

Component Based Programming

Zhiming Liu, UNU-IIST, Macao

<http://www.iist.unu.edu/~lzm>

Joint Work with He Jifeng, Xiaoshan Li and Anders P. Ravn

1. Motivation
2. rCOS Model of Components
3. rCOS Model of Processes
4. Component Based Programming
5. Conclusions

Motivation

- What are NOT components?
- What is **component-based programming** about?
- What are the benefits of component-based programming?

Two Informal Definitions

Component-based programming is about how to create *application software* (designs) by *existing modules* with *new software* that provides both *glue* between the components, and *new functionality*.

— L. de Alfaro and T.A. Henzinger

A **component** is a unit of composition with **contractually specified interfaces** and fully explicit context dependencies that can be **deployed independently** and is subject to **third party composition**.

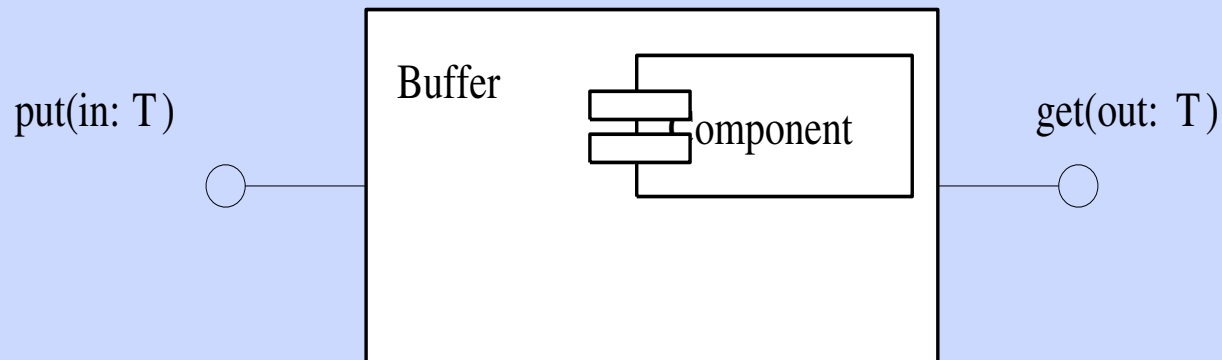
— Szyperski

rCOS Model of Components

- **Interfaces**: operation signatures for syntactic compositionality
- **Contracts**: interface specifications including the static behavior, the dynamic behavior and interaction protocol, timing ...; refinement
- **Components**: Provided and required interface + code
- **Object rCOS**: provides a common semantics for different PLs to implement components (**interoperability**)
- **Semantics of Components**: relation between components and contracts (correctness), substitutability
- **Composition Operations**: simple connectors
- **Component-Based Programming**: **glue** and **application processes**

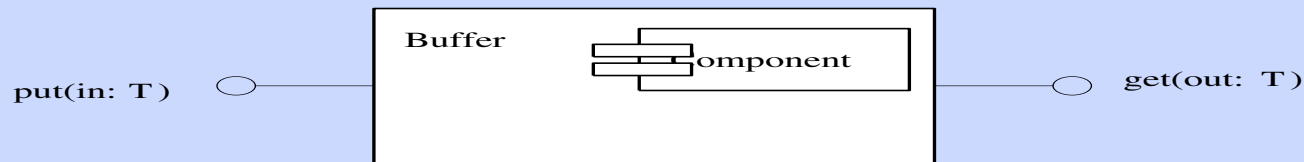
Interfaces and their Contracts

- An *interface* of a component is a description of what is needed for the component to be *used* in building and maintaining software **without the need to know the code of the component**.
- It is the interface that determines the **external features** of the component and allows the component to be used as a **black box**.
- The interfaces determine the **substitutability** of the components



Contract of Interfaces Has Multi-Views

What should be in an interface depends on the “use” of the component in different applications and environments



Static behavior: [Pre, Post]

Real-time lb, ub

Protocol: ($\{ \text{put}(x)\text{get}(x): x \text{ in } T \}^*$): 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

Separation of Concerns and Incremental Development are the keys for simplicity and dealing complexity!

