Connectors for Mobile Programs

Michel Wermelinger José Fiadeiro

DI–FCUL TR–98–1

January 1998

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/biblioteca/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Connectors for Mobile Programs*

Michel Wermelinger

Departamento de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica
Portugal
E-mail: mw@di.fct.unl.pt

José Fiadeiro

Departamento de Informática
Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal
E-mail: llf@di.fc.ul.pt

Abstract

Software Architecture has put forward the concept of connector to express complex relationships between system components, thus facilitating the separation of coordination from computation. This separation is especially important in mobile computing due to the dynamic nature of the interactions among participating processes. In this paper we present connector patterns, inspired in Mobile UNITY, that describe three basic kinds of transient interactions: action inhibition, action synchronization, and message passing.

The connectors are given in COMMUNITY, a UNITY-like program design language which has a semantics in Category Theory. We show how the categorical framework can be used for applying the proposed connectors to specific components and how the resulting architecture can be visualized by a diagram showing the components and the connectors.

Keywords: software architecture, connectors, transient interactions, UNITY

1 Introduction

As the complexity of software systems grows, the role of Software Architecture is increasingly seen as the unifying infrastructural concept/model on which to analyse and validate the overall system structure in various phases of the software life cycle. In consequence, the study of Software Architecture has emerged, in recent years, as an autonomous discipline which requires its own concepts, formalisms, methods, and tools [7, 12]. The concept of connector has been put forward to express complex relationships between system components, thus facilitating the separation of coordination from computation. This is especially important in mobile computing due to the transient nature of the interconnections that may exist between system components. In this paper we propose an architectural approach to mobility that encapsulates this dynamic nature of interaction in well-defined connectors.

More precisely, we will present connector patterns for three fundamental kinds of transient interaction: action inhibition, action synchronization, and message passing. Each pattern is parameterized by the condition that expresses the transient nature of

*This work was partially supported by JNICT through contract PRAXIS XXI 2/2.1/MAT/46/94 (ESCOLA).

the interaction. The overall architecture is then obtained by applying the instantiated connectors to the mobile system components. To illustrate our proposal, components and connectors will be written in COMMUNITY [3, 5], a program design language based on UNITY [1] and IP [6].

In this paper we will follow the approach proposed in [2] and give the semantics of connectors in a categorical framework. In this approach, programs are objects of a category in which the morphisms show how programs can be superposed. Because in Category Theory [10] objects are not characterized by their internal structure but by their morphisms (i.e., relationships) to other objects, by changing the definition of the morphisms we can obtain different kinds of relationships between the programs, *without* having to change the syntax or semantics of the programming language. In fact, the core of the work to be presented in the remainder of this paper is an illustration of that principle: by changing program morphisms in a small way such that actions can be "unfolded", transient action synchronization becomes possible.

The nature of the connectors proposed in the paper was motivated and inspired by Mobile UNITY [11, 9], an extension of UNITY that allows transient interactions among programs. However, our approaches are somewhat different. Mobile UNITY suggests the use of an interaction section to define coordination within a system of components. We advocate an approach based on explicitly identified connectors, in order to make the architecture of the system more apparent and promote interactions to first-class entities (like programs). Moreover, while we base our approach on the modification of the superposition relation between programs, Mobile UNITY introduces new special programming constructs, leading to profound changes in UNITY's syntax and computational model. However, we should point out that some of these syntactic and semantic modifications (like naming of program actions and locality of variables) were already included in COMMUNITY.

To make it easier for interested readers to compare our approach with Mobile UNITY we will use the same example as in [11]: a luggage distribution system. It consists of carts moving on a closed track transporting bags from loaders to unloaders that are along the track. Due to space limitations we have omitted many details which, while making the example more realistic, are not necessary to illustrate the main ideas.

2 Mobile Community

The framework to be used consists of programs and their morphisms. The first two subsections present the syntax and semantics of COMMUNITY, while the category of programs is left to the third subsection. As mentioned before, only the morphisms have been changed[1] with respect to previous definitions [2, 5].

A COMMUNITY program is basically a set of named, guarded actions. Action names act as *rendez-vous* points for program synchronization. At each step, zero or more actions whose guards are true execute in parallel. Each action consists of one or more assignments to execute simultaneously. The right hand side of an assignment is an expression that denotes a set of values. Non-deterministically, one of them is assigned to the attribute (program variable) on the left side. This is useful for program refinement. Each attribute used by a program is either external—its value is provided by the environment—or local—its value is initialized and modified by the program.

[1] In fact we have taken the opportunity to change also the language in a very small way unrelated to the issue at stake.

A superposition morphism from the underlying program P to the transformed program P' is a mapping from P's attributes and actions to those of P', stating in which way P is a component of P'. Therefore the morphism must map the local attributes of P into local attributes of P' and it must preserve the functionality of P, i.e., the behaviour of P is part of the behaviour of P'.

Locations are an important aspect of mobility [8]. We take the same approach as Mobile UNITY and represent location by a distinguished attribute. However, our framework allows us to handle locations in a more flexible way. In Mobile UNITY there is a global declaration of the "spatial context", (i.e., the type of the location attributes) while in COMMUNITY each program may represent space in the most appropriate manner for the computations it performs because type declarations are local. Moreover, we can distinguish whether the program controls its own motion or if it is moved by the environment by declaring the location attribute as local or external, respectively. Finally, through the choice of an appropriate morphism, it is possible to state whether a given component is a "subpart" or a "subsystem" of a given system. The former means that component and system are co-located: whenever one moves, so does the other. The latter means that the component can move independently within the system. This can be modelled by a morphism that maps (or not) the location attribute of the component to the location attribute of the system.

The formal treatment of locations will be the same as for any attribute because they have no special properties at the abstract level we are working at. However, any implementation of COMMUNITY will have to handle them in a special way, because a change in the system's location implies a change in the value of the location attribute and vice-versa. We assume therefore some special syntactic convention for location attributes such that a compiler can distinguish them from other attributes. Following the notation proposed by Mobile UNITY, in this paper location attributes start with λ.

2.1 The language

To introduce the syntax of COMMUNITY we present the program that controls a cart. Like bags and (un)loaders, carts have unique identifiers, which will be represented by external integer attributes, so that a cart cannot change its own identity. A cart can transport at most one bag at a time from a source loader to a destination unloader. Initially, the cart's destination is the loader from which it should fetch its first bag. The unloader at which a bag must be delivered depends on the bag's identifier. After delivering a bag, or if a loader is empty, the cart proceeds to the next loader. Absence of a bag will be denoted by the identifier zero.

