Software Architecture: Coordination and Evolution

J.L.Fiadeiro

University of Leicester

Plan

- **Architectures**:
  - What roles do they play in software development
  - How do they relate to coordination and evolution
- The **Coordination** dimension
  - Externalisation of **interactions** in connectors
  - Semantic modelling primitives for coordinating interactions
- **CommUnity**
  - Formalising components, connectors and configurations
  - Refinement vs Composition
- The **Distribution** dimension
  - Externalisation of **distribution** in connectors
  - Semantic modelling primitives for location-aware computing

Contributions

- ATX Software
- Antónia Lopes @ University of Lisbon
- The AGILE consortium
  - Univ. Munich
  - Univ. Pisa
  - Univ. Florence
  - Univ. Warsaw
  - Univ. Lisbon
  - ATX Software
  - ISTI - CNR

IST-2001-32747
Architectures for Mobility
Jan 02 - Apr 05

1

The Rôle of Architecture

What is it?

- A software architecture for a system is the structure or structures of the system, which comprise elements, their externally-visible behavior, and the relationships among them.

  source: L.Bass, P.Clements and R.Kazman. Software Architecture in Practice. Addison-Wesley, 1998.

- There are many views, as there are many structures, each with its own purpose and focus in understanding the organisation of the system.

What is it for?

component *(n): a constituent part*

complex *(a): composed of two or more parts*

architecture *(n):*

1 *: formation or construction as, or as if, the result of conscious act;*

2 *: a unifying or coherent form or structure*

A case of "complexity"

A case of "complexity"

7

in-the-head

mnemonics
result-driven
symbolic information
elementary control flow

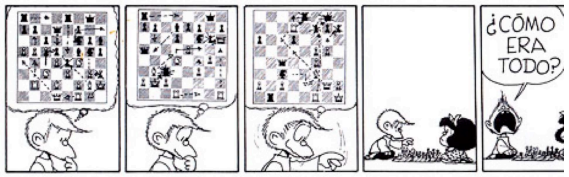
- **"One man and his problem…"** (and **his program**, and **his machine**)
- The Science of **Algorithms** and **Complexity**
- not so much **Engineering** but more of **Craftsmanship** (one of a kind)
- a case for **virtuosi**

A case of "complexity"

in-the-head
mnemonics
result-driven
symbolic information
elementary control flow

Example

- "One man and his problem…" (and **his program**, and **his machine**)



Using a pocket calculator to select the next move

A case of "complexity"

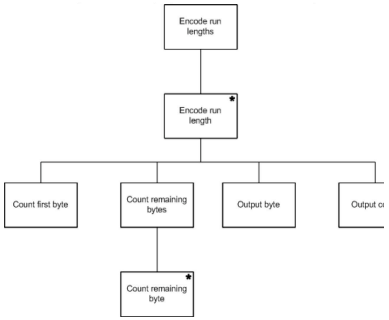
in-the-head	in-the-small
mnemonics	I/O specs
result-driven	algorithms
symbolic information	data structures and types
elementary control flow	execute once termination

- The need for commercialisation
- "One man and his problem…" (and **his program**, but their **machine)**
- The Science of **Program Analysis and Construction**
- Commerce, but not yet **Engineering**

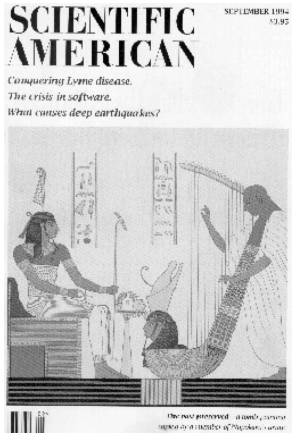
A case of "complexity"

in-the-head	in-the-small
mnemonics	I/O specs
result-driven	algorithms
symbolic information	data structures and types
elementary control flow	execute once termination

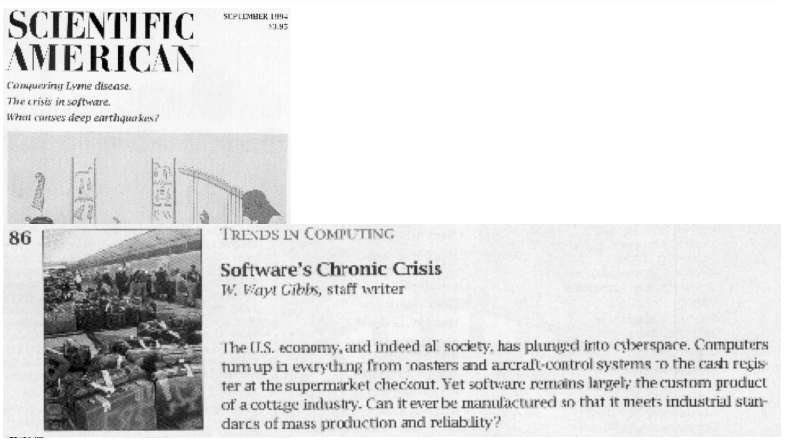
Program *Architectures*



10 years ago, the "software crisis"



10 years ago, the "software crisis"



SCIENTIFIC AMERICAN

SEPTEMBER 1994
$3.95

Conquering Lyme disease.
The crisis in software.
What causes deep earthquakes?

86

TRENDS IN COMPUTING

Software's Chronic Crisis

W. Wayt Gibbs, staff writer

The U.S. economy, and indeed all society, has plunged into cyberspace. Computers turn up in everything from toasters and aircraft-control systems to the cash register at the supermarket checkout. Yet software remains largely the custom product of a cottage industry. Can it ever be manufactured so that it meets industrial standards of mass production and reliability?

10 years ago, the "software crisis"

- *The challenge of complexity is not only large but also growing. […]. To keep up with such demand, programmers will have to change the way that they work. "**You can't build skyscrapers using carpenters**," Curtis quips.*
- *[…] Musket makers did not get more productive until Eli Whitney figured out how to manufacture interchangeable parts that could be assembled by any skilled workman. In like manner, software parts can, if properly standardized, be **reused** at many different scales.*
- *[…]In April, NIST announced that it was creating an Advanced Technology Program to help engender **a market for component-based software**.*

A case of "complexity"

in-the-head	in-the-small	in-the-large
mnemonics	I/O specs	**complex specs**
result-driven	algorithms	**system modules**
symbolic information	data structures and types	**databases, persistence**
elementary control flow	execute once termination	**continuous execution**

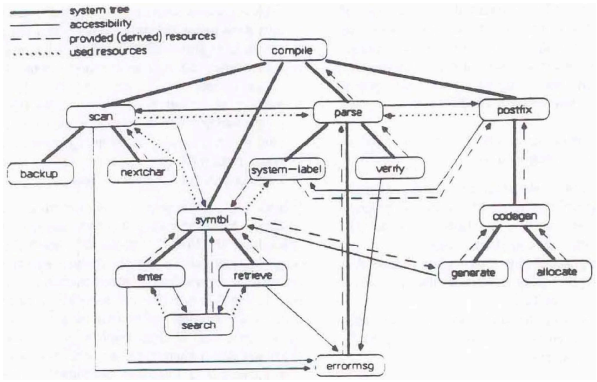
- **"One man and his problem…"** (but **their programs**)
- The Science of **Software Specification and Design**
- **Engineering**

The case for MILs

- **Modelling Interconnection Languages** for programming-in-the-large (DeRemer and Kron 75)
- Address the global structure of a system in terms of
  - what its modules and resources are
  - how they fit together in the system
- Interconnection may be data or control oriented
- Descriptions are concise, precise and verifiable

The case for MILs

Architectures of Usage



The case for new mathematics

The case for new mathematics

- **Algebraic techniques for structuring specifications**
  - "Putting Theories together to Make Specifications"
  - The theory of **Institutions**
  - The role of **Category Theory**

The case for new mathematics

- **Algebraic techniques for structuring specifications**
  - "Putting Theories together to Make Specifications"
  - The theory of **Institutions**
  - The role of **Category Theory**
- **Temporal logics** for continuous/reactive execution

The case for objects/components

- *In like manner, software parts can, if properly standardized, be **reused** at many different scales.*
- *[…]In April, NIST announced that it was creating an Advanced Technology Program to help engender **a market for component-based software**.*

The case for objects/components

13

- *In like manner, software parts can, if properly standardized, be **reused** at many different scales.*
- *[…]In April, NIST announced that it was creating an Advanced Technology Program to help engender **a market for component-based software**.*

VALENCIA 2005 University of Leicester

- Builds on a powerful methodological metaphor – **clientship**

The case for objects/components

13

- *In like manner, software parts can, if properly standardized, be **reused** at many different scales.*
- *[…]In April, NIST announced that it was creating an Advanced Technology Program to help engender **a market for component-based software**.*

VALENCIA 2005 University of Leicester

- Builds on a powerful methodological metaphor – **clientship**

- Inheritance hierarchies for **reuse**

The case for objects/components

13

- *In like manner, software parts can, if properly standardized, be **reused** at many different scales.*
- *[…]In April, NIST announced that it was creating an Advanced Technology Program to help engender **a market for component-based software**.*

VALENCIA 2005 University of Leicester

- Builds on a powerful methodological metaphor – **clientship**
- Inheritance hierarchies for **reuse**

- Software **construction** becomes like child's play

VALENCIA 2005 University of Leicester

Yet, in 2003 the crisis was going on

- *Computing has certainly got faster, smarter and cheaper, but it has also become much more complex.*
- *Ever since the orderly days of the mainframe, which allowed tight control of IT, computer systems have become ever more distributed, more heterogeneous and harder to manage. […]*
- *In the late 1990s, the internet and the emergence of e-commerce "broke IT's back". Integrating incompatible systems, in particular, has become a big headache. A measure of this increasing complexity is the rapid growth in the IT services industry. […]*

Yet, in 2003 the crisis was going on

14

- *Computing has certainly got faster, smarter and cheaper, but it has also become much more complex.*
- *Ever since the orderly days of the mainframe, which allowed tight control of IT, computer systems have become ever more distributed, more heterogeneous and harder to manage. […]*
- *In the late 1990s, the internet and the emergence of e-commerce "broke IT's back". Integrating incompatible systems, in particular, has become a big headache. A measure of this increasing complexity is the rapid growth in the IT services industry. […]*

VALENCIA 2005 University of Leicester

and the "silver bullet" became…

- *Computing is becoming a utility and software a service. This will profoundly change the economics of the IT industry. […]*
- *For software truly to become a service, something else has to happen: there has to be a wide deployment of web services. […]*
- *applications will no longer be a big chunk of software that runs on a computer but a combination of web services*

and the "silver bullet" became…

15

- *Computing is becoming a utility and software a service. This will profoundly change the economics of the IT industry. […]*
- *For software truly to become a service, something else has to happen: there has to be a wide deployment of web services. […]*
- *applications will no longer be a big chunk of software that runs on a computer but a combination of web services*

VALENCIA 2005 University of Leicester

and the "silver bullet" became…

- *Computing is becoming a utility and software a service. This will profoundly change the economics of the IT industry. […]*
- *For software truly to become a service, something else has to happen: there has to be a wide deployment of web services. […]*
- *applications will no longer be a big chunk of software that runs on a computer but a combination of web services*

The Economist, May 10, 2003

Yet a case of "complexity"?

in-the-head	in-the-small	**in-the-large**
mnemonics	I/O specs	complex specs
result-driven	algorithms	system modules
symbolic information	data structures and types	databases, persistence
elementary control flow	execute once termination	continuous execution

- **"One man and his problem…"** (but **their programs**)
- The Science of **Software Specification and Design**
- **Engineering**

Yet a case of "complexity"?

16

in-the-head	in-the-small	**in-the-large**
mnemonics	I/O specs	complex specs
result-driven	algorithms	system modules
symbolic information	data structures and types	databases, persistence
elementary control flow	execute once termination	continuous execution

- "One man and his problem…" (but their programs)
- **"One man and everybody's problems…"**

VALENCIA 2005 University of Leicester

A case of "complexity"

in-the-head	in-the-small	in-the-large	**in-the-world**
mnemonics	I/O specs	complex specs	**evolving**
result-driven	algorithms	system modules	**sub-systems & interactions**
symbolic information	data structures and types	databases, persistence	**separation data computation**
elementary control flow	execute once termination	continuous execution	**distribution & coordination**

Same complexity?

Same complexity?

- **"Physiological" complexity**
  - derives from the need to account for problems/situations that are "complicated" in the sense that they offer great difficulty in understanding, solving, or explaining
  - there is nothing necessarily wrong or faulty in them; they are just the unavoidable result of a necessary combination of parts or factors

Same complexity?

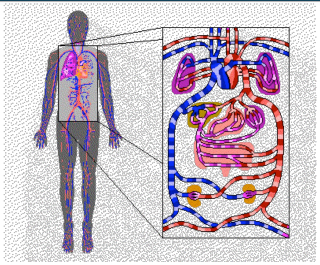
- **"Physiological" complexity**
  - derives from the need to account for problems/situations that are "complicated" in the sense that they offer great difficulty in understanding, solving, or explaining
  - there is nothing necessarily wrong or faulty in them; they are just the unavoidable result of a necessary combination of parts or factors
- **"Social" complexity**
  - derives from the number and "open" nature of interactions that involve "autonomic" parts of a system;
  - it is almost impossible to predict what properties can emerge and how they will evolve as a result of the interactions in place or the dynamics of the population itself.

Same Science & Engineering?

- **"Physiological" complexity**
  - server-to-server, static, linear interaction based on identities
  - compile or design time integration
  - **architectures of usage**
  - **product structure**

Same Science & Engineering?

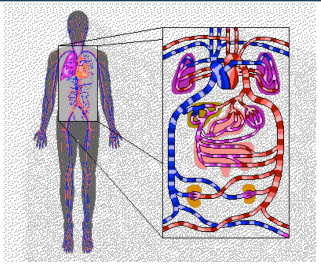
- "Physiological" complexity
  - server-to-server, static, linear interaction based on identities
  - compile or design time integration
  - **architectures of usage**
  - **product structure**



Same Science & Engineering?

19

- "Physiological" complexity
  - server-to-server, static, linear interaction based on identities
  - compile or design time integration
  - **architectures of usage**
  - **product structure**
- "Social" complexity
  - dynamic, mobile and unpredictable interactions based on properties
  - "late" or "just-in-time" integration
  - **contracts of interaction**
  - **evolving structure**



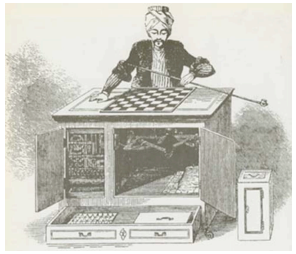
VALENCIA 2005

University of Leicester

Same Science & Engineering?

19

- "Physiological" complexity
  - server-to-server, static, linear interaction based on identities
  - compile or design time integration
  - **architectures of usage**
  - **product structure**
- "Social" complexity
  - dynamic, mobile and unpredictable interactions based on properties
  - "late" or "just-in-time" integration
  - **contracts of interaction**
  - **evolving structure**



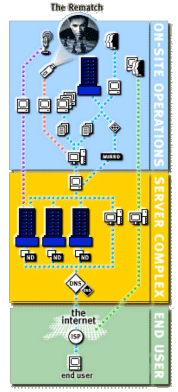
VALENCIA 2005

University of Leicester

Same Science & Engineering?

19

- "Physiological" complexity
  - server-to-server, static, linear interaction based on identities
  - compile or design time integration
  - **architectures of usage**
  - **product structure**
- "Social" complexity
  - dynamic, mobile and unpredictable interactions based on properties
  - "late" or "just-in-time" integration
  - **contracts of interaction**
  - **evolving structure**



VALENCIA 2005

University of Leicester

Two different relationships

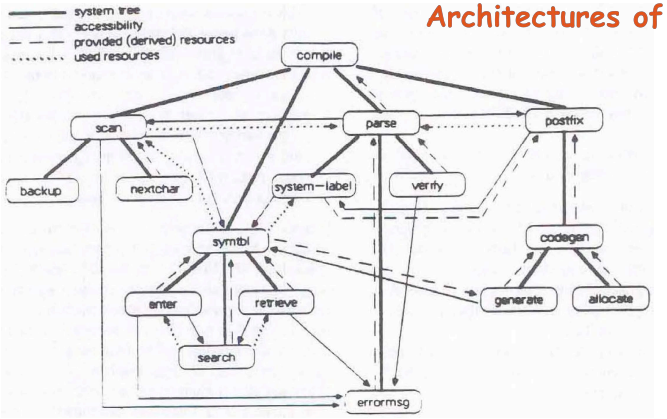
- **Implements**
  - a given module is defined in terms of facilities provided by/to other modules;
  - composition mechanisms glue pieces together by indicating for each use of a facility where its corresponding definition is provided
- **Interacts**
  - components are treated as independent entities that may interact with each other along well defined lines of communication (connectors)

Module structures

- Code/implementation structures
- Address the global structure of a system in terms of
  - what its modules and resources are
  - how they fit together in the system
  - Definition/usage graphs
- **Modelling Interconnection Languages** for programming-in-the-large (DeRemer and Kron 75)

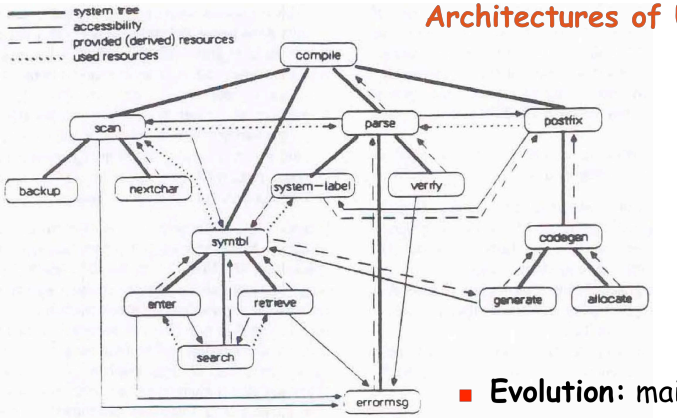
Module structures

Architectures of Usage



Module structures

Architectures of Usage



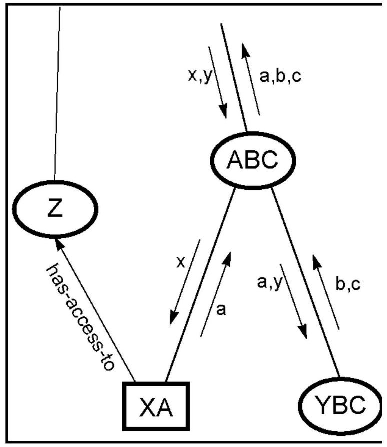
- **Evolution:** maintenanc

Module Interconnection Languages

- Address the task of integrating independently-developed subsystems, which, in the 1970s, became increasingly difficult as software systems increased in size and complexity
- The first MIL, MIL75, was described by DeRemer and Kron who argued about the differences between programming in the small and programming in the large, for which a MIL is required for knitting modules together.
- Another early MIL is MESA developed during 1975 at Xerox PARC

Example

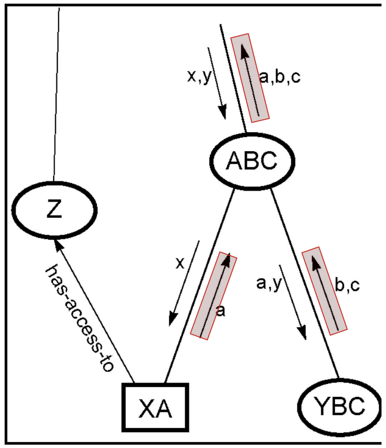
module *ABC*
  provides a, b, c
  requires x, y
  consists-of
    function XA, **module** YBC
  function XA
    must-provide a
    requires x
    has-access-to module Z
    real x, **integer** a
  end XA
  module YBC
    must-provide b, c
    requires a, y
    real y, **integer** a, b, c
    end YBC
end *ABC*



Example

23

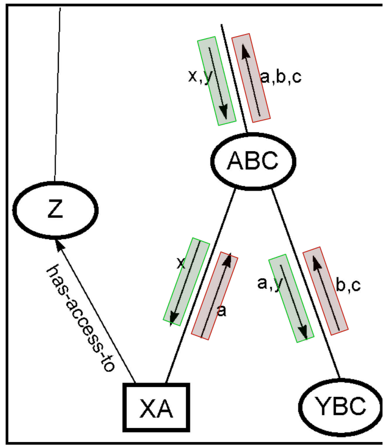
module *ABC*
  provides a, b, c
  requires x, y
  consists-of
    function XA, **module** YBC
  function XA
    must-provide a
    requires x
    has-access-to module Z
    real x, **integer** a
  end XA
  module YBC
    must-provide b, c
    requires a, y
    real y, **integer** a, b, c
    end YBC
end *ABC*



Example

23

module *ABC*
  provides a, b, c
  requires x, y
  consists-of
    function XA, **module** YBC
  function XA
    must-provide a
    requires x
    has-access-to module Z
    real x, **integer** a
  end XA
  module YBC
    must-provide b, c
    requires a, y
    real y, **integer** a, b, c
    end YBC
end *ABC*



Run-time Architectures

- The Components&Connectors view
- The "Interacts" relationship
- One generation later
  - Perry and Wolf (92)
  - Shaw and Garlan (96)
  - Bass, Clements, Kazman (98)
- Partly inspired by (civil) architects (Alexander)

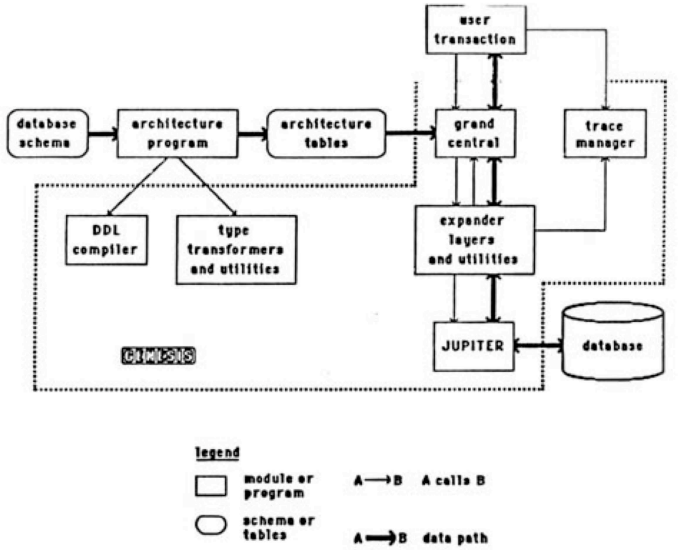


Figure 3.1 The Configuration of the GENESIS Prototype

Genesis: A Reconfiguration Database Management System, D. S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, T.E. Wise, Department of Computer Sciences, University of Texas at Austin,

Reasoning

- **Implements**
  - typically proceeds hierarchically;
  - the correctness of one module depends on the correctness of the modules that it uses
- **Interacts**
  - the correctness of each module is independent of the correctness of the other modules with which it interacts;
  - the behaviour of the aggregate system depends on the behaviour of its constituent modules *and the way they interact*.

Abstraction

- **Implements**
  - it is usually sufficient to adopt the primitives of the underlying programming language (e.g. procedure call and data sharing);
  - how a component achieves its computation
- **Interacts**
  - often involve abstractions not directly provided by the underlying programming language: pipes, filters, event broadcast, client-server protocols, etc;
  - how the computation achieved by each component is combined with the others in the overall system

Compatibility

- **Implements**
  - type checking is used to determine if a use of a facility matches its definition
- **Interacts**
  - interest is in whether protocols of communication are respected (e.g. is the server initialised before a client makes a request of it)

What is it then?

- the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

source: ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems

What is it then?

- architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design.

source: D.E.Perry and A.L.Wolf.
Foundations for the Study of Software Architectures.
ACM SIGSOFT Software Engineering Notes, October 1992.

What is it then?

- A software architecture for a system is the structure or structures of the system, which comprise elements, their externally-visible behavior, and the relationships among them.

source: L.Bass, P.Clements and R.Kazman.
Software Architecture in Practice. Addison-Wesley, 1998.

What is it then?

