An Overview of CoCoME

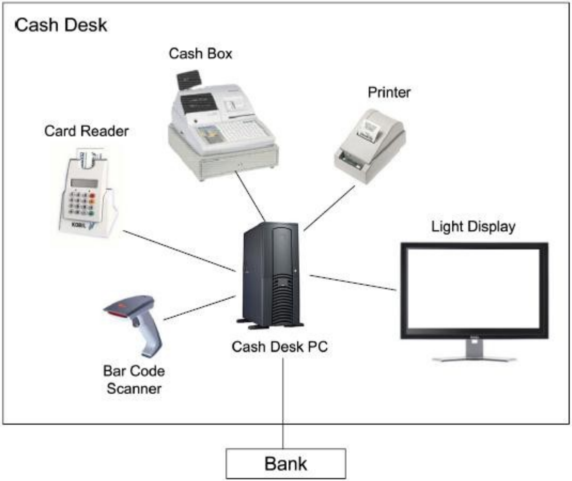
Hakim Hannousse

ECOLE DES MINES DE NANTES
DEPARTMENT OF COMPUTER SCIENCE

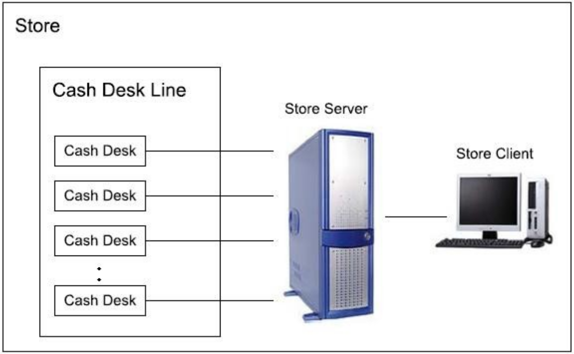
CoCoME: Common Component Modeling Example

- Context = Trading System
- Supports functional aspects : Manage sales, order products, .. etc.
- Supports non function aspects : Manage Express Checkout, Synchronization, RealTime Constraints ... etc.
- Extra-functional properties based on statistics for typical German super-markets

