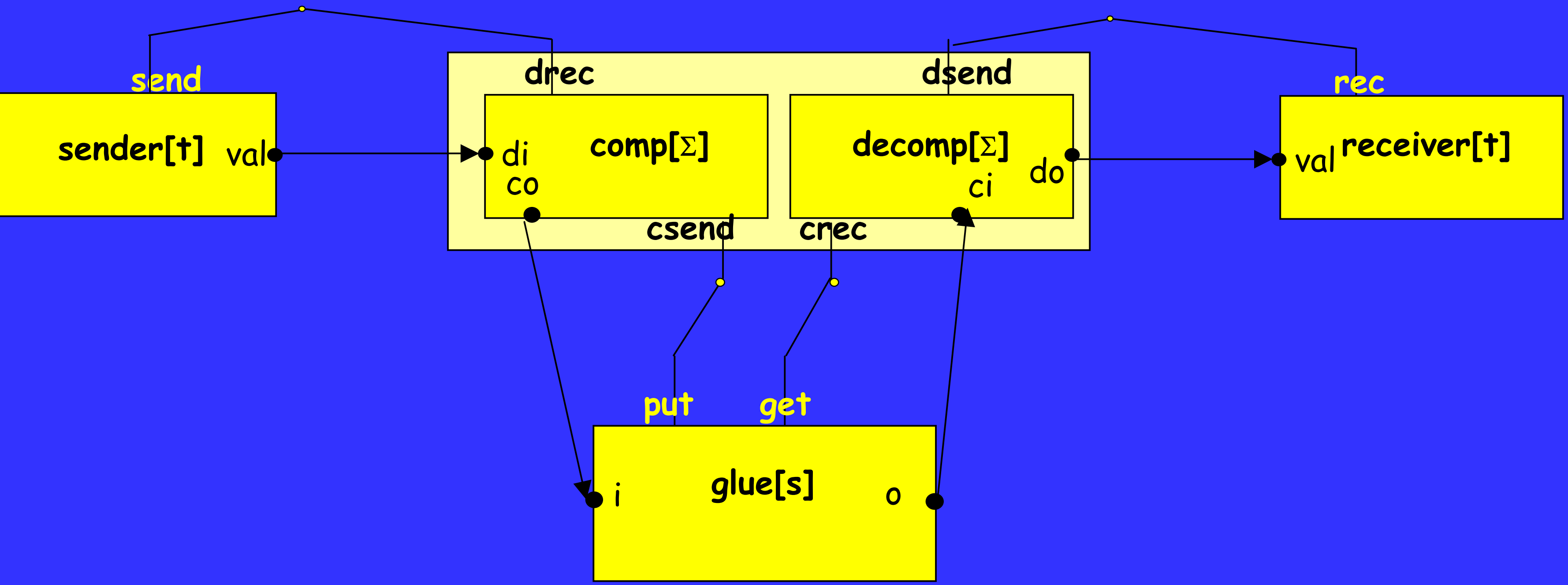


$$\begin{tikzcd}
 & & pG & & & \mathbf{pC} \\
p\theta_1 \arrow[urr] \arrow[d] & & p\theta_i \arrow[u] \arrow[d] & & p\theta_k \arrow[ull] \arrow[d] & \\
pR_1 \arrow[drr] & & pR_i \arrow[d] & & pR_k \arrow[dll] & \\
 & & G & & & \mathbf{C} \\
\theta_1 \arrow[urr] \arrow[d] & & \theta_i \arrow[u] \arrow[d] & & \theta_n \arrow[ull] \arrow[d] & \\
R_1 & & R_i & & R_n &
\end{tikzcd}$$