- visit
  www.sei.cmu.edu/architecture/definitions.html

you can add your own definition
if you don't like any of the 100 (or so…)

The key architectural concepts

Components

- units of computation or data stores
  - what Perry&Wolf call *processing* and *data elements*
- define the locus of computation and state
  - filters, databases, objects, ADTs, clients, servers, …
- may be simple or composite
  - composite components describe a (sub)system;
  - architectures consisting of composite components describe systems of systems

The key architectural concepts

Connectors

- architectural elements that model
  - interactions among components
  - rules that govern those interactions
- simple interactions
  - procedure calls
  - shared variable access
- complex and semantically-rich interactions
  - client-server or database access protocols
  - asynchronous event multi-cast
  - piped data streams

The key architectural concepts

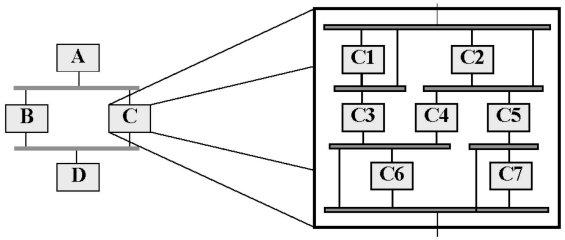
Properties

- specify information for construction & analysis
  - signatures,
  - pre/post conditions,
  - RT specs,
  - protocols,
  - performance attributes

The key architectural concepts

Configurations or topologies

- connected graphs of components and connectors that describe architectural structures
- composite components are configurations



and…

An **architectural style** is defined by:

- **Component/connector types** – the vocabulary of architectural building blocks
- **Constraints** on how the building blocks can be used, including
  - topological rules
  - interface standards
  - required properties

A style defines a **family** of architecture instances

A taxonomy (Shaw&Garlan)

- **Data Flow**
  - Batch sequential
  - Dataflow network (pipes&filters)
  - Closed loop control
- **Call-and-return**
  - Main program/subroutines
  - Information hiding
- **Interacting processes**
  - Communicating processes
  - Event systems
- **Data-oriented repository**
  - Transactional databases
  - Blackboard
  - Modern compiler
- **Data-sharing**
  - Compound documents
  - Hypertext
  - Fortran COMMON
  - LW processes
- **Hierarchical**
  - Layered

The challenge of evolution

- Reflect on the (run-time) architecture of the system the different levels of change that can take place in the application domain.
- Support evolution through dynamic reconfiguration, without interruption of service, minimising impact on the global system.

Coping with change

- In Business Systems today, change is the rule of the game…

Coping with change

40

- In Business Systems today, change is the rule of the game…

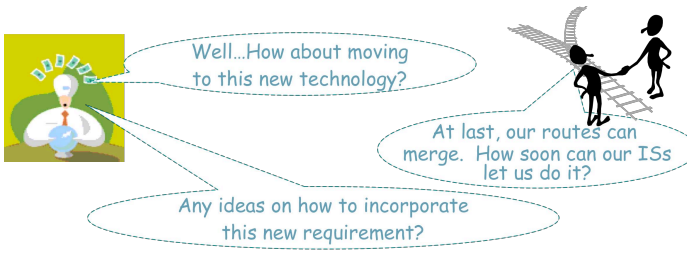
> "… the ability to change is now more important than the ability to create [e-commerce] systems in the first place. Change becomes a first-class design goal and requires business and technology architecture whose components can be added, modified, replaced and reconfigured".
>
> P.Finger, "Component-Based Frameworks for E-Commerce", Communications of the ACM 43(10), 2000, 61-66.

VALENCIA 2005 University of Leicester

Coping with change

40

- In Business Systems today, change is the rule of the game…

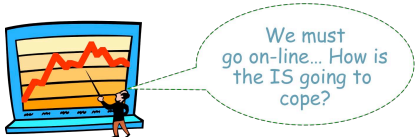


VALENCIA 2005 University of Leicester

Coping with change

40

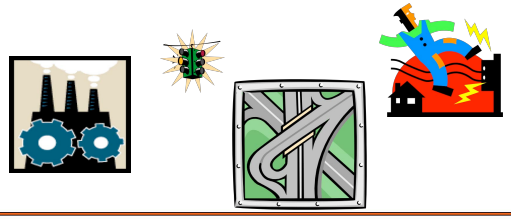
- In Business Systems today, change is the rule of the game…
- The Web is only fuelling the rate of change… (B2C, B2B, P2P,…)



VALENCIA 2005 University of Leicester

Coping with change

- In Business Systems today, change is the rule of the game…
- The Web is only fuelling the rate of change… (B2C, B2B, P2P,…)
- Critical infrastructures depend on the ability to react to failure by reconfiguring themselves (self-healing)…



Coping with change

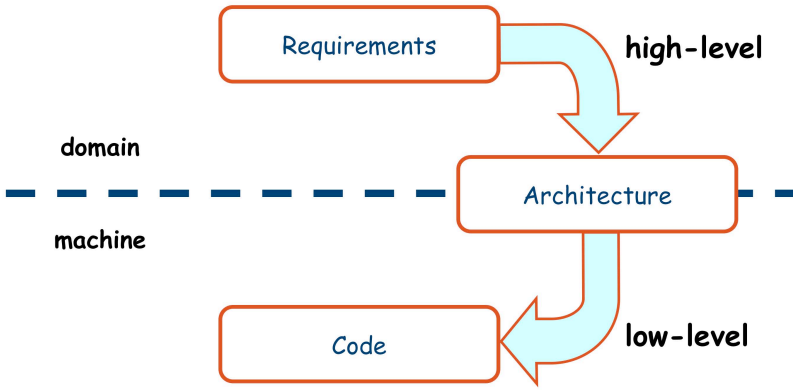
40

- In Business Systems today, change is the rule of the game…
- The Web is only fuelling the rate of change… (B2C, B2B, P2P,…)
- Critical infrastructures depend on the ability to react to failure by reconfiguring themselves (self-healing)…
- The real-time economy…

Complexity is now on evolution…

VALENCIA 2005 University of Leicester

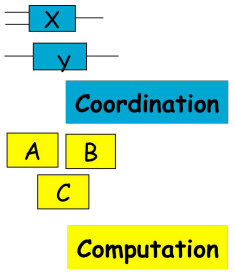
Architectures in Software Design



VALENCIA 2005 University of Leicester

Summary

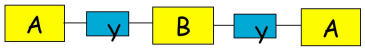
Architecture-based approaches

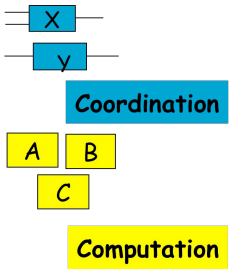


VALENCIA 2005 University of Leicester

Summary

Architecture-based approaches



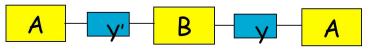


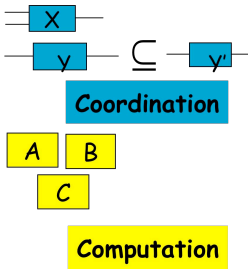
Compositionality wrt refinement

Summary

42

Architecture-based approaches





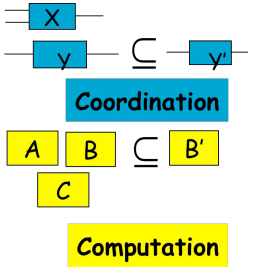
Compositionality wrt refinement

VALENCIA 2005
University of Leicester

Summary

42

Architecture-based approaches



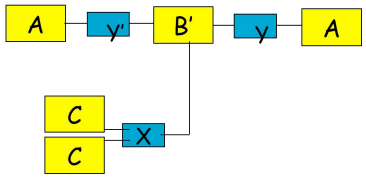
Compositionality wrt refinement

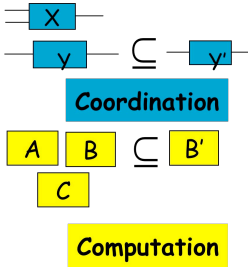
VALENCIA 2005
University of Leicester

Summary

42

Architecture-based approaches





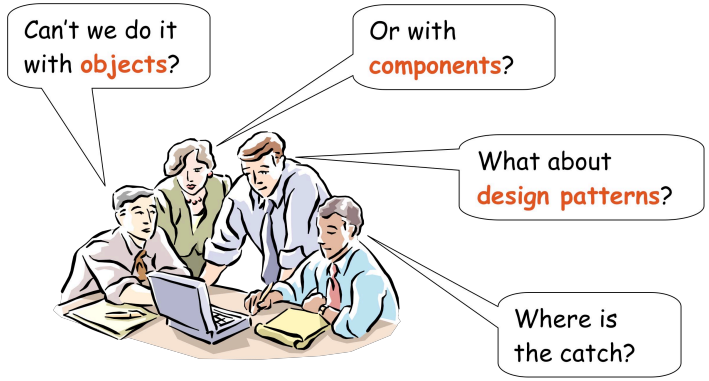
Compositionality wrt refinement
wrt evolution

VALENCIA 2005
University of Leicester

Some FAQs...

Some FAQs...

43



VALENCIA 2005
University of Leicester

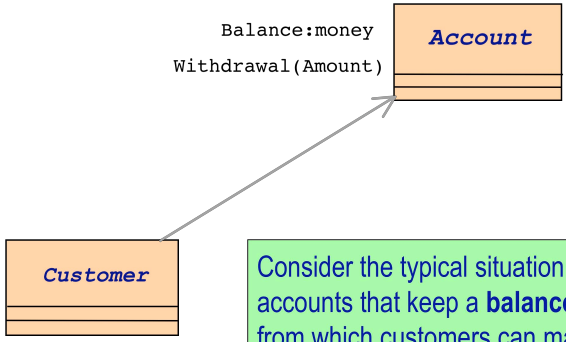
An example from Banking

Consider the typical situation of bank accounts that keep a **balance** and from which customers can make **withdrawals**.

VALENCIA 2005
University of Leicester

An example from Banking

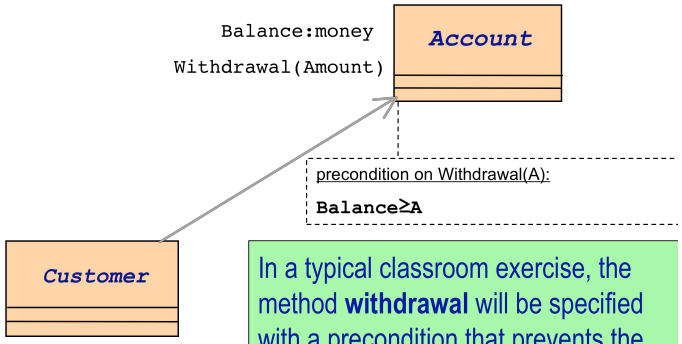
44



Consider the typical situation of bank accounts that keep a **balance** and from which customers can make **withdrawals**.

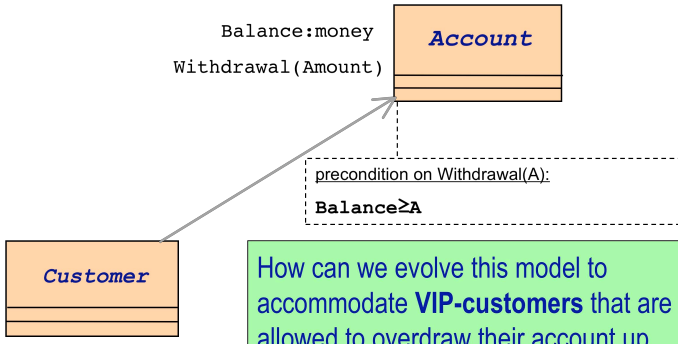
VALENCIA 2005
University of Leicester

An example from Banking



In a typical classroom exercise, the method **withdrawal** will be specified with a precondition that prevents the account from being overdrawn.

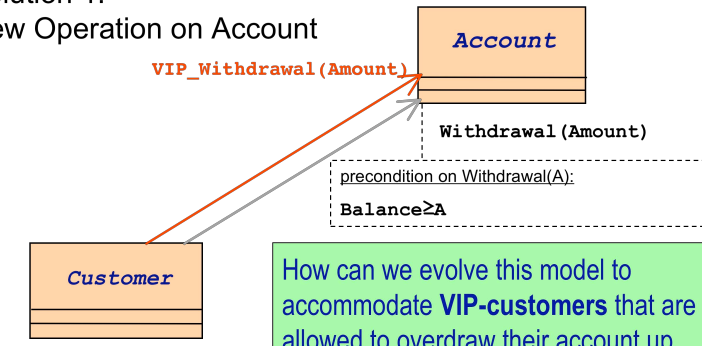
An example from Banking



How can we evolve this model to accommodate **VIP-customers** that are allowed to overdraw their account up to an agreed limit?

An example from Banking

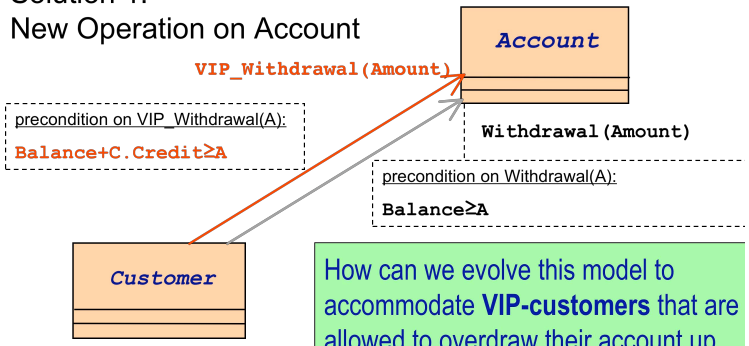
- Solution 1:
  New Operation on Account



How can we evolve this model to accommodate **VIP-customers** that are allowed to overdraw their account up to an agreed limit?

An example from Banking

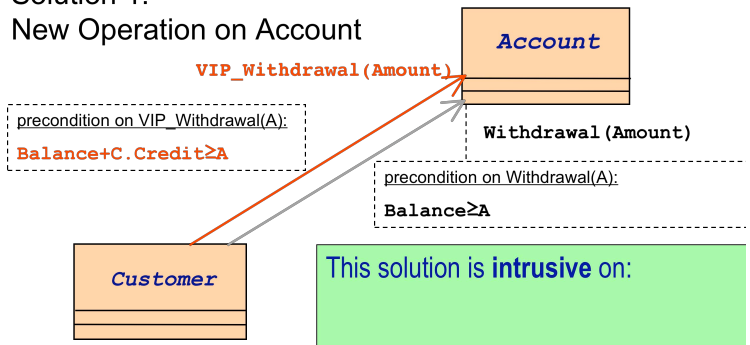
- Solution 1:
  New Operation on Account



How can we evolve this model to accommodate **VIP-customers** that are allowed to overdraw their account up to an agreed limit?

An example from Banking

- Solution 1:
  New Operation on Account



This solution is **intrusive** on:

An example from Banking

44

- Solution 1:
  New Operation on Account

This solution is **intrusive** on:

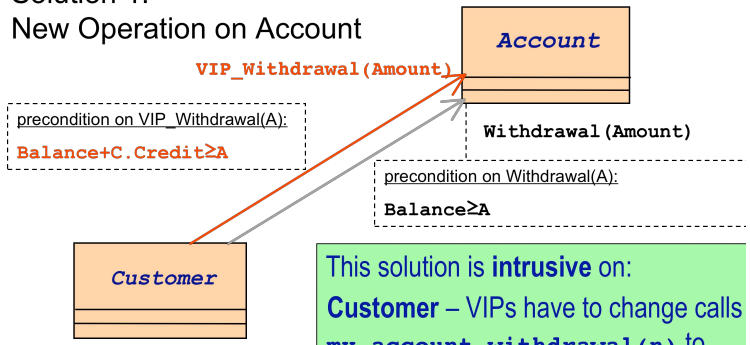
Account – its interface and implementation needs to be extended with the new operation

VALENCIA 2005
University of Leicester

An example from Banking

44

- Solution 1:
  New Operation on Account



This solution is **intrusive** on:

Customer – VIPs have to change calls `my_account.withdrawal(n)` to `my_account.VIP_withdrawal(n)`

VALENCIA 2005
University of Leicester

An example from Banking

44

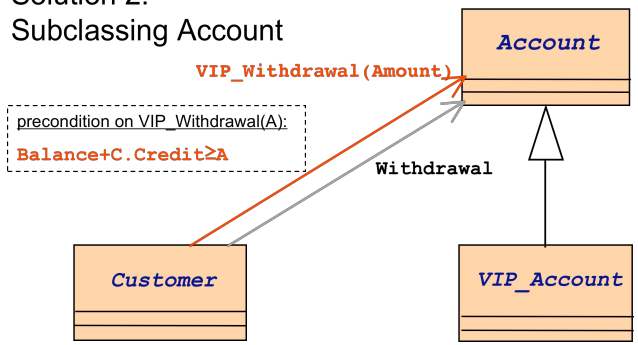
- Solution 1:
  New Operation on Account

Typically, a different solution would have been chosen…

VALENCIA 2005
University of Leicester

An example from Banking

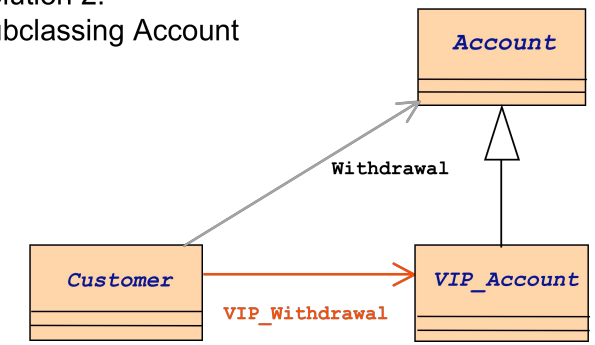
- Solution 2:
  Subclassing Account



An example from Banking

44

- Solution 2:
  Subclassing Account



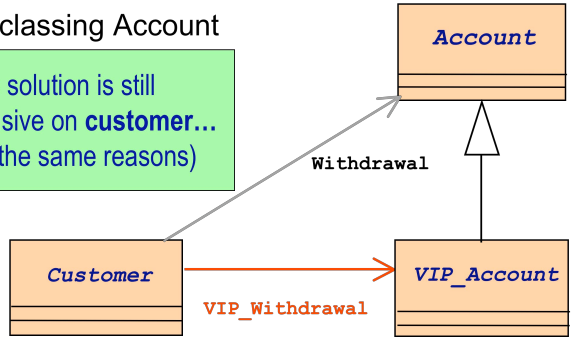
VALENCIA 2005
University of Leicester

An example from Banking

44

- Solution 2:
  Subclassing Account

This solution is still intrusive on **customer...** (for the same reasons)



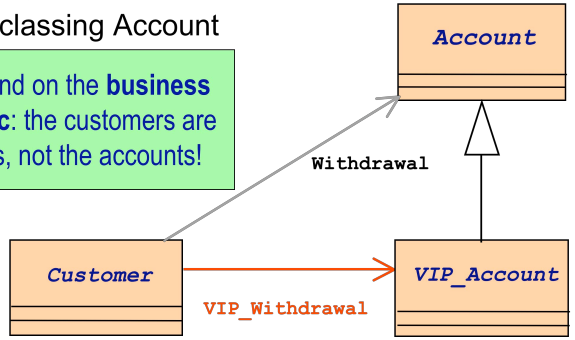
VALENCIA 2005
University of Leicester

An example from Banking

44

- Solution 2:
  Subclassing Account

... and on the **business logic**: the customers are VIPs, not the accounts!



VALENCIA 2005
University of Leicester

An example from Banking

- Problems with Solutions 1,2

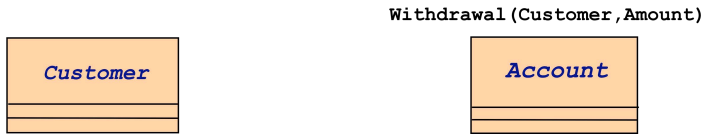
An example from Banking

45

- Problems with Solutions 1,2
  - They are intrusive on the code…
  - …and on the interconnections
  - …and on the business logic
- This is because, through clientship, business rules get **encoded** in the methods, and the methods reside in the server side.

VALENCIA 2005 University of Leicester

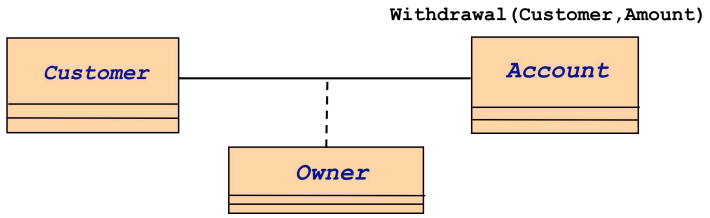
A better solution



A better solution

46

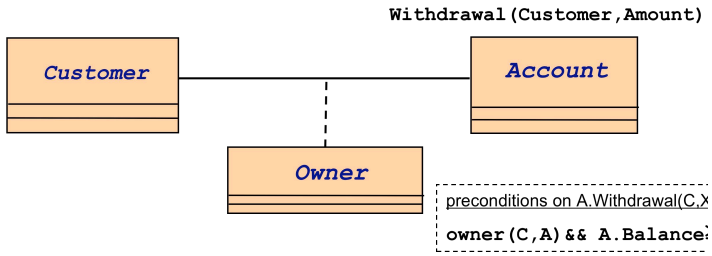
Model the relationship between customers and accounts as an **association class**…



VALENCIA 2005 University of Leicester

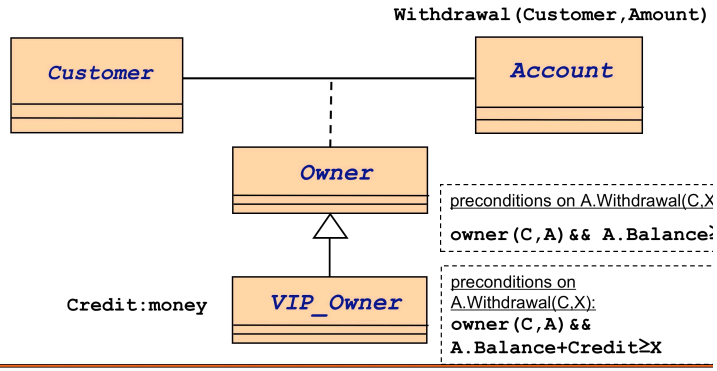
A better solution

… on which the “business rule” can be placed



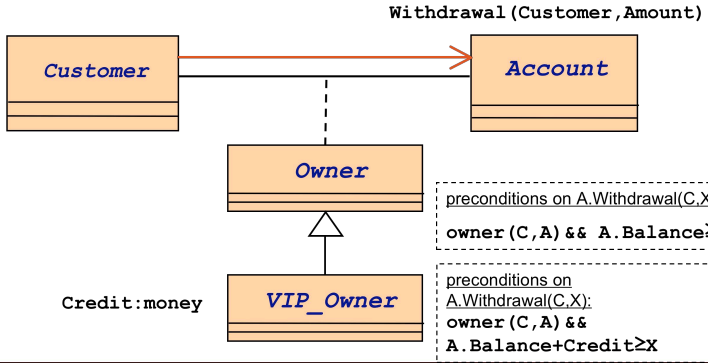
A better solution

… and specialise it to **evolve** the business rule



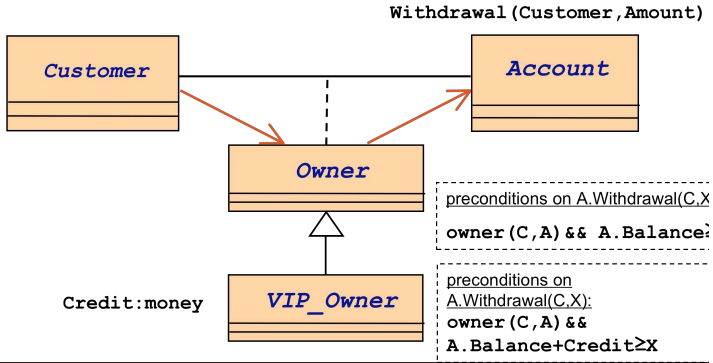
But still some problems

If the association is implemented through attributes and direct calls, we get the same problems as before…



But still some problems

A better way is to use a **mediator** through which the calls can be redirected and managed…



Mediator

- Problems with the mediator

Mediator

47

- Problems with the mediator
  - Each mediation is intrusive on the code…
  - …because it is managed explicitly by the objects involved
  - This also means that it can be interrupted
  - …and even by-passed
  - On the other hand, additional business rules means additional mediators and mediation between them…

VALENCIA 2005 University of Leicester

OO is identity-based

- What is intrinsically "wrong" with OO:

OO is identity-based

48

- What is intrinsically "wrong" with OO:
  - Feature calling, the basic mechanism through which object can interact, is **identity-based**: objects call specific features of specific objects (clientship);
  - As a result, any change on the interactions is **intrusive** on the code of the object.

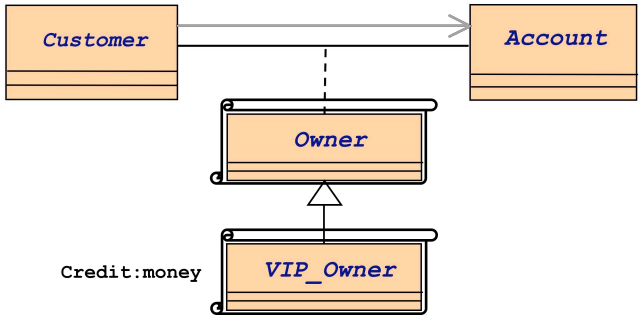
VALENCIA 2005 University of Leicester

OO is identity-based

- What is intrinsically "wrong" with OO:
  - Feature calling, the basic mechanism through which object can interact, is **identity-based**: objects call specific features of specific objects (clientship);
  - As a result, any change on the interactions is **intrusive** on the code of the object.
- We propose a way for interactions to be **externalis** and handled as first-class citizens.

