Defining a Specification Language for Distributed Components

Antonio CANSADO
Eric MADELAINE
INRIA – Sophia Antipolis

The Problem

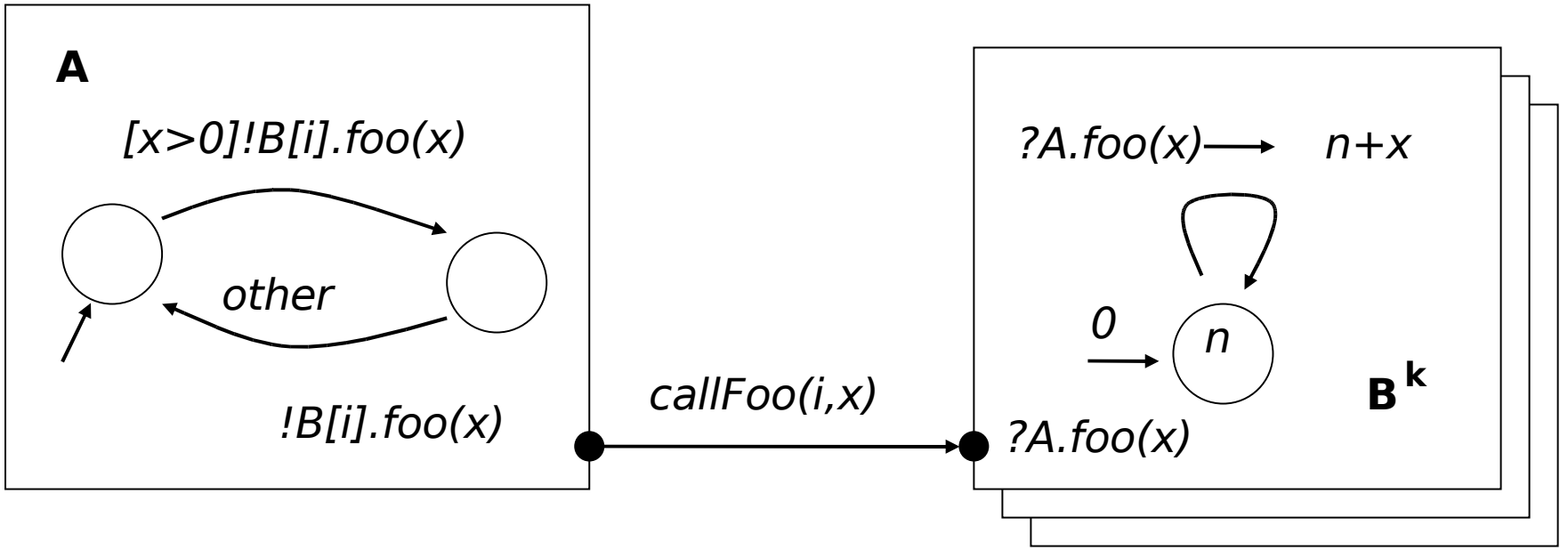
- What to specify?
  - Distributed Components – GRIDs, GCM
    - Asynchronous calls
    - Multiple components
      - Collective Interfaces
  - Dynamic structures
- Who are the users?
  - Software Engineers
    - UML, Java, XML

Basic Ideas

- Specify the assembly of hierarchical components
  - Components, required and provided interfaces
  - Bindings
- Specify behaviour
  - Potential messages, synchronisation points
  - Collective interfaces (Multicast, Gathercast)