The track is divided into segments which are further divided into ten units, and carts can move at two different speeds: slow (one length unit per time unit) and fast (two length units). The location of a cart is therefore given by an integer. A cart stops when it reaches its destination. The action to be performed at the destination depends on whether the cart is empty or full.

```
program Cart is
var   bag : int; λ, dest : int;
read  id, nbag : int;
init  bag = 0 ∧ dest = InitDest(id) ∧ λ = InitLoc(id)
do    slow: [λ≠ dest → λ := λ+ 1]
[]    fast: [λ≠ dest → λ := λ+ 2]
[]    load: [λ= dest ∧ bag = 0 → bag := nbag || dest := Dest(nbag, dest)]
[]    unload: [λ= dest ∧ bag ≠ 0 → bag := 0 || dest := Next(dest)]
```

In the examples we will omit the declarations and the definitions of the data types and predefined functions used by the programs. Since a mobile setting is a particular case of a distributed system, where programs may have been developed independently, we will assume those declarations and definitions are not global but instead local to each program. The morphisms will then be able to relate programs with different but compatible data types.

Definition 1 A *data signature* is a pair $\langle S, \Omega \rangle$ where

- S is a set of *sort symbols*;
- Ω is an $S^* \times S$-indexed family of sets of *function symbols*.

The sets are finite and mutually disjoint.

The signature of a program is given by its data types, its attributes, and its action names. Each attribute must be of some data type. An action is also typed, namely by the set of local attributes it can change.

Definition 2 A *program signature* is a tuple $\langle \Sigma, V, R, \Gamma \rangle$ where

- $\Sigma = \langle S, \Omega \rangle$ is a data signature;
- $V = \bigcup_{s \in S} V_s$ is a set of *local attributes*;
- $R = \bigcup_{s \in S} R_s$ is a set of *external attributes*;
- $\Gamma = \bigcup_{d \subseteq V} \Gamma_d$ is a set of *actions*.

The sets V_s, R_s and Γ_d are finite and mutually disjoint. The *domain* of an action $g \in \Gamma$ is the set $d \subseteq V$ such that $g \in \Gamma_d$.

Notation. The program attributes are $A = \bigcup_{s \in S} A_s = \bigcup_{s \in S} (V_s \cup R_s)$. The sort of attribute a will be denoted by s_a.

The domain of action g is denoted by $D(g)$. Inversely, for each $a \in V$ the set of actions that can change a is $D(a) = \{g \in \Gamma : a \in D(g)\}$. □

The signature of a program basically defines what expressions and propositions can be written in the program's body. To make the formalization easier, the languages of expressions and propositions to be introduced will be minimal. Other constructs (like set intersection and logical disjunction) can be defined as abbreviations.

Definition 3 Given a program signature $\langle \Sigma, V, R, \Gamma \rangle$ the language of *terms* of sort $s \in S$ is defined by

$$t_s ::= a \mid c \mid f(t_{s_1}, \ldots, t_{s_n})$$

where $a \in A_s$, $c \in \Omega_{\langle\rangle,s}$, and $f \in \Omega_{\langle s_1,\ldots,s_n\rangle,s}$. The language of *set expressions* of sort $s \in S$ is defined by

$$e_s ::= \{t_s,\ldots,t_s\} \mid s \mid (e_s \setminus e_s)$$

where $\{t_s,\ldots,t_s\}$ is possibly empty. The language of *propositions* is defined by

$$\phi ::= t_s = t_s \mid e_s \subseteq e_s \mid (\phi \wedge \phi) \mid (\neg\phi)$$

A program's body defines the data types it uses, the initial values of its local attributes, and the body of each action.

Definition 4 A *program* is a pair $\langle\theta,\Delta\rangle$ where $\theta = \langle\Sigma,V,R,\Gamma\rangle$ is a program signature and $\Delta = \langle\Phi,I,F,B\rangle$ is a program *body* where

- Φ is a (first-order) axiomatization of Σ;
- I is a proposition over V;
- F assigns to every $g \in \Gamma$ and to every $a \in D(g)$ a set expression of sort s_a;
- B assigns to every $g \in \Gamma$ a proposition over A.

Notation. If $D(g)$ is empty, then F will be denoted by `skip`. □

2.2 The model

The semantics of COMMUNITY will be based on traces. We start with the usual algebraic semantics of a signature.

Definition 5 An *algebra* $\mathcal{U}$ for a data signature $\langle S,\Omega\rangle$ assigns a set $s_{\mathcal{U}}$ to each sort $s \in S$, and a total function $f_{\mathcal{U}} : s_{1_{\mathcal{U}}} \times \cdots \times s_{n_{\mathcal{U}}} \to s_{\mathcal{U}}$ to each function symbol $f \in \Omega_{\langle s_1,\ldots,s_n\rangle,s}$.

An interpretation for a program must indicate at each instant i, given by a natural number, the value of each attribute and the actions that are executed.

Definition 6 An *interpretation structure* for a program signature $\langle\Sigma,V,R,\Gamma\rangle$ is a triple $\langle\mathcal{U},\mathcal{A},\mathcal{G}\rangle$ where

- $\mathcal{U}$ is an algebra for $\Sigma = \langle S,\Omega\rangle$;
- $\mathcal{A}$ is an S-indexed family of total functions $\mathcal{A}_s : A_s \times \omega \to s_{\mathcal{U}}$;
- $\mathcal{G} : \omega \to 2^{\Gamma}$ is a total function.

Notation. Dually, we define $\mathcal{G}(g) = \{i \in \omega | g \in \mathcal{G}(i)\}$. □

The above definition allows many traces to be models of the execution of some program. However, we will be only interested in those that obey the following locality principle: the value of local attributes can only be changed by the program itself, not by the environment. In terms of the model, it means that if the actions executing at instant i do not include the local attribute a in their domain, then its value remains the same.

Definition 7 An interpretation structure $\langle\mathcal{U},\mathcal{A},\mathcal{G}\rangle$ for program signature $\langle\Sigma,V,R,\Gamma\rangle$ is a *locus* iff $\forall i \in \omega\, \forall s \in S\, \forall a \in V_s\, D(a) \cap \mathcal{G}(i) = \emptyset \Rightarrow \mathcal{A}_s(a,i) = \mathcal{A}_s(a,i+1)$.

An interpretation structure allows us to evaluate any expression or proposition at any instant.

Definition 8 Given an interpretation structure $\mathcal{S} = \langle \mathcal{U}, \mathcal{A}, \mathcal{G} \rangle$ for program signature $\langle \Sigma, V, R, \Gamma \rangle$, the *denotation* of term t (or set expression e) of sort s at instant i, written $[\![t]\!]_{\mathcal{S}}(i)$ (resp. $[\![e]\!]_{\mathcal{S}}(i)$), is the element (resp. the subset) of $s_{\mathcal{U}}$ defined as follows:

$$\begin{array}{lll} [\![t]\!]_{\mathcal{S}}(i) & ::= & \mathcal{A}_s(a,i) \mid c_{\mathcal{U}} \mid f_{\mathcal{U}}([\![t_1]\!]_{\mathcal{S}}(i), \ldots, [\![t_n]\!]_{\mathcal{S}}(i)) \\ [\![e]\!]_{\mathcal{S}}(i) & ::= & \{[\![t_1]\!]_{\mathcal{S}}(i), \ldots, [\![t_n]\!]_{\mathcal{S}}(i)\} \mid s_{\mathcal{U}} \mid [\![e_1]\!]_{\mathcal{S}}(i) \setminus [\![e_2]\!]_{\mathcal{S}}(i) \end{array}$$

where $a \in A_s$, $c \in \Omega_{\langle\rangle,s}$, and $f \in \Omega_{\langle s_1,\ldots,s_n\rangle,s}$.

