Model Integration, Refinement, and Transformation

Zhiming Liu

Birmingham City University

zhiming.liu@bcu.ac.uk

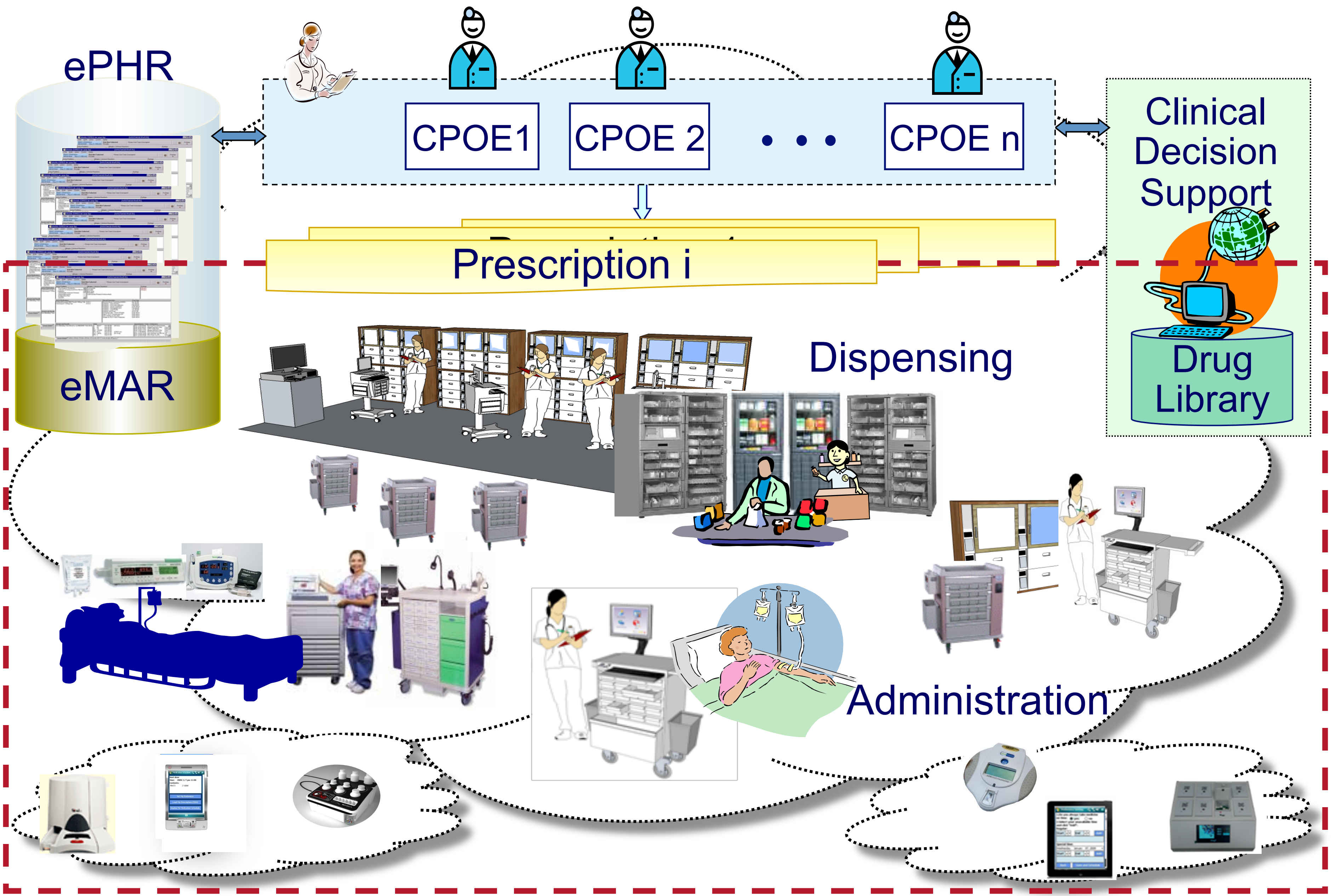
Deal with Software Complexity

- **Inherent Complexity of Software**
  1) Application domains are complex [**Requirement Analysis**]
  2) Software offers much flexibility [**Design**]
  3) The development process is still changing [**Management**]
  4) The behavior of a software system is hard to understand [**V&V**]
- **Increasing complexity of modern software**
  1) Models of different views of system data and services (model transformations)
  2) Integration of models and services, say to support collaborative workflows
  3) More and more software becomes safety critical, has increasing demand on privacy, security, maintainability, interoperability
- **Formal methods are essential for**
  1) handling complexity through abstraction, separation of concerns and divide and conquer, as well as for
  2) provably correct system design

An example scenario: Internet of Things?

- **Street lights**
  1) City council's view: lighting of streets
  2) Electricity company's view: readings on meters and/or bills
  3) Police's view: in relation with crimes.
- **Design a street light control system to serve the interests of all the three kinds of users?**
  1) How to derive an engineering/system model from the different views of users and **vice versa**?
  2) How to reason or validate the system model against requirements stated in the users' view models?
  3) How to design a system to support dynamic addition of support to users with different views?
- **Model integration and transformation**

