CoCoME Team Presentation

rCOS—Relational Calculus of Object- and Component-based Systems

Volker Stolz

United Nations University

UNU-IIST
International Institute for Software Technology

rCOS — The Team

UNU-IIST, Macao SAR

- Zhiming Liu (Team Leader)
- Dang Van Hung, Volker Stolz
- Hakim Hannousse, Lu Yang, Liu Yang, Chen Zhenbang

Aalborg, Denmark

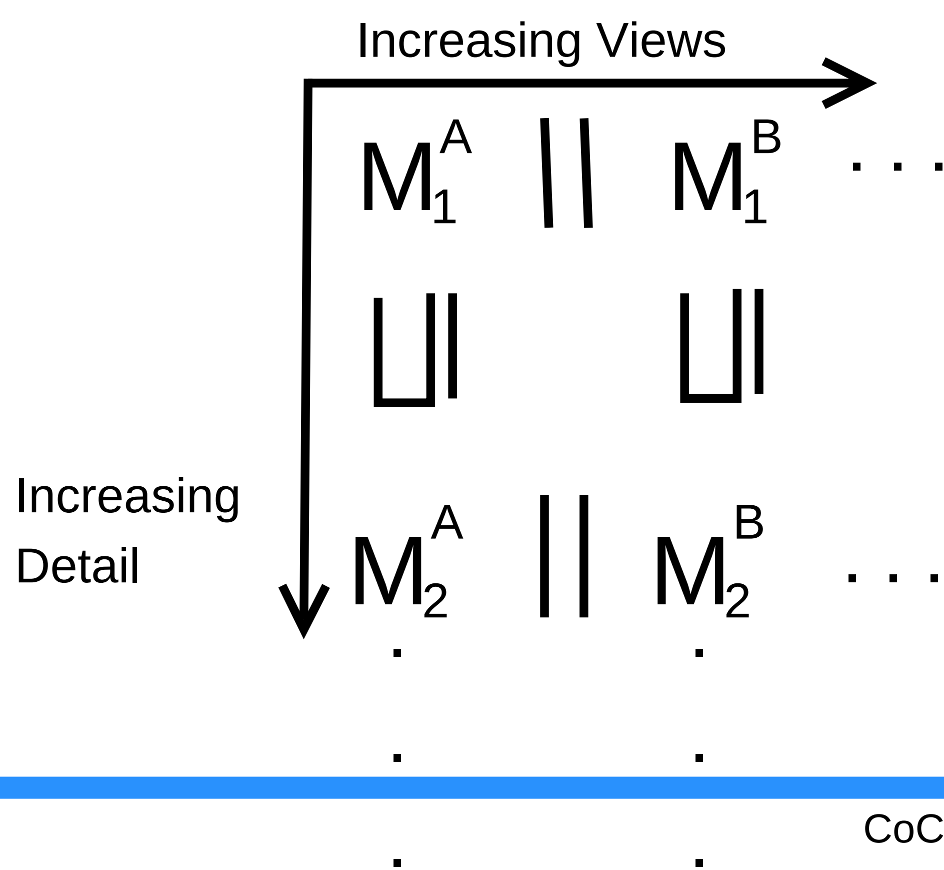
- Anders P. Ravn
- Joseph Okika (formerly UNU-IIST), Istvan Knoll

Beijing, China

- Naijun Zhan (visitor at UNU-IIST), Qu Nan

What is rCOS?

1. model different views of system on different levels of abstraction,
2. provide analysis and verification techniques that assist in showing that models have desired properties,
3. give precise definitions and rules for correctness preserving model transformations



Aspects of the Model

rCOS and its development process are. . .

- Use-case driven
- Object-oriented
- Component-based
- Provably correct

Additionally:

- Code generation through refinement
- Support for formal verification and runtime assurance
- *(Extra-functional properties)*

rCOS Component Model

Combination of:

- Object-based model
- Interface contracts and protocols

Divided into:

- Service components (passive)
- Active components (called *processes*)

Limitations in case study:

- Focus on single use case, simplified
- Any interaction OO-based (method invocation)

Overview of rCOS

Provides formal definitions and rules for manipulation of notions:

- Interfaces: operation signatures
- Contracts: interface specifications including the static and dynamic behaviors, interaction protocol, timing; refinement
- Components: Provided and required interface + code
- Semantics of Components: relation between components and contracts (correctness), substitutability
- Composition operations: simple connectors
- Coordination: connectors, coordinators and glue codes
- Classes: Linking object and component systems

With a semantic root of Hoare & He's UTP

Semantic Foundation: UTP

- A *program* is represented by a *design* $D = (\alpha, P)$, where
  - α denotes the set of variables of the program
  - P is a predicate

$$p(x) \vdash R(x, x') \triangleq (ok \wedge p(x)) \Rightarrow (ok' \wedge R(x, x'))$$

- Refinement as logical implication
- Link to the theory of predicate transformer:

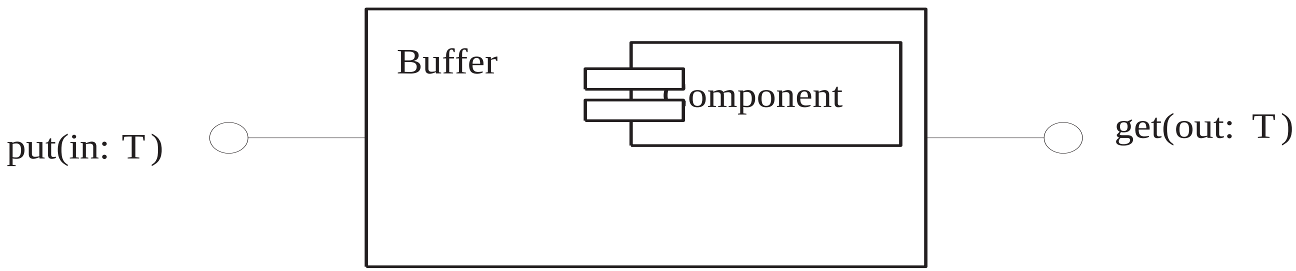
$$\mathbf{wp}(p \vdash R, q) \triangleq p \wedge \neg(R; \neg q)$$

