An Overview of CoCoME

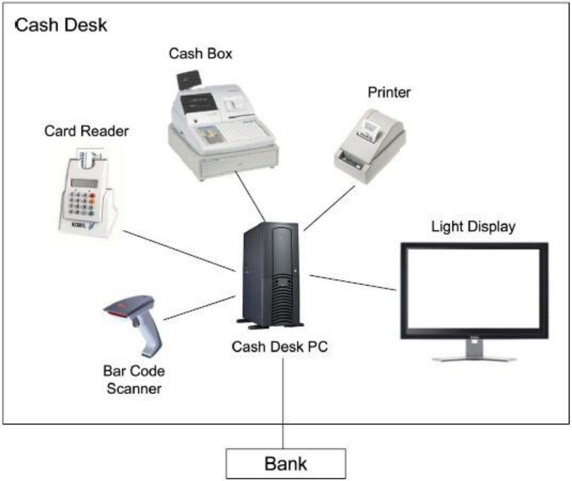
Hakim Hannousse

ECOLE DES MINES DE NANTES
DEPARTMENT OF COMPUTER SCIENCE

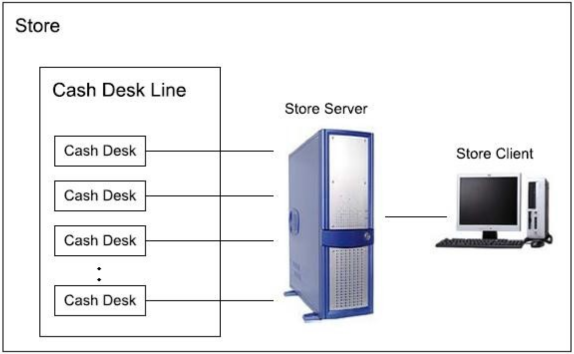
CoCoME: Common Component Modeling Example