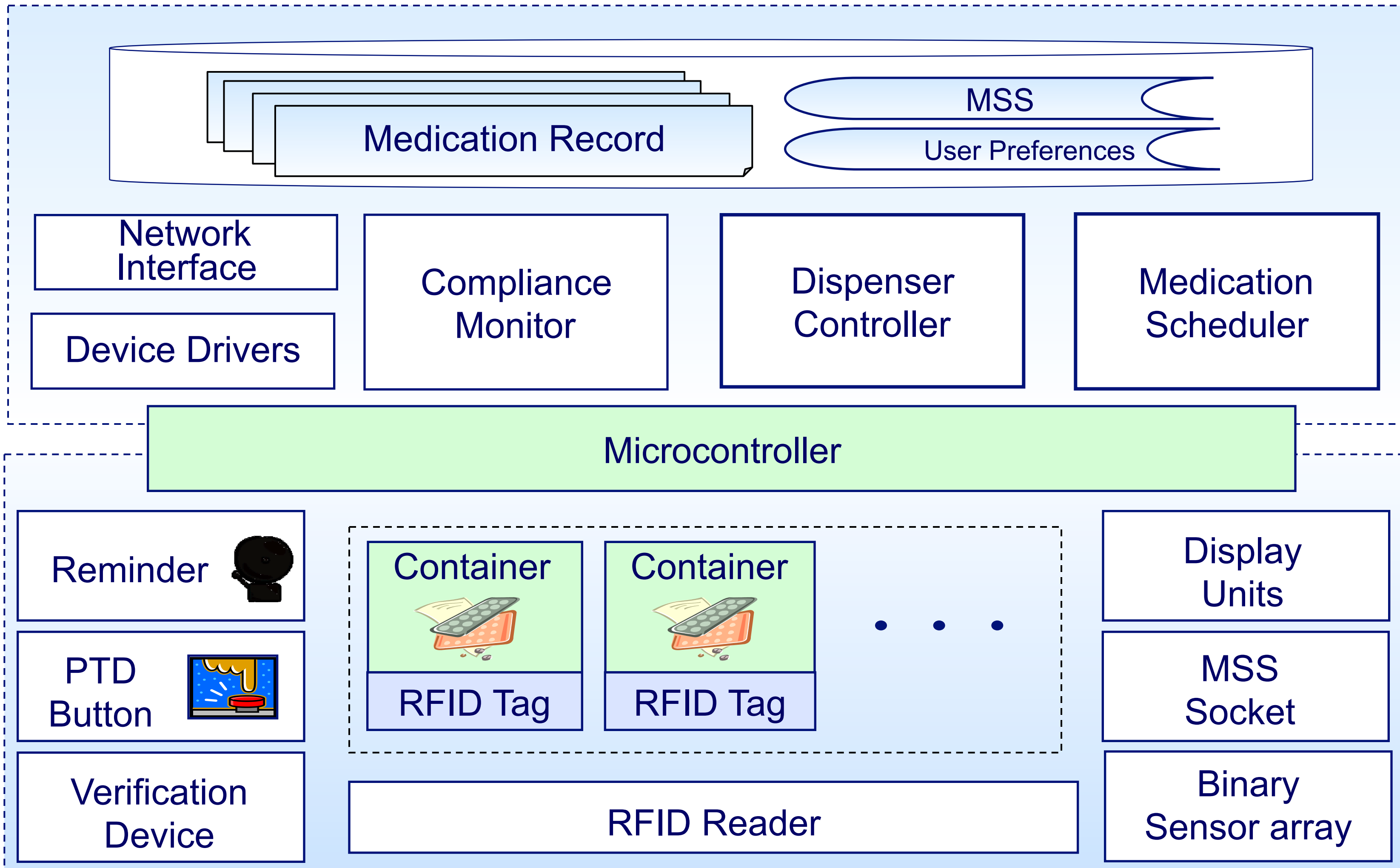
Tool Chain for Medication Use Process



Even an Intelligent Medicine Dispenser - Jane Liu

SINICA
Medication dispenser

Self-Contained Dispenser



Implement Instructions

- Pain killer: 1 tablet every 4 to 6 hours while symptom persists. If pain does not respond to 1 tablet, 2 tablets may be used but do not exceed 6 tablets in 24 hours. The smallest effective dose should be used."
  - Dose size: $[d_{min}, d_{max}] = [1, 2]$
  - Separation: $[s_{min}, s_{max}] = [4, 6]$
  - Maximum total intake: $(B, R) = (6, 24)$
  - Minimum total intake: $(L, P) = (0, 24)$
- Antibiotic: Take 2 to 4 tablets every eight hours. Keep taking this medicine for at least ten days.
  - Dose size: $[d_{min}, d_{max}] = [2, 4]$
  - Separation: $[s_{min}, s_{max}] = [8, 8]$
  - Maximum total intake: $(B, R) = (12, 24)$
  - Minimum total intake: $(L, P) = (6, 24)$
  - Duration: $[T_{min}, T_{max}] = [240, 240]$

rCOS

Problems

- Dynamic integration of new components and legacy components – plug & play
- Interface for integration for interoperable interaction among heterogeneous components, e.g. CPS- cyber and physical components, and sensors
- Specification purpose of integration – models of workflows

Objectives

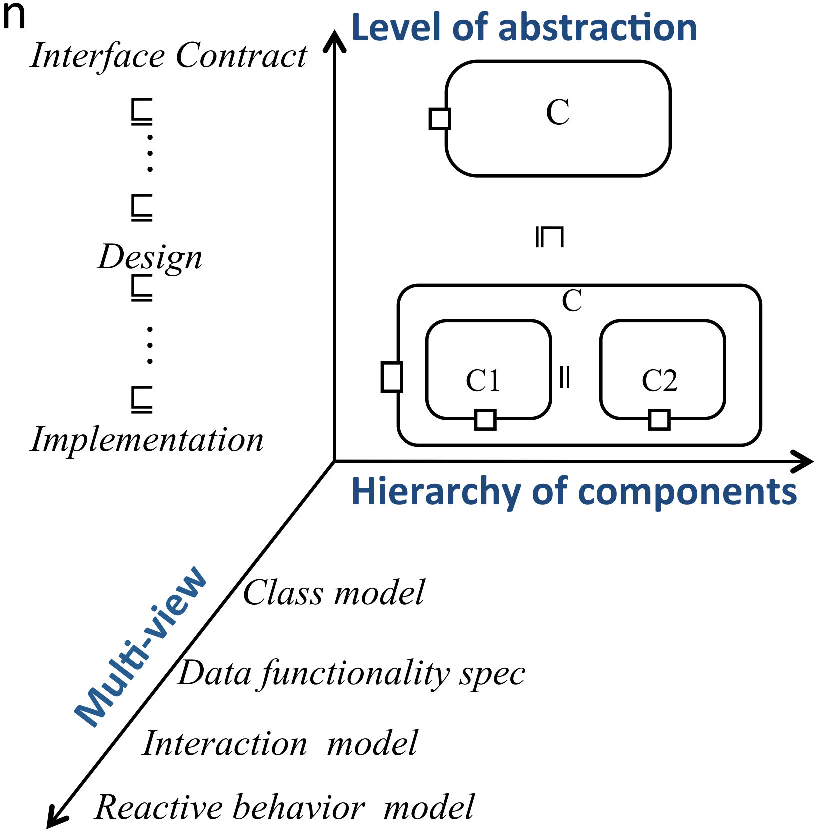
- **Unifying semantics & theories of programming (UTP)**
- models of interfaces, their refinement and composability
- models, analysis, verification and simulation
- **System architecture modeling, refinement and transformation**
- **Language and Tool support for integration**

Putting theories, methods and tools consistently together in design processes

rCOS – Integrating Models

Build ***system models*** to gain confidence in requirements and designs

- Use **abstraction** for information hiding
  - well-focused
  - problem-oriented
- Use **decomposition** and **separation of concerns**
  - divide and conquer
  - incremental development
- use **rigor/formalization**
  - repeatable process
  - analyzable artifacts



Basis for Tool Support

Architectural Components

- **Components are**
  1) Services providers, including computing devices realize functions
  2) Process that coordinating and control components through interactions and
  3) Connectors
- A memory component

```
 Component M {
    Z d;
provided interface MIF {
  W(Z v) { d:=v };
   R(;v) { v:=d };
   } }
```

- A processes – state-action transition systems, CSP or CCSP processes

```
Component C {
   Bool w = 1;
   Provided interface CIF { w(){(w:= not w)};R(){not w&(skip}
}
 Component C1: w(){w&(w:=not w)}, r()(not w&(w:=not w)
```

M@C, M@C1 are components

More General Component

```
component fM {
  Z d;
  provided interface MIF {
  W(Z v) { d:=v };
   R(;v) { v:=d };
  protocol { ?W({?W,?R}) // protocol of C, realized by guards}
  }
 actions { //fault modeling corruption
      fault {true|- d' < > d }
  }
}
```