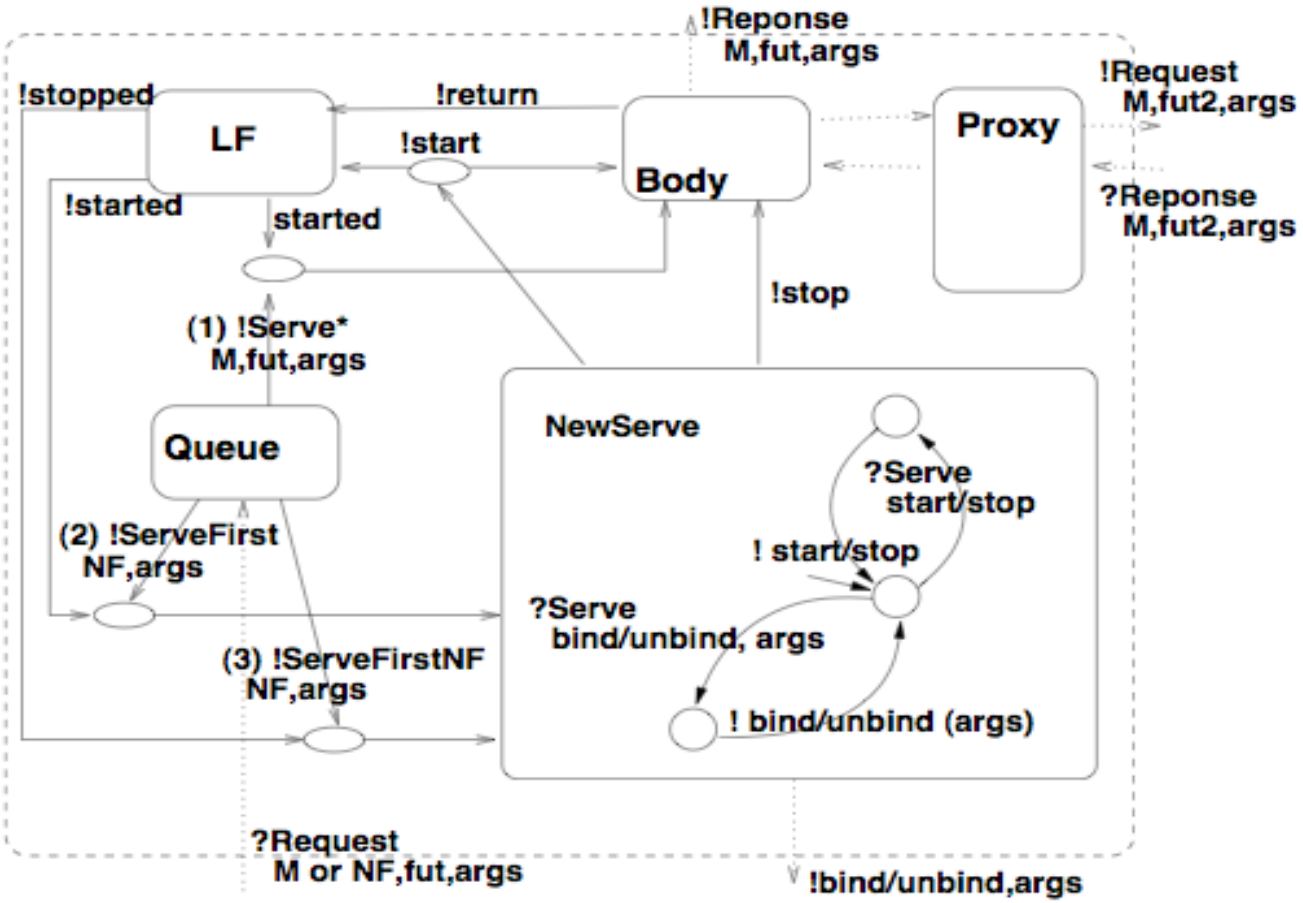
Basics

- Semantic model: pNets

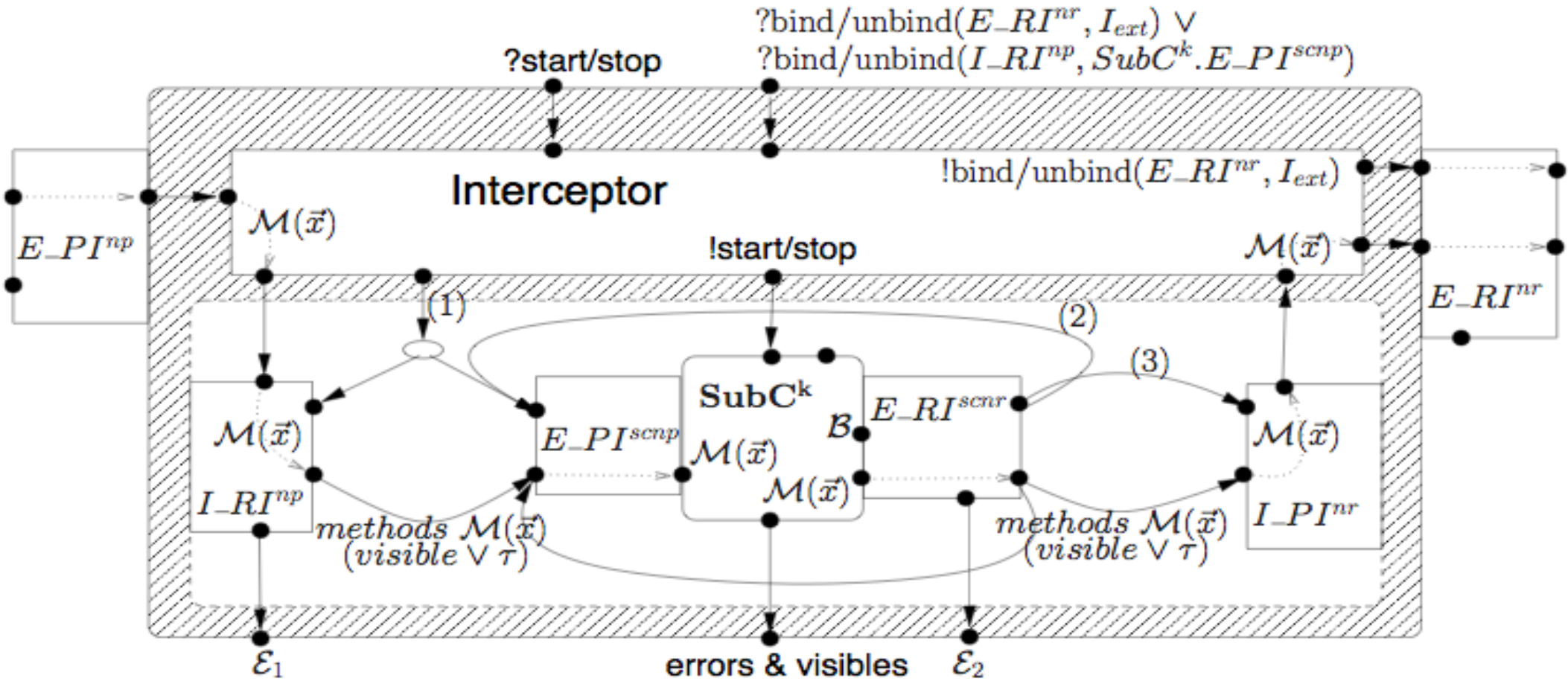


Primitive Component

- Previously defined models [FACS'05/06]



Composite Component



(1) !bind/unbind($I\_RI^{np}, SubC^k.E\_PI^{scnp}$)
(2) !bind/unbind($SubC^k.E\_RI^{scnr}, SubC^j.I\_PI^{scnr}$), $k \neq j$
(3) !bind/unbind($SubC^k.E\_RI^{scnr}, I\_PI^{nr}$)

What's wrong?

- Current approach (VERCORS)
  - Fractal ADL + behaviour (LOTOS)
    - Only static architectures
    - Different syntaxes and semantics
    - Up to the user to write “component-friendly” LOTOS code
    - Not clear at all what can and cannot be done
  - We have models, but we don't have a high-level language to express the behaviour!

What's wrong?

