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

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

An Operational Semantics of COMMUNITY Based on Graph Transformation Systems [1](#_bookmark1)

# Andrea Corradini Dan Hirsch

*Dipartimento di Informatica, Universita` di Pisa, Via F. Buonarroti 2, I-56127, Pisa, Italia* [*2*](#_bookmark1)

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 syn- chronized 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.

# Introduction

Architectural approaches to software design view system architectures as struc- tured in terms of components (where actual computations take place) glued together by connectors, which prescribe how components interact. CommUnity

[[4](#_bookmark17)] 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](mailto:andrea@di.unipi.it) , [dhirsch@di.unipi.it](mailto:dhirsch@di.unipi.it)

1571-0661 © 2004 Elsevier B.V. Open access under [CC BY-NC-ND license](http://creativecommons.org/licenses/by-nc-nd/3.0/).

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 deter- mine 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 de- sign* for the configuration, having one action for each possible action synchro- nization 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 intu- itive, inductive and complete account of the operational behaviour of CommU- nity 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 re- placement 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.

1. 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](#_bookmark17),[9](#_bookmark24)], where the reader can find the original formulation and more detailed explanations.

The syntax of a CommUnity design is shown in Figure [1](#_bookmark0). 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* ) ∪ *prv*(*V* ). Design *D* also has a set of *actions* Γ, partitioned into *shared* (*sh*(Γ)) and *private actions* (*prv*(Γ)). For each action *a* ∈ Γ, *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

design *D*

in *in(V)*

out *out(V)*

prv *prv(V)*

do [] *a* : *G*(*a*) →  *l* := *E*(*a, l*)

|  |  |  |  |
| --- | --- | --- | --- |
|  | *a* ∈ *sh*(Γ) | *l* ∈ *D*(*a*) |  |
| [] | []  *a* ∈ *prv*(Γ) | prv *a* : *G*(*a*) →  *l* ∈ *D*(*a*) | *l* := *E*(*a, l*) |

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 DΣ, where Σ is a standard algebraic signature. [3](#_bookmark3) Two sample designs (passenger and plane)

are shown in the left part of Figure [2](#_bookmark2): they are borrowed (with little modifica- tions) from an example in [[1](#_bookmark16)], 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**

**Coordination Contract Types**

**Coord−contract** departure(passenger,plane) **is**

**prv** s: [0..2],

seat: SeatId, passfl: Flight

**design** cable **is in** i

**design** cable **is in** i

**in** plfl: Flight

**do** checkin: [passfl=plfl ^ s=0

**[]** boards: [s=1 s:=2]

s:=1]

**do** a: [true  skip]

**do** a: [true  skip]

**design** plane **is prv** s: [0..2],

id: PlId

 a i

**passenger**

ac1

fl

ac2

fl

boards plfl

**design** dep **is prv** s: [0..2]

a  takesoff

i  plfl

**plane**

**out** plfl: Flight

**in** fl

**do** load\_lug: [s=0

**[]** takesoff: [s=1

s:=1]

s:=2]

**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 σ* : *C* → *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 *σV* (*v*) of *D*, and each (shared) action *a* of *C*

to a non-empty set of non-private (i.e., shared) actions *σ*Γ(*a*) = {*a*' *,... , a*' }

1 *n*

3 Typically, the channels in *V* are typed and a standard typing discipline is imposed over their manipulation. In this case Σ 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](#_bookmark5) Intuitively, such morphism identifies the channels *v* and *σV* (*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 *σ*Γ(*a*) (*a* can fire if and only if one of the actions in *σ*Γ(*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 con- straint 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](#_bookmark2), where the designs passenger and plane are connected via a third design dep and two cables. The morphisms cause the input chan- nel 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

**[]** boardsAc1Load\_lug: [spass=1 ^ sdep=0 ^ spl=0

spass:=1 || spl:=1] 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](#_bookmark2)

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](#_bookmark2) (shown in Figure [3](#_bookmark4)), 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](#_bookmark12), because their guards are always false.

4 *σ*Γ can also be defined as a surjective, partial function from actions of *D* to those of *C*.

# 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) hy- pergraph rewriting according to the *double-pushout approach* [[2](#_bookmark18),[3](#_bookmark19)]. Next we enrich the formalism first with attributions [[7](#_bookmark23)], and then with synchronization

[[6](#_bookmark22)] and with application conditions.