- 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

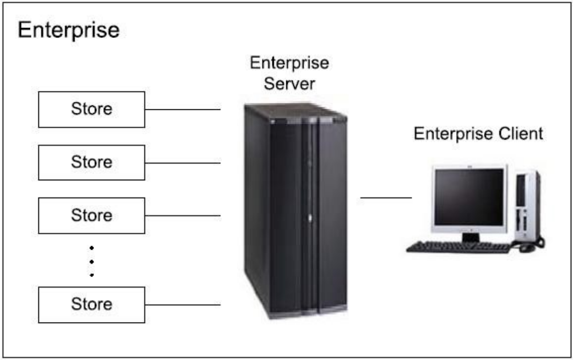
CoCoME: A Single Cash Desk



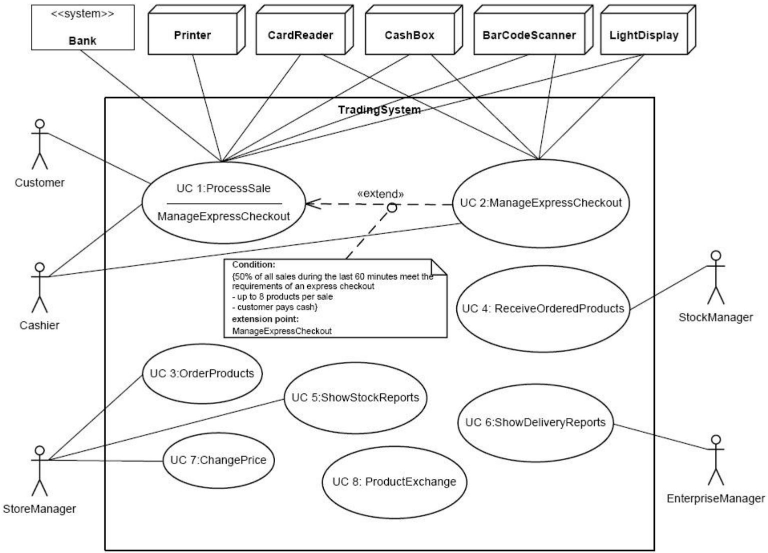
CoCoME: A single Store



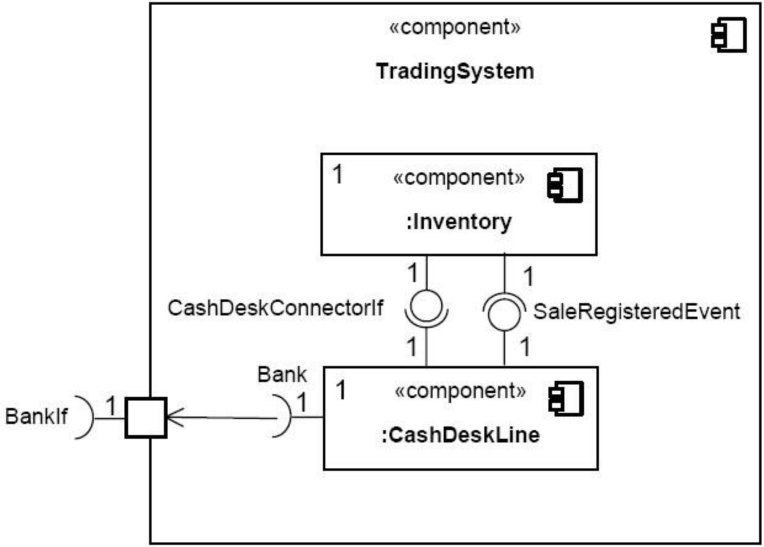
CoCoME: An Entreprise



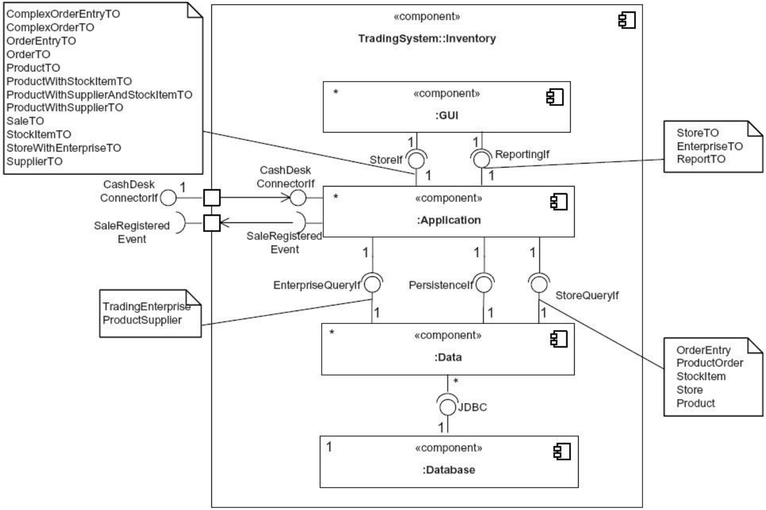
CoCoME: Provided Use Cases



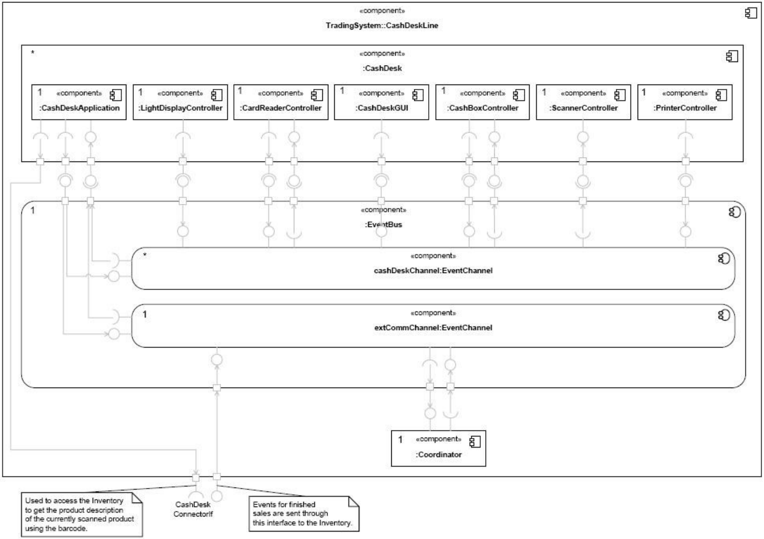
CoCoME: Component Modularization - TradingSystem



