Connectors for Mobile Programs

Michel Wermelinger, *Student Member*, *IEEE*, and José Luiz Fiadeiro

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.

Index Terms—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 analyze 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 [1], [2].

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 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 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], [4], a program design language based on UNITY [5], and IP [6].

The nature of the connectors proposed in the paper was motivated and inspired by Mobile UNITY [7], [8], 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 identified connectors, in order to make the architecture of the system more explicit 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 use the same example as in [7]: 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.

In this paper, we follow the approach proposed in [9] 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 superimposed. 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 "ramified," transient action synchronization becomes possible.

Section 2 presents the language and the morphisms, Section 3 defines the architectural diagrams, while Section 4 shows the connector patterns and instantiates them for the example application.

- *M. Wermelinger is with the Departamento de Informática, Universidade Nova de Lisboa, 2825 Monte da Caparica, Portugal. E-mail: mw@di.fct.unl.pt.*
- *J.L. Fiadeiro is with the Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, Campo Grande, 1700 Lisboa, Portugal. E-mail: llf@di.fc.ul.pt.*

Manuscript received 30 June 1997; revised 16 Dec. 1997.
Recommended for acceptance by C.-G. Roman and C. Ghezzi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 106410.

0098-5589/98/$10.00 © 1998 IEEE

2 Mobile Community

The framework to be used consists of programs and their morphisms. This section introduces just the necessary definitions. For a more thorough formal treatment, the interested reader should consult [11].

A COMMUNITY program is basically a set of named, guarded actions. Action names act as *rendezvous* points for program synchronization. At each step, one action whose guard is true executes. Each action consists of one or more assignments to execute simultaneously. Each attribute used by a program is either external—its value is provided by the environment and may change at any time—or local—its value is initialized by the program and modified only by its actions.

Attributes are typed by a fixed algebraic data type specification $\langle S, \Omega, \Phi\rangle$ where S is a set of sort symbols, Ω is an $S^* \times S$-indexed family of function symbols, and Φ is a set of first-order axioms defining the properties of the operations. We do not present the specification of the sorts and predefined functions used in this paper. The main differences to UNITY are: actions are named, all names are local, and there are two kinds of attributes. Except for the latter, Mobile UNITY also has these characteristics.

A COMMUNITY program has the following structure:

program *P* **is**
var *V*
read *R*
init *I*
do $\underset{g\in\Gamma}{[]}\ g : [B(g) \rightarrow \underset{a\in D(g)}{||}\ a := F(g, a)]$

where

- V is the set of *local attributes*, i.e., the program "variables";
- R is the set of *external attributes* used by the program, i.e., read-only attributes that are to be instantiated with local attributes of other components in the environment;
- each attribute is typed by a data sort in S;
- I is the *initialization condition*, a proposition on the local attributes;
- Γ is the set of *action names*, each one having an associated statement (see below);
- for every action $g \in \Gamma$, the *guard* $B(g)$ is a proposition on the attributes stating the necessary conditions to execute g;
- for every action $g \in \Gamma$, its *domain* $D(g)$ is the set of local attributes that g can change;
- for every action $g \in \Gamma$, and local attribute $a \in D(g)$, $F(g, a)$ is a term denoting the value to be assigned to a each time g is executed.

Formally, the signature of a program defines its vocabulary (i.e., its attributes and action names).

DEFINITION 1. *A program signature is a tuple* $\langle V, R, \Gamma\rangle$ *where*:

- $V = \bigcup_{s\in S} V_s$ *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 is 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 $D(a) = \{g \in \Gamma : a \in D(g)\}$.

A program's body defines the initial values of its local attributes and also when and how the actions modify them. For that purpose the body uses propositions and terms built from the program's attributes and the predefined function symbols.

DEFINITION 2. *A program is a pair* $\langle \theta\ \Delta\rangle$ *where* $\theta = \langle V, R, \Gamma\rangle$ *is a program signature and* $\Delta = \langle I, F, B\rangle$ *is a program* body *where*

- I *is a proposition over* V;
- F *assigns to every* $g \in \Gamma$ *and to every* $a \in D(g)$ *a term of sort* s_a;
- B *assigns to every* $g \in \Gamma$ *a proposition over* A.

NOTATION. If $D(g)$ is empty, then F is denoted by **skip**.

Locations are an important aspect of mobility [12]. 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. 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.

The formal treatment of locations is 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.