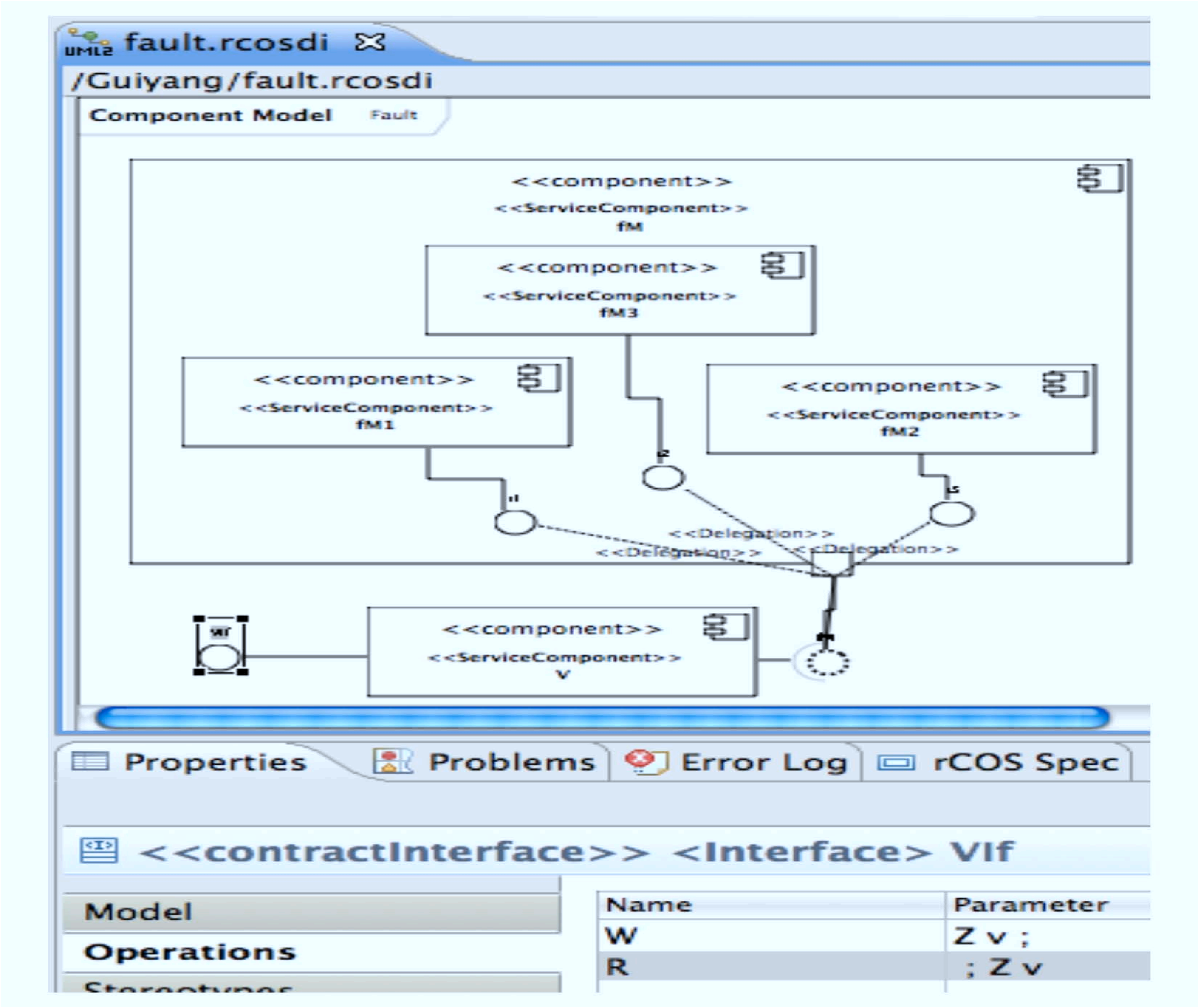
fMi=fM[fMi.W/W,fMi.R/R], i=1,2,3, renaming as a built-in connector

Separation of Concerns

```
component V {//a connector
    provided interface VIF {
        W(Z v) { fM1.W(v);fM2.W(v);fM3.W(v) };
        R(;v) { v := vote (fM1.R(v), fM2.R(v), fM3.R(v)) };
     protocol { ?W({?W,?R}) }
    }
    required interface {  // union of fM1, fM2, fM3;
    protocol { // interleaving of all fMi's protocols }
   }
}
```

- **M ⊑ V<<(fM1||FM2||fM3)   provided at most one memory is corrupted**
- **Verification, need auxiliary variable**

System Composition



Semantic Foundation - UTP

- A semantic definition is about a way to observe the execution of a program
- For a sequential program P, we observe the relation between the initial states and final states
  - let $\alpha(P)' = \{x' \mid x \text{ in } \alpha(P)\}$
  - A sequential program defines relation between its initial and final states, described as a design $p(x) \vdash R(x,x')$ defined by
    - partial correctness $p \Rightarrow R$
    - total correctness $L(p|-R) = (ok \wedge p \Rightarrow ok' \wedge R)$
- Framed design: $\beta: p(x) \vdash R(x,x')$

Theorem: Programs are Indeed Designs

Skip = {}:true|- true

x:=e = {x}:true|- x' = e

D1;D2 = **∃x0. D1[x0/x']∧D2[x0/x]** **//** (p |- R)**

if b then D1 else D2 = **b∧D1∨ ¬b ∧D2** **// ** (p |- R)**

D1⊓D2 = **D1∨D2** **// ** (p |- R)**

b*D = **if b then (D; b*D) else skip** **//** (theory of fixed point)**

chaos = **false ⊢ true**

Refinement of Sequential Programs

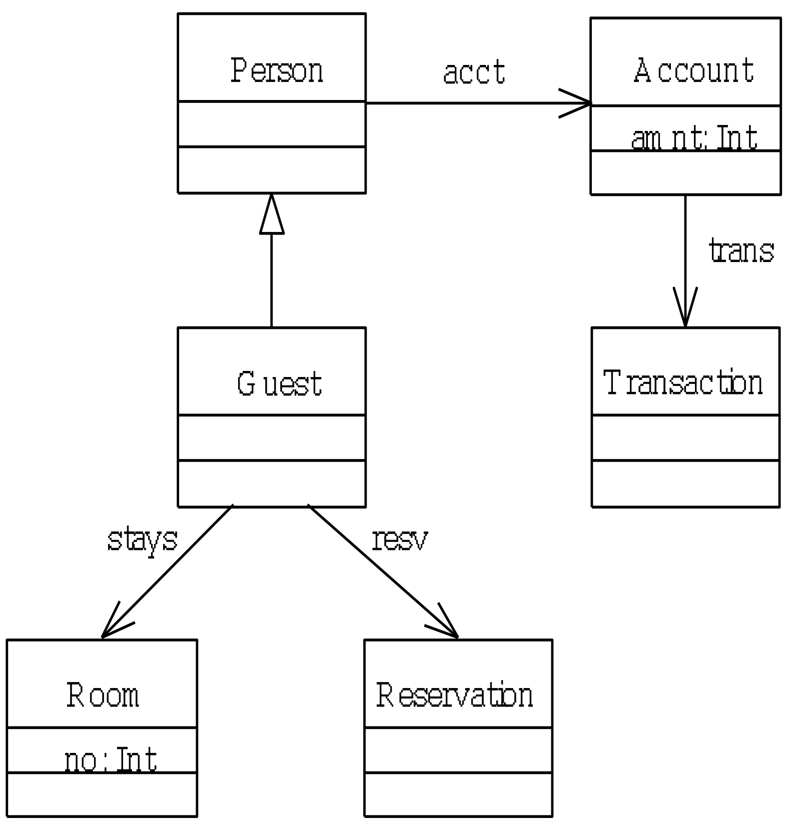
- Refinement: **D1 ⊑ D2 if ∀ x,x', ok,ok'. (D2 ⇒ D1)**
- Theorem **(Designs, ⊑, chaos)** is complete lattice, and **b*D** is a design
- Theorem: **p1 ⊢ R1 ⊑ p2 ⊢ R2 iff**
  [P1 ⇒ P2] and [R2 ∧ P1 ⇒ R1]
- Laws of programming:
  1. **if** b **then** D1 **else** D2 = **if** ¬ b **then** D2 **else** D1
  2. Chaos; D = chaos
  3. D; skip = skip; D = D
  4. (D1 <| b|> D2); D) = (D1;D)<|b|> (D2;D)
- Data refinement: (α1, D1) ⊑ (α2, D2) iff there exists design ρ(α2, α1') such that
  ρ; D1 ⊑ **D2; ρ**

