Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 109 (2004) 111–124

www.elsevier.com/locate/entcs

# An Operational Semantics of CommUnity Based on Graph Transformation Systems [1]

Andrea Corradini    Dan Hirsch

*Dipartimento di Informatica, Università di Pisa,*
*Via F. Buonarroti 2, I-56127, Pisa, Italia* [2]

**Abstract**

We propose an operational semantics, based on graph transformation, of CommUnity, a simple program design language. Each action of a single CommUnity design is modeled by a synchronized hyperedge replacement rule. Synchronized actions of several interconnected designs in a configuration result automatically from the individual rules thanks to the rule synchronization mechanism.

*Keywords:* Graph transformation systems, operational semantics, software architectures.

## 1 Introduction

Architectural approaches to software design view system architectures as structured in terms of components (where actual computations take place) glued together by connectors, which prescribe how components interact. CommUnity [4] is a simple parallel program design language, developed to support analysis and formalization of architectural semantic primitives. The basic CommUnity components, called *designs*, specify the effect of certain *actions* over suitable *channels*. Some channels may be affected by actions (*local* ones), others may only be read (*input* ones) as their value is provided by the environment.

[1] Research partially supported by the EU FET – GC Project IST-2001-32747 AGILE and the EC RTN 2-2001-00346 SegraVis (Syntactic and Semantic Integration of Visual Modelling Techniques).

[2] Email: andrea@di.unipi.it, dhirsch@di.unipi.it

doi:10.1016/j.entcs.2004.02.060

If the effect of actions is fully specified, as we shall assume along the paper, and the channels are associated with corresponding values (i.e., the design is *anchored*), the firing of an action is possible when its *guard* holds true, and its effect is a multiple assignment on the local channels. However, this can only happen in an environment providing values for the input channels.

In a *configuration*, designs are interconnected through *cables*, which determine design action synchronizations, and channel sharing. Some constraints are imposed: e.g., two local channels cannot be shared, as they could contain inconsistent values. Such constraints guarantee the existence of a *colimit design* for the configuration, having one action for each possible action synchronization of individual designs. The operational behaviour of a configuration is described simply as the legal sequences of actions of the colimit design.

We propose here to use graph transformation systems to provide an intuitive, inductive and complete account of the operational behaviour of CommUnity configurations. A configuration is represented as a hypergraph, including one (hyper)edge for each design and auxiliary edges for encoding the anchoring and the interconnections among designs. Each action of a single CommUnity design is modeled by a conditional, attributed, synchronized hyperedge replacement rule. The synchronization mechanism guarantees that in order to fire, some of such rules must synchronize resulting in a step that faithfully corresponds to the firing of a synchronized action of the colimit design.

The proposed semantics does not exploit fully the graphical representation of configurations, as rules in this paper only change node attributes. However, full expressiveness of graph transformation will be used when modeling general CommUnity reconfigurations, which is the goal of ongoing research: this will be possible because the graphical encoding of a configuration state explicitly represents the morphisms among designs as suitable interconnections.

## 2 CommUnity designs and configurations

We briefly introduce here CommUnity *designs* and *configurations*. Definitions are less general and simplified with respect to those in, e.g., [4,9], where the reader can find the original formulation and more detailed explanations.

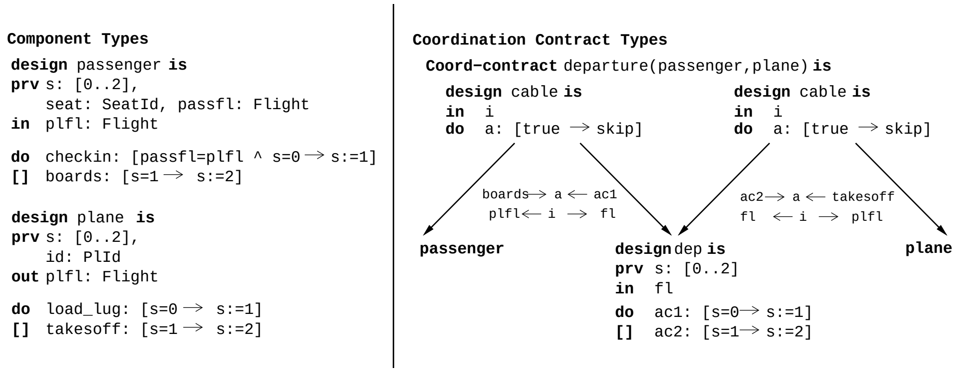
The syntax of a CommUnity design is shown in Figure 1. Design $D$ has a set of *channels* $V$, partitioned into *input* ($in(V)$), *output* ($out(V)$), and *private channels* ($prv(V)$). The *local channels* of $D$ are defined as $loc(V) = out(V) \cup prv(V)$. Design $D$ also has a set of *actions* $\Gamma$, partitioned into *shared* ($sh(\Gamma)$) and *private actions* ($prv(\Gamma)$). For each action $a \in \Gamma$, $G(a)$ is its *guard* (a boolean condition over channels, omitted when it is `true`), $D(a)$ is its *domain*, i.e., the set of local channels it can change, and for every local channel $l$ in