Integration of Methods and Tools: type checking, static analysis, event-based protocol analysis, State Machine-Based safety and liveness analysis

Contracts

A *contract* is a tuple $Ctr = (I, Init, MSpec, Prot)$

- $MSpec$ assigns each operation to a guarded design $g \& D$.
- $Prot$ is called the *protocol* and is a set of sequences of call events; each is of the form $?op_1(x_1), \dots, ?op_k(x_k)$

Example : One-place buffer B_1

- $B_1.I = \langle q : Seq(int), put(item : int;), get(; res) \rangle$
- $B_1.Init = q = \langle \rangle$
- $B_1.MSpec(put) = q = \langle \rangle \& true \vdash q' = \langle item \rangle$,
 $B_1.MSpec(get) = q \neq \langle \rangle \& true \vdash res' = head(q) \wedge q' = \langle \rangle$
- $B_1.Prot \dots$

Consistency

A contract Ctr is *consistent*, if it will never enter a deadlock state if its environment interacts with it according to the protocol:

For all $\langle ?op_1(x_1), \dots, ?op_k(x_k) \rangle \in Prot$,

$$\mathbf{wp} \left(\begin{array}{l} Init; g_1 \& D_1[x_1/in_1]; \dots; g_k \& D_k[x_k/in_k], \\ \neg wait \wedge \exists op \in MDec \bullet g(op) \end{array} \right) = true$$

Refinement of Contracts

The notion of *refinement* contracts is about preserving the correctness criteria of views described in the contract.

- it defines notion of **black-box** *substitutability* of a component
- A component is an implementation of a contract, and in the implementation it can *use* existing components too.
- substitutability in CBSE must be compositional so component system's **predicability** can be derived from that of the components (for both functional properties and Nonfunctional QoS)
- **Reuse of components and proofs**

Component Compositions

Define two components C_1 and C_2 as follows

$$\begin{aligned}
 C_1.FDec &= \{x_1 : Seq(int)\} \\
 C_1.MDec &= \{put(in : int;), get_1(; out : int)\} \\
 C_1.Code(put) &= (x_1 := \langle in \rangle \triangleleft (x_1 = \langle \rangle) \triangleright \\
 &\quad (put_1(head(x_1);); x_1 := \langle in \rangle) \text{ when } C_1 \text{ is full, forwards it to } C_2 \\
 C_1.Code(get_1) &= (x \neq \langle \rangle) \longrightarrow (out := head(x_1); x_1 = \langle \rangle) \\
 C_1.InMDec &= \{put_1(in : int;)\} \\
 C_2.FDec &= \{x_2 : Seq(int)\} \\
 C_2.MDec &= \{put_1(in : int;), get(; out : int)\} \\
 C_2.Code(put)_1 &= (x_2 = \langle \rangle) \longrightarrow x_2 := \langle in \rangle \\
 C_2.Code(get) &= (out := head(x_2); x_2 := \langle \rangle) \triangleleft (x_2 \neq \langle \rangle) \triangleright \\
 &\quad get_1(; out) \text{ when } C_2 \text{ is empty, gets } C_1 \\
 C_2.InMDec &= \{get_1(in : int;)\}
 \end{aligned}$$

