# A Graph Transformation Approach to Software Architecture Reconfiguration ⋆

Michel Wermelinger

*Departamento de Informática, Faculdade de Ciências e Tecnologia*
*Universidade Nova de Lisboa, 2829-516 Caparica, Portugal*
*E-mail: mw@di.fct.unl.pt*

José Luiz Fiadeiro

*Departamento de Informática, Faculdade de Ciências*
*Universidade de Lisboa, Campo Grande, 1700 Lisboa, Portugal*
*E-mail: jose@fiadeiro.org*

**Abstract**

The ability of reconfiguring software architectures in order to adapt them to new requirements or a changing environment has been of growing interest. We propose a uniform algebraic approach that improves on previous formal work in the area due to the following characteristics. First, components are written in a high-level program design language with the usual notion of state. Second, the approach deals with typical problems such as guaranteeing that new components are introduced in the correct state (possibly transferred from the old components they replace) and that the resulting architecture conforms to certain structural constraints. Third, reconfigurations and computations are explicitly related by keeping them separate. This is because the approach provides a semantics to a given architecture through the algebraic construction of an equivalent program, whose computations can be mirrored at the architectural level.

## 1 Introduction

One of the topics which is raising increased interest in the Software Architecture community is the ability to specify how an architecture evolves over

⋆ This work was partially supported by Fundação para a Ciência e Tecnologia through project POSI/32717/00 (FAST—Formal Approach to Software Architecture).

time, in particular at run-time, in order to adapt to new requirements or new environments, to failures, and to mobility [25,5,34,19]. This topic raises several issues [24], among which:

**modification time and source** Architectures may change before execution, or at run-time (called dynamic reconfiguration). Run-time changes may be triggered by the current state or topology of the system (called programmed reconfiguration [6]) or may be requested unexpectedly by the user (called ad-hoc reconfiguration [6]).

**modification operations** The four fundamental operations are addition and removal of components and connections. Although their names vary, those operators are provided by most reconfiguration languages (like [6,21,1]). In programmed reconfiguration, the changes to perform are given with the initial architecture, but they may be executed when the architecture has already changed. Therefore it is necessary to query at run-time the state of the components and the topology of the architecture.

**modification constraints** Often changes must preserve several kinds of properties: structural (e.g., the architecture has a ring structure), functional, and behavioural (e.g., real-time constraints).

**system state** The new system must be in a consistent state.

There is a growing body of work on architectural reconfiguration, some of it related to specific Architecture Description Languages (ADL), and some of a formal, ADL-independent nature. Most of the proposals exhibit one of the following drawbacks.

- Arbitrary reconfigurations are not possible: Darwin [17] only allows component replication; ACME [23] only allows optional components and connections; Wright [1] requires the number of distinct configurations to be known in advance.
- The languages used for the representation of computations are very simple and at a low level of abstraction; for instance, rewriting of labels [14], process calculi [22,2,1,18], term rewriting [29,10], graph rewriting [28]. They do not capture some of the abstractions used by programmers and often lead to cumbersome specifications.
- The combination of reconfiguration and computation, needed for run-time change, leads to additional formal constructs: [14] uses constraint solving, [22,1,2] define new semantics or language constructs for the process calculi, [10] must dynamically change the rewriting strategies, [28] imposes many constraints on the form of graph rewrite rules because they are used to express computation, communication, and reconfiguration. This often results in a proposal that is not very uniform, or has complex semantics, or does not make the relationship between reconfiguration and computation very clear.

To overcome some of these disadvantages, we propose to use a uniform algebraic framework based on Category Theory and a program design language with explicit state. The former allows us to represent both architectures and their reconfigurations, and to explicitly relate the computational with the architectural level in a direct and simple way. On the other hand, the language incorporates some of the usual programming constructs while keeping a simple syntax to be formally tractable.

To be more precise, architectures are graphs whose nodes are programs—written in CommUnity, a Unity-like language with the usual notion of state—and arcs denote superposition relationships. Reconfiguration is specified through conditional graph rewriting rules that depend on the state of the involved components. Rules are based on the double-pushout approach to graph transformation and are defined in a way which guarantees that:

- components are removed in a quiescent state [15] (i.e., when not interacting with other components);
- new components are introduced in a correctly initialised state;
- the resulting architecture conforms to certain structural constraints specified by a fixed graph constraining the possible interconnections between components.

The rules also allow to transfer state between old components and their replacements. Moreover, the categorical underpinnings provide a semantics for configurations in terms of a construction that returns a program equivalent to the given architecture. Computations are performed on the components of the architecture in a way that is consistent with the semantics.

We assume the reader is familiar with basic notions of Category Theory and with the double-pushout approach to graph transformation. The appendix contains a brief review of typed graphs and introduces the notation used.

The running example is inspired by the airport luggage distribution system used to illustrate Mobile Unity [27]. One or more carts move continuously in the same direction on a $U$ units long circular track. A cart advances one unit at each step. Along the track there are stations. There is at most one station per unit. Each station corresponds to a check-in counter or to a gate. Carts take bags from check-in stations to gate stations. All bags from a given check-in go to the same gate. A cart transports at most one bag at a time. When it is empty, the cart picks a bag up from the nearest check-in.

Carts must not bump into each other, e.g., if a cart is moving and the cart in front of it is stopped at a station loading or unloading a bag. This is avoided by changing the movement interactions between carts, depending on their location. We also consider that management decides to equip each cart with two counters to compute how many bags are processed on average for each

completed lap. The rationale is to check how efficient is the track layout, i.e., the distribution of the stations along the circuit.

## 2 CommUnity

CommUnity [8] is a parallel program design language initially developed to show how programs fit into Goguen's categorical approach to General Systems Theory. It is an action based version of Unity [3], but it also draws elements from IP [11]. Since then, the language and its framework have been extended to provide a formal platform for architectural design of open, reactive, reconfigurable systems [7,32,31,16].

We assume a fixed algebraic data type specification. In this paper we use sorts `int` (integers) and `bool` (booleans) with the usual constants and operations, including a function *if* : `bool`×`int`×`int` → `int` with the obvious meaning. We also need `list(int)` for lists of integers. A value for a list is written $[l_1, l_2, \ldots]$ and thus [] is the empty list. The operations 'head' and 'tail' perform as usual, and '+' represents list concatenation.

Formal definitions and proofs of the results in this and the next section can be found in [30,16].

### *2.1 Programs*

The syntax of a CommUnity program is

$$
\begin{array}{ll}
\texttt{prog}\ P & \\
\texttt{in} & in(V) \\
\texttt{out} & out(V) \\
\texttt{prv} & prv(V) \\
\texttt{init} & I \\
\texttt{do} & \underset{a \in sh(A)}{[]}\ a : G(a) \rightarrow \underset{l \in D(a)}{\|}\ l := E(a,l) \\
[] & \underset{a \in prv(A)}{[]}\ \texttt{prv}\ a : G(a) \rightarrow \underset{l \in D(a)}{\|}\ l := E(a,l)
\end{array}
$$

where

- $in(V)$ is the set of *input* variables. They are imported from the environment of the program, i.e., they are to be connected with output variables of other components in the environment. Their values can be read but not modified by the program.

- $out(V)$ and $prv(V)$ are the sets of *output* and *private* variables, respectively. They are called *local* to the program, because the environment cannot modify them. Output variables are accessible to the environment (can be read) but private variables are not. We define $loc(V) = out(V) \cup prv(V)$.
- $I$ is a proposition over the local variables, defining their admissible values in the initial state, i.e., the state in which the component is added to the system.
- $prv(A)$ is the set of *private actions*. Their execution is uniquely under the control of the program and, thus, it is the program that determines when a private action is performed.
- $sh(A)$ is the set of *shared actions*. Their execution is also under the control of the environment, i.e., their execution may require synchronisation with actions of other components. In a sense, shared actions provide interaction points as in IP.
- $G(a)$, a boolean expression over the variables, is the *guard* of $a$, i.e., when $G(a)$ is false, $a$ cannot be executed. Normally, we omit the guard when it is 'true'.
- $D(a)$ is the *domain* of $a$, defined as the set of local variables that action $a$ can change—its write frame.
- For every local variable $l$ in $D(a)$, $l$:=$E(a,l)$ is an assignment, with $E(a, l)$ an expression of the same sort as $l$. When an action has empty domain we use `skip` to denote the absence of assignments.