Contracts for Change

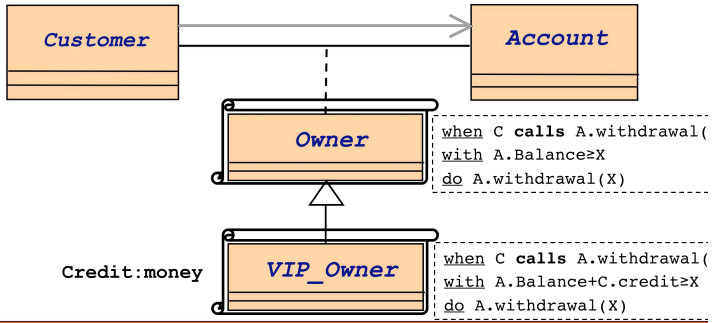
- A solution inspired on Architectural Connectors



Contracts for Change

- A solution inspired on Architectural Connectors

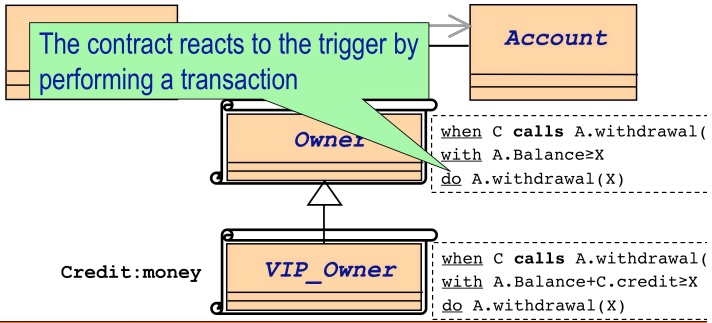
The customer still calls the account but the call is intercepted by the contract, without any of the parties being aware…



Contracts for Change

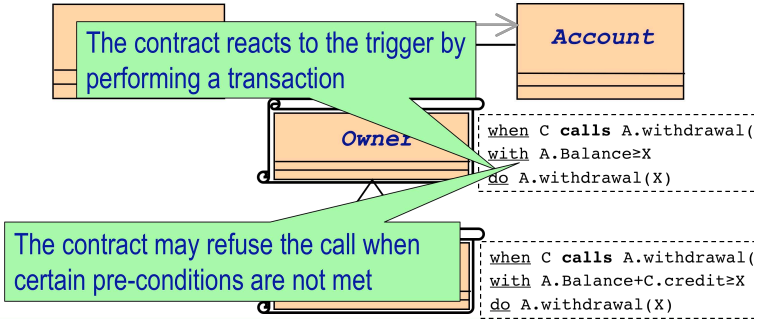
- A solution inspired on Architectural Connectors

The customer still calls the account but the call is intercepted by the contract, without any of the parties being aware…



- A solution inspired on Architectural Connectors

The customer still calls the account but the call is intercepted by the contract, without any of the parties being aware…



The contract reacts to the trigger by performing a transaction

The contract may refuse the call when certain pre-conditions are not met

2

The Coordination Dimension

The CCC approach

- **A confluence of contributions from**
  - Coordination Languages and Models
    Separation between "computation" and "coordination"
  - Software Architectures
    Connectors as first-class citizens
  - Parallel Program Design
    Superposition
- **An Academia/Industry partnership**

ATX SOFTWARE ⟺ / University of Leicester

The CCC approach

- **The Strategy**

The CCC approach

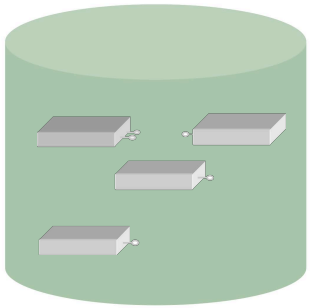
3

- **The Strategy**

- Recognize that change in the application domain occurs at different levels;

The CCC approach

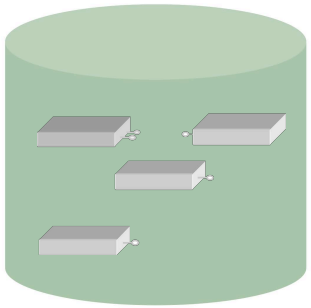
- Distinguish Computation Resources…



The CCC approach

4

- Distinguish Computation Resources…



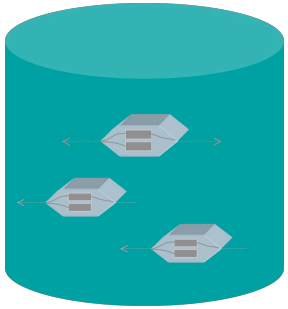
- Units that model **core** business/domain entities and provide services through computations performed **locally**
- These tend to be **stable** components, for which modifications imply major re-engineering

VALENCIA 2005

University of Leicester

The CCC approach

- …from Coordination Resources



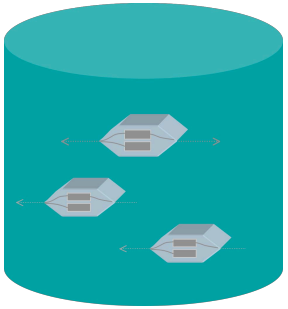
VALENCIA 2005

University of Leicester

The CCC approach

5

- …from Coordination Resources



Units that model **volatile** "business" rules and processes and can be **superposed, at run time**, on the core units to…

- …**coordinate** their interactions
- …**regulate** their behaviour
- …**adapt** their behaviour
- …**monitor** their behaviour

VALENCIA 2005

University of Leicester

The CCC approach

- **The Strategy**
  - Recognize that change in the application domain occurs at different levels;

The CCC approach

6

- **The Strategy**
  - Recognize that change in the application domain occurs at different levels;
  - Reflect these levels in the architecture of the system;

VALENCIA 2005 University of Leicester

The CCC approach

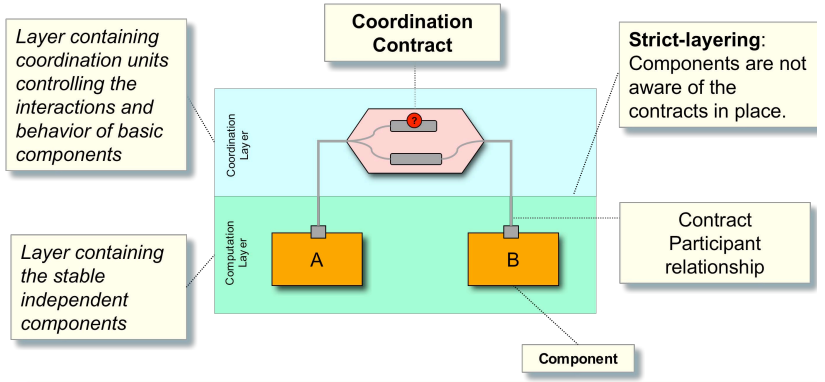
- Change-oriented layered architecture

VALENCIA 2005 University of Leicester

The CCC approach

7

- Change-oriented layered architecture



VALENCIA 2005 University of Leicester

The Strategy

- Recognize that change in the application domain occurs at different levels;
- Reflect these levels in the architecture of the system;

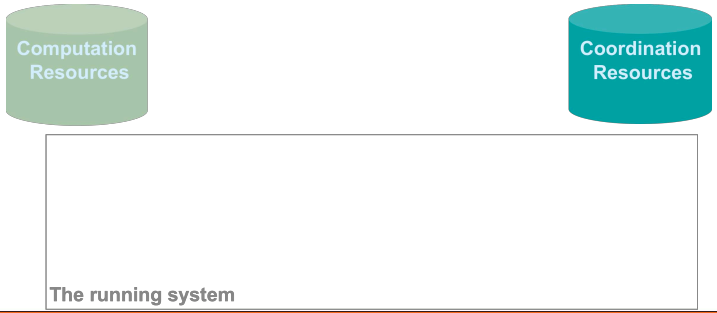
The CCC approach 8

The Strategy

- Recognize that change in the application domain occurs at different levels;
- Reflect these levels in the architecture of the system;
- Manage evolution according to the architecture.

VALENCIA 2005 University of Leicester

The Configuration Layer

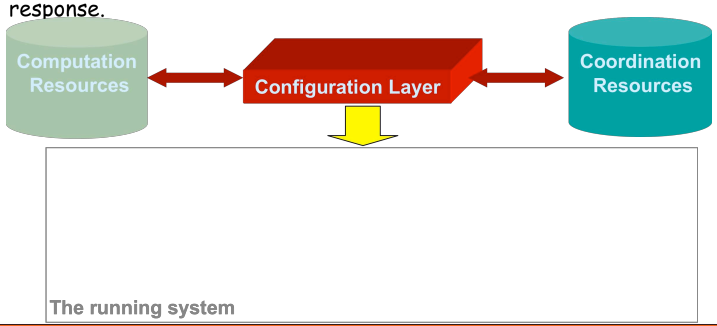


VALENCIA 2005 University of Leicester

The CCC approach 9

The Configuration Layer

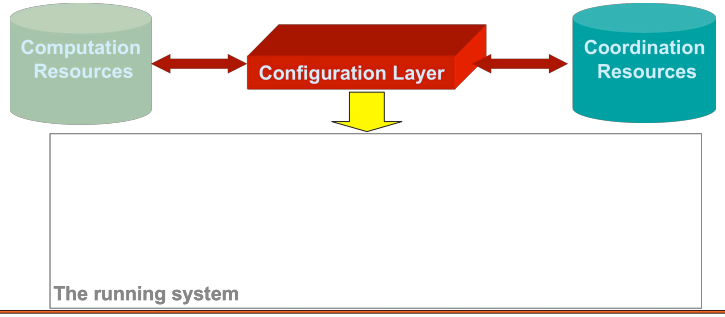
Services that model business activities and through which the system can be configured, at run-time, to provide the required response.



VALENCIA 2005 University of Leicester

- The Configuration Layer

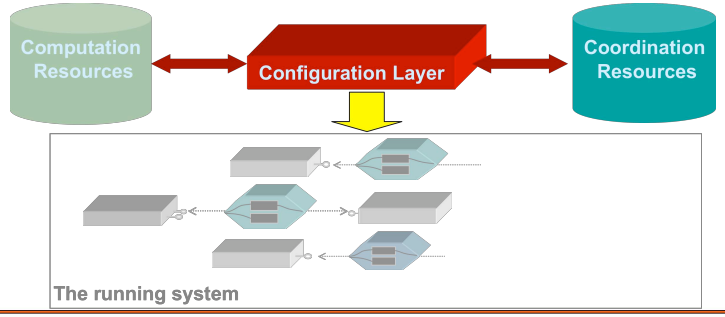
These services can be either invoked by authorized users or triggered by events (self-adaptation).



The CCC approach 9

- The Configuration Layer

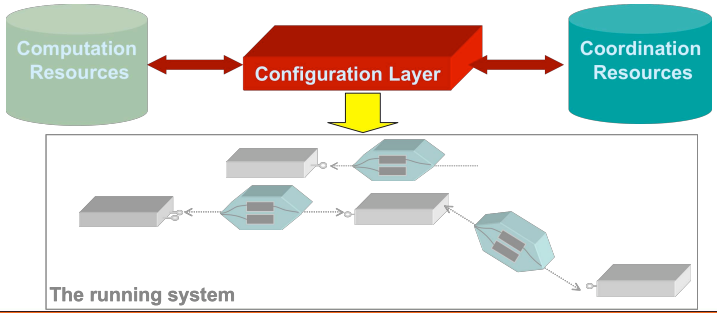
These services can be either invoked by authorized users or triggered by events (self-adaptation).



The CCC approach 9

- The Configuration Layer

These services can be either invoked by authorized users or triggered by events (self-adaptation).



The CCC approach

- Semantic primitives for **Coordination**
- Semantic primitives for **Configuration**
- Full mathematical **semantics** – CommUnity
- A **micro-architecture** for deployment over platforms for component-based development
- An instantiation of this micro-architecture for Java components – the **Coordination Development Environment (CDE)**

Semantic primitives for coordination

- **Coordination laws** that provide abstract models of services in terms of reactions to be performed upon detection of triggers.
- **Coordination interfaces** that identify the types of components that can instantiate the service as a law.

Overview


```
coordination law standard-withdrawal
partners
     a:account-debit;
     c:customer-withdrawal
rules
          when  c.withdrawal(n,a)
          with  a.balance()≥n & c.owns(a)
          do    a.debit(n);
end law
```

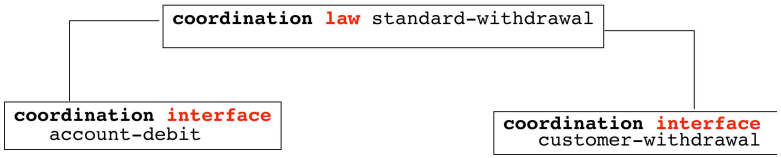

Overview


```
coordination law standard-withdrawal
partners
     a:account-debit;
     c:customer-withdrawal
rules
          when  c.withdrawal(n,a)
          with  a.balance()≥n & c.owns(a)
          do    a.debit(n);
end law
```

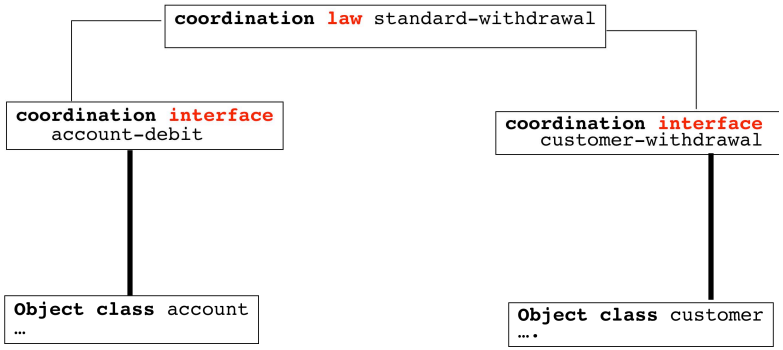
```
coordination interface
   account-debit
import types
   money;
services
   balance():money;
   debit(a:money): post balance()
       = old balance()-a
end interface
```

```
coordination interface
   customer-withdrawal
import types
   money, account;
services
   owns(a:account):Boolean
events
   withdrawal(n:money;a:account
end interface
```

Just-in-time integration

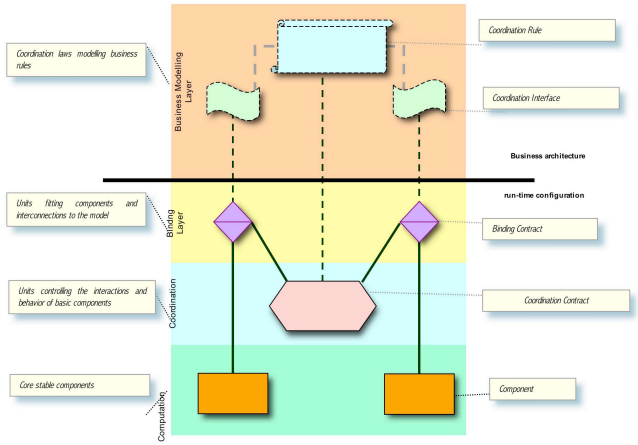


Just-in-time integration



Binding may require **adaptation**…

Instantiation



Coordination interfaces

- **Coordination interfaces** correspond to the roles of architectural connectors.
- They identify types of components according to services and events:
  - ***services*** identify operations that components that are instances of the interface need to provide for a contract to operate according to the law;
  - ***events*** identify situations produced during the execution of the components that are required to be detected as triggers for the contract to react and activate a *coordination rule* as discussed below.

Coordination interfaces


```
coordination interface customer-withdrawal
import types     money, account
services    owns(a:account):Boolean
events    withdrawal(n:money;a:account)
end interface
```

- We require to detect as triggers events that consist of customers performing withdrawals, and be provided with services that query about the account ownership relation
- In traditional object-oriented modelling, typical events are feature calls: a withdrawal would normally be modelled as a direct call to the debit operation of the corresponding account – a.debit(n).

Coordination interfaces


```
coordination interface account-debit
import types     money
services
       balance():money
       debit(n:money)   post balance()= old balance – a
end interface
```

- The inclusion of properties, e.g. pre and post-conditions on services, provide means for requirements to be specified on the components that can be bound to the interface.
- A special section **properties** may be used for other kinds of requirements.

Coordination laws


```
coordination law standard-withdrawal
partners     a:account-debit; c:customer-withdrawal
rules  when   c.withdrawal(n,a)
       with   a.balance()≥n and c.owns(a)
       do     a.debit(n)
end law
```

- A coordination law corresponds to a connector (type).
- The partners are logical parameters typed by coordination interfaces and correspond to the connector's roles.
- The coordination rules provide the glue of the connector.

Coordination rules


```
coordination law standard-withdrawal
partners     a:account-debit; c:customer-withdrawal
rules  when   c.withdrawal(n,a)
       with   a.balance()≥n and c.owns(a)
       do     a.debit(n)
end law
```

- Each coordination rule identifies, under **when**, a trigger to which the contracts that instantiate the law will react – a request by the customer for a withdrawal in the case at hand.
- The trigger can be just an event observed directly over one of the partners or a more complex condition built from one or more events.

Coordination rules


```
coordination law standard-withdrawal
partners    a:account-debit; c:customer-withdrawal
rules  when  c.withdrawal(n,a)
       with  a.balance()≥n and c.owns(a)
       do    a.debit(n)
end law
```

- Under **with** we include conditions (guards) that should be observed for the reaction to be performed.
- If any of the conditions fails, the reaction is not performed and the occurrence of the trigger fails.
- Failure is handled through whatever mechanisms are provided by the language used for deployment.

Coordination rules


```
coordination law standard-withdrawal
partners    a:account-debit; c:customer-withdrawal
rules  when  c.withdrawal(n,a)
       with  a.balance()≥n and c.owns(a)
       do    a.debit(n)
end law
```

- The reaction to be performed to occurrences of the trigger is identified under **do** as a set of operations – a debit for the amount and on the account identified in the trigger.
- This set may include services provided by one or more of the partners as well as operations that are proper to the law itself.
- The whole interaction is handled as a single transaction.

Example: VIP-withdrawal


```
coordination law VIP-withdrawal
partners    a:account-debit; c:customer-withdrawal
operation   credit():money
rules  when  c.withdrawal(n,a)
       with  a.balance()+credit()≥n and c.owns(a)
       do    a.debit(n)
end law
```

- The credit-limit is assigned to the law itself rather than the customer or the account. This is because we may want to be able to assign different credit limits to the same customer but for different accounts, or for the same account but for different owners.
- Would it make sense to have a separate partner of the law providing the credit?

Interfacing with external events

- Coordination interfaces can also act as useful abstractions for either events or services that lie outside the system, or global phenomena that cannot be localised in specific components.
- In the case of events, this allows for the definition of reactions that the system should be able to perform to triggers that are either global (e.g. a deadline) or are detected outside the system.
- In the case of reactions, this allows us to identify services that should be procured externally. This is particularly useful for B2B operations and the modelling of Web-services.

Example: interfaces for transfers


```
coordination interface external-transfer
import types    money, account, transfer-id
events   transfer(n:money;a:account;t: transfer-id)
end interface
```

```
coordination interface account-credit
import types    money
Services     credit(n:money)
end interface
```


Example: law for transfers


```
coordination law external-transfer-handler
partners    a:account-credit; t:external-transfer
operation   ackn(t:transfer-id)
rules  when  transfer(n,a,t)
       with  a.exists
       do    n≥1000:a.credit(n-100)
             and n<1000:a.credit(n*0.9)
             and ackn(t)
end law
```


Monitoring behaviour

- Assume that new legislation is passed that requires credits over a certain amount to be reported to the central bank – e.g. as a means of detecting money laundering.
- Rather than revise the implementation of credits to take care of this new requirement, it is better to superpose a contract over every account to perform the required monitoring activity.

Example: monitoring big credits


```
coordination law report-big-credits
partners      a:account-credit-event
operation     big():money;
              report(n:money);
              set-big(n:money) post big()=n
rules  when   a.credit(n) and n≥big()
       do     report(n)
end law
```

```
coordination interface account-credit-event
import types     money
events          credit(n:money)
end interface
```

Regulating behaviour

- Contracts can also be used for superposing *regulators* over certain components of the system.
- For instance, consider the situation in which the bank decides to penalise customers who fail to keep a given minimum average balance by charging a monthly commission.

Example: commission on balances


```
coordination law commission-on-low-balance
partners      a:average-balance
operation     minimum(), charge():money
rules
       when  end-of-month
       do    minimum()>a.average():a.debit(charge())
end law
```

```
coordination interface average-balance
import types     money
services  debit(n:money); average():money
end interface
```

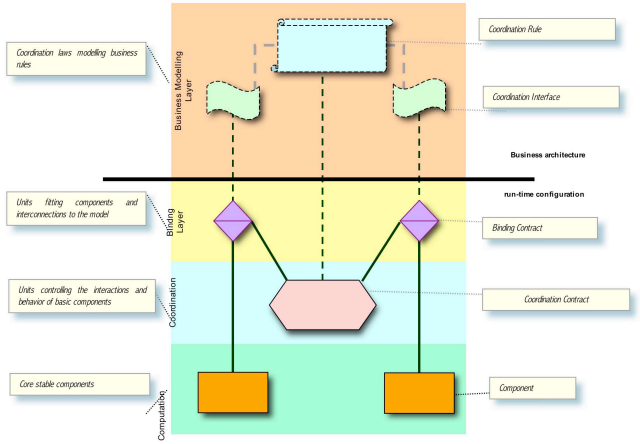

Example: openplan


```
coordination law flexible-package
partners      c,s:account-debit&credit
operation     minimum(), maximum():money
rules
       when  c.balance()<minimum()
       do    let N=min(s.balance(),maximum()-c.balance())
             in s.debit(N)and c.credit(N)

       when  c.balance()>maximum()
       do    let N=c.balance()-maximum()
             in s.credit(N)and c.debit(N)
end law
```


Example: openplan


```
coordination interface account-debit&credit
import types money
events        balance():money
services      debit(a:money);
              credit(a:money);
              balance():money
properties
       balance() after debit(a) is balance()-a;
       balance() after credit(a) is balance()+a
end interface
```

Recall



Design primitives for coordination

- **Coordination contracts** instantiate coordination laws for particular kinds of components.
- For instance, in the case of **OO programming environments**,
  - **services** are provided through methods of the classes that instantiate the partners (coordination interfaces)
  - **events** are provided by object(instance) method calls or system/class method calls

Notation


```
coordination contract standard-withdrawal
  partners x : Account; y : Customer;
  coordination
            when    y ->> x.withdrawal(z)
            with    x.balance()>z and y.owns(x)
            do      call x.withdrawal(z)
end contract
```


Notation


```
coordination contract VIP package
  partners x : Account; y : Customer;
  attributes Credit: Integer;
  coordination
      when   y ->> x.withdrawal(z)
      with   x.balance()+Credit()>z and y.owns(x)
      do     x.withdrawal(z)
end contract
```

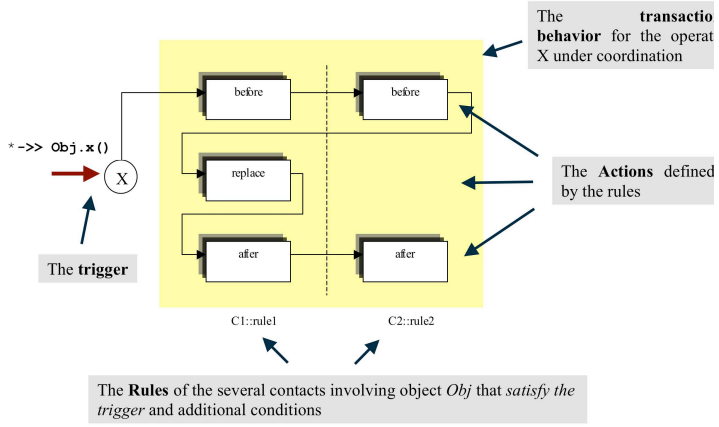
Intuitive semantics

- Instead of interacting with a mediator that delegates execution on the supplier, the client calls directly the supplier (the partners in the contract are not aware that they are being coordinated by a third party);
- The contract "*intercepts*" the call and superposes whatever forms of behaviour are prescribed;
- This means that it is not possible to bypass the coordination being imposed through the contract.