* 1. *Hypergraph rewriting*

Hypergraphs are made of nodes and labeled edges. Each edge may be con- nected 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* ∆ = ⟨*CN ,* Λ*, rnk* ⟩, where *CN* is a set of *connection names*, Λ is a set of *edge labels*, and *rnk* : Λ → P*fin* (*CN* ) is a function associating to each edge label a finite set of connection names.

A ∆*-(hyper)graph H* is a tuple ⟨*VH, EH , labH, cH*⟩, where *VH* is a set of nodes, *EH* is a set of (hyper)edges, *labH* : *EH* → Λ is the labeling function, and *cH* : *EH* → [*CN* →◦ *VH* ] is the connection function, associating with each edge a partial function from connection names to nodes. It is required that *dom*(*cH*(*e*)) = *rnk* (*labH*(*e*)) for each *e* ∈ *EH* . If *cH*(*e*)(*x*) = *v* we call *v* the *x*-node of *e*. Hypergraph morphisms are defined in the expected way.

*l r*

A *rule* is a span of *injective* ∆-graph morphisms *q* = (*L* ←*› K ‹*→ *R*),

where *L*, *K*, *R* are finite ∆-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 ∆-graph

*G* is a morphism *g* : *L* → *G* which is injective on edges. [5](#_bookmark6) In this case we write

*G* ⇒*q H*, if both squares in the following diagram are pushouts:

*L* ,*rl* r*K*z,˛*r* z*R,*

*g*

,z,,˛,

*G* ,*r* r*D* z*H,*

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

A ∆*-graph transformation system (GTS)* R = ⟨*P, π*⟩ is made of a set of rule names *P* and of a function *π* which assigns to each *p* ∈ *P* a rule *π*(*p*) = (*Lp* ←*› Kp ‹*→ *Rp*). We write *G* ⇒R *H* if *G* ⇒*p H* for some *p* ∈ *P* . Given a GTS R = ⟨*P, π*⟩, let R⊕ = ⟨*P* ⊕*, π*⊕⟩ be the GTS where *P* ⊕ is the (carrier of the) free commutative monoid generated by *P* (excluding unit), and *π*⊕ is the free extension of *π*, mapping *p*⊕*p*' to the disjoint union of rules *π*⊕(*p*) and *π*⊕(*p*'). [6](#_bookmark7)

Then we say that there is a *parallel rewriting* from *G* to *H* in R iff *G* ⇒R⊕ *H*.

* 1. *Attributed hypergraph rewriting*

Let Σ be a standard algebraic *one-sorted* signature, which is fixed for the rest of the section. [7](#_bookmark8) A Σ*-attributed* ∆*-hypergraph* (or Σ*,*∆*-graph*) is a triple *G* = ⟨|*G*|*,* A*G, att G*⟩, where |*G*| is a ∆-graph (the *underlying* ∆*-graph* of *G*), A*G* is a Σ-algebra, and *att G* : *V*|*G*| →◦ |A*G*| is an *attribution*, i.e., a partial function mapping nodes of |*G*| to elements of the carrier of the algebra A*G*. A Σ*,*∆*-graph morphism f* : *G* → *G*' is a pair ⟨|*f* |*, fA*⟩, where |*f* | : |*G*| → |*G*'| is a ∆-graph morphism, and *fA* : A*G* → A*G*' is a total Σ-homomorphism such that *fA* ◦ *att G* = *att G*' ◦ |*f* |*V* .

This definition of attributed graph is very general, because every graph is equipped with its own Σ-algebra. We stick here to the case where the attributed graphs appearing in a rule have associated *T*Σ(*X*), the free algebra

over Σ generated by a countable set of variables *X*, while the graphs which are rewritten all have as algebra a fixed *data algebra* DΣ. An *attributed rule*

*l r*

is a span of *injective* Σ,∆-graph morphisms *r* = (*L* ←*› K ‹*→ *R*), such that:

|*l*| |*r*|

* the underlying span of ∆-graph morphisms (|*L*| ←*›* |*K*| *‹*→ |*R*|) is a rule;
* the Σ-algebra associated with *L*, *K*, *R* is *T*Σ(*X*), and *lA, rA* 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;
* *V ar*(*R*) ⊆ *V ar*(*L*), where *V ar*(*H*) ⊆ *X* is the set of variables appearing in the attributions of *H*.