The behaviour of a closed program, i.e., a program with no input variables, is as follows. The program starts its execution in some state that satisfies the initial condition. At each step, one of the actions whose guard is true is selected and its assignments are executed simultaneously. Furthermore, private actions that are infinitely often enabled are selected infinitely often. The behaviour of an open program can only be given in the context of a configuration in which its input variables have been connected to output variables of other components. We address this issue in Section 2.3.

We now present programs to be used in the remaining of the paper. The program that controls a cart is

| | |
|---|---|
| `prog` | Cart |
| `in` | idest, ibag : int |
| `out` | obag : int |
| `prv` | loc, dest : int |
| `init` | $0 \leq \text{loc} < U \wedge \text{-1} \leq \text{dest} < U \wedge (\text{dest} = \text{-1} \Leftrightarrow \text{obag} = 0)$ |
| `do` | move: $\text{loc} \neq \text{dest} \rightarrow \text{loc} := \text{loc} +_U 1$ |
| [] | get: $\text{dest} = \text{-1} \rightarrow \text{obag} := \text{ibag} \parallel \text{dest} := \text{idest}$ |
| [] | put: $\text{loc} = \text{dest} \rightarrow \text{obag} := 0 \parallel \text{dest} := \text{-1}$ |

where $+_U$ is addition modulo $U$.

Locations are represented by integers from zero to the track length minus one. Bags are represented by integers, the absence of a bag being denoted by zero. Whenever the cart is empty, its destination is an impossible location, so that the cart keeps moving until it gets a bag and a valid gate location through action 'get'. When it reaches its destination, the cart unloads the bag through action 'put'. Notice that since input variables may be changed arbitrarily by the environment, the cart must copy their values to output variables to make sure the correct bag is unloaded at the correct gate.

To be able to compute how many bags are processed per lap on average, we add two counters.We memorise the current position so that we know when a lap has been completed. The bag counter is incremented when a bag is fetched from the check-in.

```
prog Cart_Stat
in      idest, ibag : int
out     obag : int
prv     loc, dest, sloc, laps, bags : int
 init   0 ≤ loc < U ∧ -1 ≤ dest < U ∧ (dest = -1 ⇔ obag = 0)
        ∧ sloc = loc ∧ laps = 0
do      move: loc ≠ dest ∧ loc +U 1 ≠ sloc →  loc := loc +U 1
[]      lap: loc ≠ dest ∧ loc +U 1 = sloc →  loc := loc +U 1 ‖ laps := laps + 1
[]      get: dest = -1 →  obag := ibag ‖ dest := idest ‖ bags := bags + 1
[]      put: loc = dest → obag := 0 ‖ dest := -1
```

A check-in counter starts with a non-empty queue of bags, and loads one by one onto passing carts.

```
prog Check_In
out     bag, dest : int
prv     loc : int; next : bool; q : list(int)
init    0 ≤ dest < U ∧ 0 ≤ loc < U ∧ q ≠ [] ∧ next
do      prv new: q ≠ [] ∧ next → bag := head(q) ‖ q := tail(q) ‖ next := false
[]      put: ¬next → next := true
```

Variable 'next' is used to impose sequentiality among the actions. To build a system for our example, the 'put' action must be synchronised with a cart's 'get' action and variables 'bag' and 'dest' must be shared with 'ibag' and 'idest', respectively.

A gate starts with an empty queue of bags and adds each new bag to the front.

```
prog Gate
in      bag : int
prv     loc : int; q : list(int)
init    0 ≤ loc < U ∧ q = []
do      get:  q := [bag] + q
```

In an architecture for our example, action 'get' must be synchronised with a cart's 'put' action, and variable 'bag' must be shared with 'obag'.

To take program state into account, we introduce a fixed set $LV$ of typed variables, called *logical variables*. A *program instance* is then defined as a pair $\langle P, \epsilon \rangle$ with $P$ a program and $\epsilon : loc(V) \to Terms(LV)$ assigns to each local variable $l$ of $P$ a term—built from the logical variables and the functions of the data type signature—of the same sort as $l$. No valuation is assigned to input variables because those are not under control of the program. Notice also that the valuation may return an arbitrary term, not just a ground term. Although in the running system the value of each program variable is given by a ground term, we need logical variables to be able to write reconfiguration rules whose left-hand sides match components with possibly infinite distinct combinations of values for their variables (see Section 4).

For the rest of the paper, $LV = \{l_n, b_n, d_n, i_n : \text{int}; r_n : \text{bool}; q_n : \text{list(int)} \mid n = 0, 1, 2\}$. We write $x_0$ simply as $x$. We represent program instances in tabular form (see next section). If $P$ has no local variables, $\epsilon$ is empty and we write simply $\boxed{P}$.

### *2.2 Superposition*

A morphism $\sigma : P \to P'$ from a program $P$ to a program $P'$ states that $P$ is a component of the system $P'$ and, as shown in [8], captures the notion of program superposition [3,11]. Mathematically speaking, the morphism is defined as follows.

Each variable $v$ of $P$ is mapped to a variable $\sigma(v)$ of $P'$ of the same sort as $v$. Moreover, output (resp. private) variables of $P$ are mapped to output (resp. private) variables of $P'$. However, if $v$ is an input variable, $\sigma(v)$ may be either an input or output variable. The latter case accounts for $v$ being shared with an output variable of another component of $P'$. Because colimits compute a composition that is "minimal", it does not internalise variables (i.e., they do not become private). If required, this must be done explicitly through a hiding operation, for which a categorical semantics can also be given [16].

Each action name $a$ of $P$ is mapped to a (possibly empty) set of action names

$\sigma(a) = \{a'_i \mid i = 1, \ldots, n\}$ of $P'$. Those actions correspond to the different possible behaviours of $a$ within the system $P'$. If $a$ is shared (resp. private), so is each $a'_i$. Moreover, if $a$ and $b$ are distinct actions of $P$, then $\sigma(a)$ and $\sigma(b)$ are disjoint, i.e., internal synchronisation is not allowed. Each action $a'_i$ must preserve the functionality of $a$, possibly adding more things specific to other components of $P'$. In particular, the guard of $a'_i$ must not be weaker than the guard of $a$, and the assignments of $a$ must be contained in $a'_i$, up to the variable renaming introduced by the morphism.

Finally, each action $a'$ of $P'$ that modifies a given variable $\sigma(l)$ must be in the image set of some action of $P$ that changes $l$. In other words, the new actions of $P'$—i.e., those unrelated to the actions of $P$— are not allowed to change the local variables of $P$. This enforces that a program's local variables are only under its control, even if the program is combined with other programs into a larger one. It corresponds to the requirement in UNITY that new actions may only modify the superposed variables, they cannot contain assignments to the underlying variables.

We define *refinement morphisms* as a subset of superposition morphisms, namely those that do not alter the border between a program and its environment. More precisely, for $\sigma$ to be a refinement morphism, it must map input variables of $P$ into input variables of $P'$, and it must be injective over the non-private variables of $P$, i.e., $P'$ may not collapse the externally visible variables of $P$. Moreover, if $a$ is a shared action of $P$, then $\sigma(a)$ must be non-empty, i.e., shared actions of $P$ must be implemented by $P'$. Finally, for $P'$ to *refine* $P$, there must be a refinement morphism $\sigma : P \to P'$ and the initialization condition of $P'$ must imply that of $P$ (after renaming the variables according to $\sigma$).