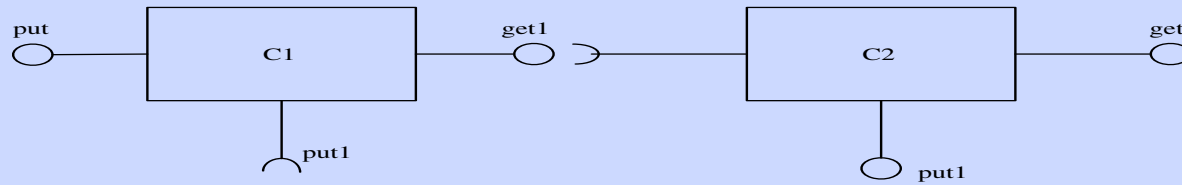


Figure 1: Chaining Composition

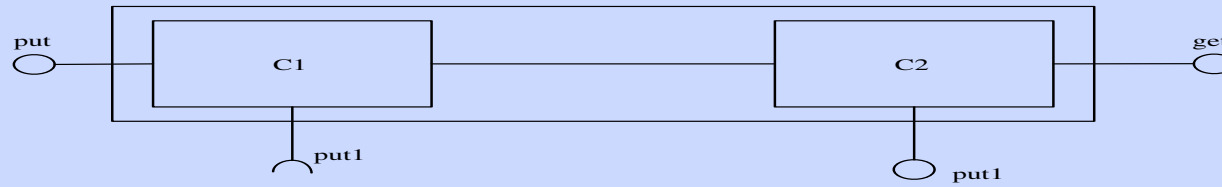


Figure 2: Hiding after Chaining

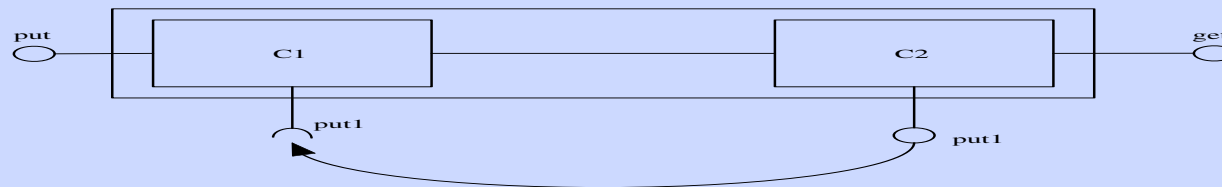
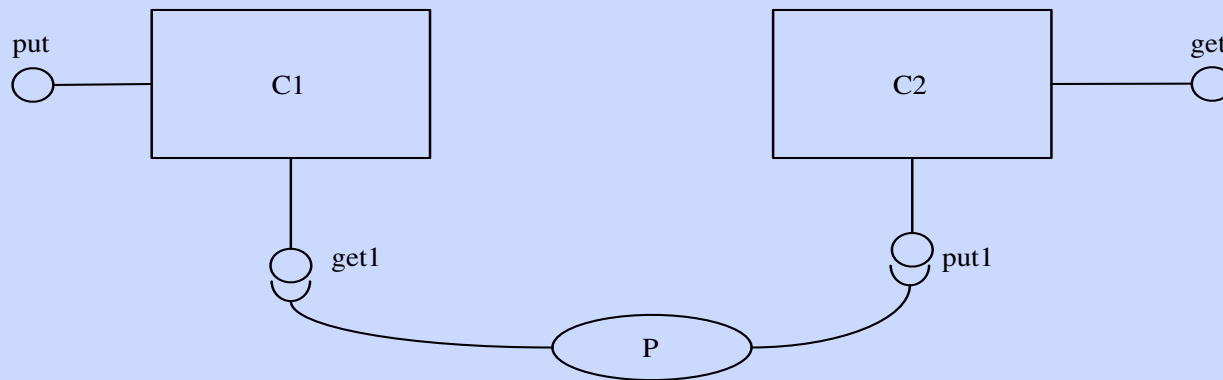


Figure 3: Feedback Composition

Glue Components

- A glue program is modelled as a *process* that is an **active entity that only calls methods of components**
- The glue of C by P is realized as *synchronous composition* $C|||P$



$P.get_1(; out) :: (x = 0) \&true \vdash x' = 1 \wedge v' = out'$

$P.put_1(in;) :: (x = 1) \wedge v = in \&trye \vdash x' = 0 \wedge v' = v$

What if change get_1 to $(x = 0) \&true \vdash x' = 1 \wedge v' = out' + a$?

