Coordination-based Evolution

José Fiadeiro **Antónia Lopes**

Michel Wermelinger

LabMAC/University of Lisbon and ATX Software
PORTUGAL

with **Luís Andrade**

Presentation at the FSE network meeting, 18/1/02

Objectives

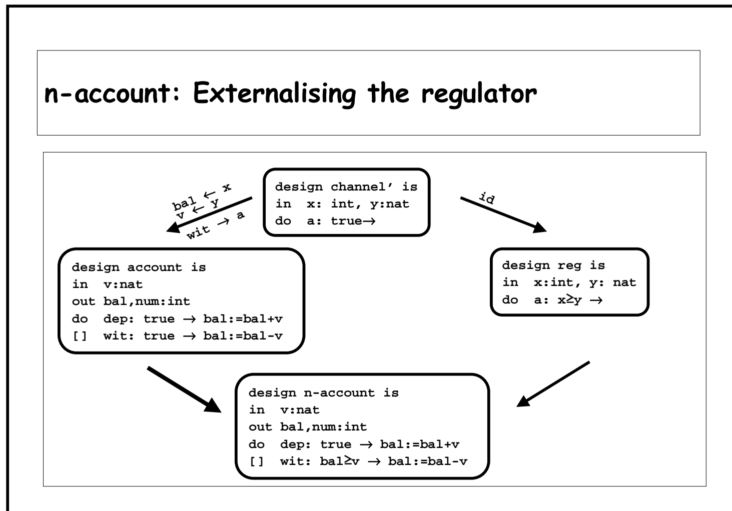
- To provide a well founded approach to "Coordination", abstracting a mathematical characterisation from existing languages and models, and using it to generalise the concept to other contexts.
- To show how the proposed characterisation can be used for giving semantics to "Software Architectures".
- To show how "coordination technologies" can provide an effective solution for software evolution in volatile business domains.

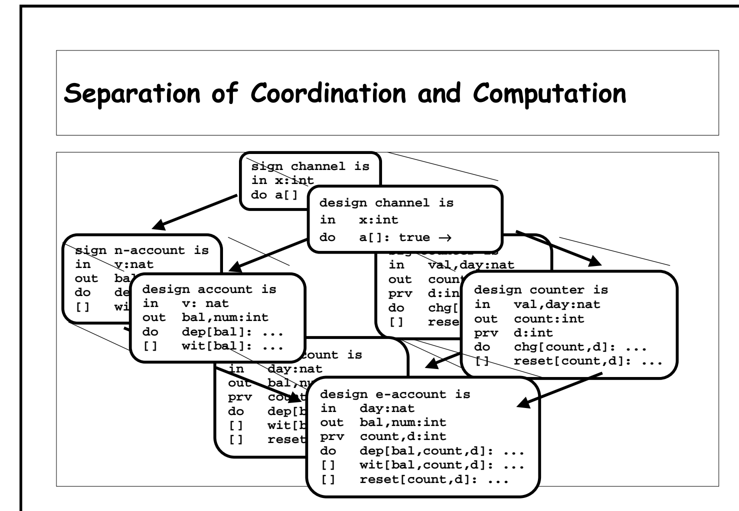
Why Coordination ?

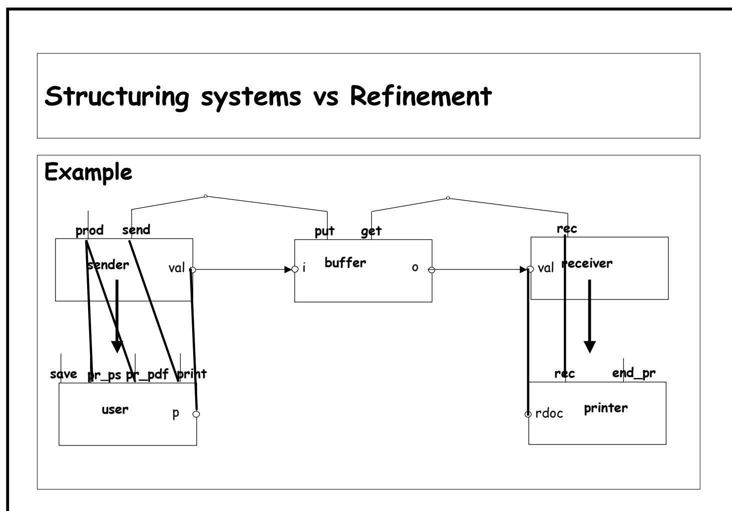
- "Recent" languages like Linda, Gamma, Manifold, … have promoted the separation between **computation** (what is responsible for the functionality of services in basic components) and **coordination** (the mechanisms that are made available for components to interact);
- "Programming by **emergence**": local functionalities + interactions
- **Black-box** view of components: interactions can evolve without changing the computations.