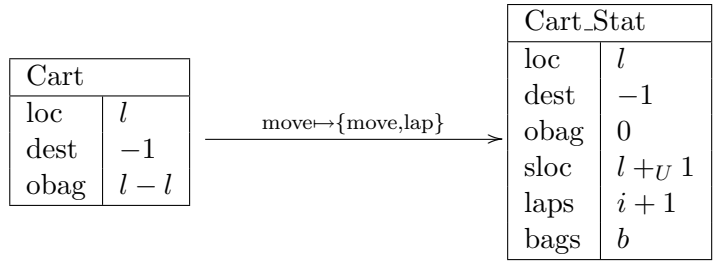
It is obvious that 'Cart_Stat' refines 'Cart', i.e., that morphism

$$\text{Cart} \xrightarrow[\text{move}\mapsto\{\text{move,lap}\},\ \text{get}\mapsto\text{get},\ \text{put}\mapsto\text{put}]{\substack{\text{idest}\mapsto\text{idest},\ \text{ibag}\mapsto\text{ibag}\\ \text{obag}\mapsto\text{obag},\ \text{loc}\mapsto\text{loc},\ \text{dest}\mapsto\text{dest}}} \text{Cart\_Stat}$$

obeys the above conditions and that 'Cart_Stat' strengthens the initialisation condition of 'Cart'. Notice how action 'move' is divided in two sub-cases, each strenghening the guard and adding more assignments. Henceforth, when presenting superposition morphisms we omit the identity mappings.

A morphism $\sigma : \langle P, \epsilon \rangle \to \langle P', \epsilon' \rangle$ between program instances is simply a superposition morphism $\sigma : P \to P'$ that preserves state. To be more precise, the algebraic data type axioms must entail $\epsilon(l) = \epsilon'(\sigma(l))$ for any local variable $l$ of $P$ and any substitution of the logical variables.

The previous refinement may be extended to the following morphism, where the instance on the right represents a cart that has completed at least one lap and will complete another one with the next move:



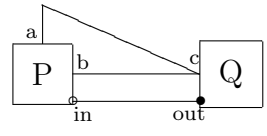

Programs and their morphisms constitute a category $\mathfrak{Prog}$, and program instances and their morphisms form a category $\mathfrak{Inst}$. Moreover, there is a forgetful functor $\mathcal{IP} : \mathfrak{Inst} \to \mathfrak{Prog}$. Given a diagram $D$ in $\mathfrak{Inst}$, we write $Vars(D)$ for the set of logical variables that appear in the program instances of $D$.

### *2.3 Configurations*

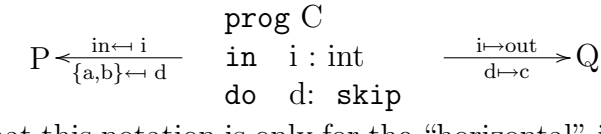
Interactions between programs are established through action synchronisation and memory sharing. This is achieved by relating the relevant action and variable names of the interacting programs.

In Category Theory, all relationships between objects must be made explicit through morphisms. In the particular case of CommUnity programs, it means that names are not global. To state that variable (or action) $a_1$ of program $P_1$ is the same as variable (resp. action) $a_2$ of $P_2$ one needs a third, "mediating" program $C$—the *channel*—containing just a variable (resp. action) $a$ and two morphisms $\sigma_i : C \to P_i$ that map $a$ to $a_i$. A channel is a degenerate program that provides the basic interaction mechanisms (synchronisation and memory sharing) between two given programs and adds no computations of its own. Thus a channel has only input variables and shared actions with true guards and no assignments.

To make examples clearer and more compact, we indicate the non-private variables and actions of a program around its name, and connect directly the shared variables and the synchronised actions. Inspired by the Darwin notation [17], we use black circles for output variables and white circles for input variables. Actions have no special notation. For example,




is an abbreviation for




We point out that this notation is only for the "horizontal" interconnection of

non-private variables and actions of programs. For the "vertical" relationships (i.e., general superposition involving also the private names) we continue to use arrows labelled with the mappings.

Problems arise if two synchronised actions update a shared variable in distinct ways. As actions only change the values of local variables, it is sufficient to impose that output variables are not shared, neither directly through a single channel nor indirectly through a sequence of channels. We call such diagrams *configurations*. This restriction forces interactions between programs to be synchronous communication of values (from output to input variables), a very general mode of interaction that is suitable for the modular development of reusable components, as needed for architectural design.

It can be proved that every finite configuration has a colimit, which returns the minimal program that simulates the execution of the overall system. Briefly put, the colimit is obtained by taking the disjoint union of the variables (modulo shared variables) and the cartesian product of actions (modulo synchronized ones) to denote parallel execution of non-synchronised actions. Actions are synchronized by taking the conjunction of the guards and the parallel composition of assignments. An example is provided in the next section.

A *configuration instance* is a diagram $D$ in $\mathfrak{Inst}$ such that $\mathcal{IP}(D)$ is a configuration. Since output variables are not shared, they have no conflicting valuations. Therefore every configuration instance has a colimit, given by the colimit of the underlying configuration together with the union of the valuations of the program instances.

## 3 Architectures

### *3.1 Connectors*

Software Architecture has put forward the notion of connector to encapsulate the interactions between components. An $n$-ary connector consists of $n$ roles $R_i$ and one glue $G$ stating the interaction between the roles. These act as "formal parameters", restricting which components may be linked together through the connector. We represent a connector (instance) by a configuration (instance) of the form

$$\begin{array}{ccc} & \overset{\gamma_1}{\swarrow} & C_1 \xrightarrow{\rho_1} R_1 \\ G & \vdots & \vdots \\ & \underset{\gamma_n}{\nwarrow} & C_n \xrightarrow{\rho_n} R_n \end{array}$$

where each channel $C_i$ indicates which variables and actions of role $R_i$ are used in the interaction specification, i.e., the glue. An $n$-ary connector can be

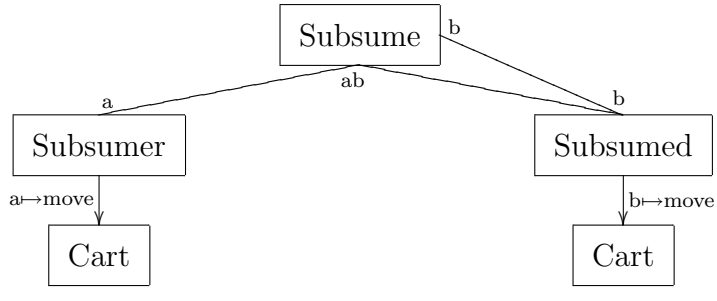
applied to components $P_1, \ldots, P_n$ when $P_i$ refines $R_i$, for each $i = 1, \ldots, n$. This corresponds to the intuition that the "actual arguments" (i.e., the components) must instantiate the "formal parameters" (i.e., the roles).

An *architecture* is then a configuration where all programs interact through connectors, and all roles are instantiated, i.e., there are no "dangling" roles. An *architecture instance* is the obvious extension to $\mathfrak{Inst}$. Therefore any architecture (instance) has a semantics given by its colimit.

We now present the connectors necessary for the remaining of this paper.

### *3.1.1 Subsumption*

The logical analogy to synchronisation is equivalence. However, to avoid a cart $c_1$ colliding with the cart $c_2$ right in front of it we only need implication: if $c_1$ moves, so must $c_2$, but the opposite is not necessary. The analogy with implication also extends to the counter-positive: if $c_2$ cannot move, e.g., because it is (un)loading a bag, then neither can $c_1$. We call this "one-way" synchronisation *action subsumption*. For our example, the movement of $c_1$ subsumes the movement of $c_2$. The connector is only possible because our morphisms allow an action to ramify into a set of actions. In this case, the movement action of $c_2$ ramifies in two: one for the case in which it must co-occur with the movement of $c_1$, the other when it can occur freely. The generic action subsumption connector and its application to two carts is




with
```
prog Subsumer        prog Subsume        prog Subsumed .
do  a: skip          do  ab: skip        do  b: skip
                     []  b: skip
```