Definition 9 Given an interpretation structure $\mathcal{S}$ for program signature θ, the interpretation of proposition ϕ at instant i, written $(\mathcal{S}, i) \models \phi$, is defined as follows:

- $(\mathcal{S}, i) \models t_1 = t_2$ iff $[\![t_1]\!]_{\mathcal{S}}(i) = [\![t_2]\!]_{\mathcal{S}}(i)$;
- $(\mathcal{S}, i) \models e_1 \subseteq e_2$ iff $[\![e_1]\!]_{\mathcal{S}}(i) \subseteq [\![e_2]\!]_{\mathcal{S}}(i)$;
- $(\mathcal{S}, i) \models (\phi_1 \wedge \phi_2)$ iff $(\mathcal{S}, i) \models \phi_1$ and $(\mathcal{S}, i) \models \phi_2$;
- $(\mathcal{S}, i) \models (\neg\phi)$ iff $(\mathcal{S}, i) \not\models \phi$.

For an interpretation to correspond to a model for a given program it must satisfy the following requirements:

- at instant 0 the initialization condition must be true;
- an action can be executed only if its guard is true;
- if the assignment $a := F(g,a)$ of action g is executed at some instant, the value of a in the next instant is some element of the set denoted by the expression $F(g,a)$.

Definition 10 A *model* for a program $\langle \theta, \Delta \rangle$ is an interpretation structure $\mathcal{S} = \langle \mathcal{U}, \mathcal{A}, \mathcal{G} \rangle$ for θ such that:

- $\mathcal{U}$ satisfies the axioms Φ;
- $(\mathcal{S}, 0) \models I$;
- $\forall i \in \omega \, \forall g \in \mathcal{G}(i) \; (\mathcal{S}, i) \models B(g)$;
- $\forall i \in \omega \, \forall g \in \mathcal{G}(i) \, \forall a \in D(g) \; \mathcal{A}(a, i+1) \in [\![F(g,a)]\!]_{\mathcal{S}}(i)$.

Notice that according to this definition an action will never be executed if it includes an assignment $a := \emptyset$. Moreover, if at a given instant two or more actions would assign different values to the same attribute, then only one of them may be selected for execution at that instant.

2.3 The morphisms

Two programs will be related by a superposition morphism if one is a component of the other. In particular, this means that every data type and predefined function of the component must correspond to some type or function of the system.

Definition 11 Given data signatures $\Sigma = \langle S, \Omega \rangle$ and $\Sigma' = \langle S', \Omega' \rangle$, a *data morphism* $\delta : \Sigma \to \Sigma'$ is a total function $\delta_\sigma : S \to S'$ and a $S^* \times S$-indexed family of total functions $\delta_\omega : \Omega_{\langle s_1,\ldots,s_n\rangle,s} \to \Omega'_{\langle \delta_\sigma(s_1),\ldots,\delta_\sigma(s_n)\rangle,\delta_\sigma(s)}$.

Notation. Henceforth the indices σ and ω will be omitted. □

In the previous definitions of COMMUNITY [5, 2], a morphism between two programs P and P' was just a mapping between their attributes and their actions. In this paper we introduce a small but fundamental change. In a mobile setting, a program may synchronize each of its actions with different actions from different programs at different times. To allow this, a program morphism may associate an action g of the base program P with a *set* of actions $\{g_1, \ldots, g_n\}$ of the superposed program P'. The intuition is that those actions correspond to the behaviour of g when synchronizing with other actions of other components of P'. Each action g_i must preserve the basic functionality of g, adding the functionality of the action that has been synchronized with g. The morphism is quite general. First, the set $\{g_1, \ldots, g_n\}$ may be empty. In that case, action g has been effectively removed from P'. Put in other words, it has been permanently inhibited, as if the guard had been made false. Second, it is not required that only one of the g_i can execute at each instant.

Morphisms must preserve the types, the locality, and the domain of attributes. Preserving locality means that local attributes are mapped to local attributes, and preserving domains means that new actions of the system are not allowed to change local attributes of the components.

Definition 12 Given program signatures $\theta = \langle \Sigma, V, R, \Gamma \rangle$ and $\theta' = \langle \Sigma', V', R', \Gamma' \rangle$, a *signature morphism* $\sigma : \theta \to \theta'$ consists of a data morphism $\sigma_\delta : \Sigma \to \Sigma'$ together with total function $\sigma_\alpha : A \to A'$ and partial function $\sigma_\gamma : \Gamma' \to \Gamma$ such that

- $\forall s \in S\ \sigma_\alpha(V_s) \subseteq V'_{\sigma_\delta(s)} \wedge \sigma_\alpha(R_s) \subseteq A'_{\sigma_\delta(s)}$;
- $\forall a \in V\ \sigma_\gamma(D'(\sigma_\alpha(a))) \subseteq D(a)$;
- $\forall g' \in \Gamma'\ \sigma_\alpha(D(\sigma_\gamma(g'))) \subseteq D'(g')$.

Notation. In the following, the indices α, δ, and γ will be omitted. We will denote the pre-image of σ_γ by $\sigma^{\leftarrow}$. Also, if x is a term (or expression or proposition) of θ, then $\sigma(x)$ is the term (resp. expression or proposition) of θ' obtained in the usual way. □

Our first result is that signatures and their morphisms constitute a category. This basically asserts that morphisms can be composed. In other words, the "component-of" relation is transitive (and reflexive, of course).

Proposition 1 *Program signatures and signature morphisms constitute a category $\mathcal{SIG}$.*

Given an interpretation for some signature of some program, morphisms allow us to obtain interpretations for its components.

Definition 13 Given a signature morphism $\sigma : \theta_1 \to \theta_2$ and an interpretation structure $\mathcal{S}$ for θ_2, the σ-*reduct* is the interpretation structure $\mathcal{S}|_\sigma = \langle \mathcal{U}|_\sigma, \mathcal{A}|_\sigma, \mathcal{G}|_\sigma \rangle$ for θ_1 where $\mathcal{U}|_\sigma(x) = \mathcal{U}(\sigma(x))$, $\mathcal{A}|_\sigma(a,i) = \mathcal{A}(\sigma(a),i)$ and $\mathcal{G}|_\sigma(g) = \bigcup_{g' \in \sigma^{\leftarrow}(g)} \mathcal{G}(g')$.