To give an example of a COMMUNITY program, we present the specification of a cart. Like bags and (un)loaders, carts have unique identifiers, which are 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, each further divided into 10 units. The location of a cart is, therefore, given by an integer. Carts can move at two different speeds: slow (one length unit per time unit) and fast (two length units). 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, 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)]
```

We now turn to program morphisms, the categorical notion that expresses relationships between (certain) pairs of programs. In the previous definitions of COMMUNITY [4], [9], a morphism between two programs P and P' is just a mapping from P's attributes and actions to those of P', stating in which way P is a component of P'. It is therefore called a superposition morphism, since it captures the notion of superposition of [5], P being the underlying program and P' the transformed one.

In this paper, we keep the basic intuition but introduce a small, although 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 superimposed program P'. The intuition is that those actions correspond to the behavior 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: 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. Due to technical reasons the mapping between actions of P and sets of actions of P' is formalized as a partial function from P' to P. However, in examples and informal discussions we use the "set version" of the action mapping.

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 3. *Given program signatures* $\theta = \langle V, R, \Gamma\rangle$ *and* $\theta' = \langle V', R', \Gamma\rangle$, a signature morphism $\sigma : \theta \to \theta'$ *consists of a total function* $\sigma_\alpha : A \to A'$ *and a partial function* $\sigma_\gamma : \Gamma' \to \Gamma$ *such that*

- $\forall s \in S\ \sigma_\alpha(V_s) \subseteq V'_s \wedge \sigma_\alpha(R_s) \subseteq A'_s$;
- $\forall a \in V \sigma_\gamma(D'(\sigma_\alpha(a))) \subseteq D(a)$;
- $\forall g' \in \Gamma'$ if $\sigma_\gamma(g')$ is defined, then $\sigma_\alpha(D(\sigma_\gamma(g'))) \subseteq D'(g')$.

NOTATION. In the following, the indices α and γ are omitted. We denote the preimage of σ_γ by $\sigma^{\leftarrow}$. Also, if x is a term (or proposition) of θ, then $\sigma(x)$ is the term (resp. proposition) of θ' obtained by replacing each attribute a of x by $\sigma(a)$.

Notice that through the choice of an appropriate morphism, it is possible to state whether a given component and a given system are colocated (i.e., whenever one moves, so does the other) or if the component can move independently within the system. This can be modeled by a morphism that maps (or not) the location attribute of the component to the location attribute of the system.

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* $\mathcal{SIG}$.