A *match* of an attributed rule *r* = (*L* ←*› K ‹*→ *R*) in a Σ,∆-graph *G* over DΣ is a Σ,∆-graph morphism *g* : *L* → *G* which is injective on edges. Such a map determines an *evaluation θg* : *V ar*(*L*) → DΣ defined as *θg*(*x*) = *att G*(*gV* (*n*)) iff *att L*(*n*) = *x* (this is well defined thanks to linearity of *L*). By the freeness of *T*Σ(*X*), *θg* extends uniquely to a homomorphism *θg* : *T*Σ(*X*) → DΣ: 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 *rθg* = (*Lθg* ←*› Kθg ‹*→ *Rθg*), where graphs are over DΣ. In this situation we write *G* ⇒*r H* if *H* = ⟨|*H*|*,* DΣ*, att H* ⟩, there is a

∆-rewriting step |*G*| ⇒|*rθg* | |*H*|, and *att H* is uniquely determined by lifting

the double-pushout diagram to the category of Σ,∆-graphs, assuming that all Σ-homomorphism components are identities over DΣ.

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.

* 1. *Conditional Synchronized Hypergraph Rewriting*

We introduce now a powerful synchronization mechanism for attributed rules, which in the terminology of [[6](#_bookmark22)] is called *Hoare synchronization with value pass- ing*. 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 Σ. These events may prevent the application of the rule at a given match, unless one or more other rules, annotated with corresponding match- ing 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* = ∪*n*∈N*Evn* be a fixed, ranked set of *event names*, and EΣ(*X*) be the collection of *well-formed events over X*, containing all terms of the form *e*(*t*1*,... , tn*), with *e* ∈ *Evn* and *ti* ∈ *T*Σ(*X*) for all *i* ∈ {1*,... , n*}. A (*con- ditional* ) *synchronized* (*attributed* ) *rule* is a triple ⟨*r, synch, cond*⟩, namely an attributed rule *r* = (*L* ←*› K ‹*→ *R*) equipped with a partial function *synch* : *V*|*L*| →◦ EΣ(*X*), and with a formula *cond* in a logic sufficiently ex- pressive to encode CommUnity guards. Denoting by *V ar*(*synch*) the set of

variables appearing in the events in *synch*(*V*|*L*|), the following condition must hold: *V ar*(*R*) ∪ *V ar*(*cond*) ⊆ *V ar*(*L*) ∪ *V ar*(*synch*). This condition will guar- antee 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* Σ*,*∆*-hyper)graph transformation sys- tem (SGTS)* S = ⟨*P, π*⟩ is made of a set of rule names *P* and of a function *π* mapping each *p* ∈ *P* to a synchronized rule *π*(*p*) = ⟨*rp, synchp, condp*⟩. Then, the parallel SGTS S⊕ = ⟨*P* ⊕*, π*⊕⟩ has *P* ⊕, the free commutative monoid gener- ated by *P* (excluding unit) as rule names. Moreover, if *p* = *p*1 ⊕ *...* ⊕ *pk* ∈ *P* ⊕, with *pi* ∈ *P* and *π*(*pi*) = ⟨*ri, synchi, condi*⟩ for all *i* = 1*,... , k*, then the par-

allel synchronized rule *π*⊕(*p*) = ⟨*r, synch, cond*⟩ 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 *ri*’s, as *synch* the obvious partial function induced by all *synchi*’s, and as *cond* the *conjunction* of all *condi*’s.

Let *s* = ⟨*r* = (*L* ←*› K ‹*→ *R*)*, synch, cond*⟩ be a (typically parallel) syn- chronized rule, and let *g* : *L* → *G* be a match to a Σ,∆-graph *G*. Further-

*def*

more, let *θg* : *V ar*(*L*) → DΣ be the evaluation induced by *g*, and *synch*(*g*) =

*g*(*dom*(*synch*)) ⊆ *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*;
* ∀*n* ∈ *synch*(*g*), *g*−1(*n*) ⊆ *dom*(*synch*), i.e., each node mapped by *g* to a synchronization node is annotated with an event;
* there exists a ground substitution *θ* : *V ar*(*synchθg*) → DΣ such that for each node *n* ∈ *synch*(*g*) and for all nodes *m, m*' ∈ *g*−1(*n*), *synch*(*m*)*θgθ* =

*synch*(*m*')*θgθ*; furthermore *θ* must be minimal, in the sense that it must be a most general unifier of the involved partitioned set of events;

* the formula *condθgθ* evaluates to true.