- Guarded designs $g\&D$: $(\alpha,\ P \lhd g \rhd (true \vdash wait' \wedge v' = v))$
- Reactive programs as guarded designs, linking event-based and state-based models

Extended to OOP: rCOS in [TCS 365 (1-2)]

Contract of Interface

Specifies what is needed for a component to be *used* in building and maintaining a software without the need to know its design



Static behavior: [Pre, Post]

Real-time lb, ub

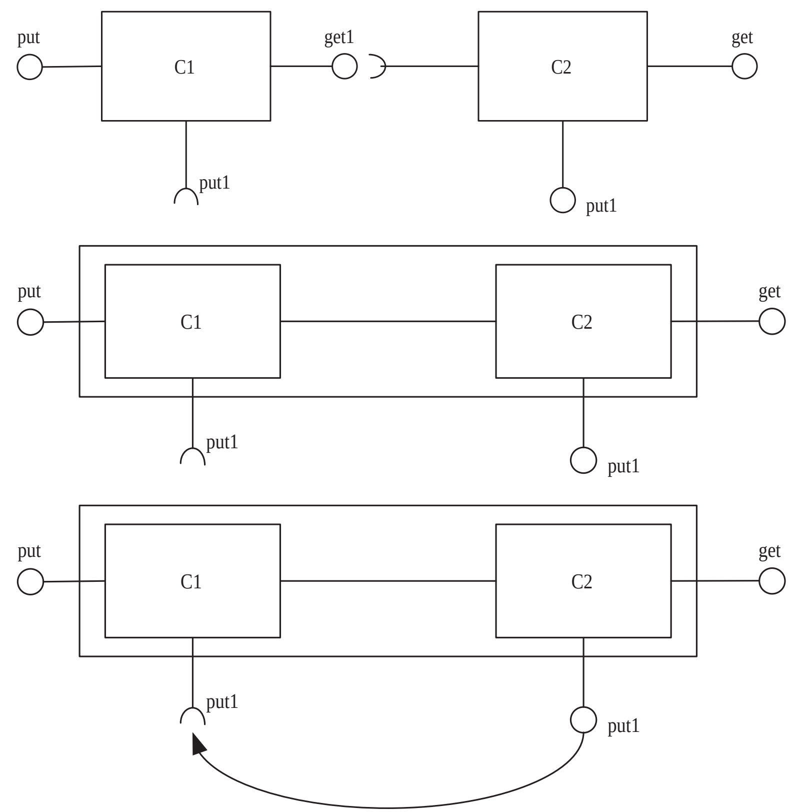
Protocol: ({put(x)get(x): x in T })*: Sequence charts, traces

Dynamic behavior: empty&[Pre,Post] ---- State Machine/Transition systems

Location, address, Resources, ... QoS

Desired to be Multi-View and Multi-Notational & Consistency is Crucial

Component Compositions



Operations:

- Plugging
- Renaming
- Hiding
- Feedback

Also **disjoint union.**

Component Pluggability

Check component protocols:

- Coordination through *synchronous composition*
- P_1 and P_2 does not lead to a deadlock
- Can be checked with e.g. FDR

rCOS World View

Applying rCOS methodology to CoCoME:

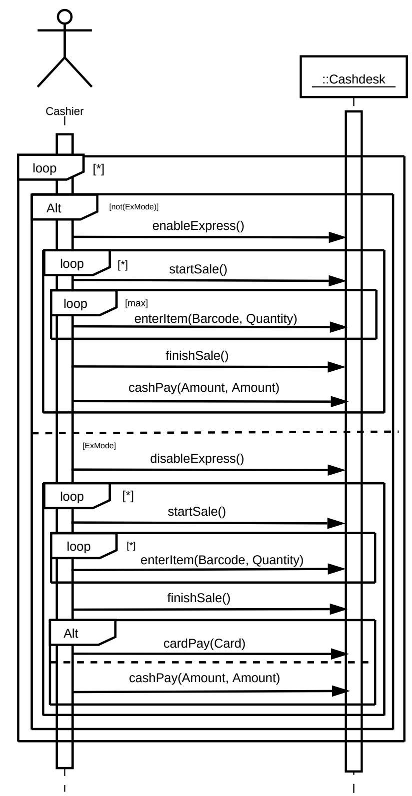
- input: informal problem description
- use case-driven model of requirements using diagrams and UTP
- refinement to code
- from objects to components
- formal verification/analysis
- runtime assurance of properties

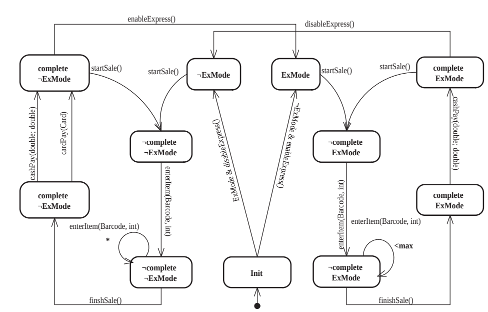
Formalised Requirements

- Sequence diagrams (interaction with Actor)
- State diagrams (control flow in Use Case Controller class)
- Regular trace for rCOS component contracts