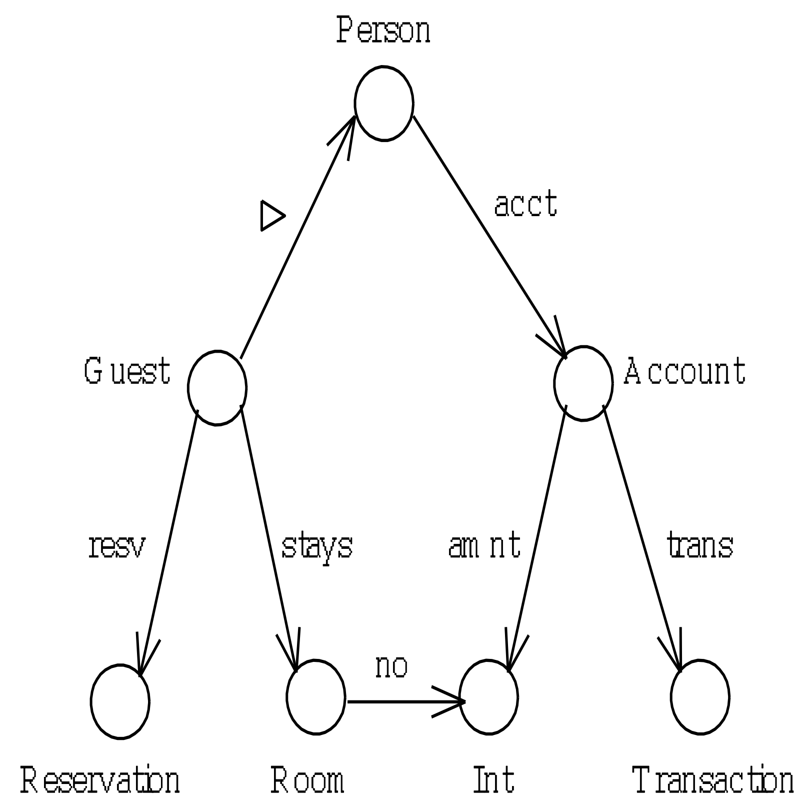
Dijkstra wp and Hoare Logic, OO

- $\{pre\}\ p \vdash R\ \{post\} \triangleq (p \wedge pre) \Rightarrow (R \wedge post')$
- $wp(p \vdash R,\ q) \triangleq p \wedge \neg(R;\ \neg q)$

Both calculus of wp and Hoare logic can be used for reasoning and verification in rCOS

Class Graph


Person
acct
Account
am nt: Int
trans
Transaction
Guest
stays
resv
Room
no: Int
Reservation


Person
acct
Guest
Account
resv
stays
am nt
trans
no
Reservation
Room
Int
Transaction

OO Programs

- An oo program P consists a list of class declarations and a main method

  ClassDecls•Main, where Main = (var, c)

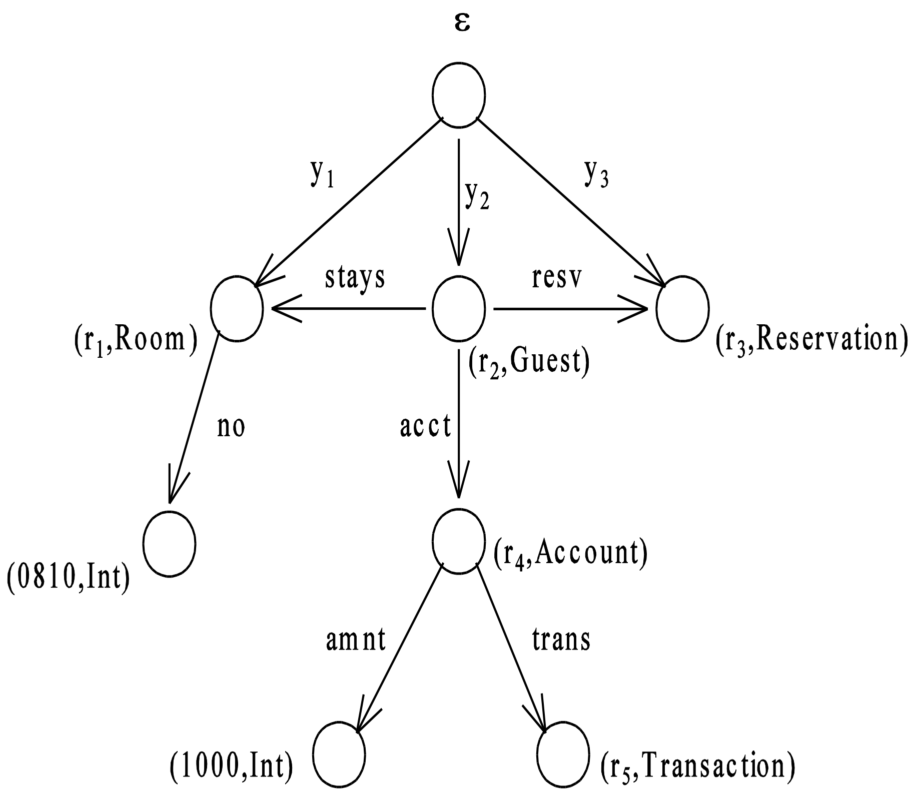
  - ClassDecls can be represented by a **UML Class Diagram**, but by a directed and label graph in rCOS
  - A state of P can be represented a **UML Object Diagram,** but by a rooted, directed and labeled graph
  - The execution of a deterministic command c changes one state graph to another -- relate an initial state to a final
  - When non-determinism allows, the semantics of c can be defined as