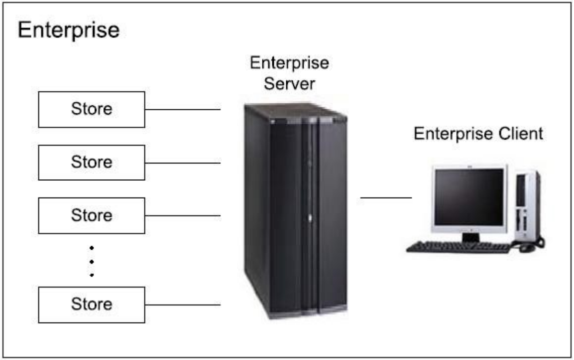
CoCoME: A Single Cash Desk



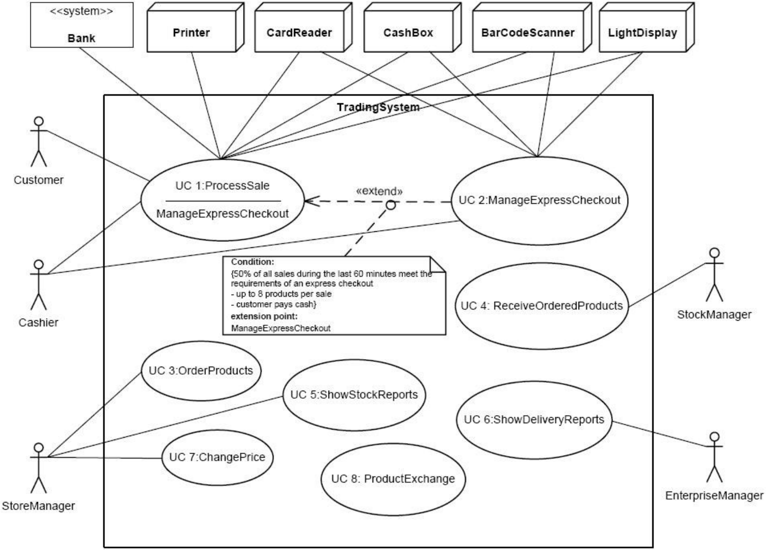
CoCoME: A single Store



CoCoME: An Entreprise



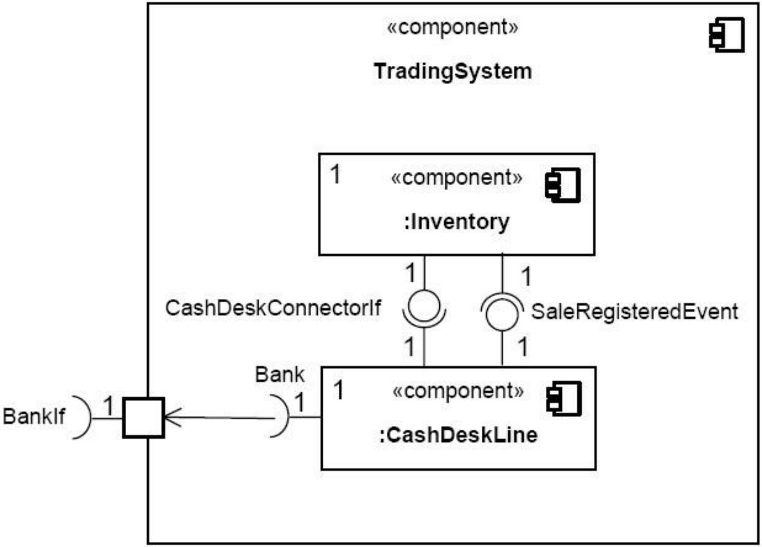
CoCoME: Provided Use Cases



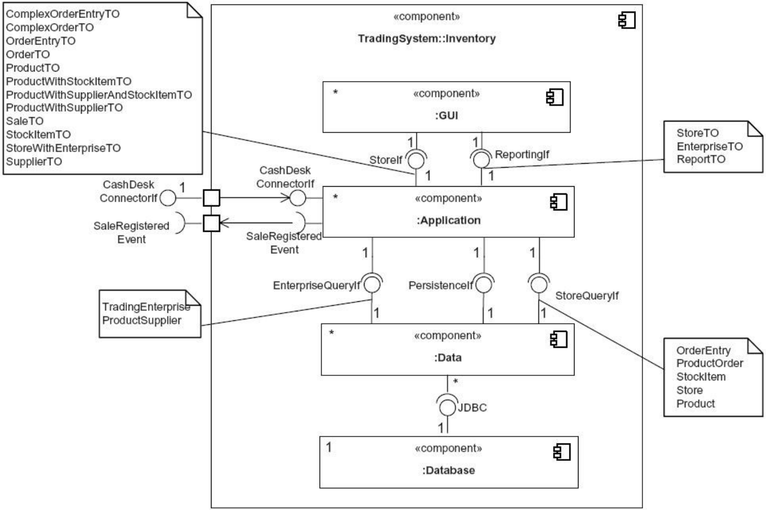
Condition:
{50% of all sales during the last 60 minutes meet the requirements of an express checkout
- up to 8 products per sale
- customer pays cash}

extension point:
ManageExpressCheckout

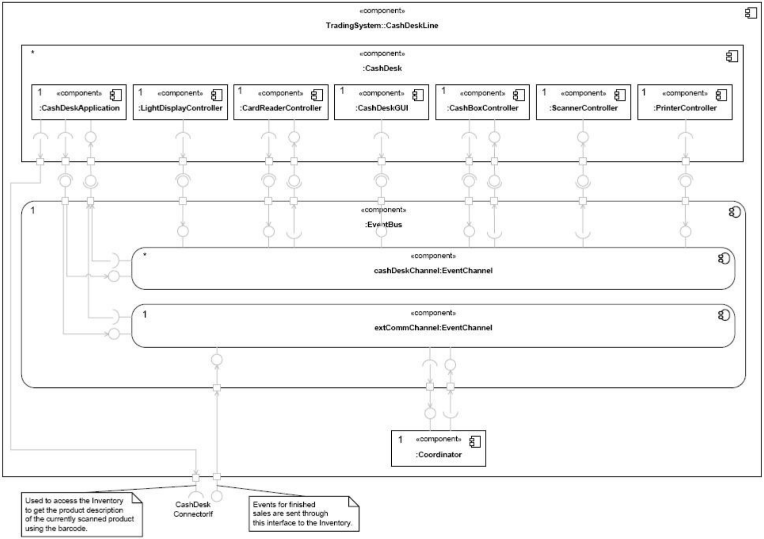
CoCoME: Component Modularization - TradingSystem



CoCoME: Component Modularization - Inventory



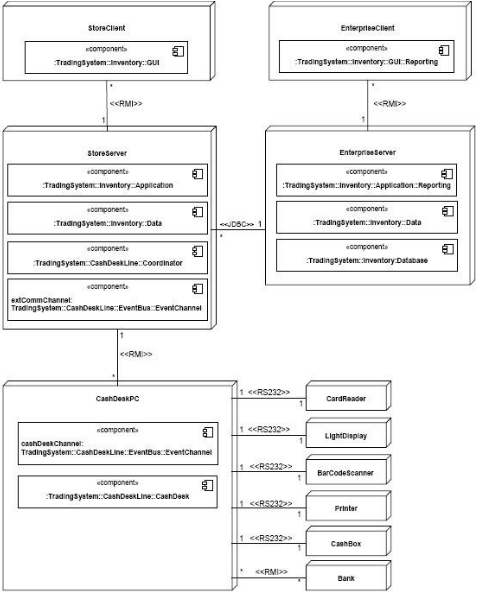
CoCoME: Component Modularization - CashDeskLine