he Compression hoc

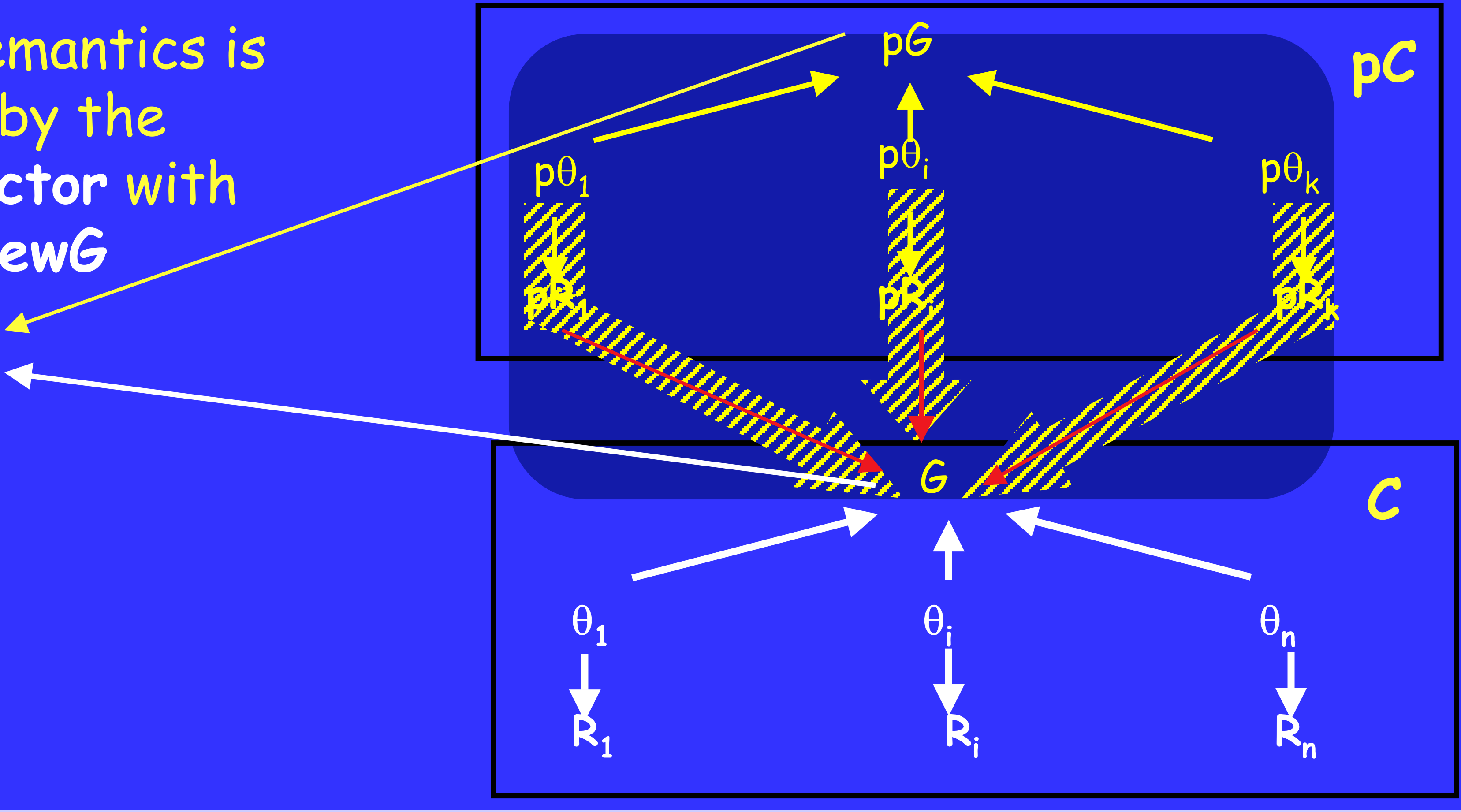


he Compression hoc: semantics

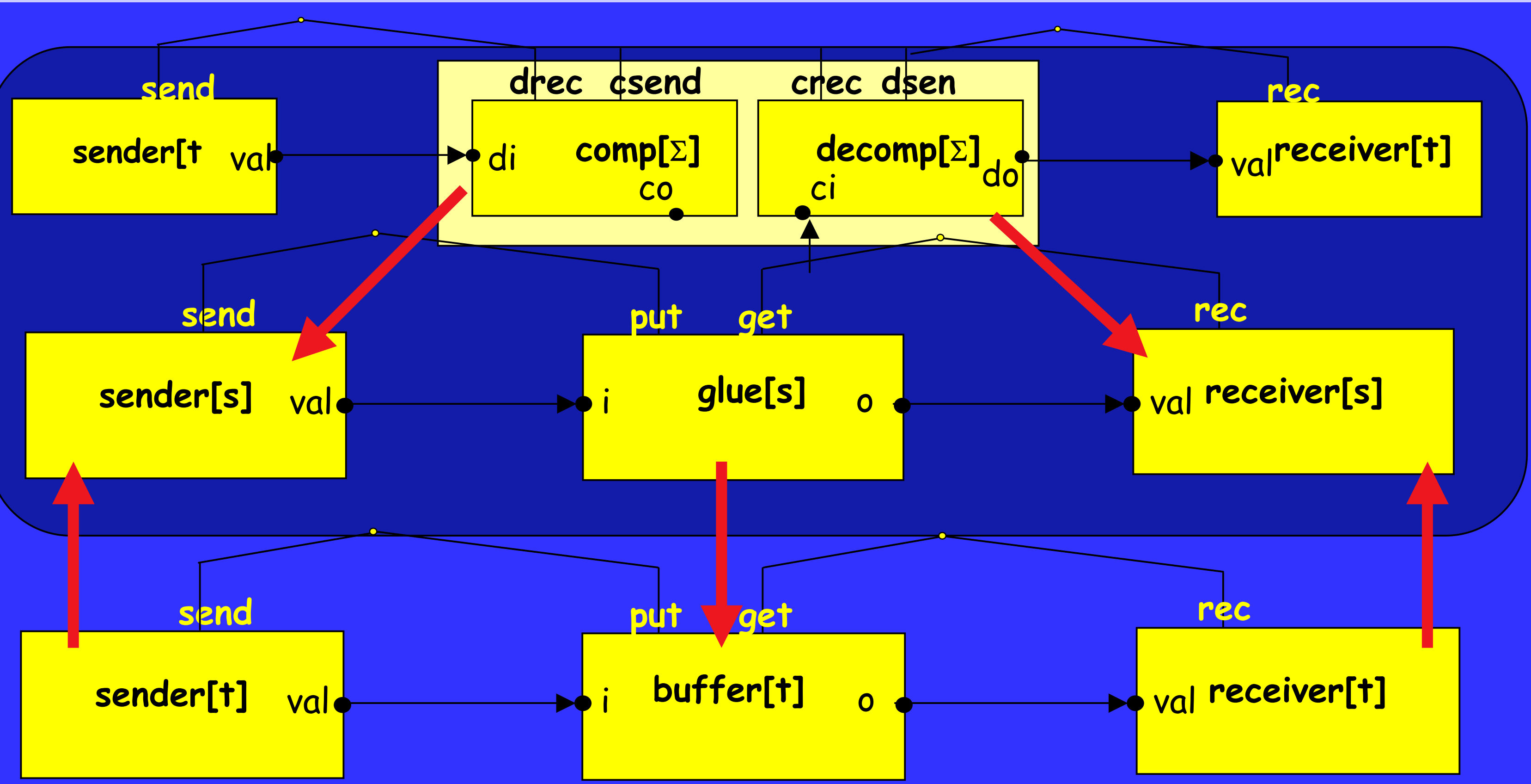


Categorical Semantics of HOCs

Its semantics is given by the **connector** with glue **newG**

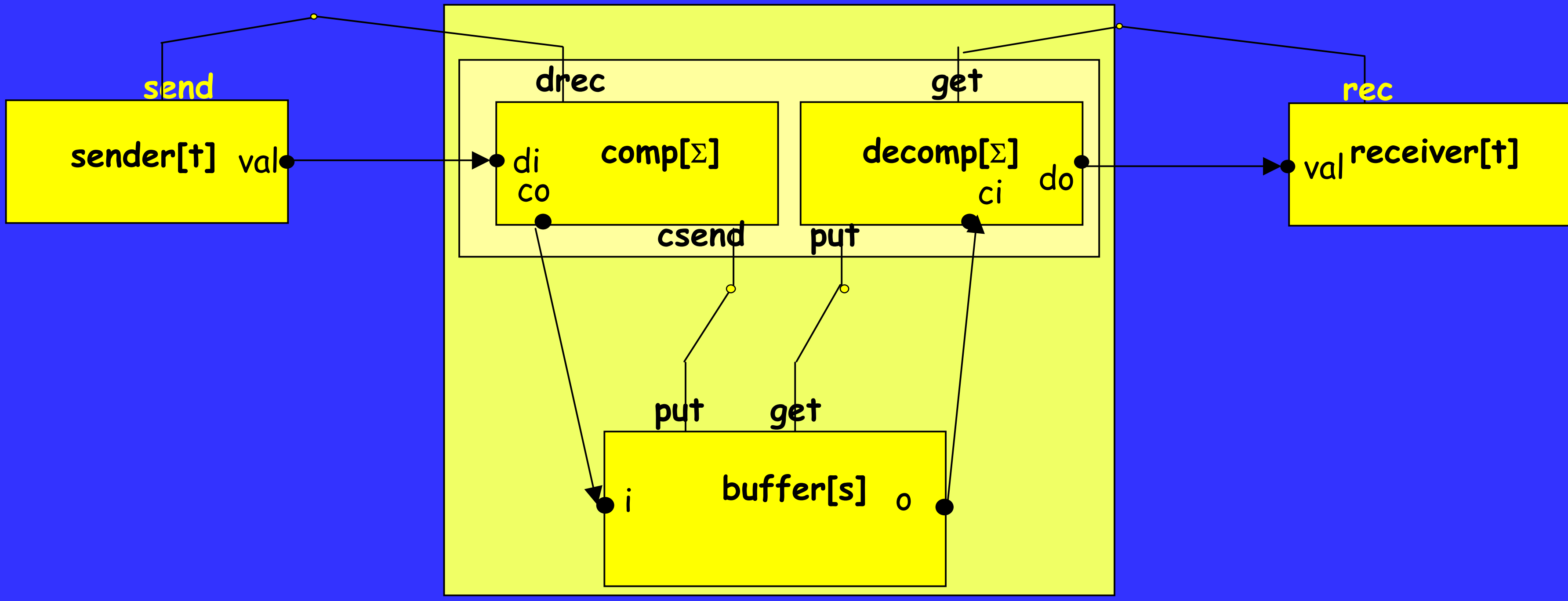


Instantiation of Compression with Async



Instantiation of Compression with Async

The semantics of this instantiation is given by the connector

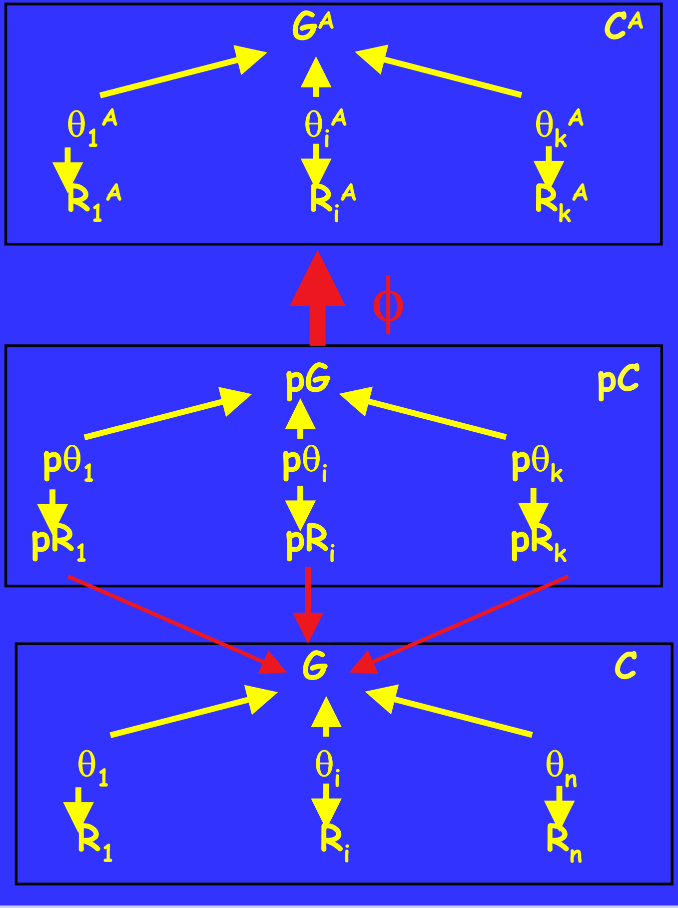


Categorical Semantics of HOCs: Instantiation

An instantiation of a hoc consists of a fitting morphism

$$\phi : pC \rightarrow C^A$$

from the formal parameter to the actual parameter (a connector C^A)



Categorical Semantics of HOCs: Instantiation

A fitting morphism

$$\phi : pC \to C^A$$

consists of a pair of refinement morphisms

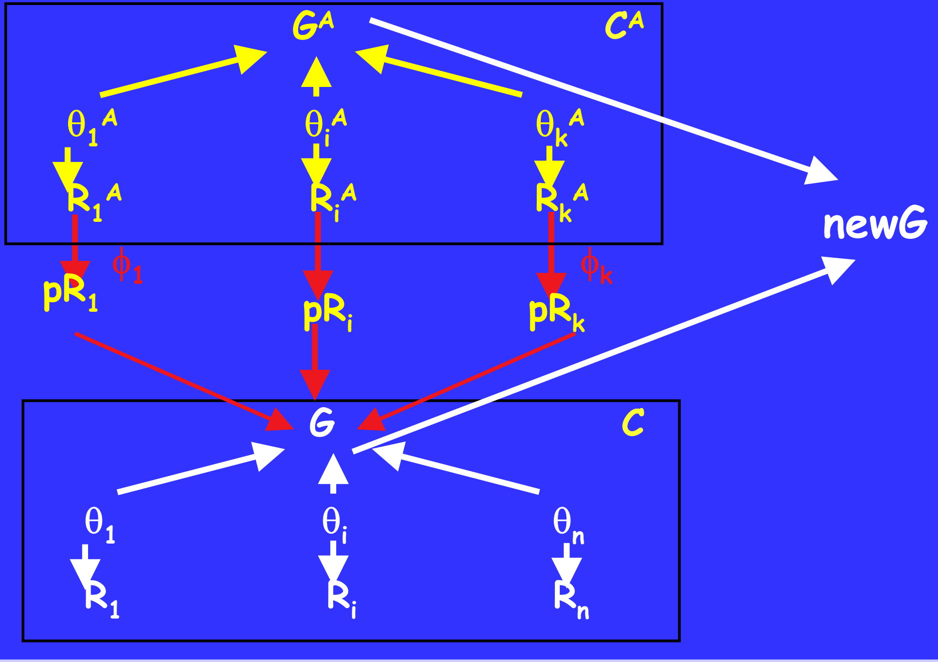
$$\begin{tikzcd}
 & & pG \arrow[drr, red] & & \\
p\theta_i \arrow[urr] \arrow[d] & & & & pG^A \\
pR_i & & \theta_i^A \arrow[urr] \arrow[d] & & \\
 & & R_i^A \arrow[ull, red] & &
\end{tikzcd}$$

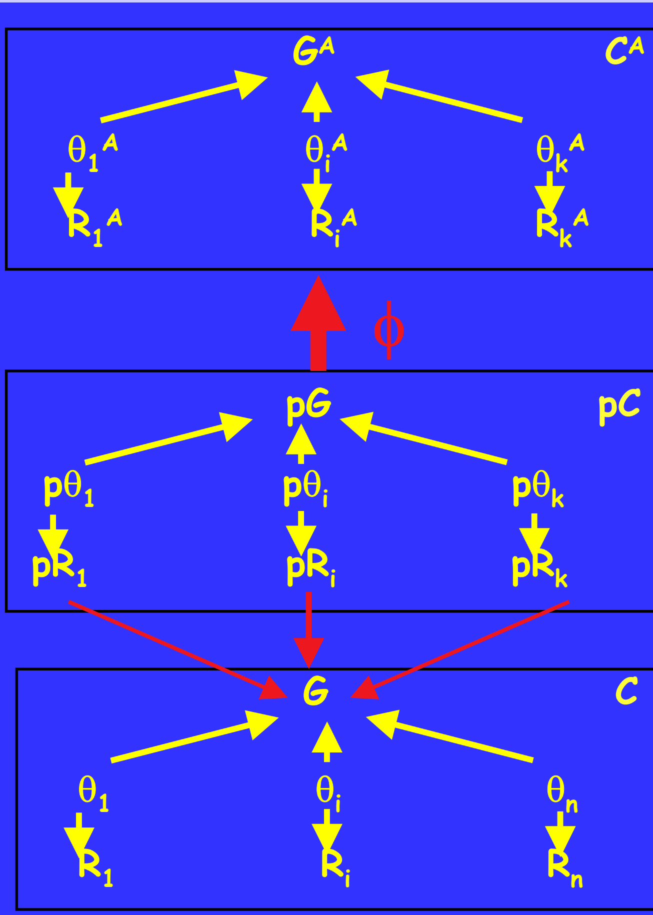
for each connection s.t. ...

$$\begin{tikzcd}
 & & G^A & & C^A \\
\theta_1^A \arrow[urr] \arrow[d] & & \theta_i^A \arrow[u] \arrow[d] & & \theta_k^A \arrow[ull] \arrow[d] \\
R_1^A & & R_i^A & & R_k^A \\
 & & & & \\
 & & pG \arrow[uuuu, red, "\phi"'] & & pC \\
p\theta_1 \arrow[urr] \arrow[d] & & p\theta_i \arrow[u] \arrow[d] & & p\theta_k \arrow[ull] \arrow[d] \\
pR_1 & & pR_i & & pR_k
\end{tikzcd}$$

Categorical Semantics of HOCs: Instantiation

The semantics of a hoc instantiation is the **connector** with same roles as C and its glue is **newG**





Generalisations

Hocs can be combined giving rise also to a hoc **parametrised instantiation**

We defined hocs with one parameter only but the extension to several parameters is straightforward

- Hocs with 1 parameter always model transformation/adaptation of a connector
- Hocs with several parameters allow us to describe more complex operations s.a.
    - ☑aggregation of connectors
    - ☑a “pipe” of connectors
    - ☑fault-tolerance service

Reconfiguration: Motivation

systems have to evolve due to changes in functional requirements (business rules) or to respond to changes in the environment (e.g., failures, transient interactions)

for safety or economical reasons, some systems cannot be shut down to be changed

domain with some interest in SA community but little formal work

Reconfiguration: Issues involved

Time: before or at run-time (dynamic reconfiguration)

Source: user (ad-hoc); topology/state (programmed)

Operations: add/delete components/connections; query topology/state

Constraints: structural integrity; state consistency; application invariants

Specification: architecture description, modification, constraint languages

Management: explicit/centralised (configuration manager); implicit/distributed (self-organisation)

Reconfiguration: Related Work

Work done in Distributed Systems, Mobile Computing, Software Architecture has at least one of the following drawbacks:

- not addressed at the architectural level
- arbitrary reconfigurations not supported
- only low-level behaviour specification (process calculi, term rewriting, etc.)
- interaction between computation and reconfiguration is complex, implicit, or blurred

On the other hand, they sometimes provide tool support, in particular automated analysis.

econfiguration: Approach

Explore the categorical approach to software architectures and parallel program design

- architecture = categorical diagram; system behaviour = colimit
- architecture = graph; reconfiguration = rewriting

Develop a reconfiguration language for easier specification and analysis.