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

- the initialization condition is not weakened;
- the assignments are equivalent;
- the guards are not weakened.

DEFINITION 4. *A* superposition morphism $\sigma : \langle\theta, \Delta\rangle \to \langle\theta', \Delta'\rangle$ *is a signature morphism* $\sigma : \theta \to \theta'$, *such that*:

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

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

The category of signatures extends to programs.

PROPOSITION 2. *Programs and superposition morphisms constitute a category* $\mathcal{PROG}$.

To give an example of a program morphism, consider the need to prevent carts from colliding at intersections. We achieve that goal in two steps, the second of which to be presented in Section 4.2. When two carts enter two segments that intersect, due to the semantics of COMMUNITY allowing only one cart to move at each step, one of the carts will be further away from the intersection. The first step to avoid collisions is to force that cart 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 a second (external) location attribute λ' is needed. The *Cart* program is thus transformed into an *InhibitedCart* as given by the diagram in Fig. 1, where the inhibition condition is I = CrossingSegments(λ', λ) $\wedge$ DistanceToCrossing(λ') < DistanceToCrossing(λ). Action *fast* may execute only when this condition is false. The morphism is an injection: $\lambda \mapsto \lambda$, fast $\mapsto$ fast, etc. The next section shows how the *InhibitedCart* program can be obtained by composition of two components.

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_1 and P_2, the signature S is constructed as follows: for each pair of

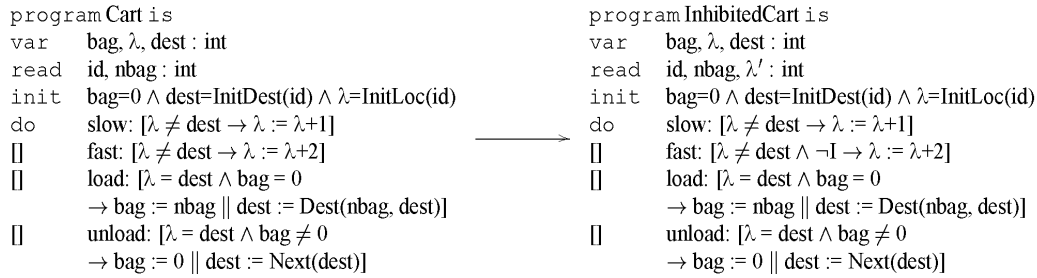


Fig. 1. A program morphism.

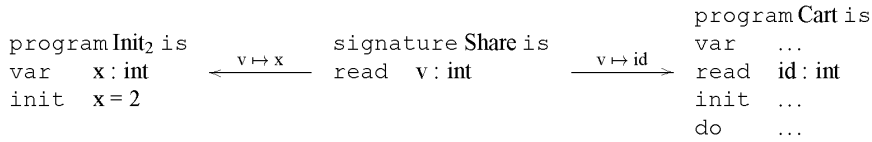


Fig. 2. Variable initialization.

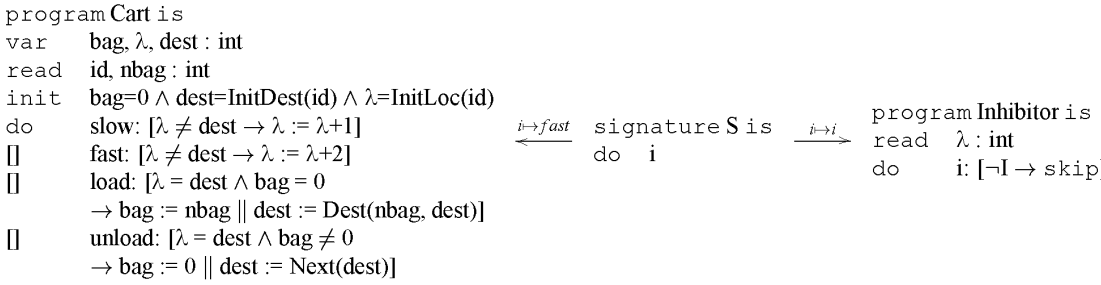


Fig. 3. Action synchronization.

attributes (or actions) $a_1 \in P_1$ and $a_2 \in P_2$ that are to be shared (resp. synchronized), the signature contains one attribute (resp. action) a; the morphism from S to P_i maps a to a_i. We have morphisms only between signatures or only between programs, but a signature $\theta = \langle V, R, \Gamma \rangle$ can be seen as a program $\mathcal{F}(\theta)$ with an "empty" body [9]. In categorical terms, the operator $\mathcal{F}$ is a functor (i.e., a map between objects and morphisms of different categories).

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

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 Cart_2 **is**
var bag, λ, dest, 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 $a := F(g_1, a)$ and the guard is strengthened by $F(g_1, a) = F(g_2, a)$. If the actions are "incompatible" (i.e., the terms denote different values for a) then the equality is false and therefore the synchronized action will never execute, as expected.