Special reactions

- When the trigger is a method call `-->Obj.x()`, the **do** may be structured in terms of:
  - **Before:** operations to be performed when the call is intercepted and before it is delegated;
  - **Replace**: operation to be performed instead of the call;
  - **After**: operations to be performed after the call is forwarded or its replacement executed

Coordination Semantics



Controlling a sluice-gate

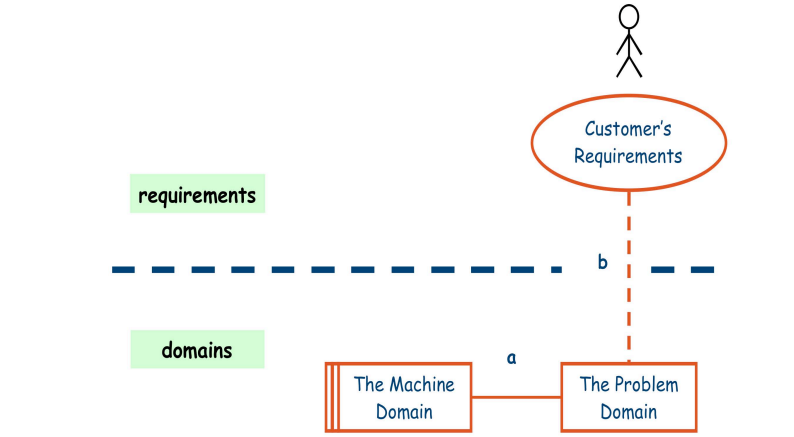
A rising and falling gate is used in an irrigation system. A computer system is needed to raise and lower the sluice gate in response to the commands of an operator.

The gate is opened and closed by rotating vertical screws controlled by a motor that accepts clockwise and anticlockwise pulses only when stopped.

There are sensors at the top and bottom of the gate travel indicating when the gate is fully **opened** and fully **shut**.

The operator **commands** are to **raise**, **lower**, or **stop** the gate.

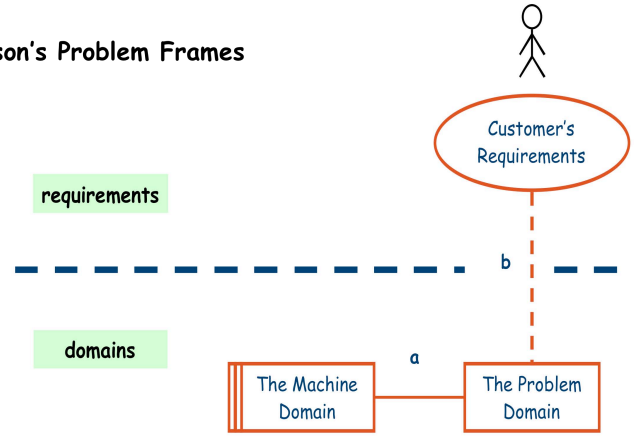
Requirements modelling



Requirements modelling

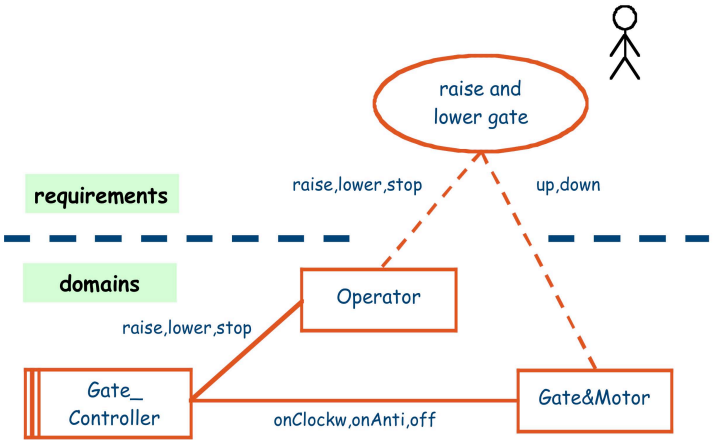
40

Jackson's Problem Frames



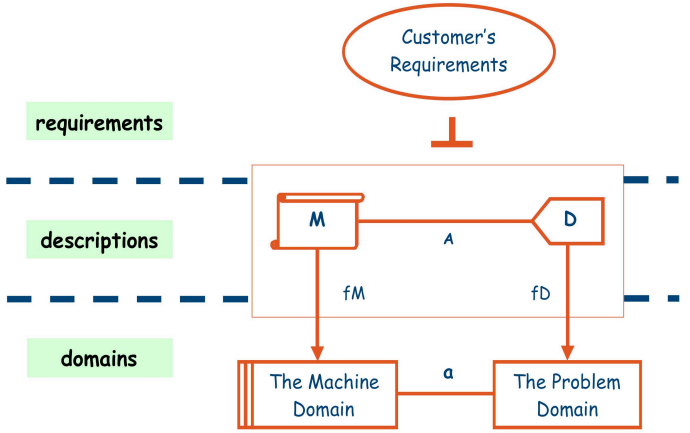
VALENCIA 2005
University of Leicester

The Problem Frame



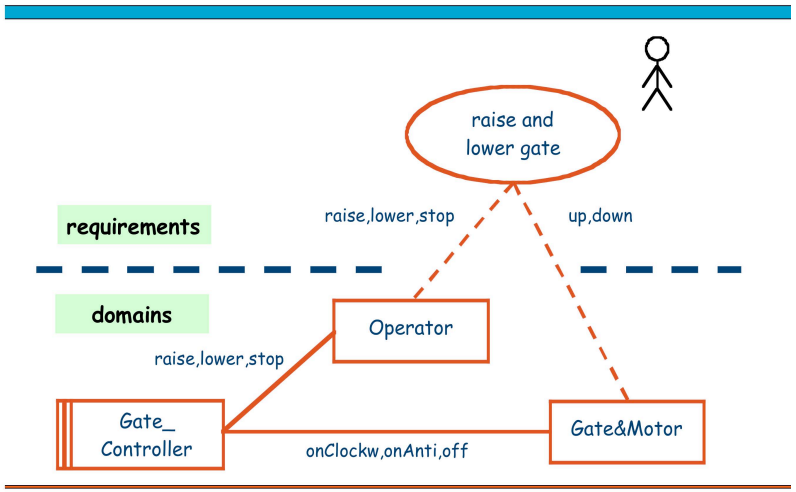
VALENCIA 2005
University of Leicester

Requirements & coordination



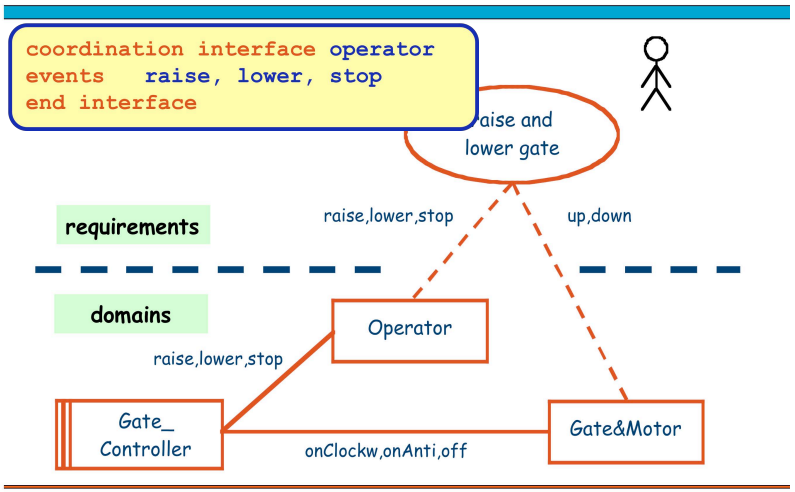
VALENCIA 2005
University of Leicester

The Cooordinated Problem Frame



The Cooordinated Problem Frame

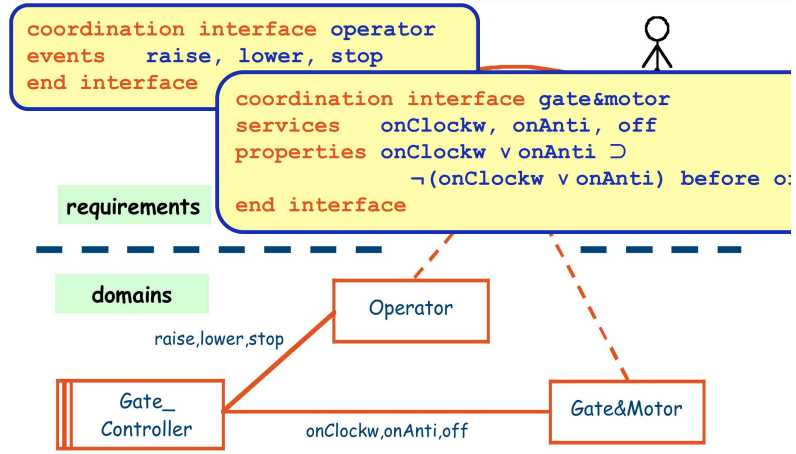

```
coordination interface operator
events   raise, lower, stop
end interface
```



The Cooordinated Problem Frame


```
coordination interface operator
events   raise, lower, stop
end interface
```

```
coordination interface gate&motor
services   onClockw, onAnti, off
properties onClockw ∨ onAnti ⊃
           ¬(onClockw ∨ onAnti) before o:
end interface
```

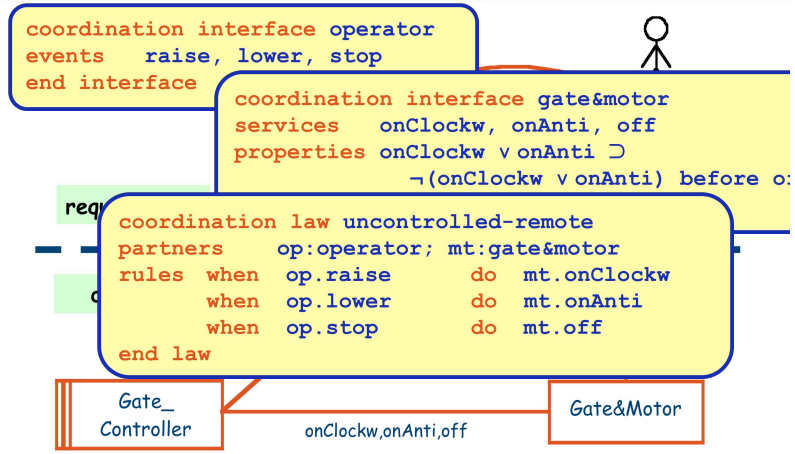


The Cooordinated Problem Frame

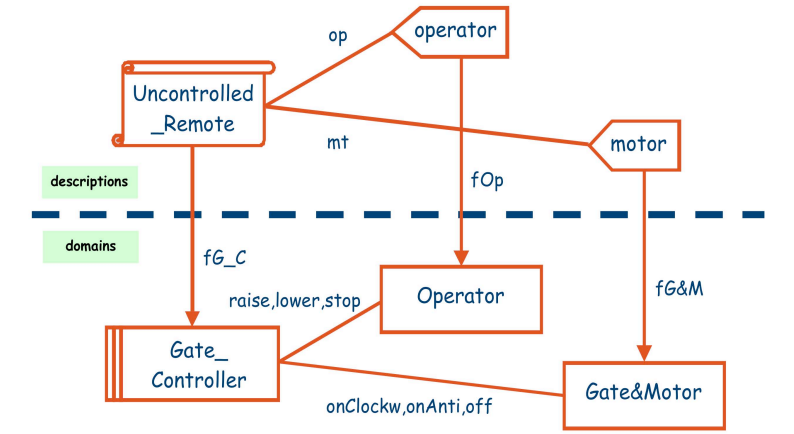

```
coordination interface operator
events   raise, lower, stop
end interface
```

```
coordination interface gate&motor
services   onClockw, onAnti, off
properties onClockw ∨ onAnti ⊃
           ¬(onClockw ∨ onAnti) before o:
```

```
coordination law uncontrolled-remote
partners    op:operator; mt:gate&motor
rules  when  op.raise      do  mt.onClockw
       when  op.lower      do  mt.onAnti
       when  op.stop       do  mt.off
end law
```

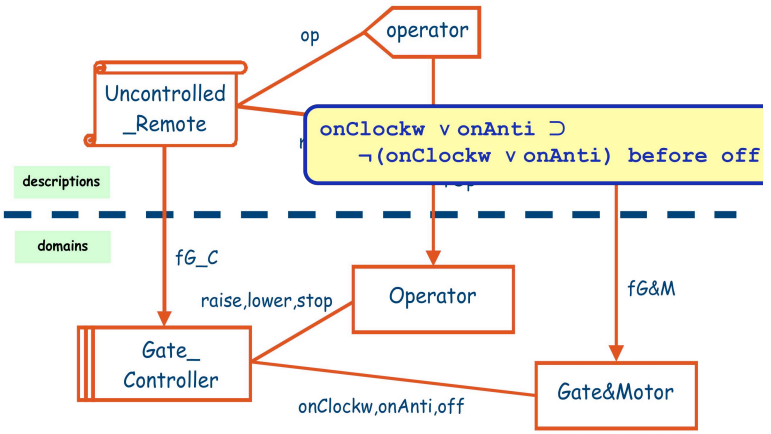


The Cooordinated Problem Frame



The Cooordinated Problem Frame

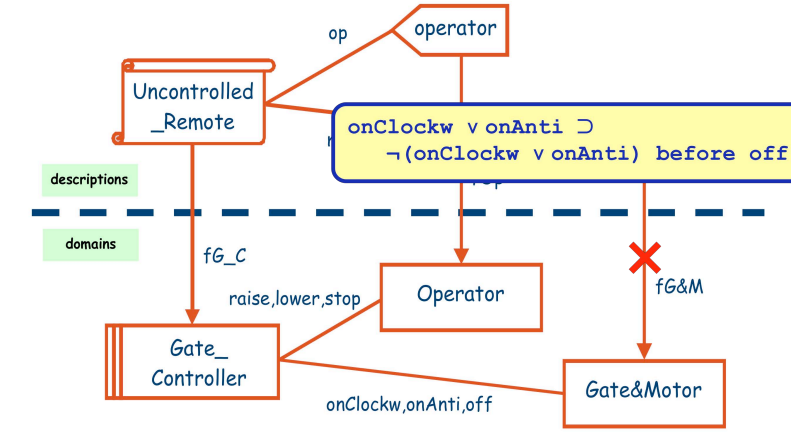
44



VALENCIA 2005
University of Leicester

The Cooordinated Problem Frame

44



VALENCIA 2005
University of Leicester

A new requirement

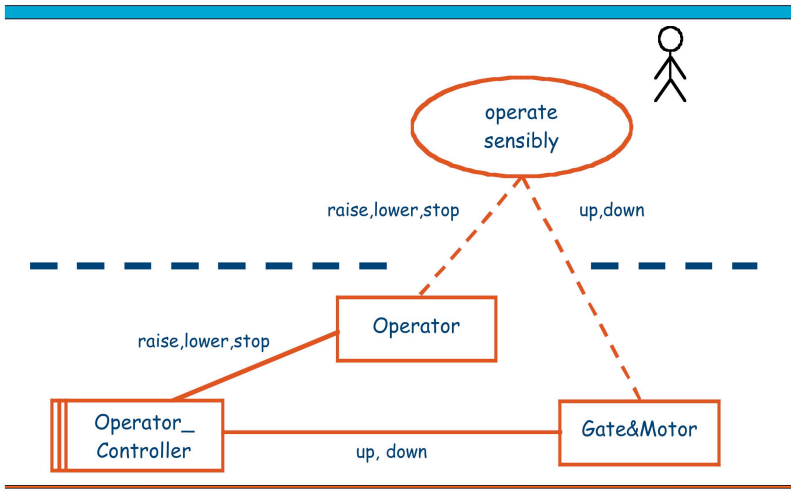
Because the operator commands the sluice machine directly with no external control, and the motor accepts clockwise and anticlockwise pulses when in operation, there is nothing to prevent undesirable sequences of commands and the motor can get broken.

As a result, we now want to control the operator in such a way that:

- issuing a **lower** command is only accepted when the motor is stopped and the gate has not reached the bottom
- issuing a **raise** command is only accepted when the motor is stopped and the gate has not reached the top
- issuing a **stop** command is only accepted when the motor is in operation.

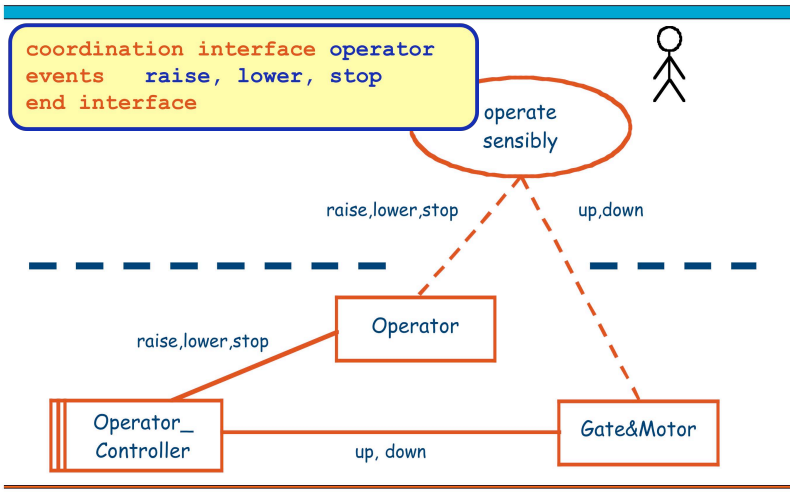
VALENCIA 2005
University of Leicester

A new problem frame



A new problem frame

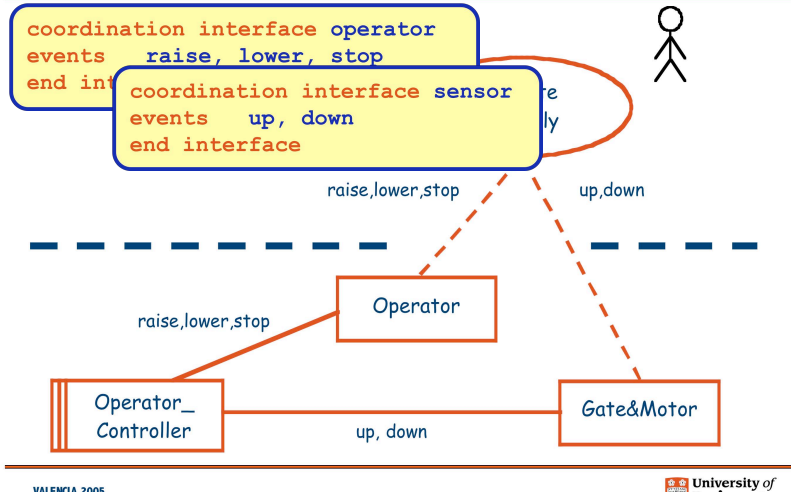

```
coordination interface operator
events   raise, lower, stop
end interface
```



A new problem frame


```
coordination interface operator
events   raise, lower, stop
end int
```

```
coordination interface sensor
events   up, down
end interface
```

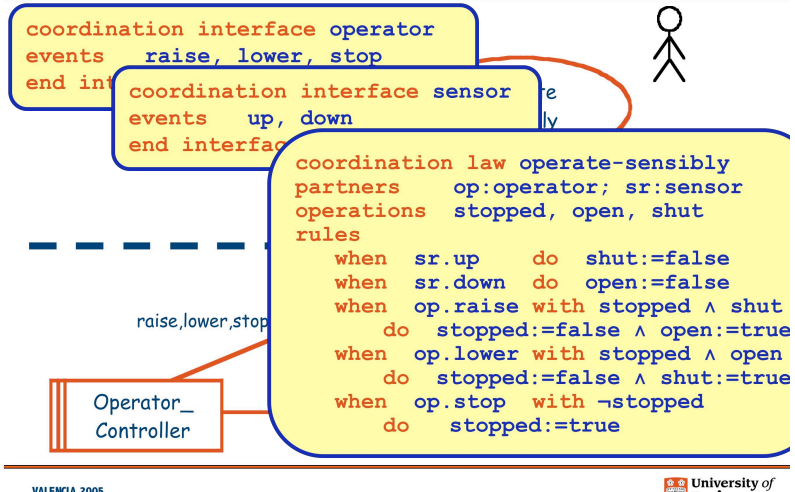


A new problem frame

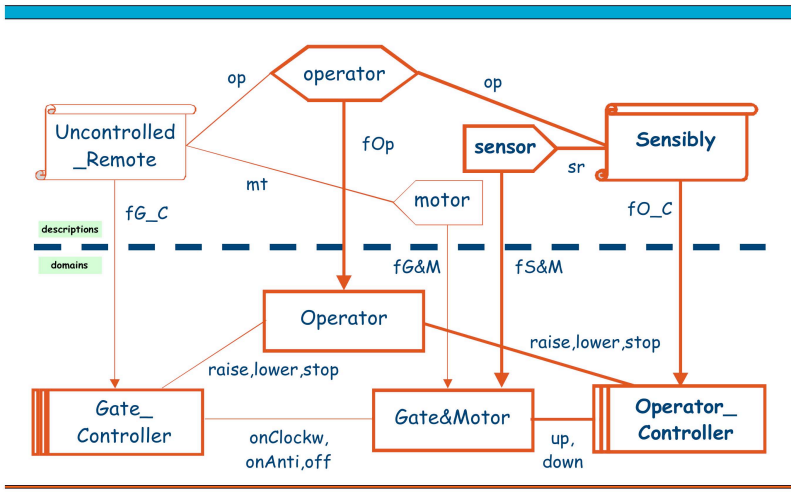

```
coordination interface operator
events   raise, lower, stop
end int
```

```
coordination interface sensor
events   up, down
end interfac
```

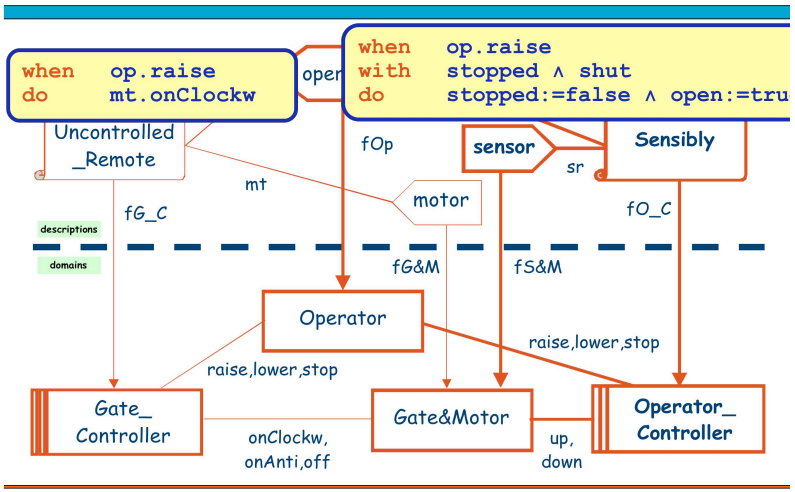
```
coordination law operate-sensibly
partners    op:operator; sr:sensor
operations  stopped, open, shut
rules
   when  sr.up    do  shut:=false
   when  sr.down  do  open:=false
   when  op.raise with stopped ∧ shut
      do  stopped:=false ∧ open:=true
   when  op.lower with stopped ∧ open
      do  stopped:=false ∧ shut:=true
   when  op.stop  with ¬stopped
      do   stopped:=true
```



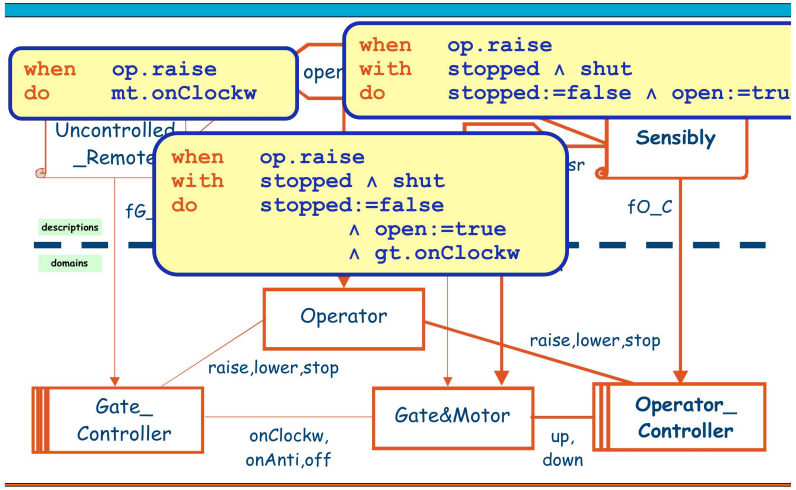
The Cooordinated Problem Frame



The Cooordinated Problem Frame



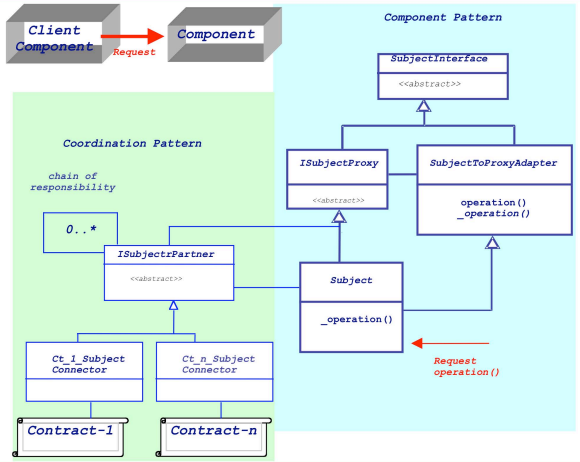
The Cooordinated Problem Frame



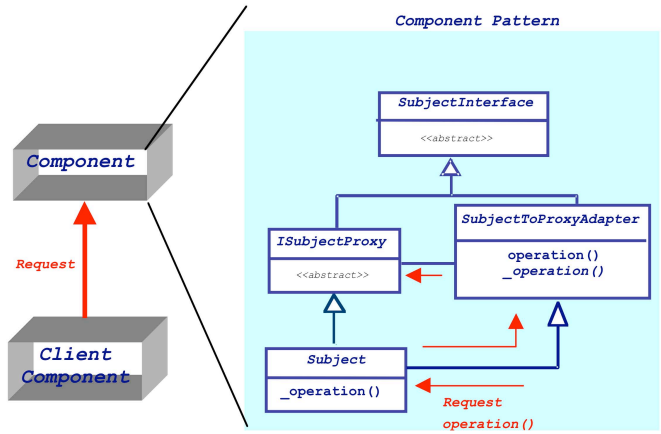
A Micro-Architecture

- None of the standards for component-based software development - CORBA, JavaBeans, COM - can support superposition as a first-class mechanism.
- Because of this, ATX proposed a micro-architecture that exploits polymorphism and subtyping, and is based on well known design patterns, such as the Chain of Responsibility, and the Proxy or Surrogate.