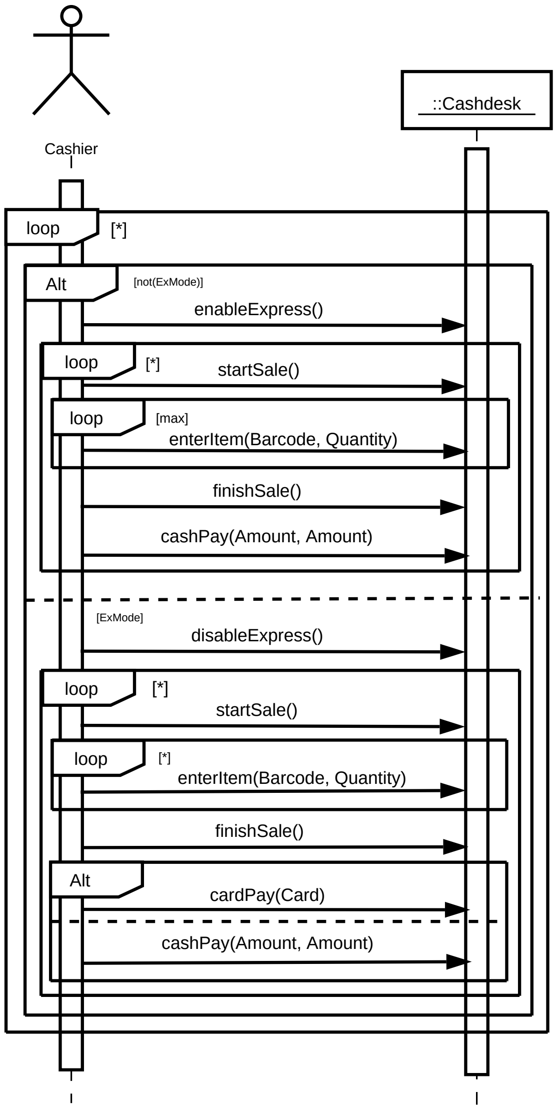
  Derived from informal requirements:
- Functional specifications of operations (pre/post)

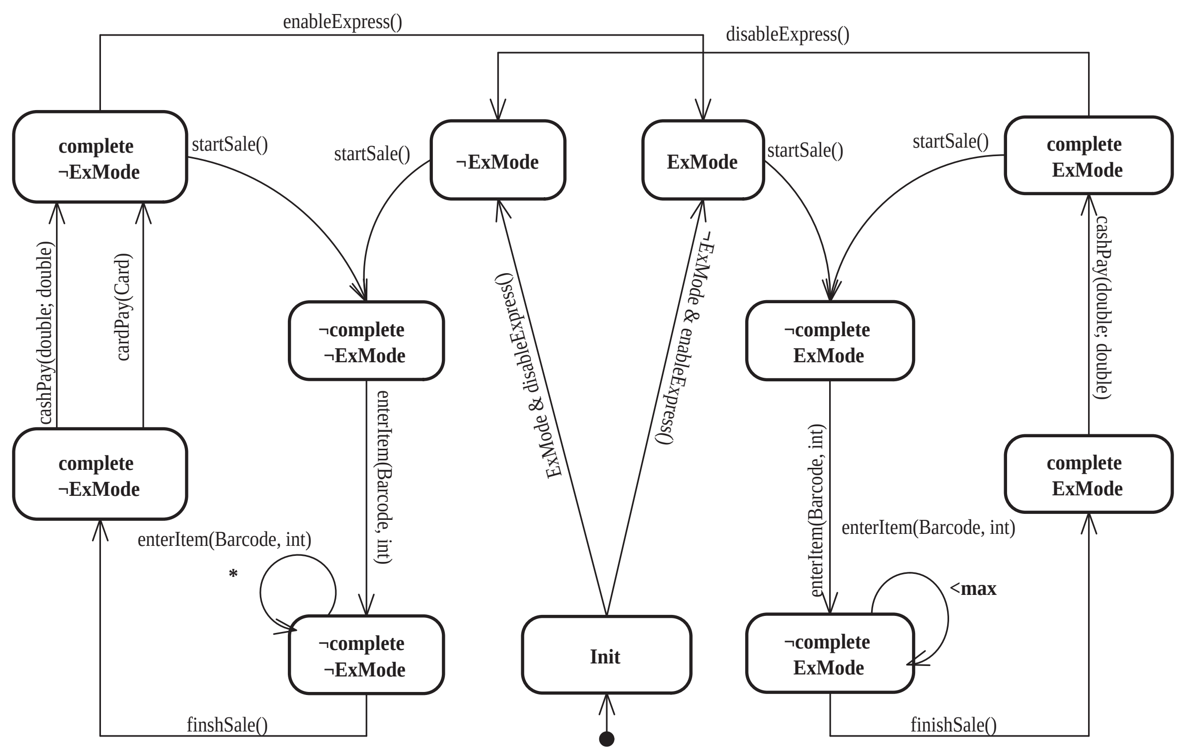
$\Longrightarrow$ Separation of control and data