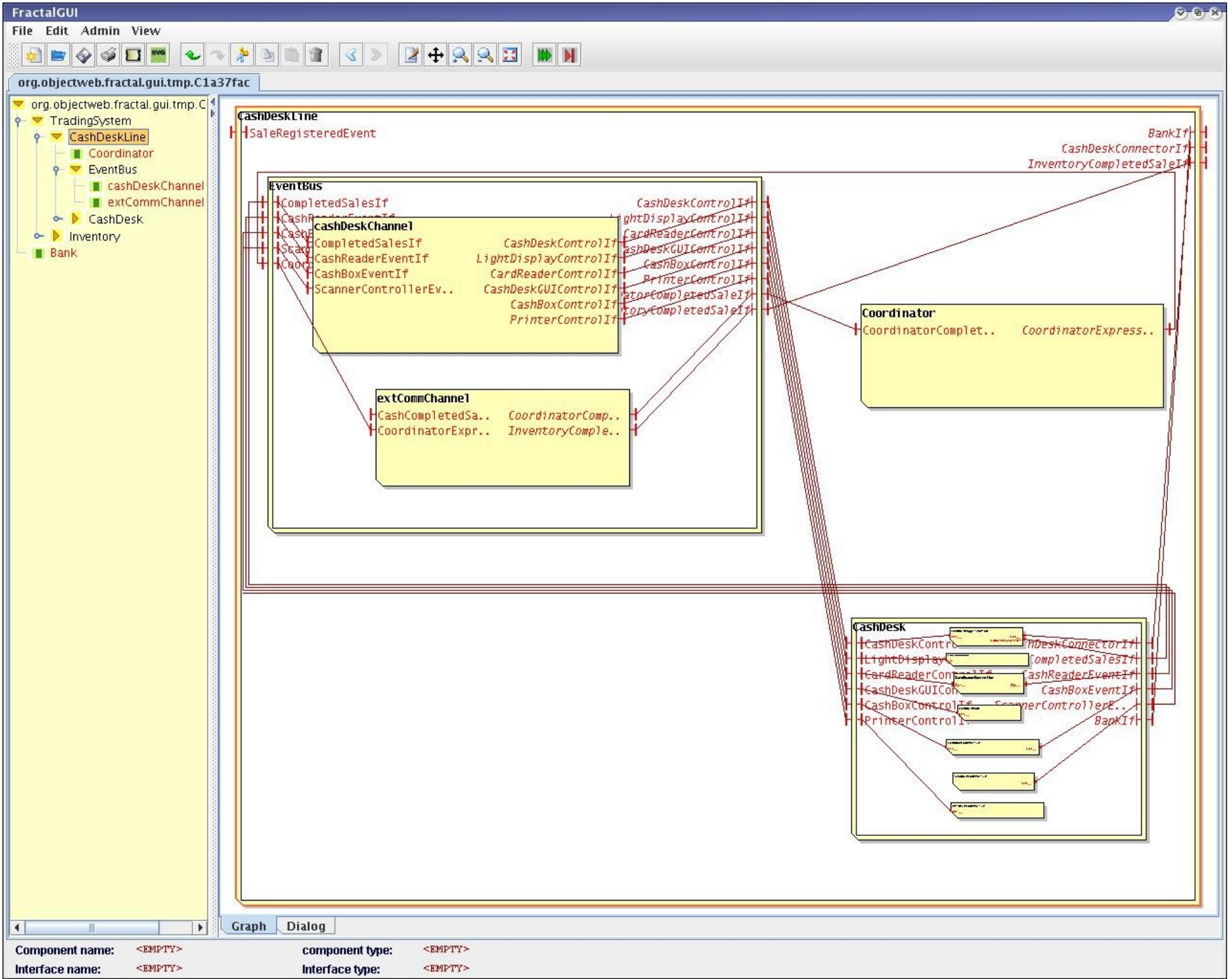
- Missing info in ADL
  - Useful only for component description and static deployment
  - Inadequate when dealing with multiple components
    - How to specify a binding between 2 specific components 'i' and '(i+1)%n'?
  - Difficult to soundly attach the component behaviour

CoCoME

- Provide a common example for Component-based software development (CBSD)
- Evaluate and compare the practical appliance of existing component models

Point-of-Sale (POS)

- M cashdesks, N stores
- N local databases, 1 main database
- Mandatory Features
  - Multiple components, collective interfaces
  - Asynchronous communications
- Optional Features
  - Dynamic reconfiguration
  - Fault tolerance