Coordination micro-architecture (1)



Coordination micro-architecture (2)



Classes defining the pattern

- ***SubjectInterface*** – an abstract class that defines the operations of the *Subject* under potential coordination.
- ***SubjectToProxyAdapter*** – a concrete class that defines the ability to alternatively use a proxy or internal methods for the implementation of a given *Subject* interface. Allows, at run time and using the polymorphic entity proxy, for delegeting requests to *ISubjectPartner* that links the *Subject* to the contracts that coordinate it. If no contract is involved it forwards requests directly to *Subject*.

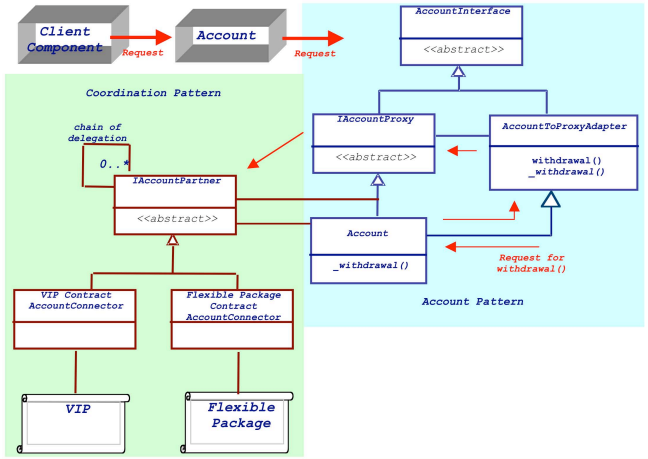
Classes defining the pattern

- ***ISubjectProxy*** – it represents an object with the capability
  implementing the *Subject* interface. It is an abstract class that defin
  the common interface of *Subject* and *ISubjectPartner*. The interfa
  is inherited from *SubjectInterface* to guarantee that all these class
  offer the same interface as *Subject* with which real subject clien
  have to interact.
- ***Subject*** – the concrete domain class, candidate for coordination, whi
  provides the concrete implementation of the various services a
  inherits from *SubjectToProxyAdapter*.

Classes defining the pattern

- ***ISubjectPartner*** - defines the general abilities of a concept to b under coordination. Maintains the connection between contracts ar the real object (*Subject*). The class is responsible for delegati received requests to *CtSubjectConnectors* according to a **chain responsibility**.
- ***Ct-i-SubjectConnector*** - a partner that represents the specificities *Subject* coordination for a given contract in which *Subject* is participant.
- ***Contract*** - a coordination object that defines the rules that will b superimposed on *Subject*.

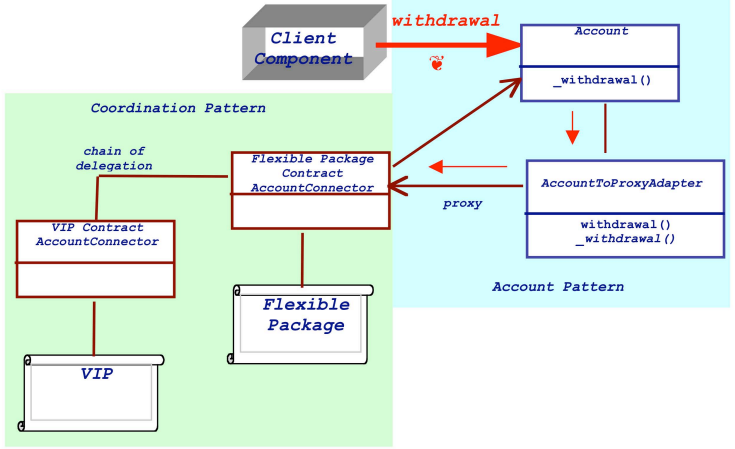
Account coordination (3)



Account coordination (4)

- If there are no contracts coordinating a real subject, the contract pattern can be simplified.
- In this scenario, the only overhead imposed by the pattern is an extra call from *SubjectToProxyAdapter* to *Subject*.
- The following diagrams shows how the contracts, VIP and Flexible Package, superposes new behaviour when requests *withdrawal()* are invoked on a real object of type *Account*.

Account coordination (5)



Operational view

- Before *the subject* gives rights to *the real object* to execute the request, it intercepts the request and gives right to *the contract* to decide if the request is valid and perform other actions.
- This allows us to impose other contractual obligations on the interaction between the caller and the callee.
- Moreover, it allows the contract to perform other actions before or after the real object executes the request.
- Only if the contract authorises can the connector ask the involved objects to execute and commit, or undo execution because of violation of post-conditions established by the contract.

Summary

- Having this form of coordination available as a primitive construction when specifying components and their interactions avoids the burden of having to code such a micro-architecture each time.
- In the meanwhile, tools which provide automatic code generation from high level specifications, can hide the implementation complexity of coordination, allowing the developer just to specify the contract itself.
- Even if the design solution that was proposed is not adopted, there are many advantages in using the coordination primitives at the more abstract modelling levels.

Resources

Resources

- Software Design in Java 2
  K.Lano, J.Fiadeiro and L.Andrade
  Palgrave Macmillan

- http://www.fiadeiro.org/jose/CommUnity/publications.html
  for publications in general (section on coordination contracts)

- www.atxsoftware.com/CDE for the **Coordination Development Environment**

3

CommUnity:
The Mathematics of Architecture

Need for formality

- **Architectures = Box & Lines ?**
  - is there a shared understanding of what they mean?
  - how easy is it to communicate details ("up" and "down")?
  - what degree of analytic leverage are we given?
  - how informed are we for selecting among alternatives?
- **We need a formal approach supporting**
  - abstraction: capturing the essential
  - precision: knowing what exactly is being addressed
  - analysis: predicting what properties will emerge
  - refinement: coding according to standard reference models
  - automation: tool support

Formalise what?

- **Architecture of specific systems**
  - what will the system do? what properties will it have?
  - what will happen if we replace this component/connector by this other one?
- **Architectural styles**
  - what does it mean to conform to an architectural style?
  - what system properties can be inferred from its style?
- **Architectural notions in general**
  - what does composition/interconnection/refinement mean?
  - what degree of heterogeneity to they support?

To infer what?

- Structure
  - How is the system organised? How can it evolve?
- Compatibility
  - Is the system well composed?
- Function
  - What behaviour will the system exhibit?
- Resource
  - How fast and how big?
- Invariants
  - What evolution-independent properties are guaranteed?
- Interoperability
  - How is system structure constraining usage in more general contexts?

How?

- Architecture description languages (ADLs) have been proposed as a possible answer
- Several prototype ADLs and supporting tools have been proposed

Rapide	events with simulation and animation
UniCon	emphasizing heterogeneity and compilation
Wright	formal specification of connector interactions
Aesop	style-specific arch design languages
Darwin	service-oriented architectures
SADL	SRI language emphasizing refinement
Meta-H	arch description for avionics domain
C-2	arch style using implicit invocation
ACME	open-ended approach ("XML for architectures")

Purpose of ADLs

- An ADL is a language that provides features for modelling a software system's conceptual architecture, at least:
  - components
  - connectors
  - configurations
- The purpose of an ADL is to
  - provide models, notations, and tools to describe components and their interactions
  - support large-scale, high-level designs
  - support principled selection and application of architectural paradigms

CommUnity

- Not a full-fledged ADL
  - its purpose is not to support large-scale, industrial architectural design
  - but to serve as a test bed for formalising architectural notions and techniques
  - and a prototype for extensions (e.g. mobility)
  - but has found its way into industrial practice
- Full mathematical semantics
  - the semantics is largely "language independent"
  - supports reasoning and prototyping
  - supports heterogeneity (based on General Systems Theory)

Origins

A confluence of contributions from

- (Re)Configurable Distributed Systems
  - exoskeletal software
- Parallel Program Design
  - superposition
- Coordination Models and Languages
  - separation of concerns (Computation / Coordination)
- The categorical imperative
  - Goguen's approach to General Systems Theory

Architectural elements

- **Components**
  - model entities/devices/machines (software or "real world"), that keep an internal state, perform computations, and are able to synchronise with their environment and exchange information through channels
  - "designs" given in terms of communication channels and actions
- **Connectors**
  - model entities whose purpose is to coordinate interactions between components
  - "structured designs" given in terms of a "glue" and collection of "roles" (as in Wright)
  - can be superposed at run-time over given components
- **Configurations**
  - diagrams in a category of designs as objects and superposition as morphisms;
  - composition (emergent behaviour) given by colimit construction

Designing components

An example

The design of a "naïve" bank account

```
design n-account is
   out num:nat, bal:int
   in  v: nat
   do  dep: true → bal:=v+bal
   []  wit: bal≥v → bal:=bal-v
```

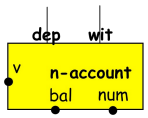

Designing components

10

An example

The design of a "naïve" bank account

```
design n-account is
   out num:nat, bal:int
   in  v: nat
   do  dep: true → bal:=v+bal
   []  wit: bal≥v → bal:=bal-v
```



Channels

- Provide for interchange of data
  - actions do **not** have I/O parameters!
  - reading from a channel does not consume the data!
- **Output** channels `out(V)`
  - allow the environment to observe the state of the component, and for the component to transmit data to the environment
  - the component controls the data that is made available; the environment can only read the data
- **Input** channels `in(V)`
  - allow the environment to make data available to the component
  - the environment controls the data that is made available; the component can only read the data
- **Private** channels `prv(V)`
  - model communication inside (different parts of) the component;
  - the environment can neither read from nor write into private channels

Actions

- Provide for **synchronisation** with the environment (e.g. to transmit or receive new data made available through the channels)
- Provide for the **computations** that make available or consume data

```
do g[D(g)] : L(g), U(g) → R(g)
```

- **Write** frame `D(g)`
  - the local channels (out, prv) into which the action can write data
- **Computation** `R(g)`
  - how the execution of the action uses the data read on the input channels and changes the data made available on the local channels
- **Guards** `L(g), U(g)`
  - set of states in which the action may be enabled `L(g)`
  - set of states in which the action must be enabled `U(g)`
  - `U(g)⊃ L(g)`

Designing components

Another example

The design of a VIP-account that ***may*** accept a withdrawal when the balance together with a given credit amount is greater than the requested amount, and ***will*** accept any withdrawal for which there are funds available to match the requested amount:

```
design vip-account[CRE:nat] is
   out num: nat, bal:int
   in  v: nat
   do  dep[bal]: true → bal'=v+bal
   []  wit[bal]: bal+CRE≥v, bal≥v → bal'≤bal-v
```


Operational Semantics

- A design is called a **program** if, for every action g,
  - L(g) and U(g) coincide
  - R(g) defines a conditional multiple assignment.
- Execution of a program on a given state:
  - any of the actions whose enabling condition holds can be selected by the environment, in which case its assignments are executed atomically
  - **private** actions `[prv]` are internally selected in a fair way: every private action that is infinitely often enabled is selected an infinite number of times

Superposition

- A structuring mechanism for the design of systems that allows to build on already designed components by "augmenting" them while "preserving" their properties.
- Typically, the additional behaviour results from the introduction of new channels and corresponding assignments (that may use the values of the channels of the base design).

Applying Superposition

An example

Extending the design of n-account to control how many days the balance has exceeded a given amount since it was last reset.

```
design e-account[MAX:int] is
  out  num:nat, bal:int, count:int
  in   v,day:nat
  prv  d:int
  do   dep[bal,d,count]: true →
           bal:=v+bal
       ‖   d:=day
       ‖   if bal≥MAX then count:=count+(day-d)
  []   wit[bal,d,count]:
           bal≥v → bal:=bal-v
       ‖   d:=day
       ‖   if bal≥MAX then count:=count+(day-d)
  []   reset [d,count]:
           true, false → count:=0‖d:=day
```


Characterising Superposition

The relationship between a design P_1 and a design P_2 obtained from P_1 through the superposition of additional behaviour, can be modelled as a mapping between the channels and actions of the two designs

$$\sigma: P_1 \rightarrow P_2$$

subject to some constraints.

Superposition Morphisms

A superposition morphism $\sigma: P_1 \rightarrow P_2$ consists of

- a total function $\sigma_{ch}: V_1 \rightarrow V_2$ s.t.
  - Sorts, privacy and availability of channels are preserved
  - Input channels may become output channels
- a partial mapping $\sigma_{ac}: \Gamma_2 \rightarrow \Gamma_1$ s.t.
  - Privacy/availability of actions is preserved
  - Domains of channels are preserved

Superposition Morphisms

A superposition morphism $\sigma: P_1 \rightarrow P_2$ consists of

- a total function $\sigma_{ch}: V_1 \rightarrow V_2$ s.t.
  - $sort_2(\sigma_{ch}(v)) = sort_1(v)$
  - $\sigma_{ch}(out(V_1)) \subseteq out(V_2)$
  - $\sigma_{ch}(in(V_1)) \subseteq out(V_2) \cup in(P_2)$
  - $\sigma_{ch}(prv(V_1)) \subseteq prv(V_2)$
  - Sorts, privacy and availability of channels are preserved
  - Input channels may become output channels
- a partial mapping $\sigma_{ac}: \Gamma_2 \rightarrow \Gamma_1$ s.t.
  - $\sigma_{ac}(sh(\Gamma_2)) \subseteq sh(\Gamma_1)$
  - $\sigma_{ac}(prv(\Gamma_2)) \subseteq prv(\Gamma_1)$
  - $\sigma_{ch}(D_1(\sigma_{ac}(g))) \subseteq D_2(g)$
  - $\sigma_{ac}(D_2(\sigma_{ch}(v))) \subseteq D_1(v)$
  - Privacy/availability of actions is preserved
  - Domains of channels are preserved

Superposition Morphisms

and, moreover, for every g in Γ_2 s.t. $\sigma_{ac}(g)$ is defined

- Effects of actions must be preserved or made more deterministic
- The bounds for enabling conditions of actions can be strengthened but not weakened

Superposition Morphisms

19

and, moreover, for every g in Γ_2 s.t. $\sigma_{ac}(g)$ is defined

- $R_2(g) \supset \underline{\sigma}(R_1(\sigma_{ac}(g)))$
- $L_2(g) \supset \underline{\sigma}(L_1(\sigma_{ac}(g)))$
- $U_2(g) \supset \underline{\sigma}(U_1(\sigma_{ac}(g)))$

- Effects of actions must be preserved or made more deterministic
- The bounds for enabling conditions of actions can be strengthened but not weakened

VALENCIA 2005 University of Leicester

Superposition Morphisms: Examples


```
design n-account is
  out  num:nat, bal:int
  in   v:nat
  do   dep[bal]: true → bal:=v+bal
  []   wit[bal]: bal≥v → bal:=bal-v
```

inclusion

```
design e-account[MAX:int] is
  out  num:nat, bal:int, count:int
  in   v,day:nat
  prv  d:int
  do   dep[bal,d,count]: true →
           bal:=v+bal
       ||  d:=day
       ||  if bal≥MAX then count:=count+(day-d)
  []   wit[bal,d,count]:
           bal≥v → bal:=bal-v
       ||  d:=day
       ||  if bal≥MAX then count:=count+(day-d)
  []    reset [d,count]:
           true, false → count:=0||d:=day
```


Superposition Morphisms: Examples

Another example

```
design account is
  out  num:nat, bal:int
  in   v: nat
  do   dep: true → bal:=v+bal
  []   wit: true → bal:=bal-v
```

inclusion

```
design n-account is
  out  num:nat, bal:int
  in   v: nat
  do   dep: true → bal:=v+bal
  []   wit: bal≥v → bal:=bal-v
```

Externalising superposed behaviour

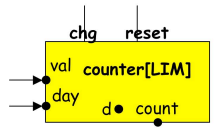
- These examples represent two typical kinds of superposition
  - monitoring
  - regulation
- The superposed behaviour can be captured by a component
  - monitor
  - regulator

Support reuse

- and the new design is obtained by interconnecting the underlying design with this component.

Externalising the counter

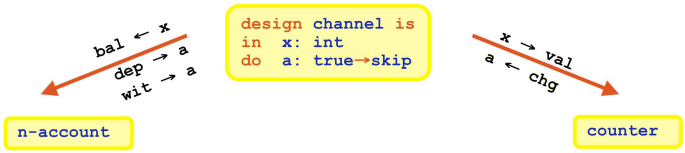
A design of a counter that counts how many days a value has exceed a given value, since the last time it was reset




```
design counter[LIM:int] is
   in  val,day:nat
   out count:int
   prv d:int
   do  chg[d,count]: true →
           d:=day
       ||  if val≥LIM then count:=count+(day-d)
   []  reset[d,count]: true, false → count:=0||d:=day
```


Externalising the counter

To identify which channels and actions of the account are involved in the monitoring by the counter, we use the diagram



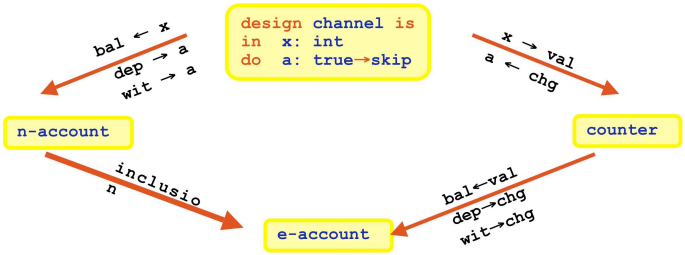
This diagram captures the configuration of a system with two components — n-account and counter — that are interconnected through a third design (a communication channel)

Configurations

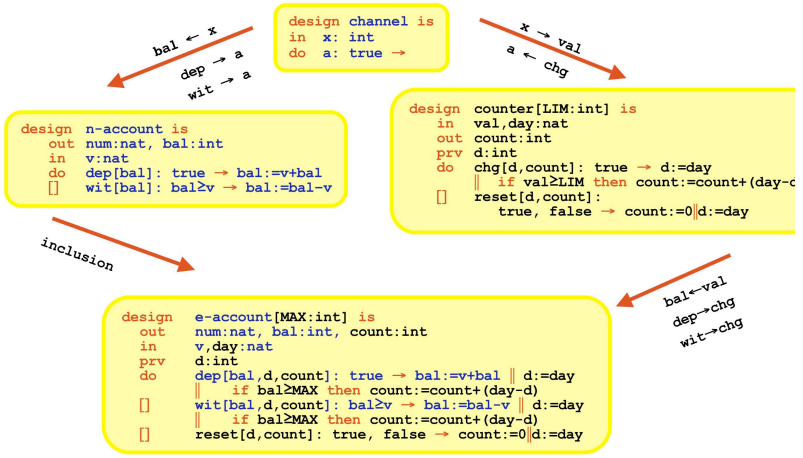
- Using diagrams whose nodes are labelled by designs and whose arcs are labelled by superposition morphisms, it is possible to design large systems from simpler components.
- Interactions between components are made explicit through the corresponding name bindings.
- Name bindings are represented as additional nodes labelled with designs and edges labelled by morphisms.

Semantics of Configurations

What's the relationship between **e-account** and the configuration?

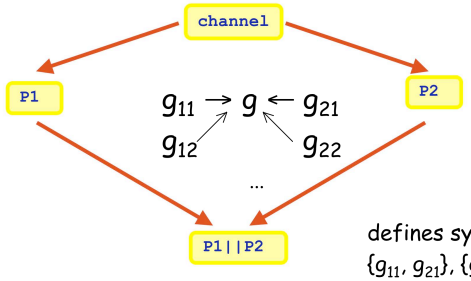


Semantics of Configurations



Semantics of Configurations

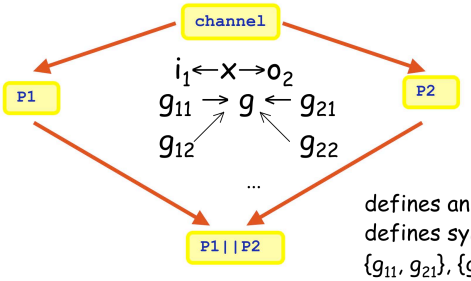
The semantics of configurations is given by the "amalgamated sum" (colimit) of the diagram.



defines synchronisation sets
$\{g_{11}, g_{21}\}, \{g_{12}, g_{21}\},\ldots$

Semantics of Configurations

The semantics of configurations is given by the "amalgamated sum" (colimit) of the diagram.



defines an I/O connection
defines synchronisation sets
$\{g_{11}, g_{21}\}, \{g_{12}, g_{21}\},\ldots$

Semantics of Configurations

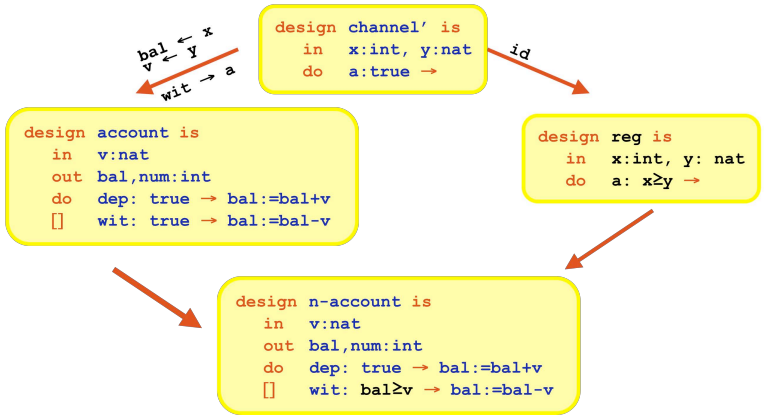
The colimit of such design diagrams

- Amalgamates channels involved in each i/o interconnection and the result is an output channel of the system design
- Represents every synchronisation set $\{g_1,g_2\}$ by a single action $g_1|g_2$ with
  - safety bound: conjunction of the safety bounds of g_1 and g_2
  - progress bound: conjunction of the progress bounds of g_1 and g_2
  - conditions on next state: conjunction of conditions of g_1 and g_2

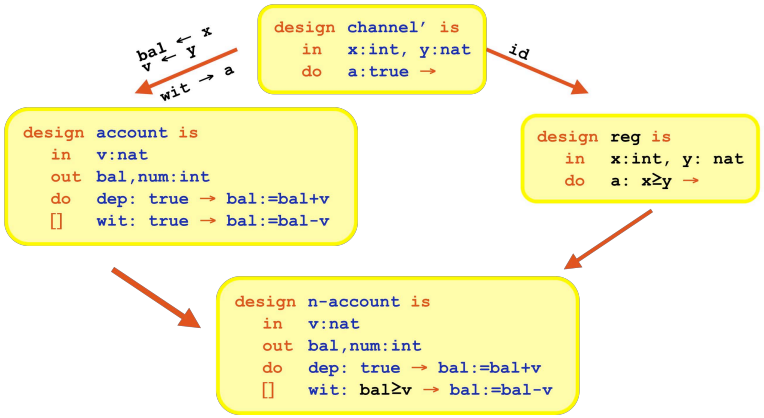
Configurations

- Not every diagram represents a meaningful configuration.
- Restrictions on diagrams that make them well-formed configurations:
  - An output channel of a component cannot be connected (directly or indirectly through input channels) with output channels of the same or other components.
  - Private channels and private actions cannot be involved in the connections.
- These restrictions cannot be captured by the notion of superposition because they involve the whole diagram.