As an illustration, the pushout of the diagram in Fig. 3 is program *InhibitedCart* shown in the previous section: actions *fast* and *i* were paired together, joining their guards and assignments. Notice that attribute λ of the *Inhibitor* program has been renamed to λ′ because names are local.

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

PROPOSITION 3. *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 use connectors, a basic concept of Software Architecture [2]. 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., [9] proposes roles to be specifications written in temporal logic) but for simplicity we assume that roles and glues are members of the same category. We, therefore, adopt only the basic definitions of [9].

DEFINITION 5. A connection *is a tuple* $\langle C, G, R, \gamma, \rho \rangle$ *where*

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

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 [8].

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 6. *An* instantiation *of a connector* $\{\langle C_i, G, R_i, \gamma_i, \rho_i \rangle\}$ *with programs* P_i *is a set of morphisms* $\iota_i : R_i \to P_i$ in $\mathcal{PROG}$. *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 diagram

$$P_1 \overset{\iota_1}{\leftarrow} R_1 \overset{\rho_1}{\leftarrow} \mathcal{F}(C_1) \overset{\gamma_1}{\to} G \overset{\gamma_2}{\leftarrow} \mathcal{F}(C_2) \overset{\rho_2}{\to} R_2 \overset{\iota_2}{\to} P_2$$

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.

A distributed system 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 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 consider three kinds of interactions:

- *Inhibition*. An action may not execute.[1]
- *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 attrbutes 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.

DEFINITION 7. *Given a set* $\mathcal{P}$ *of programs, 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 $\langle g, M, P, g', P', I \rangle$;

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* $\mathcal{P}$.

The following sections present the connector patterns corresponding to the above interactions. The glue of a connector only needs to include the attributes that occur in the interaction condition. However, to make the formal definitions easier, the glue patterns will include *all* the attributes of all the roles. Due to the locality of names, attributes from different roles must be put together with the disjoint union operator (written $\uplus$) to avoid name clashes.

For further simplication, we assume that the interaction condition only uses attributes from the interacting programs P and P', and thus only those roles are presented in the patterns. If this is not the case, the instantiated connector must have further roles that provide the remaining attributes. The next section provides an example.

1. In this case the interaction is between the program and its environment.