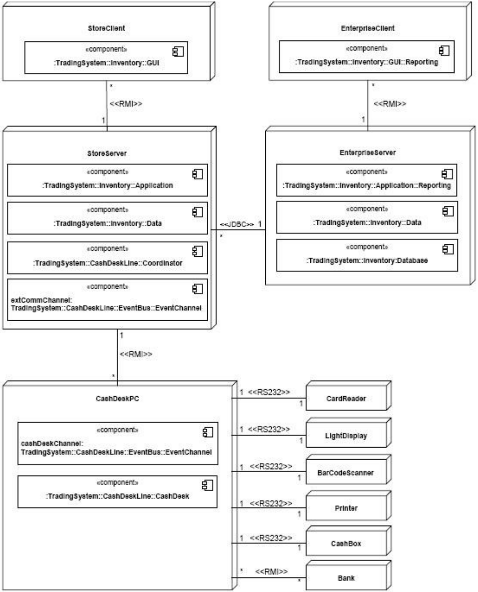
CoCoME: Component Modularization - Inventory



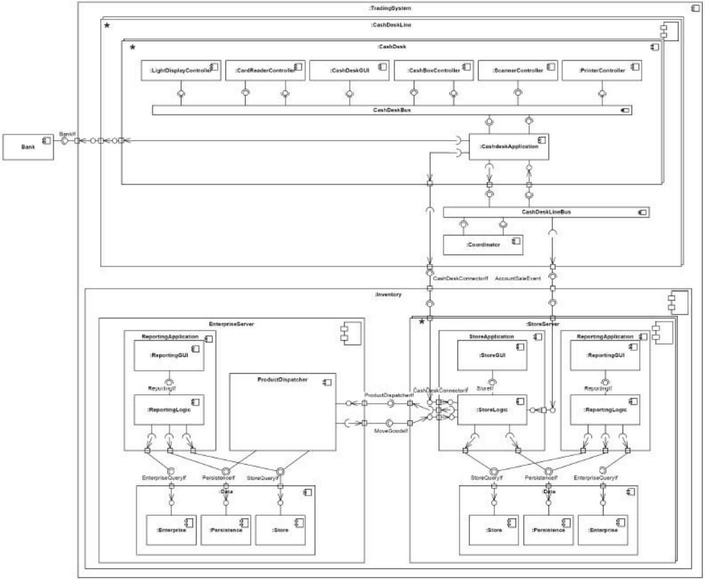
CoCoME: Component Modularization - CashDeskLine



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

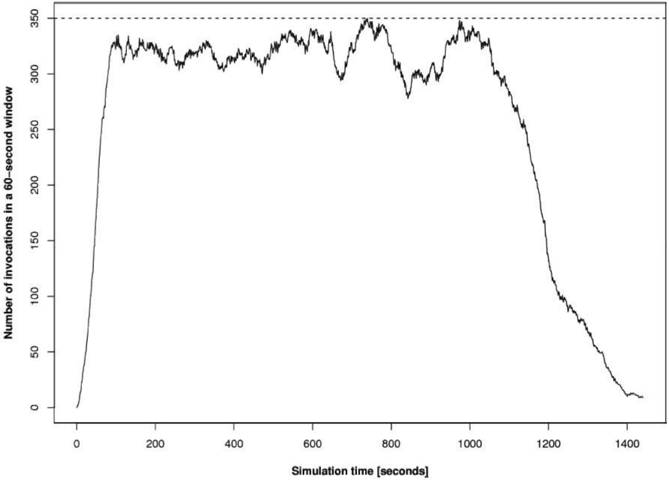
CoCoME in Fractal : Implementation View

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.