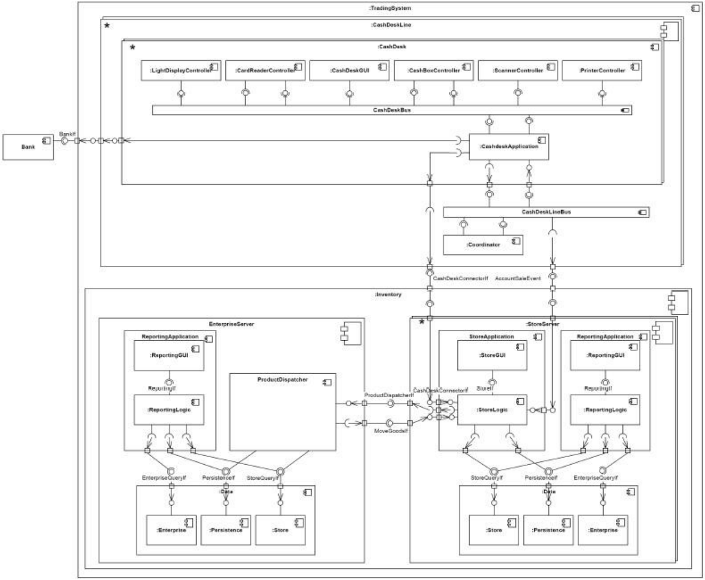
Used to access the Inventory to get the product description of the currently scanned product using the barcode.

Events for finished sales are sent through this interface to the Inventory.

CoCoME: Component Modularization - Deployment System



CoCoME in Fractal : Structural View



CoCoME in Fractal : Behavioral View

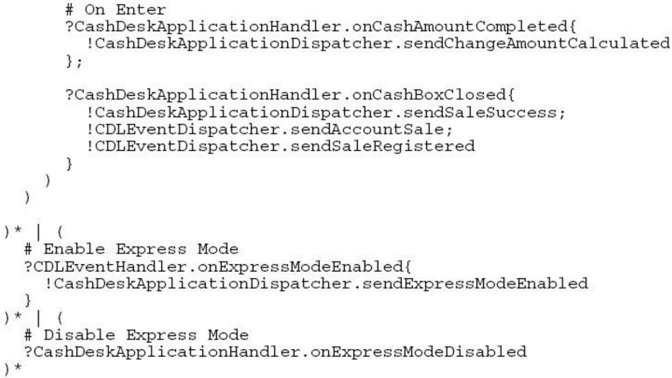
1. A frame protocol is associated to each component
2. Frame protocol language used by FractalBPC :

$$
\begin{array}{lcl}
P & ::= & !I.M\uparrow \\
 & | & ?I.M\downarrow \\
 & | & !I.M\downarrow \\
 & | & ?I.M\uparrow \\
 & | & ?I.M\{P\} \\
 & | & !I.M\{P\} \\
 & | & P+ \\
 & | & P* \\
 & | & P_1|P_2 \\
 & | & P_1;P_2
\end{array}
$$

CoCoME in Fractal : Behavioral View

```
# INITIALISED
(
  ?CashDeskApplicationHandler.onSaleStarted
);

# SALE_STARTED
(
  ?CashDeskApplicationHandler.onProductBarcodeScanned{
      !CashDeskConnector.getProductWithStockItem;
      !CashDeskApplicationDispatcher.sendProductBarcodeNotValid+
      !CashDeskApplicationDispatcher.sendRunningTotalChanged
  }
)*; # <--- LOOP

?CashDeskApplicationHandler.onSaleFinished;

# SALE_FINISHED
(
  ?CashDeskApplicationHandler.onPaymentMode
);

# PAYING_BY_CASH
(
  (
    (
     ?CashDeskApplicationHandler.onCashAmountEntered
    )*;
```

CoCoME in Fractal : Behavioral View



```
      # On Enter
      ?CashDeskApplicationHandler.onCashAmountCompleted{
        !CashDeskApplicationDispatcher.sendChangeAmountCalculated
      };

      ?CashDeskApplicationHandler.onCashBoxClosed{
        !CashDeskApplicationDispatcher.sendSaleSuccess;
        !CDLEventDispatcher.sendAccountSale;
        !CDLEventDispatcher.sendSaleRegistered
      }
    )
  )

)* | (
  # Enable Express Mode
  ?CDLEventHandler.onExpressModeEnabled{
    !CashDeskApplicationDispatcher.sendExpressModeEnabled
  }
)* | (
  # Disable Express Mode
  ?CashDeskApplicationHandler.onExpressModeDisabled
)*
```

CoCoME in Fractal : Deployment View

1. FractalRMI are used rather than Sun RMI
2. JMS are not used for implementing buses, they are replaced by components routing messages
3. Deployment is described using FractalADL and implemented using FractalRMI