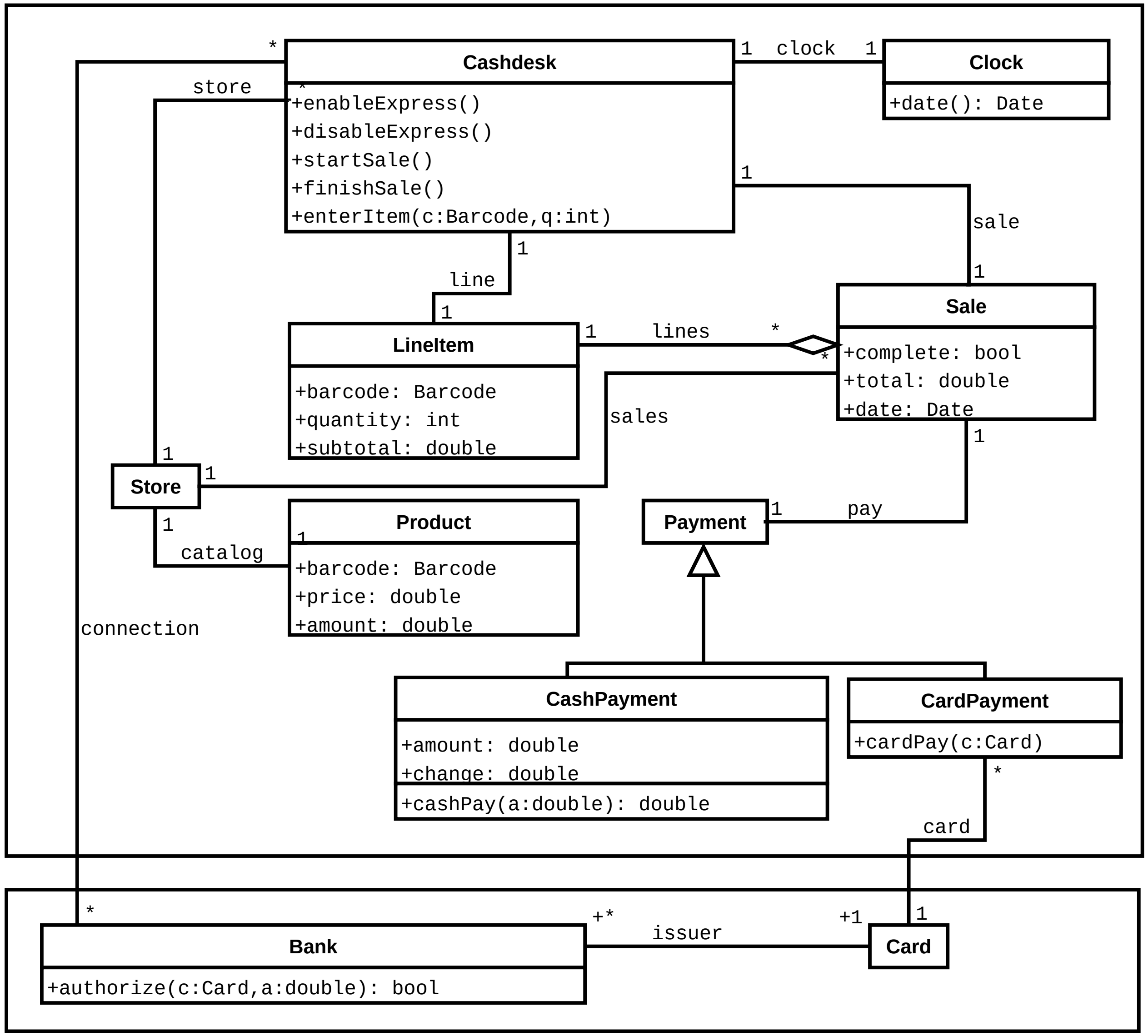


Use Case 1: Sequence and State Diagram





Use Case 1: Class Diagram



Functional Specifications

Object-oriented rCOS:

- Pre-/Post conditions
- Invariants
- Class definition

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

Well-known data structures:

- Lists, Sets, Bags, Arrays
- . . . and their operations (`find`, `add`)
- Quantification (implemented through iteration)

Use Case 1: Functional Specification

Use Case	**UC 1: Process Sale**
class	*Cashdesk*
method	*enterItem(Barcode c, int q)*
	pre: /* there exists a product with the input barcode c */
	store.catalog.find(c) $\neq$ *null*
	post: /* a new *line* is created with its *barcode c* and *quantity q*, and then */
	line' $=$ *LineItem.New(c/barcode,q/quantity)*
	/* the *subtotal* of the *line item* is set, and then */
	; *line.subtotal'* $=$ *store.catalog*.find(c).*price* $\times$ q
	/* add *line* to the current *sale* */
	; *sale.lines*.add(*line*)
invariant	*store* $\neq$ *null* $\wedge$ *store.catalog* $\neq$ *null* $\wedge$ *sale* $\neq$ *null*

Functional Specification, cont'd

Method *finishSale()*

pre: *true*

post: /* *sale* is set to *complete*, and */

$\textit{sale.complete}' = \textit{true}$

/* *sale*'s *total* is calculated */

$\wedge\ \textit{sale.total}' = \text{addAll}[[l.\textit{subtotal} \mid l \in \textit{sale.lines}]]$

Method *cashPay(double a; double c)*

pre: $a \geq \textit{sale.total}$ /* amount is no less than the total */

post: /* the *CashPayment* of the *sale* is created, and then */

$\textit{sale.pay}' = \textit{CashPayment.New(a/amount, a-sale.total/change)}$

/* the completed *sale* is logged in *store*, and */

; *store.sales*.add(*sale*) /* the inventory is updated */

$\wedge\ \forall l \in \textit{sale.lines},\ p \in \textit{store.catalog} \bullet (\textbf{if } p.barcode = l.barcode \textbf{ then}$