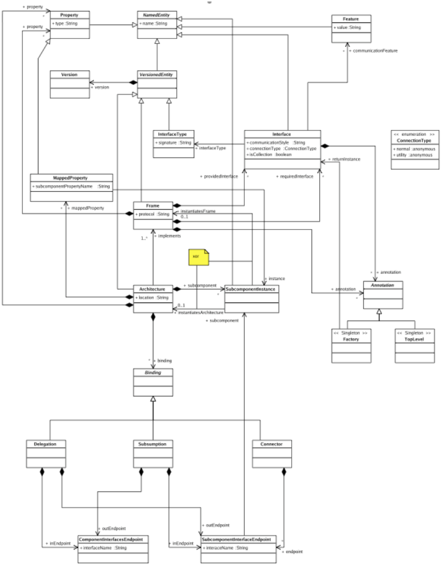
CoCoME in Fractal : Testing Results



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

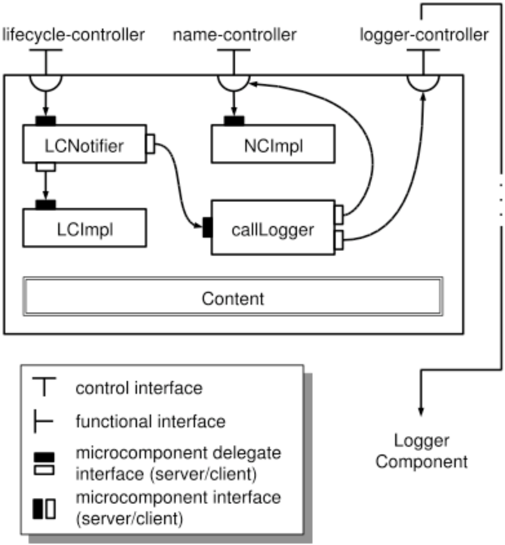


SOFA is ADL-Based design

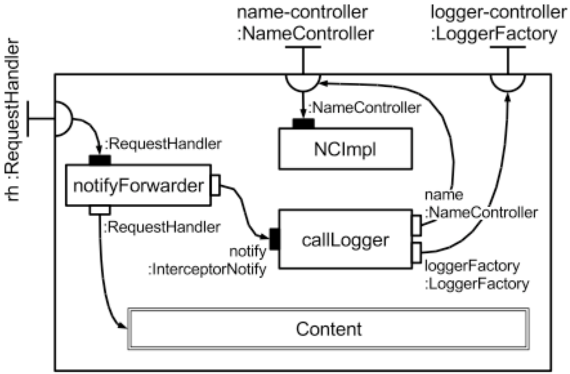
```
<frame name="TradingSystemFrame">
  <requires name="BankIf" itf-type="BankIf"/>
</frame>
```

```
<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>
```

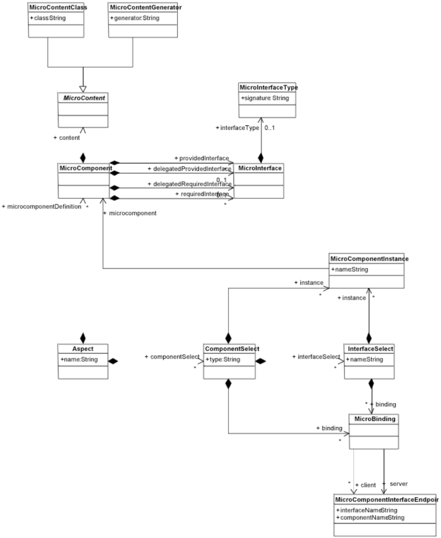
SOFA 2.0 Supports Controllers Extensions Using Aspects

```
<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>
```

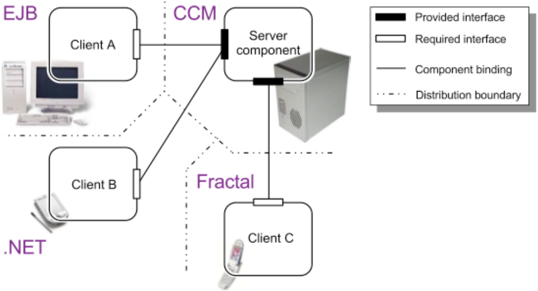
SOFA 2.0 Supports Controllers Extensions Using Aspects

```
<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>
```