$$\begin{array}{llll}
\textbf{design} & D & & \\
\textbf{in} & \mathit{in}(V) & & \\
\textbf{out} & \mathit{out}(V) & & \\
\textbf{prv} & \mathit{prv}(V) & & \\
\textbf{do} & \underset{a \in sh(\Gamma)}{[]} \quad a : G(a) \rightarrow & \underset{l \in D(a)}{\|} & l := E(a,l) \\
[] & \underset{a \in prv(\Gamma)}{[]} \quad \text{prv } a : G(a) \rightarrow & \underset{l \in D(a)}{\|} & l := E(a,l)
\end{array}$$

Fig. 1. Syntax of a CommUnity design.

$D(a)$, $l := E(a,l)$ is an assignment, where $E(a,l)$ is an expression over channels (`skip` denotes the absence of assignments).

An *anchored design* is a design equipped with an *evaluation*, associating each local channel to an element of a fixed data algebra $\mathcal{D}_\Sigma$, where $\Sigma$ is a standard algebraic signature.[3] Two sample designs (`passenger` and `plane`) are shown in the left part of Figure 2: they are borrowed (with little modifications) from an example in [1], where the check-in and boarding of passengers and the take off of planes at an airport are modeled using CommUnity.



```
Component Types
design passenger is
prv s: [0..2],
    seat: SeatId, passfl: Flight
in  plfl: Flight

do  checkin: [passfl=plfl ^ s=0 → s:=1]
[]  boards: [s=1 →  s:=2]

design plane  is
prv s: [0..2],
    id: PlId
out plfl: Flight

do  load_lug: [s=0 →  s:=1]
[]  takesoff: [s=1 →  s:=2]
```

```
Coordination Contract Types
Coord-contract departure(passenger,plane) is
  design cable is            design cable is
  in  i                      in  i
  do  a: [true → skip]       do  a: [true → skip]

design dep is
prv  s: [0..2]
in   fl
do   ac1: [s=0→ s:=1]
[]   ac2: [s=1→ s:=2]
```

Fig. 2. Two CommUnity designs (left) and a configuration (right).

Interactions between designs are established through action synchronization and memory sharing, which are obtained by introducing degenerated designs called *cables*, and explicit morphisms from cables to designs. As cables add no computations of their own, they have only input channels and shared actions with `true` guards and no assignments. A *(superposition) morphism* $\sigma : C \rightarrow D$ from a cable $C$ to a design $D$ maps each (input) channel $v$ of $C$ to a non-private (i.e., input or output) channel $\sigma_V(v)$ of $D$, and each (shared) action $a$ of $C$ to a non-empty set of non-private (i.e., shared) actions $\sigma_\Gamma(a) = \{a'_1, \ldots, a'_n\}$

[3] Typically, the channels in $V$ are typed and a standard typing discipline is imposed over their manipulation. In this case $\Sigma$ is a many-sorted signature, having the relevant types as sorts. All these details are not elaborated here for the sake of conciseness.

of $D$.[4] Intuitively, such morphism identifies the channels $v$ and $\sigma_V(v)$ (they become *shared*: any value associated to one of them is available at the other), and synchronizes action $a$ with any of the actions in $\sigma_\Gamma(a)$ ($a$ can fire if and only if one of the actions in $\sigma_\Gamma(a)$ fires simultaneously).

A *configuration* is a finite diagram made of designs, cables and morphisms from cables to designs, such that every input channel is shared (through an undirected path of morphisms) with exactly one output channel. This constraint guarantees that if the configuration designs are anchored, each channel is associated with exactly one value. A sample configuration is shown in the right part of Figure 2, where the designs `passenger` and `plane` are connected via a third design `dep` and two cables. The morphisms cause the input channel `plfl` of `passenger` to be shared with the output channel `plfl` of `plane`. Furthermore, action `board` of `passenger` (synchronized with `ac1` of `dep`) is forced to fire before action `takesoff` of `plane` (synchronized with `ac2`).

```
design 1passenger1plane is
prv spl, sdep, spass: [0..2], id:PlId
    seat: SeatId, passfl: Flight
out plfl: Flight
do  checkin: [passfl=plfl ^ spass=0→ spass:=1]
[]  load_lug: [spl=0 → spl:=1]
[]  checkinLoad_lug: [passfl=plfl ^ spass=0 ^ spl=0→ spass:=1 || spl:=1]
[]  boardsAc1Load_lug: [spass=1 ^ sdep=0 ^ spl=0→ spass:=2 || sdep:=1 || spl:=1]
[]  checkinAc2Takesoff: [passfl=plfl ^ spass=0 ^ sdep=1 ^ spl=1→ spass:=1 || sdep:=2 || spl:=2]
[]  boardsAc1: [spass=1 ^ sdep=0→ spass:=2 || sdep:=1]
[]  ac2Takesoff: [sdep=1 ^ spl=1 → sdep:=2 || spl:=2]
```

Fig. 3. The colimit of the configuration of Figure 2

Every (anchored) configuration has a colimit (in the category of (anchored) designs and superposition morphisms), which is a design having as channels the disjoint union of channels of the configuration (modulo shared channels), and as actions the cartesian product of actions (modulo synchronized ones).