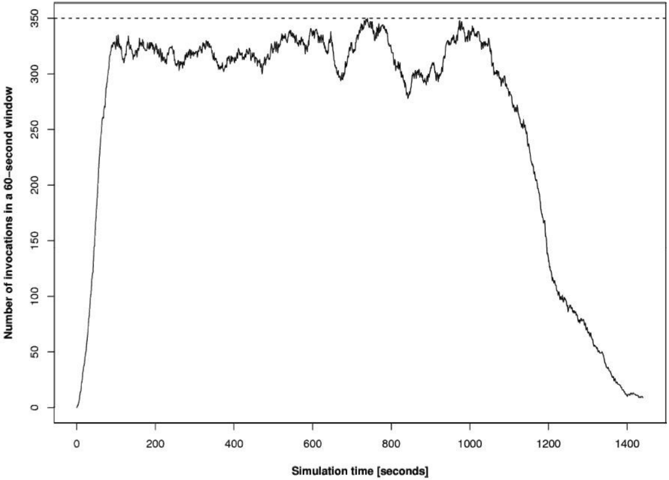
CoCoME in Fractal : Implementation View

1. The architecture is modeled using FractalGUI
2. The resulting model is extended by hand to integrate behavior protocols
3. A tool is used to get the skeleton of the appliaction
4. The code of the CoCoME implementation is adapted and insered to the corresponding components

CoCoME in Fractal : Testing process

- FracalBPC is used to check components communicates behaviors
  1. The GUI components are not considered for testing
  2. Extra-functional proprieties are independently tested from the functionality testing
  3. The trading system is automatically lunched.

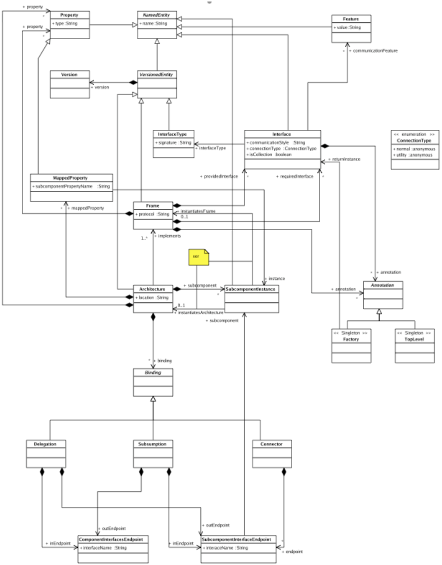
CoCoME in Fractal : Testing Results



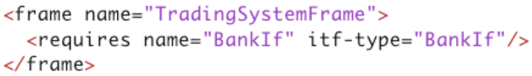
SOFA is a Hierarchical Component Model

- Each Component is defined by:
  - Frame : provided and required interfaces
  - Architecture : subcomponents and their interconnections
- A Component has two parts :
  - Control part
  - Content part
- Components are bound using connectors

SOFA is defined by means of a Meta-Model



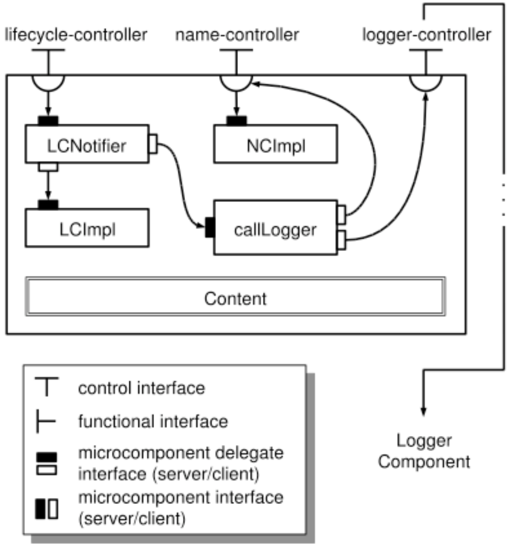
SOFA is ADL-Based design



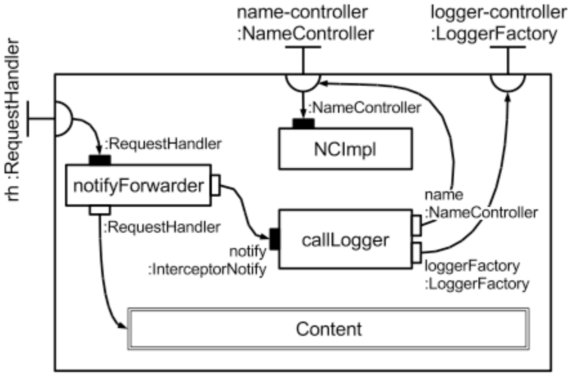
```
<frame name="TradingSystemFrame">
  <requires name="BankIf" itf-type="BankIf"/>
</frame>
```

```
<architecture name="TradingSystemArch" frame="TradingSystemFrame">
  <sub-comp name="CashDeskLine" frame="CashDeskLineFrame"
arch="CashDeskLineArch"/>
  <sub-comp name="Inventory" frame="InventoryFrame" arch="InventoryArch"
>
  <connection>
    <endpoint itf="AccountSaleEventHandlerIf" sub-comp="CashDeskLine"/>
    <endpoint itf="AccountSaleEventHandlerIf" sub-comp="Inventory"/>
  </connection>
  <connection>
    <endpoint itf="CashDeskConnectorIf" sub-comp="CashDeskLine"/>
    <endpoint itf="CashDeskConnectorIf" sub-comp="Inventory"/>
  </connection>
  <connection>
    <endpoint itf="BankIf" sub-comp="CashDeskLine"/>
    <endpoint itf="BankIf"/>
  </connection>
</architecture>
```