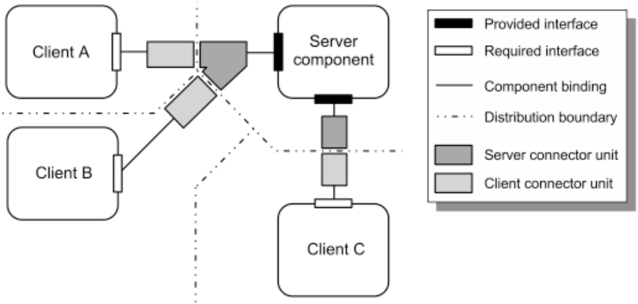
SOFA 2.0 - Microcomponent Meta-Model



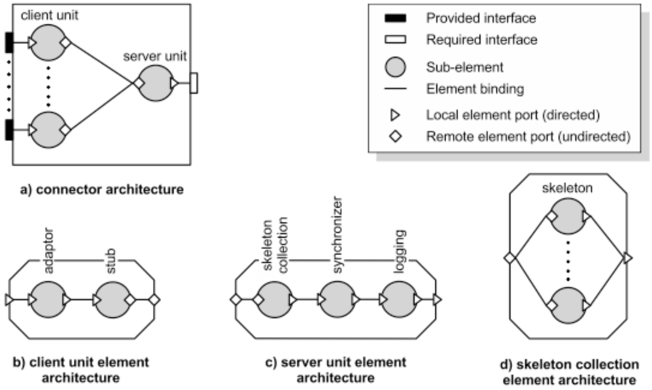
SOFA 2.0 Supports Heterogeneous Deployment via First-Entity Class Connectors (1)



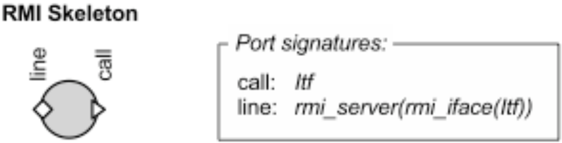
SOFA 2.0 Supports Heterogeneous Deployment via First-Entity Class Connectors (2)



