A Graph Transformation Approach to Architecture-Based Reconfiguration

(work in progress)

Michel Wermelinger

joint work with José Luiz Fiadeiro and Antónia Lopes

Laboratório de Modelos e Arquitecturas Computacionais
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1700 Lisboa, Portugal

CommUnity

- Introduction: Motivation, Case Study
- Programs: Syntax, Semantics
- Superposition
- Configurations
- CommUnity with State

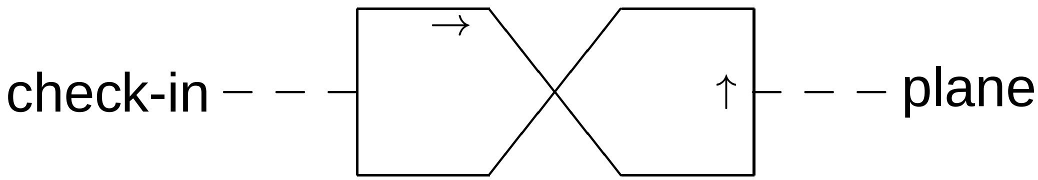
Introduction

Motivation

- developed by José Fiadeiro, Tom Maibaum (1995)
- action-based version of UNITY
- shows how programs fit into Goguen's categorical approach to General Systems Theory
- formal platform for architectural design of open, reactive, reconfigurable systems

Case Study

- luggage distribution system inspired by Mobile Unity



- track with $U > 30$ segments with segments 7 and 28 crossing
- carts move in given direction and start in distinct positions
- there are bag (un)loaders along the track
- an unloader is connected to a check-in counter, a loader to a plane

Case Study

- each cart carries a bag from an unloader to a loader
- avoid collisions
  - of a moving cart and a stopped cart (un)loading a bag
  - of two carts approaching the crossing

Programs

Syntax

$$
\begin{array}{ll}
\texttt{prog} & P \\
\texttt{in} & I \\
\texttt{out} & O \\
\texttt{init} & ic \\
\texttt{do} & \mathop{[]}_{a \in A} \; a\text{: } G(a) \rightarrow \; \mathop{\|}_{o \in D(a)} \; o :\in E(a, o)
\end{array}
$$

- ic is condition over O; guards G are conditions over $O \cup I$
- input variables are read-only, i.e., $D(a) \subseteq O$
- $E(a, o)$ is a set of terms of the same sort as o

Example

`prog` Cart

`in`	idest : int
`out`	loc, odest : int
`init`	$0 \leq$ loc $< U \wedge$ odest = -1
`do`	move: loc $\neq$ odest $\rightarrow$ loc := (loc + 1) mod U
[]	get: odest = -1 $\rightarrow$ odest := idest

henceforth “loc $+_U$ 1”

Example

`prog` Check_In

`out`	loc, dest : int; next : bool
`init`	$0 \leq$ loc $< U \wedge$ next
`do`	new: next $\rightarrow$ dest $:\in$ int $\|$ next := false
[]	put: $\neg$next $\rightarrow$ next := true

Semantics

in the values of I are given by the environment and may change at each step

init the initial values of O are chosen non-deterministically satisfying ic

[] at each step choose one action randomly

$\boldsymbol{a}$: $\boldsymbol{g} \rightarrow$ **...** if g is true, execute the action

$\boldsymbol{o} :\in \boldsymbol{E}(\boldsymbol{a}, \boldsymbol{o})$ non-deterministic assignment of a set element to o

$\displaystyle\Big\|_{\boldsymbol{o} \in \boldsymbol{D}(\boldsymbol{a})}$ **...** evaluate the set expressions and then assign

$\boldsymbol{D}(\boldsymbol{a})$ if a is executed, the values of $o \notin D(a)$ do not change

Notation

- `skip` is the empty command (when $D(a) = \emptyset$)

$$a\text{: } G(a) \rightarrow \texttt{skip}$$

- := is the deterministic assignment

$$\text{c := c + 1} \quad \equiv \quad \text{c} :\in \{\text{c + 1}\}$$

- `true` guards are omitted

$$\text{inc: c := c + 1} \quad \equiv \quad \text{inc: true} \rightarrow \text{c := c + 1}$$

- domain of output variable: $D(o) = \{a \in A \mid o \in D(a)\}$
- variables: $V = O \cup I$
- the initialization condition is omitted when tautology

Superposition

morphism $\sigma : P \to P'$ given by two functions

- $\sigma : V \to V'$ is total and preserves sorts
- $\sigma : A' \rightharpoonup A$ is partial

s.t.

- output variables do not become input variables: $\sigma(O) \subseteq O'$
- action and variable domains are preserved:
  $v \in D(\sigma(a')) \Rightarrow \sigma(v) \in D'(a')$ and $a' \in D'(\sigma(v)) \Rightarrow \sigma(a') \in D(v)$
- guards may be strengthened: $G'(a') \Rightarrow G(\sigma(a'))$
- the initialization condition may be strengthened: $ic' \Rightarrow \sigma(ic)$
- assignments more deterministic: $E'(a', \sigma(o)) \subseteq \sigma(E(\sigma(a'), o))$

Example

Cart

lap$\mapsto$move, move$\mapsto$move, get$\mapsto$get $\downarrow$ idest$\mapsto$idest, odest$\mapsto$odest, loc$\mapsto$l

Cart_Stat

`prog` Cart_Stat

`in` idest : int

`out` l, odest, sl, laps : int

`init` 0 $\leq$ l $<$ U $\wedge$ odest = -1 $\wedge$ sl = l $\wedge$ laps = 0

`do` move: l $\neq$ odest $\wedge$ l $+_U$ 1 $\neq$ sl $\rightarrow$ l := l $+_U$ 1

[] lap: l $\neq$ odest $\wedge$ l $+_U$ 1 = sl $\rightarrow$ l := l $+_U$ 1 $\|$ laps := laps + 1

[] get: odest = -1 $\rightarrow$ odest := idest

Notation

- action mapping is given in same direction as morphism, using set notation when necessary

```
prog P                        prog P'
                x↦{y,z}
do  x: skip     ------->      do  y: skip     ⇒ y ↦ x ∧ z ↦ x
                              []  z: skip
```

$$\texttt{prog}\ P \quad \texttt{do}\ \ x\texttt{: skip} \xrightarrow{x \mapsto \{y,z\}} \texttt{prog}\ P' \quad \texttt{do}\ \ y\texttt{: skip} \ \ [\!]\ \ z\texttt{: skip} \quad \Rightarrow y \mapsto x \wedge z \mapsto x$$

- morphisms are implicitly given through common names

$$\texttt{prog}\ P \quad \texttt{do}\ \ x\texttt{: skip} \longrightarrow \texttt{prog}\ P' \quad \texttt{do}\ \ x|y\texttt{: skip} \quad \Rightarrow x \mapsto x|y$$

$$\texttt{prog}\ P \quad \texttt{do}\ \ a\texttt{: skip} \longrightarrow \texttt{prog}\ P' \quad \texttt{do}\ \ a_1\texttt{: skip} \ \ [\!]\ \ a_2\texttt{: skip} \quad \Rightarrow a \mapsto \{a_1, a_2\}$$

Configurations

- diagrams in the category of programs and superposition morphisms
- basic interactions through identification of variables (sharing) and actions (synchronization)
- output variables may not be shared