CommUnity with State

Typed logical variables LV to denote the current state of components;

Nodes of configurations are designs with valuations ε: $loc(V) \rightarrow Terms(LV)$

- State only for variables controlled by the design
- Non-ground terms in the reconfiguration rules
- Ground terms in run-time configuration

Superposition morphisms must preserve state: $\varepsilon(l) = \varepsilon'(\sigma(l))$ for any local variable l

Graph Transformation

Graph category

- Objects: directed graphs with labelled nodes and arcs
- Morphisms: total functions between nodes and arcs preserving structure and labels

Production p: $L \xleftarrow{l} K \xrightarrow{r} R$

- graph L transformed into R through common subgraph K
- l and r are injective morphisms
- can be applied to graph G if match m: $L \rightarrow G$ exists

Graph Transformation: Derivation

$G \xRightarrow{p,m} H$ if 2 pushouts exist

$$\begin{tikzcd}
L \arrow[d, "m"'] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

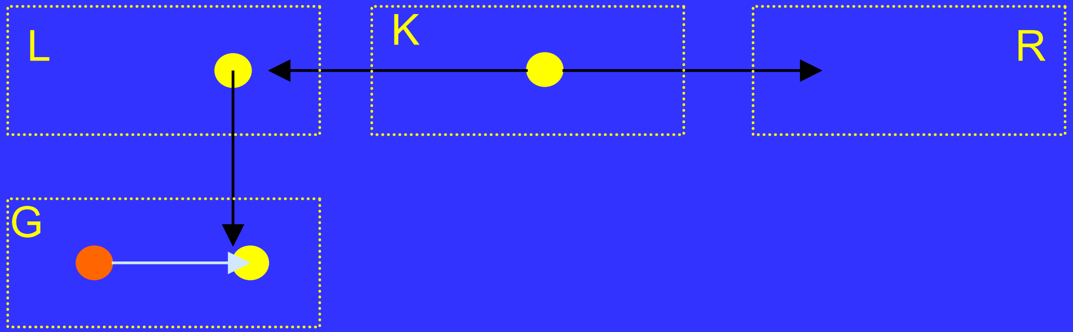
$D = G - (L - K)$ and $H = D + (R - K)$

Injection l guarantees D is unique

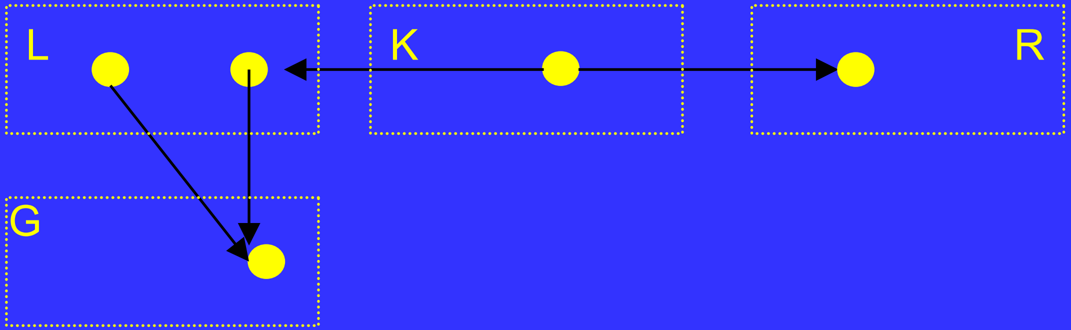
Injection r guarantees p is reversible

Application Conditions

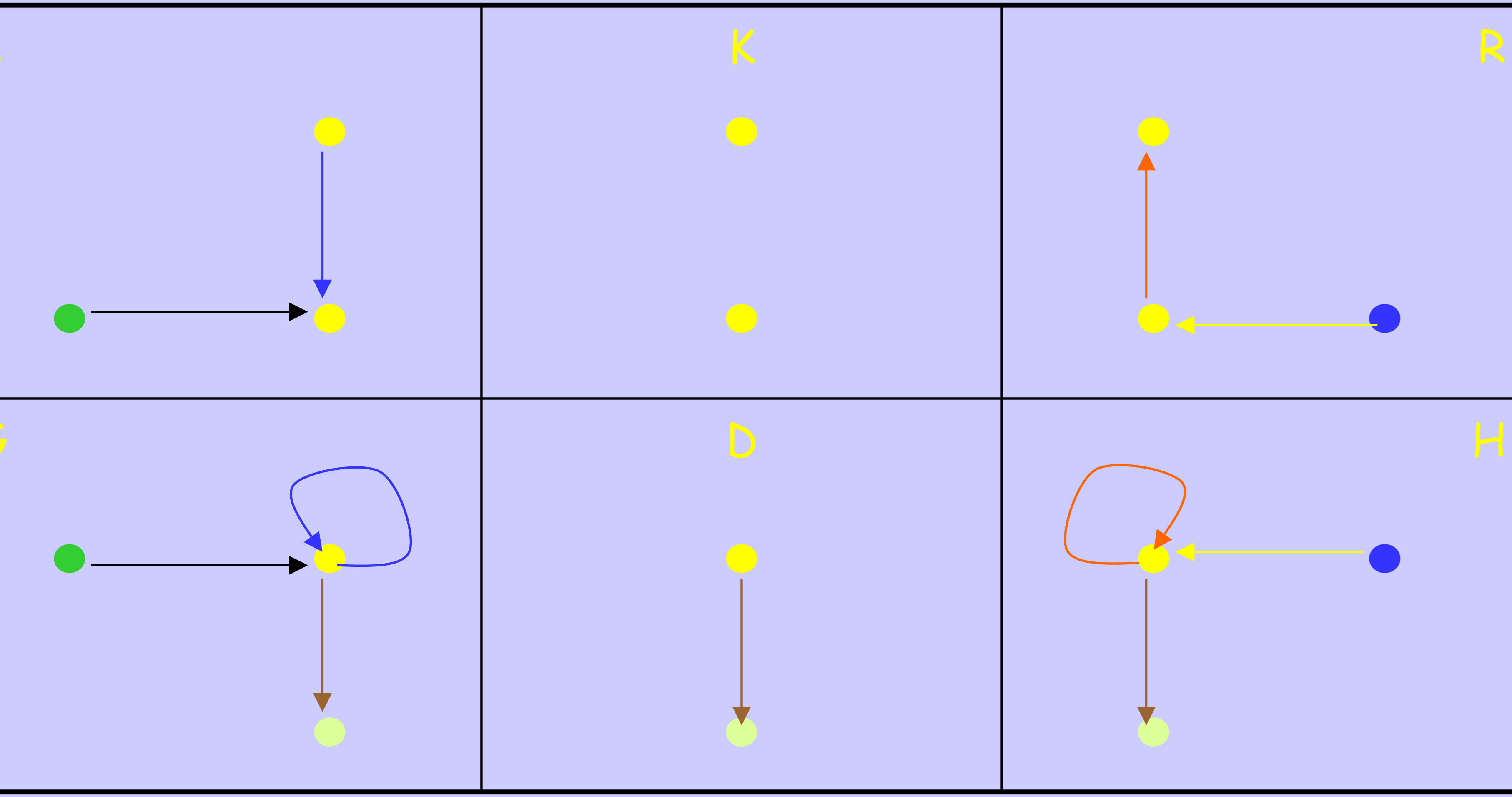
D does not exist if a node to be removed has arcs



D does not exist if a node is to be removed and kept



Example



Dynamic Reconfiguration

Run-time configurations: well-formed configurations with nodes labelled by designs with ground terms

Rules: $L \xleftarrow{l} K \xrightarrow{r} R$ if C

- parameterised by the algebraic specifications used in L,K,R
- C is condition over Vars(L), the logical variables ocurring in L
- Vars(R) $\subseteq$ Vars(L) to determine state of new components

Step: $G \xRightarrow{p,m,\phi} H$ with a substitution ϕ: Vars(L) $\rightarrow$ Terms($\emptyset$) s.t. $\phi(C)$ is true and $G \xRightarrow{\phi(p),m} H$ is a derivation with $\phi(p) = \phi(L) \leftarrow \phi(K) \rightarrow \phi(R)$

Reconfiguration: derivation sequence; does not change state (i.e., labelling)

Example

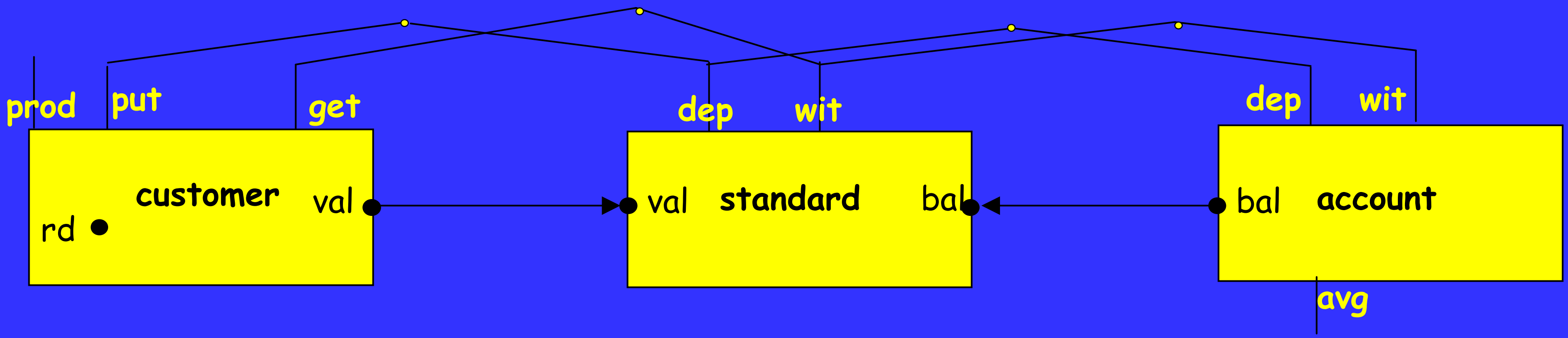
Managing the way Customers interact with their bank Accounts

```
design customer is
out   val:int
prv   rd:bool
do    prod[val,rd]:¬rd,false→rd’
[]    put[rd]:rd,false → ¬rd’
[]    get[rd]:rd,false → ¬rd’
```

```
design account is
out num:nat; bal, avgbal: int
in  v: nat
do  dep: true → bal’ = bal + v
[]  wit: true → bal’ = bal - v
[]  avg[avgbal]: true →
```