Under these conditions, we write *G* ⇒*s H* if *G* ⇒*r*' *H* using the match *g*, where *r*' is the attributed rule *r*' = (*L* ←*› K ‹*→ *Rθ*).

# 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 configura- tion. The graph representing a configuration includes one edge for each design,

and *auxiliary edges* for encoding the anchorings and the morphisms. Figure [4](#_bookmark10)

shows the graphs encoding two anchored designs, while Figure [5](#_bookmark11) shows the graphical representation of a simple CommUnity morphism.

vr

vw

**Vloc**

**Vloc**

plflr i1 **I** i2 vr plflw vw

**Vloc**

idw

**plane**

vval

vw

*0*

vr

vval

sr sw

idr

load\_lug takesoff

vval

*ATR500 LH1832*

*0*

*10A*

vr

vval

passflw

vw

**passenger**

sw

vw

passflr

seatw

vw

plflr i1 **I**

**Vloc**

**Vloc**

**Vloc**

vr

vr

vval

vval

sr

seatr

checkin

boards

i2

*LH1832*

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

sa sa’1

D sa

v1r

a0

**Cn**

a1 sa’1

a2 sa’2

D’

an

i1

**I**

i2 vs

sa’n

v2r

i1

**I**

vt i2

**Vin**

sa sa’n

D v1 v2 D’

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](#_bookmark11), a morphism *σ* : *D* → *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 *σ*Γ(sa). Figure [6](#_bookmark12) shows a complete anchored configuration, and its graphical representation.

# 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 mecha- nism. Three event names are used: go for synchronizing actions, and get and set for reading and changing the channel values.

Figures [7](#_bookmark13) and [8](#_bookmark14) show the rules for all the actions of the designs of the running example. In general, the rule encoding action *a* : *G*(*a*) →  *l*∈*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))

**cable**

**cable**

a i

ac1 fl

ac2 fl

a i

takesoff plfl

**plane**

s id

plfl

**dep**

s

0

ATR500 LH1832

0

*LH1832*

**cable**

**cable**

ir a

a ir

**Vloc**

i1

**I**

i1

**I**

plflw

i2 i1 plflr

**I**

takesoff

vs i2

i2 vs

vs

vt

vt

vs

vt

**Vin**

**Vin**

**Vin**

**Vin**

i2

**I**

i1

a0

a0

a1

a1

a0

a0

a1

**C1**

**C1**

**C1**

**C1**

**plane**

load\_lug

ac1

ac2

**dep**

**Vloc Vloc**

**Vloc**

*ATR500* *0*

*0*

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



sr sw idr

idw

vr vw vr

vw

vval

vval

boards plfl

**passenger**

0

10A

s seat

passfl LH1832

plflr

i1

i2 vt

**I**

boards

a1

**passenger**

checkin

**Vloc Vloc Vloc**

*0*

*LH1832*

*10A*

annotating the lw-node, for each channel *l* ∈ *D*(*a*).

Figure [9](#_bookmark15) 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](#_bookmark20) shows the application of a parallel synchronized rule to the con- figuration of Figure [6](#_bookmark12). The left part shows the start graph where synchroniza- tion nodes (drawn as gray hexagons) are annotated with the resulting event (edges taking part to the rewriting are marked as double boxes). The applied