System

system is given by colimit of configuration of programs P_i:

output variables disjoint union modulo identified variables

input variables disjoint union except those identified with output variables

initialization condition conjunction

actions all tuples $a_1|a_2|\ldots$ with at most one action from each program and obeying synchronization constraints

action guards conjunction of $G(a_k)$

action bodies disjoint union of parallel assignments

Example

Cart $\xleftarrow[\text{get} \hookleftarrow \text{a}]{\text{idest} \hookleftarrow \text{i}}$ `prog` Load $\xrightarrow[\text{a} \mapsto \text{put}]{\text{i} \mapsto \text{dest}}$ Check_In

`in` i : int

`do` a: true $\rightarrow$ `skip`

henceforth Load $\equiv \langle$i : int $|$ a$\rangle$

`prog` System

`out` loc_1, odest, loc_2, dest : int; next : bool

`init` $0 \leq \text{loc}_1, \text{loc}_2 < U \wedge \text{odest} = \text{-1} \wedge \text{next}$

`do` move: $\text{loc}_1 \neq \text{odest} \rightarrow \text{loc}_1 := \text{loc}_1 +_U 1$

[] send: $\text{odest} = \text{-1} \wedge \neg\text{next} \rightarrow \text{odest} := \text{dest} \parallel \text{next} := \text{true}$

[] new: $\text{next} \rightarrow \text{dest} :\in \text{int} \parallel \text{next} := \text{false}$

[] move|new: $\text{loc}_1 \neq \text{odest} \wedge \text{next}$

$\rightarrow \text{loc}_1 := \text{loc}_1 +_U 1 \parallel \text{dest} :\in \text{int} \parallel \text{next} := \text{false}$

CommUnity with State

Definitions

logical variables typed, to denote state: LV = $\{x, y$: int; b, b' : bool$\}$

programs with valuation $\epsilon : O \rightarrow Terms(LV)$

- non ground terms in the reconfiguration rules
- terms only for variables controlled by the program

morphisms preserve state: $\epsilon(o) = \epsilon'(\sigma(o))$

Example

cart that completed at least one round and is about to finish another one

Cart	
loc	$-1 + x + 1$
odest	-1

$\xrightarrow[\text{move} \mapsto \{\text{move}, \text{lap}\}]{\text{loc} \mapsto \text{l}}$

Cart_Stat	
l	x
odest	-1
sl	$x +_U 1$
laps	$y + 1$

Connectors

- Introduction: Motivation, Characterization, Definition
- Catalog:
  - (Partial) Synchronization
  - (Conditional) Inhibition

Introduction

Motivation

- encapsulate interactions between components of a system
- allow separation between computation and coordination
- facilitate reconfiguration

Characterization

connector glue + at least a role

glue specifies an interaction

role restricts to which component connector can be applied ("formal parameter")

roles are not "sub-programs" of glue nor vice-versa:

- some attributes/actions of roles only restrict application
- some attributes/actions of glue are only for coordination

architecture bipartite graph of components and connectors, but:

- C2 allows connections between connectors
- Darwin does not distinguish component from connector

Definition

channel common vocabulary of glue and role

$$\langle I \mid A \rangle \equiv \quad \texttt{prog}\ P \quad \texttt{in}\ I \quad \texttt{do}\ \underset{a \in A}{[]}\ a : \text{true} \rightarrow \texttt{skip}$$

connection diagram $G \xleftarrow{\gamma} \langle I \mid A \rangle \xrightarrow{\rho} R$

G is glue, R is role

connector multiset of connections with common glue

application to components P_i is multiset of morphisms $\iota_i : R_i \rightarrow P_i$

Definition

example applied binary connector

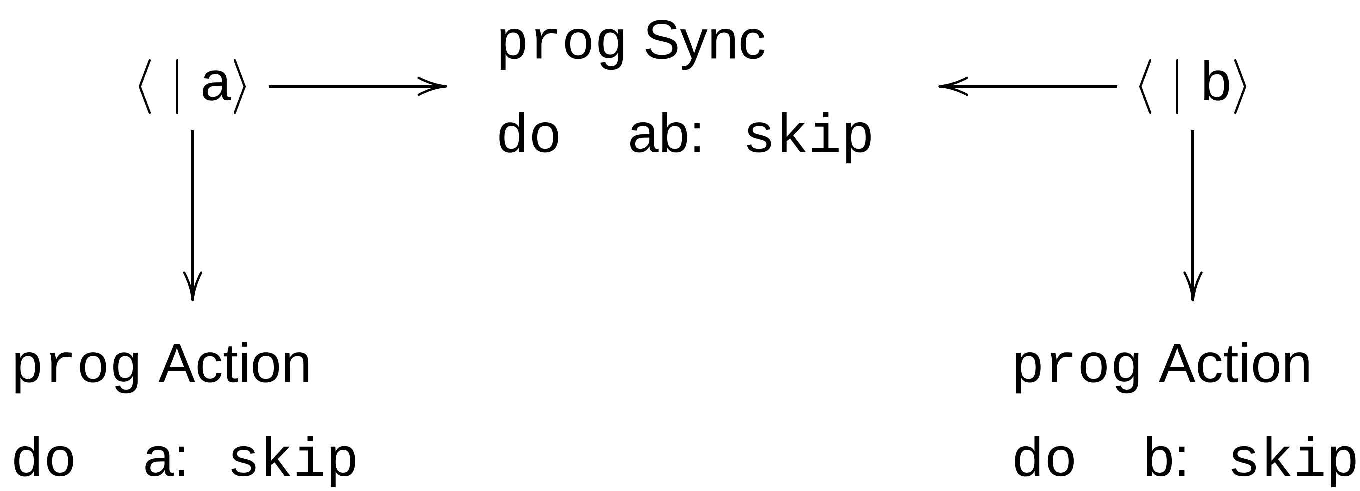
$$
\begin{array}{ccccc}
\langle I_1 \mid A_1 \rangle & \xrightarrow[\gamma_1]{} & G & \xleftarrow[\gamma_2]{} & \langle I_2 \mid A_2 \rangle \\
\downarrow{\scriptstyle \rho_1} & & & & \downarrow{\scriptstyle \rho_2} \\
R_1 & & & & R_2 \\
\downarrow{\scriptstyle \iota_1} & & & & \downarrow{\scriptstyle \iota_2} \\
P_1 & & & & P_2
\end{array}
$$

remark P_1 and P_2 usually distinct components (because internal synchronization not allowed) but may be same program (component type)