The operational behaviour of a configuration is described in terms of its colimit: at each step, an action of the colimit whose guard evaluates to `true` is selected non-deterministically, and the corresponding multiple assignment is executed. For example, in the colimit of the configuration of Figure 2 (shown in Figure 3), actions `checkin` of `passenger` and `load_lug` of `plane` are independent (they can fire alone or simultaneously), while `boardsAc1` models synchronous execution of actions `boards` of `passenger` and `ac1` of `dep`; actions `boardsAc1Load_lug` and `checkinAc2Takesoff` will never be fired from the anchored configuration of Figure 6, because their guards are always false.

[4] $\sigma_\Gamma$ can also be defined as a surjective, partial function from actions of $D$ to those of $C$.

## 3 Graph transformation systems

We introduce here the notion of graph transformation that we shall use to model the evolution of CommUnity configurations. The formalism we use can be defined as *attributed, conditional, synchronized hyperedge replacement with value passing Hoare synchronization*. Indeed, we borrow various ingredients presented elsewhere from the literature, composing them in an original way that is tailored to our specific needs. We start introducing (parallel) hypergraph rewriting according to the *double-pushout approach* [2,3]. Next we enrich the formalism first with attributions [7], and then with synchronization [6] and with application conditions.

### *3.1 Hypergraph rewriting*

Hypergraphs are made of nodes and labeled edges. Each edge may be connected to several nodes specified by a connection function. Each 'tentacle', i.e., each connection between an edge and a node, is labeled by a 'name' (instead of a number, as usual in literature). The number of connections of an edge and the names of such connections are uniquely determined by the edge label: this information is provided by a fixed *edge signature* $\Delta = \langle CN, \Lambda, rnk \rangle$, where $CN$ is a set of *connection names*, $\Lambda$ is a set of *edge labels*, and $rnk : \Lambda \to \mathcal{P}_{fin}(CN)$ is a function associating to each edge label a finite set of connection names.

A $\Delta$-*(hyper)graph* $H$ is a tuple $\langle V_H, E_H, lab_H, c_H \rangle$, where $V_H$ is a set of nodes, $E_H$ is a set of (hyper)edges, $lab_H : E_H \to \Lambda$ is the labeling function, and $c_H : E_H \to [CN \nrightarrow V_H]$ is the connection function, associating with each edge a partial function from connection names to nodes. It is required that $dom(c_H(e)) = rnk(lab_H(e))$ for each $e \in E_H$. If $c_H(e)(x) = v$ we call $v$ the $x$-node of $e$. Hypergraph morphisms are defined in the expected way.

A *rule* is a span of *injective* $\Delta$-graph morphisms $q = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$, where $L$, $K$, $R$ are finite $\Delta$-graphs, $L$ has no isolated nodes, the left-hand side morphism $l$ is surjective on nodes (i.e., no node is deleted by the rule), and graph $K$ is discrete (it contains no edge). A *match* of such a rule in a $\Delta$-graph $G$ is a morphism $g : L \to G$ which is injective on edges. [5] In this case we write $G \Rightarrow_q H$, if both squares in the following diagram are pushouts:

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

[5] This ensures that the *gluing conditions* of the DPO approach are satisfied.

A $\Delta$-*graph transformation system (GTS)* $\mathcal{R} = \langle P, \pi \rangle$ is made of a set of rule names $P$ and of a function $\pi$ which assigns to each $p \in P$ a rule $\pi(p) = (L_p \hookleftarrow K_p \hookrightarrow R_p)$. We write $G \Rightarrow_{\mathcal{R}} H$ if $G \Rightarrow_p H$ for some $p \in P$. Given a GTS $\mathcal{R} = \langle P, \pi \rangle$, let $\mathcal{R}^{\oplus} = \langle P^{\oplus}, \pi^{\oplus} \rangle$ be the GTS where $P^{\oplus}$ is the (carrier of the) free commutative monoid generated by $P$ (excluding unit), and $\pi^{\oplus}$ is the free extension of $\pi$, mapping $p \oplus p'$ to the disjoint union of rules $\pi^{\oplus}(p)$ and $\pi^{\oplus}(p')$.[6] Then we say that there is a *parallel rewriting* from $G$ to $H$ in $\mathcal{R}$ iff $G \Rightarrow_{\mathcal{R}^{\oplus}} H$.