FractalGUI
File Edit Admin View
org.objectweb.fractal.gui.tmp.C1a37fac
org.objectweb.fractal.gui.tmp.C
TradingSystem
CashDeskLine
Coordinator
EventBus
cashDeskChannel
extCommChannel
CashDesk
Inventory
Bank
CashDeskLine
SaleRegisteredEvent
BankIf
CashDeskConnectorIf
InventoryCompletedSaleIf
EventBus
CompletedSalesIf
CashDeskControlIf
cashDeskChannel
CompletedSalesIf
CashReaderEventIf
CashBoxEventIf
ScannerControllerEv..
CashDeskControlIf
LightDisplayControlIf
CardReaderControlIf
CashDeskGUIControlIf
CashBoxControlIf
PrinterControlIf
extCommChannel
CashCompletedSa..
CoordinatorExpr..
CoordinatorComp..
InventoryComple..
Coordinator
CoordinatorComplet..
CoordinatorExpress..
CashDesk
Graph
Dialog
Component name: <EMPTY>
component type: <EMPTY>
Interface name: <EMPTY>
Interface type: <EMPTY>

Language Goals

- Friendly syntax to Software Engineers
- Generate Java code (ProActive control flow)
  - At least for simulation
- Generate behavioural model (pNets)
  - Only a subset of specifications will be verifiable

Fundamentals

- Component Software
- ProActive Middleware
  - Single control thread (Primitives)
  - No explicit synchronisation primitives
    - Transparent to the user
  - RDV, futures, wait-by-necessity
    - i.e., Data-flow synchronisation

Key!

Scope

- Considered
  - Architecture (composites, primitives)
  - Control flow, Data flow
  - Synchronisation points
  - Dynamic structures

Design

- Not considered
  - Data computation
  - Physical infrastructure
  - Non-functional exceptions (network problems)

Implementation

Middleware

How?

- Control flow, Data flow
  - Sequence, branching
    - Method calls
    - Return values
    - Data usage
      - Synchronisation points

Choices...

- Language semantics
  - UML
    - Component diagrams
    - Statechart diagrams

    CTTool
  - Java-like
    - Architecture definition
    - Code

    JDCSpec

Choices...

- 'Futures'
  - Explicit
    - !call(method,args);
    - ?use(value);

pNets

  - Implicit
    - value = interface.method(args);
    - print(value);

JDCSpec

Why implicit synchronisations?

- Allow switching from async to sync calls
  - Reuse specification for async and sync implementations
  - Check the effect of exceptions
    - Exceptions -- synchronous / barriers

User need

- Transparent first order continuations

```
client = databaseItf.getClient(id);
managerItf.check(client);
```

ProActive

Data types

- First order types
  - Integers
  - Intervals
  - Enumerated types
  - Records (without recursion)
  - Arrays