Catalog

Synchronization

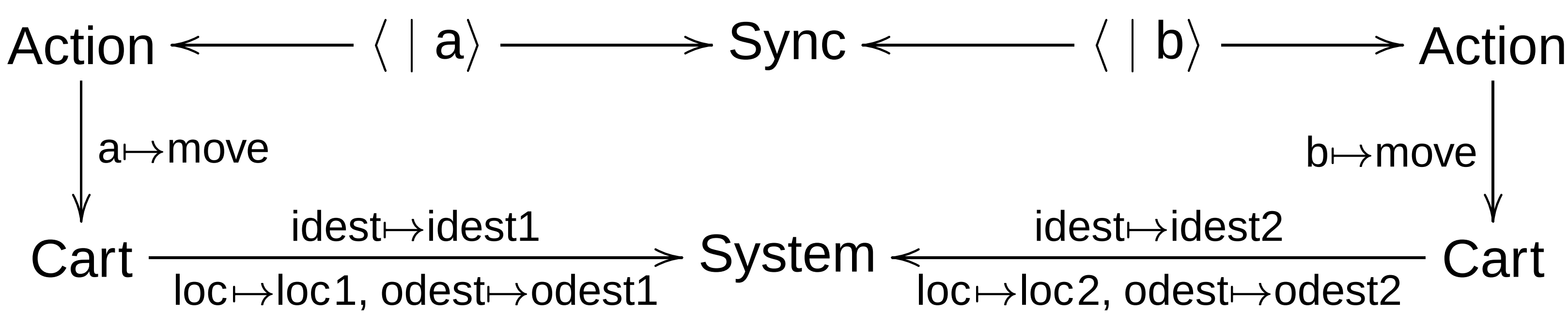
two actions ‘a’ and ‘b’ occurr simultaneously



logical analogy: equivalence

Synchronization

example: keep distance between carts



Synchronization

`prog` System

`in` idest1, idest2 : int

`out` loc1, odest1, loc2, odest2 : int

`init` $\bigwedge_{i=1,2}$ 0 $\leq$ loc$i$ $< U \wedge$ odest$i$ = -1 $\wedge$ obag$i$ = 0

`do` move1|move2: loc1 $\neq$ odest1 $\wedge$ loc2 $\neq$ odest2

$\rightarrow$ loc1 := loc1 $+_U$ 1 $\|$ loc2 := loc2 $+_U$ 1

[] get1: ...

[] get2: ...

[] get1|get2: ...

Subsumption

action 'a' subsumes action 'b': when 'a' executes, so does 'b'

"partial synchronisation" of 'a' with 'b' but not vice-versa

$\langle \mid a \rangle$ —— $a \mapsto ab$ ——→

```
prog Subsume
do   ab: skip
[]   b: skip
```

←—— $\{ab, b\} \leftarrow\!\!\dashv b$ —— $\langle \mid b \rangle$

```
prog Subsumer
do   a: skip
```

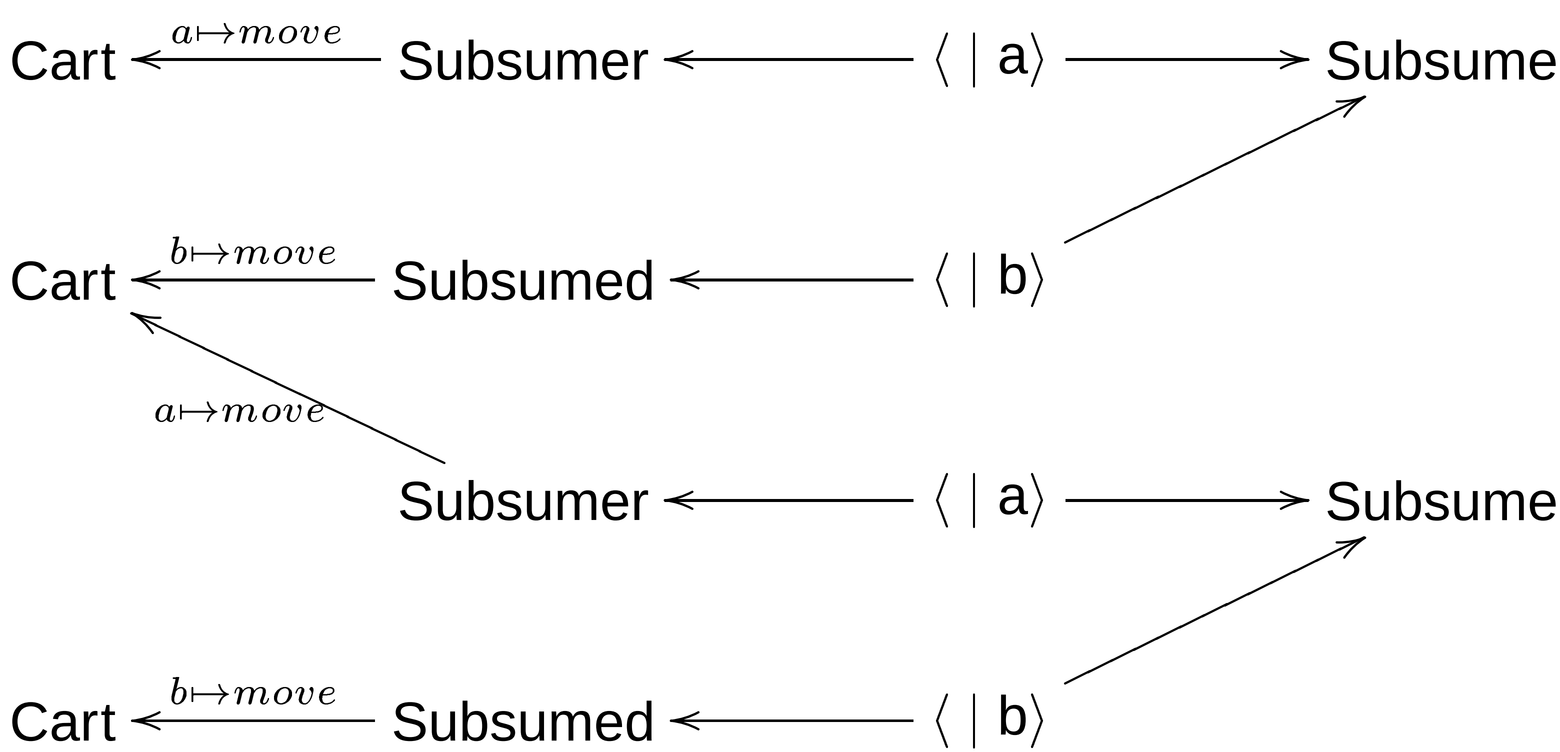
```
prog Subsumed
do   b: skip
```

logical analogy: implication

counter-positive: if 'b' cannot occur, neither can 'a'

Subsumption

example: if a cart moves, so does the cart in front of it



Subsumption

`prog` System

`in`	$\text{idest}_{1..3}$: int
`out`	$\text{loc}_{1..3}$, $\text{odest}_{1..3}$: int
`init`	$\bigwedge_{i=1,2,3} 0 \leq \text{loc}_i \leq U\text{- 1} \wedge \text{odest}_i = \text{-1}$
`do`	$\text{mv}_1 \vert \text{mv}_2 \vert \text{mv}_3\text{:} \ \bigwedge_{i=1,2,3} \text{loc}_i \neq \text{odest}_i \rightarrow \ldots$
[]	$\text{mv}_2 \vert \text{mv}_3\text{:} \ \bigwedge_{i=2,3} \text{loc}_i \neq \text{odest}_i \rightarrow \ \Vert_{\, i=2,3} \text{loc}_i := \text{loc}_i +_N 1$
[]	$\text{mv}_3\text{: loc}_3 \neq \text{odest}_3 \rightarrow \ \text{loc}_3 := \text{loc}_3 +_N 1$
$[]_{i=1,2,3}$	$\text{get}_i\text{: odest}_i = \text{-1} \rightarrow \ \text{odest}_i := \text{idest}_i$
$[]_{i=1,2,3}$	$\text{get}_i \vert \text{get}_{i+_3 1}\text{:} \ \text{odest}_i = \text{-1} \wedge \text{odest}_{i+_3 1} = \text{-1} \rightarrow \ldots$
[]	$\text{get}_1 \vert \text{get}_2 \vert \text{get}_3\text{:} \ \ldots$ [] $\text{get}_1 \vert \text{mv}_2 \vert \text{mv}_3\text{:} \ \ldots$
[]	$\text{get}_1 \vert \text{get}_2 \vert \text{mv}_3\text{:} \ \ldots$ [] $\text{get}_1 \vert \text{mv}_3\text{:} \ \ldots$ [] $\text{get}_2 \vert \text{mv}_3\text{:} \ \ldots$