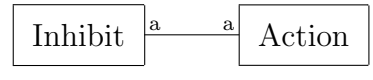
Although the two roles are isomorphic, the binary connector is not symmetric because the connections treat the two actions differently: 'b' may be executed alone at any time, while 'a' must co-occur with 'b'. Hence, action 'a' is the one that we connect to the 'move' action of $c_1$, while action 'b' is associated to the movement of $c_2$. The colimit object of the above configuration is given in Figure 1. Notice how it contains all possible combinations of the non-synchronised actions 'get' and 'put' of each cart.

```
prog CollisionFreeCarts
in   idest1, ibag1, idest2, ibag2 : int
out  obag1, obag2 : int
prv  loc1, loc2, dest1, dest2 : int
do   move1move2: loc1 ≠ dest1 ∧ loc2 ≠ dest2
     → loc1 := loc1 +U 1 ‖ loc2 := loc2 +U 1
[]   move2: loc2 ≠ dest2 → loc2 := loc2 +U 1
[]   get1: dest1 = -1 → obag1 := ibag1 ‖ dest1 := idest1
[]   put1: loc1 = dest1 → obag1 := 0 ‖ dest1 := -1
[]   get2: dest2 = -1 → obag2 := ibag2 ‖ dest2 := idest2
[]   put2: loc2 = dest2 → obag2 := 0 ‖ dest2 := -1
[]   get1get2: dest1 = -1 ∧ dest2 = -1
     → obag1 := ibag1 ‖ dest1 := idest1 ‖ obag2 := ibag2 ‖ dest2 := idest2
[]   get1put2: dest1 = -1 ∧ loc2 = dest2
     → obag1 := ibag1 ‖ dest1 := idest1 ‖ obag2 := 0 ‖ dest2 := -1
[]   put1get2: loc1 = dest1 ∧ dest2 = -1
     → obag1 := 0 ‖ dest1 := -1 ‖ obag2 := ibag2 ‖ dest2 := idest2
[]   put1put2: loc1 = dest1 ∧ loc2 = dest2
     → obag1 := 0 ‖ dest1 := -1 ‖ obag2 := 0 ‖ dest2 := -1
```

Fig. 1. The program resulting from applying Subsume to two carts.

### *3.1.2 Inhibition*

To inhibit an action we must let its guard become false. Due to the semantics of colimit, this can be done without changing the guard directly. It suffices to synchronise the action with one that has a false guard, obtaining the unary connector



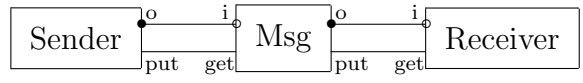

with glue `prog` Inhibit `do` a: false → `skip` and role `prog` Action `do` a: `skip` .

### *3.1.3 Asynchronous Communication*

We assume a sender wants to transmit a message $M$, which is a set of output variables. If a receiver wants to get the message, it must provide input variables $M'$ which correspond in number and sort to those of $M$. The sender produces the values, stores them in $M$, and waits for an acknowledge to produce new values for $M$. For that purpose, we assume the sender has an action 'put' which must be executed before the new message is produced. Similarly, the receiver must be informed when a new message has arrived, so that it may start processing it. For that purpose we assume that a receiver has a single

action ‘get’ which is the first action to be executed upon the receipt of a new message.

The connector explicitly models message transmission as the parallel assignment of the message variables. For this to be possible, the output variables $M$ of the sender must be input variables of the glue, and the input variables $M'$ of the receiver must be output variables of the glue. The glue’s actions are also symmetrical to those of the sender and receiver: there is a ‘get’ action to be synchronised with the action ‘put’ of the sender, thus performing the transmission *and* the notification of the sender, and there is a ‘put’ action to be synchronised with the ‘get’ action of the receiver. This decouples the sender’s action from the receiver’s, thus imposing the asynchronicity. The message passing connector presented next only transmits a single variable of sort $t$. It can be trivially generalised to messages as a set of variables.




`prog` Msg
`in` i : $t$
`out` o : $t$
`prv` ready : bool
`init` ready
`do` get: ready → o := i ‖ ready := false
[] put: ¬ready → ready := true

`prog` Sender
`out` o : $t$
`do` put: `skip`

`prog` Receiver
`in` i : $t$
`do` get: `skip`

### *3.2 Initial Architectures*

An important architecture instance for system specification is the one that provides the initial values for the variables. For that purpose, each local variable is associated to a ground term such that all initialisation conditions are satisfied. An example initial architecture, using instances of the previous connectors, is given in Figure 2. Notice that the ‘put’ action of the check-in counter is inhibited because there is no cart yet at that location to load the bag. Similarly for the carts’ and gate’s ‘get’ actions.

### *3.3 Architectural Styles*

In general, a role may be instantiated by different components, and it may be even the case that the same component can instantiate the same role in different ways (e.g., there are three morphisms from ‘Action’ to ‘Cart’). But normally only a few of all the possibilities are meaningful to the application at

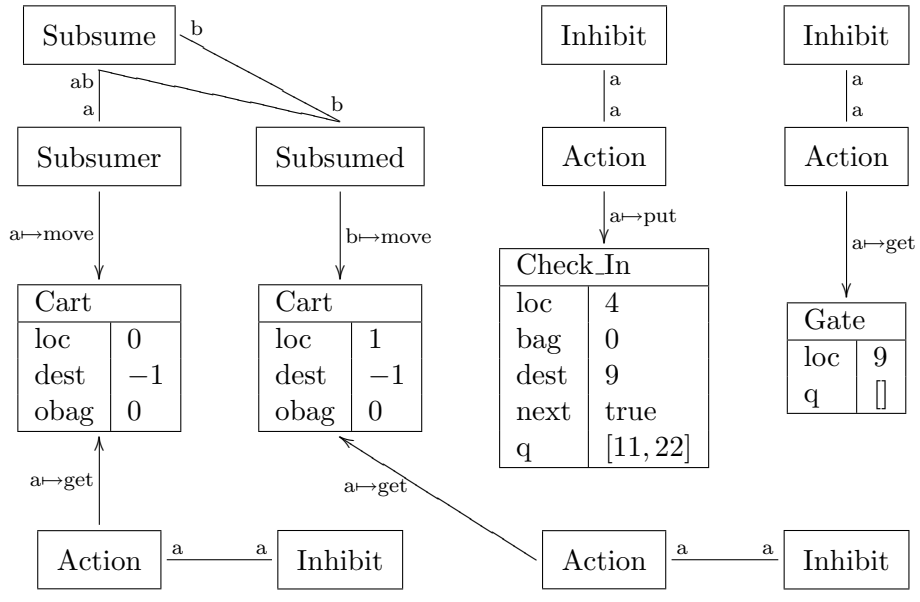



Fig. 2. An initial architecture with two carts and two stations.

hand. The allowed ways to apply connectors to components can be described by typed graphs. This leads to a declarative notion of *architectural style*: it consists of a set of components, a set of connectors, and a diagram $AS$ in $\mathfrak{Prog}$ using only those connectors and components. It is important to notice that $AS$ is not necessarily a configuration: since it shows in a single diagram all morphisms that may occur in architectures, it may happen that output variables are shared in $AS$.

Every architecture instance written by the user must then come equipped with a morphism to $AS$ proving that it obeys the restrictions imposed by $AS$. Additionally, the typing must be meaningful. For example, in our case, a cart in the architecture instance cannot be typed by a gate in the style.