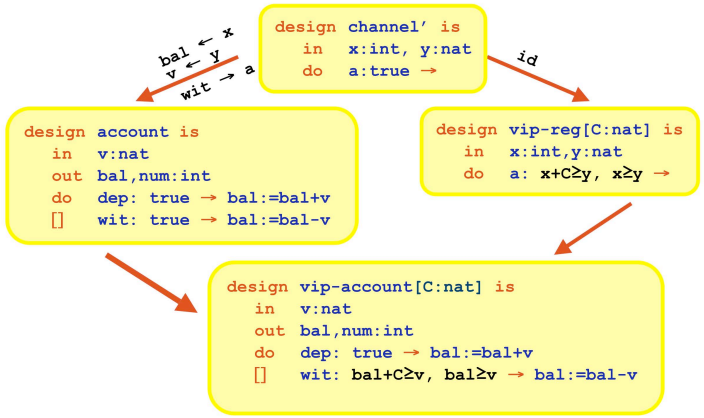
Externalising the regulator



Externalising the regulator



vip-account



Recall: architectural elements

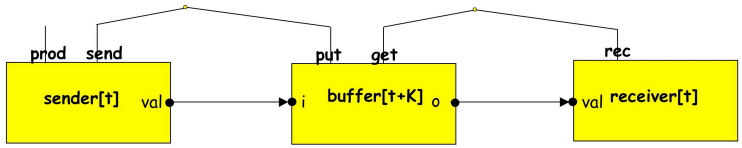
- **Components**
  - model entities/devices/machines (software or "real world"), that keep an internal state, perform computations, and are able to synchronise with their environment and exchange information through channels
  - "designs" given in terms of communication channels and actions
- **Connectors**
  - model entities whose purpose is to coordinate interactions between components
  - "structured designs" given in terms of a "glue" and collection of "roles" (as in Wright)
  - can be superposed at run-time over given components
- **Configurations**
  - diagrams in a category of designs as objects and superposition as morphisms;
  - composition (emergent behaviour) given by colimit construction

From simple to complex interactions

- The configuration diagrams presented so far express **simple** and **static** interactions between components
  - action synchronisation
  - the interconnection of input channels of a component with output channels of other components
- More complex interaction protocols can also be described by configurations...

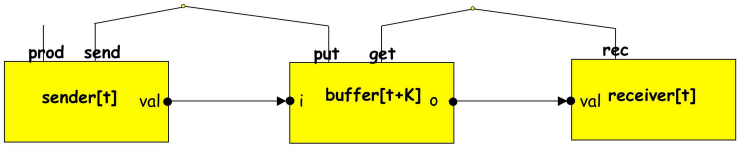
Bounded asynchronous interaction

A generic sender and receiver communicating asynchronously, through a bounded buffer



Bounded asynchronous interaction

A generic sender and receiver communicating asynchronously, through a bounded buffer

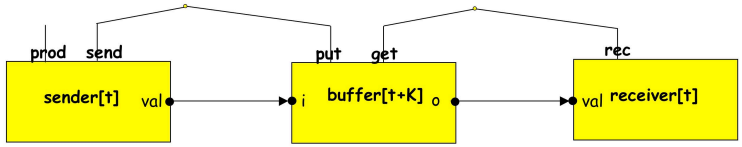



```
design sender[t] is
out   val:t
prv   rd:bool
do    prod[val,rd]:¬rd,false→rd'
[]    send[rd]:rd,false → ¬rd'
```


Bounded asynchronous interaction

35

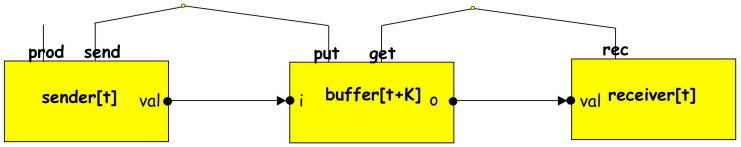
A generic sender and receiver communicating asynchronously, through a bounded buffer




```
design sender[t] is
out   val:t
prv   rd:bool
do    prod[val,rd]:¬rd,false→rd'
[]    send[rd]:rd,false → ¬rd'
```

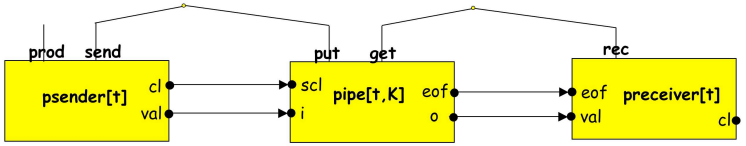
```
design receiver[t] is
in    val:t
do    rec:true,false→
```


Bounded asynchronous interaction

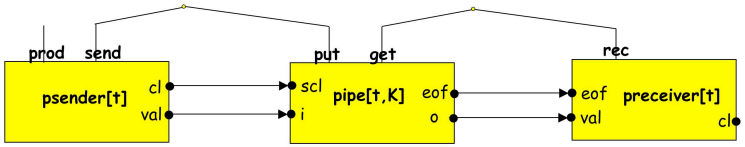



```
design buffer[t; K:nat] is
in     i:t
out    o:t
prv    q:queue(K,t); rd:bool
do     put:¬full(q)→q:=enqueue(i,q)
[]prv  next:¬empty(q)∧¬rd →o:=head(q)‖q:=tail(q)‖rd:=true
[]     get:rd → rd:=false
```


Communicating through a pipe

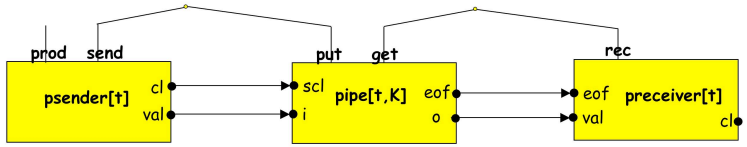


Communicating through a pipe




```
design psender[t] is
out   val:t, cl:bool
prv   rd:bool
do    prod[val,rd]:¬rd∧¬cl,false→rd'
[]prv close[cl]:¬rd∧¬cl,false→cl'
[]    send[rd]:rd,false→¬rd'
```

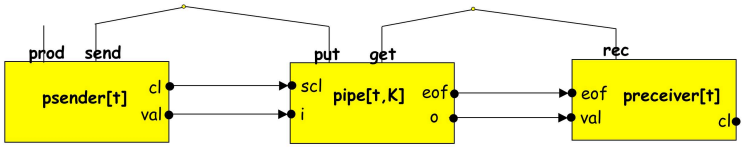

Communicating through a pipe




```
design psender[t] is
out   val:t, cl:bool
prv   rd:bool
do    prod[val,rd]:¬rd∧¬cl,false→rd'
[]prv close[cl]:¬rd∧¬cl,false→cl'
[]    send[rd]:rd,false→¬rd'
```

```
design preceiver[t] is
in    val:t, eof:bool
out   cl:bool
do    rec:¬eof∧¬cl,false→
[] prv close:¬cl,¬cl∧eof→cl'
```

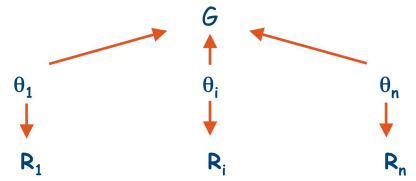

Communicating through a pipe




```
design pipe[t,K:nat] is
in    i:t, scl:bool
out   o:t, eof:bool
prv   q:queue(K,t);rd:bool
do    put: ¬full(q)→q:=enqueue(i,q)
[]prv next: ¬empty(q)∧¬rd→o:=head(q)‖q:=tail(q)‖rd:=true
[]    get: rd→rd:=false
[]prv signal: scl∧empty(q)∧¬rd→eof:=true
```


Connectors

- A **connector** is a well-formed configuration of the form



G is the glue and **R**'s are the roles

- Its **semantics** is given by the colimit of the diagram

Refinement

Connectors can be **applied** (instantiated) to components that refine (are instances of) their roles

A **refinement** mapping

$$\sigma: P_1 \rightarrow P_2$$

supports the identification of a way in which the design P_1 is refined by P_2.

Refinement morphisms

A refinement morphism $\sigma: P_1 \rightarrow P_2$ consists of

- a total function $\sigma_{ch}: V_1 \rightarrow \mathbf{Term}(V_2)$ s.t.

- a partial mapping $\sigma_{ac}: \Gamma_2 \rightarrow \Gamma_1$ s.t.

Refinement morphisms

A refinement morphism $\sigma: P_1 \rightarrow P_2$ consists of

- a total function $\sigma_{ch}: V_1 \rightarrow \mathbf{Term}(V_2)$ s.t.

Sorts are preserved as well as the border between the component and its environment

- a partial mapping $\sigma_{ac}: \Gamma_2 \rightarrow \Gamma_1$ s.t.

Domains of channels are preserved
Every action that models interaction has to be implemented

Refinement morphisms

A refinement morphism $\sigma: P_1 \rightarrow P_2$ consists of

- a total function $\sigma_{ch}: V_1 \rightarrow \mathbf{Term}(V_2)$ s.t.
  - $sort_2(\sigma_{ch}(v)) = sort_1(v)$
  - $\sigma_{ch}(out(V_1)) \subseteq out(V_2)$
  - $\sigma_{ch}(in(V_1)) \subseteq in(V_2)$
  - $\sigma_{ch}(prv(V_1)) \subseteq Term(loc(V_2))$

Sorts are preserved as well as the border between the component and its environment

- a partial mapping $\sigma_{ac}: \Gamma_2 \rightarrow \Gamma_1$ s.t.
  - $\sigma_{ac}(sh(\Gamma_2)) \subseteq sh(\Gamma_1)$
  - $\sigma_{ac}(prv(\Gamma_2)) \subseteq prv(\Gamma_1)$
  - $\sigma_{ac}^{-1}(g) \neq \emptyset,\ g \in sh(\Gamma_1)$
  - $\sigma_{ch}(D_1(\sigma_{ac}(g))) \subseteq \underline{D}_2(g)$
  - $\sigma_{ac}(\underline{D}_2(\sigma_{var}(v))) \subseteq D_1(v),\ v \in loc(V_1)$

Domains of channels are preserved
Every action that models interaction has to be implemented

Refinement morphisms

and, moreover, for every g in Γ_2 s.t. $\sigma_{ac}(g)$ is defined

and for every g_1 in Γ_1

Effects of actions must be preserved or made more deterministic.

The interval defined by the safety and progress bounds of each action must be preserved or reduced

Refinement morphisms

42

and, moreover, for every g in Γ_2 s.t. $\sigma_{ac}(g)$ is defined

and for every g_1 in Γ_1

Effects of actions must be preserved or made more deterministic.

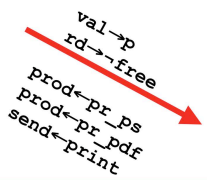
The interval defined by the safety and progress bounds of each action must be preserved or reduced

VALENCIA 2005 University of Leicester

- $R_2(g) \supset \underline{\sigma}(R_1(\sigma_{ac}(g)))$
- $L_2(g) \supset \underline{\sigma}(L_1(\sigma_{ac}(g)))$
- $\underline{\sigma}(U_1(g_1)) \supset \bigvee_{\{g2:\underline{\sigma}(g2)=g1\}} U_2(g_2)$

worduser - a refinement of sender


```
design sender(ps+pdf) is
out val:ps+pdf
prv rd:bool
do  prod[val,rd]:¬rd,false→rd'
[]  send[rd]:rd,false → ¬rd'
```

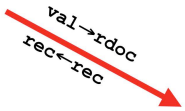



```
design user is
out p:ps+pdf
prv free:bool, w:MSWord
do  save[w]: true,false →
[]  pr_ps[p,free]: free → p:=ps(w)‖free:=false
[]  pr_pdf[p,free]: free → p:=pdf(w)‖free:=false
[]  print[free]: ¬free → free:=true
```

VALENCIA 2005 University of Leicester

printer: a refinement of receiver


```
design receiver(ps+pdf) is
in  val:ps+pdf
do  rec[]:true,false→
```




```
design printer is
out rdoc:ps+pdf
prv busy:bool, pdoc:ps+pdf
do  rec:¬busy→pdoc:=rdoc‖busy:=true
[]  end_print:busy,false→busy:= false
```

VALENCIA 2005 University of Leicester

Structuring systems vs Refinement

It is essential that

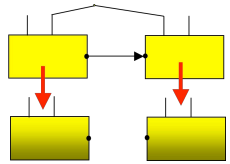
the gross modularisation of a system
in terms of
components and their interconnections

be "respected" when component designs are refined into more concrete ones

Compositionality

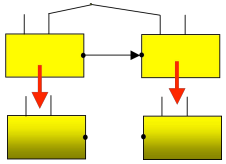
Structuring systems vs Refinement

If the descriptions of the components of a system are refined into more concrete ones



Structuring systems vs Refinement

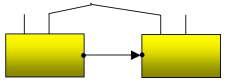
If the descriptions of the components of a system are refined into more concrete ones



1. It is possible to propagate the interactions previously defined

Structuring systems vs Refinement

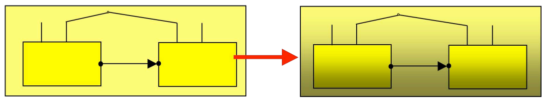
If the descriptions of the components of a system are refined into more concrete ones



1. It is possible to propagate the interactions previously defined

Structuring systems vs Refinement

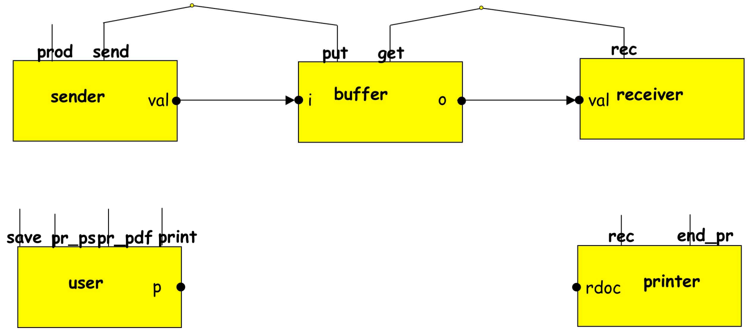
If the descriptions of the components of a system are refined into more concrete ones



1. It is possible to propagate the interactions previously defined
2. The resulting description of the system refines the previous one

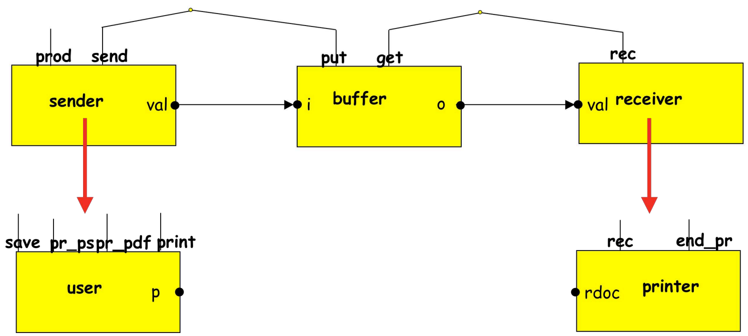
Connector instantiation

Example



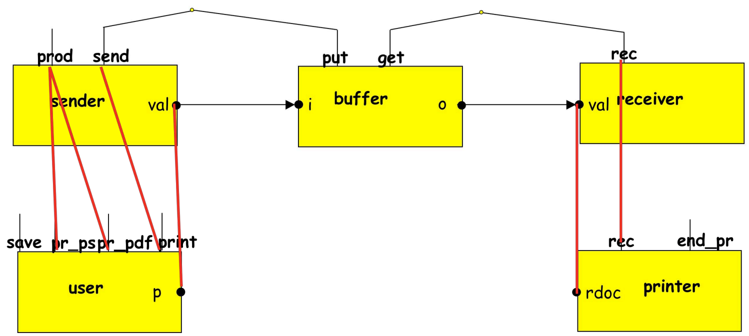
Connector instantiation

Example



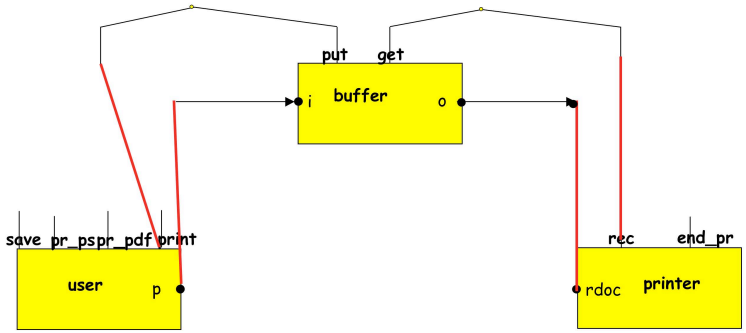
Connector instantiation

Example

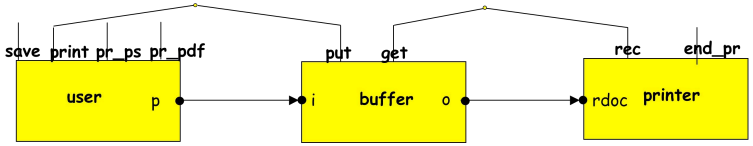


Propagation of the interconnections

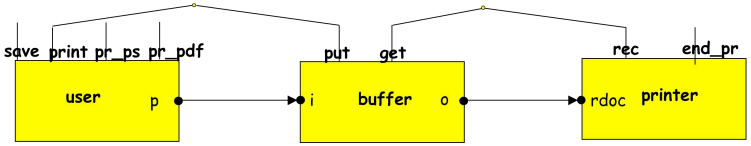
Example



Compositionality

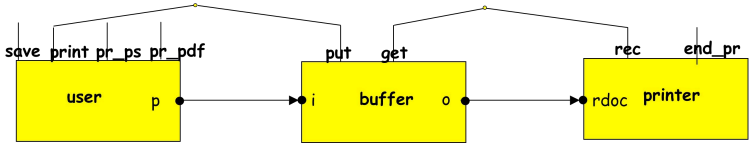


Compositionality



Compositionality ensures that properties inferred from the more abstract description hold also for the more concrete (refined) one

Compositionality

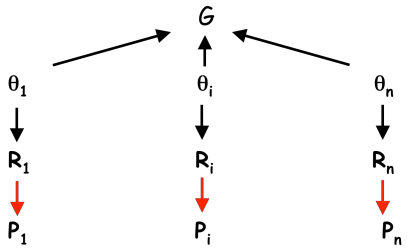


Compositionality ensures that properties inferred from the more abstract description hold also for the more concrete (refined) one

Eg: **in order message delivery** does not depend on the speed at which messages are produced and consumed

Connectors - Instantiation

- An **instantiation** of a connector consists of, for each of its roles R, a design P together with a refinement morphism ϕ:R→P

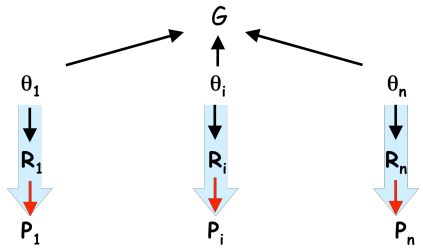


The **semantics** of a connector instantiation is the colimit of the diagram

Connectors - Instantiation

50

- An **instantiation** of a connector consists of, for each of its roles R, a design P together with a refinement morphism ϕ:R→P



The **semantics** of a connector instantiation is the colimit of the diagram

VALENCIA 2005
University of Leicester

Systematising Configurations

We have seen that

- Complex interaction protocols can be described by configurations, independently of the concrete components they will be applied to; they can be used in different contexts
- The use of such interaction protocols in a given configuration corresponds to defining the way in which the generic participating components are refined by the concrete components

Systematising Configurations

51

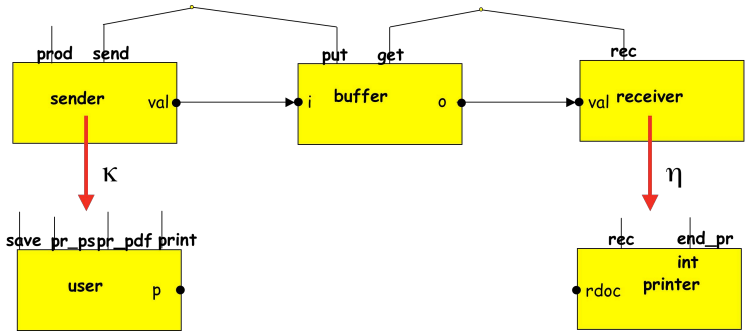
We have seen that

- Complex interaction protocols can be described by configurations, independently of the concrete components they will be applied to; they can be used in different contexts
- The use of such interaction protocols in a given configuration corresponds to defining the way in which the generic participating components are refined by the concrete components

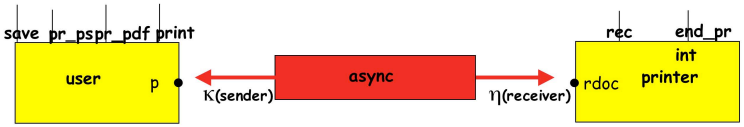
Connector Types

Instantiation of Connectors

VALENCIA 2005
University of Leicester



We may elevate the abstractions used to describe configurations...



... and define them in terms of computational components and connectors

4

The Distribution Dimension

However

- **Mobility**

  A new factor of complexity in the development of software systems (Web, mobile communication) that cannot be relegated to lower level design.

  As components move across a network, the connectors in place may no longer ensure the required interactions.

Motivation

- Software systems have to deal with **Highly Dynamic Environments**

Network connectivity	CPU	Directory Information
Bandwidth	Memory	Printers
Battery Power	Screen size	…

- **Change of environment** due to
  - **Mobility** of components
  - **Mobility** of hosts
  - Variety of devices
  - …

Motivation

Approaches to deal with the **Dynamics of Environment**

- **Traditional approach**: based on **Exceptions**.

  Systems are developed for being executed in particular conditions; at runtime, different conditions are unexpected and are considered exceptions.

- **Context-aware Paradigm**: based on a notion of **Context**.

  Systems have means to observe the surrounding environment and are developed taking into account different conditions in which they can be executed.