Inhibition

an action 'a' no longer occurs

equivalent to synchronizing with an action that never executes

$\langle \mid a \rangle \longrightarrow$

`prog` Inhibit

`do` a: false $\rightarrow$ `skip`

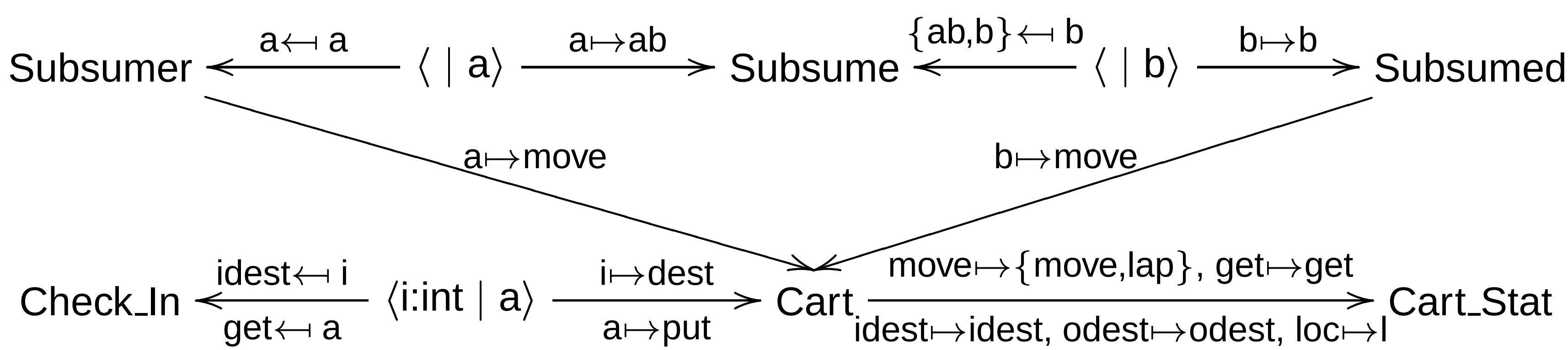
$\downarrow$

`prog` Action

`do` a: `skip`

Architectural Types

state restrictions on possible morphisms between components and connectors



Reconfiguration

- Introduction
- Graph Transformation
- Dynamic Reconfiguration
- Coordination
- Future Work
- Conclusion

Introduction

Motivation

- systems evolve: new requirements or new environment (failures, transient interactions)
- for safety or economical reasons, some systems cannot be shut off to be changed
- domain with some interest in SA community but little formal work

Issues

time before or at run-time (dynamic reconfiguration)

source user (ad-hoc); topology or state (programmed)

operations add/delete components/connections; query topology/state

constraints structural integrity; state consistency; application invariants

specification architecture description, modification, constraint languages

management explicit/centralised (configuration manager); implicit/distributed (self-organisation)

Related Work

- Distributed Systems, Mobile Computing, Software Architecture
- not at architectural level
- not arbitrary reconfigurations
- low-level behaviour specification (process calculi, term rewriting, etc.)
- interaction between computation and reconfiguration: complex, implicit, or blurred
- tool support, in particular automated analysis

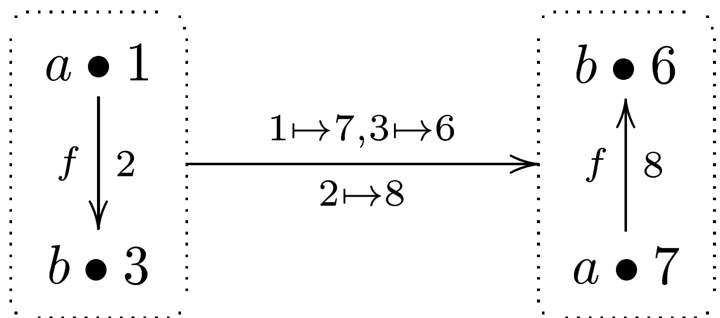
Approach

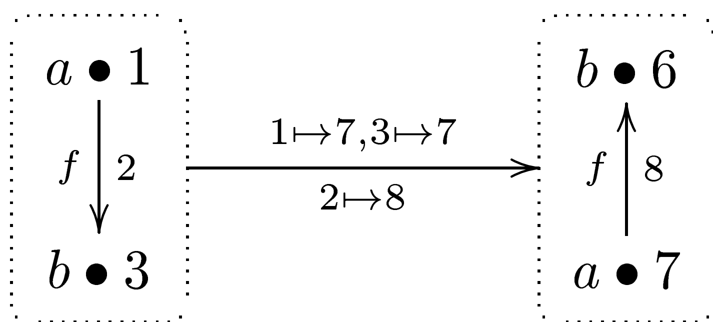
- use parallel program design language with state for computations
- category of programs with superposition
- architecture = categorical diagram; system = colimit
- architecture = graph; reconfiguration = rewriting
- apply algebraic graph transformation
  - uses category theory
  - much work done on it
  - double-pushout approach avoids side-effects
- conditional rules to add/remove components/connectors
- typed graphs for reconfiguration-invariant architectural type

Graph Transformation

Graph Category

- objects: directed graphs with labelled nodes and arcs
- morphisms: total functions between nodes and arcs preserving structure and labels

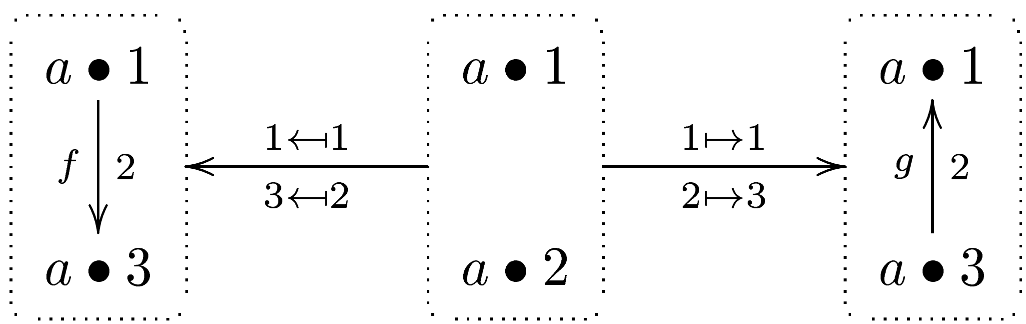
- example: 

- counter-example: 

Production

$$p : L \stackrel{l}{\longleftarrow} K \stackrel{r}{\longrightarrow} R$$

- left and right sides are graphs L and R
- L transformed into R through common subgraph K
- l and r are injective morphisms
- example:



- can be applied to G if $m : L \to G$ exists

Derivation

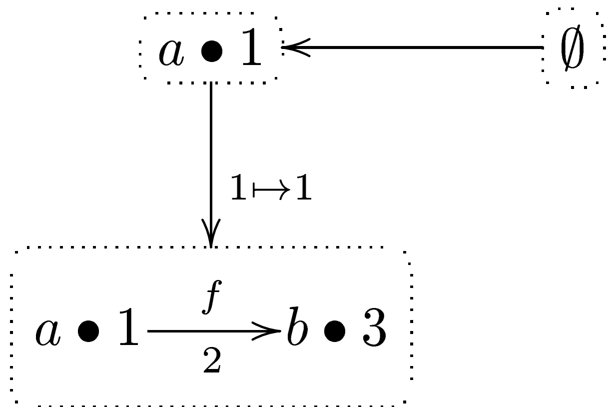
- $G \overset{p,m}{\Longrightarrow} H$ if 2 pushouts exist:

$$\begin{array}{ccccc} L & \xleftarrow[l]{} & K & \xrightarrow[r]{} & R \\ \big\downarrow m & & \big\downarrow d & & \big\downarrow m^* \\ G & \xleftarrow{l^*} & D & \xrightarrow{r^*} & H \end{array}$$

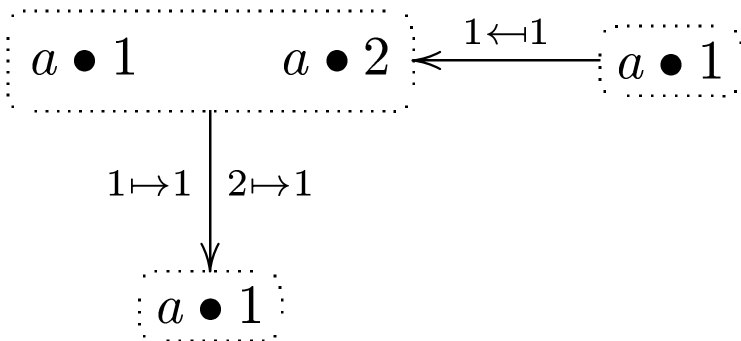
- $D = G - (L - K)$ and $H = D + (R - K)$
- injection l guarantees D is unique
- injection r guarantees p is reversible

Derivation

- D does not exist if a node to be removed has arcs



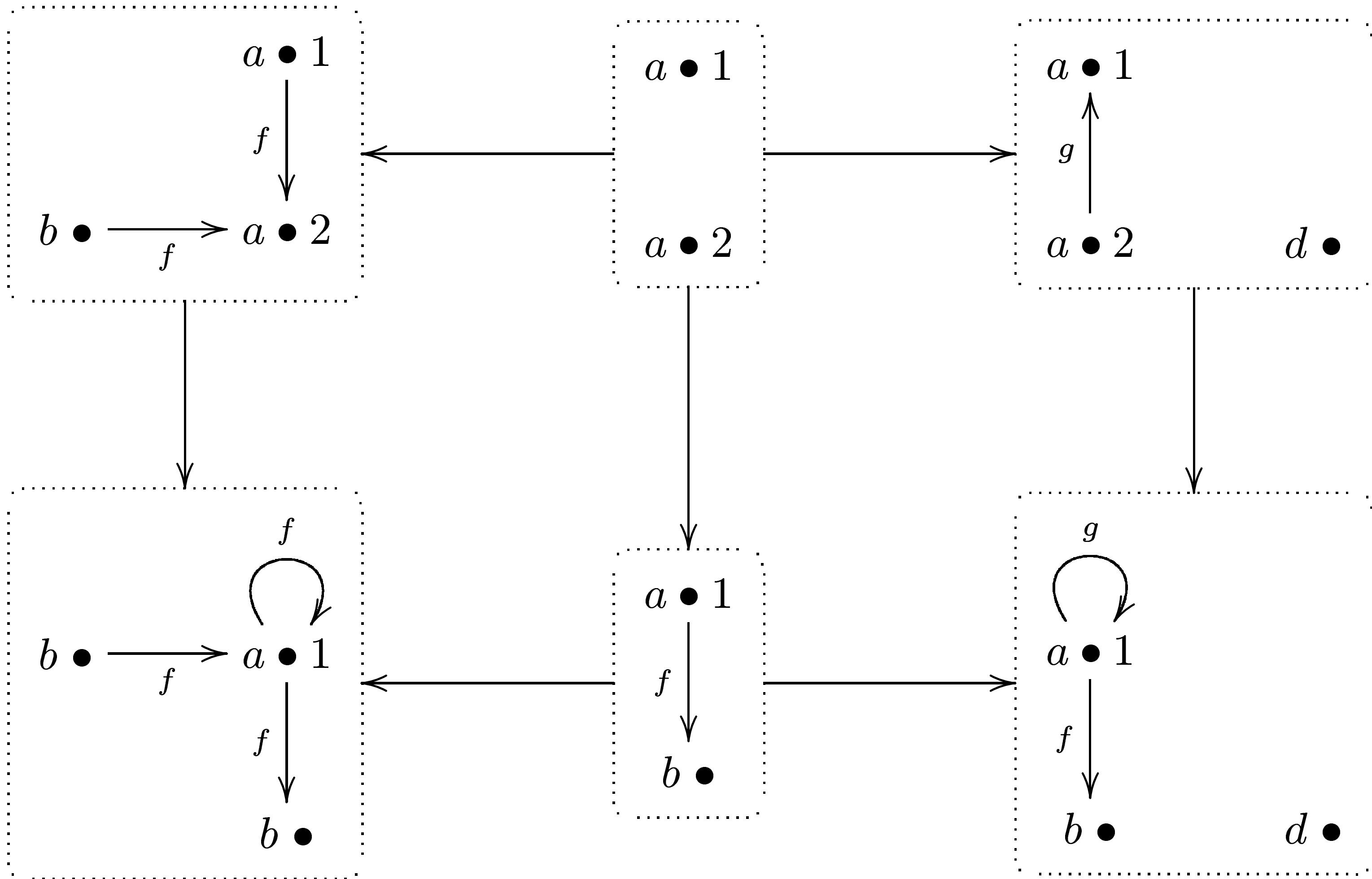
- D does not exist if a node is to be removed and kept



Example

- substitution of an arc
- removal of a connected node
- creation of an unconnected node
- non injective application

Example



Dynamic Reconfiguration

Definitions

architectures graphs labelled by programs with state and morphisms

reconfiguration derivation sequence; does not change state

initial architecture state are ground terms

rules $L \xleftarrow{l} K \xrightarrow{r} R$ **if** B

- B is condition over $Vars(L)$
- $Vars(R) \subseteq Vars(L)$ to determine state of new components