Superposition of a program P' on a base program P is captured by a morphism between their signatures that obeys the following conditions:

- the abstract data types do not have less properties;
- the initialization condition is not weakened;
- the assignments are not less deterministic;
- the guards are not weakened.

Definition 14 A *superposition morphism* $\sigma : \langle \theta, \Delta \rangle \to \langle \theta', \Delta' \rangle$ is a signature morphism $\sigma : \theta \to \theta'$ such that

- $\Phi' \models_{\theta'} \sigma(\Phi)$;
- $\Phi' \models_{\theta'} I' \Rightarrow \sigma(I)$;
- $\forall g \in \Gamma\ \forall a \in D(g)\ \forall g' \in \sigma^{\leftarrow}(g)\ \Phi' \models_{\theta'} F'(g', \sigma(a)) \subseteq \sigma(F(g,a))$;
- $\forall g \in \Gamma\ \forall g' \in \sigma^{\leftarrow}(g)\ \Phi' \models_{\theta'} B'(g') \Rightarrow \sigma(B(g))$.

where $\models_{\theta'}$ means validity in first-order sense.

Program morphisms preserve the locality principle.

Proposition 2 *If $\sigma : \langle \theta, \Delta \rangle \to \langle \theta', \Delta' \rangle$ is a superposition morphism then the reduct of every locus of $\langle \theta', \Delta' \rangle$ is a locus of $\langle \theta, \Delta \rangle$.*

The category of signatures extends to programs.

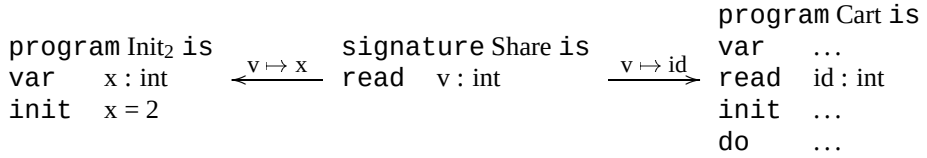
Proposition 3 *Programs and superposition morphisms constitute a category $\mathcal{PROG}$.*

3 The Architecture

The configuration of a system is described by a diagram of components and channels. The components are programs, and the channels are given by signatures that specify how the programs are interconnected. Given programs P and P', the signature S is constructed as follows: for each pair of attributes (or actions) $a \in P$ and $a' \in P'$ that are to be shared (resp. synchronized), the signature contains one attribute (resp. action) b; the morphism from S to P maps b to a and the morphism from S to P' maps b to a'. We have morphisms only between signatures or only between programs, but it can be proven that a signature can be seen as a program with an "empty" body.

Proposition 4 *Category $\mathcal{SIG}$ is fully embedded in category $\mathcal{PROG}$.*

As a simple example, consider the following diagram, which connects the generic cart program with a program that initializes an integer attribute with the value 2 through a channel that represents attribute sharing.



The program that describes the whole system is given by the colimit of the diagram, which can be obtained by computing the pushouts of pairs of components with a common channel. The program P resulting from the pushout of P_1 and P_2 is obtained as follows. The initialization condition is the conjunction of the initialization conditions of the components, and the attributes of P are the union of the attributes of P_1 and P_2, renaming them such that only those that are to be shared will have the same name. An attribute of P is local only if it is local in at least one component.

For the above example, the resulting pushout will represent the cart with identifier 2.

```
program Cart2 is
var    bag : int; λ, dest : int; id : int;
read   nbag : int;
init   bag = 0 ∧ dest = InitDest(id) ∧ λ = InitLoc(id) ∧ id = 2
do     ...
```

As for the actions of P, they are basically a subset of all pairs of actions g_1g_2 where $g_i \in \Gamma_i$ (for $i = 1,2$). Only those pairs such that g_1 and g_2 are mapped to the same action of the channel may appear in P. If an action of P_1 (or P_2) is not mapped to any action of the channel—i.e., it is not synchronized with any action of P_2 (resp. P_1)—then it appears "unpaired" in P. Synchronizing two actions g_1 and g_2 (i.e., joining them into a single one g_1g_2) involves taking the union of their domains, the conjunction of their guards, and the parallel composition of their assignments. If the actions have a common attribute a then the resulting assignment is $a := F(g_1,a) \cap F(g_2,a)$. If the actions are "incompatible" then the resulting set expression returns the empty set and therefore the synchronized action will never execute, as expected. An example is given in Section 4.2.

The next result states that every finite diagram has a colimit.

Proposition 5 *Category $\mathcal{PROG}$ is finitely cocomplete.*

Channels (i.e., signatures) only allow us to express simple static connections between programs. To express more complex or transient interactions, we will use connectors, a basic concept of Software Architecture [12]. A connector consists of a glue linked to one or more roles through channels. The roles constrain what objects the connector can be applied to. In a categorical framework, the connectors (and therefore the architectures) that can be built depend on the categories used to represent glues, roles, and channels, and on the relationships between those categories. It is possible to use three different categories for the three parts of a connector (e.g., [2] proposes roles to be specifications written in temporal logic) but for simplicity we will assume that roles and glues are members of the same category. We therefore adopt only the basic definitions of [2].

Definition 15 An *architectural school* is a full embedding $\mathcal{F}$ of a category $\mathcal{CHAN}$ into a category $\mathcal{DESC}$ which admits colimits on all finite diagrams in which shared objects are of the form $\mathcal{F}(C)$ where $C : \mathcal{CHAN}$.

A *connection* is a tuple $\langle C, G, R, \gamma, \rho \rangle$ where

- $C : \mathcal{CHAN}$ is the *channel*;
- $G : \mathcal{DESC}$ is the *glue*;
- $R : \mathcal{DESC}$ is the *role*;
- $\gamma : \mathcal{F}(C) \to G$ and $\rho : \mathcal{F}(C) \to R$ are morphisms in $\mathcal{DESC}$.

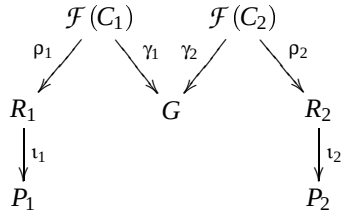
A *connector* is a finite set of connections with the same glue.

The semantics of a connector is given by the colimit of the connections diagram. By definition, there are superposition morphisms from each object in the diagram to the colimit. Therefore superposition becomes in a sense "symmetric", a necessary property to capture interaction [9].

A connector can be applied only to programs which are instantiations of the roles. In categorical terms, there must exist morphisms from the roles to the programs.

Definition 16 Given an architectural school $\mathcal{F} : \mathcal{CHAN} \to \mathcal{DESC}$, a *correct instantiation* of a connector $\{\langle C_i, G, R_i, \gamma_i, \rho_i \rangle\}$ with descriptions P_i is a set of morphisms $\iota_i : R_i \to P_i$ in $\mathcal{DESC}$. The resulting system is the colimit of the diagram formed by morphisms γ_i and $\iota_i \circ \rho_i$.