Standard Connector

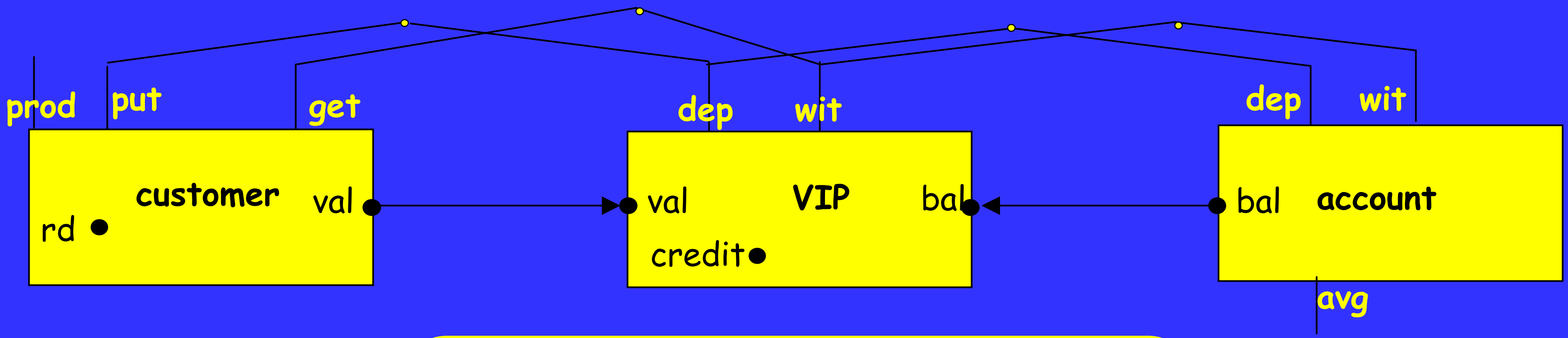
Customers may be subject to the standard rules for withdrawing money



```
design standard is
in  val: nat; bal: int
do  dep: true →
[]  wit: bal ≥ val →
```

IP Connector

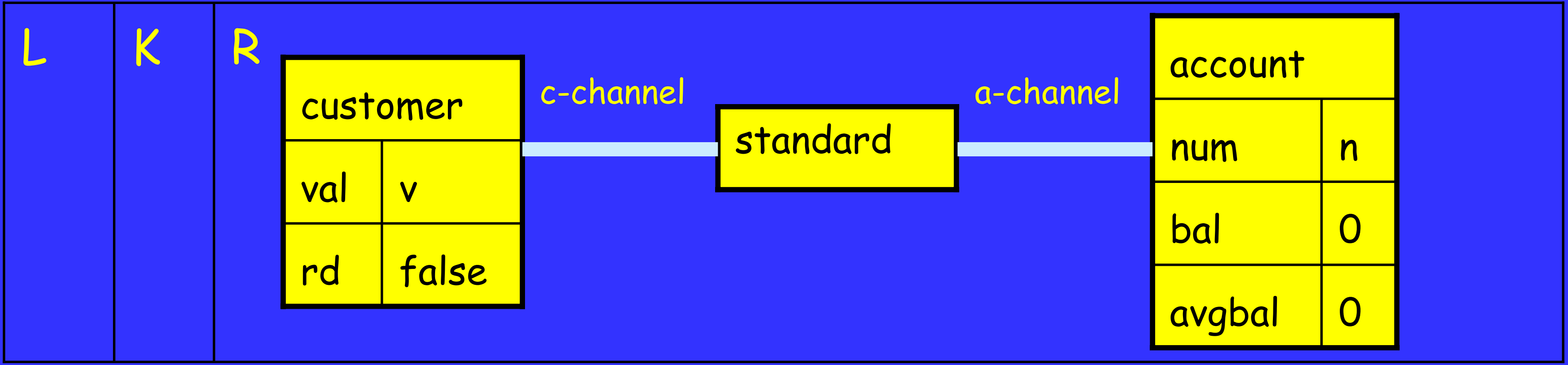
Customers may subscribe VIP-contracts that allow them to overdraw up to some limit as long as the average balance is greater than 1000.



```
design VIP is
in  val: nat; bal: int
prv credit:nat
do  dep: true →
[]  wit: bal+credit ≥ val →
```

Creating a client/account pair

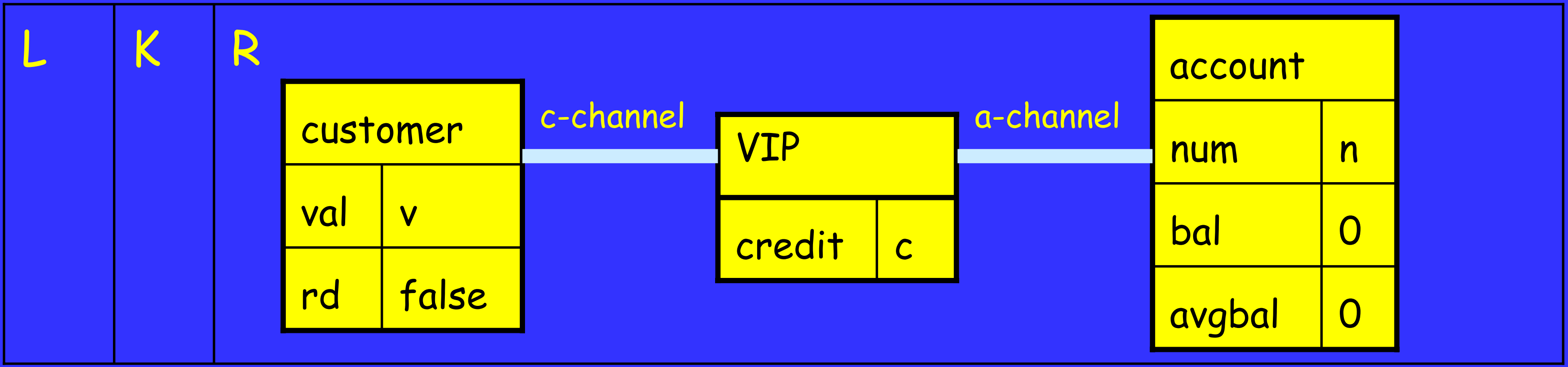
When a client/account pair is created, a decision has to be taken on the kind of contract that binds them. A production is defined for each kind:



This is a rule template, parameterised by the values to be assigned to the account number and the value the customer will deposit.

Creating a client/account pair

In the case of the VIP-contract, the credit limit has to be negotiated with the bank.



L

K

R

customer

val	v
rd	false

c-channel

VIP

credit	c

a-channel

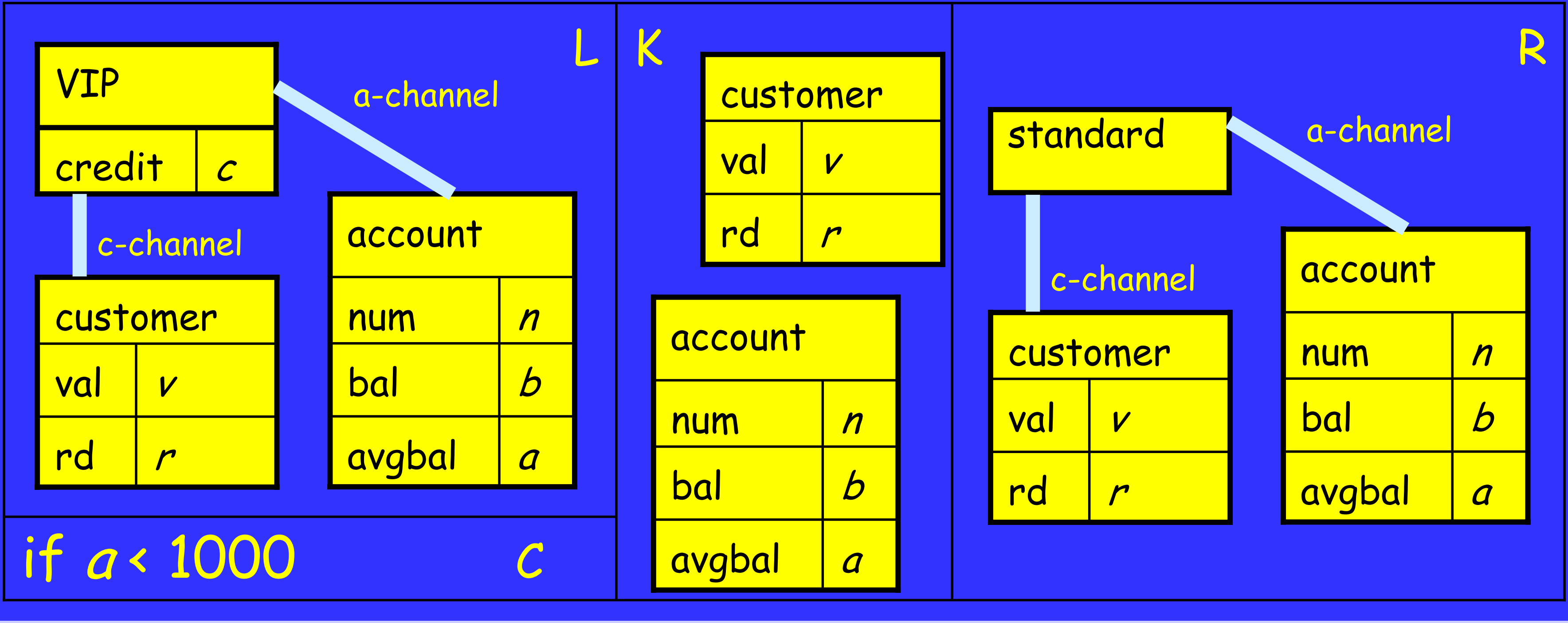
account

num	n
bal	0
avgbal	0

Again, this is a rule template that now also includes a parameter for the credit limit.

Modifying the contract

The following rule restores a VIP contract to standard when the average balance is below 1000.