Context-awareness in the Design of Mobile Systems

Formalisms for designing mobile systems should support context-awareness and

- consider contexts as **first-class entities**
- support the **explicit design** of individualized contexts
- support the **separation** between

  context **sensing** and deliver to the system and **using** contexts

and this requires to identify

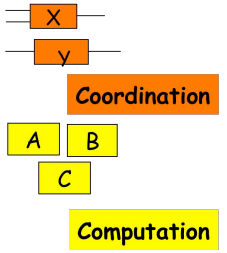
- Essential features of contexts
- Design Primitives for defining contexts
- Proper abstractions for modelling context-awareness

Key ideas for Mobility

- New forms of coordination that have emerged in mobile computing sa transient interaction and remote evaluation can be modelled through connectors
- Distribution can be separated from Coordination and Computation
- Distribution connectors can be offered as architectural primitives
- Location-aware architectural models can be developed incrementally through the refinement of higher-level descriptions that abstract from mobility

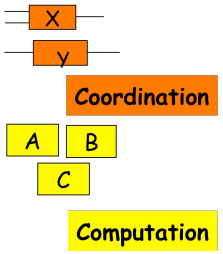
Motivation

Architecture-based approaches



Motivation

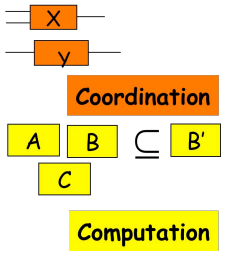
Architecture-based approaches



Compositionality wrt refinement

Motivation

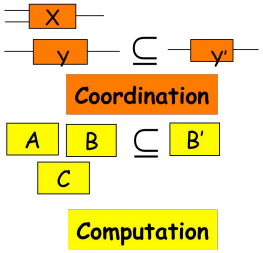
Architecture-based approaches



Compositionality wrt refinement

Motivation

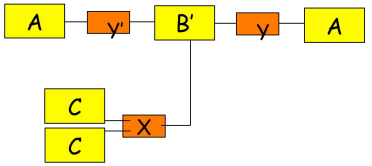
Architecture-based approaches

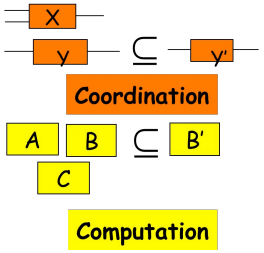


Compositionality wrt refinement

Motivation

Architecture-based approaches

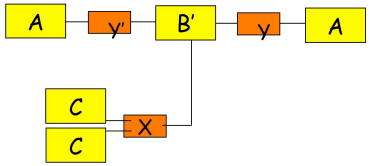


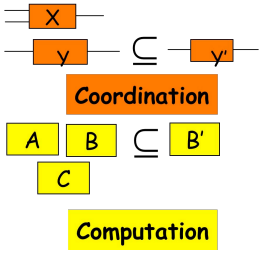


Compositionality wrt refinement, evolution

Motivation

Architecture-based approaches



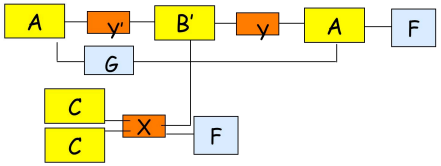


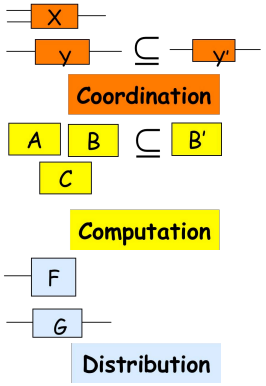
Compositionality wrt refinement, evolution

Reconfigurability / Adaptability

Motivation :Goal

Architectural Approach to Mobility

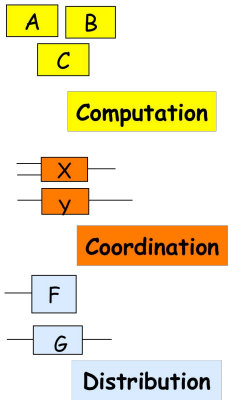




Compositionality wrt refinement, evolution

Reconfigurability / Adaptability

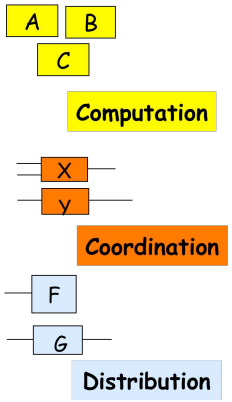
The Key Ingredients of Contexts



The Key Ingredients of Contexts

- Computations, as performed by individual components, are constrained by the **resources and services available** at the positions where the components are located

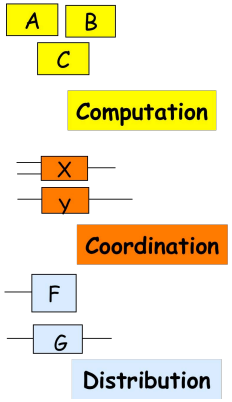
  a piece of mobile code that relies on numerical operations with high-precision will fail to compute when executing in locations where memory is scarce either because the available memory is not enough or because the operation is not even available at that location



The Key Ingredients of Contexts

- Computations, as performed by individual components, are constrained by the **resources and services available** at the positions where the components are located
- Communication among components can only take place when they are located in positions that are **in touch** with each other

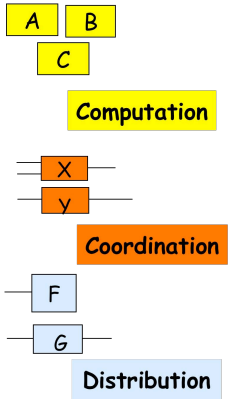
  the physical links that support comm between the positions of the space of mobility may be subject to failures or interruptions, making communication temporarily impossible



The Key Ingredients of Contexts

- Computations, as performed by individual components, are constrained by the **resources and services available** at the positions where the components are located
- Communication among components can only take place when they are located in positions that are **in touch** with each other
- Movement of components from one position to another is constrained by **reachability**

  typically the space has some structure given by walls and doors or barriers erected in communication networks by system administrators



Active Observables

We require that **Cxt** includes

- **rssv: ->nat$_\infty$x2$^\Omega$**

  to represent the resources and services that are available for computation

- **bt: ->2^{Loc}**

  to represent the set of locations that can be reached through communication

- **reach: ->2^{Loc}**

  to represent the set of locations that can be reached through movement

Adaptive and Embedded Systems?

Example. Designing an airport luggage handling system

```
design cart is
prv busy:bool
do move[]: ¬busy, false → true
[] dock[busy]: ¬busy, false → busy'
[] undock[busy]: busy, false → ¬busy'
```


Adaptive and Embedded Systems?

9

Example. Designing an airport luggage handling system

```
design cart is
prv busy:bool
do move[]: ¬busy, false → true
[] dock[busy]: ¬busy, false → busy'
[] undock[busy]: busy, false → ¬busy'
```

Private channel

Adaptive and Embedded Systems?

9

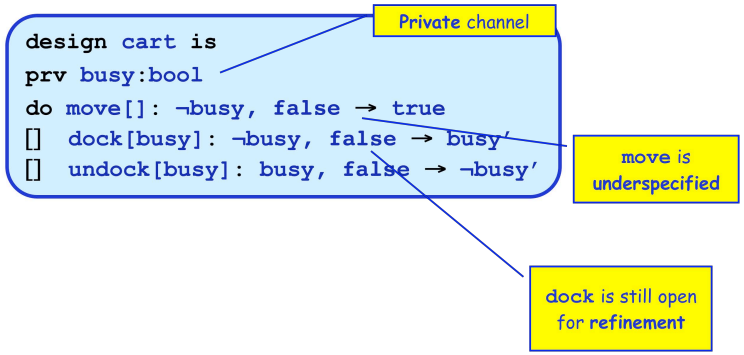
Example. Designing an airport luggage handling system

```
design cart is
prv busy:bool
do move[]: ¬busy, false → true
[] dock[busy]: ¬busy, false → busy'
[] undock[busy]: busy, false → ¬busy'
```

Private channel

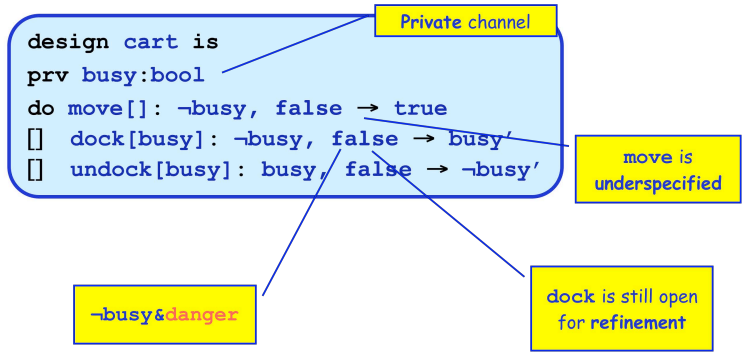
move is underspecified

Example. Designing an airport luggage handling system



Adaptive and Embedded Systems? 9

Example. Designing an airport luggage handling system



Location-awareness 10

Example. How the behaviour of a cart can be made location-dependent

```
design located cart is
inloc pos:Loc
in next:Loc
prv busy@pos:bool, dest@pos:Loc
do move@pos[]: ¬busy∧pos≠dest → true
[] dock@pos[busy]: ¬busy∧pos=dest → busy'
[] undock@pos[busy]:
          busy∧pos=dest → ¬busy'||dest'=next
```


Location-awareness 10

Example. How the behaviour of a cart can be made location-dependent

```
design located cart is
inloc pos:Loc
in next:Loc
prv busy@pos:bool, dest@pos:Loc
do move@pos[]: ¬busy∧pos≠dest → true
[] dock@pos[busy]: ¬busy∧pos=dest → busy'
[] undock@pos[busy]:
          busy∧pos=dest → ¬busy'||dest'=next
```

Position controlled by the environment

On the move…

Designs are defined in terms of extended **signatures**

—**channels** $x@l$ and **action names** $g@\{l_1,l_2,\ldots\}$

Position where the value is available

—**location variables** (input, output) typed over Loc with a distinguished λ

and **located actions**

$$g@l : L(g),U(g) \rightarrow R(g)$$

Position where code is executed

The Space of Mobility

- Explicit but not fixed representation of the space of mobility; can be modelled to fit the application domain:
  - Location variables have sort Loc, a special **data type**
  - The space consists of the set of possible values of Loc
- Mobility is associated with the movement of channels and actions (unit of mobility)

Semantics

The semantics of CommUnity designs is defined in terms of

- An algebra U for the data types
- An infinite sequence of pairs of binary relations over U_{Loc}

$$(bt_i,reach_i)_{i\in N}$$

— n bt m : n and m are positions "in touch" with each other

Coordination among components takes place only when they are in touch with each other

— n reach m : position n is reachable from m

Movement of a component to a new position is possible only when this position is reachable from the current one

Example

Example. Controlling how a cart moves

```
design controlled located cart is
outloc pos:Loc
inloc  cpoint:Loc
in  next:Loc
prv busy@pos:bool, dest@pos:Loc,
      in@cpoint:bool, mode@pos:[slow,fast]
do move@pos: ¬busy∧pos≠dest → c(pos,pos',mode)
[] dock@pos: ¬busy∧pos=dest → busy'
[] undock@pos: busy∧pos=dest → ¬busy'‖dest'=next
[] prv enter @pos: true → mode'=slow
            @cpoint: ¬in → in'
[] prv leave @pos: true → mode'=fast
            @cpoint: in → ¬in'
```

Example

Example. Controlling how a cart moves

```
design controlled located cart is
outloc pos:Loc
inloc  cpoint:Loc
in  next:Loc
prv busy@pos:bool, dest@pos:Loc,
       in@cpoint:bool, mode@pos:[slow,fast]
do move@pos: ¬busy∧pos≠dest → c(pos,pos',mode)
[]  dock@pos: ¬busy∧pos=dest → busy'
[]  undock@pos: busy∧pos=dest → ¬busy'‖dest'=next
[]  prv enter @pos: true → mode'=slow
            @cpoint: ¬in → in'
[]  prv leave @pos: true → mode'=fast
            @cpoint: in → ¬in'
```

reach

University of Leicester

Example

14

Example. Controlling how a cart moves

```
design controlled located cart is
outloc pos:Loc
inloc  cpoint:Loc
in  next:Loc
prv busy@pos:bool, dest@pos:Loc,
       in@cpoint:bool, mode@pos:[slow,fast]
do move@pos: ¬busy∧pos≠dest → c(pos,pos',mode)
[]  dock@pos: ¬busy∧pos=dest → busy'
[]  undock@pos: busy∧pos=dest → ¬busy'‖dest'=next
[]  prv enter @pos: true → mode'=slow
            @cpoint: ¬in → in'
[]  prv leave @pos: true → mode'=fast
            @cpoint: in → ¬in'
```

reach

bt

VALENCIA 2005

University of Leicester

Externalisation of the distribution aspects


```
design controlled located cart is
outloc pos:Loc
inloc  cpoint:Loc
in  next:Loc
prv busy@pos:bool, dest@pos:Loc,
       in@cpoint:bool, mode@pos:[slow,fast]
do move@pos: ¬busy∧pos≠dest → c(pos,pos',mode)
[]  dock@pos: ¬busy∧pos=dest → busy'
[]  undock@pos: busy∧pos=dest → ¬busy'‖dest'=next
[]  prv enter @pos: true → mode'=slow
            @cpoint: ¬in → in'
[]  prv leave @pos: true → mode'=fast
            @cpoint: in → ¬in'
```


University of Leicester

Externalisation of the distribution aspects

15

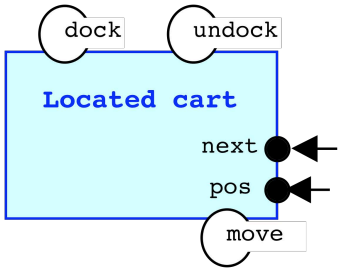
```
design controlled located cart is
outloc pos:Loc
inloc  cpoint:Loc
in  next:Loc
prv busy@pos:bool, dest@pos:Loc,
       in@cpoint:bool, mode@pos:[slow,fast]
do move@p
[]  dock@
[]  undoc
[]  prv e

[]  prv l
```

```
design mode controller is
outloc theirs:Loc
inloc  mine:Loc
prv in@mine:bool, mode@theirs:[slow,fast]
do control@theirs: true → c(theirs,theirs',mode)
[]  prv enter @theirs: true → mode'=slow
            @mine: ¬in → in'
[]  prv leave @theirs : true → mode'=fast
            @mine: in → ¬in'
```

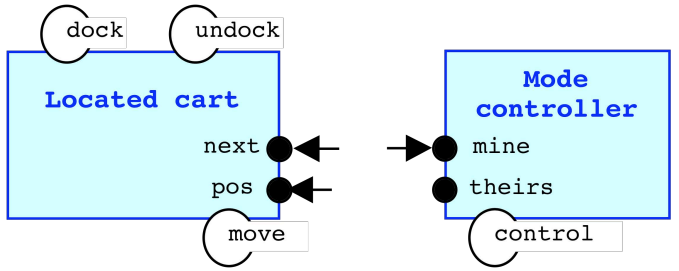
VALENCIA 2005

University of Leicester

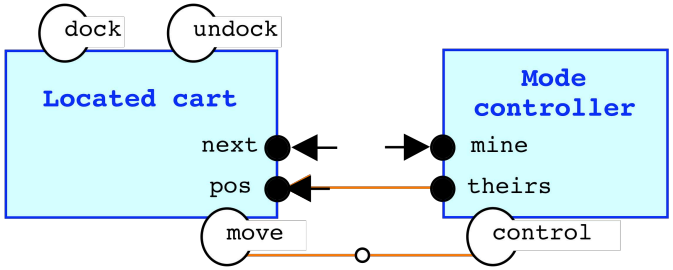
Architectural design



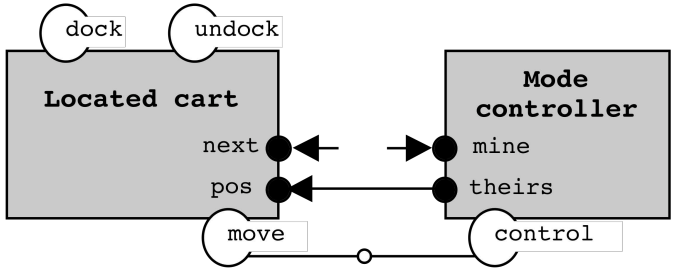
Architectural design



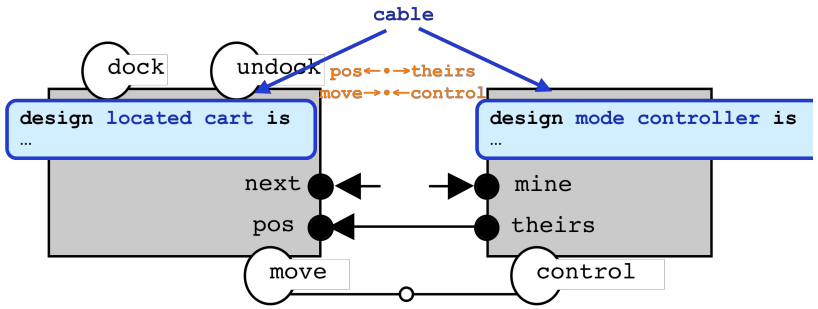
Architectural design



Categorical semantics

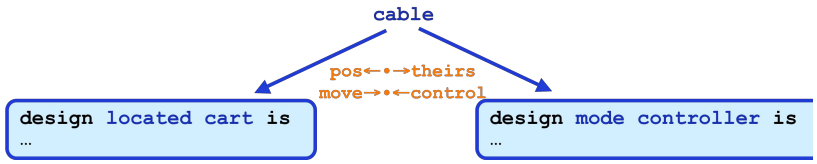


Categorical semantics



Categorical semantics

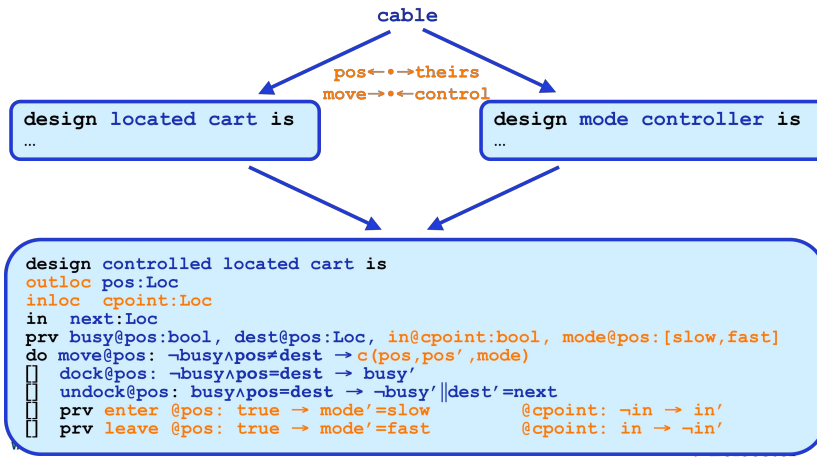
17



VALENCIA 2005
University of Leicester

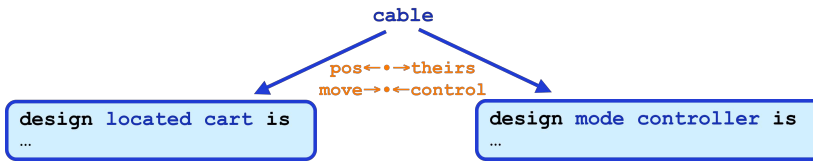
Categorical semantics

17



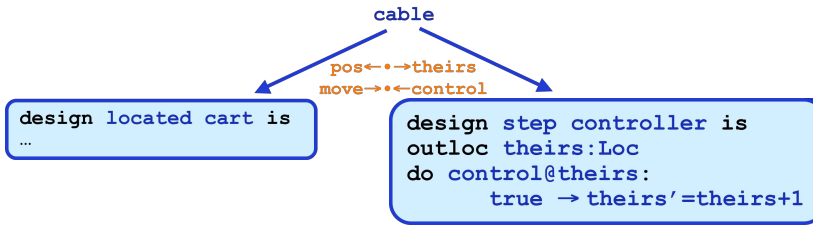
Reconfigurability / adaptativity

18

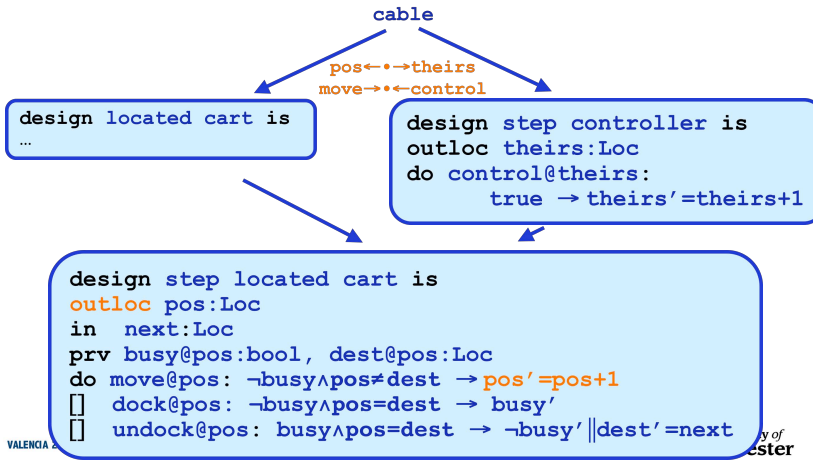


VALENCIA 2005
University of Leicester

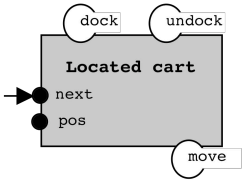
Reconfigurability / adaptativity



Reconfigurability / adaptativity

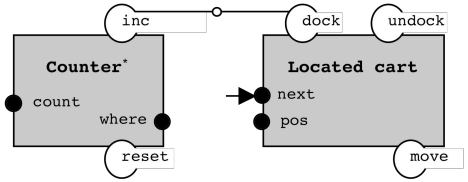


Separation of concerns



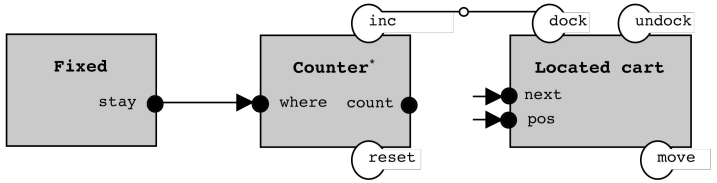

```
design counter is
inloc  where:Loc
out    count@where:nat
do inc@where: true → count'=count+1
[] reset@where: true → count'=0
```


Separation of concerns




```
design counter is
inloc  where:Loc
out    count@where:nat
do inc@where: true → count'=count+1
[] reset@where: true → count'=0
```

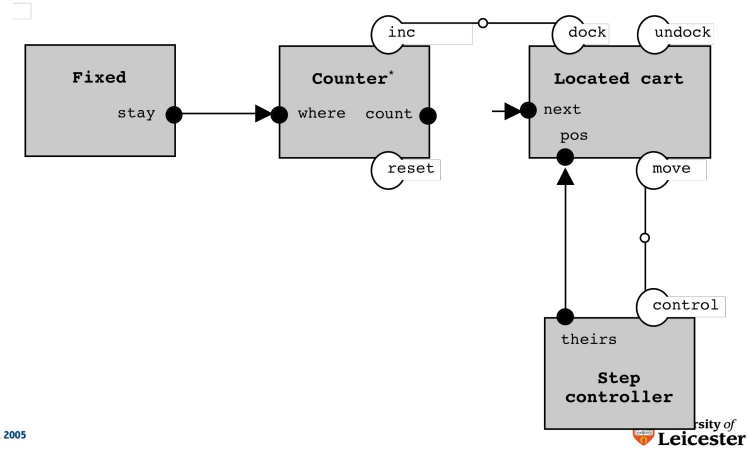
Separation of concerns




```
design fixed is
outloc  stay:Loc
```


Separation of concerns

19



VALENCIA 2005
University of Leicester

Another example

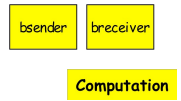
- A **sender-receiver system** in which the sender produces, in one go, words of N bits that are then transmitted, one by one, to the receiver through synchronous message passing.
- The sender is fixed and the receiver is a mobile component: once a word defining a location is received, the component should move its execution to that location; if this is not possible, it discards that word and starts the reception of another one.

VALENCIA 2005
University of Leicester

Another example

20

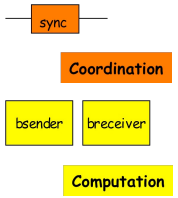
- A **sender-receiver system** in which the sender produces, in one go, words of N bits that are then transmitted, one by one, to the receiver through synchronous message passing.
- The sender is fixed and the receiver is a mobile component: once a word defining a location is received, the component should move its execution to that location; if this is not possible, it discards that word and starts the reception of another one.



VALENCIA 2005
University of Leicester

Another example

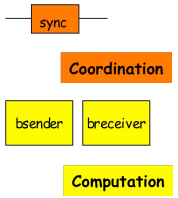
- A **sender-receiver system** in which the sender produces, in one go, words of N bits that are then transmitted, one by one, to the receiver through synchronous message passing.
- The sender is fixed and the receiver is a mobile component: once a word defining a location is received, the component should move its execution to that location; if this is not possible, it discards that word and starts the reception of another one.

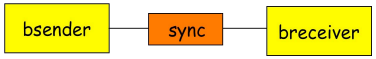


Another example

20

- A **sender-receiver system** in which the sender produces, in one go, words of N bits that are then transmitted, one by one, to the receiver through synchronous message passing.
- The sender is fixed and the receiver is a mobile component: once a word defining a location is received, the component should move its execution to that location; if this is not possible, it discards that word and starts the reception of another one.



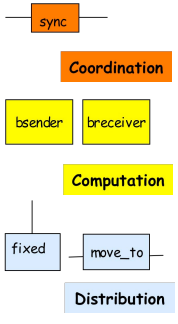


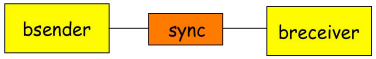
VALENCIA 2005
University of Leicester

Another example

20

- A **sender-receiver system** in which the sender produces, in one go, words of N bits that are then transmitted, one by one, to the receiver through synchronous message passing.
- The sender is fixed and the receiver is a mobile component: once a word defining a location is received, the component should move its execution to that location; if this is not possible, it discards that word and starts the reception of another one.