$p.amount' = \textit{p.amount} - \textit{l.quantity})$

Checking Consistency of Specifications

- Static consistency (think “*compiler*”):
  - all types and methods are defined
  - type checking of signatures
  - consistency with class diagram
- Dynamic consistency:
  - sequence diagram implements protocol required by state diagram
  - regular trace is abstraction of diagrams
  - application dependent properties

Dynamic checking: e.g. through FDR

Object-oriented Design

From functional specifications to code:

- rCOS allows big-step refinement
- provably correct rules/design patterns
- often directly realizable in high-level programming languages:

`updateInventory()`

$\forall l \in$ *sale.lines*, $p \in$ *store.catalog* • (**if** p.*barcode*=l.*barcode* **then**
p.*amount*' = p.*amount* -l.*quantity*)

yields almost executable Java with assertions:

```
class Product::        update(int qty) { amount := amount-qty }
class set(Product)::   update(Barcode code, int qty) {
                         Iterator i := iterator();
                         while (i.hasNext()) {
                           Product p := i.next();
                           if p.barcode=code then p.update(qty); }
class Store::          update(Barcode code, int qty) { catalog.update(code,qty) }
```

Object-oriented Design, cont'd

```
public void enterItem(Barcode code, int quantity) throws Exception {
    if (find(code) != null)
    {
        line = new LineItem(code, quantity);
        line.setSubtotal(find(code).getPrice()*quantity);
        sale.addLine(line);
    }
    else
    {
        throw new Exception("Can't␣Find␣Product");
    }

}

protected Product find(Barcode code) {
    if( code == null )
    {
        System.out.println("Product::find():code␣is␣null");
        return null;
    }
    return Store.store.find(code);
}
```

UNITED NATIONS UNIVERSITY

Software Engineering, Formalised

Navigation path in method `C.m()`:

$$C :: \quad m()\{ \quad c(a_{11}.\dots.a_{1k_1}.x_1, \dots, a_{\ell 1}.\dots.a_{\ell k_\ell}.x_\ell)$$
$$\wedge \quad \mathit{le}' = e(b_{11}.\dots.b_{1s_1}.y_1, \dots, b_{ts1}.\dots.b_{ts_t}.y_t)\ \}$$

Refinement:

$$C :: \quad \mathit{check}()\{\mathit{return}' = c(a_{11}.get_{\pi_{a_{11}}x_1}(), \dots, a_{\ell 1}.get_{\pi_{a_{\ell 1}}x_\ell}())\}$$
$$m()\{\textbf{if } \mathit{check}() \textbf{ then } r_1.\mathit{do\text{-}m}_{\pi_{r_1}}(b_{11}.get_{\pi_{b_{11}}y_1}(), \dots, b_{s1}.get_{\pi_{b_{s1}}y_s}())\}$$

$$T(a_{ij}) :: \quad get_{\pi_{a_{ij}}x_i}()\{\mathit{return}' = a_{ij+1}.get_{\pi_{a_{ij+1}}x_i}()\}\ (i : 1..\ell, j : 1..k_i - 1)$$

$$T(a_{ik_i}) :: \quad get_{\pi_{a_{ik_i}}x_i}()\{\mathit{return}' = x_i\}\ (i : 1..\ell)$$

$$T(r_i) :: \quad \mathit{do\text{-}m}_{\pi_{r_i}}(d_{11}, \dots, d_{s1})\{r_{i+1}.\mathit{do\text{-}m}_{\pi_{r_{i+1}}}(d_{11}, \dots, d_{s1})\}$$
$$\text{for } i : 1..f - 1$$

$$T(r_f) :: \quad \mathit{do\text{-}m}_{\pi_{r_f}}(d_{11}, \dots, d_{s1})\{x' = e(d_{11}, \dots, d_{s1})\}$$

$$T(b_{ij}) :: \quad get_{\pi_{b_{ij}}y_i}()\{\mathit{return}' = b_{ij+1}.get_{\pi_{b_{ij+1}}y_i}()\}\ (i : 1..t, j : 1..s_i - 1)$$

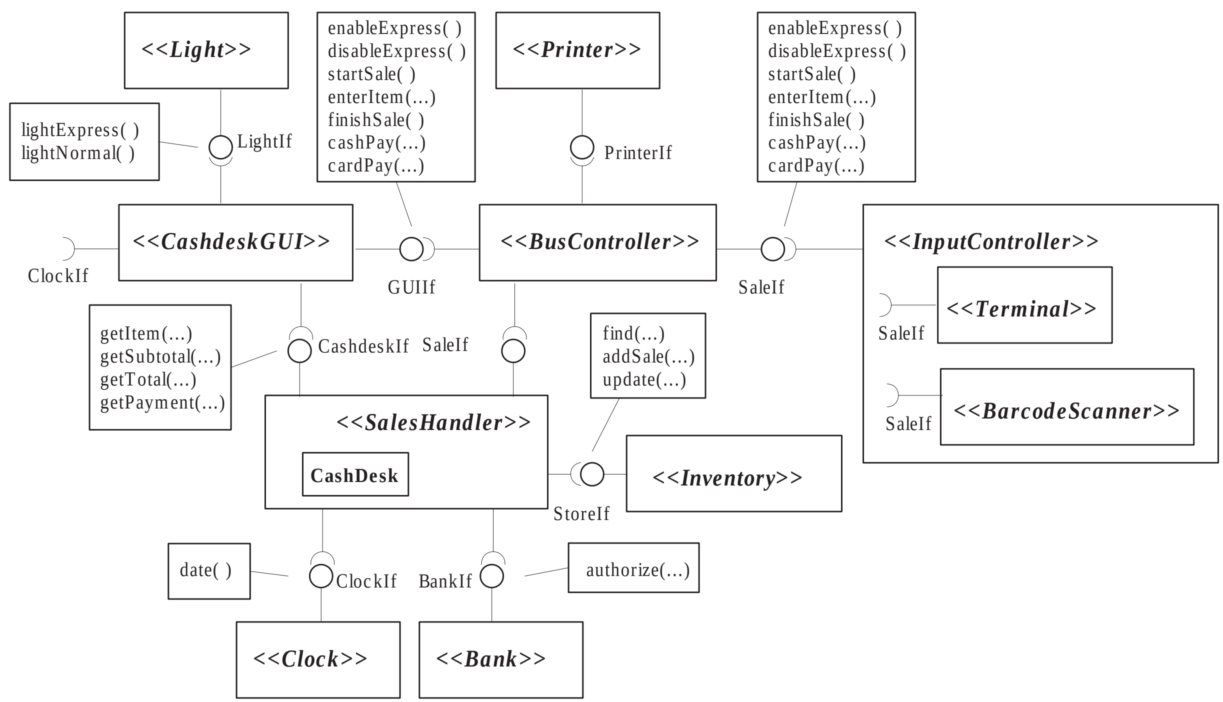
$$T(b_{is_i}) :: \quad get_{\pi_{b_{is_i}}y_i}()\{\mathit{return}' = y_i\}\ (i : 1..t)$$