## 3.2 Attributed hypergraph rewriting

Let $\Sigma$ be a standard algebraic *one-sorted* signature, which is fixed for the rest of the section.[7] A $\Sigma$-*attributed* $\Delta$-*hypergraph* (or $\Sigma$,$\Delta$-*graph*) is a triple $G = \langle |G|, \mathcal{A}_G, att_G \rangle$, where $|G|$ is a $\Delta$-graph (the *underlying* $\Delta$-*graph* of $G$), $\mathcal{A}_G$ is a $\Sigma$-algebra, and $att_G : V_{|G|} \rightharpoonup |\mathcal{A}_G|$ is an *attribution*, i.e., a partial function mapping nodes of $|G|$ to elements of the carrier of the algebra $\mathcal{A}_G$. A $\Sigma$,$\Delta$-*graph morphism* $f : G \to G'$ is a pair $\langle |f|, f_A \rangle$, where $|f| : |G| \to |G'|$ is a $\Delta$-graph morphism, and $f_A : \mathcal{A}_G \to \mathcal{A}_{G'}$ is a total $\Sigma$-homomorphism such that $f_A \circ att_G = att_{G'} \circ |f|_V$.

This definition of attributed graph is very general, because every graph is equipped with its own $\Sigma$-algebra. We stick here to the case where the attributed graphs appearing in a rule have associated $T_\Sigma(X)$, the free algebra over $\Sigma$ generated by a countable set of variables $X$, while the graphs which are rewritten all have as algebra a fixed *data algebra* $\mathcal{D}_\Sigma$. An *attributed rule* is a span of *injective* $\Sigma$,$\Delta$-graph morphisms $r = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$, such that:

- the underlying span of $\Delta$-graph morphisms $(|L| \overset{|l|}{\hookleftarrow} |K| \overset{|r|}{\hookrightarrow} |R|)$ is a rule;
- the $\Sigma$-algebra associated with $L$, $K$, $R$ is $T_\Sigma(X)$, and $l_A, r_A$ are the identities;
- graph $L$ is *linear*, i.e., all its attributed nodes are labeled either by ground terms or by variables, and no variable labels more than one node;
- $Var(R) \subseteq Var(L)$, where $Var(H) \subseteq X$ is the set of variables appearing in the attributions of $H$.

A *match* of an attributed rule $r = (L \hookleftarrow K \hookrightarrow R)$ in a $\Sigma$,$\Delta$-graph $G$ over $\mathcal{D}_\Sigma$ is a $\Sigma$,$\Delta$-graph morphism $g : L \to G$ which is injective on edges. Such a map determines an *evaluation* $\theta_g : Var(L) \to \mathcal{D}_\Sigma$ defined as $\theta_g(x) = att_G(g_V(n))$ iff $att_L(n) = x$ (this is well defined thanks to linearity of $L$). By the freeness of $T_\Sigma(X)$, $\theta_g$ extends uniquely to a homomorphism $\theta_g : T_\Sigma(X) \to \mathcal{D}_\Sigma$: by applying it to all attributions appearing in the attributed rule, we get

6 This definition only works up to isomorphims.

7 All notions easily generalize to the *many-sorted* case, which is used indeed in the examples.

a corresponding span $r\theta_g = (L\theta_g \hookleftarrow K\theta_g \hookrightarrow R\theta_g)$, where graphs are over $\mathcal{D}_\Sigma$.

In this situation we write $G \Rightarrow_r H$ if $H = \langle |H|, \mathcal{D}_\Sigma, att_H \rangle$, there is a $\Delta$-rewriting step $|G| \Rightarrow_{|r\theta_g|} |H|$, and $att_H$ is uniquely determined by lifting the double-pushout diagram to the category of $\Sigma$,$\Delta$-graphs, assuming that all $\Sigma$-homomorphism components are identities over $\mathcal{D}_\Sigma$.

The definition of parallel rewriting is the same as for the unattributed case, provided that forming a parallel attributed rule the component rules are renamed apart, in order to avoid variable name clashes.

### 3.3 *Conditional Synchronized Hypergraph Rewriting*

We introduce now a powerful synchronization mechanism for attributed rules, which in the terminology of [6] is called *Hoare synchronization with value passing*. Intuitively, a rule may have some nodes of the left-hand side annotated with *events*, which are atomic predicates built over a set of *event names* and terms over $\Sigma$. These events may prevent the application of the rule at a given match, unless one or more other rules, annotated with corresponding matching events on the same nodes, are applied synchronously. In order to model faithfully the actions of CommUnity designs, we shall equip rules also with boolean conditions that will encode the guards.

Let $Ev = \cup_{n \in \mathbb{N}} Ev_n$ be a fixed, ranked set of *event names*, and $\mathcal{E}_\Sigma(X)$ be the collection of *well-formed events over* $X$, containing all terms of the form $e(t_1, \ldots, t_n)$, with $e \in Ev_n$ and $t_i \in T_\Sigma(X)$ for all $i \in \{1, \ldots, n\}$. A (*conditional*) *synchronized* (*attributed*) *rule* is a triple $\langle r, synch, cond \rangle$, namely an attributed rule $r = (L \hookleftarrow K \hookrightarrow R)$ equipped with a partial function $synch : V_{|L|} \nrightarrow \mathcal{E}_\Sigma(X)$, and with a formula *cond* in a logic sufficiently expressive to encode CommUnity guards. Denoting by $Var(synch)$ the set of variables appearing in the events in $synch(V_{|L|})$, the following condition must hold: $Var(R) \cup Var(cond) \subseteq Var(L) \cup Var(synch)$. This condition will guarantee that all variables in *cond* and in the right-hand side of the rule will be instantiated by any legal match of the left-hand side.