SOFA 2.0 is A Microcomponent-Based Component Controller Model (1)



SOFA 2.0 is A Microcomponent-Based Component Controller Model (2)



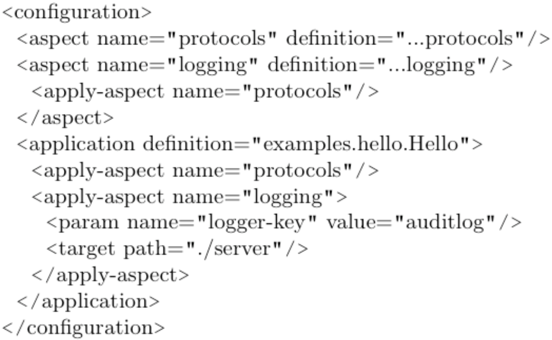
SOFA 2.0 Supports Controllers Extensions Using Aspects

```
<aspect-definition name="logging" >
 <frame-addon-definition name="logger-itfs" >
  <interface signature="LoggerFactory"
    role="client" name="logger-controller"/>
 </frame-addon-definition>
 <component name="logger"
   definition="logger-adl" />

 <microcomponent-definition
   name="callLogger" >
  <interface signature="InterceptorNotify"
    role="server" name="notify" />
  <interface signature="LoggerFactory"
    role="client" name="loggerFactory" />
  <interface signature="NameController"
    role="client" name="name" />
  <content class="LoggerInterceptor" />
 </microcomponent-definition>

 <microcomponent-definition
   name="notifyForwarder" >
  <interface signature="InterceptorNotify"
    role="client" name="notify" />
  <dynamic-interface role="delegateserver" />
  <dynamic-interface role="delegateclient" />
  <content
    generator="InterceptorNotifyGenerator"/>
 </microcomponent-definition>
```

SOFA 2.0 Supports Controllers Extensions Using Aspects

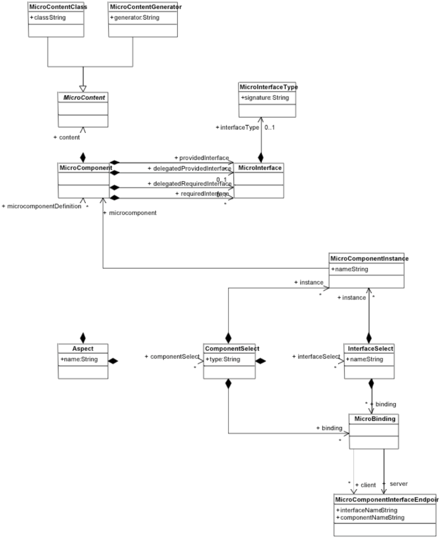
```
 <select-component type="any" >

   <frame-addon definition="logger-itfs" />
   <component-binding client="this.logger-controller"
       server="logger.logFactory" />

   <select-interface name="*" type="functional">

      <microcomponent name="logFwd"
         definition="notifyForwarder"
         flow="passthrough" />
      <microcomponent name="logCalls"
         definition="callLogger" flow="standalone"/>

      <binding client="logCalls.loggerFactory"
         server="this.logger-controller" />
      <binding client="logCalls.name"
         server="this.name-controller" />
      <binding client="logFwd.notify"
         server="logCalls.notify" />

   </select-interface>
 </select-component>
</aspect-definition>
```

SOFA 2.0 Supports Controllers Extensions Using Aspects

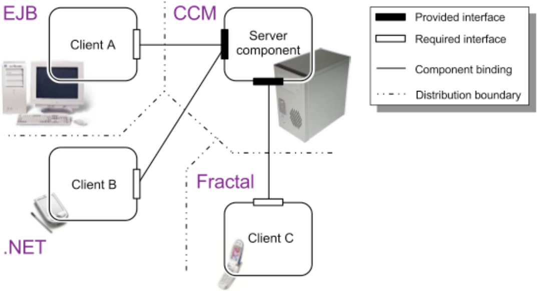


```
<configuration>
  <aspect name="protocols" definition="...protocols"/>
  <aspect name="logging" definition="...logging"/>
    <apply-aspect name="protocols"/>
  </aspect>
  <application definition="examples.hello.Hello">
    <apply-aspect name="protocols"/>
    <apply-aspect name="logging">
      <param name="logger-key" value="auditlog"/>
      <target path="./server"/>
    </apply-aspect>
  </application>
</configuration>
```