    p |-- R

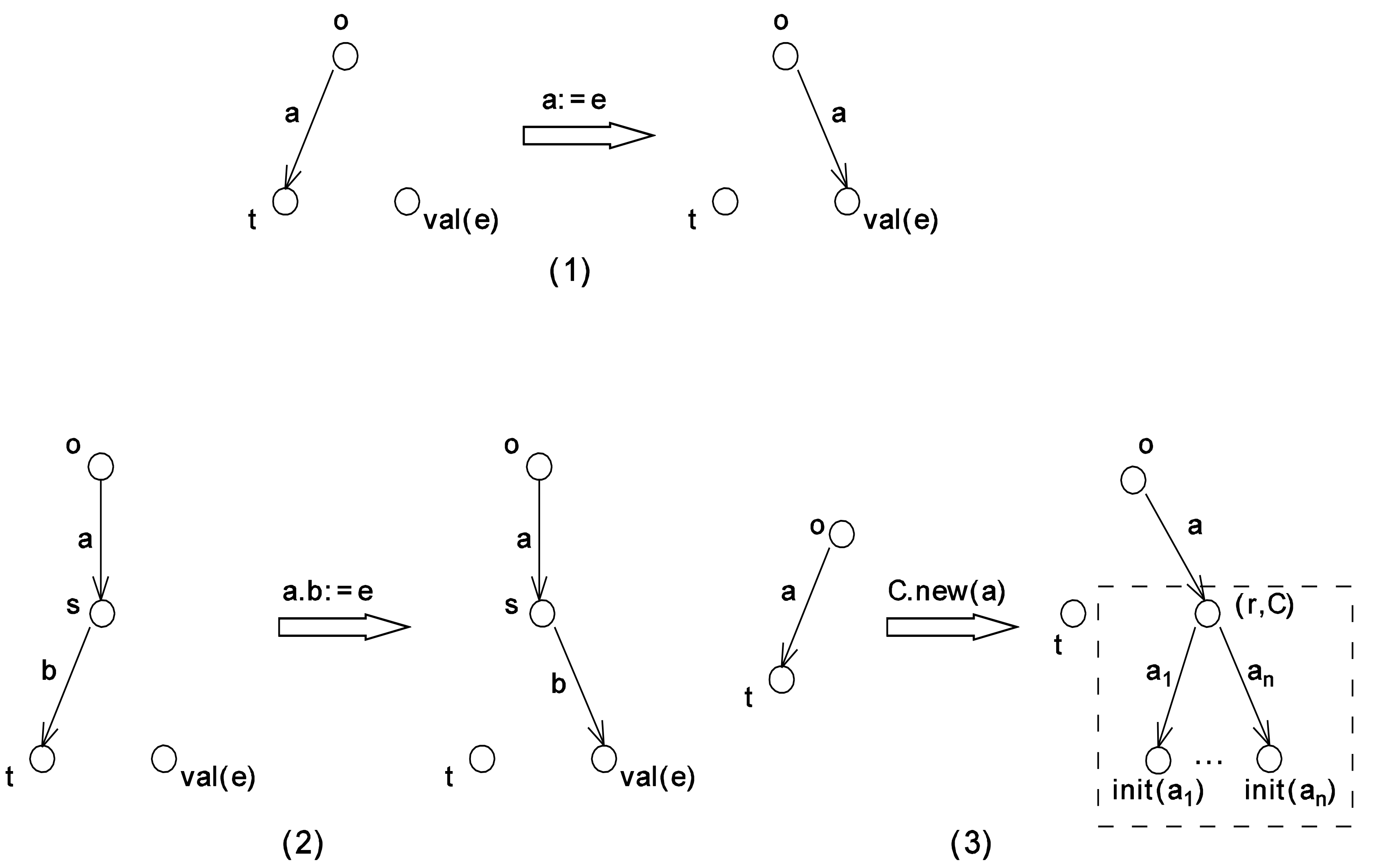
  where p is a predicate on state graphs and R is relation between graphs

State Graph

- The root is the instance of main class
- An object and its related objects is a sub-graph rooted with object
- Primitive attributes are leaves
- Object (graph) and state graph are typed by class graph
- State graphs with local variables (stack)?



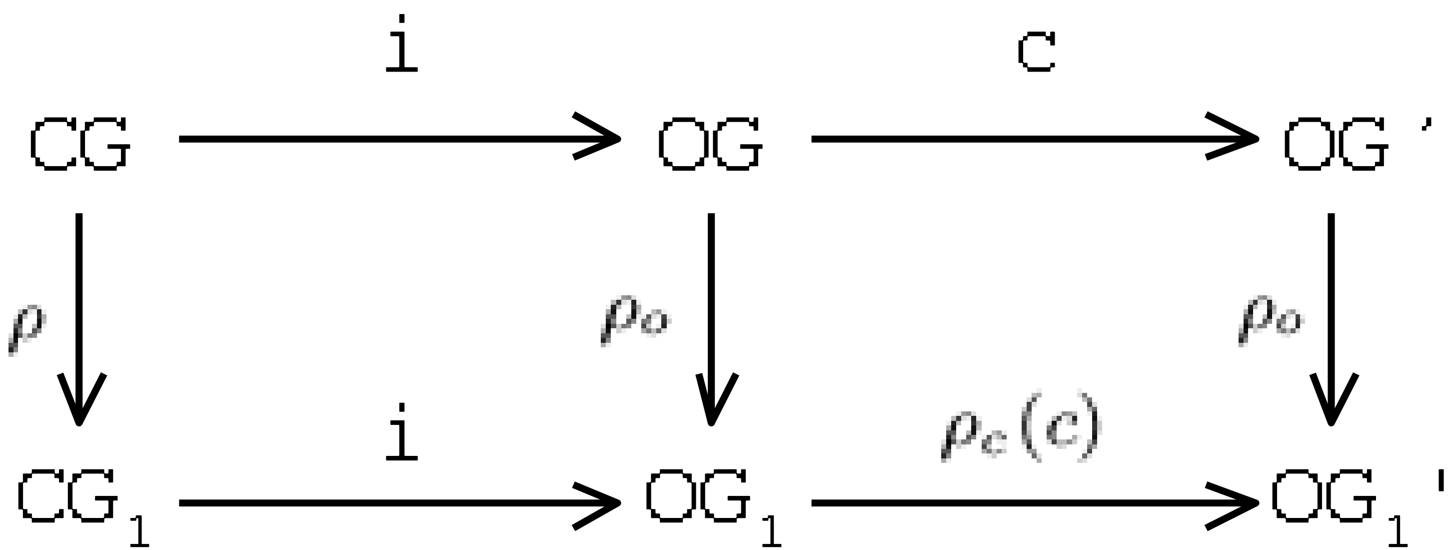
OO Semantics [TCS 2009,



OO Refinement [FACSJ 2009]