A *synchronized (attributed conditional* $\Sigma$,$\Delta$*-hyper)graph transformation system (SGTS)* $\mathcal{S} = \langle P, \pi \rangle$ is made of a set of rule names $P$ and of a function $\pi$ mapping each $p \in P$ to a synchronized rule $\pi(p) = \langle r_p, synch_p, cond_p \rangle$. Then, the parallel SGTS $\mathcal{S}^\oplus = \langle P^\oplus, \pi^\oplus \rangle$ has $P^\oplus$, the free commutative monoid generated by $P$ (excluding unit) as rule names. Moreover, if $p = p_1 \oplus \ldots \oplus p_k \in P^\oplus$, with $p_i \in P$ and $\pi(p_i) = \langle r_i, synch_i, cond_i \rangle$ for all $i = 1, \ldots, k$, then the parallel synchronized rule $\pi^\oplus(p) = \langle r, synch, cond \rangle$ is obtained by first applying suitable variable renamings in order to avoid name clashes among rules, and then taking as $r$ the disjoint union of all $r_i$'s, as *synch* the obvious partial function induced by all $synch_i$'s, and as *cond* the *conjunction* of all $cond_i$'s.

Let $s = \langle r = (L \hookleftarrow K \hookrightarrow R), synch, cond\rangle$ be a (typically parallel) synchronized rule, and let $g : L \to G$ be a match to a $\Sigma$,$\Delta$-graph $G$. Furthermore, let $\theta_g : Var(L) \to \mathcal{D}_\Sigma$ be the evaluation induced by $g$, and $synch(g) \overset{def}{=} g(dom(synch)) \subseteq V_{|G|}$ be the *syncronization nodes* of $g$, i.e., those which are image of nodes of $L$ annotated with events. Then rule $s$ can be applied to $g$ if

- every edge of $G$ connected to a node in $synch(g)$ is in the image of $g$;
- $\forall n \in synch(g)$, $g^{-1}(n) \subseteq dom(synch)$, i.e., each node mapped by $g$ to a synchronization node is annotated with an event;
- there exists a ground substitution $\theta : Var(synch\theta_g) \to \mathcal{D}_\Sigma$ such that for each node $n \in synch(g)$ and for all nodes $m, m' \in g^{-1}(n)$, $synch(m)\theta_g\theta = synch(m')\theta_g\theta$; furthermore $\theta$ must be minimal, in the sense that it must be a most general unifier of the involved partitioned set of events;
- the formula $cond\theta_g\theta$ evaluates to `true`.

Under these conditions, we write $G \Rightarrow_s H$ if $G \Rightarrow_{r'} H$ using the match $g$, where $r'$ is the attributed rule $r' = (L \hookleftarrow K \hookrightarrow R\theta)$.

## 4 CommUnity configurations as attributed hypergraphs

We describe now how to represent CommUnity configurations as hypergraphs. The synchronized rules presented in the next section, when applied to such a graph, will simulate the operational behaviour of the colimit of the configuration. The graph representing a configuration includes one edge for each design, and *auxiliary edges* for encoding the anchorings and the morphisms. Figure 4 shows the graphs encoding two anchored designs, while Figure 5 shows the graphical representation of a simple CommUnity morphism.

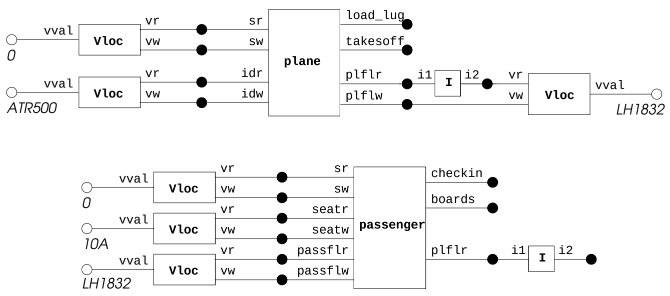


Fig. 4. Graphs for two anchored designs.

As it can be grasped from the figures, the edge signature includes as edge labels all design names, as well as labels `Vloc`, `I`, `Vin` and `Cn` for auxiliary

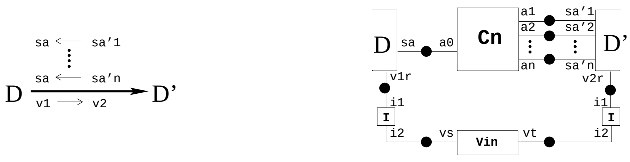


Fig. 5. A CommUnity morphism (left) and its graphical encoding (right).

edges. For each design name, the connection names include `xr` for each input channel $x$ (this name will label the "read tentacle" connecting the edge to the channel), `lr` and `lw` for each local channel (labeling the corresponding "read" and "write" tentacles), and the names of all shared actions (labeling the tentacles used for synchronizing actions; thus private actions have no tentacle).