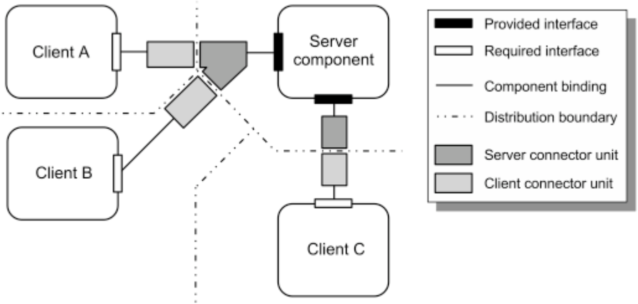
SOFA 2.0 - Microcomponent Meta-Model



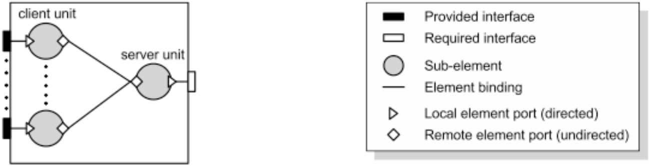
SOFA 2.0 Supports Heterogeneous Deployment via First-Entity Class Connectors (1)



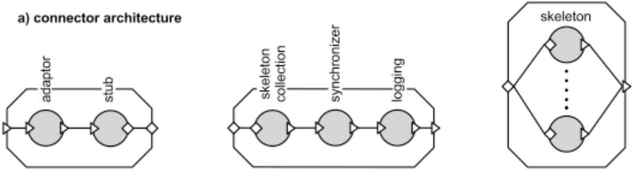
SOFA 2.0 Supports Heterogeneous Deployment via First-Entity Class Connectors (2)



SOFA 2.0 - Connector Architecture



a) connector architecture



b) client unit element architecture

c) server unit element architecture

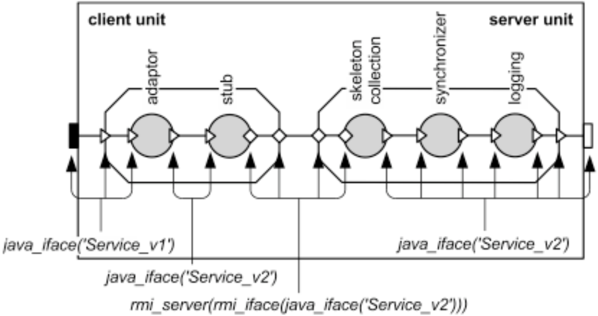
d) skeleton collection element architecture

SOFA 2.0 - Formal Signature of ports on an element

Port signatures:

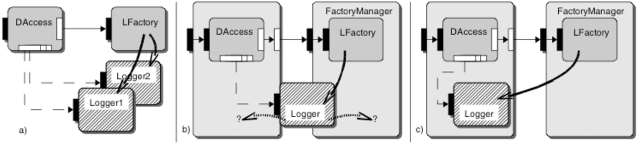
call: *Itf*
line: *rmi_server(rmi_iface(Itf))*

SOFA 2.0 - Interface adaptation and propagation through a connector

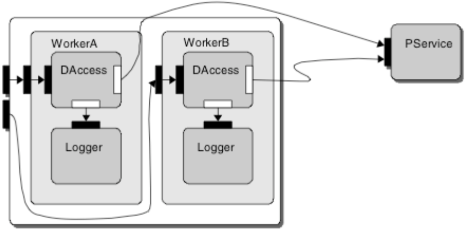


SOFA 2.0 - Supports Dynamic Reconfiguration Only w.r.t Configuration Patterns

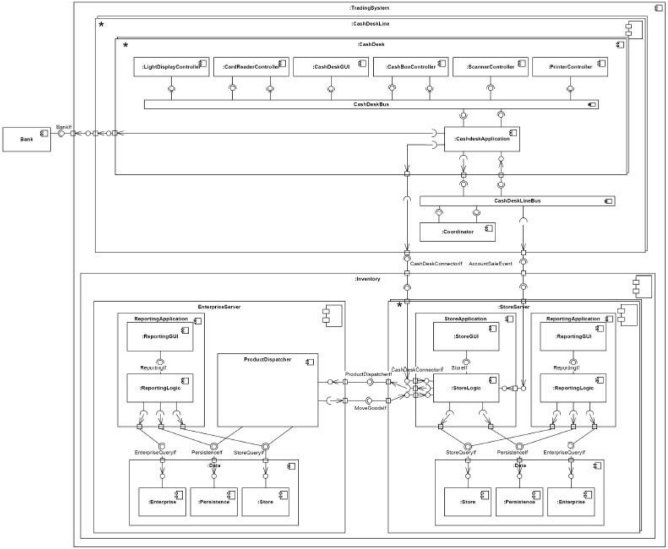
1. Nested Factory Pattern



2. Removing Component Pattern
3. Utility Interface Pattern



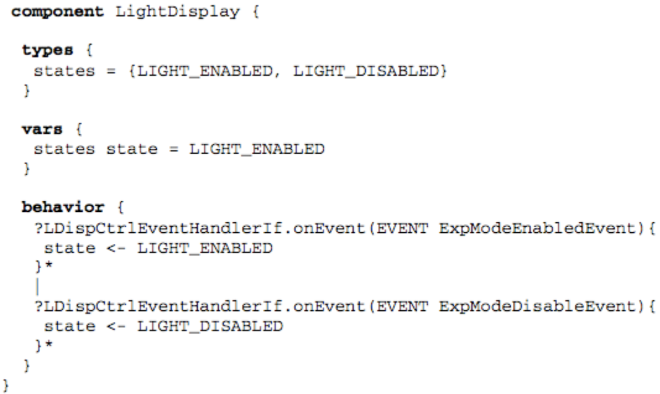
SOFA 2.0 - Structural View



SOFA 2.0 - Behavioral View

SOFA 2.0 uses EBP to describe component's behaviar :