Prod\_load\_lug

s=0

**plane**

**get(s)**

**set(1)**

sr sw

idr idw

load\_lug **go**

takesoff

plflr plflw

sr sw

idr idw

**plane**

load\_lug takesoff

plflr plflw

Prod\_takesoff

s=1

**plane**

**get(s)**

**set(2)**

sr sw

idr idw

load\_lug takesoff **go**

plflr plflw

sr sw

idr idw

**plane**

load\_lug takesoff

plflr plflw

Prod\_checkin

**get(s)**

**set(1)**

**get(passfl)**

passfl=plfl ^ s=0

**go**

|  |  |  |
| --- | --- | --- |
| sr | **passenger** | checkin |
| sw | boards |
| seatr | plflr |
| seatw |
| passflr |
| passflw |  |

|  |  |  |
| --- | --- | --- |
| sr | **passenger** | checkin |
| sw | boards |
| seatr | plflr |
| seatw |
| passflr |
| passflw |  |

**get(plfl)**

Prod\_boards

**get(s)**

**set(2)**

s=1

**passenger**

sr sw seatr seatw

**passenger**

checkin boards **go**

sr sw seatr seatw

checkin boards

passflr passflw

plflr

passflr passflw

plflr

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

Prod\_ac1

s=0

**dep**

**get(s)**  sr

**set(1)**  sw

Prod\_ac2

s=1

ac1

**dep**  ac2 flr

**get(s)**  sr

**set(2)**  sw

ac1 **go**

ac2

flr

 **go**

sr

sw

**dep**

sr

ac1

**dep**  ac2 flr

sw

ac1

ac2

flr

Prod\_a

true

**go**

a

**cable**

ir

a

**cable**

ir

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 an- chored* CommUnity *conﬁguration C, let* G(*C*) *be its graphical representation as for Section* [*4*](#_bookmark9)*, and let* R *be the Synchronized GTS including all rules for the designs in C and all rules for auxiliary edges. Then* G(*C*) ⇒R⊕ *H using a par- allel rule which contains the rules encoding the actions* {*a*1*,... , ak*} *for k >* 0*,*

**go**

vval

vval

*x*

**go**

**set(w)**

vr

vw

*w* vval

vval

vw

a0

The value x of a variable is read

vr

**go**

**get(x)**

**Vloc**

**I**

vs

**Vloc**

**get(x)**

**Vin**

**go**

vs

**Vloc**

**go**

a0

**set(w)**

vval

**Vloc**

*w*

**Vloc**

a0

a0

vr

vw

a0

**get(x)**

*x*

**I**

**go**

*x*

**get(x)**  i1 **I**  i2 **get(x)**

i1 i2

vt

a0

The value of a variable is set to a new value w

vr

**get(x)**

vw

vr

vw

i1 i2

**I**

*x*

vt

*x*

**get(x)**

vval

i1 i2

vr

vw

The value x is read and in parallel is set to a new value w

**Vin**

Fig. 9. Rules for auxiliary boxes.

**Cn**

a1 a2

an

**Cn**

a1

a2

an

**Cn**

a1

a2

an

**Cn**

a1

a2

an

**Vloc**

*x*

*if and only if in the colimit design of C the synchronized action* ⟨*a*1*,... , ak*⟩ *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 conﬁgu- ration C*' *such that* G(*C*') *is isomorphic to H.*

# Conclusion and Future Work

We proposed a quite direct and intuitive graphical representation of

Com-

mUnity configurations, and their operational semantics using synchronized hyperedge rewriting. One advantage of the proposed encoding is that super- position morphisms are explicitly represented using auxiliary edges. Some preliminary investigations suggest that this makes possible to model general

**Cn**

a1

a2

an

**Cn**

a1

a2

an

*reconﬁgurations* as attributed graph transformation rules, possibly with Neg-

ative Application Conditions [[5](#_bookmark21)]. We intend to develop this further, and to study the relationships with [[9](#_bookmark24)], where reconfiguration is specified as condi- tional double-pushout rules over configuration diagrams.

# Acknowledgement

We thank Jos´e 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.