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 8. *The* inhibition connector pattern *corresponding to inhibition interaction* $\langle g, P, I\rangle$ *is the diagram of* Fig. 4.

For illustration, the action inhibition example of Sections 2 and 3 can be achieved through the connector given in Fig. 5. Notice that the connector has two roles, one for the cart whose action is to be temporarily inhibited, the other for the cart that provides the context for the inhibition to occur.

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

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 [8]. 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. We will not handle coselection because we believe that the intuitive notion of synchronization corresponds to coexecution.

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

DEFINITION 9. *The* synchronization connector pattern *corresponding to synchronization interaction* $\langle g, P, g', P', I\rangle$ *is the diagram of* Fig. 6.

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 "nonexclusive coexecution" in [8]: 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 synchronization connector pattern, just keeping the synchronized action gg'.

Continuing with the example, the second step to avoid collisions at crossings is to force the nearest cart to move fast whenever the most distant one moves. Since the latter can only move slowly, the nearest cart is guaranteed to pass the crossing first. Using the same interaction condition as in Fig. 5 one gets the diagram in Fig. 7.

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 synchronization connector (Fig. 7). The diagram is:

$$\begin{array}{ccccccccc}
\text{Cart} & \leftarrow & \text{Context} & \rightarrow & \text{InhibitCrossing} & \leftarrow & \text{Target} & \rightarrow & \text{Cart} \\
\downarrow & & & & & & & & \downarrow \\
\text{Cart}_1 & & & & & & & & \text{Cart}_2 \\
\uparrow & & & & & & & & \uparrow \\
\text{Cart} & \leftarrow & C_1 & \rightarrow & \text{SynchCrossing} & \leftarrow & C_2 & \rightarrow & \text{Cart}
\end{array}$$

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

program System **is**
var $\text{bag}_i : \lambda_i, \text{dest}_i, \text{id}_i, : \text{int}$
read $\text{nbag}_i : \text{int}$
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: $[\lambda_1 \neq \text{dest}_1 \wedge \neg I \rightarrow \lambda_1 := \lambda_1 + 2]$
[] slow_2: $[\lambda_2 \neq \text{dest}_2 \wedge \neg I \rightarrow \lambda_2 := \lambda_2 + 1]$
[] $\text{fast}_1\ \text{slow}_2$: $[\lambda_1 \neq \text{dest}_1 \wedge \lambda_2 \neq \text{dest}_2 \wedge I$
$\rightarrow \lambda_1 := \lambda_1 + 2 \ ||\ \lambda_2 := \lambda_2 + 1]$
[] fast_2: $[\lambda_2 \neq \text{dest}_2 \wedge \neg I \rightarrow \lambda_2 := \lambda_2 + 2]$
[] load_i: $[\lambda_i = \text{dest}_i \wedge \text{bag}_i = 0$
$\rightarrow \text{bag}_i := \text{nbag}_i \ ||\ \text{dest}_i := \text{Dest}(\text{nbag}_i, \text{dest}_i)]$