As an illustration, an instantiated connector with two roles has the following diagram:



4 Interactions

An interaction between two programs involves conditions and computations. Therefore it cannot be specified just by a signature; we must use a connector, where the programs are instances of the roles, the interaction is the glue, and each channel states exactly what is the part of each program in the interaction.

Distributed systems may consist of many components, but usually they can be classified into a relatively small set of different types. Since interaction patterns normally do not depend on the individual components but on their types, it is only necessary to define connectors for the existing component types. To obtain the resulting system, the connectors will be instantiated with the actual components. Therefore, in the following we will only consider the programs that correspond to component types. In the luggage distribution example there are only three different program types: carts, loaders, and unloaders. The programs for the individual components only differ in the initialization condition for the identifier attribute.

In a mobile setting one of the important aspects of interactions is their temporary nature. This is represented by conditions: an interaction takes place only while some proposition is true. Usually that proposition is based on the location of the interacting parties. We will consider three kinds of interactions:

inhibition An action may not execute.[2]

synchronization Two actions are executed simultaneously.

communication The values of some local attributes of one program are passed to corresponding external attributes of the other program.

For each kind of interaction we develop a connector template which is parameterized by the interaction conditions. This means that, given the interacting programs (i.e., the roles) and the conditions under which they interact, the appropriate connector can be instantiated.

Given the set of components that will form the overall system, the possible interactions are specified as follows:

- An inhibition interaction states that an action g of some program P will not be executed whenever the interaction condition I is true.
- A synchronization interaction states that action g of program P will execute simultaneously with action g' of program P' whenever I is true.
- A communication interaction states that the value of the local attributes M (the "message") of program P can be written into the external attributes M' of program P' if I is true. The sets M and M' must be compatible. Moreover, each program must indicate which action is immediately executed after sending (resp. receiving) the message.

To make the formal definitions easier, we will consider that all interacting programs use the same data types.

Definition 17 Given a set $\mathcal{P}$ of programs over the same abstract data types $\langle \Sigma, \Phi \rangle$, a *transient interaction* is either one of the following:

- a *transient inhibition* $\langle g, P, I \rangle$;
- a *transient synchronization* $\langle g, P, g', P', I \rangle$;
- a *transient communication* $\langle g, M, P, g', M', P', I \rangle$;

where

- $P, P' \in \mathcal{P}$;
- $M \subseteq V, M' \subseteq R'$ and there is a bijection $f : M \to M'$ such that $\forall a \in M\ s_a = s_{f(a)}$;
- $g \in \Gamma, g' \in \Gamma'$;
- I is a proposition over attributes of $\mathcal{P}$.

[2] In this case the interaction is between the program and its environment.

The interaction condition I will be written according to the signature of the glue, and the channels will show for each attribute occurring in I to which role it belongs. We will write A_I to denote the attributes that appear in I. In the following we will assume I uses only attributes from P and P': $A_I \subseteq A \uplus A'$. If not, the instantiated connector must have further roles that provide the remaining attributes. The example in the next section will provide such a case.

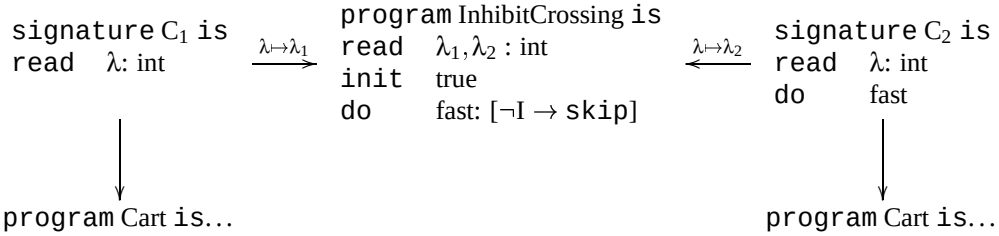
4.1 Inhibition

Inhibition is easy and elegant to express: if an action is not to be executed while I is true, then it can be executed only while $\neg I$ is true.

Definition 18 The *inhibition connector pattern* corresponding to inhibition interaction $\langle g, P, I\rangle$ consists of just one connection $\langle C, G, P, \gamma, \rho\rangle$ where

- $G \equiv$ `read` A_I
 `init` true
 `do` $g : [\neg I \rightarrow$ `skip`$]$
- $C \equiv$ `read` A_I
 `do` g
- γ and ρ are injections.

Returning to our example, we want to prevent carts from colliding at intersections. We achieve that goal in two steps, the second of which to be presented in the next section. The first step is the following: when two carts enter two segments that intersect, the one further away from the intersection is only allowed to move slowly. In other words, its `fast` action is inhibited. Notice that in this case the inhibition depends on the presence of another cart, and therefore the connector has two roles. The diagram is



and I=CrossingSegments(λ_1,λ_2)$\wedge$DistanceToCrossing(λ_1)$<$DistanceToCrossing(λ_2).

An application of this connector and the resulting colimit will be presented in the next subsection.

4.2 Synchronization

Synchronizing two actions g and g' of two different components can be seen as merging them into a single action gg' of the system, the only difference between the static and the mobile case being that in the latter the merging is only done while some condition is true. When gg' executes, it corresponds to the simultaneous execution of g and g'. Therefore, if g would be executed by a component, the system will in fact execute gg' which means that it is also executing g', and vice-versa. To sum it up, when two actions synchronize either both execute simultaneously or none is executed.

This contrasts with the approach taken by Mobile UNITY which allows two kinds of synchronization: coexecution and coselection [9]. The former corresponds to the notion exposed above, while the latter forces the two actions to be selected simultaneously but if one of them is inhibited or its guard is false then only the other action executes. This extends the basic semantics of UNITY where only one action can be selected at a time. Since COMMUNITY already allows (but does not impose) simultaneous selection of multiple actions, and because we believe that the intuitive notion of synchronization corresponds to coexecution, we will not handle coselection.

The key to represent synchronization of two actions subject to condition I is to unfold each action in two, one corresponding to its execution when I is false and the other one when I is true. Put in other words, each action has two "sub-actions", one for the normal execution and the other for synchronized execution. As the normal sub-action can only execute when the condition is false, it is inhibited when I is true, and the opposite happens with the synchronization sub-action. Therefore we can use the same technique as for inhibition. Since there are two actions to be synchronized, and the synchronization sub-action must be shared by both, there will we three (instead of four) sub-actions. To facilitate understanding, the name of a sub-action will be the set of the names of the actions it is part of.