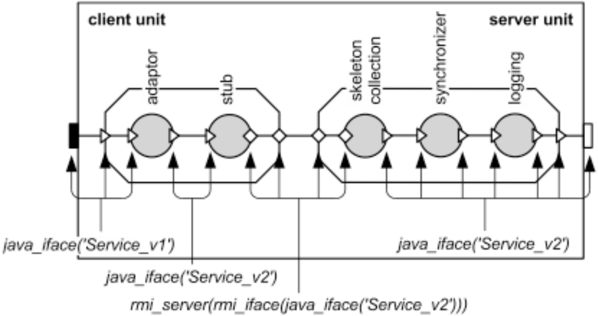
SOFA 2.0 - Connector Architecture



SOFA 2.0 - Formal Signature of ports on an element

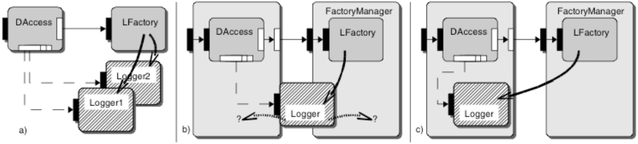


SOFA 2.0 - Interface adaptation and propagation through a connector

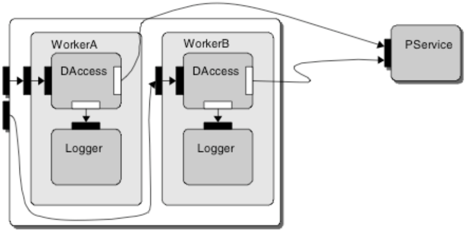


SOFA 2.0 - Supports Dynamic Reconfiguration Only w.r.t Configuration Patterns

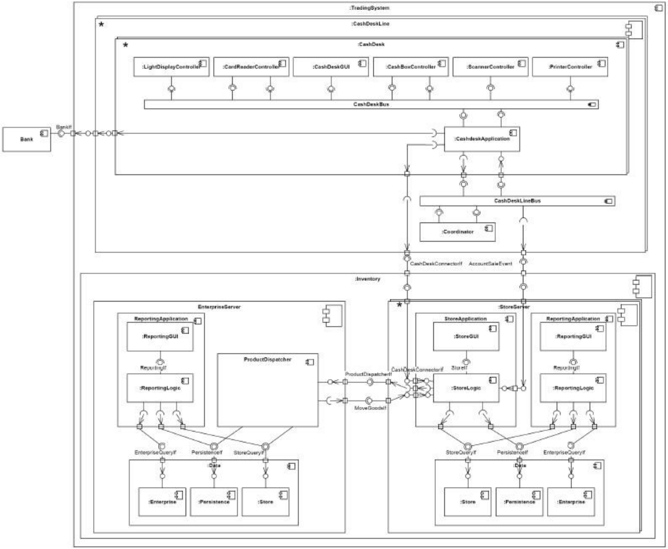
1. Nested Factory Pattern



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

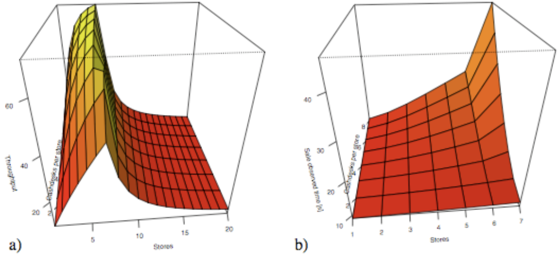


Fig. 4. Calculated a) throughput and b) average service time of Use Case 1

rCOS : Refinement Calculus for Object oriented Systems

- rCOS originally is designed to support only object oriented systems
- rCOS syntax is similar to that of Java
- rCOS Semantics is based on Hoare's theory
- rCOS main feature is the object-oriented refinement

rCOS Syntax

$$
\begin{array}{lllll}
\textbf{class} & C\ [\textbf{extends}\ D]\ \{ & & & \\
\textbf{attributes} & T\,x = d, \ldots, T_k\ x = d & & & \\
\textbf{methods} & m(T\ \mathit{in};\ V\ \mathit{return})\ \{ & & & \\
 & \quad \text{pre:} & & c \vee \ldots \vee c & \\
 & \quad \text{post:} & \wedge & (R; \ldots; R) \vee \ldots \vee (R; \ldots; R) & \\
 & & \wedge & \ldots\ldots & \\
 & & \wedge & (R; \ldots; R) \vee \ldots \vee (R; \ldots; R)\ \} & \\
 & \ldots\ldots & & & \\
\textbf{invariant} & \mathit{Inv} & & & \\
 & \} & & &
\end{array}
$$

rCOS Semantics

command: c	**design: $[\![c]\!]$**	**description**
skip	$\{\}$: *true* $\vdash$ *true*	does not change anything, but terminates
chaos	$\{\}$: *false* $\vdash$ *true*	anything, including non-termination, can happen
$x := e$	$\{x\} : true \vdash x' = val(e)$	side-effect free assignment; updates x with the value of e
$m(e; v)$	$[\![\texttt{var}\ in, out]\!]$; $[\![in{:=}e]\!]$; $[\![body(m)]\!]$; $[\![v{:=}out]\!]$; $[\![\texttt{end}\ in,\ out]\!]$	$m(in; out)$ is the signature with input parameters *in* and output parameters *out*; $body(m)$ is the body command of the procedure/method

rCOS Refinement

Refinement of Designs. The refinement relation between designs is then defined to be logical implication. A design $D_2 = (\alpha, P_2)$ is a **refinement** of design $D_1 = (\alpha, P_1)$, denoted by $D_1 \sqsubseteq D_2$, if P_2 entails P_1

$$\forall x, x', \ldots, z, z' \cdot (P_2 \Rightarrow P_1)$$

where $x, x', \ldots, z, z'$ are variables contained in α. We write $D_1 = D_2$ if they refine each other.

rCOS Component Model : Interface & interface inheritance

A *primitive interface* is a collection of *features* where a feature can be either a *field* or a *method*. We thus define a primitive interface as a pair of feature declaration sections:

$$I = \langle FDec, \ MDec \rangle$$

where *FDec* is a set of *field declarations*, denoted by *I.FDec*, and *MDec* a set of *method declarations*, denoted by *I.MDec*, respectively.

