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 Arquiteturas 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

check-in → ✴️

- 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**

```
prog P
in I
out O
init ic
do [] a: G(a) \rightarrow \parallel o \in E(a, o)
    a \in A \quad o \in D(a)
```

- `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 ≤ loc < U ∧ odest = -1
do    move: loc ≠ odest → loc := (loc + 1) mod U
[]    get: odest = -1 → odest := idest

henceforth “loc + U 1”
Example

prog Check_In
out loc, dest : int; next : bool
init 0 ≤ loc < U ∧ next
do new: next → dest ∈ int || next := false
[] put: ¬next → next := true
Semantics

\textbf{in} \ the values of \( I \) are given by the environment and may change at each step

\textbf{init} \ the initial values of \( O \) are chosen non-deterministically satisfying \( \text{ic} \)

\[ \square \ \text{at each step choose one action randomly} \]

\( a: g \rightarrow \ldots \) \( \text{if } g \text{ is true, execute the action} \)

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

\( \parallel \ldots \text{evaluate the set expressions and then assign} \)

\( o \in D(a) \)

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

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

\[
a: G(a) \rightarrow \text{skip}
\]

- **:=** is the deterministic assignment

\[
c := c + 1 \equiv c :\in \{c + 1\}
\]

- **true** guards are omitted

\[
\text{inc: } c := c + 1 \equiv \text{inc: true } \rightarrow c := c + 1
\]

- 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 $\sigma : P \rightarrow P'$ given by two functions

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

s.t.

- output variables do not become input variables: $\sigma(O) \subseteq O'$
- action and variable domains are preserved: 
  
  $v \in D(\sigma(a')) \Rightarrow \sigma(v) \in D'(a') \text{ and } a' \in D'(\sigma(v)) \Rightarrow \sigma(a') \in D(v)$

- 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→move, move→move, get→get idest→idest, odest→odest, loc→l

Cart.stat

prog Cart_stat

in idest : int

out l, odest, sl, laps : int

init 0 ≤ l < U ∧ odest = -1 ∧ sl = l ∧ laps = 0

do move: l ≠ odest ∧ l +\( U \) 1 ≠ sl → l := l +\( U \) 1

[] lap: l ≠ odest ∧ l +\( U \) 1 = sl → l := l +\( U \) 1 || laps := laps + 1

[] get: odest = -1 → odest := idest
**Notation**

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

\[
\begin{align*}
\text{prog } P & \xrightarrow{x \mapsto \{y, z\}} \text{prog } P' \\
\text{do } x: \text{skip} & \quad \text{do } y: \text{skip} \quad \Rightarrow y \mapsto x \land z \mapsto x \\
[] & \quad z: \text{skip}
\end{align*}
\]

- morphisms are implicitly given through common names

\[
\begin{align*}
\text{prog } P & \quad \Rightarrow x \mapsto x | y \\
\text{do } x: \text{skip} & \quad \text{do } x | y: \text{skip} \\
\text{prog } P' & \\
\text{do } a: \text{skip} & \quad \text{do } a_1: \text{skip} \quad \Rightarrow a \mapsto \{a_1, a_2\} \\
[] & \quad a_2: \text{skip}
\end{align*}
\]
- 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

\[
\text{prog \ Load}
\]

\[
\text{Cart} \xleftarrow{i_{\text{dest}} \leftarrow i \quad \text{get} \leftarrow a} \quad \text{in} \quad i : \text{int} \quad \text{do} \quad a : \text{true} \rightarrow \text{skip}
\]

\[
\text{henceforth \ Load} \equiv \langle i : \text{int} \mid a \rangle
\]

\[
\text{prog \ System}
\]

\[
\text{out} \quad \text{loc}_1, \text{odest}, \text{loc}_2, \text{dest} : \text{int}; \text{next} : \text{bool}
\]

\[
\text{init} \quad 0 \leq \text{loc}_1, \text{loc}_2 < U \land \text{odest} = -1 \land \text{next}
\]

\[
\text{do} \quad \text{move} : \text{loc}_1 \neq \text{odest} \rightarrow \text{loc}_1 := \text{loc}_1 + U \ 1
\]

\[
\[] \quad \text{send} : \text{odest} = -1 \land \neg \text{next} \rightarrow \text{odest} := \text{dest} \mid \text{next} := \text{true}
\]

\[
\[] \quad \text{new} : \text{next} \rightarrow \text{dest} : \in \text{int} \mid \text{next} := \text{false}
\]

\[
\[] \quad \text{move}|\text{new} : \text{loc}_1 \neq \text{odest} \land \text{next}
\]

\[
\rightarrow \text{loc}_1 := \text{loc}_1 + U \ 1 \mid \text{dest} : \in \text{int} \mid \text{next} := \text{false}
\]
Definitions