Why Category Theory ?

- The mathematical tool, par excellence, for addressing "structure" and "modularity".
- In Category Theory, entities are characterised in terms of the relationships they have to other entities and not in terms of their internal representation.
  - The information one gets from the structure of an entity is determined from the way that entity "interacts" with the other entities.
  - This is analogous, for instance, to the encapsulation mechanisms made available by Abstract Data Types and Object-Oriented Programming.







Generalisations

This categorical framework provides

- an ADL-independent semantics for existing principles and techniques of SA
- a basis for extending the capabilities of existing ADLs.

  Examples:
  - Heterogeneous connectors
  - Higher-order connectors

Reconfiguration: Related Work

Work done in Distributed Systems, Mobile Computing, Software Architecture has at least one of the following drawbacks:

- not addressed at the architectural level
- arbitrary reconfigurations not supported
- only low-level behaviour specification (process calculi, term rewriting, etc.)
- interaction between computation and reconfiguration is complex, implicit, or blurred

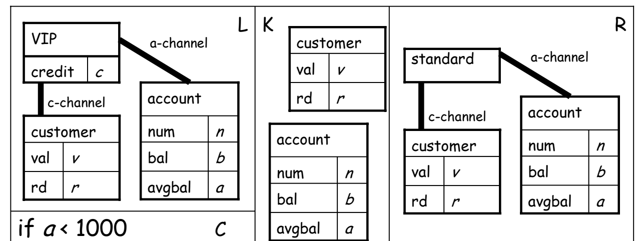
On the other hand, they sometimes provide tool support, in particular automated analysis.

Reconfiguration: Approach

- Explore the categorical approach to software architectures and parallel program design
  - architecture = categorical diagram; system behaviour = colimit
  - architecture = graph; reconfiguration = rewriting
- Develop a reconfiguration language for easier specification and analysis.

Modifying the contract

The following rule restores a VIP contract to standard when the average balance is below 1000.



Reconfiguration Specification

- rewrite rules are cumbersome to write: repetition of nodes in graphs K and L; dummy nodes/arcs to control the way rules are applied
- ideal: reconfiguration language with high-level programming constructs
- but: ADLs only provide minimal reconfiguration support; distributed systems have powerful languages but do not have architectural abstractions
- goal: compact, conceptually elegant language with formal semantics for describing reconfiguration within architectural description of a system

Main script

```
script Main
prv i : record(a : Account)
script RestoreStandard ... end script
for i in match {a:Account | with
                            a.avgbal<1000}
  loop
     RestoreStandard(i.a)
  end loop
end script
```

Auxiliary Script

```
script RestoreStandard
in   a: Account
prv i: record(c:Customer; co:VIP)
for i in match {c:Customer;co:VIP |co(c, a)}
  loop
     remove i.co;
     create standard(i.c, a);
  end loop
end script
```


Notation for coordination contracts

```
coordination contract Traditional package
  partners x : Account; y : Customer;
  constraints ?owns(x,y)=TRUE;
  coordination
     tp:  when  y ->> x.withdrawal(z)
          do    call x.withdrawal(z)
          with  x.Balance() > z
end contract
```