[] unload_i: $[\lambda_1 = \text{dest}_i \wedge \text{bag}_i \neq 0$
$\rightarrow \text{bag}_i := 0 \ ||\ \text{dest}_i := \text{Next}(\text{dest}_i)]$

To see that synchronization is transitive, consider the example in Fig. 8 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 one of the actions can occur. This happens, indeed, because the pushout of two morphisms $\sigma(g) = \{g_1, \ldots, g_n\}$ and $\sigma'(g) = \{g'_1, \ldots, g'_m\}$ is basically given by the pairs $\{g_1g'_1, \ldots, g_1g'_m, \ldots, g_ng'_m\}$ with morphisms $\mu(g_i) = \{g_ig'_1, \ldots, g_ig'_m\}$ and $\mu'(g'_j) = \{g_1g'_j, \ldots, g_ng'_j\}$. Putting into words: if an action g "ramifies" into actions $\sigma(g) = \{g_1, \ldots,$

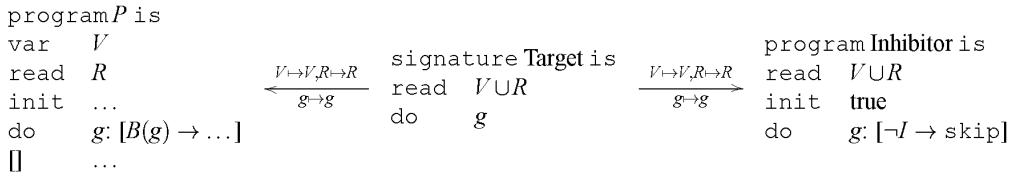


Fig. 4. Inhibition connector pattern.

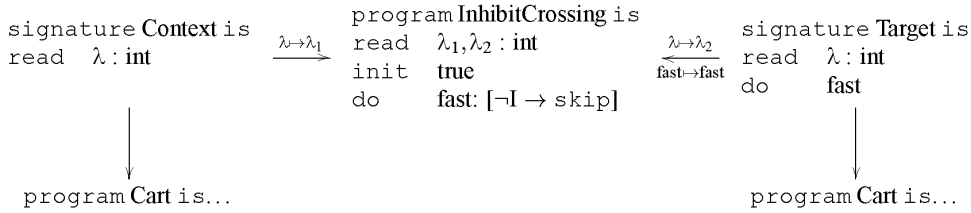


Fig. 5. Inhibition when I = CrossingSegments (λ_1, λ_2) $\wedge$ DistanceToCrossing (λ_1) < DistanceToCrossing (λ_2).

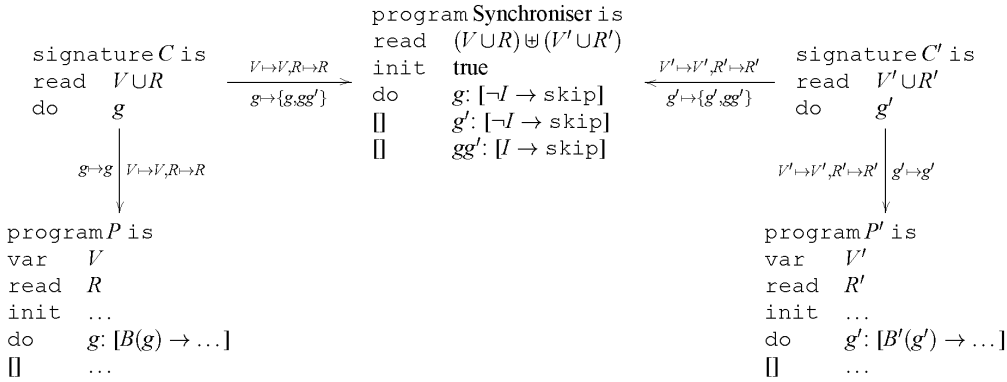


Fig. 6. Synchronization connector pattern.

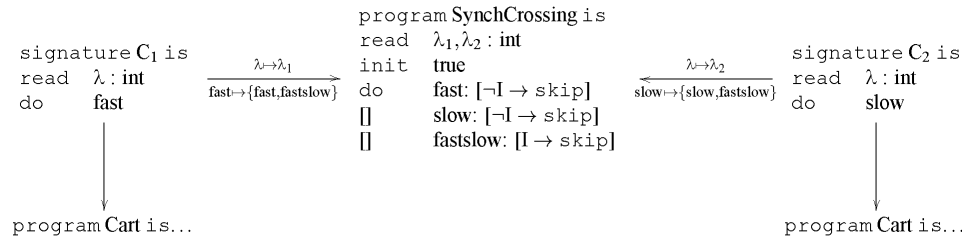


Fig. 7. Synchronizing fast and slow.

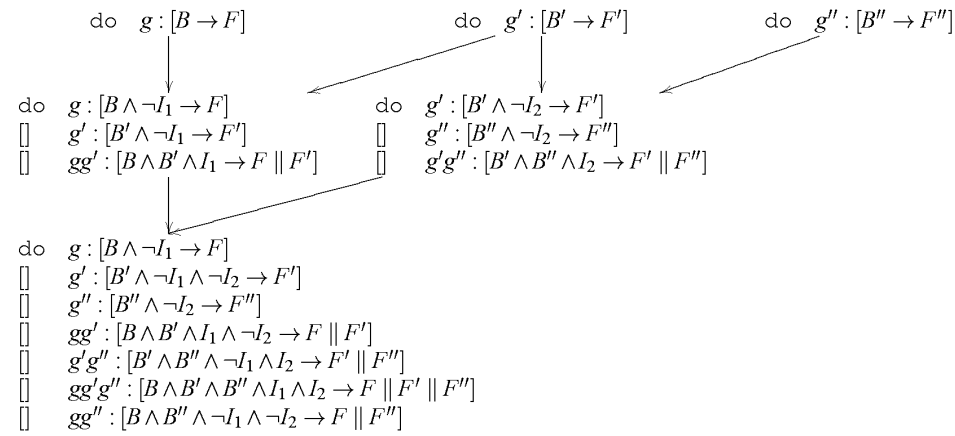


Fig. 8. Transitivity of synchronization.

$g_n\}$, it means that whenever g would be executed, any member 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 ramified in two distinct ways, in the pushout any combination of the subactions can occur whenever g executes. The pushout morphisms just state to which combinations each subaction belongs.

As one can see in Fig. 8, 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 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** $F_x \;||\; F_y$ 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 [8], 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 were to apply the usual construction, all those attributes would become a single one in the resulting system, which is clearly unintended.

Therefore, we 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.[2] 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:[3] 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 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 a given proposition I is true. Therefore the glue contains an action *wait* : $[I \rightarrow M' := M]$ 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 an 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 10. *The* communication connector pattern *corresponding to communication interactions* $\langle g_i, M_j, P_i, g'_j, M'_i, P'_j, I_{ij}\rangle$ (*for* $i = 1, \ldots, n$ *and* $j = 1, \ldots, m$) *is the diagram in* Fig. 9.

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 an attribute of type queue of integers. Although the locations of stations are fixed they must be represented explicitly in order to represent

2. It is always possible to write P' in such a way.

3. This is similar to pointed processes in the π-calculus, or to ports in distributed systems.