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

**programs** with valuation \( \epsilon : O \rightarrow \text{Terms}(LV) \)

- 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

<table>
<thead>
<tr>
<th>Cart</th>
<th>Cart_State</th>
</tr>
</thead>
<tbody>
<tr>
<td>loc</td>
<td>l</td>
</tr>
<tr>
<td>odest</td>
<td>-1</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{loc} & \rightarrow l \\
\text{move} & \rightarrow \{\text{move, lap}\}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Cart_State</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>(x)</td>
</tr>
<tr>
<td>odest</td>
<td>-1</td>
</tr>
<tr>
<td>sl</td>
<td>(x + U)</td>
</tr>
<tr>
<td>laps</td>
<td>(y + 1)</td>
</tr>
</tbody>
</table>
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

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

**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

\[ \langle I_1 \mid A_1 \rangle \xrightarrow{\gamma_1} G \xleftarrow{\gamma_2} \langle I_2 \mid A_2 \rangle \]

\[ \begin{array}{c}
\rho_1 \\
R_1 \\
\iota_1 \\
P_1 \\
\end{array} \quad \begin{array}{c}
\rho_2 \\
R_2 \\
\iota_2 \\
P_2 \\
\end{array} \]

**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’ occur simultaneously

logical analogy: equivalence
Synchronization

example: keep distance between carts

Action ←  \langle | a \rangle  →  Sync ←  \langle | b \rangle  →  Action

\[
\text{Cart} \xrightarrow{\text{idest} \rightarrow \text{idest1}} \text{System} \xleftarrow{\text{idest} \rightarrow \text{idest2}} \text{Cart}
\]

\[
\text{Cart} \xrightarrow{\text{loc} \rightarrow \text{loc1}, \text{odest} \rightarrow \text{odest1}} \text{System} \xleftarrow{\text{loc} \rightarrow \text{loc2}, \text{odest} \rightarrow \text{odest2}} \text{Cart}
\]
Synchronization

prog System

in    idest1, idest2 : int
out   loc1, odest1, loc2, odest2 : int
init  \( \bigwedge_{i=1,2} 0 \leq loc_i < U \land odest_i = -1 \land obag_i = 0 \)
do    move1|move2: loc1 \neq odest1 \land loc2 \neq odest2
       \rightarrow loc1 := loc1 +_{U} 1 \parallel 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

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

\[
\begin{array}{c}
\text{Cart} \xleftarrow{a \to move} \text{Subsumer} \leftarrow \langle \mid a \rangle \xrightarrow{\text{Subsume}} \\
\text{Cart} \xleftarrow{b \to move} \text{Subsumed} \leftarrow \langle \mid b \rangle \xrightarrow{\text{Subsume}} \\
\text{Subsumer} \xleftarrow{a \to move} \langle \mid a \rangle \xrightarrow{\text{Subsume}} \\
\text{Cart} \xleftarrow{b \to move} \text{Subsumed} \leftarrow \langle \mid b \rangle \xrightarrow{\text{Subsume}}
\end{array}
\]
Subsumption

prog System

in       idest_{1..3} : int
out      loc_{1..3}, odest_{1..3} : int
init     \( \land_{i=1,2,3} 0 \leq \text{loc}_i \leq U - 1 \land \text{odest}_i = -1 \)
do      \text{mv}_1 | \text{mv}_2 | \text{mv}_3: \land_{i=1,2,3} \text{loc}_i \neq \text{odest}_i \rightarrow \ldots
[]      \text{mv}_2 | \text{mv}_3: \land_{i=2,3} \text{loc}_i \neq \text{odest}_i \rightarrow \parallel 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 = -1 \rightarrow \text{odest}_i := \text{idest}_i
[i=1,2,3] \text{get}_i | \text{get}_{i+1}: \text{odest}_i = -1 \land \text{odest}_{i+1} = -1 \rightarrow \ldots
[]      \text{get}_1 | \text{get}_2 | \text{get}_3: \ldots [] \text{get}_1 | \text{mv}_2 | \text{mv}_3: \ldots
[]      \text{get}_1 | \text{get}_2 | \text{mv}_3: \ldots [] \text{get}_1 | \text{mv}_3: \ldots [] \text{get}_2 | \text{mv}_3: \ldots
Inhibition

an action ‘a’ no longer occurs

equivalent to synchronizing with an action that never executes

\[
\langle | a \rangle \rightarrow \text{prog Inhibit}
\]

\[
do \quad a: \text{false} \rightarrow \text{skip}
\]

\[
\text{prog Action}
\]

\[
do \quad a: \text{skip}
\]
state restrictions on possible morphisms between components and connectors