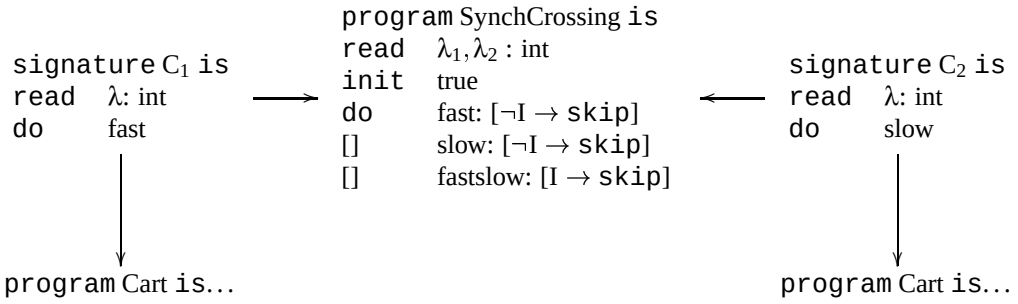
Definition 19 The *synchronization connector pattern* corresponding to synchronization interaction $\langle g, P, g', P', I\rangle$ consists of two connections $\langle C, G, P, \gamma, \rho\rangle$ and $\langle C', G, P', \gamma', \rho'\rangle$ where

- $G \equiv$ `read` A_I
  `init` true
  `do` $g : [\neg I \to$ `skip`$]$
  $[]$ $g' : [\neg I \to$ `skip`$]$
  $[]$ $gg' : [I \to$ `skip`$]$
- $C \equiv$ `read` $A_I \cap A$
  `do` g
- $C' \equiv$ `read` $A_I \cap A'$
  `do` g'
- γ, ρ, γ', and ρ' are injection morphisms with the exceptions $\gamma(g) = \{g, gg'\}$ and $\gamma'(g') = \{g', gg'\}$.

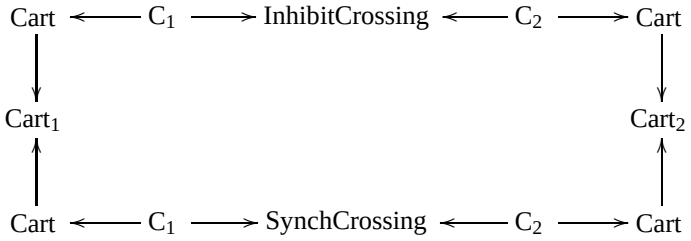
In the colimit, the action gg' will have the guards and the assignments of g and g'. Therefore, if either $B(g)$ or $B(g')$ is false, or if the assignments are incompatible, then gg' will not get executed.

This connector describes what is called "non-exclusive coexecution" in [9]: outside the interaction period the actions execute as normal. It is also possible to simulate exclusive coexecution which means that the actions are only executed (synchronously) when the interaction condition is true. To that end, simply eliminate actions g and g' from the inhibition connector shown above, just keeping the synchronized action gg'.

Continuing with the example, the second step to avoid collisions at crossings is to force the two carts to move simultaneously. Since the most distant cart can only move slowly, the nearest cart is guaranteed to pass the crossing first. Using the same interaction condition as in the previous section one gets the following diagram.



To prevent collisions between Cart_1 and Cart_2 (obtained as shown in Section 3) one must consider two symmetrical cases, depending on which cart is nearer to the intersection. Let us assume that Cart_1 is nearer. Thus we must block the `fast` action of Cart_2 with the inhibitor shown in the previous section and synchronize its `slow` action with the `fast` action of Cart_1 using the connector above. The diagram is

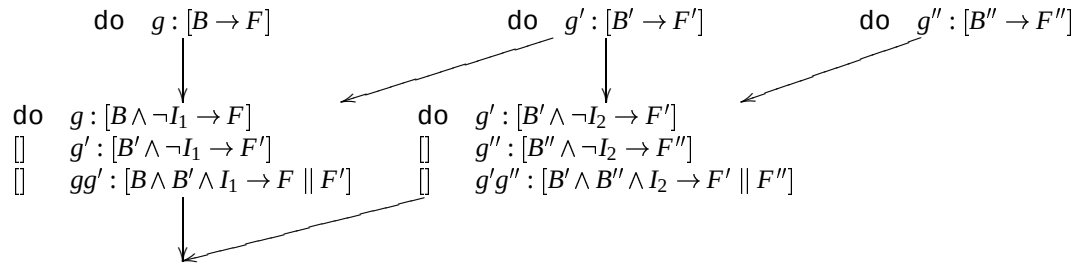


with the following pushout (where i ranges over 1 and 2 to abbreviate code duplication)

`program` System `is`
`var` bag_i : int; λ_i, dest_i : int; id_i : int;
`read` nbag_i : int;
`init` $\text{bag}_i = 0 \wedge \text{dest}_i = \text{InitDest}(\text{id}_i) \wedge \lambda_i = \text{InitLoc}(\text{id}_i) \wedge \text{id}_i = i$
`do` slow_1: $[\lambda_1 \neq \text{dest}_1 \rightarrow \lambda_1 := \lambda_1 + 1]$
[] fast_1: $[\lambda_1 \neq \text{dest}_1 \wedge \neg\text{I} \rightarrow \lambda_1 := \lambda_1 + 2]$
[] slow_2: $[\lambda_2 \neq \text{dest}_2 \wedge \neg\text{I} \rightarrow \lambda_2 := \lambda_2 + 1]$
[] $\text{fast}_1\text{slow}_2$: $[\lambda_1 \neq \text{dest}_1 \wedge \lambda_2 \neq \text{dest}_2 \wedge \text{I}$
$\rightarrow \lambda_1 := \lambda_1 + 2 \parallel \lambda_2 := \lambda_2 + 1]$
[] fast_2: $[\lambda_2 \neq \text{dest}_2 \wedge \neg\text{I} \rightarrow \lambda_2 := \lambda_2 + 2]$
[] load_i: $[\lambda_i = \text{dest}_i \wedge \text{bag}_i = 0$
$\rightarrow \text{bag}_i := \text{nbag}_i \parallel \text{dest}_i := \text{Dest}(\text{nbag}_i, \text{dest}_i)]$
[] unload_i: $[\lambda_i = \text{dest}_i \wedge \text{bag}_i \neq 0 \rightarrow \text{bag}_i := 0 \parallel \text{dest}_i := \text{Next}(\text{dest}_i)]$

To see that synchronization is transitive, consider the following example where action g' is synchronized with two other actions g and g'' whenever I_1 and I_2 are true, respectively. The resulting system must provide actions for all four combinations of the truth values of the interaction conditions. For example, if $I_1 \wedge I_2$ is true then all actions *must* occur simultaneously, but if $I_1 \vee I_2$ is false, then any subset of the actions can occur. This happens indeed because the pushout of two morphisms $\gamma(g) = \{g_1, \ldots, g_n\}$ and $\gamma'(g) = \{g'_1, \ldots, g'_m\}$ is basically given by the pairs $\{g_1g'_1, \ldots, g_1g'_m, \ldots, g_ng'_m\}$ with morphism $\sigma(g_i) = \{g_ig'_1, \ldots, g_ig'_m\}$ and $\sigma'(g'_j) = \{g_1g'_j, \ldots, g_ng'_j\}$. Putting into words: if an action g "unfolds" into actions $\sigma(g) = \{g_1, \ldots, g_n\}$, it means that whenever g would be executed, any subset of $\sigma(g)$ executes in the superposed program, and vice-versa, the execution of any g_i implies that g is executed in the base program. Therefore, if g can be unfolded in two distinct ways, in the pushout any combination of the sub-

actions can occur whenever g executes. The pushout morphisms just state to which combinations each sub-action belongs.