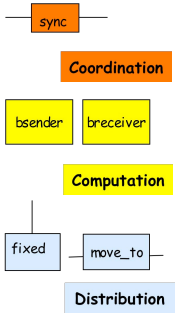


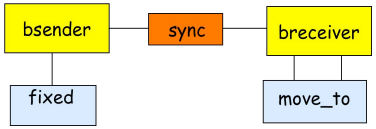
VALENCIA 2005
University of Leicester

Another example

20

- A **sender-receiver system** in which the sender produces, in one go, words of N bits that are then transmitted, one by one, to the receiver through synchronous message passing.
- The sender is fixed and the receiver is a mobile component: once a word defining a location is received, the component should move its execution to that location; if this is not possible, it discards that word and starts the reception of another one.





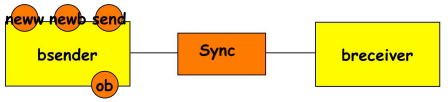
VALENCIA 2005
University of Leicester

Coordination concerns

Example. A simple sender-receiver system

Coordination concerns

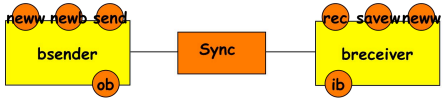
Example. A simple sender-receiver system




```
design bsender is
out ob:bit
prv w:array(N,bit), k:nat, rd:bool
do  neww:k=N→w:∈array(N,bit)‖k:=0
[]  newb:¬rd∧k<N→rd:=true‖ob:=word[k]‖k:=k+1
[]  send:rd→rd:=false
```


Coordination concerns

Example. A simple sender-receiver system

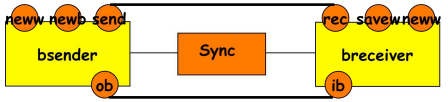



```
design bsender
out ob:bit
prv w:array(N,
do  neww:k=N→w
[]  newb:¬rd∧k
[]  send:rd→rd
```

```
design breceiver is
in  ib:bit
out w:array(N,bit), k:nat
prv recw:array(N,bit), rd:bool
do  rec:k<N→recw[k]:=ib‖k:=k+1‖rd:=false
[]  savew:¬rd∧k=N→rd:=true‖w:=recw
[]  neww:rd∧k=N→rd:=false‖k:=0
```


Coordination concerns

Example. A simple sender-receiver system

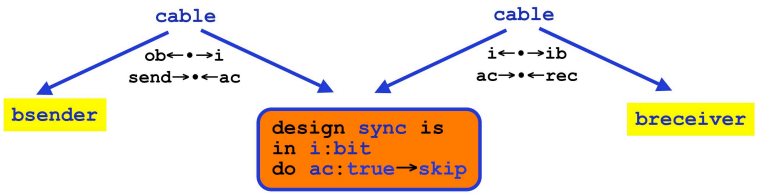



```
design bsender
out ob:bit
prv w:array(N,
do  neww:k=N→w
[]  newb:¬rd∧k
[]  send:rd→rd
```

```
design breceiver is
in  ib:bit
out w:array(N,bit), k:nat
prv recw:array(N,bit), rd:bool
do  rec:k<N→recw[k]:=ib‖k:=k+1‖rd:=false
[]  savew:¬rd∧k=N→rd:=true‖w:=recw
[]  neww:rd∧k=N→rd:=false‖k:=0
```

The configuration

Example. A simple sender-receiver system



Colimit semantics


```
design sync-send-rec is
out b:bit, w_r:array(N,bit), k_r:nat
prv rd_r,rd_s: bool, recw_r, w_s:array(N,bit), k_s:nat
do  sendrec:rd_s∧k_r<N →rd_s:=false‖word_r[k_r]:=b ‖k_r:=k_r+1‖rd_r:=false
[]  savew:¬rd_r∧k_r=N→rd_r:=true‖w_r:=word
[]  rneww:rd_r∧k_r=N→rd_r:=false‖k_r:=0
[]  sneww:k_s=N→w_s:∈array(N,bit)‖k_s:=0
[]  newb:¬rd_s∧k_s<N→rd_s:=true‖b:=w_s[k_s]‖k_s:=k_s+1
```


Making designs location-aware

Example. A mobile bit receiver that once a word defining a location is received, moves to that location

```
design mobreceiver is
outloc l
in  ib:bit
out w@l:array(N,bit), k@l:nat
prv recw@l:array(N,bit), rd@l:bool
do  rec@l:k<N→recw[k]:=ib ‖k:=k+1‖rd:=false
[]  savew@l:¬rd∧k=N→rd:=true‖w:=recw
[]  neww@l:rd∧k=N→rd:=false‖k:=0‖l:=if(loc?(w),loc(w),l)
```

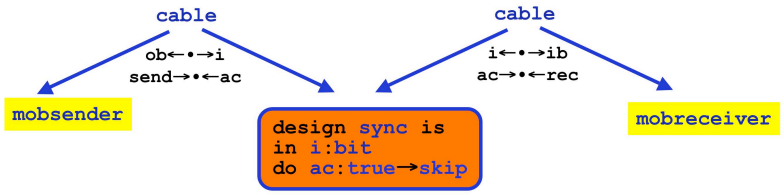

Making designs location-aware

Example. A fixed sender i.e. placed at a fixed position

```
design mobsender is
outloc l
out ob@l:bit
prv w@l:array(N,bit), k@l:nat, rd@l:bool
do  neww@l:k=N→w:∈array(N,bit)‖k:=0
[]  newb@l:¬rd∧k<N→rd:=true‖ob:=w[k]‖k:=k+1
[]  send@l:rd→rd:=true
```

Making architectures location-aware

Example. A location-aware version of the sender-receiver system with the previous mobile receiver and a fixed sender



Making architectures location-aware

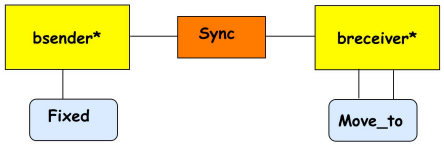

```
design mobsys is
outloc ls, lr
out b@ls:bit, wr@lr:array(N,bit), kr@lr:nat
prv rdr@lr,rds@ls: bool, recwr@lr, ws@ls:array(N,bit), ks@ls:nat
do  sendrec@ls:rds→rds:=true
          @lr:kr<N→wordr[kr]:=b ‖kr:=kr+1‖rdr:=false
[]  savew@lr:¬rdr∧kr=N→rdr:=true‖wr:=word
[]  rneww@lr:rdr∧kr=N→rdr:=false‖kr:=0‖lr:=if(loc?(wr),loc(wr),lr)
[]  sneww@ls:ks=N→ws:∈array(N,bit)‖ks:=0
[]  newb@ls:¬rds∧ks<N→rds:=true‖b:=ws[ks]‖ks:=ks+1
```


Making architectures location-aware

- Finer-grain model obtained in an intrusive way - mobility requirements taken into account by rewriting the components (superposition of behaviour)
- Distribution and mobility aspects of the system are not explicitely represented in the architecture; they cannot be refined or evolved independently of the architectural elements

Externalisation of distribution

By making use of distribution connectors



- bsender* and breceiver*: extensions of bsender and breceiver with an input location variable assigned to every constituent
- Move_to: connector with two roles a glue defining the movement of the subject_of_move to the destination provided by the dest_provider
- Fixed: connector with one role subject and a glue defining that subject is a non mobile component

Making designs location-aware


```
design mobsender is
outloc l
out ob@l:bit
prv w@l:array(N,bit), k@l:nat, rd@l:bool
do  neww@l:k=N→w:∈array(N,bit)‖k:=0
[]  newb@l:¬rd∧k<N→rd:=true‖ob:=w[k]‖k:=k+1
[]  send@l:rd→rd:=true
```


Making designs location-aware


```
design bsender is
out ob:bit
prv w:array(N,bit), k:nat, rd:bool
do  neww:k=N→w:∈array(N,bit)‖k:=0
[]  newb:¬rd∧k<N→
      rd:=true‖ob:=word[k]‖k:=k+1
[]  send:rd→rd:=false
```

```
design mobsender is
outloc l
out ob@l:bit
prv w@l:array(N,bit), k@l:nat, rd@l:bool
do  neww@l:k=N→w:∈array(N,bit)‖k:=0
[]  newb@l:¬rd∧k<N→rd:=true‖ob:=w[k]‖k:=k+1
[]  send@l:rd→rd:=true
```


Making designs location-aware


```
design bsender is
out ob:bit
prv w:array(N,bit), k:nat, rd:bool
do  neww:k=N→w:∈array(N,bit)‖k:=0
[]  newb:¬rd∧k<N→
      rd:=true‖ob:=word[k]‖k:=k+1
[]  send:rd→rd:=false
```

```
design fixed is
outloc l
```

```
design mobsender is
outloc l
out ob@l:bit
prv w@l:array(N,bit), k@l:nat, rd@l:bool
do  neww@l:k=N→w:∈array(N,bit)‖k:=0
[]  newb@l:¬rd∧k<N→rd:=true‖ob:=w[k]‖k:=k+1
[]  send@l:rd→rd:=true
```


Making designs location-aware


```
design bsender is
out ob:bit
prv w:array(N,bit), k:nat, rd:bool
do  neww:k=N→w:∈array(N,bit)‖k:=0
[]  newb:¬rd∧k<N→
      rd:=true‖ob:=word[k]‖k:=k+1
[]  send:rd→rd:=false
```

```
design fixed is
outloc l
```

cable2

Making designs location-aware


```
design mobreceiver is
outloc l
in  ib:bit
out w@l:array(N,bit), k@l:nat
prv recw@l:array(N,bit), rd@l:bool
do  rec@l:k<N→recw[k]:=ib ‖k:=k+1‖rd:=false
[]  savew@l:¬rd∧k=N→rd:=true‖w:=recw
[]  neww@l:rd∧k=N→rd:=false‖k:=0‖l:=if(loc?(w),loc(w),l)
```


Making designs location-aware


```
design breceiver is
in  ib:bit
out w:array(N,bit), k:nat
prv recw:array(N,bit), rd:bool
do  rec:k<N→recw[k]:=ib ‖k:=k+1‖rd:=false
[]  savew:¬rd∧k=N→rd:=true‖w:=recw
[]  neww:rd∧k=N→rd:=false‖k:=0
```

```
design mobreceiver is
outloc l
in  ib:bit
out w@l:array(N,bit), k@l:nat
prv recw@l:array(N,bit), rd@l:bool
do  rec@l:k<N→recw[k]:=ib ‖k:=k+1‖rd:=false
[]  savew@l:¬rd∧k=N→rd:=true‖w:=recw
[]  neww@l:rd∧k=N→rd:=false‖k:=0‖l:=if(loc?(w),loc(w),l)
```


Making designs location-aware


```
design breceiver is
in  ib:bit
out w:array(N,bit), k:nat
prv recw:array(N,bit), rd:bool
do  rec:k<N→recw[k]:=ib ‖k:=k+1‖rd:=false
[]  savew:¬rd∧k=N→rd:=true‖w:=recw
[]  neww:rd∧k=N→rd:=false‖k:=0
```

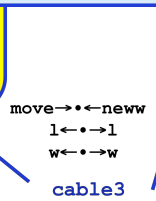
```
design move_to is
outloc l
in w:array(bit,N)
do move@l: l:=if(loc?(w),loc(w),l)
```

```
design mobreceiver is
outloc l
in  ib:bit
out w@l:array(N,bit), k@l:nat
prv recw@l:array(N,bit), rd@l:bool
do  rec@l:k<N→recw[k]:=ib ‖k:=k+1‖rd:=false
[]  savew@l:¬rd∧k=N→rd:=true‖w:=recw
[]  neww@l:rd∧k=N→rd:=false‖k:=0‖l:=if(loc?(w),loc(w),l)
```

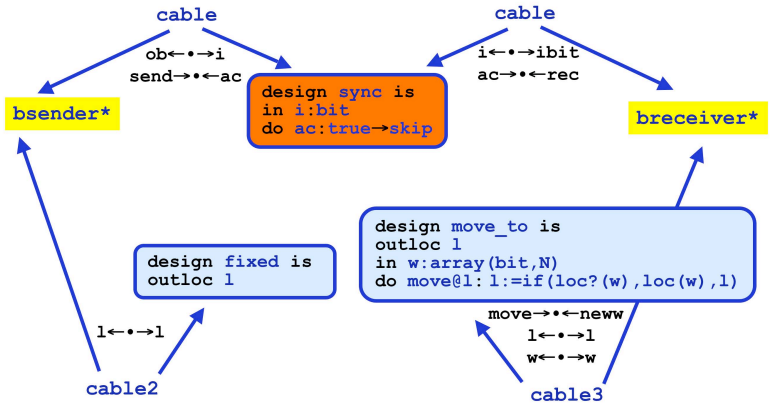

Making designs location-aware


```
design breceiver is
in  ib:bit
out w:array(N,bit), k:nat
prv recw:array(N,bit), rd:bool
do  rec:k<N→recw[k]:=ib ‖k:=k+1‖rd:=false
[]  savew:¬rd∧k=N→rd:=true‖w:=recw
[]  neww:rd∧k=N→rd:=false‖k:=0
```

```
design move_to is
outloc l
in w:array(bit,N)
do move@l: l:=if(loc?(w),loc(w),l)
```



Externalisation of distribution



Supporting Incremental Development

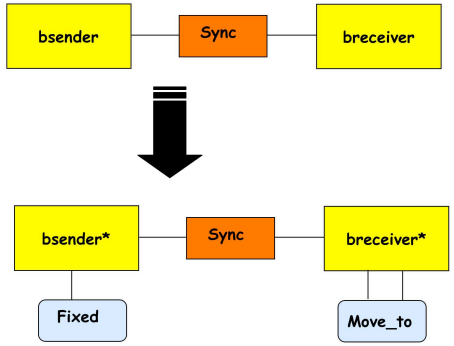


Diagram in DSGN

Diagram in MDSGN

Supporting Incremental Development

Example. Extensions of mob(breceiver)

- breceiver*- extension with one input location variable assigned to every channel and action; this form of extension implies that the component can only be moved as a whole
- breceiver#-extension with one input location variable for each action and for each channel; in this way we have means for them to be controlled independently
- ...

Supporting Incremental Development

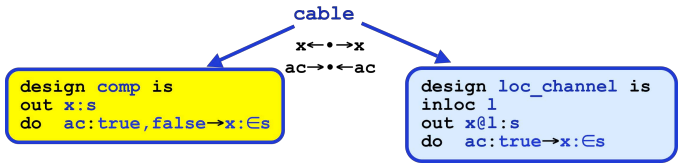
These extensions can be achieved through location connectors

- Purpose: to locate individual channels or actions
- Represent standard solutions that can be used across different components and connectors glues

Supporting Incremental Development

These extensions can be achieved through location connectors

- Purpose: to locate individual channels or actions
- Represent standard solutions that can be used across different components and connectors glues



However

- **Business**

  Many businesses are selling the same services through different channels, each of which has specific features.

  For instance, withdrawing money is subject to location-specific rules in addition to the coordination-related ones. It is different to withdraw money at your local branch, another branch, an ATM…

  The system should self-adapt to changes of location without interfering with the coordination business rules.

Location laws


```
location law ATM-withdrawal
locations    atm:ATMW-LI; bank:BANKW-LI
rules  when  atm.withdrawal(n) and BT(atm,bank)
       with  n≤bank.maxatm() and n≤atm.cash()
       do    atm.give(n)
       when  atm.withdrawal(n) and not BT(atm,bank)
                 and REACH(atm,bank)
       let   N=min(atm.default(),n) in
       with  N≤atm.cash()
       do    atm.give(N)
       mv    bank.internal(N,atn.acco())
end law
```


Be in touch


```
rules  when  atm.withdrawal(n) and BT(atm,bank)
       with  n≤bank.maxatm() and n≤atm.cash()
       do    atm.give(n)
```

- **BT** indicates whether the two locations are **"in touch"** meaning that they can communicate and synchronise actions at both locations;
- If they are, coordination laws apply to the partners that are located there. The guards (*with* conditions) of coordination and location rules apply, and the reactions of both are performed atomically.

Reach


```
when  atm.withdrawal(n) and not BT(atm,bank)
                  and REACH(atm,bank)
      let   N=min(atm.default(),n) in
      with  N≤atm.cash()
      do    atm.give(N)
      mv    bank.internal(N,atm.acco())
```

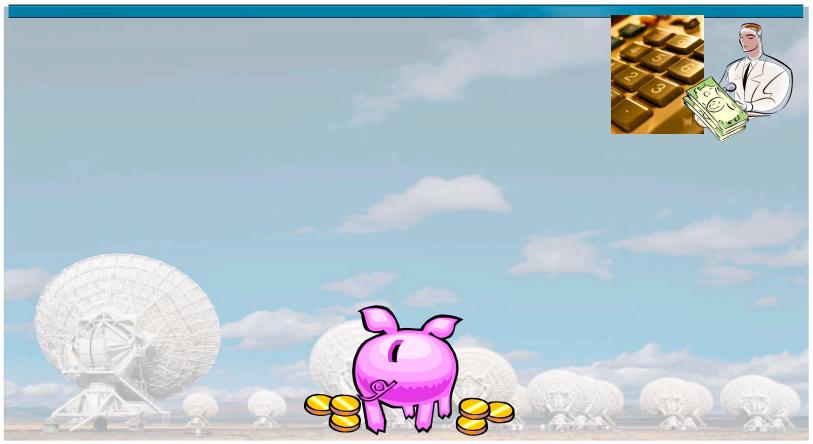
- **REACH** indicates that one location can be **"reached"** from the other, meaning that services can move across;
- **mv** indicates that the service is sent for execution at the other location.

Location interfaces

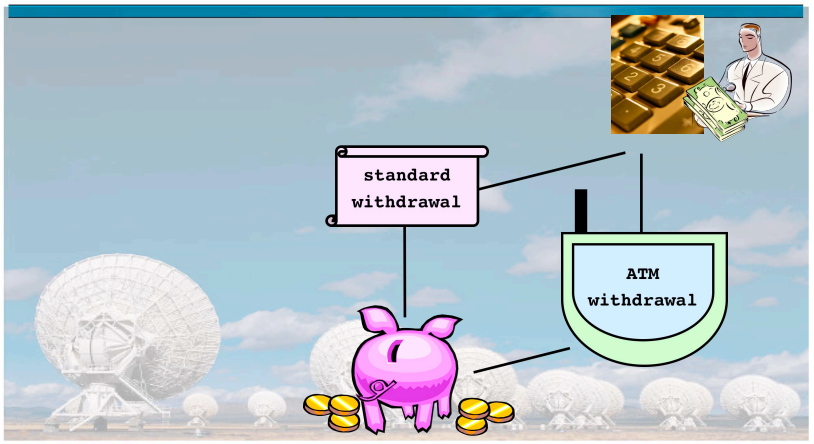

```
location interface BANKW-LI
location type     BANK
services      internal(n:money,a:ACCOUNT)
              maxatm():money
end interface
```

```
location interface ATMW-LI
location type     ATM
services      default(),cash():money,acco():ACCOUNT
              give(n:money) post cash()=oldcash()-n
events        withdraw(n:money)
end interface
```

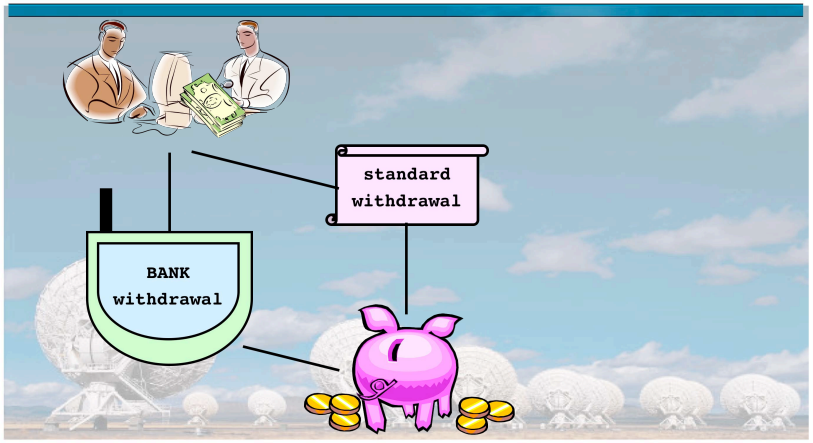

Instantiation



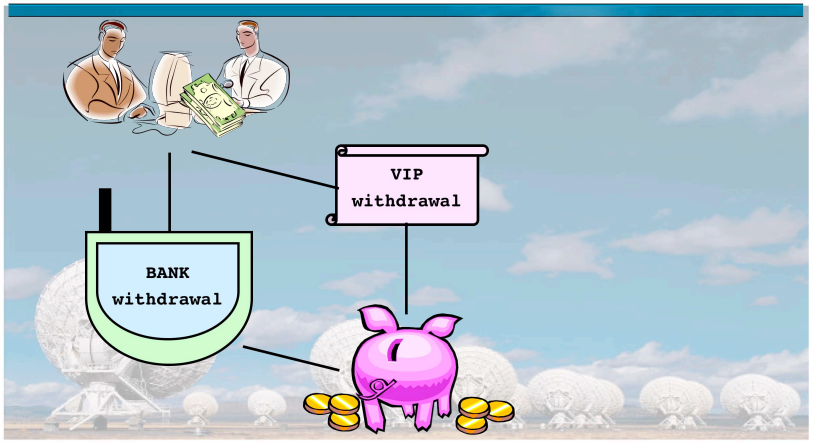
Instantiation



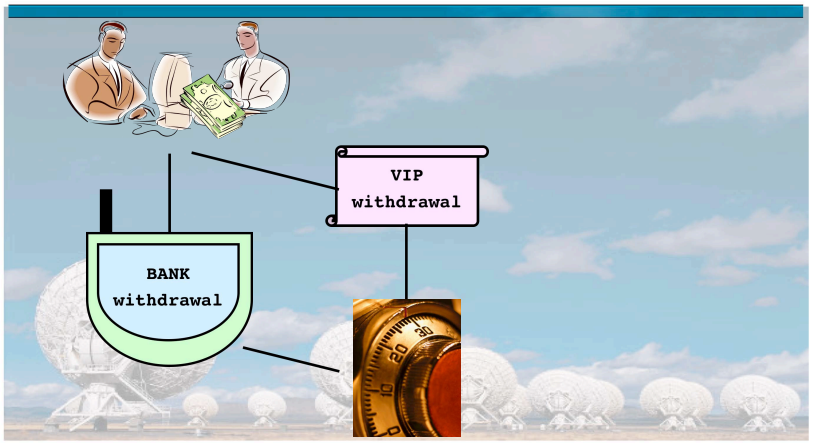
Instantiation



Instantiation



Instantiation



Other Work

- Operational semantics in KLAIM
- Reconfiguration with Hypergraph Rewriting
- Refinement in Tile Logic
- Domain specific extensions
  (embedded, collaborative, event-based, …)

See the movie!

See the movie!

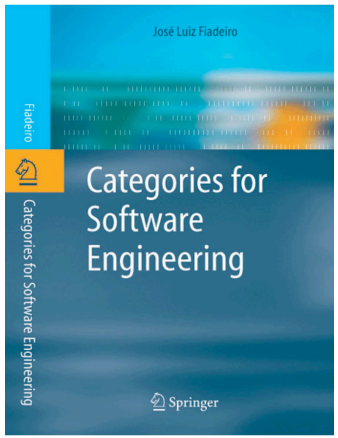
43

www.fiadeiro/jose/CommUnity

VALENCIA 2005

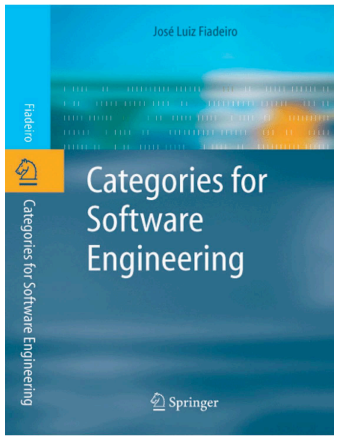
University of Leicester

Buy the book!



Buy the book!

44



Categories for Software Engineering

Springer 2004

ISBN 3-540-20909-3

VALENCIA 2005

University of Leicester

Get the t-shirt (www.cs.le.ac.uk)

Get the t-shirt (www.cs.le.ac.uk)

45

- **Lectureships** in Software Engineering

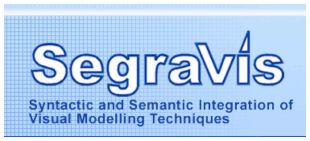
VALENCIA 2005

University of Leicester

Get the t-shirt (www.cs.le.ac.uk)

45

- **Lectureships** in Software Engineering



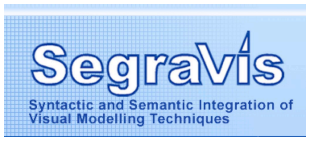
VALENCIA 2005

University of Leicester

Get the t-shirt (www.cs.le.ac.uk)

45

- **Lectureships** in Software Engineering
- Research assistantships on **SENSORIA**



VALENCIA 2005

University of Leicester