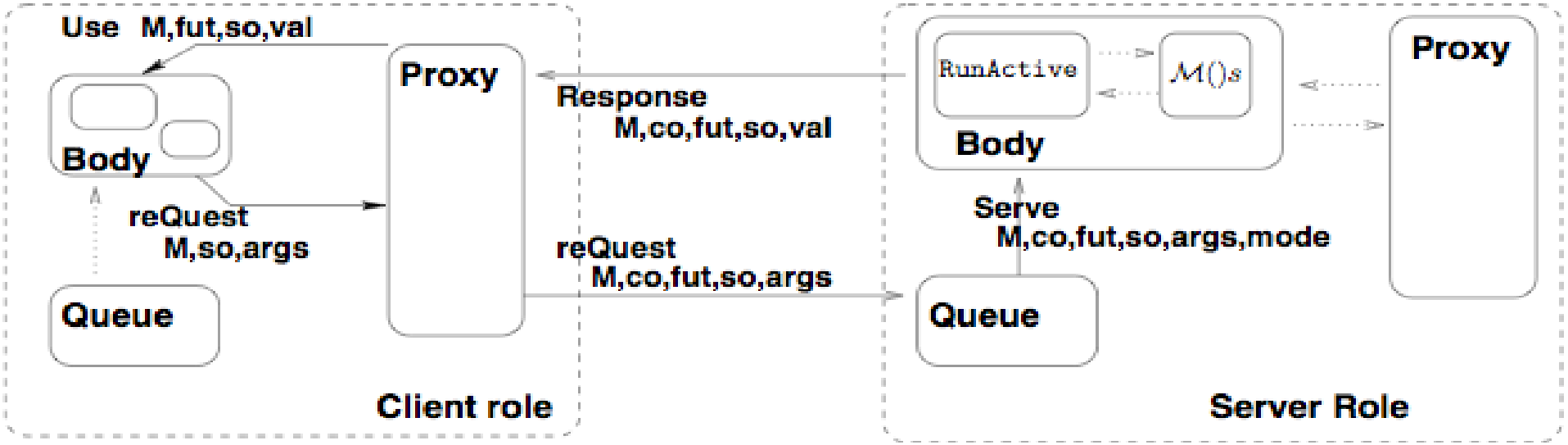
Data domains

- Definition of data domains inside the specification
  - Required for the verification and the code generation
- Finite model left as a second abstraction
  - Specific to the formula to prove

Proposition

- Start from a simplified Java
  - Simple data types
- Augment with component primitives
  - “component”, “interface”
  - “bind”, “unbind”, “start”, “stop”, ...
- Ideally, the user should only care about the business code
  - Implicit synchronisations
  - Transparent futures

A ProActive view



What's inside a primitive?

- Implicit
  - Infinite queue, proxies
- Explicit (given by the user)
  - Activity (Serving Policy)
  - Methods (defined by the server interfaces)
    - Local variables
    - Access to client interfaces
  - Persistent variables
    - Visible to all methods and to the activity

Restrictions

- A method may only call client interfaces or local methods (no service method calls)
- A method cannot manipulate the queue

What's inside a composite?

- Implicit
  - Infinite queue
- Explicit (given by the user)
  - Activity (Serving Policy)
  - Direct Subcomponents
    - External interfaces
  - Deploy policy
    - Bindings
    - Life-cycle management

Breaking the rules...

- Sometimes we don't want “transparency”
  - Wait for a value
    - future.waitForValue();
  - Check if value has arrived
    - future.isAwaited();
  - Collective interfaces
    - listOfValues.waitForAny();
    - listOfValues.waitForN();