Reconfiguration Specification

rewrite rules are cumbersome to write: repetition of nodes in graphs K and L; dummy nodes/arcs to control the way rules are applied

ideal: reconfiguration language with high-level programming constructs

but: ADLs only provide minimal reconfiguration support; distributed systems have powerful languages but do not have architectural abstractions

goal: compact, conceptually elegant language with formal semantics for describing reconfiguration within architectural description of a system

econfiguration Language Elements (1)

configuration variables:

- typed over data sorts
- typed over components and connectors (node references)
- maintain information about current configuration
- designs cannot access them: separation of computation from reconfiguration

query: expression that returns list of tuples of nodes matching the given criteria on topology and state

Reconfiguration Language Elements (2)

basic commands:

- create/remove components and connectors
- update configuration variables
- semantics given by reconfiguration rules

complex commands: sequence, choice, and iteration

scripts:

- group commands into a unit
- may be nested and recursive
- may have parameters and local configuration variables

Main script

```
script Main
prv i : record(a : Account)
script RestoreStandard ... end script
for i in match {a:Account | with
                         a.avgbal<1000}
  loop
     RestoreStandard(i.a)
  end loop
end script
```

Main script

```
script Main
orv i : record(a : Account)
script RestoreStandard ... end script
for i in match {a:Account | with
                         a.avgbal<1000}
 loop
    RestoreStandard(i,a)
 end loop
end script
```

node reference

local configuration variable

Main script

```
script Main
prv i : record(a : Account)
script RestoreStandard ... end script
for i in match {a:Account | with
                            a.avgbal<1000}
 loop
     eStandard(i.a)
 end loop
end scr
```

list iterator

condition on state

match {Decl | ...} returns list(record(Decl))

Auxiliary Script

```
cript RestoreStandard
n   a: Account
rv i: record(c:Customer; co:VIP)
or i in match {c:Customer;co:VIP |co(c, a)}
  loop
     remove i.co;
     create standard(i.c, a);
  end loop
nd script
```

input parameter

refers the glue

condition on topology

role instantiation

Creating a VIP connector

```
script CreateVIP
in   n, limit : nat
out c : Customer
prv a : Account
c := create Customer with
        rd := false || val := 0;
a := create Account with
        bal := 0 || avgbal := 0 || num := n;
create VIP(c, a) with credit := limit
end script
```

state initialisation

Interpretation Loop

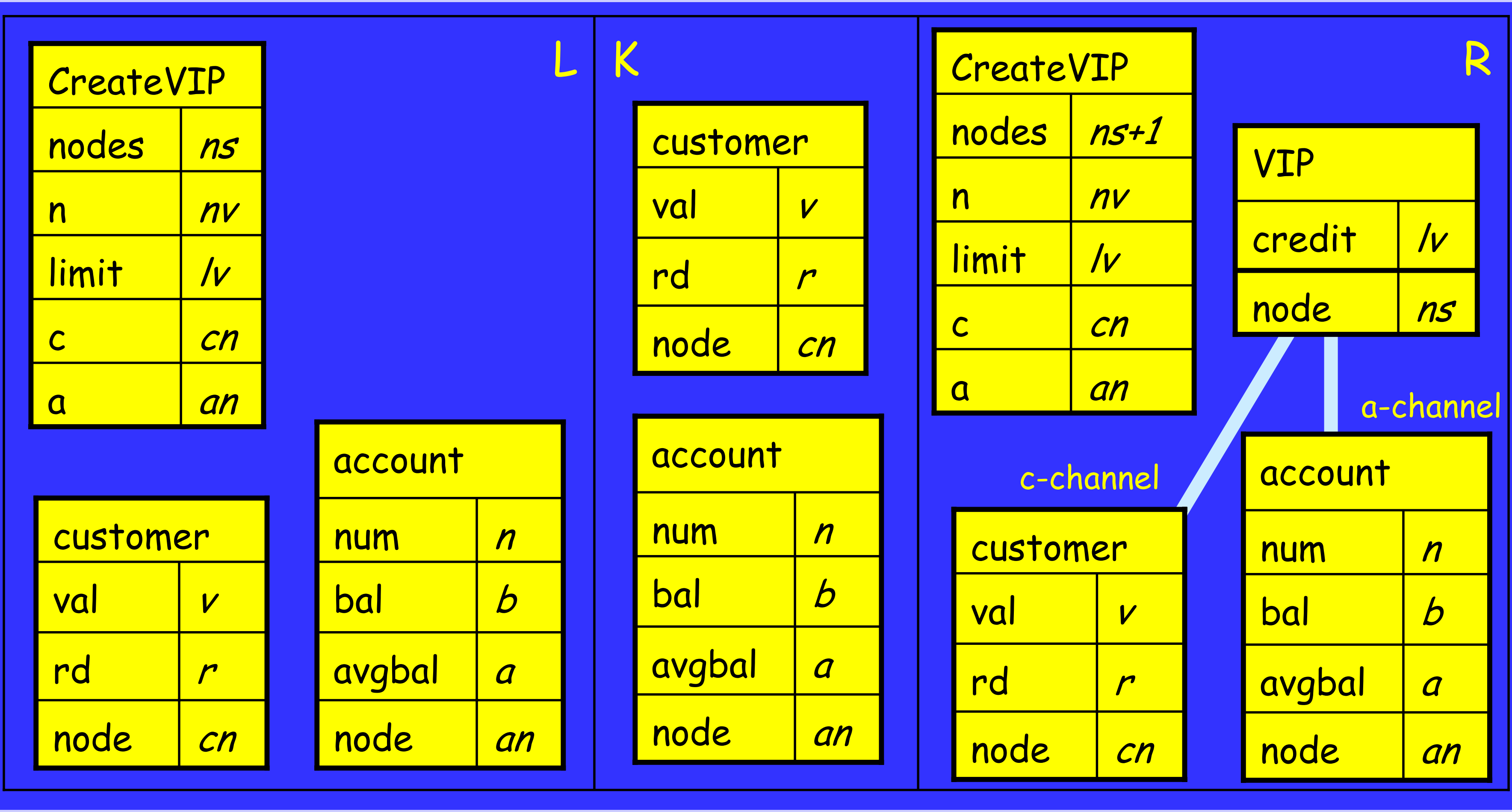
1. Execute one computation step over the current run-time configuration
2. Let the user call a top-level script if s/he wishes (ad-hoc reconfiguration)
3. Call a parameterless script 'Main', if it exists (programmed reconfiguration)
4. Go to step 1

The administrator may change the set of scripts at any time.

emantics

- one new private variable 'node:nat' for each component and glue design
- configuration designs with private variables only:
  - one design for each lexical scope level (script)
  - one private variable per configuration variable in that level
  - node references translated to integer variables
  - undefined node references translated to value 0
  - one variable 'nodes:nat' to count how many nodes created
- one or more rules for each basic command:
  - L has designs for configuration and nodes referred in command
  - R includes updated configuration design

Semantics of
`create VIP(c, a) with credit := limit`



L

CreateVIP	
nodes	*ns*
n	*nv*
limit	*lv*
c	*cn*
a	*an*

customer	
val	*v*
rd	*r*
node	*cn*

account	
num	*n*
bal	*b*
avgbal	*a*
node	*an*

K

customer	
val	*v*
rd	*r*
node	*cn*

account	
num	*n*
bal	*b*
avgbal	*a*
node	*an*

R

CreateVIP	
nodes	*ns+1*
n	*nv*
limit	*lv*
c	*cn*
a	*an*

VIP	
credit	*lv*
node	*ns*

customer	
val	*v*
rd	*r*
node	*cn*

account	
num	*n*
bal	*b*
avgbal	*a*
node	*an*

4

Coordination Contracts

Motivation

Coordination Technologies (ATX Software)

- A semantic modelling primitive (coordination contracts) with the expressive power of architectural connectors
- An architecture-centred development methodology (construction and evolution)
- Design patterns that implement contracts
- A contract development environment

Simple account

```
class Account
Operations
Deposit(in amount: Integer)
                → balance:=balance+amount

Withdraw(amount:Integer)
               → balance:=balance-amount;

attributes
  number : Integer;
  balance : Integer := 0;
end class
```

Notation for coordination contracts

```
coordination contract Traditional package
  partners x : Account; y : Customer;
  constraints ?owns(x,y)=TRUE;
  coordination
      tp:  when  y ->> x.withdrawal(z)
           do    call x.withdrawal(z)
           with  x.Balance() > z
end contract
```

IPs

```
coordination contract VIP package
  partners x : Account; y : Customer;
  constants VIP_BALANCE: Integer;
  attributes Credit : Integer;
  constraints
      ?owns(x,y)=TRUE;
      x.AverageBalance() >= VIP_BALANCE
  coordination
      tp:  when y ->> x.withdrawal(z)
           do   x.withdrawal(z)
           with x.Balance() + Credit() > z
end contract
```

reas of Application

- Defining business rules – Account Flexible Package
- Dynamic Type reconfiguration – A.C. Controller
- Specification of behaviour with state transitions – Electronic devices
- Use Cases – Automatic Teller Machine
- Design Patterns – Model and Observer
- Concurrency – Dining Philosophers
- Connectors of architectural layers

The Flexible Package

```
coordination contract AccountPackage
  partners c : Account; s : Account;
  attributes mn,mx : Integer;
  constraints c.owner=s.owner;
  coordination
  stoc: when (c.bal() < mn) do {
          s.withdrawal(min(s.l(),mx-c.bal())),
          c.deposit(min(s.bal(),mx-c.bal())}
   ctos: when (c.bal() > mx)
         do { c.withdrawal(c.bal()-mx),
                s.deposit(c.bal()-mx)}
end contract
```

Coordination Rules

A Coordination Rule has the form

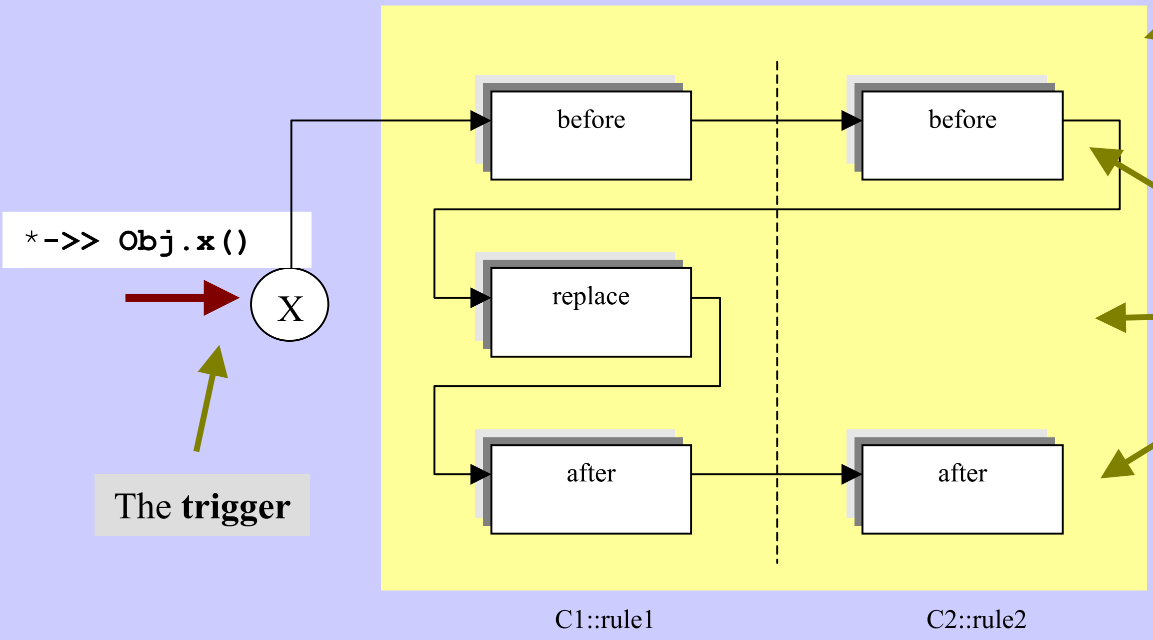
<name> :   **when** *<trigger>*
with *<guardCondition>*
do *<set of actions>*

The **trigger** defines when a rule must be considered active.
It may be a *condition*, or a *request* to a participant operation