- Example :
  $(?i.open; (?i.read+?i.write)*; ?i.close)|?ctrl.status*$
- EBP supports types and local variable declarations
- EBP supports switch and while statements



```
component LightDisplay {

 types {
  states = {LIGHT_ENABLED, LIGHT_DISABLED}
 }

 vars {
  states state = LIGHT_ENABLED
 }

 behavior {
  ?LDispCtrlEventHandlerIf.onEvent(EVENT ExpModeEnabledEvent){
   state <- LIGHT_ENABLED
  }*
  |
  ?LDispCtrlEventHandlerIf.onEvent(EVENT ExpModeDisableEvent){
   state <- LIGHT_DISABLED
  }*
 }
}
```

SOFA 2.0 - Deployment View

- SOFAnode distributed runtime environment is used for the deployment issue.
- SOFAnode = Repository + deployment docks
- Connectors encapsulate middleware and support different communication style.
- SOFA Application lifecycle:
    1. Defining primitive components or frame components by the developer
    2. Uploads them in the repository
    3. Assembly process to construct component architectures
    4. A deployer assigns components to docks, sets components properties values and the control aspects to be applied in the applications in the deplyment plan
    5. Connectors are generated automatically
    6. Launch the appliaction

SOFA 2.0 - Verification and Analysis

- Compliance both vertical and Horizontal via Promela. (EBP2PR)

Table 1. The result of vertical compliance verification of the CashDesk component

# of states	EBP2PR [s]	Verification [s]	Total time [s]
3 335 950	41.5	46.1	95,6

- Verification of Code against Frame Protocols via JPF tool.
- Runtime Checking against Code
- Performance Analysis

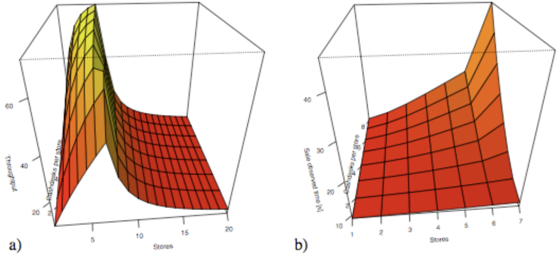


Fig. 4. Calculated a) throughput and b) average service time of Use Case 1

rCOS Component Model : Component Definition

- A Component is defined by its interface and its contract :
  - Interface : $I =< FDecl, MDecl >$
  - Contract : $Ctr =< I, Init, MSpec, Prot >$
- Example : Buffer Component

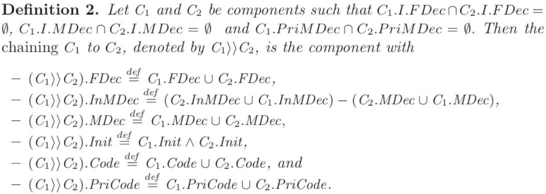
  $I_{Buffer} =< buff : seq(Int), \{put(inx : int), get(outy : int)\} >$
  $Ctr_{Buffer} =< I_{Buffer}, Init_{Buffer}, MSpec_{Buffer}, Prot_{Buffer} >$
  $Init_{Buffer} = |buff| = 0,$
  $MSpec_{Buffer}(put(inx : int)) = (\vdash buff' =< x > \bullet buff),$
  $MSpec_{Buffer}(get(outy : int)) = (\vdash buff' = tail(buff) \wedge y' = head(buff)),$

  $Prot_{Buffer} = (put, get) * +(put; (get; put)*)$
- A Component in rCOS is a tuple:

  $C =< I, Init, MCode, PriMDec, PriMCode, InMDec >$

rCOS Component Model : Component Composition Operators

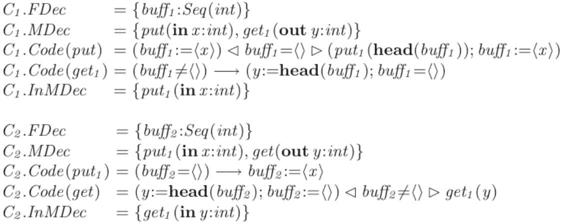
- Chaining operator :



Definition 2. *Let* C_1 *and* C_2 *be components such that* $C_1.I.FDec \cap C_2.I.FDec = \emptyset$, $C_1.I.MDec \cap C_2.I.MDec = \emptyset$ *and* $C_1.PriMDec \cap C_2.PriMDec = \emptyset$. *Then the chaining* C_1 *to* C_2, *denoted by* $C_1\rangle\rangle C_2$, *is the component with*