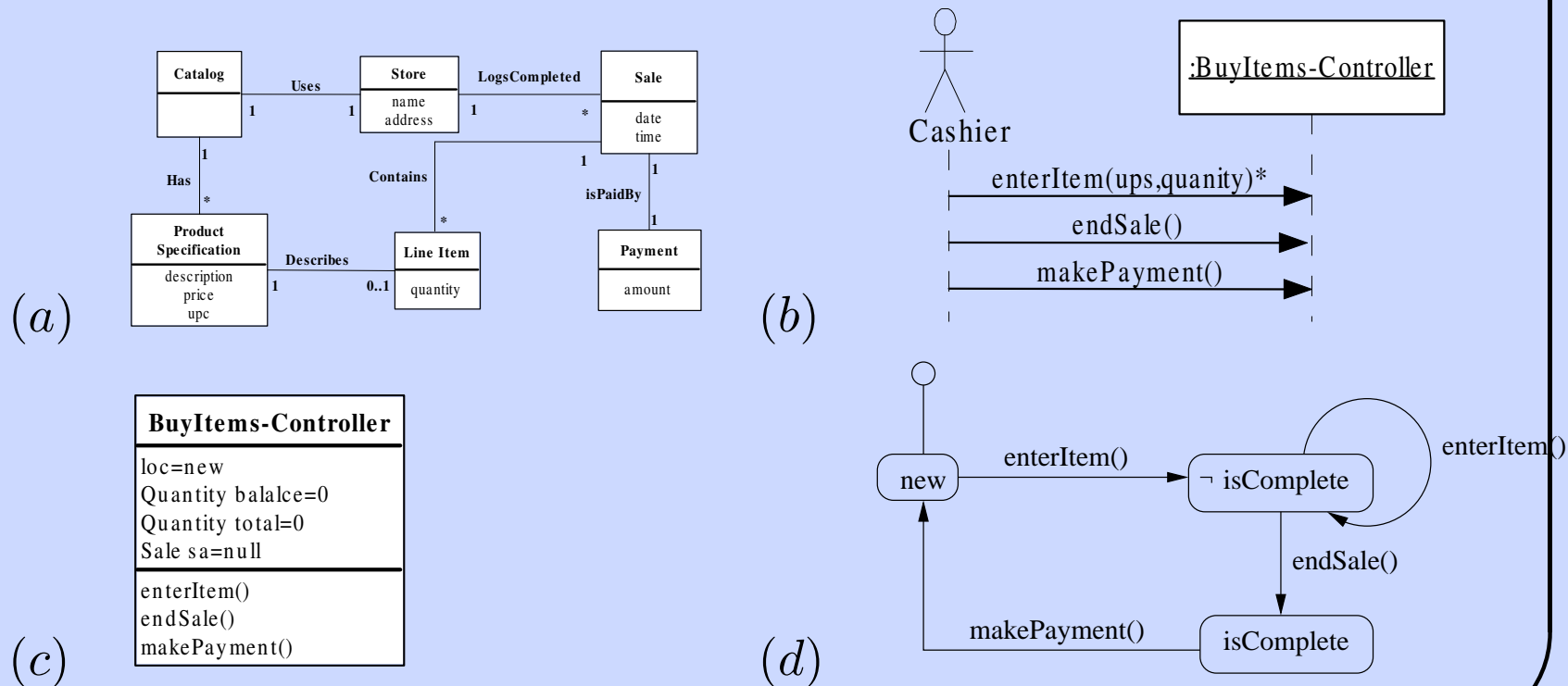
What is the capacity of the glued buffer?

Component-Based Programming

1. A set of components, pre-developed or newly developed
2. Build new components by component operators (connectors)
3. Build new components by programming glue processes
4. Application tasks are programmed as processes (threads) that keep calling services from components according to a flow of control.
5. Application tasks are also modeled as processes
6. **Interoperability is ensured because processes only interact with components via interfaces method invocation, without the need to know the implementation of the component**
7. **Reuse of components and proofs about components can is also supported:** any property proved about a components can be used as a lemma in proving properties of a component or a program that uses that component.