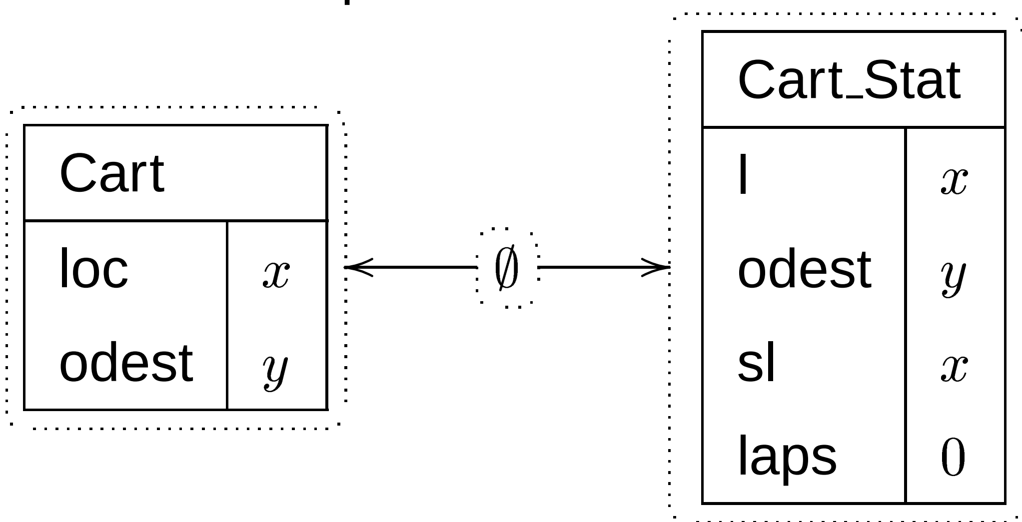
Definitions

derivation $G \overset{p,m}{\underset{\phi}{\Longrightarrow}} H$

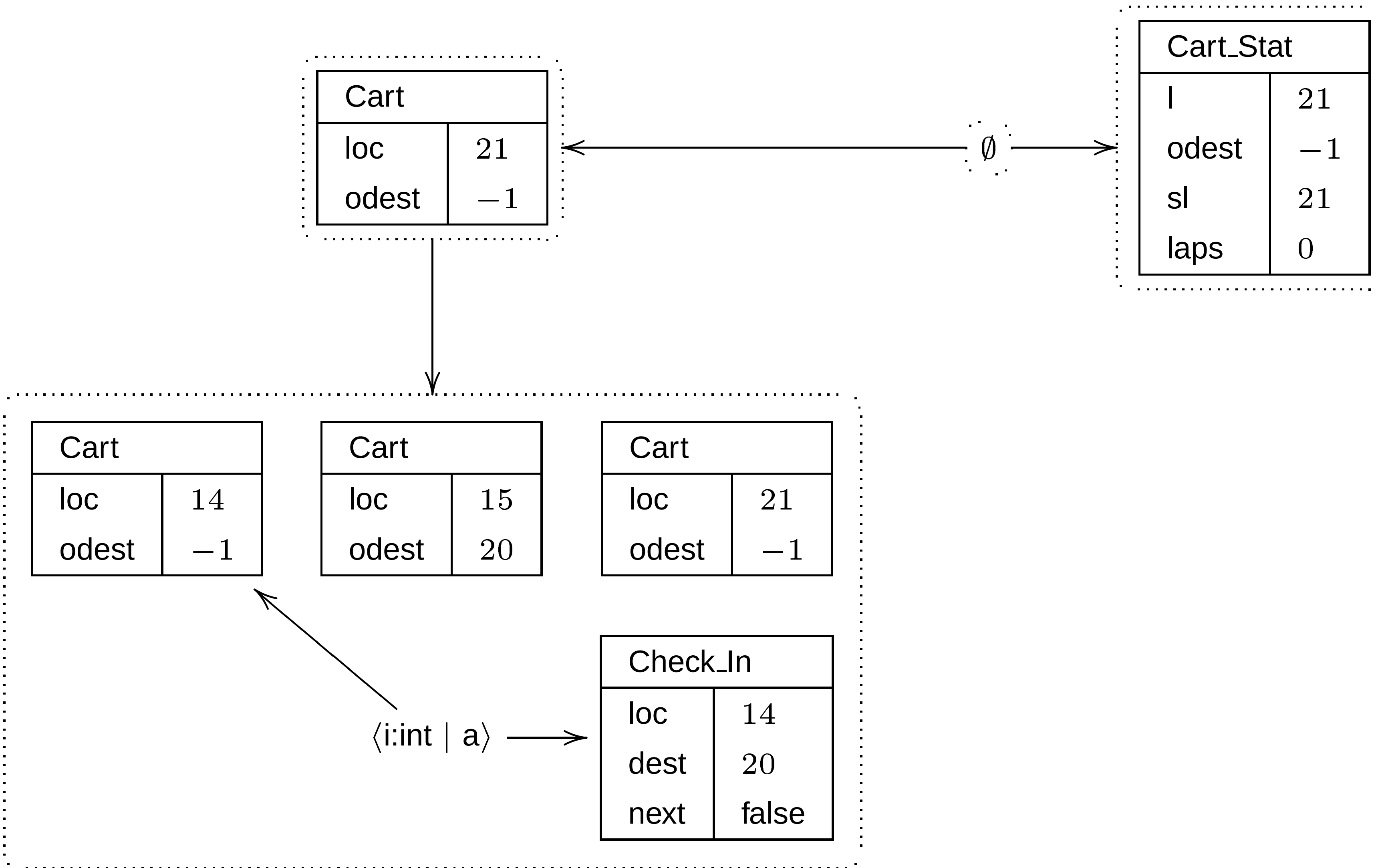
- substitution $\phi : Vars(L) \rightarrow Terms(\emptyset)$
- $\phi(B)$ is true
- $G \overset{\phi(p),m}{\Longrightarrow} H$ is derivation with
$\phi(p) = \phi(L) \overset{l}{\longleftarrow} \phi(K) \overset{r}{\longrightarrow} \phi(R)$
- each new program in R satisfies $\phi(\epsilon(ic))$

Examples (1)