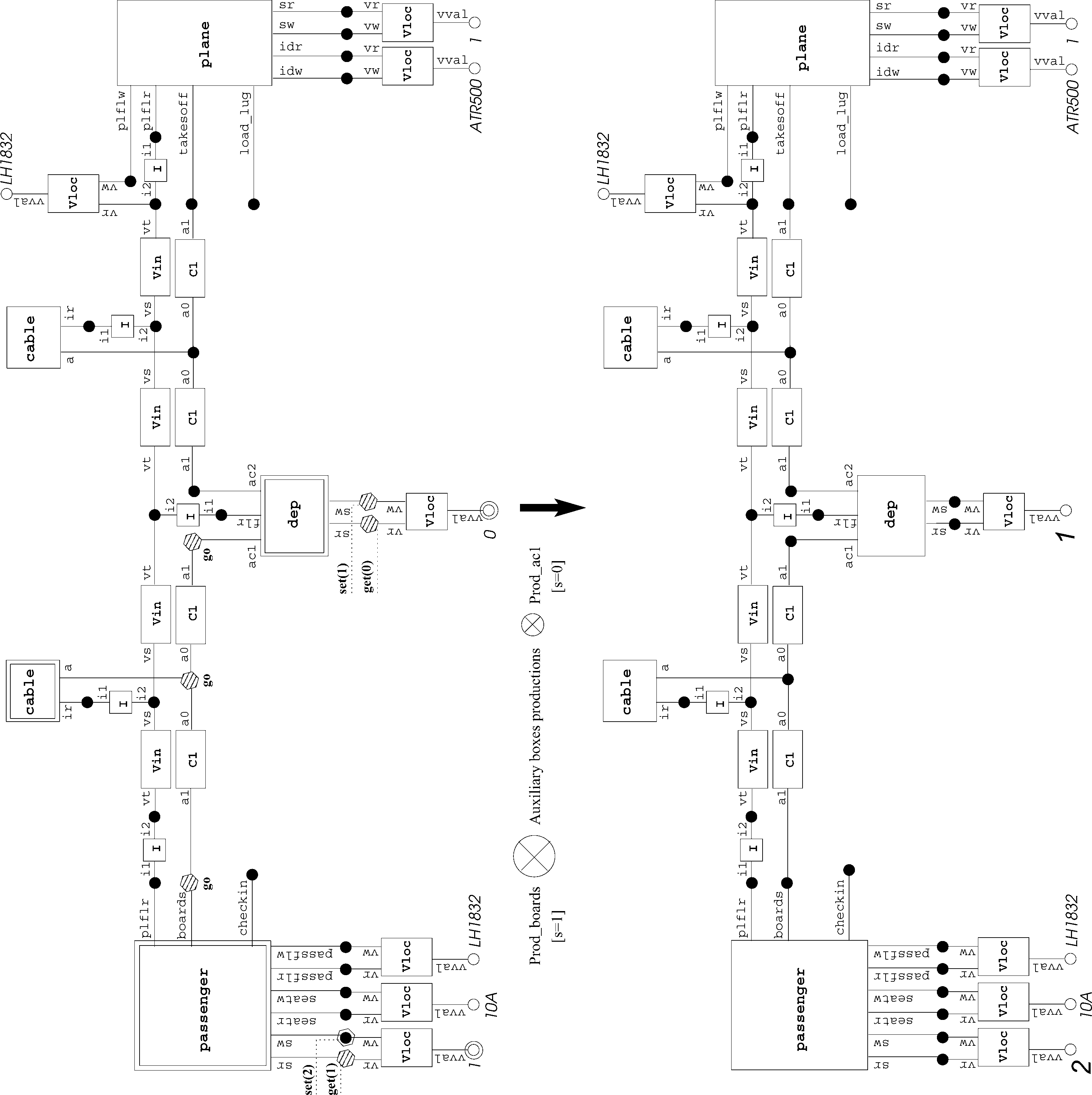


Fig. 10. Rewriting Step for Colimit action *boardsAc1*

1. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. L¨owe. Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In *[*[*8*](#_bookmark25)*]*, chapter 3. World Scientific, 1997.
2. F. Drewes, H.-J. Kreowski, and A. Habel. Hyperedge replacement graph grammars. In *[*[*8*](#_bookmark25)*]*, chapter 2. World Scientific, 1997.
3. J. Fiadeiro and T. Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28:111–138, 1997.
4. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions.

*Fundamenta Informaticae*, 26(3–4):287–313, 1996.

1. D. Hirsch. *Graph Transformation Models for Software Architecture Styles*. PhD thesis, Dept. of Computer Science, Universidad de Buenos Aires, May 2003.
2. M. L¨owe, 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.
3. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Foundations*, volume 1. World Scientific, 1997.
4. M. Wermelinger and J. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44:133–155, 2002.