rCOS: Models of Application Requirements

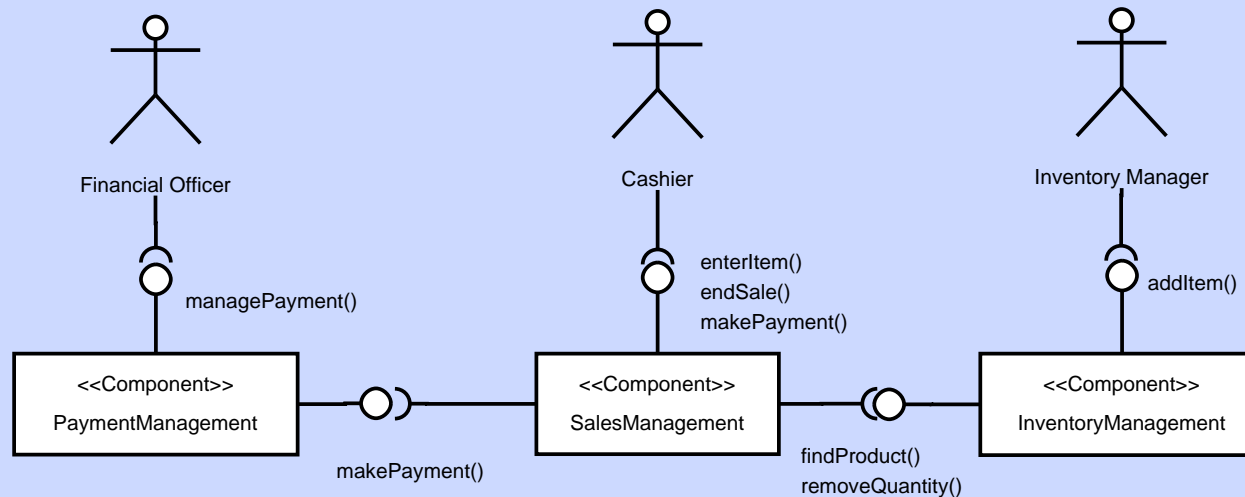
- Conceptual class model
- use-case model – **static functionality + interaction and dynamic behavior**
- consistency and integration defined



rCOS: Requirements Analysis, Verification and Reasoning

- Syntactic type checking (consistency between data/class model and functional model)
- Model checking, such as SAL, for invariant properties and liveness properties
- Vertical refinement to add functionality without violating proved properties
- Refine static functionalities by proved laws and theorem prover
- Prototyping for requirement validation requirement

rCOS: Component Architecture Design



- Assign classes and associations to components according to the use cases – **partition the state space**.
- Decomposing use case sequence diagrams into component sequence diagrams – **define interaction protocols of components**.
- Verifying the decomposition against the application requirements model.

Generate probably correct code

```
class A {          | class C-handler{  | class B1{
  B refB1;         |   C element;     |   B2 refB2;
  C refC;          |   Vector<C> vector; |   meth_1(){
  meth_0(){        |   }               |   refB2.meth_9();
    refB1.meth_1(); |                   |   }
  command_3;      |                   | }
  (g)* {          | class C{          |
    refC.meth_4(); |   A refA;         |
  }               |   meth_4(){       | class B2{
  }               |     refA.meth_5(); |   meth_9(){
  meth_5(){}      |   }               |     SD_11();
}                 |                   | }
                 |                   | }
```

Also, Assertions and invariants can be carried forward and generated by transformations – Spec#-like program

Real-Time

- *Real-time contracts*: $g \& D : [l, u, d]$
- *Real-time Object rCOS* for construction and reasoning about real-time components with real-time contracts
- *Real-time process* for **gluing** real-time components, and for modelling **real-time application tasks**: time of their local computation and time constraints depending on components
- *Scheduling* **real-time application tasks**
- General **QoS** – designs with relations about resources

Fault-Tolerance, Security, Web Systems

- Using **glue processes**, such as checkpointing and recovery processes, to tolerate faulty components?
- Design **components**, such as voting, to tolerate faults in existing components
- Use **redundant components and coordinating (glue) processes** to mask faults in single components
- Using components for access checking, encoding interfaces and decoding interfaces, and even coding and decoding functionality and protocols (?) by adding different key management components
- *Orchestration* and *choreography* are the most important application process for web service.

Conclusion

- If the results of the research in formal theories are ever to be generally applied by software engineers, our methods and techniques and tools will have to be **incorporated into existing program development environments**.
- Different models and tools for formal verification and analysis must be **integrated via a *common semantics***.
- rCOS can be such a common semantics
- We need to implement the ideas to show they are possible and the way to do it.