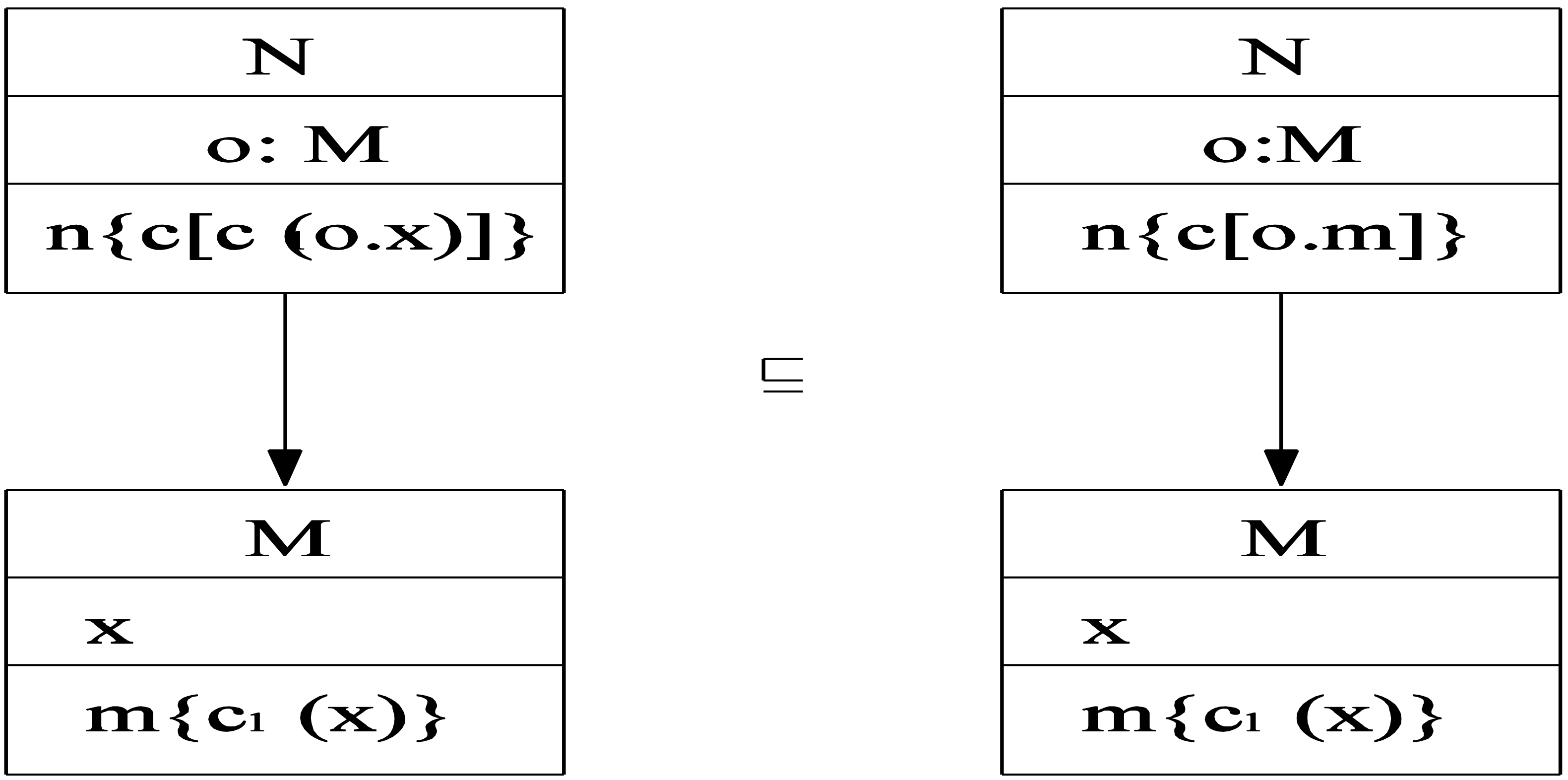
What does ClassDecls•c ⊑ ClassDecls1•c1 mean?

1. When ClassDecls = ClassDecls1, refinement defined as the same before
2. **CG ⊑ CG1 if the following diagram commutes**



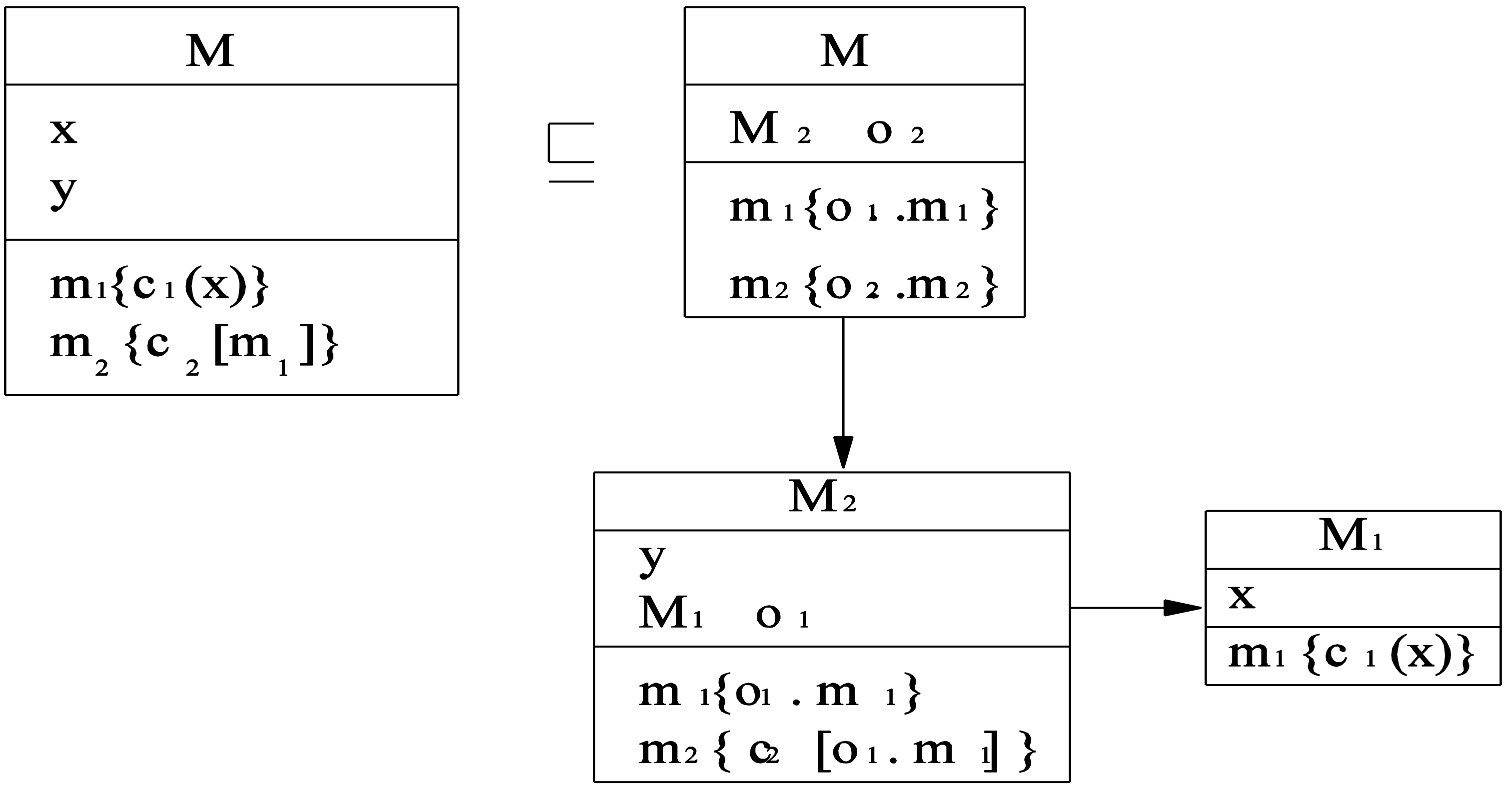
Design Patterns Refinement

Expert Pattern



Design Patterns Refinement

Class Decomposition (Low Coupling)



Concurrent Programs

- Concurrent program with shared variable
- Closed and execution of the actions is controlled by the program itself

```
Program P {
   variable T x = init_x
   action
    [a1: g1&D1 ⊓ … ⊓ an: gn&Dn]
  }
```

- TLA, Back's action systems, SAL
- Labeled state transition systems with failure-divergence semantics

Example

Memory-Processor Interaction systems

```
Program {
  var d: int, v: int; op: {rdy, r, w};
   init: op=w∧v∈int;
   Mw: op=w& (d:=v)   ∧(op:=rdy)
   Mr:   op=r &(v:=d)∧(op:=rdy)
   Pw: op=rdy & (v:=radom(int))∧(op:=w)
   Pr:   op=rdy & (op:=w)

   Act= Pw [] Pr [] Mw [] Mr
}
```