The *local boxes*, i.e., the edges labeled with `Vloc`, are used to handle the access to local channels. The tentacle labeled by `vval` points to the attributed node storing the channel value, while those labeled with `vr` and `vw` provide access for reading and changing the value. *Interface boxes*, labeled by `I` are used to provide read access to non-private (i.e., input or output) channels.

As shown in Figure 5, a morphism $\sigma : D \to D'$ is represented using *input boxes* (labeled by `Vin`) and *OR boxes* (labeled by `Cn`). Input boxes connect the interface boxes of the channels identified by the morphism, while OR boxes connect the tentacle of the source design labeled by an action `sa` to the non-empty set of tentacles labeled by the actions in $\sigma_\Gamma$(`sa`). Figure 6 shows a complete anchored configuration, and its graphical representation.

## 5 Evolution of configurations via synchronized rules

The behaviour of a configuration is specified by a set of synchronized *hyper-edge replacement* rules, i.e., rules where the left-hand side contains a single hyperedge. Every design action is modeled by a rule, and additional rules are provided for the auxiliary edges. Almost all rules (but those for `Vloc` edges which change the value) have the same graph as left- and right-hand sides, thus all the computation is performed through the synchronization mechanism. Three event names are used: `go` for synchronizing actions, and `get` and `set` for reading and changing the channel values.

Figures 7 and 8 show the rules for all the actions of the designs of the running example. In general, the rule encoding action $a : G(a) \to \;\|_{l \in D(a)}\; l := E(a,l)$ will have $G(a)$ as condition, an event `go` annotating the $a$-node if the action is not private, an event `get(x)` annotating the `xr`-node if the value of channel `x` is used in the guard or in an assignment, and an event `set(E(a,l))`

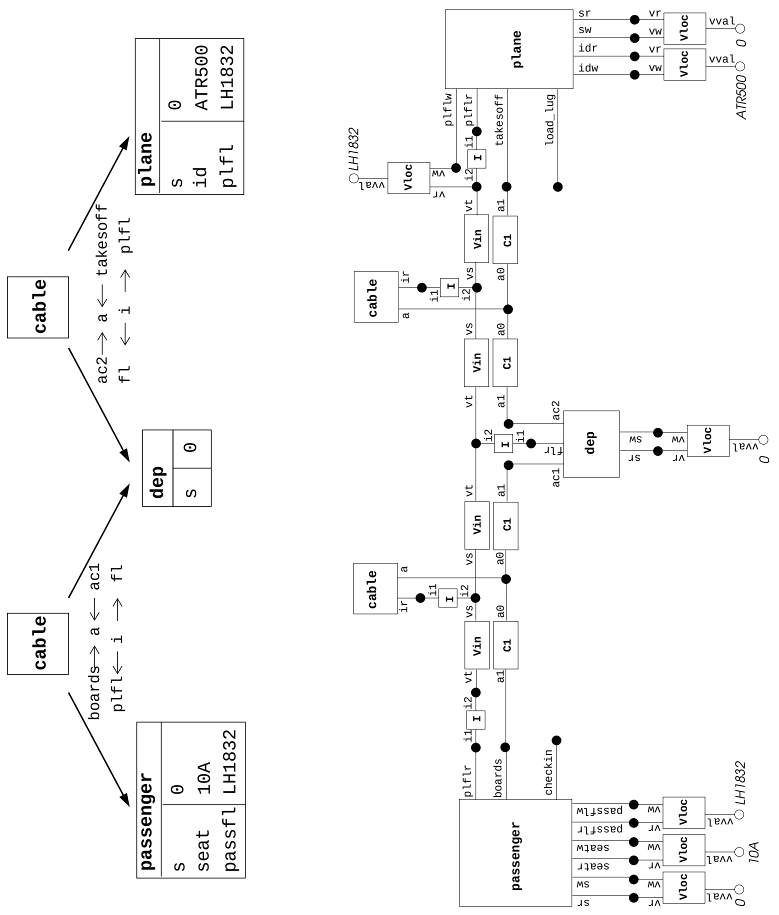


Fig. 6. An anchored configuration (left) and its graphical representation (right).

annotating the lw-node, for each channel $l \in D(a)$.

Figure 9 shows the rules for auxiliary edges. The rules for OR boxes force simultaneous execution of the action connected to the a0-node with exactly one of the actions connected to the other nodes. The rules for local boxes allow to read and/or to modify (using events **get** and **set**) a channel value, represented by the attributed vval-node. The rules for input and interface boxes forward a channel value to make it available at a shared input channel.