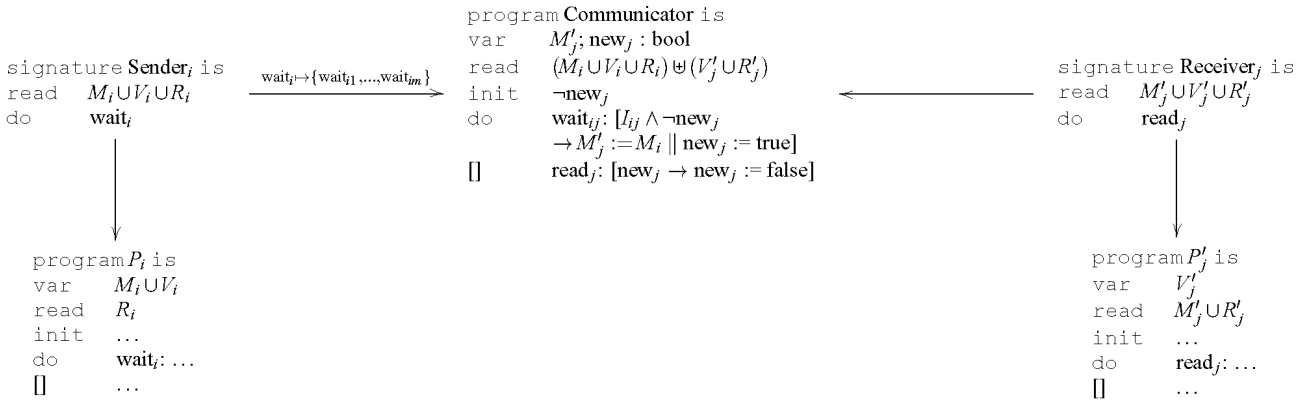


Fig. 9. Communication connector pattern.

the communication condition, namely that cart and station are colocated. 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. The loader program must have 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).

The *c* carts are connected to the *l* loaders through a connector with *c* identical roles (each one being the Cart program of Section 1) and *l* identical roles, each being the Loader program. In Fig. 10 we only show the roles and respective morphisms for the *i*th loader (sender) and the *j*th cart (receiver). Similarly, there is a connector with *u* roles for the unloaders and *c* roles for the carts. The *i*th cart (sender) is connected to the *j*th unloader (receiver) as shown in Fig. 11.

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 diagram in Fig. 12, 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 [7], [8], 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], [4].

As argued in [3], [13], the general benefits of working within a categorical framework are:

- mechanisms for interconnecting components into complex systems can be formalized through universal constructs (e.g., colimits);
- extralogical design principles are internalized through properties of universal constructs (e.g., the locality of names);
- different levels of design (e.g., signatures and programs) can be related through functors.

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.

One of the ideas we wish to explore is to remove the interaction condition from the glue's actions and instead associate it to the application of the whole connector. The diagram of the system architecture thus becomes dynamic, at each moment including only the connectors whose conditions are true. A preliminary description of this approach is described in [14]. Another possibility is to apply graph rewriting techniques to the system diagrams. A third venue is to change (again) the definition of morphism to represent the notion of "changes-to" instead of "component-of." In other words, a morphism from *P* to *P′* indicates that *P* may

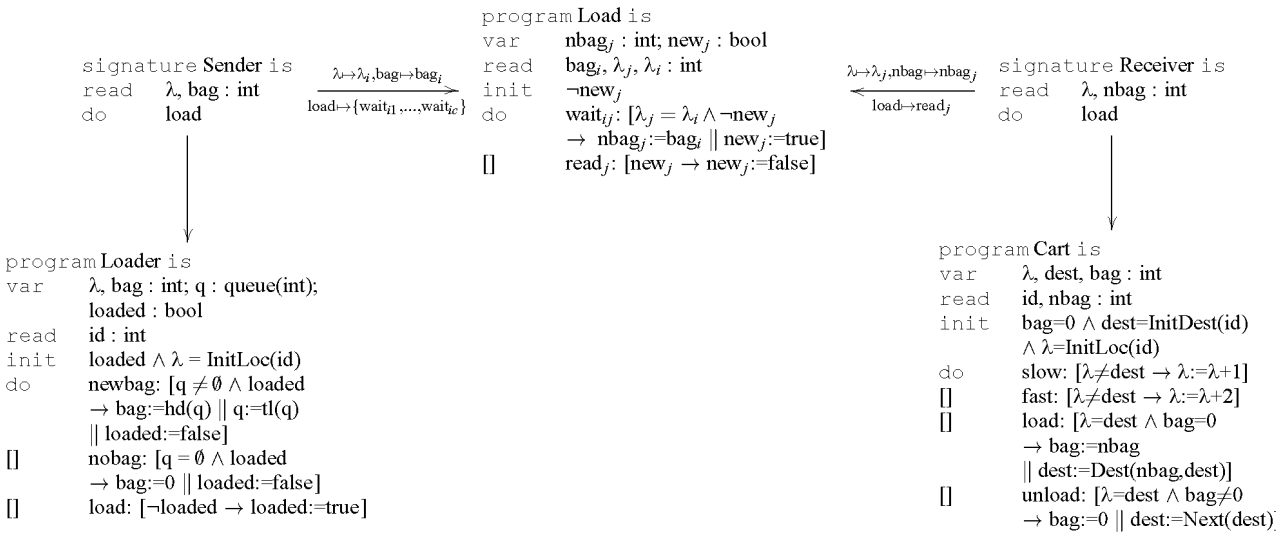


Fig. 10. Communication between loaders and carts.

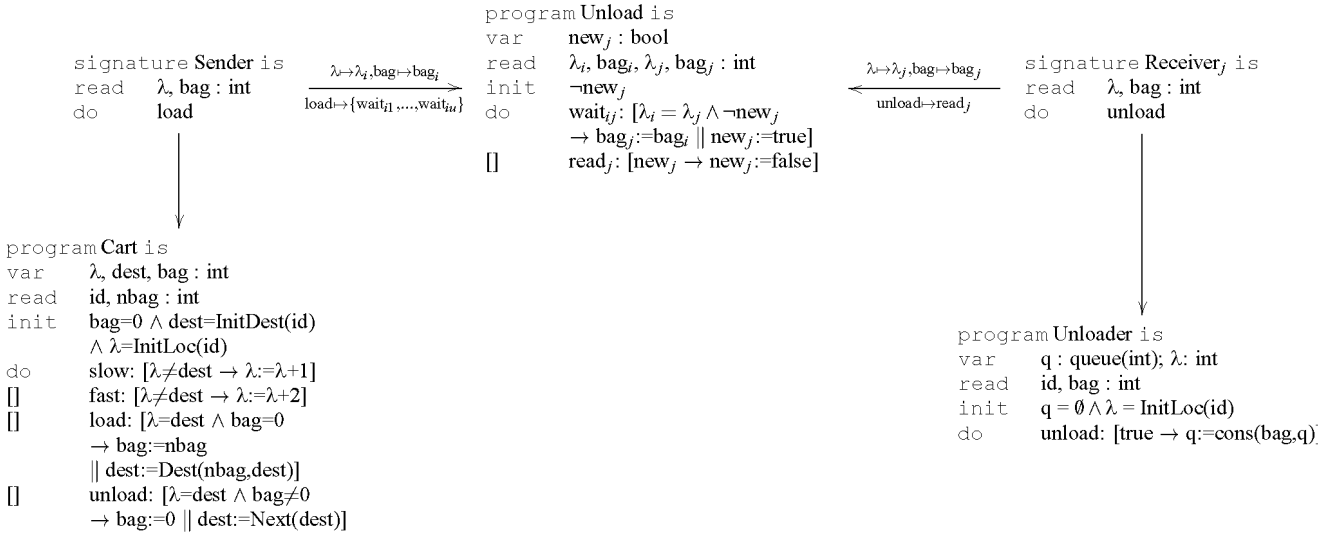


Fig. 11. Communication between carts and unloaders.

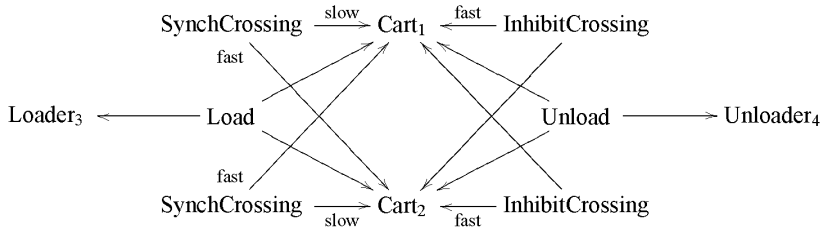


Fig. 12. The system architecture.

become *P′*. For the moment, these are just some of our ideas to capture Software Architecture evolution in a categorical setting. Their suitability and validity must be investigated.

ACKNOWLEDGMENTS

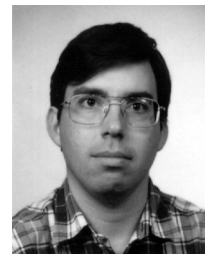
We would like to thank Antónia Lopes for many fruitful discussions and the anonymous referees for suggestions on how to improve the presentation. This work was partially supported by JNICT through contract PRAXIS XXI 2/2.1/MAT/46/94 (ESCOLA) and by project ARTS under contract to EQUITEL SA.