Annotations through JML

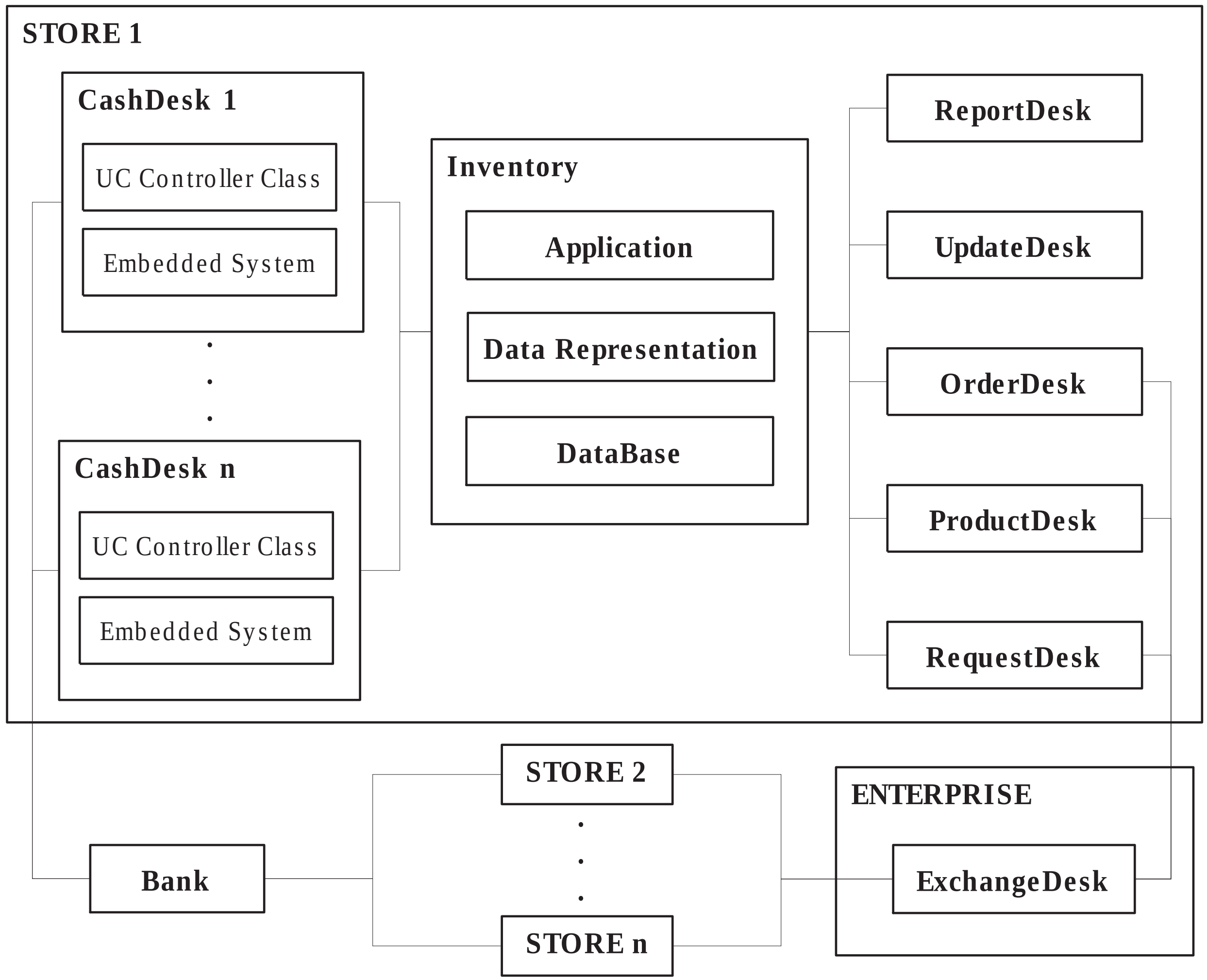
```
/*@ public normal_behaviour
  @   requires (\exists Object o; theStore.theProductList.contains(o);
  @                ((Product)o).theBarcode.equals(code));
  @   assignable theLine, theSale;
  @   ensures   theLine != \old(theLine) &&
  @                 theLine.theBarcode.equals(code) &&
  @                 theLine.theQuantity == quantity &&
  @                 (\exists Object o; theStore.theProductList.contains(o);
  @                 ((Product)o).theBarcode.equals(code) ==>
  @                 theLine.theTotal == ((Product)o).thePrice * quantity) &&
  @                 theSale.theLines.size() == (\old(theSale.theLines.size()) + 1) &&
  @                 theSale.theLines.contains(theLine);
  @ also
  @ public exceptional_behaviour
  @         requires !(\exists Object o; theStore.theProductList.contains(o);
  @                        ((Product)o).theBarcode.equals(code));
  @         signals_only Exception;
  @*/
public void enterItem(Barcode code, int quantity) throws Exception;
```