The **guard condition** imposes additional constraints on the reaction to the trigger, when regulated by this rule

The **actions** describe the behavior defined by the rule:
- extra behaviour to be executed **before** or **after** the trigger operation,
- or **replacement** behavior for the trigger operation

Coordination Semantics



`*->> Obj.x()`

C1::rule1

C2::rule2

The **transactional behavior** for the operation X under coordination

The **Actions** defined by the rules

The **trigger**

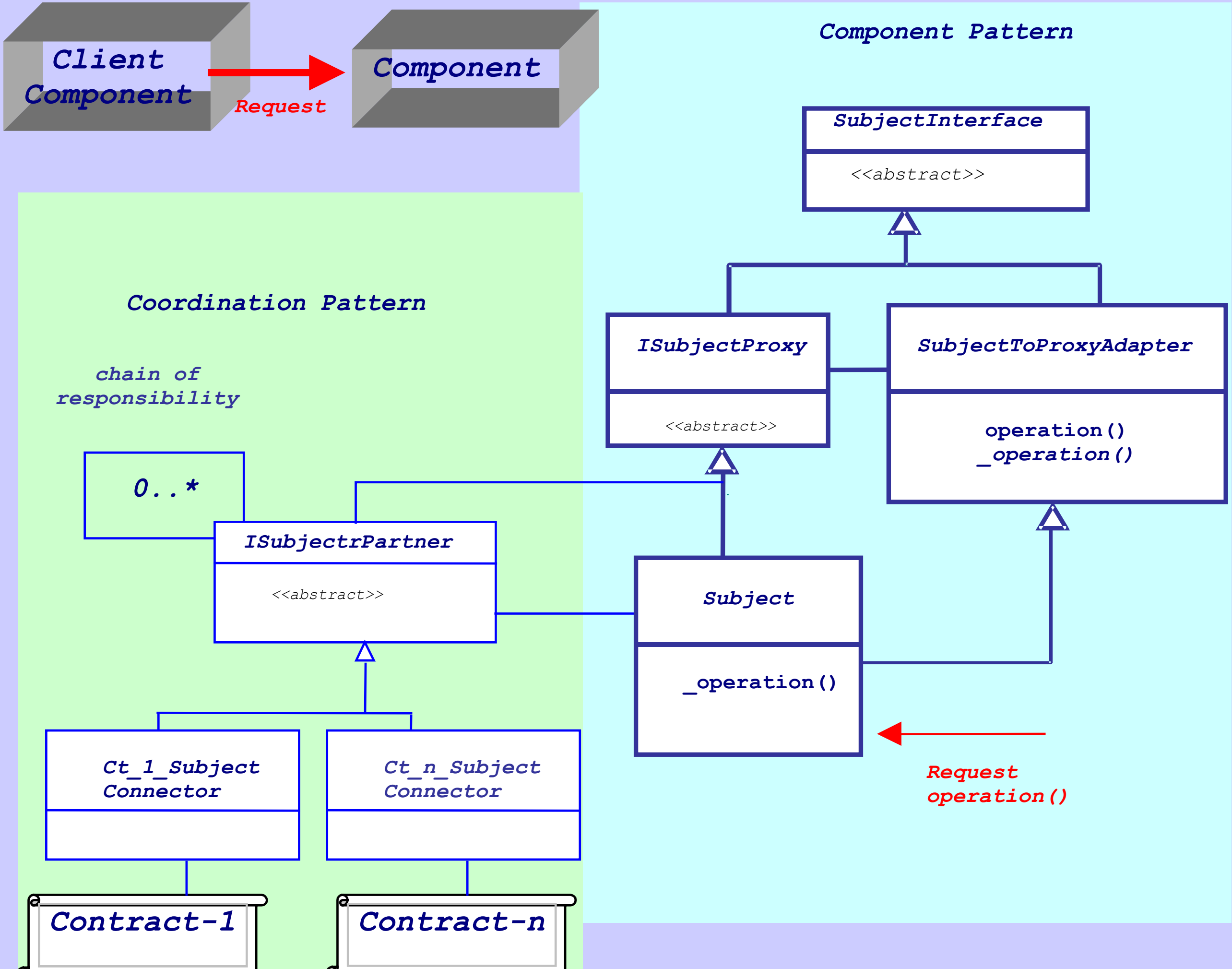
The **Rules** of the several contacts involving object *Obj* that *satisfy the trigger* and additional conditions

A design pattern for coordinations

None of the standards for component-based software development - CORBA, JavaBeans, COM - can support superposition as a first-class mechanism.

Because of this, we propose our solution as a design pattern that exploits polymorphism and subtyping, and is based on other well known design patterns, such as the Chain of Responsibility, and the Proxy or Surrogate.

A coordination design pattern



Client Component

Request

Component

Component Pattern

SubjectInterface

<<abstract>>

Coordination Pattern

chain of responsibility

ISubjectProxy

<<abstract>>

SubjectToProxyAdapter

operation()
_operation()

0..*

ISubjectrPartner

<<abstract>>

Subject

_operation()

Request operation()

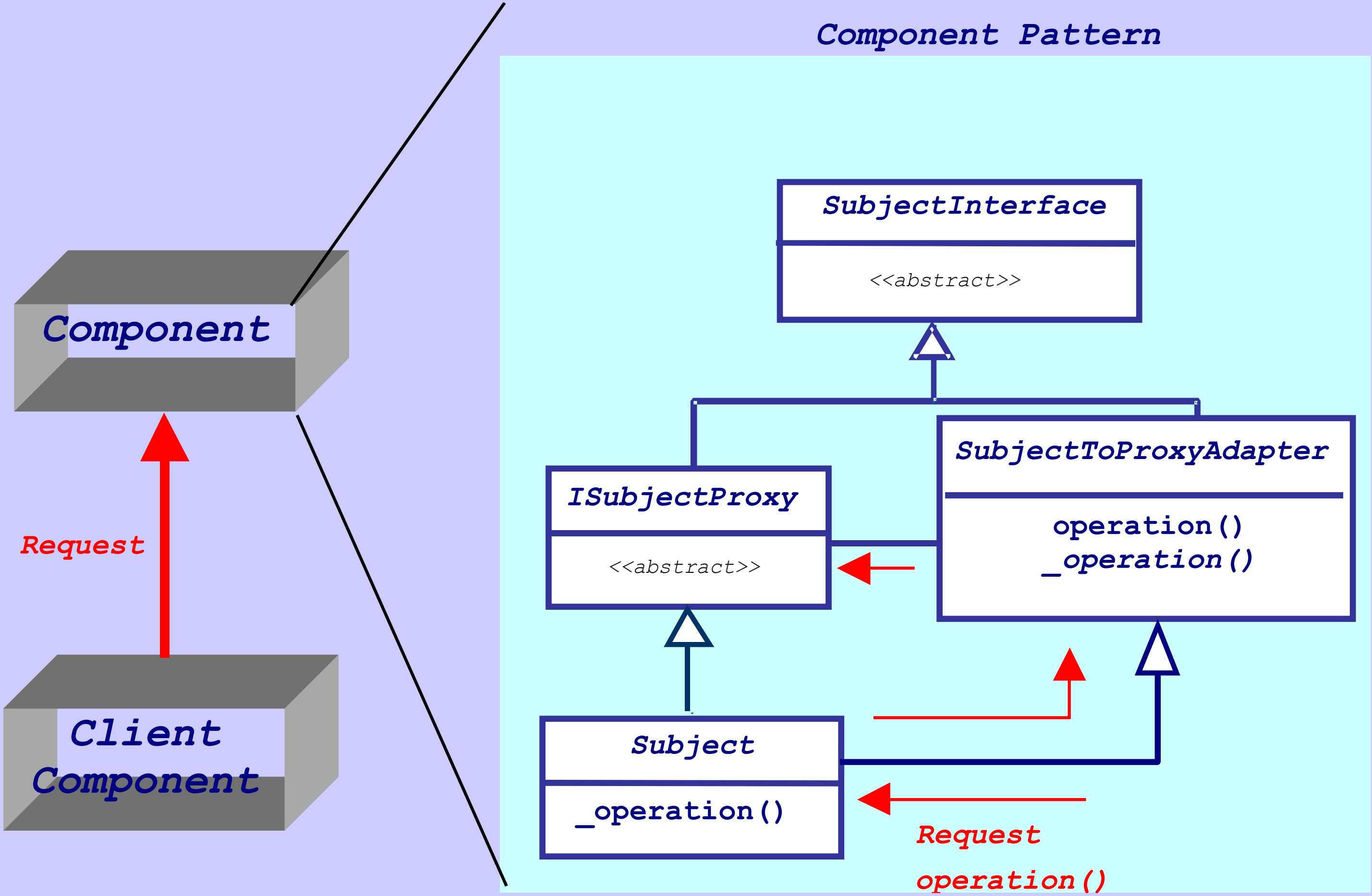
Ct_1_Subject Connector

Ct_n_Subject Connector

Contract-1

Contract-n

A coordination design pattern



Component

Request

Client Component

Component Pattern

SubjectInterface

<<abstract>>

ISubjectProxy

<<abstract>>

SubjectToProxyAdapter

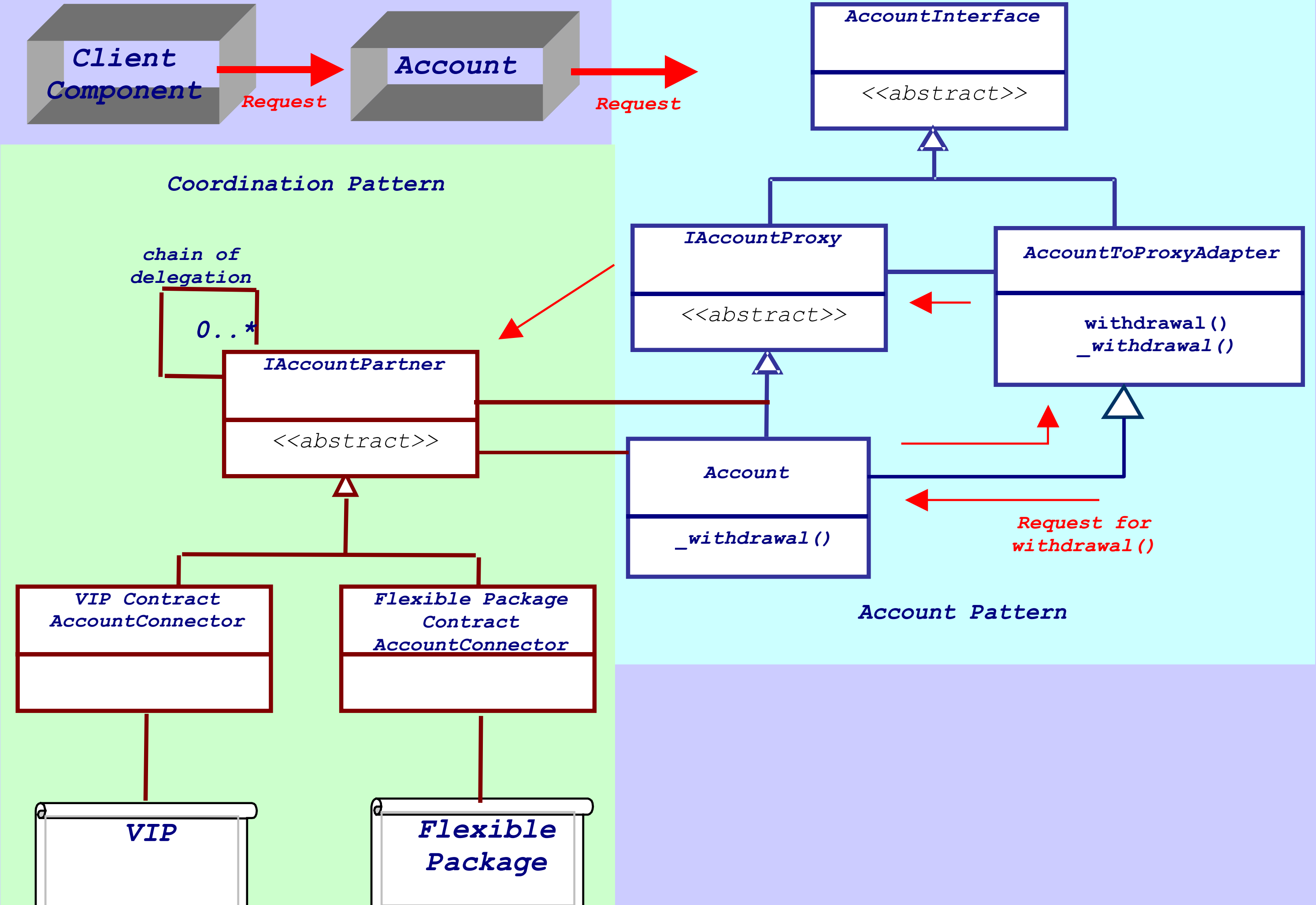
operation()
_operation()