Figure 10 shows the application of a parallel synchronized rule to the configuration of Figure 6. The left part shows the start graph where synchronization nodes (drawn as gray hexagons) are annotated with the resulting event (edges taking part to the rewriting are marked as double boxes). The applied

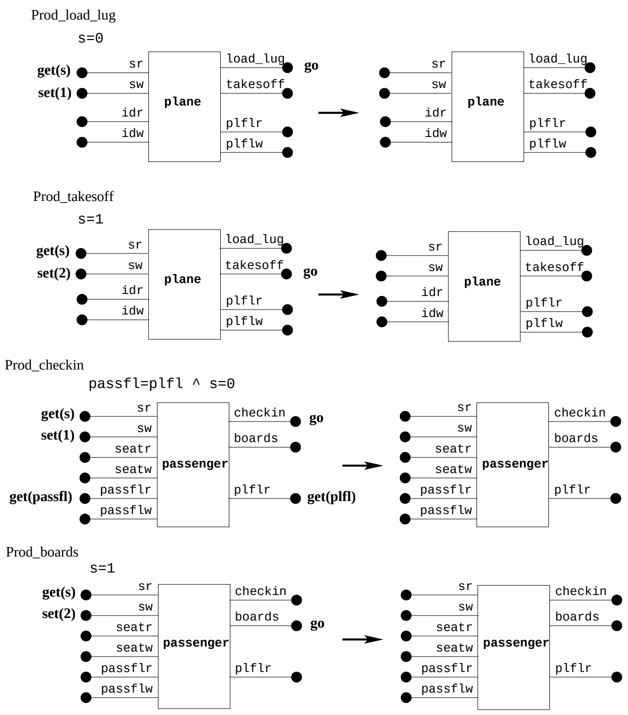


Fig. 7. Rules for actions of `passenger` and `plane`.

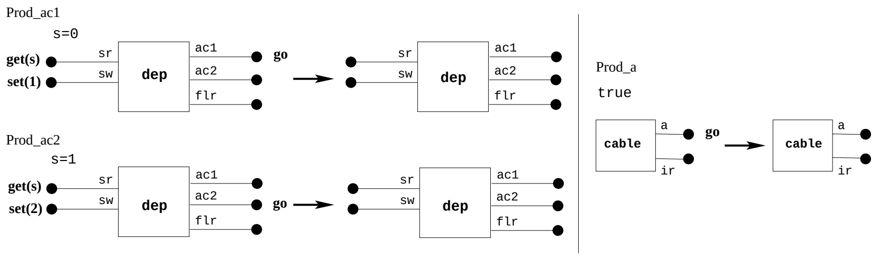


Fig. 8. Rules for designs `dep` and `cable`.

rules are those for actions `boards` and `ac1`, and four for auxiliary edges.

Even if we did not elaborate a formal proof, a careful inspection of the constructions we presented makes us confident that the following result holds.

**Claim 5.1 (Correctness and completeness of the encoding)** *For an anchored* CommUnity *configuration* $C$*, let* $\mathcal{G}(C)$ *be its graphical representation as for Section 4, and let* $\mathcal{R}$ *be the Synchronized GTS including all rules for the designs in* $C$ *and all rules for auxiliary edges. Then* $\mathcal{G}(C) \Rightarrow_{\mathcal{R}^{\oplus}} H$ *using a parallel rule which contains the rules encoding the actions* $\{a_1, \ldots, a_k\}$ *for* $k > 0$*,*

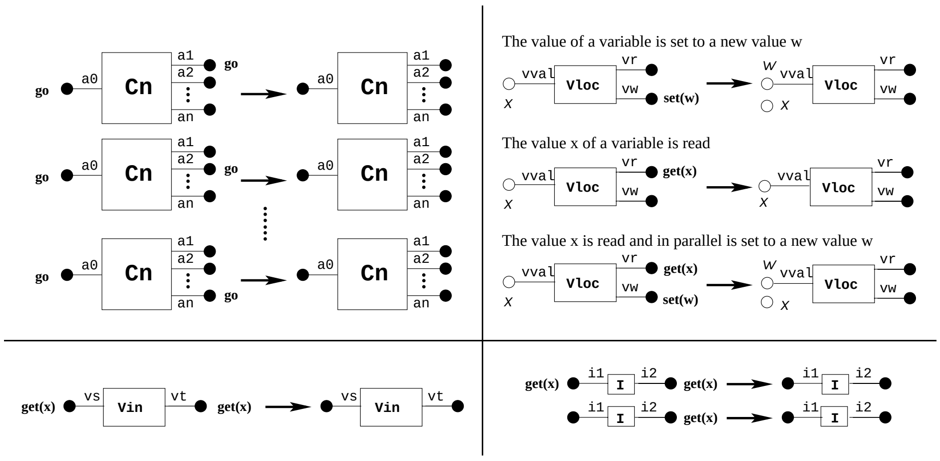

The value of a variable is set to a new value w
The value x of a variable is read
The value x is read and in parallel is set to a new value w
go
a0
a1
a2
an
Cn
Vloc
vval
vr
vw
set(w)
get(x)
Vin
vs
vt
I
i1
i2

Fig. 9. Rules for auxiliary boxes.