From OOA to CBA

- Identify *components* and inter-component *interfaces* (based on traces from Use Cases)
- Model *hardware* for closed system and verification
- Add middleware and change OO interface to concrete interaction mechanisms: RMI, eventChannel,. . .



Enterprise Overview



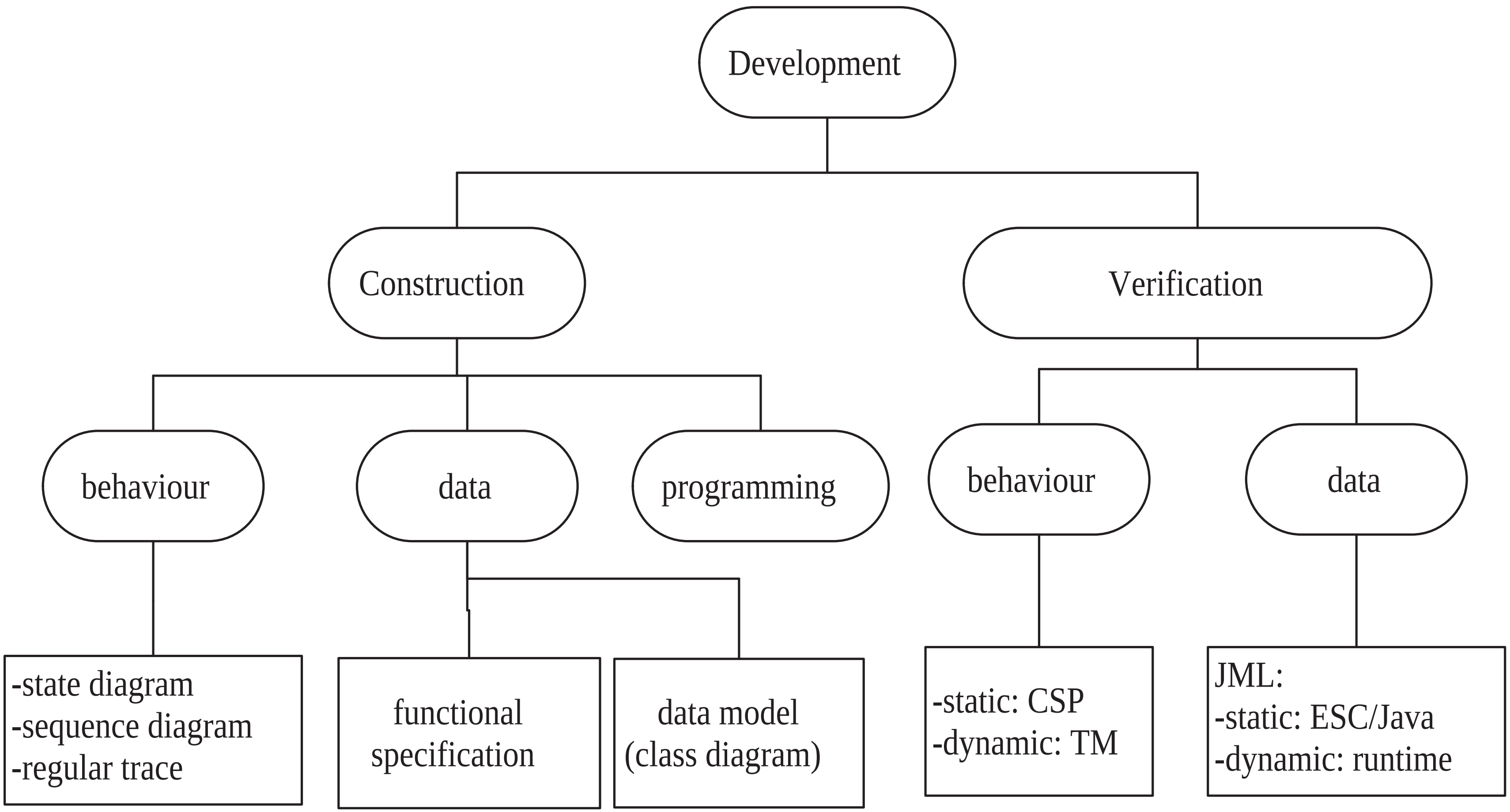
Components, rCOS View

define SaleIf { enableExpress(), disableExpress(), startSale (), enterItem (..),
finishSale (), cardPay (..), cashPay(..) }

component Terminal
required interface SaleIf
protocol { ([disableExpress!] startSale! enterItem!* finishSale! [cardPay! | cashPay!])* }