Subject

_operation()

Request
operation()

Account coordination

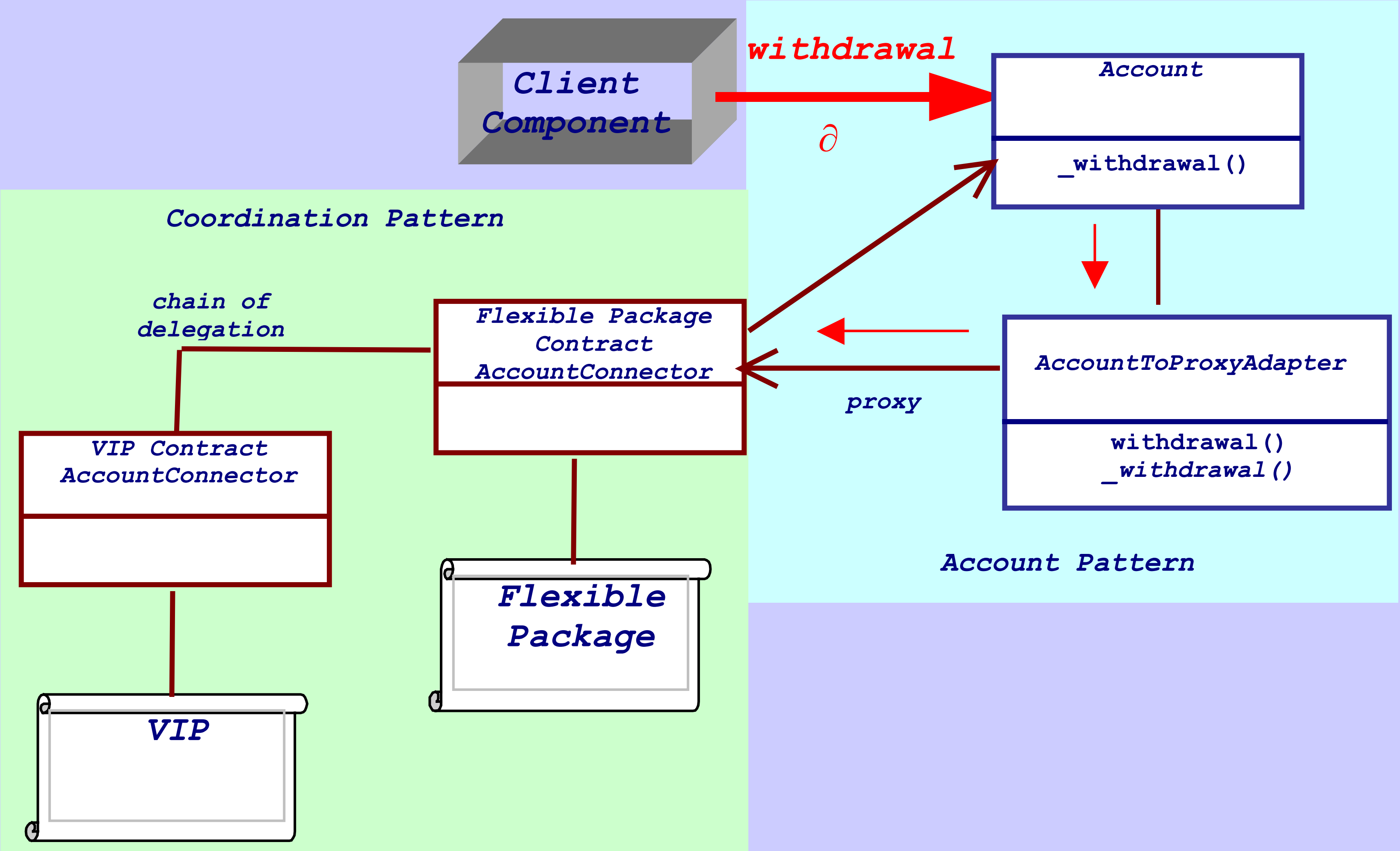


Account coordination

If there are no contracts coordinating a real subject, the contract pattern can be simplified.

In this scenario, the only overhead imposed by the pattern is an extra call from *SubjectToProxyAdapter* to *Subject*.

Account coordination



Operational view

Before *the subject* gives rights to *the real object* to execute the request, it intercepts the request and gives right to *the contract* to decide if the request is valid and perform other actions.

This allows us to impose other contractual obligations on the interaction between the caller and the callee.

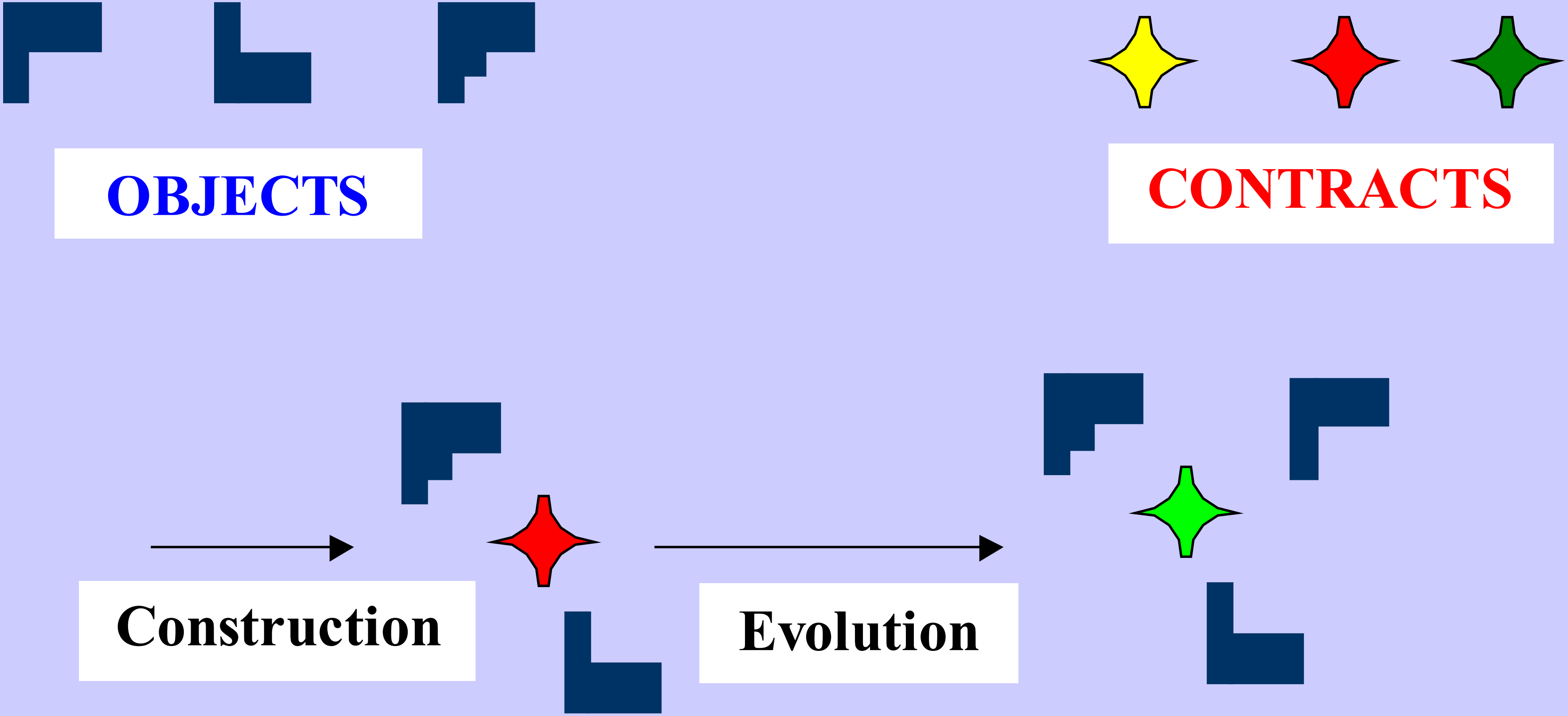
This is the situation of the first model discussed in section 2 where new pre-conditions were established between Account Withdrawals and their Customers.

Operational view

On the other hand, it allows the contract to perform other actions before or after the real object executes the request.

Only if the contract authorises can the connector ask *the involved objects* to execute and commit, or undo execution because of violation of post-conditions established by the contract.