**Definition 1** *An architecture instance $D$* conforming *to a style $AS$, also called* $AS$-architecture instance*, is a pair $\langle D, t_D \rangle$ with $t_D$ such that the following diagram in $\mathfrak{Graph}$ commutes.* [1]

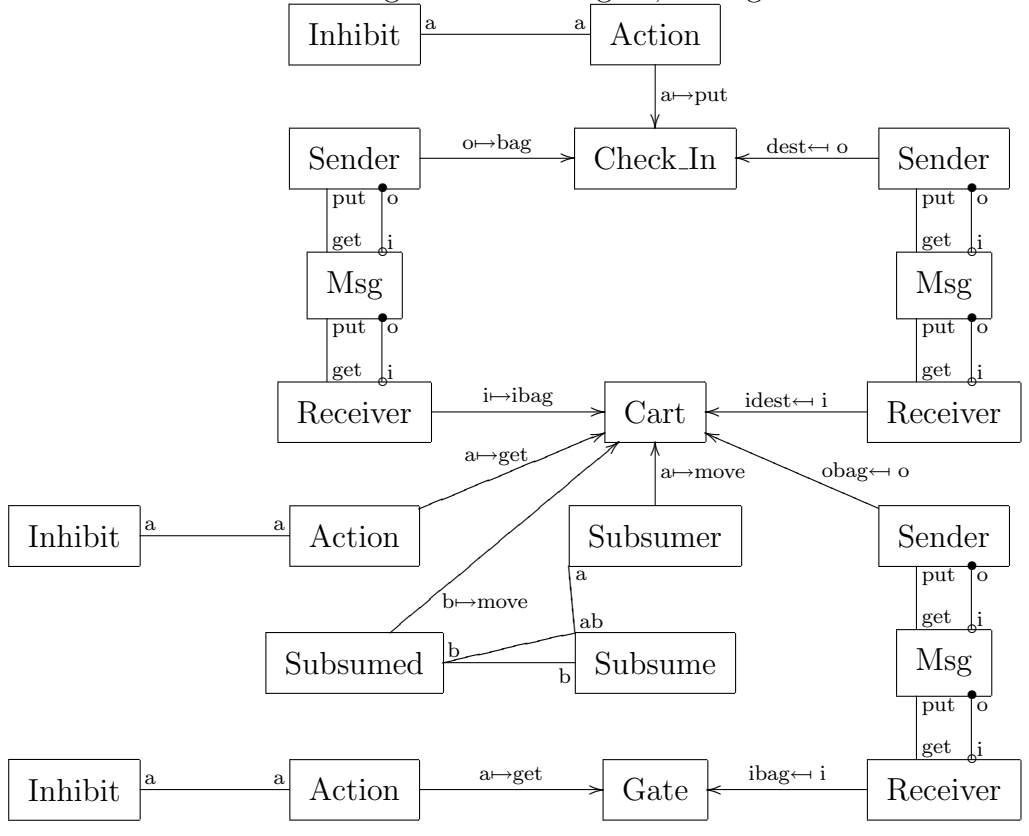
$$\begin{array}{ccc} G_{\mathfrak{Prog}} & \xleftarrow{\mathcal{IP}} & G_{\mathfrak{Inst}} \\ {\scriptstyle \delta_{AS}}\uparrow & & \uparrow{\scriptstyle \delta_D} \\ \Delta_{AS} & \xleftarrow{t_D} & \Delta_D \end{array}$$

Another approach (introduced by [22] and adopted by [14,29]) is to view styles as graph grammars that generate all graphs (i.e., architectures) belonging to the style. In this case it is necessary to prove explicitly that the reconfiguration rules do not generate graphs that do not belong to the style, while in the typing

[1] See the end of the appendix for the meaning of the notation.

approach this is automatically enforced (Proposition 6 in the next section). On the other hand, the graph grammar approach to style is more expressive than the typed graph approach: for instance, it allows to state constraints on the number of components or abstract architectural patterns like pipe-filter and layer. Since a graph grammar is just a set of graph productions and a given start graph, that approach can be straightforwardly used within our framework, where graph productions are substituted by reconfiguration rules (described in the next section). However, we believe that typed graphs are sufficient, simpler, and more straightforward in many occasions, namely when only the kinds of interactions between the components have to be restrained.

The style for our example states, for instance, that the action subsumption connector is only to be used for carts' movement; it prevents the 'put' action of a counter to subsume the 'get' action of a gate, among other combinations.




## 4 Reconfiguration

Basically, dynamic reconfiguration is a rewriting process over architecture instances, i.e., graphs typed over the objects and morphisms of $\mathfrak{Inst}$. This ensures that reconfiguration and computation are kept separate because, due to the preservation of the typing enforced by typed graph morphisms, the state of components and connectors that are not deleted nor added by a rule does

not change.

Dynamic reconfiguration rules depend on the current state. Thus they must be conditional rewrite rules. Within the algebraic graph transformation framework it is possible to define conditional graph productions in a uniform way, using only graphs and graph morphisms [13]. However, for our representation of components it is simpler, both from the practical and formal point of view, to represent conditions as boolean expressions over the logical variables appearing on the left-hand side instances.

As for components introduced by the rule, we provide full control to the rule writer, letting him specify exactly in which state new components are added to the architecture. For that purpose we require that the logical variables occurring on the right-hand side of a rule also occur on the left-hand side.

**Definition 2** *A* dynamic reconfiguration rule $\langle p, mc \rangle$ *is a graph production* $p$ *typed over* $G_{\mathfrak{Inst}}$ *where* $L$*,* $K$*, and* $R$ *are architecture instances,* $Vars(R) \subseteq Vars(L)$*, and the* matching condition $mc$ *is a proposition over* $Vars(L)$*.*

If there is an architectural style, then the three instances in a reconfiguration rule must conform to the style, and the morphisms between them must also preserve the typing given by the style.

**Definition 3** *An AS-*dynamic reconfiguration rule *for a style AS is a pair* $\langle p_{AS} : (\langle L, t_L \rangle \xleftarrow{l} \langle K, t_K \rangle \xrightarrow{r} \langle R, t_R \rangle), mc \rangle$ *where*

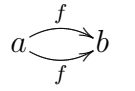
- $\langle L, t_L \rangle, \langle K, t_K \rangle, \langle R, t_R \rangle$ *are AS-architecture instances,*
- $\langle p : (L \xleftarrow{l} K \xrightarrow{r} R), mc \rangle$ *is a dynamic reconfiguration rule,*
- $l; t_L = t_K = r; t_R$*.*

When a production only adds nodes and arcs, it may be reapplied again immediately because the left-hand side is a sub-graph of the right-hand side. If the left-hand side is matched more than once to the same part of the graph to be rewritten, then no real new information is being added. Moreover, this leads to infinite rewriting sequences. We thus restrict the allowed derivations.

**Definition 4** *A direct derivation* $G \overset{p,m}{\Longrightarrow} H$ *typed over a graph* $TG$ *is called* productive *if there are no typed morphisms* $lr : L \to R$ *and* $x : R \to G$ *such that* $lr; x = m$*.*

The existence of morphism $lr$ indicates that it may be possible to apply the production in such a way that no node or arc is deleted. The remaining conditions check that the match $m$ is being applied to a part of $G$ that corresponds to the right-hand side and therefore can have been generated by a previous application of this production. Our definition is a particular case of productions with application conditions in the sense of [13]: a derivation $G \overset{p,m}{\Longrightarrow} H$ is

productive if $p$ is applicable to $G$ using the negative application condition $lr$.

As an example, consider the labelled graph without edges $a \quad b \quad a$ and the production  . A sequence of two direct derivations might lead to the graph  whereas a sequence of two productive derivations can only result in $a \xrightarrow{f} b \xleftarrow{f} a$ and then no further productive derivation is possible.

We can now define a *dynamic reconfiguration step* as a productive direct derivation from a given architecture instance $G$ to an architecture instance $H$. In the algebraic graph transformation approach, there is no restriction on the obtained graphs, but in reconfiguration we must check that the result is indeed an architecture instance, otherwise the rule (with the given match) is not applicable. For example, two separate connector addition rules may each be correct but applying them together may yield indirect sharing of output variables.

At any point in time, the current system is given by an architecture instance without logical variables. Therefore applying a rule to an architecture instance must also involve a compatible substitution of the logical variables occurring in the rule by ground terms. Applying the substitution to the whole rule, we obtain a rule without logical variables whose left hand side can be directly matched to the current architecture. The reconfiguration proceeds as a normal derivation (i.e., as a double pushout over typed graphs). However, the notion of state introduces two constraints. First, the substitution must obviously satisfy the matching condition. Second, the state of each program instance added by the right-hand side satisfies the respective initialisation condition.