Reactive Designs

- Introduce Boolean observables wait and wait'
- A design D is reactive if $W(D)=D$, where

  $$W(D) \hat{=} \underline{\text{if}}\ wait\ \underline{\text{then}}\ wait'\ \underline{\text{else}}\ D$$

- Guarded design: $g\&D \hat{=} \underline{\text{if}}\ g\ \underline{\text{then}}\ D\ \underline{\text{else}}\ wait'$
- Properties
  1) $W(W(D))=W(D)$
  2) If D is reactive, so is $g\&D$
  3) $g\&(p \vdash R) = g\&(W(p \vdash R))$
  4) Domain of reactive design is closed under sequential programming
- Refinement $\sqsubseteq$ is defined as implication

Components as Reactive Programs

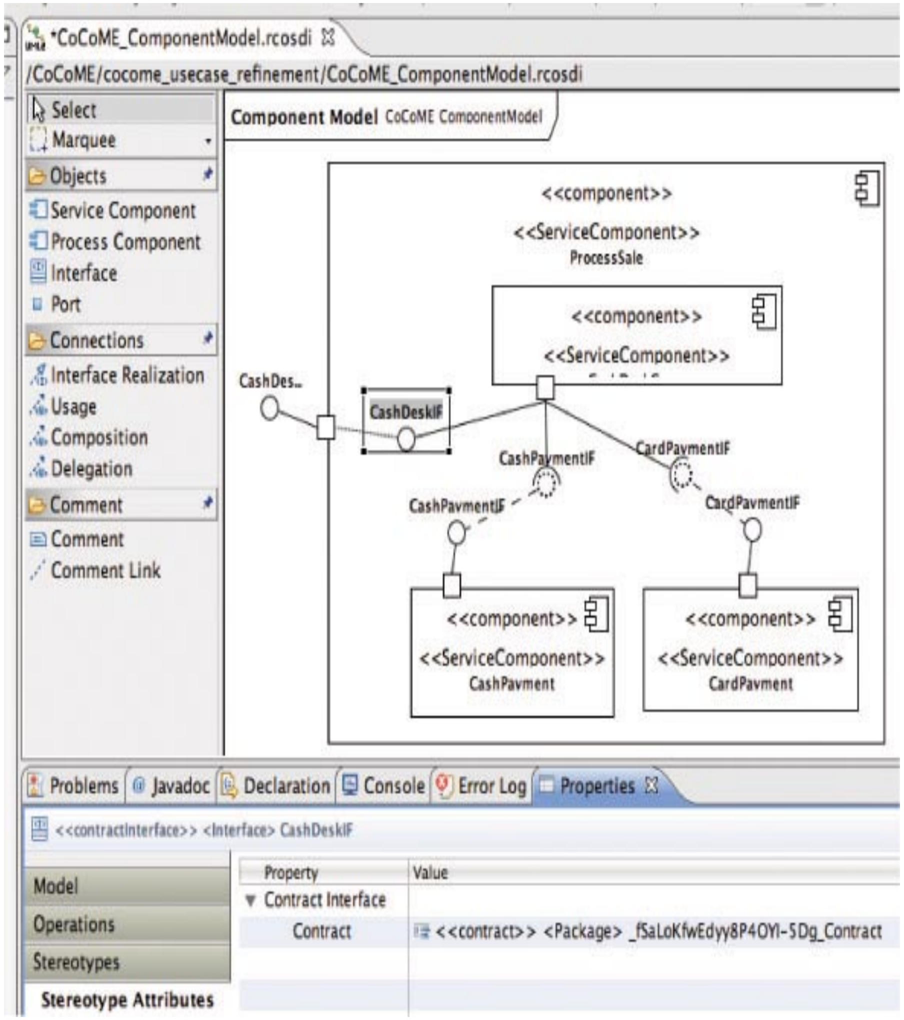
1. Component K=(V, Init, pIF, iA, rIF,Fd)
   - Fd(m())= g&p|-R – an I/o automaton + local data functionality
   - Fd(a)= g&p|-R
   - K=(V,Init, pIF, F, Prot), where Fd(m())= p|-R – local data functionality + sequence diagram (sequence charts)
   - Failure-Divergence Semantics: (Fail(K), Div(K))
2. Closed component K= (V,Init,pIF,iA,Fd)
   - local functionality + Interface Automata
   - Local functionally + sequence diagrams
   - Failure-Divergence Semantics
3. Open components and processes for composing and coordinating components

Semantics and Refinement of Components

1. Universal model of components, integrated from different system views [LNCS 0850]
2. Failure-Divergence Refinement with Upwards and Downwards Simulation [FSEN 2007]
3. Interface model: Non-blockable input automata, non-refusal input traces [ICTAC 2013]

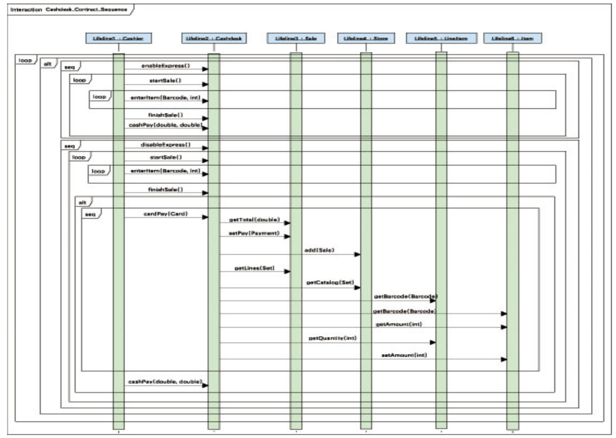
Component-Based Design with rCOS

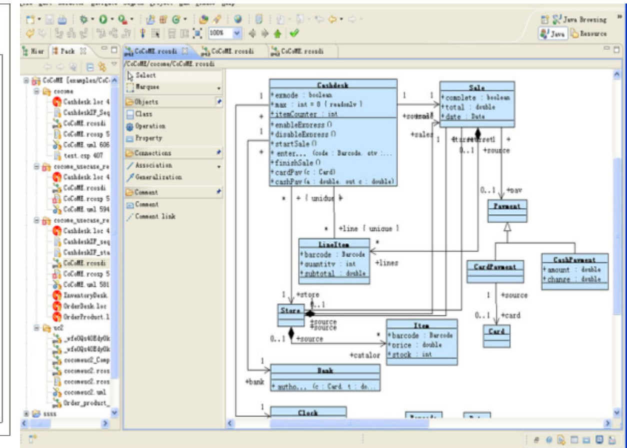
1. **Interface models allow:**
   - independent components design, development and deployment
   - use of components without the need to know their design and implementation
   - reuse of designs, proofs, and code
2. **Composition operators allow:**
   - coordination through **connectors** and **coordinators**
   - composability checking by reasoning about **functionality refinement** and **interaction compatibility**