*if and only if in the colimit design of $C$ the synchronized action $\langle a_1, \ldots, a_k \rangle$ is enabled to fire. Furthermore, if $D$ is the anchored design resulting after the firing of the synchronized action, then $D$ is a colimit of an anchored configuration $C'$ such that $\mathcal{G}(C')$ is isomorphic to $H$.*

## 6 Conclusion and Future Work

We proposed a quite direct and intuitive graphical representation of CommUnity configurations, and their operational semantics using synchronized hyperedge rewriting. One advantage of the proposed encoding is that superposition morphisms are explicitly represented using auxiliary edges. Some preliminary investigations suggest that this makes possible to model general *reconfigurations* as attributed graph transformation rules, possibly with Negative Application Conditions [5]. We intend to develop this further, and to study the relationships with [9], where reconfiguration is specified as conditional double-pushout rules over configuration diagrams.

## Acknowledgement

We thank José Fiadeiro, Antonia Lopes and Michel Wermelinger for their feedback about CommUnity.

## References

[1] AGILE Project IST-2001-32747. The airport case study. Deliverable 4.1, 2003.

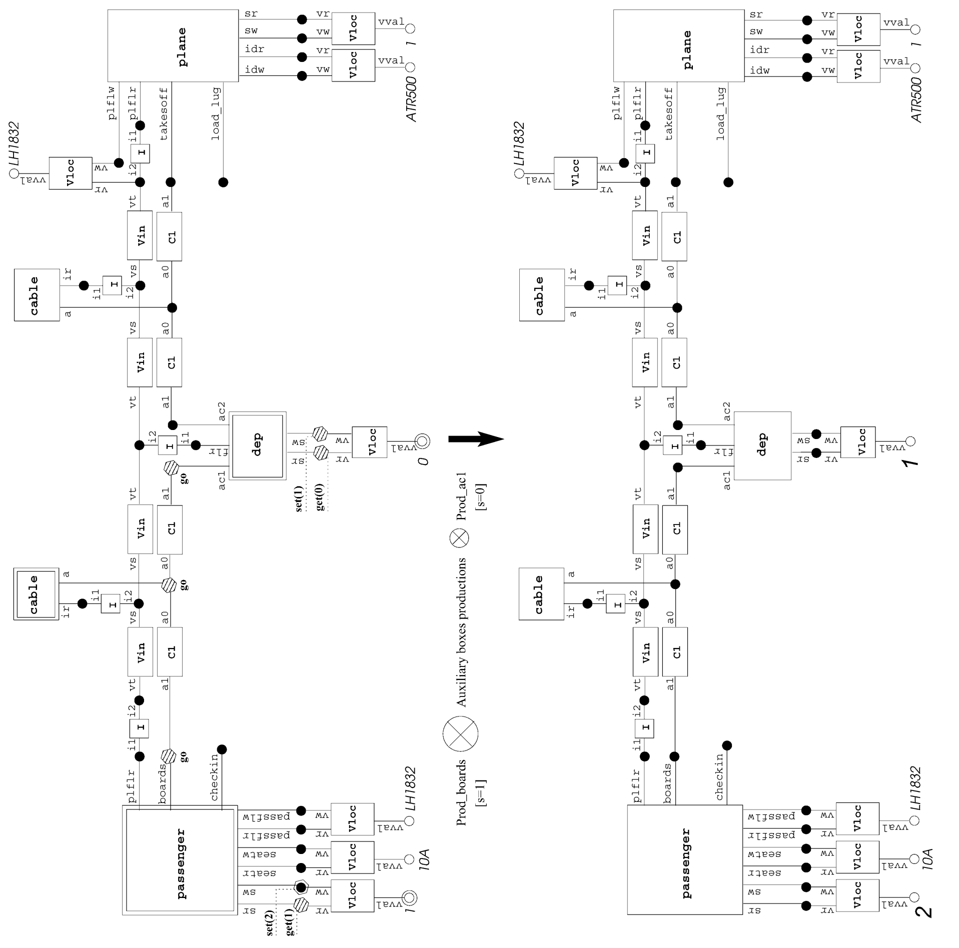


Fig. 10. Rewriting Step for Colimit action *boardsAc1*

[2] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In [8], chapter 3. World Scientific, 1997.

[3] F. Drewes, H.-J. Kreowski, and A. Habel. Hyperedge replacement graph grammars. In [8], chapter 2. World Scientific, 1997.

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

[5] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3–4):287–313, 1996.

[6] D. Hirsch. *Graph Transformation Models for Software Architecture Styles.* PhD thesis, Dept. of Computer Science, Universidad de Buenos Aires, May 2003.

[7] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley, New York, 1993.

[8] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*, volume 1. World Scientific, 1997.

[9] M. Wermelinger and J. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44:133–155, 2002.