component SalesHandler
required interface ClockIf { date() }
required interface BankIf { authorize (..) }
required interface StoreIf { update (..), find (..), addSale(..) }
provided interface SaleIf
provided pure interface CashdeskIf { getItem(..), getSubTotal (..), getTotal (..), getPayment() }
protocol { ([?enableExpress (?startSale date! (?enterItem find !)$^{(max)}$?finishSale
?cardPay authorize! addSale!)*
| ?disableExpress (?startSale date! (?enterItem find !)* ?finishSale
[?cardPay authorize! addSale! update!*
| ?cashPay addSale! update!*])*)* }
class *Cashdesk* **implements** SaleIf, CashdeskIf

rCOS Development Process



Formal Verification: Consistency

```
-- State Diagram:
datatype Mode = on | off
State = Init(on) [] Init(off)
-- Resolve outgoing branches non-deterministically:
Init(mode) = (if mode == on then disableExpress -> StateNormalMode(off)
                               else STOP)
             [] (if mode == off then enableExpress -> StateExpressMode(on)
                               else STOP)
StateNormalMode(mode) = (startSale -> enterItem -> StateEnterItemLoopStar)
                      ; finishSale -> ((StateCashPay [] StateCardPay)
                        ; ((enableExpress -> StateExpressMode(on))
                           [] StateNormalMode(mode)))

StateEnterItemLoopStar = SKIP [] (enterItem -> StateEnterItemLoopStar)
StateCashPay = cashPay -> SKIP
StateCardPay = cardPay -> SKIP

StateEMode(c) = if c == 0 then SKIP
                         else (SKIP [] (enterItem -> StateEMode(c-1)))

-- Check trace equivalence:
assert State [T= Trace
--^ does not hold as trace abstracts from the guard,
--  permits: enableExpress -> ... -> enableExpress
assert Trace [T= State
```

Formal Verification: Pluggability

Component composition verified by FDR:

```
channel bc_enterItem , bc_startSale , bc_finishSale
channel sale_startSale , sale_enterItem , sale_finishSale
BusController = (bc_startSale -> sale_startSale -> cdg_startSale -> SKIP)
              ; BCItemLoop
              ; (bc_finishSale -> sale_finishSale -> cdg_finishSale -> SKIP)
              ; BusController
BCItemLoop = bc_enterItem -> sale_enterItem -> cdg_enterItem -> (SKIP [] BCItemLoop)

CDG2 = CDG1 [| {| cdg_startSale , cdg_enterItem , cdg_finishSale |} |] BusController

-- can we still execute the initial protol or did we lose anything?
-- we need to hide events newly introduced through par. composition
assert CDG2 \ {| bc_enterItem , bc_startSale , bc_finishSale , sale_startSale ,
                 sale_enterItem , sale_finishSale |} [T= CDG
-- Did we introduce too much behaviour?
assert CDG [T= CDG2 \ {| bc_enterItem , bc_startSale , bc_finishSale , sale_startSale ,
                         sale_enterItem , sale_finishSale |}
```

Effort

- 1 day for initial outline of a single use case specification
- 4-man week for remaining use cases
- 1 day for sample refinement of first use case
- 4-man week for remaining use cases
- one week of experimenting with CSP/FDR to get first results
- generated 65 classes / 4000 LoC in Java

  Only for business logic, does not yet include:
  - GUI
  - middle ware
  - glue code

Requirements Elicitation Takes Time!

Where did most of the time go?

- a lot of time spent on synchronizing with ongoing changes as the requirements stabilize
- even more time necessary to ensure that specifications are actually consistent (implementing and checking the diagrams in a specification language like CSP)
- MOST of the time: discussing HOW to make a change

Advantage of *separation of concerns*:

- most problems related to control flow, not data
  $\Rightarrow$ different aspects of single use case tackled in parallel

Limitations

CoCoME specification only semi-formal rCOS:

- no machine-readable notation
- no automation/tool-support yet
- formal “pen-and-paper” proofs impractical for even medium-sized systems that keep evolving

On the practical side:

- how to model *middle ware* and *deployment*?
- currently unmanaged *glue* code

Future Work

Desirables:

- managed specifications within single framework
- assure static consistency
- (correct) translation into input for respective tools
  - refinement (patterns, QVT)
  - software (business logic AND runtime checking)
  - verification (CSP, JML)
- interpret/visualize verification results

$\Longrightarrow$ framework for rCOS methodology

Tool Support

Required tools:

- machine-readable syntax, foundation for tools (Eclipse Modeling Framework EMOF? DSLs?)
- static constistency/welformedness checker
- PVS prover (Aalborg, DK)
- dynamic consistency checker (CSP/FDR)
- model-driven development-tool support through transformations (QVT)
- generate runtime assurance: assertions, trace properties
- back end: Java, Spec#

Additional Investigation

Not shown here:

- Extra-functional properties (Realtime, QoS)
- SOA/Webservice-based prototype
- SMALLTALK-based prototype

More information:

`http://www.iist.unu.edu/cocome/`

Conclusion

Application of rCOS in case study:

Component = OO class(es) + Contract

- driven by use cases
- ensures consistency of multi-view specifications
- provably correct refinement steps into OO code
- assertions and verification based on formal specification

To do:

- automate development / tool support
- formally model middle ware

Bibliography

- **Harnessing Theories for Tool Support**,
  Z. Liu, V. Mencl, A. P. Ravn, L. Yang; ISoLA 2006
- **Automating Correctness Preserving Model-to-Model Transformation in MDA**,
  L. Yang, V. Mencl, V. Stolz, Z. Liu; AWCVS 2006
- **Separation of Concerns and Consistent Integration in Requirements Modelling**,
  X. Chen, Z. Liu, V. Mencl; SOFSEM 2007.
- **A Refinement Driven Component-based Design**,
  Z. Chen, Z. Liu, V. Stolz, L. Yang; ICECCS 2007.