**Definition 5** *Given a style AS, an AS-architecture instance $\langle G, t_G \rangle$, an AS-dynamic reconfiguration rule $\langle p_{AS}, mc \rangle$, and a substitution $\phi : Vars(L) \to Terms(\emptyset)$, an AS-*dynamic reconfiguration step $\langle G, t_G \rangle \overset{\phi(p_{AS}),m}{\Longrightarrow} \langle H, t_H \rangle$ *is a productive direct derivation $G \overset{\phi(p_{AS}),m}{\Longrightarrow} H$ typed over $G_{\mathfrak{Inst}}$ such that*

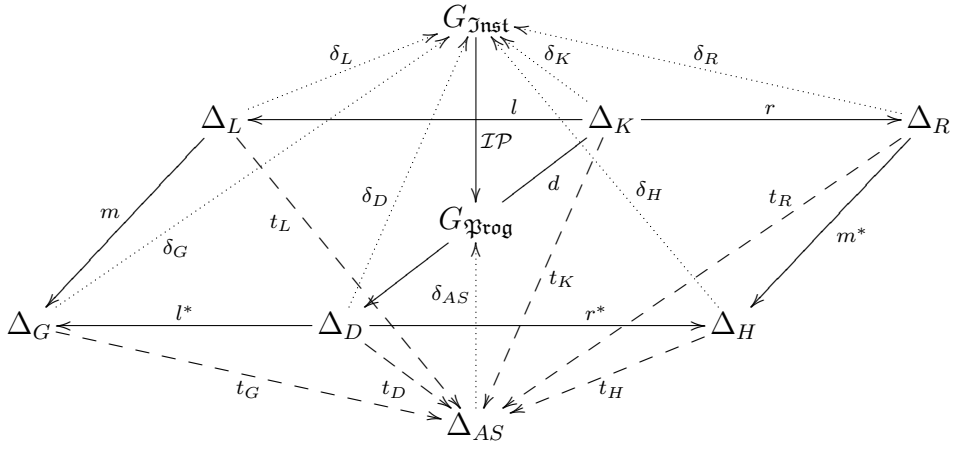
- *$\phi(p_{AS})$ is the rule obtained through replacement of every program instance $\langle P, \epsilon \rangle$ by $\langle P, \epsilon' \rangle$, with $\epsilon'(l) = \phi(\epsilon(l))$ for every $l \in loc(V)$,*
- *$\phi(mc)$ is true,*
- *for each $\langle P, \epsilon \rangle$ in $R \setminus r(K)$, $\phi(\epsilon(I))$ is true,*
- *$H$ is an architecture instance,*
- *$m; t_G = t_L$.*

A rule conforming to a given style can only be applied to architecture instances conforming to the same style, and the last condition states that the match must preserve the typing given by the style. This guarantees that the resulting

architecture also conforms to the style.

**Proposition 6** *The result of a dynamic reconfiguration step conforming to a style AS is always an AS-architecture instance* $\langle H, t_H \rangle$ *with unique* $t_H$*.*

**PROOF.** Consider the following diagram in $\mathfrak{Graph}$.




The morphism $t_D$ exists and is unique because $\langle \Delta_D, t_D \rangle$ is the pushout complement object in $(\mathfrak{Graph} \downarrow \Delta_{AS})$. We now prove $\langle D, t_D \rangle$ is an $AS$-architecture, i.e., that it satisfies Definition 1:

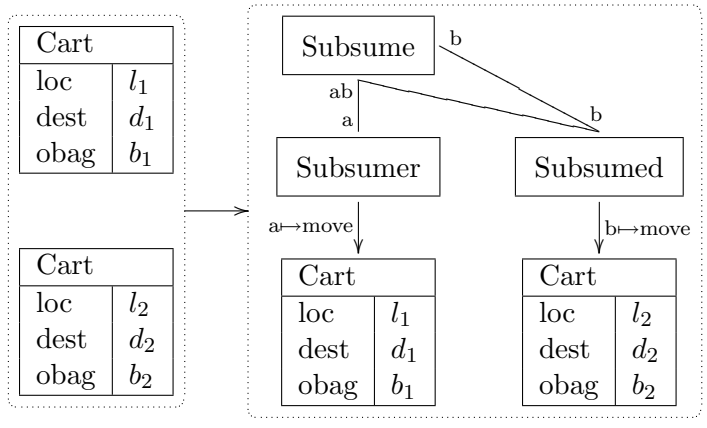
$$\begin{aligned} t_D; \delta_{AS} &= l^*; t_G; \delta_{AS} && \text{construction of } t_D \text{ in } (\mathfrak{Graph} \downarrow \Delta_{AS}) \\ &= l^*; \delta_G; \mathcal{IP} && \langle G, t_G \rangle \text{ is an } AS\text{-architecture} \\ &= \delta_D; \mathcal{IP} && \text{construction of } \delta_D \text{ in } (\mathfrak{Graph} \downarrow G_{\mathfrak{Prog}}) \end{aligned}$$

Similarly, $t_H$ exists and is unique because $\langle H, t_H \rangle$ is the pushout object in $(\mathfrak{Graph} \downarrow \Delta_{AS})$ and we have

$$\begin{aligned} r^*; t_H; \delta_{AS} &= t_D; \delta_{AS} && \text{construction of } t_H \text{ in } (\mathfrak{Graph} \downarrow \Delta_{AS}) \\ &= \delta_D; \mathcal{IP} && \langle D, t_D \rangle \text{ is an } AS\text{-architecture} \\ &= r^*; \delta_H; \mathcal{IP} && \text{construction of } \delta_H \text{ in } (\mathfrak{Graph} \downarrow G_{\mathfrak{Prog}}) \end{aligned}$$

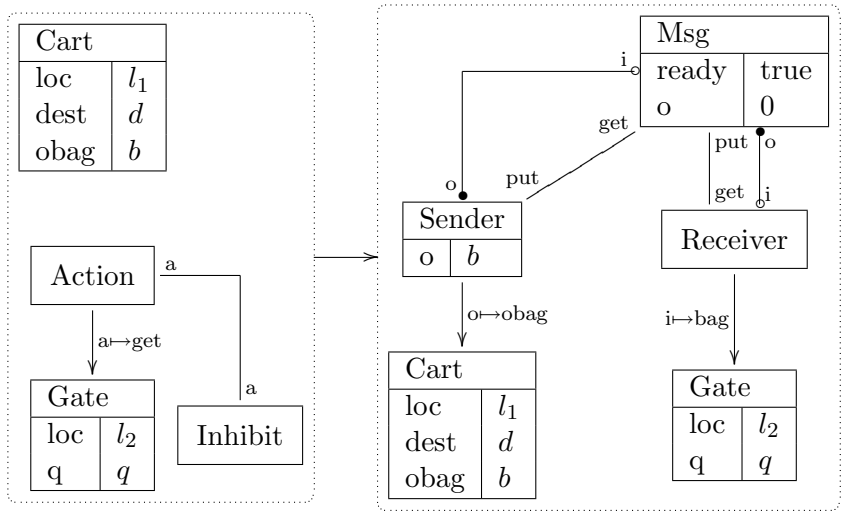
From the uniqueness it results $t_H; \delta_{AS} = \delta_H; \mathcal{IP}$. $\square$

We now start presenting the reconfiguration rules for our running example. Due to page width constraints, we omit the interface graph. A rule $L \xleftarrow{l} K \xrightarrow{r} R$ is simply written as $L \rightarrow R$ where the arrow is only used




**if** $l_2 = l_1 +_U 1$

Fig. 3. Adding an action subsumption connector.



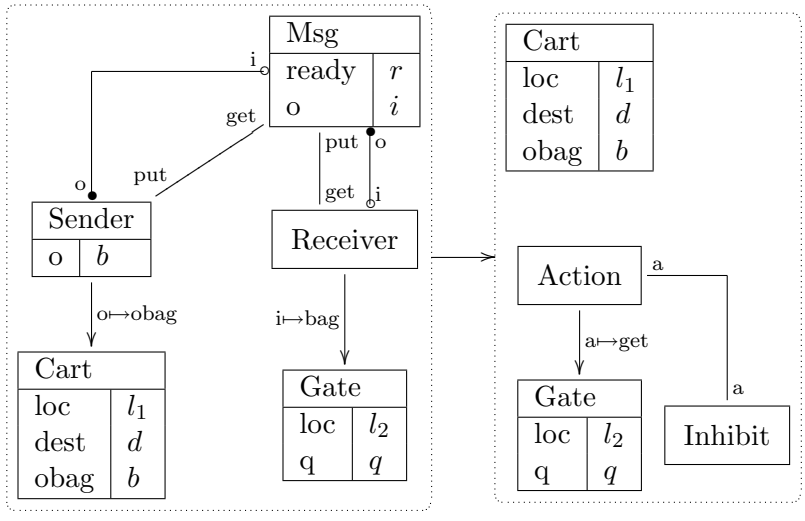

**if** $l_1 = l_2 \wedge d = l_2 \wedge b \neq 0$

Fig. 4. Before unloading a bag.

as a separator. It does not correspond to any total graph morphism. Also, a dynamic rule $\langle p, mc \rangle$ is written $p$ **if** $mc$ or simply $p$, if $mc$ is a tautology.