REFERENCES

[1] "Special Issue on Software Architecture," D. Garlan and D. Perry, eds., *IEEE Trans. Software Eng.*, vol. 21, no. 4, Apr. 1995.

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

[3] J.L. Fiadeiro and T. Maibaum, "Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality," *Proc. Third Symp. Foundations of Software Eng., SIGSOFT'95*, pp. 72–80, ACM Press, 1995.
[4] J.L. Fiadeiro and T. Maibaum, "Categorial Semantics of Parallel Program Design," *Science of Computer Programming*, vol. 28, pp. 111–138, 1997.
[5] K.M. Chandy and J. Misra, *Parallel Program Design—A Foundation*. Addison-Wesley, 1988.
[6] N. Francez and I. Forman, *Interacting Processes*. Addison-Wesley, 1996.
[7] G.–C. Roman, P.J. McCann, and Jerome Y. Plun, "Mobile UNITY: Reasoning and Specification in Mobile Computing," *ACM TOSEM*, vol. 6, no. 3, pp. 250–282, July 1997.
[8] P.J. McCann and G.–C. Roman, "Mobile UNITY: A Language and Logic for Concurrent Mobile Systems," Technical Report WUCS-97-01, Dept. of Computer Science, Washington Univ., St. Louis, Dec. 1996.
[9] J.L. Fiadeiro and A. Lopes, "Semantics of Architectural Connectors," *Proc. TAPSOFT'97*, pp. 505-519, *Lecture Notes in Computer Science, 1214*. Springer-Verlag, 1997.
[10] B.C. Peirce, *Basic Category Theory for Computer Scientists*. MIT Press, 1991.
[11] M. Wermelinger and J. Fiadeiro, "Connectors for Mobile Programs," Technical Report DI-FCUL TR-98-1, Dept. of Computer Science, Univ. of Lisbon, Portugal, Jan. 1998.
[12] U. Leonhardt and J. Magee, "Towards a General Location Service for Mobile Environments," *Proc. Third Int'l Workshop Service in Distributed and Networked Environments*, 1996.
[13] J.L. Fiadeiro and T. Maibaum, "A Mathematical Toolbox for the Software Architect," *Proc. Eighth Int'l Workshop on Software Specification and Design*, pp. 46–55. IEEE CS Press, 1996.
[14] M. Wermelinger and J.L. Fiadeiro, "Towards an Algebra of Architectural Connectors: A Case Study on Synchronization for Mobility," *Proc. Ninth Int'l Workshop Software Specification and Design*, pp. 135-142, IEEE CS Press, Apr. 1998.



Michel Wermelinger is a PhD candidate in the Computer Science Department at the New University of Lisbon, Portugal, working on dynamic reconfiguration of Software Architectures. He is a student member of the ACM and the IEEE and a member of the IEEE Computer Society.

José Luiz Fiadeiro is an associate professor in the Computer Science Department at the University of Lisbon, Portugal. His research interests include software specification formalisms and methods, especially as applied to component-based, reactive systems, and their integration in the wider area of general systems theory. His main contributions have been in the formalization of specification and program design techniques using modal logics (temporal and dynamic), and of their underlying modularization principles using Category Theory.