do $g : [B \wedge \neg I_1 \to F]$
[] $g' : [B' \wedge \neg I_1 \wedge \neg I_2 \to F']$
[] $g'' : [B'' \wedge \neg I_2 \to F'']$
[] $gg' : [B \wedge B' \wedge I_1 \wedge \neg I_2 \to F \parallel F']$
[] $g'g'' : [B' \wedge B'' \wedge \neg I_1 \wedge I_2 \to F' \parallel F'']$
[] $gg'g'' : [B \wedge B' \wedge B'' \wedge I_1 \wedge I_2 \to F \parallel F' \parallel F'']$
[] $gg'' : [B \wedge B'' \wedge \neg I_1 \wedge \neg I_2 \to F \parallel F'']$

As one can see, for all combinations of I_1 and I_2 the correct actions are executed. The colimit includes the combination of all actions that share the name g': actions g' and gg' of the left middle pushout are synchronized with g' and $g'g''$ on the right in the four possible ways.

4.3 Communication

In Mobile UNITY communication is achieved through variable sharing. The interaction $x \approx y$ `when` C `engage` I `disengage` $F_x \parallel F_y$ states the sharing condition C, the (shared) initial value I of both variables, and the final value F_x and F_y of each variable. The operational semantics states that whenever a program changes x, y gets the same value, and vice-versa. This approach violates the locality principle. Furthermore, as pointed out in [9], several restrictions have to be imposed in order to avoid problems like, e.g., simultaneous assignments of different values to shared variables.

We also feel that communication is a more appropriate concept than sharing for the setting we are considering, namely mobile agents that engage into transient interactions over some kind of network. In the framework of COMMUNITY programs, communication can be seen as some kind of sharing of local and external attributes, which keeps the locality principle. We say "some kind" because we cannot use the same mechanism as in the static case, in which sharing meant to map two different attributes of the components into a single one of the system obtained by the colimit. In the mobile case the same local attribute may be shared with different external attributes at different times, and vice-versa. If we would apply the usual construction, all those attributes would become a single one in the resulting system, which is clearly unintended.

We therefore will obtain the same effect as transient sharing using a communication perspective. To be more precise, we assume program P wants to send a message M, which is a set of local attributes. If P' wants to receive the message, it must provide external attributes M' which correspond in number and type to those of M. Program P produces the values, stores them in M, and waits for the message to be read by P'. Since COMMUNITY programs are not sequential, "waiting" has to be understood in a restricted sense. We only assume that P will not produce another message before the previous one has been read (i.e., messages are not lost); it may however be executing

other unrelated actions. To put it in another way, after producing M, program P is expecting an acknowledge to produce the new values for the attributes in M. For that purpose, we assume P has an action g which must be executed before the new message is produced. Similarly, program P' must be informed when a new message has arrived, so that it may start processing it. For that purpose we assume that P' has a single action g' which is the first action to be executed upon the receipt of a new message[3]. That action may simply start using M' directly or it may copy it to local attributes of P'.

To sum up, communication is established via one single action for each program[4]: the action g of P is waiting for M to be read, the action g' of P' reads M (i.e., starts using the values in M'). As expected, it is up for the glue of the interaction connector to transfer the values from M to M' and to notify the programs.

The solution is to explicitly model the message transmission as the parallel assignment of the message attributes, which we will abbreviate as $M' := M$. For this to be possible, the local attributes M of P must be external attributes of the glue, and the external attributes M' of P' must be local attributes of the glue. The assignment can be done in parallel with the notification of P. Moreover, the programs may only communicate when proposition I is true. Therefore the glue contains an action $wait : [I \to M' := M]$ to be synchronized with the "waiting" action g of P. The "reading" action g' of P' can only be executed after the message has been transmitted. The solution is to have another action *read* in the glue that is synchronized with g'. To make sure that *read* is executed after *wait* we use a boolean attribute. Thus g' is inhibited while no new values have been transferred to M'. Again, this is like a blocking `read` primitive, except that P' may execute actions unrelated to M'.

Since a receiver may get messages from different senders $i = 1,\ldots,n$ (at different times or not), there will be several possible assignments $M' := M_i$. Due to the locality principle, all assignments to a attribute must be in a single program. Therefore for each message type a receiver might get, there will be a single glue connecting it to all possible senders. On the other hand, a message might be sent to different receivers $j = 1,\ldots,m$. Therefore there will be several possible assignments $M'_j := M$ associated with the same wait action of the sender of message M. So there must be a single glue to connect a sender with all its possible recipients. To sum up, for each message type there will be a single glue acting like a "demultiplexer": it synchronizes sender i with receiver j when interaction condition I_{ij} is true. This assumes that the possible communication patterns are known in advance.

Definition 20 The *communication connector pattern* corresponding to communication interactions $\langle g_i, M_i, P_i, g'_j, M'_j, P'_j, I_{ij}\rangle$ (for $i = 1,\ldots,n$ and $j = 1,\ldots,m$) consists of connections $\langle C_i, G, P_i, \gamma_i, \rho_i\rangle$ and $\langle C'_j, G, P'_j, \gamma'_j, \rho'_j\rangle$ where

- $G \equiv$ `var` M'_j; new_j : bool
  `read` $M_i; \bigcup_{i,j} A_{I_{ij}} \setminus M'_j$
  `init` $\neg new_j$
  `do` $\text{wait}_{ij} : [I_{ij} \wedge \neg new_j \to M'_j := M_i \parallel new_j := \text{true}]$
  [] $\text{read}_j : [new_j \to new_j := \text{false}]$
- $C_i \equiv$ `read` $M_i; A_i \cap \bigcup_{i,j} A_{I_{ij}}$
  `do` wait_i

[3]It is always possible to write P' in such a way.
[4]This is similar to pointed processes in the π-calculus, or to ports in distributed systems.

- $C'_j \equiv$ `read` $M'_j; A'_j \cap \bigcup_{i,j} A_{I_{ij}}$
  `do` read_j

- $\gamma_i, \rho_i, \gamma'_j, \rho'_j$ are injections except that $\gamma_i(g_i) = \{\text{wait}_{i1}, \ldots, \text{wait}_{im}\}$.

Notice that several actions wait_{ij} may occur simultaneously, in particular for the same receiver j if the messages sent have the same value. To distinguish messages sent by different senders, even if their content is the same, one can add a local integer attribute `s` to the glue and add the assignment `s := i` to each action $\texttt{wait}_{ij}$. This prevents two different senders from sending their messages simultaneously.