substitution of isolated component

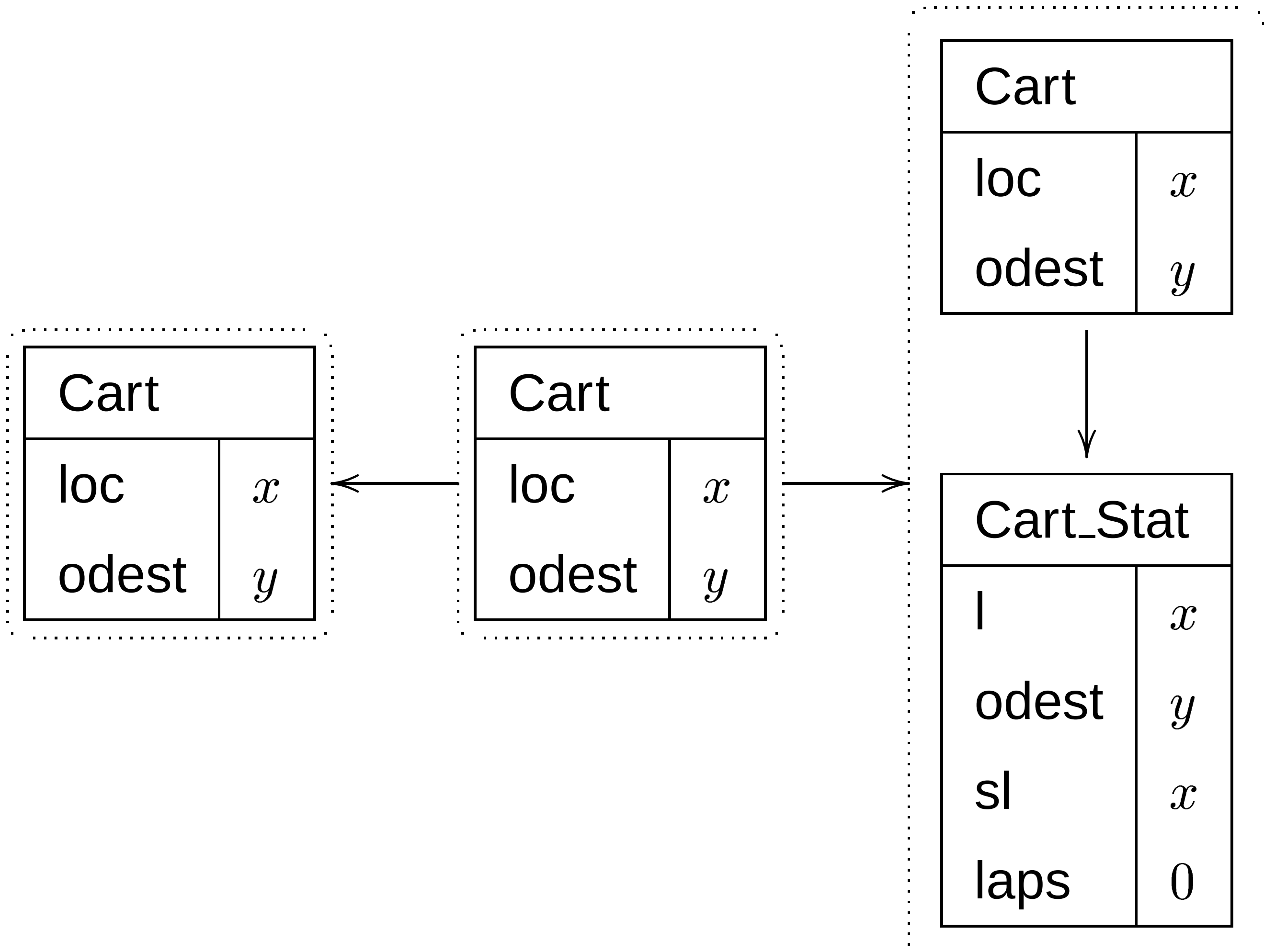


Examples (1)

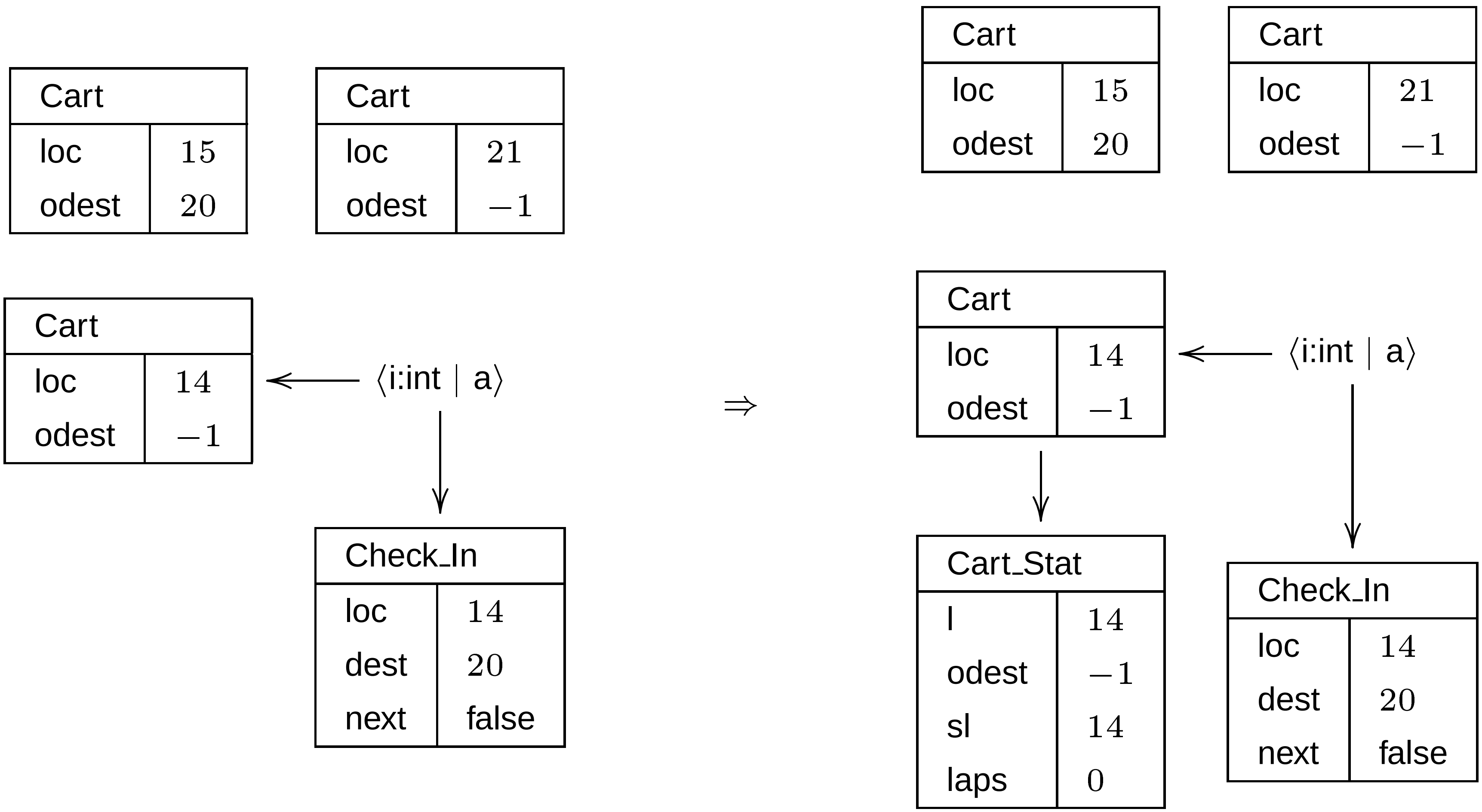

Cart
loc 21
odest −1
∅
Cart_Stat
l 21
odest −1
sl 21
laps 0
Cart
loc 14
odest −1
Cart
loc 15
odest 20
Cart
loc 21
odest −1
⟨i:int | a⟩
Check_In
loc 14
dest 20
next false

Examples (2)