Definition 1. (**Interface inheritance**) *Let I_i $(i = 1, 2)$ be interfaces. I_1 and I_2 are composable if no field of I_i is redefined in I_j for $i \neq j$. When they are composable, notation $I_2 \oplus I_1$ represents an interface with the following field and method sectors*

$$FDec \stackrel{def}{=} FDec_1 \cup FDec_2$$
$$MDec \stackrel{def}{=} MDec_2 \cup \{op(in : U, out : V) | op \in MDec_1 \wedge op \notin MDec_2\}$$

rCOS Component Model : Method Hiding

Definition 2. (**Hiding**) *Let I be an interface and S a set of method names. The notation $I \backslash S$ denotes the interface I after removal of methods of S from its method declaration sector.*

$$FDec \stackrel{def}{=} I.FDec, \quad MDec \stackrel{def}{=} I.MDec \setminus S$$

rCOS Component Model : Contract Definition

Definition 1. *(**Contract**) A contract is a pair $Ctr = (I, MSpec)$, where*

1. *I is an interface,*
2. *$MSpec$ maps each method $op(in : U, out : V)$ of I to a specification design with the alphabet*

$$in\alpha \stackrel{def}{=} \{in\} \cup I.FDec,\ out\alpha \stackrel{def}{=} \{out'\} \cup I.FDec'$$

rCOS Component Model : Composable Contracts

Definition 2. **(Composable contracts)** *Contracts* $Ctr_i = (I_i, MSpec_i)$, $i = 1, 2$, *are composable if*

1. I_1 *and* I_2 *are composable, and*
2. *for any method* op *occurring in both* I_1 *and* I_2,

$$MSpec_1(op(x : U, y : V)) = MSpec_2(op(u : U, v : V))[x, x', y, y'/u, u', v, v']$$

In this case their composition $Ctr_1 \| Ctr_2$ *is defined by*

$$I \stackrel{def}{=} I_1 \oplus I_2, \quad MSpec \stackrel{def}{=} MSpec_1 \oplus MSpec_2$$

where $MSpec_1 \oplus MSpec_2$ *denotes the overriding* $MSpec_1(op)$ *with* $MSpec_2(op)$ *if* op *occurs in both* I_1 *and* I_2.

rCOS Component Model : Reactive Contracts

Definition 3. *(**Reactive Contract**) A reactive contract is tuple $Ctr=(I, Init, MSpec, Prot)$, where*

- *I is an interface*
- *Init is a design that initialises the state and is of the form*

  $$true \vdash Init(v') \wedge \neg wait', \text{ where } Init \text{ is a predicate}$$

- *MSPec assigns each operation to a guarded design (α, g, D).*
- *Prot, called the* protocol, *is a set of sequences of call events. Each is of the form*

  $$?op_1(x_1), \ldots, ?op_k(x_k)$$

  where $?op_i(x_i)$ is a (receipt of) call to operation op_i in I.MDec with an input value x_i.

rCOS Component Model : Contract Definition

Definition 4. **(Semantics of Contracts)** *The* dynamic behavior *of Ctr is described by the triple* (*Prot*, $\mathcal{F}(Ctr)$, $\mathcal{D}(Ctr)$)*, where*

- *the set* $\mathcal{D}(Ctr)$ *consists of the sequences of interactions between Ctr and its environment which lead the contract to a divergent state*

$$\mathcal{D}(Ctr) \stackrel{def}{=} \{\langle ?op_1(x_1), op_1(y_1)!, \ldots, ?op_k(x_k), op_k(y_k)!, ?op_{k+1}(x_{k+1})\rangle \cdot s \mid \exists v, v', wait' \bullet (Init; g_1\&D_1[x_1, y_1/in_1, out_1']; \ldots; g_k\&D_k[x_k, y_k/in_k, out_k'])[true/ok][false/ok']\}$$

where $op_i(y_i)!$ *represents the return event generated at the end of execution of* op_i *with the output value* y_i*,* in_i *and* out_i *are the input and output parameters of* op_i*, and* $g_i\&D_i$ *is the guarded design of method* op_i*.*