In the luggage delivery example, communication takes place when a cart arrives at a station (i.e., a loader or an unloader), the bag being the exchanged message. Loaders are senders, unloaders are receivers, and carts have both roles. The bags held by a station will be stored in a attribute of type queue of integers. Although the locations of stations are fixed they must be represented explicitly in order to represent the communication condition, namely that cart and station are co-located. Since it is up for the connector to describe the interaction, the programs for the stations just describe the basic computations: loaders remove bags from their queues, unloaders put bags on their queues. Notice how the loader program has separate actions to produce the message (i.e., the computation of the value of the `bag` attribute) and to send the message (i.e., the bag has been loaded onto the cart).

```
program Loader is
var   bag : int; cargo : queue(int); loaded : bool; λ: int;
read  id : int;
init  loaded ∧ λ= InitLoc(id)
do    newbag: [cargo ≠ ∅ ∧ loaded
      → bag := hd(cargo) || cargo := tl(cargo) || loaded := false]
[]    nobag: [cargo = ∅ ∧ loaded → bag := 0 || loaded := false]
[]    load: [¬loaded → loaded := true]

program Unloader is
var   cargo : queue(int); λ: int;
read  id, bag : int;
init  cargo = ∅ ∧ λ = InitLoc(id)
do    unload: [true → cargo := cons(bag, cargo)]
```

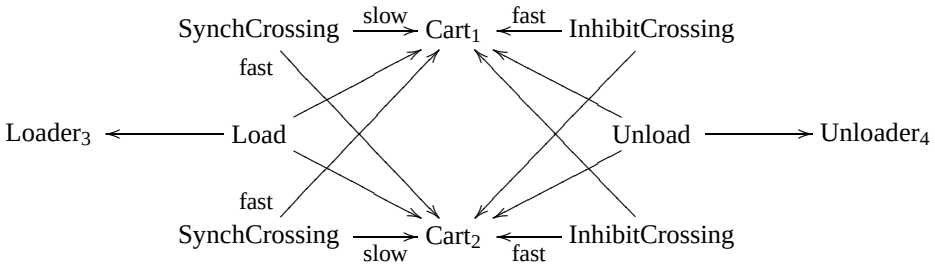
Each unloader (receiver) u_j is connected to every cart (sender) c_i through the glue

```
program Unload is
read  λ_{c_i}, λ_{u_j} : int; new_j : boolean
init  ¬new_j
do    wait_ij: [λ_{c_i} = λ_{u_j} ∧ ¬new_j → bag_{u_j} := bag_{c_i} || new_j := true]
[]    read_j: [new_j → new_j := false]
```

Each cart (receiver) c_i is connected to every loader (sender) l_k through the glue

```
program Load is
read  nbag_{c_i}, bag_{l_k} : int; λ_{c_i}, λ_{l_k} : int; new_i : boolean;
init  ¬new_i
do    wait_ik: [λ_{c_i} = λ_{l_k} ∧ ¬new_i →  nbag_{c_i} := bag_{l_k} || new_i := true]
[]    read_i: [new_i → new_i := false]
```

Let X_i be the program obtained by the pushout of programs Init_i (of Section 3) and X. Then the program corresponding to a system consisting of two carts, one loader, and one unloader is obtained by computing the colimit of the following diagram, which only shows the role instantiation morphisms between the connectors (which have the same name as their glues) and the components. Notice that the binary connectors dealing with crossings are not symmetric; they distinguish which cart is supposed to be nearer to the crossing. Therefore one must apply those connectors twice to each pair of carts.



5 Concluding Remarks

We have shown how some fundamental kinds of transient interactions, inspired by Mobile UNITY [11, 9], can be represented using architectural connectors. The semantics has been given within a categorical framework, and the approach has been illustrated with a UNITY-like program design language [3, 5].

As argued in [3, 4], the general benefits of working within a categorical framework are:

- mechanisms for interconnecting components into complex systems can be formalized through universal constructs;
- extra-logical design principles are internalized through properties of universal constructs;
- different levels of design can be related (e.g., programs and specifications).

For this work in particular, the synergy between Software Architecture and Category Theory resulted in several conceptual and practical advantages.

First, systems are constructed in a principled way: for each interaction kind there is a connector template to be instantiated with the actual interaction conditions; the instantiated connectors are applied to the interacting programs thus forming the system architecture, which can be visualized by a diagram; the program corresponding to the overall system is obtained by "compiling" (i.e., computing the colimit of) the diagram.

Second, separation between computation and coordination, which is already supported by Software Architecture, has been reinforced by two facts. On the one hand, the glue of a connector uses only the signatures of the interacting programs, not their bodies. On the other hand, the superposition morphisms impose the locality principle.

Third, to capture transient interactions, only the morphism between program actions had to be changed; the syntax and semantics of the language remained the same.

There are two ways of dealing with architectures of mobile components. In a system with limited mobility or with a limited number of different component types, all possible interaction patterns can be foreseen, and thus a *static* architecture with all possible interconnections can represent such a system. To cope with systems having

a greater degree of mobility, one must have *evolving* architectures, where components and connectors can be added and removed unpredictably. This paper, being inspired by Mobile UNITY, follows the first approach. Our future work will address the second approach.

Acknowledgements

We would like to thank Antónia Lopes for many fruitful discussions.

References

[1] K. Mani Chandy and Jayadev Misra. *Parallel Program Design—A Foundation*. Addison-Wesley, 1988.

[2] José Luiz Fiadeiro and Antónia Lopes. Semantics of architectural connectors. In *Proceedings of TAPSOFT'97*, volume 1214 of *LNCS*, pages 505–519. Springer-Verlag, 1997.

[3] José Luiz Fiadeiro and Tom Maibaum. Interconnecting formalisms: Supporting modularity, reuse and incrementality. In *SIGSOFT'95: Third Symposium on Foundations of Software Engineering*, pages 72–80. ACM Press, 1995.

[4] José Luiz Fiadeiro and Tom Maibaum. A mathematical toolbox for the software architect. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 46–55. IEEE Computer Society Press, 1996.

[5] José Luiz Fiadeiro and Tom Maibaum. Categorial semantics of parallel program design. *Science of Computer Programming*, 28:111–138, 1997.

[6] Nissim Francez and Ira Forman. *Interacting Processes*. Addison-Wesley, 1996.

[7] David Garlan and Dewayne Perry (eds). Special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[8] Ulf Leonhardt and Jeff Magee. Towards a general location service for mobile environments. In *Third International Workshop on Service in Distributed and Networked Environments*, 1996.

[9] Peter J. McCann and Gruia-Catalin Roman. Mobile UNITY: A language and logic for concurrent mobile systems. Technical Report WUCS-97-01, Department of Computer Science, Washington University in St. Louis, December 1996.

[10] Benjamin C. Peirce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[11] Gruia-Catalin Roman, Peter J. McCann, and Jerome Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. Technical Report WUCS-96-08, Department of Computer Science, Washington University in St. Louis, January 1997. To appear in ACM TOSEM.

[12] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.