The rule to avoid a cart colliding with the one in front of it is given in Figure 3. Notice that although logical variables $d_i$ and $b_i$ are not used in any way, they must be stated explicitly because they are part of the program instances that label the graph nodes. To remove the action subsumption connector when it is not longer needed we just use the opposite rule, obtained by switching the left- and right-hand sides, and negating the condition.

The rule in Figure 4 connects a cart to a gate when it passes in front of it. Now only action 'put' can execute (because 'get' is inhibited by the initial architecture and the guard of 'move' is false at this point). This will unload the bag and trigger the opposite rule in Figure 5 to remove the connector.



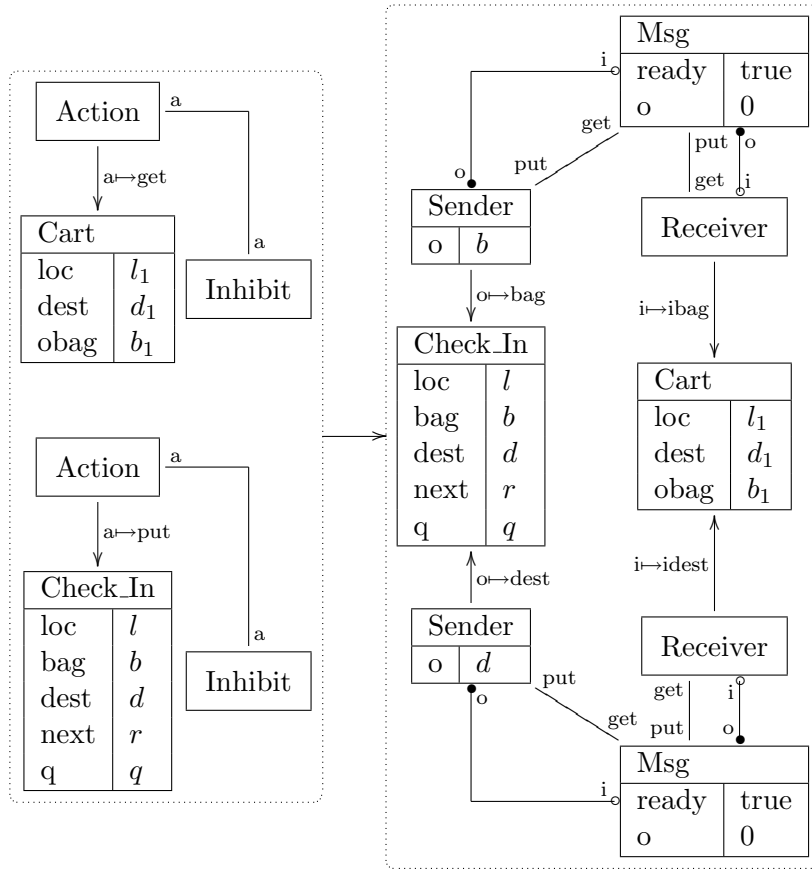

**if** $l_1 \neq l_2 \vee d \neq l_2 \vee b = 0$

Fig. 5. After unloading a bag.

A cart and a check-in station interact when they are co-located, the cart is empty, and the check-in has undelivered bags. In that case the cart gets a new bag and its destination (Figure 6).

To add statistics to the system, it is necessary to add to the style a diagram similar to the one in Section 3.3, but using 'Cart_Stat' instead of 'Cart'. The rules shown so far must also be duplicated for 'Cart_Stat'. Finally we need the replacement rule given in Figure 7. The double-pushout approach guarantees that a cart is replaced by one with statistics only when it is not connected to any other component. This is important both for conceptual reasons—components are not removed during interactions [15]—as technical ones: there will be no "dangling" roles. This example also shows how a rule describes transfer of state from an old to a new component. The transfer may involve both copy of values and arbitrarily complex calculations of new values from the old ones. In this case the initial value of the 'bags' counter depends on whether the cart is carrying a bag or not.

An architecture instance is not just a labelled graph, it is a diagram with a precise semantics, given by its colimit. We can define a computation step of the system as being performed on the colimit and then propagated back to the components of the architecture through the inverse of their morphisms to the colimit. This keeps the state of the program instances in the architectural diagram consistent with the state of the colimit, and ensures that at each point in time the correct conditional rules are applied. As [20] we adopt a two-phase approach: each computation step is followed by a reconfiguration sequence. In this way, the specification of the components is simpler, because it is guaranteed that the necessary interconnections are in place as soon as required by the state of the components. In our example, a cart simply moves




**if** $l = l_1 \wedge d_1 = -1 \wedge b_1 = 0 \wedge q \neq []$

Fig. 6. Before loading a bag from a check-in station

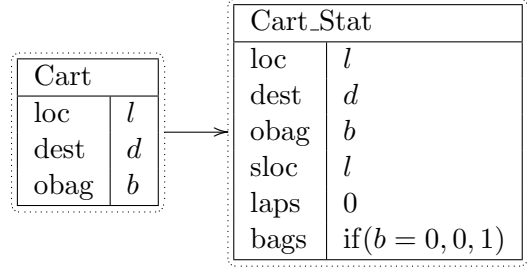



Fig. 7. Replacement of 'Cart' by 'Cart_Stat'.

forward without any concern for its location. Without the guarantee that an action subsumption connector will exist whenever necessary, a cart would have to know at all times the locations of the other carts to be sure it would not collide with one of them. And this would make the system much more complex.

**Definition 7** *Given a style AS, an initial architecture instance G conforming to AS, and a set of AS-dynamic reconfiguration rules, the configuration*

*manager performs the following steps:*

*(1) allow the user to change AS and the set of rules;*
*(2) find a maximal sequence of AS-dynamic reconfiguration steps starting with $G$, obtaining a new diagram $G'$;*
*(3) compute the colimit $S$ of $G'$;*
*(4) if none of the actions of $S$ can be executed, stop, otherwise update the valuation of $S$ according to the chosen action;*
*(5) propagate back the changes to the valuations of the program instances of $G'$, call the new diagram $G$, and go back to step 1.*

The first step caters for ad-hoc reconfiguration. For our system, it allows to make the necessary additions to handle 'Cart_Stat' programs.

It should be stressed that the above definition only provides the semantics of the reconfiguration process. An actual implementation would not compute the colimit explicitly, but execute the architecture directly in a distributed way, taking the sharing of variables and synchronisation of actions into account.

## 5 Concluding Remarks

This paper presents an algebraic foundation for software architecture reconfiguration. The approach is based on three pillars: the general framework of Category Theory; the category of typed graphs and their morphisms; the category of CommUnity programs with morphisms that capture superposition and refinement. The first two allow us to use in a straightforward way the double pushout approach to graph transformation. The main advantages of this approach are:

- Architectures, reconfigurations, and connectors are represented and manipulated in a graphical yet mathematical rigorous way at the same language-independent level of abstraction, resulting in a very uniform framework based simply on diagrams and their colimits.
- The chosen program design language is at a higher level of abstraction than process calculi or term rewriting, allowing a more intuitive representation of program state and computations.
- Computations and reconfigurations are kept separate but related in an explicit, simple, and direct way through the colimit construction.
- Typed graph morphisms capture in a declarative way some simple architectural invariants.
- Several practical problems—maintaining the style during reconfiguration, transferring the state during replacement, removing components in a quiescent state, adding components properly initialized—are easily handled.