- $(C_1\rangle\rangle C_2).FDec \overset{def}{=} C_1.FDec \cup C_2.FDec,$
- $(C_1\rangle\rangle C_2).InMDec \overset{def}{=} (C_2.InMDec \cup C_1.InMDec) - (C_2.MDec \cup C_1.MDec),$
- $(C_1\rangle\rangle C_2).MDec \overset{def}{=} C_1.MDec \cup C_2.MDec,$
- $(C_1\rangle\rangle C_2).Init \overset{def}{=} C_1.Init \wedge C_2.Init,$
- $(C_1\rangle\rangle C_2).Code \overset{def}{=} C_1.Code \cup C_2.Code,$ *and*
- $(C_1\rangle\rangle C_2).PriCode \overset{def}{=} C_1.PriCode \cup C_2.PriCode.$

Example :

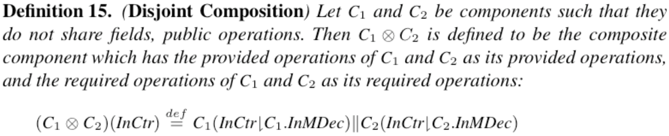


$$\begin{aligned}
C_1.FDec &= \{buff_1{:}Seq(int)\}\\
C_1.MDec &= \{put(\mathbf{in}\ x{:}int), get_1(\mathbf{out}\ y{:}int)\}\\
C_1.Code(put) &= (buff_1{:=}\langle x\rangle) \lhd buff_1{=}\langle\rangle \rhd (put_1(\mathbf{head}(buff_1)); buff_1{:=}\langle x\rangle)\\
C_1.Code(get_1) &= (buff_1{\neq}\langle\rangle) \longrightarrow (y{:=}\mathbf{head}(buff_1); buff_1{=}\langle\rangle)\\
C_1.InMDec &= \{put_1(\mathbf{in}\ x{:}int)\}
\end{aligned}$$

$$\begin{aligned}
C_2.FDec &= \{buff_2{:}Seq(int)\}\\
C_2.MDec &= \{put_1(\mathbf{in}\ x{:}int), get(\mathbf{out}\ y{:}int)\}\\
C_2.Code(put_1) &= (buff_2{=}\langle\rangle) \longrightarrow buff_2{:=}\langle x\rangle\\
C_2.Code(get) &= (y{:=}\mathbf{head}(buff_2); buff_2{:=}\langle\rangle) \lhd buff_2{\neq}\langle\rangle \rhd get_1(y)\\
C_2.InMDec &= \{get_1(\mathbf{in}\ y{:}int)\}
\end{aligned}$$

rCOS Component Model : Component Composition Operators

- Disjoint Composition :

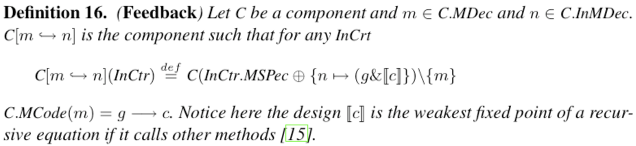


Definition 15. *(**Disjoint Composition**) Let C_1 and C_2 be components such that they do not share fields, public operations. Then $C_1 \otimes C_2$ is defined to be the composite component which has the provided operations of C_1 and C_2 as its provided operations, and the required operations of C_1 and C_2 as its required operations:*

$$(C_1 \otimes C_2)(InCtr) \overset{def}{=} C_1(InCtr\lfloor C_1.InMDec)\|C_2(InCtr\lfloor C_2.InMDec)$$

rCOS Component Model : Component Composition Operators

- Feedback :



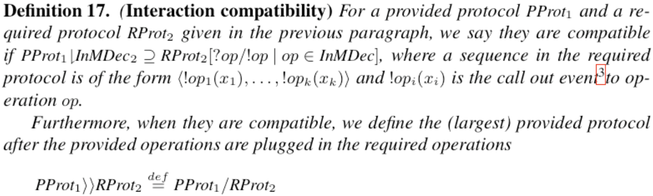
Definition 16. *(**Feedback**) Let C be a component and* $m \in C.MDec$ *and* $n \in C.InMDec$. $C[m \hookrightarrow n]$ *is the component such that for any InCrt*

$$C[m \hookrightarrow n](InCtr) \overset{def}{=} C(InCtr.MSPec \oplus \{n \mapsto (g\&[\![c]\!])\})\backslash\{m\}$$

$C.MCode(m) = g \longrightarrow c$. *Notice here the design* $[\![c]\!]$ *is the weakest fixed point of a recursive equation if it calls other methods [15].*

rCOS Component Model : Component Composition

- Component Interaction Compatibility :



Definition 17. *(***Interaction compatibility***)* *For a provided protocol $PProt_1$ and a required protocol $RProt_2$ given in the previous paragraph, we say they are compatible if $PProt_1 \lfloor InMDec_2 \supseteq RProt_2[?op/!op \mid op \in InMDec]$, where a sequence in the required protocol is of the form $\langle !op_1(x_1), \ldots, !op_k(x_k) \rangle$ and $!op_i(x_i)$ is the call out event[3] to operation op.*

Furthermore, when they are compatible, we define the (largest) provided protocol after the provided operations are plugged in the required operations

$$PProt_1\rangle\rangle RProt_2 \overset{def}{=} PProt_1/RProt_2$$