Architectural Types

Subsumer $\xrightarrow{a \rightarrow a} \langle | a \rangle \xrightarrow{a \rightarrow ab} \xrightarrow{\{ab,b\} \rightarrow b} \xrightarrow{b \rightarrow b} \text{Subsumed}$

Subsume $\xrightarrow{\text{a} \rightarrow \text{move}} \xrightarrow{\text{b} \rightarrow \text{move}}$

Check_In $\xrightarrow{\text{idest} \rightarrow \text{i}} \langle \text{i: int} | a \rangle \xrightarrow{\text{i} \rightarrow \text{dest}} \xrightarrow{\text{move} \rightarrow \{\text{move, lap}\}} \xrightarrow{\text{idest} \rightarrow \text{idest}, \text{odest} \rightarrow \text{odest}, \text{loc} \rightarrow \text{l}} \xrightarrow{\text{get} \rightarrow \text{get}} \xrightarrow{\text{Cart} \rightarrow \text{Cart Stat}}$
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/connector
- 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
Reconfiguration

Graph Transformation

- example:

- counter-example:
Production

\[ p : L \xleftarrow{l} K \xrightarrow{r} R \]

- 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 \( m : L \rightarrow G \) exists
Derivation

- \( G \xrightarrow{p,m} H \) if 2 pushouts exist:

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

- \( D = G - (L - K) \) 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

\[
\begin{array}{c}
\text{a} \bullet 1 \quad \emptyset
\end{array}
\]

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

\[
\begin{array}{c}
\text{a} \bullet 1 \quad \text{b} \bullet 3
\end{array}
\]
Example

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

Reconfiguration Graph Transformation
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 $L \xleftarrow{l} K \xrightarrow{r} R \text{ if } B$

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

Dynamic Reconfiguration

Definitions

derivation

\[ G \xrightarrow{p,m} \phi H \]

- substitution \( \phi : Vars(L) \rightarrow Terms(\emptyset) \)
- \( \phi(B) \) is true
- \( G \xrightarrow{\phi(p),m} H \) is derivation with
  \[ \phi(p) = \phi(L) \xleftarrow{l} \phi(K) \xrightarrow{r} \phi(R) \]
- each new program in \( R \) satisfies \( \phi(\epsilon(ic)) \)
Examples (1)

substitution of isolated component

<table>
<thead>
<tr>
<th>Cart</th>
<th>loc</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>odest</td>
<td></td>
<td>y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cart_Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
</tr>
<tr>
<td>odest</td>
</tr>
<tr>
<td>sl</td>
</tr>
<tr>
<td>laps</td>
</tr>
</tbody>
</table>
Examples (1)

Cart
loc   21
odest -1

Cart
loc   14
odest -1

Cart
loc   15
odest -1

Cart
loc   20
odest -1

Cart
loc   21
odest -1

Check_In
loc   14
dest   20
next   false

Cart Stat
l   21
odest   -1
sl   21
laps   0
Examples (2)

Component refinement

```
Cart
loc  x
odest  y

Cart
loc  x
odest  y

Cart
loc  x
odest  y

Cart_stat
l  x
odest  y
sl  x
laps  0
```
Examples (2)

Cart
loc  15
odest  20

Cart
loc  21
odest  -1

Cart
loc  21
odest  -1

Cart
loc  14
odest  14

Check_In
loc  14
dest  20
next  false

Cart
loc  15
odest  20

Cart
loc  14
odest  -1

Cart
loc  14
odest  -1

Check_In
loc  14
dest  20
next  false
Examples (3)

transient action subsumption to avoid collisions

if $x_2 = x_1 + U_1 \lor x_2 = x_1 + U_2$

opposite rule to remove connector when no longer needed
Examples (3)

Check_In
<table>
<thead>
<tr>
<th>loc</th>
<th>dest</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$y$</td>
<td>$b$</td>
</tr>
</tbody>
</table>

\[
\langle i: \text{int} \mid a \rangle 
\]

Cart
<table>
<thead>
<tr>
<th>loc</th>
<th>odest</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>-1</td>
</tr>
</tbody>
</table>

Cart
<table>
<thead>
<tr>
<th>loc</th>
<th>odest</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>20</td>
</tr>
</tbody>
</table>

Cart
<table>
<thead>
<tr>
<th>loc</th>
<th>odest</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>-1</td>
</tr>
</tbody>
</table>

Subsumer

\[
\langle i: \text{int} \mid a \rangle 
\]

Subsume

\[
\langle \mid b \rangle 
\]

Check_In
<table>
<thead>
<tr>
<th>loc</th>
<th>dest</th>
<th>next</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>20</td>
<td>false</td>
</tr>
</tbody>
</table>

Subsumed
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