Within project FAST, we are considering the following possibilities for future work:

- Implement the approach, e.g., by incorporating a library to compute colimits on graphs [35] into a CommUnity tool to be developed.
- Look into and try to adapt work on graph rewriting termination [26] and sequential independence to be able to analyse the possible reconfiguration sequences.
- Adapt and extend the logic presented in [9] for reasoning about the reconfiguration process.

This future research lines are along the spirit of the work presented in this paper, namely an investigation into solid formal foundations for dynamic reconfiguration, not the development of an actual specification language. For that purpose we are currently designing a language to specify architectures, complex constraints on them, and reconfiguration scripts. A preliminary attempt has been already presented [33]. The goal is to be able to specify more expressive invariants than those described by typed graphs, and to use high-level programming constructs (like sequencing, choice, and iteration) to easily control in which way the basic changes (i.e., addition and removal of components and connectors) are executed. Practical feedback from this ongoing research will be gathered by incorporating such reconfiguration primitives into a tool being built to construct and manage coordination contracts among components implementing core business functionalities [12].

## Acknowledgements

We thank Antónia Lopes for her comments on a previous version of this work.

## References

[1] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Fundamental Approaches to Software Engineering*, LNCS 1382, pages 21–37. Springer-Verlag, 1998.

[2] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *Software Architecture*, pages 107–125. Kluwer Academic Publishers, 1999.

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

[4] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamentae Informatica*, 26(3–4):241–266, 1996.

[5] Description of EDCS technology clusters. *ACM SIGSOFT Software Engineering Notes*, 22(5):33–42, Sept. 1997.

[6] M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proc. of the 12th Brazilian Symposium on Computer Networks*, pages 175–187, 1994.

[7] J. L. Fiadeiro and A. Lopes. Semantics of architectural connectors. In *Proc. of TAPSOFT'97*, LNCS 1214, pages 505–519. Springer-Verlag, 1997.

[8] J. L. Fiadeiro and T. Maibaum. Categorial semantics of parallel program design. *Science of Computer Programming*, 28:111–138, 1997.

[9] J. L. Fiadeiro, N. Martí-Oliet, T. Maibaum, J. Meseguer, and I. Pita. Towards a verification logic for rewriting logic. In *Recent Trends in Algebraic Development Techniques*, LNCS 1827, pages 438–458. Springer-Verlag, 2000.

[10] J. L. Fiadeiro, M. Wermelinger, and J. Meseguer. Semantics of transient connectors in rewriting logic. Position Paper for the First IFIP Working Intl. Conference on Software Architecture, Feb. 1999.

[11] N. Francez and I. Forman. *Interacting Processes*. Addison-Wesley, 1996.

[12] J. Gouveia, G. Koutsoukos, L. Andrade, and J. L. Fiadeiro. Tool support for coordination-based software evolution. In *Proc. TOOLS Europe*. Prentice-Hall, 2001.

[13] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3–4), 1996.

[14] D. Hirsch, P. Inverardi, and U. Montanari. Reconfiguration of software architecture styles with name mobility. In *Coordination Languages and Models*, LNCS 1906, 2000.

[15] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov. 1990.

[16] A. Lopes. *Não-determinismo e Composicionalidade na Especificação de Sistemas Reactivos*. PhD thesis, Universidade de Lisboa, Jan. 1999.

[17] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proc. of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14. ACM Press, 1996.

[18] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Software Architecture*, pages 35–50. Kluwer Academic Publishers, 1999.

[19] J. N. Magee and D. E. Perry. Welcome to ISAW-3. In *Proc. 3rd Intl. Software Architecture Workshop*, pages vii–viii. ACM Press, 1998.

[20] P. J. McCann and G.-C. Roman. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering*, 24(2), Feb. 1998.

[21] N. Medvidovic. ADLs and dynamic architecture changes. In *Joint Proc. of the SIGSOFT'96 Workshops*, pages 24–27. ACM Press, 1996.

[22] D. L. Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–553, July 1998.

[23] R. T. Monroe, D. Garlan, and D. Wile. *Acme StrawManual*, Nov. 1997.

[24] P. Oreizy. Issues in the runtime modification of software architectures. Technical Report UCI-ICS-TR-96-35, Department of Information and Computer Science, University of California, Irvine, Aug. 1996.

[25] D. E. Perry. State-of-the-art: Software architecture. In *Proc. of the 19th Intl. Conference on Software Engineering*, pages 590–591. ACM Press, 1997.

[26] D. Plump. On termination of graph rewriting. In *Proc. of the 21st Intl. Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 1017, pages 88–100. Springer-Verlag, 1995.

[27] G.-C. Roman, P. J. McCann, and J. Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM TOSEM*, 6(3):250–282, July 1997.

[28] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation*, 1998.

[29] M. Wermelinger. Towards a chemical model for software architecture reconfiguration. *IEE Proceedings—Software*, 145(5):130–136, Oct. 1998.

[30] M. Wermelinger. *Specification of Software Architecture Reconfiguration.* PhD thesis, Universidade Nova de Lisboa, Sept. 1999.

[31] M. Wermelinger and J. L. Fiadeiro. Connectors for mobile programs. *IEEE Transactions on Software Engineering*, 24(5):331–341, May 1998.

[32] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. Superposing connectors. In *Proc. of the 10th Intl. Workshop of Software Specification and Design*, pages 87–94. IEEE Computer Society Press, 2000.

[33] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proc. of ESEC/FSE*, pages 21–32. ACM Press, 2001.

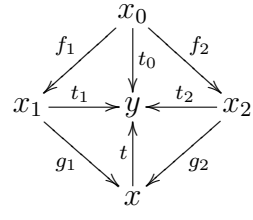
[34] A. L. Wolf. Succeedings of the Second Intl. Software Architecture Workshop. *ACM SIGSOFT Software Engineering Notes*, 22(1):42–56, Jan. 1997.

[35] D. Wolz. *Colimit Library for Graph Transformations and Algebraic Development Techniques.* PhD thesis, Technische Universität Berlin, 1998.

## A Typed Graphs

A typed graph $\langle G, t\rangle$ is a graph $G$ equipped with a morphism $t : G \to TG$ to a fixed graph $TG$, the type graph [4]. Intuitively, $TG$ restricts the allowed nodes and arcs, and $t$ provides the typing of $G$'s nodes and arcs. A special case of typed graphs are labelled graphs: $TG$ contains one node for each node label, and between each pair of nodes there is one arc for each arc label.

A typed graph morphism $f : \langle G, t\rangle \to \langle G', t'\rangle$ is a graph morphism $f : G \to G'$ that preserves the typing, i.e., $t = f; t'$. The category $\mathfrak{Graph}_{TG}$ of graphs typed by $TG$ is the comma category $(\mathfrak{Graph} \downarrow TG)$. Computing the colimit in comma categories amounts to calculate it in the underlying category, and the same for pushout complements. Consider the following diagram in $\mathfrak{C}$:




If $g_1$ and $g_2$ are the pushout of $f_1$ and $f_2$ in $\mathfrak{C}$, then they are also the pushout in $(\mathfrak{C} \downarrow y)$. In fact, $t_1$ and $t_2$ are a cocone of $f_1$ and $f_2$ in $\mathfrak{C}$, and therefore $t$ exists and is unique due to the universal property of colimits. On the other hand, if $f_2$ and $g_2$ are a pushout complement of $f_1$ and $g_1$ in $\mathfrak{C}$, they are also in $(\mathfrak{C} \downarrow y)$ with $t_2 = g_2; t$.

All the underlying mathematical machinery for this work is based uniformely on typed graphs. A category $\mathfrak{C}$ can be seen as a graph $G_{\mathfrak{C}}$ with objects as nodes and morphisms as arrows, subject to the usual conditions on identities and compositionality. A diagram $D = \langle \Delta_D, \delta_D\rangle$ in $\mathfrak{C}$ is then simply a graph $\Delta_D$ typed (via $\delta_D$) by $G_{\mathfrak{C}}$.