Coordination Rules

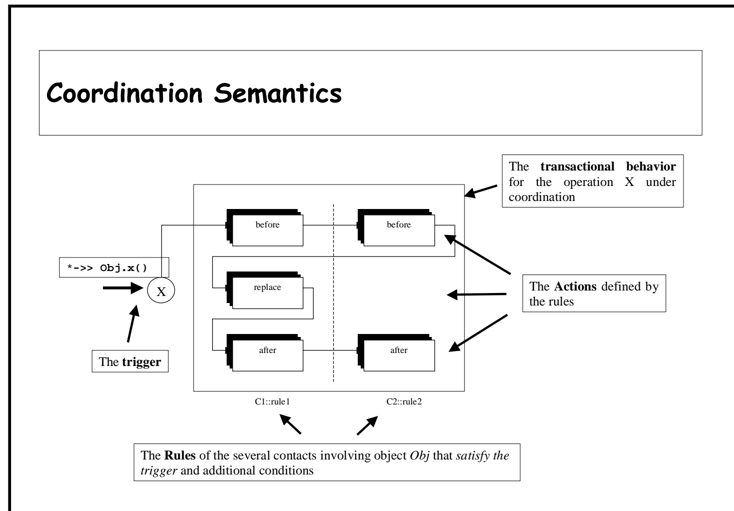
- A Coordination Rule has the form

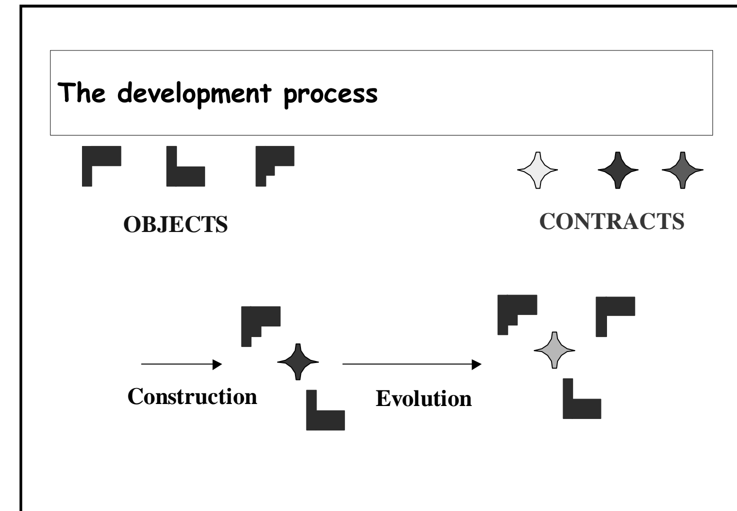
<name> : **when** *<trigger>*
with *<guardCondition>*
do *<set of actions>*

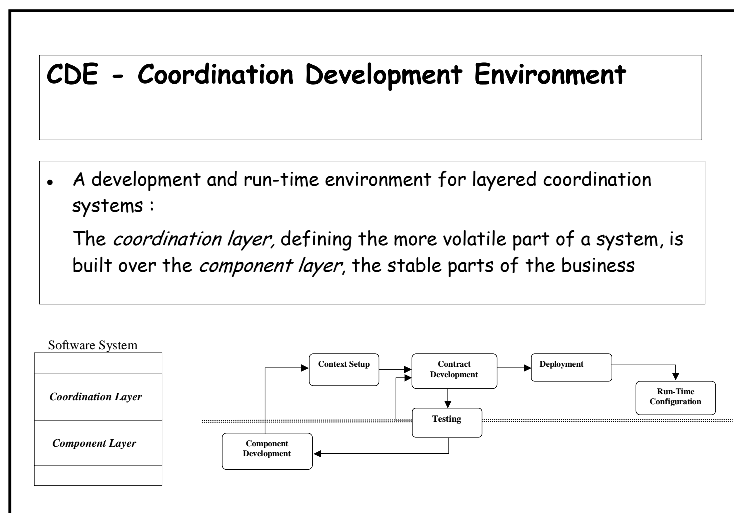
The **trigger** defines when a rule must be considered active. It may be a *condition*, or a *request* to a participant operation

The **guard condition** imposes additional constraints on the reaction to the trigger, when regulated by this rule

The **actions** describe the behavior defined by the rule:
- extra behaviour to be executed **before** or **after** the trigger operation,
- or **replacement** behavior for the trigger operation







Concluding remarks

- Increased separation of the domain concepts (objects) from the business rules that regulate their behaviour;
- Coordination features available as first-class citizens through a specific semantic primitive;
- Support for different levels of change, reflecting the evolution of the domain:
  - Flexible mechanisms for inheritance of behaviour;
  - Separation of coordination from computation.

Claimed contributions

- Increased separation of the domain concepts from the business rules that regulate their behaviour;
  - Recognising two different dynamics in system evolution: changes to the way components operate and changes to the way components are integrated (**white** vs **black** box);
  - More flexibility in the software development process (**plug and play**);
  - Better integration/coordination of third-party, closed components (e.g. legacy systems)
- One step closer to a real industry of components.

URLs

- Papers:
  - www.atxsoftware.com/publications.html (also includes papers on CommUnity and the categorical approach to software architecture)
- Coordination Development Environment:
  - www.atxsoftware.com/CDE
- CommUnity Workbench:
  - http://ctp.di.fct.unl.pt/~mw/sw/cw