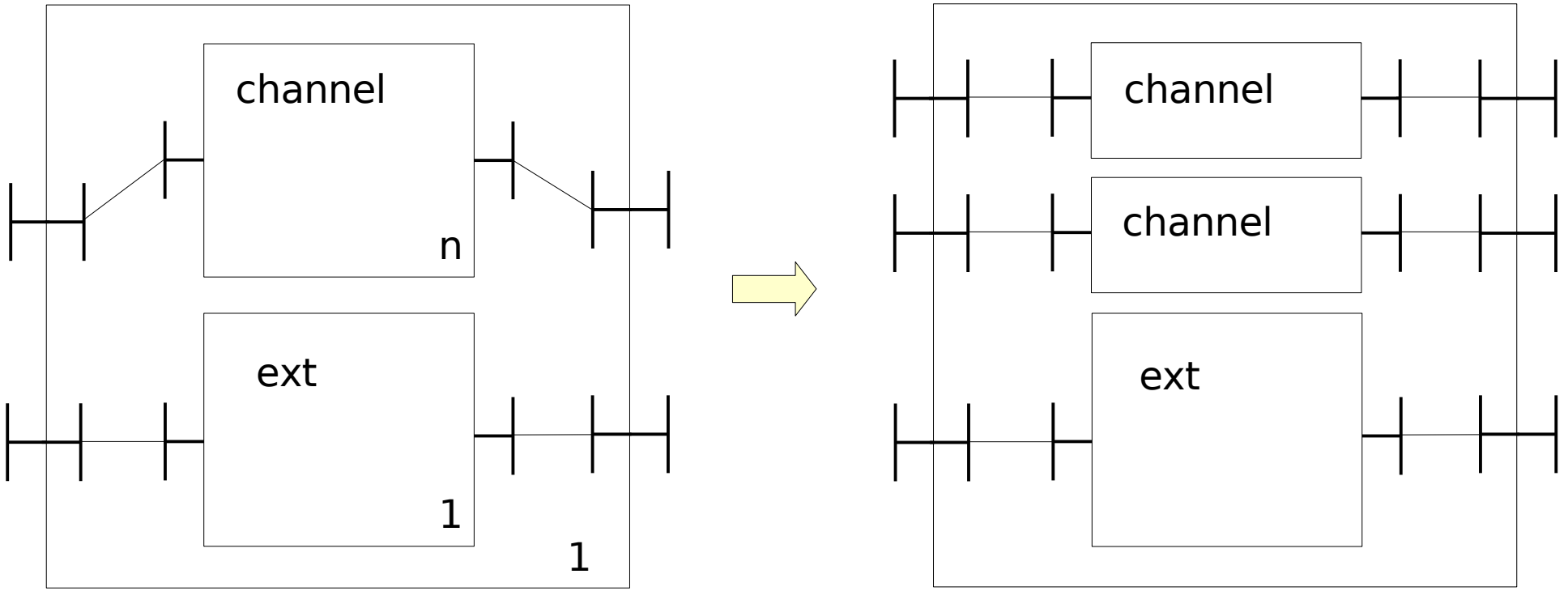
Communication policies

- All calls are asynchronous unless stated
  - Explicit in method signature

```
interface AnInterface {
      sync void foo();  // mandatory synchronous call
      A bar();
}

// explicitly force synchronous call
A a = interface.bar();
a.waitForValue();
```

Multiple Components



Collective Communication

- Collective interfaces

```
interface printerCtrl[numberOfClients];
bind(channel[i].printerCtrlI, this.printerCtrl[i]);
```

Multiple component

Collective interface

- Multicast interfaces

```
interface serversItf[numberOfServers];
serversItf.flush();
```

Exceptions

- try/catch annotators
  - Exception may not leave block
    - No automatic continuation for futures carrying exceptions
    - Implicit barrier before leaving block

```
try {
    a = interface1.f();     // interface1.f() throws 'Exception'
    b = interface1.g();
    c = interface2.h(a);    // waits for 'a' & throws 'Exception'
    interface3.i(b);        // asynchronous
} catch (Exception e) {     // barrier for exceptions
    ...
}
// do something else AFTER the barrier is released
```

Indeterministic choice

- Hide computation

```
bool valid = clientItf.m();
Domain is enum {“A”, “B”};
__INTERNAL(valid);
condition = __ANY(Domain);
switch (condition) {
   case A:
      // do something
      break;
   case B:
      // do something
      break;
}
```

Block

nism

What else to specify?

- Specialised Membrane behaviour
  - User defined controllers
  - Non-Functional components

Example

```
behavior component CashDeskApplication is Primitive
{

    // primitive attributes
    CashState cashState;
    ...

    // set default policy as FIFO
    ServingPolicy = FIFO;

    // initialize the component
    init() {
        cashState = IDLE;
    }
```

Example

```
server interface cashDeskControlIf {
  void productBarcodeScannedEvent(int barcode) {
    switch (cashState) {
      case IDLE:
        // ignore signal
        break;
      case STARTED:
        Product product =
            cashDeskConn.getProductWithStockItem(barcode);
        __INTERNAL(product);
        cashDeskConn.runningTotalChanged(product);
        break;
    }
  }

  void paymentModeEvent(PaymentMode mode) {
    paymentMode = mode;
  }
```

Summary

- Specification
  - Single language for both structural and behavioural aspects
  - Sound with component problematic
  - Can be verified (most cases)
    - Effects of sync/async calls can be checked

Summary

- Compared with pNets:
  - Higher-level, without ambiguity in what a label/sentence stands for
  - Specific to component-based software
  - (Hopefully) quicker learning curve

Summary

- Generated code:
  - Control code is correct-by-construction
    - Data-flow synchronisation!
  - Avoids writing cumbersome code
    - In particular when dealing with exceptions
  - Suitable for simulation
    - Executable

Additional thoughts

- Keep the specification close to the implementation
  - Attached to the ADL
  - Annotations / comments in Java code