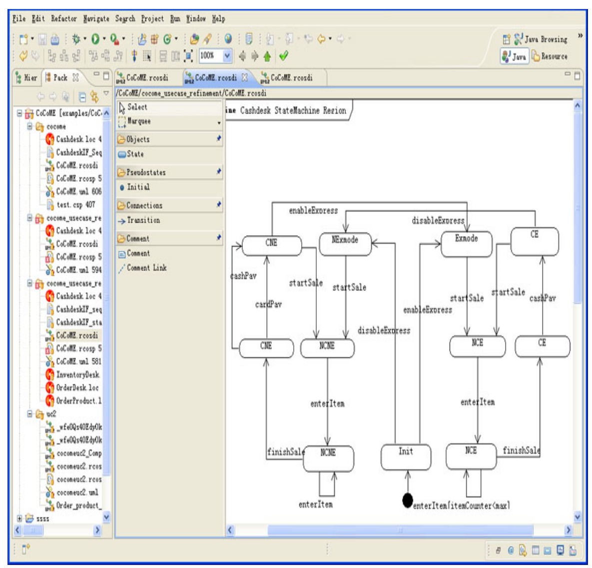


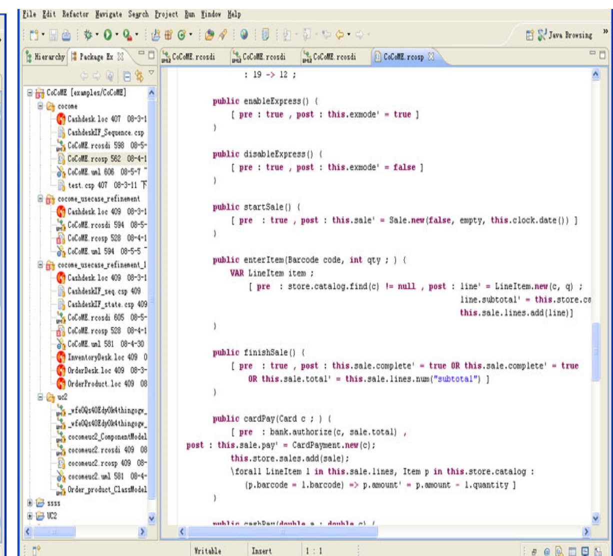
Model-Driven Development with rCOS

- Each phase is based on the construction of ***verifiable models***
- Models are analyzed and verified
- Refined models are constructed by ***model transformations***
- ***Code*** is generated from design models
- ***Proof obligations*** are generated by model transformations
- ***rCOS modeler*** integrates UML model notation into rCOS







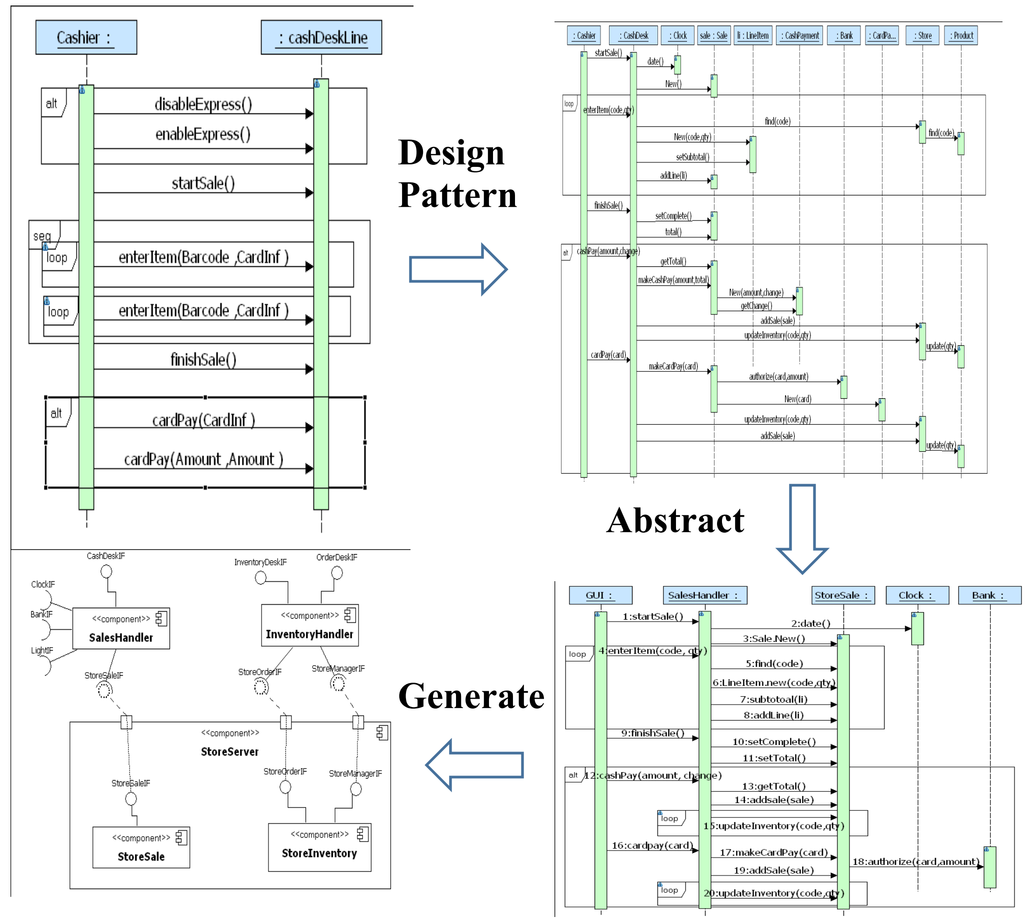


Model Transformations in rCOS Modeler

1. model a ***use case*** as a component
2. refine use case operations by ***design patterns*** to generate an ***oo interaction model***
3. generate design class model
4. transform the oo interaction model to a ***component interaction model***
5. generate the component diagram
6. transform oo interfaces to ***specific middlewares***, e.g. RMI, CORBA etc.
7. integrate use cases

Code generation performed after

3 and/or 6



Component-Orientation vs Object-Orientation [LNCS 7253]

- Classes and objects are not explicitly composable
- Most component-based technologies are implemented in OO languages
- Useful design patterns are mostly proposed for OO structures
- Static functionality decomposition is essentially characterized by Expert Pattern
- OO design model can be transformed into a component-based design model with interactive tool support.

OO is an important part of rCOS

Model Transformations vs Refinement Laws

- **Traditional refinement calculi provide syntactic rules for transforming specification**
  - Preserve semantic correctness
  - Support program derivation
  - Refinement laws are too fine grained and cannot be complete
  - Refinement of OO programs is not well developed
- **MTs preserve semantic correctness**
  - Design patterns and l model refactoring can be implemented as MTs with semantic conditions of the application
  - These conditions are generated as proof obligations by a transformation
  - Automation is crucial for MT to support code generation, and transformation between PSMs
  - Model transformations can be used to relate models of different users' views.

Further Aspects of rCOS

1. Service oriented systems: modeling and verification of web-services, choreography, orchestration and long running transactions [ICTAC 2010, FACS 2010]
2. Real-time [LNCS 5454, 2009]
3. Security: access control connectors [ISoLA 2008]
4. Aspect orientation: AspectJ as connectors

Future work

- Further tool support development (http://rcos.iist.unu.edu)
- Real-time, QoS, component-based fault-tolerant design
- Application development from a given repository of components
    - based on the knowledge of the repository
    - using a model of ontology of the components

- CPS: integration of cyber (software) components and physical components
- Applications in Healthcare and environmental health, in particular