The development process



OBJECTS

CONTRACTS

Construction

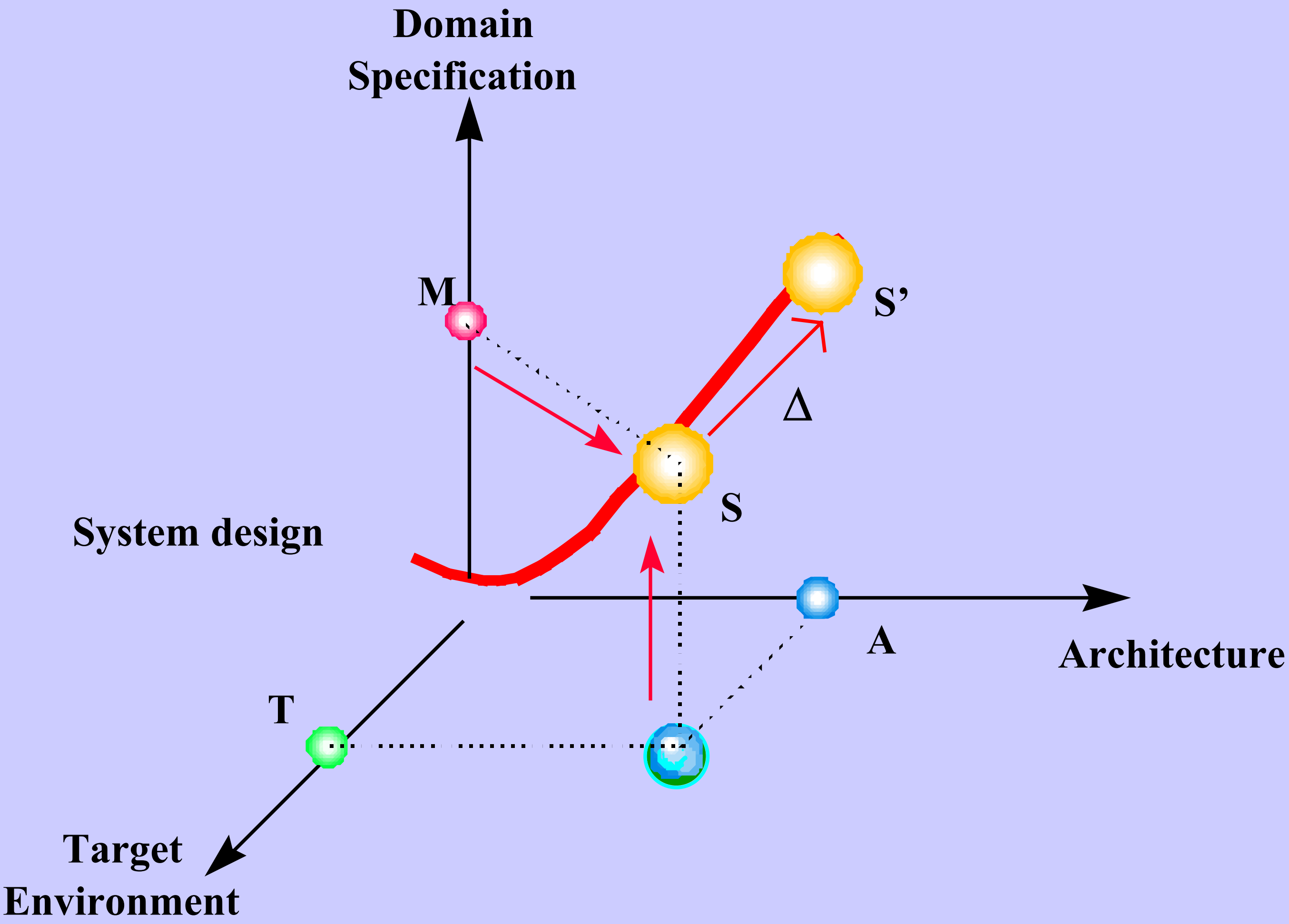
Evolution

he implementation space

A three-dimensional space with the following dimensions is proposed for producing code, for any specific implementation plataform, from high level specifications:

- Domain Specification: an ideal model of the business problem without any details concerning implementation;
- Architecture: a model that represents architectural designs;
- Target Environment: the technology used to implemente the business problem according with the choosen architecture.

he implementation space



The implementation space

The architecture of the system is defined by the way modules are interconnected and objects are coordinated.

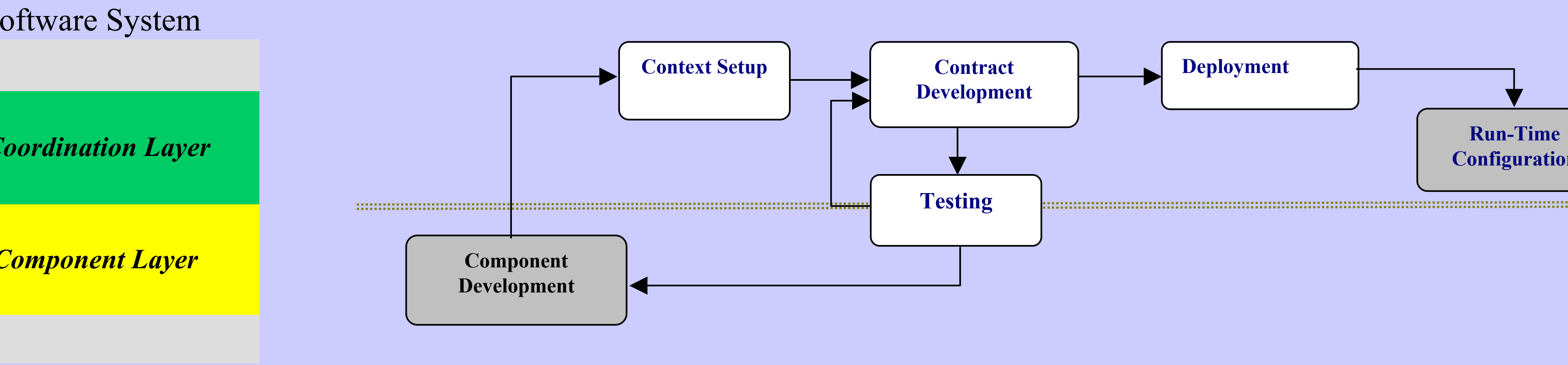
Hence, modules are vital for decomposing large specifications and specifying parts with sufficient precision that one can construct each part knowing only the specification of the other parts.

The nature of the components and their relationships is influenced by infrastructural constraints like the distribution strategy, type of interaction with the system environment, etc.

CDE - Coordination Development Environment

A development and run-time environment for layered coordination systems :

The *coordination layer*, defining the more volatile part of a system, is built over the *component layer*, the stable parts of the business



CDE: Development Activities

Registration: components are registered as candidates for coordination.

Edition: Contract types are defined connecting registered components. Coordination rules are defined on those contracts.

Deployment: the code necessary to implement the coordinated components and the contract semantics in the final system is produced according to the contract design pattern.

CDE: Run time Activities

Animation: facilities are provided allowing testing/prototyping of contract semantics

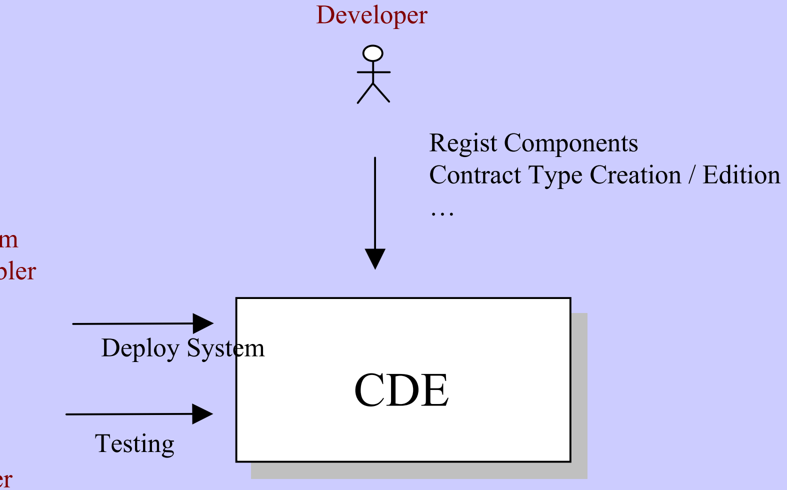
Registration: contract types are registered in the system.

Configuration: contracts are configured in the system (enabling/disabling rules, priorities, etc)

Evolution: concrete contracts are created between specific system elements, regulating its behaviour.

CDE - User interaction

Development



m
bler

er

Developer

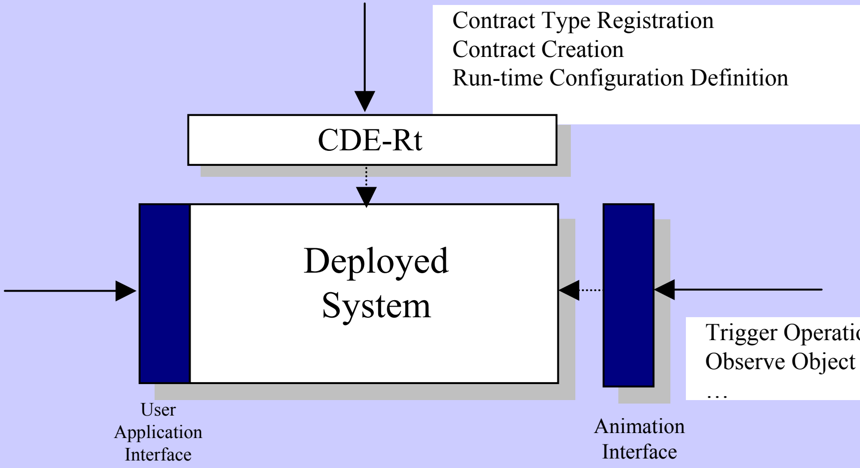
Regist Components
Contract Type Creation / Edition
…

Deploy System

Testing

CDE

Run time



Contract Type Registration
Contract Creation
Run-time Configuration Definition

CDE-Rt

Deployed System

User Application Interface

Animation Interface

Trigger Operati
Observe Object
…

Concluding remarks

Increased separation of the domain concepts (objects) from the business rules that regulate their behaviour;

Coordination features available as first-class citizens through a specific semantic primitive;

Support for different levels of change, reflecting the evolution of the domain:

- Flexible mechanisms for inheritance of behaviour;
- Separation of coordination from computation.

laimed contributions

- Increased separation of the domain concepts from the business rules that regulate their behaviour;
  - Recognising two different dynamics in system evolution: changes to the way components operate and changes to the way components are integrated (**white** vs **black** box);
  - More flexibility in the software development process (**plug and play**);
  - Better integration/coordination of third-party, closed components (e.g. legacy systems)
- One step closer to a real industry of components.

URLs

Papers:

- www.atxsoftware.com/publications.html (also includes papers on CommUnity and the categorical approach to software architecture)

Coordination Development Environment:

- www.atxsoftware.com/CDE

CommUnity Workbench:

- http://ctp.di.fct.unl.pt/~mw/sw/cw

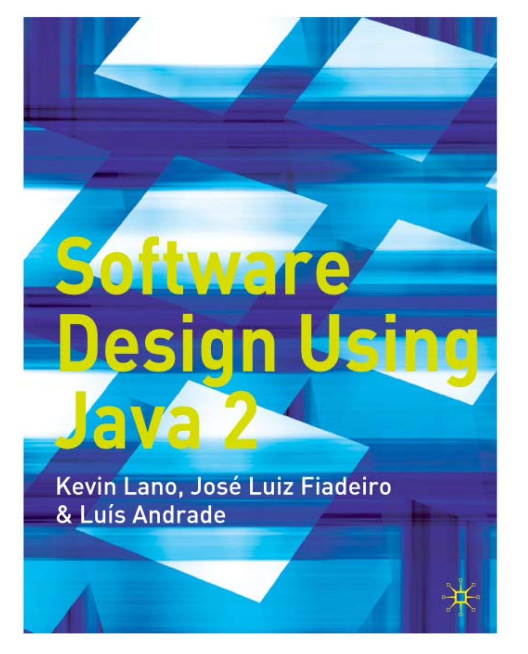
About to appear…

Software Design in Java 2

K.Lano, J.Fiadeiro and L.Andrade

Palgrave Macmillan

due Fall 2002



Software Design Using Java 2

Kevin Lano, José Luiz Fiadeiro
& Luís Andrade