- $\mathcal{F}(Ctr)$ *is the set of pairs* (s, X) *where* s *is a sequence of interactions between C and its environment, and X denotes a set of methods which the contract may refuse to respond to after it has engaged all events in* s

$$rej \stackrel{def}{=} (true, false, true, false/ok, wait, ok', wait')$$

$$rej_1 \stackrel{def}{=} (true, false, true, true/ok, wait, ok', wait')$$

$$\begin{aligned}
\mathcal{F}(Ctr) \stackrel{def}{=} & \{(\langle\rangle, X) \mid \exists v' \bullet Init[rej] \wedge \forall ?op \in X \bullet \neg guard(op)[v'/v]\} \\
\cup & \left\{ \begin{array}{l} (\langle ?op_1(x_1), op_1(y_1)!, \ldots, ?op_k(x_k), op_k(y_k)!\rangle, X) \mid \\ \exists v' \bullet (Init; g_1\&D_1[x_1, y_1/in_1, out_1']; \\ \ldots; \\ g_k\&D_k[x_k, y_k/in_k, out_k'])[rej] \wedge \forall ?op \in X \bullet \neg guard(op)[v'/v] \end{array} \right\} \\
\cup & \left\{ \begin{array}{l} (\langle ?op_1(x_1), op_1(y_1)!, \ldots, ?op_k(x_k), op_k(y_k)!\rangle, X) \mid \\ \exists v' \bullet (Init; g_1\&D_1[x_1, y_1/in_1, out_1']; \\ \ldots; \\ g_k\&D_k[x_k, y_k/in_k, out_k'])[rej] \wedge op_k! \notin X \end{array} \right\} \\
\cup & \left\{ \begin{array}{l} (\langle ?op_1(x_1), op_1(y_1)!, \ldots, ?op_k(x_k)\rangle, X) \mid \\ \exists v' \bullet (Init; g_1\&D_1[x_1, y_1/in_1, out_1']; \\ \ldots; \\ g_{k-1}\&D_{k-1}[x_{k-1}, y_{k-1}/in_{k-1}, out_{k-1}'])[rej]; g_k\&D_k[x_k/in_k][rej_1] \end{array} \right\} \\
\cup & \{s, X) \mid s \in \mathcal{D}(Ctr) \wedge \forall ?op \in X \bullet \neg g_i[v'/v]\}
\end{aligned}$$

rCOS Component Model : Contract Refinement

We define the traces of a contract as those traces in the failure set

$$T(Ctr) \stackrel{def}{=} \{s \mid \exists X \bullet (s, X) \in \mathcal{F}(Ctr)\}$$

Definition 6. **(Contract Refinement)** *Contract Ctr_1 is refined by contract Ctr_2, denoted by $Ctr_1 \sqsubseteq Ctr_2$, if*

1. *Ctr_2 provides no less services than Ctr_1:$Ctr_1.MDec \subseteq Ctr_2.MDec$*
2. *Ctr_2 is not more likely to diverge than Ctr_1: $\mathcal{D}(Ctr_1) \supseteq \mathcal{D}(Ctr_2) \lfloor Ctr_1.MDec$, and*
3. *Ctr_2 is not more likely to deadlock than Ctr_1: $\mathcal{T}(Ctr_1) \supseteq \mathcal{T}(Ctr_2) \lfloor Ctr_1.MDec$.*

rCOS Component Model : Removing Services

Definition 7. *(**Removing Services**) Let* $Ctr = (I, Init, MSPec)$ *be a contract and* S *a subset of the operations* $MDec$*, then contract* $Crt \backslash S \stackrel{def}{=} (I \backslash S, Init, MSPec \lfloor (MDec - S))$*, where we use "$-$" for set difference.*

rCOS Component Model : Component Definition

Definition 11. *(Component) A component C is a tuple*

$$(I, MCode, PriMDec, PriMCode, InMDec)$$

where

1. *I is an interface.*
2. *PriMDec is a set of method declarations which are private to the component.*
3. *The tuple* $(I, MCode, PriMDec, PriMCode)$ *has the same structure as a general contract, except that the functions MCode and PriMCode map each method* op *in the sets I.MDec and PriMDec respectively to a guarded command of the form* $g \longrightarrow c$*, where* g *is called the guard, denoted as* $guard(op)$ *and* c *is a command, denoted as* $body(op)$.
4. *InMDec denotes the set of input methods which are called by public or internal methods, but not defined in* $MDec \cup PriMDec$.

rCOS Component Model : Example

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

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

$Prot_{Buffer} = (put, get) * +(put; (get; put)*)$

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*

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

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:*

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

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*

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

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