component refinement

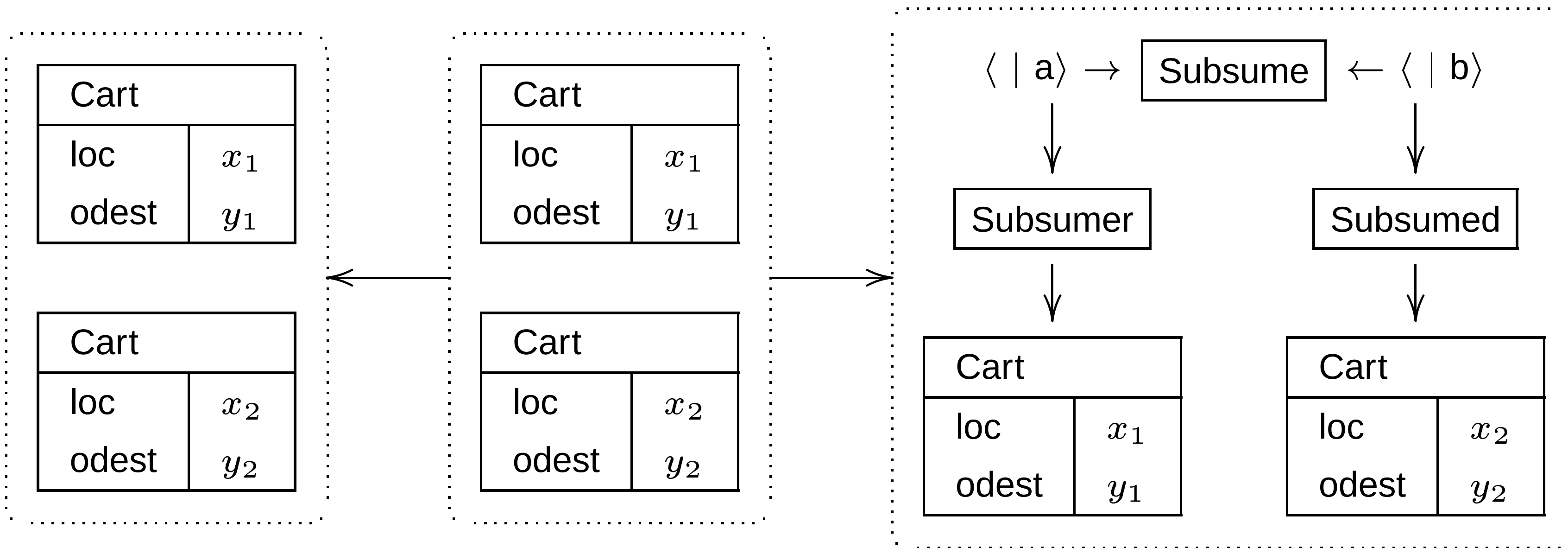


Examples (2)


Cart
loc 15
odest 20
Cart
loc 21
odest −1
Cart
loc 14
odest −1
⟨i:int | a⟩
Check_In
loc 14
dest 20
next false
⇒
Cart
loc 15
odest 20
Cart
loc 21
odest −1
Cart
loc 14
odest −1
⟨i:int | a⟩
Cart_Stat
l 14
odest −1
sl 14
laps 0
Check_In
loc 14
dest 20
next false

Examples (3)

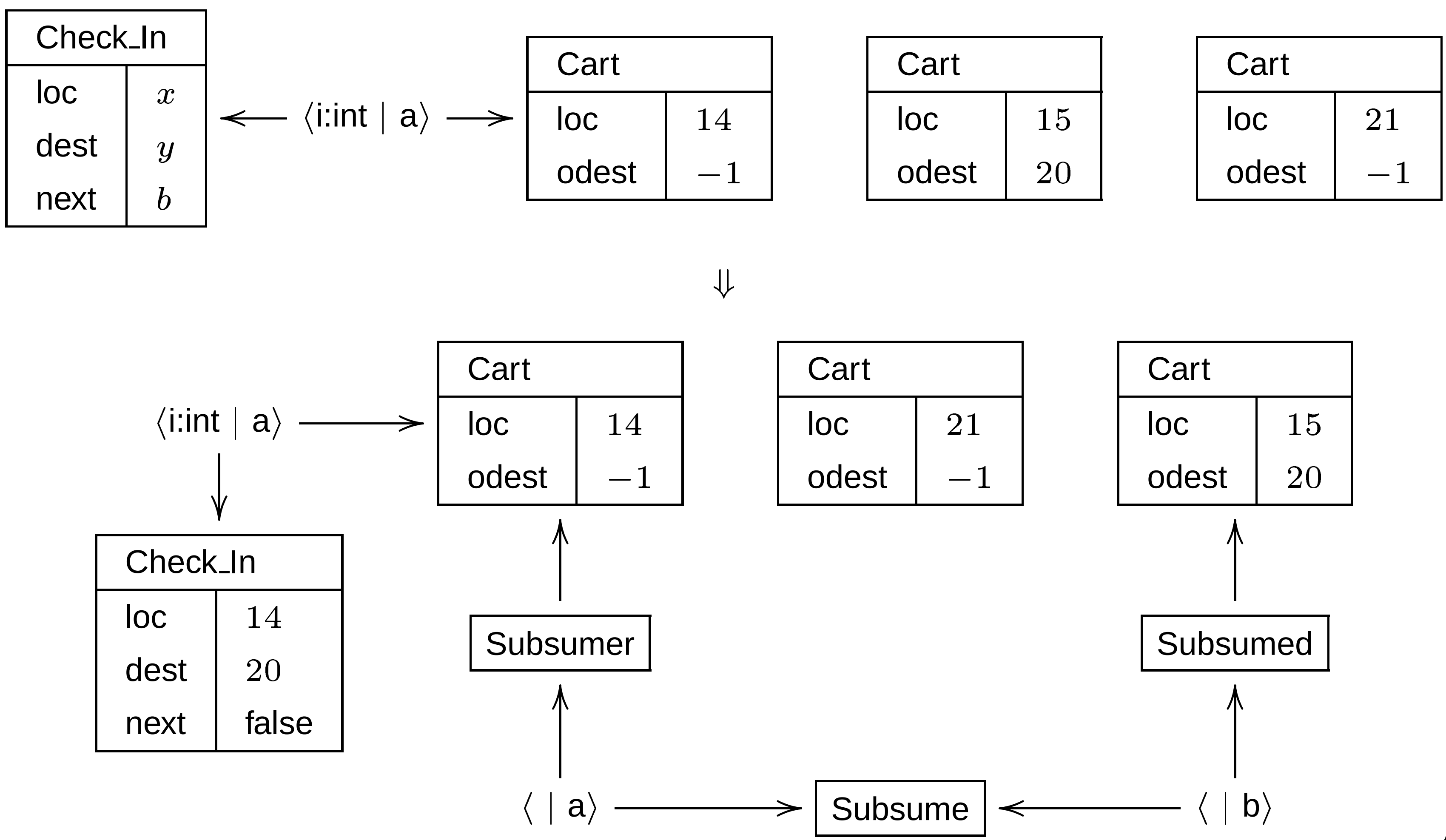
transient action subsumption to avoid collisions



if $x_2 = x_1 +_U 1 \vee x_2 = x_1 +_U 2$

opposite rule to remove connector when no longer needed

Examples (3)


Check_In
loc x
dest y
next b
⟨i:int | a⟩
Cart
loc 14
odest −1
Cart
loc 15
odest 20
Cart
loc 21
odest −1
⇓
⟨i:int | a⟩
Cart
loc 14
odest −1
Cart
loc 21
odest −1
Cart
loc 15
odest 20
Check_In
loc 14
dest 20
next false
Subsumer
Subsumed
⟨ | a⟩
Subsume
⟨ | b⟩

Coordination

Starting with initial architecture, execute repeatedly:

1. Change set of rules and architectural type, if necessary.
2. Execute reconfiguration sequence.
3. Compute colimit of current architecture.
4. Perform a computation step on the colimit.
5. Propagate new state back to components.

An implementation may execute actions directly on the architecture.

Conclusion

Future Work

- extend to the full language
- develop (re)configuration language and tool
- analysis of termination and uniqueness of reconfiguration
- hierarchic architectures

Advantages

- expressive, simple, uniform, explicit, algebraic framework to specify dynamic reconfiguration
- diagrams represent connectors, architectures, reconfiguration rules, and architectural types in graphical yet mathematical rigorous way
- colimits to obtain connector semantics, systems, reconfiguration steps and to relate explicitly computation and reconfiguration
- simple higher level program design language with intuitive state representation
- handle state transfer and removal/addition in correct state
- simple, declarative constraints on possible interactions