and Architectures

to Components

From Classes

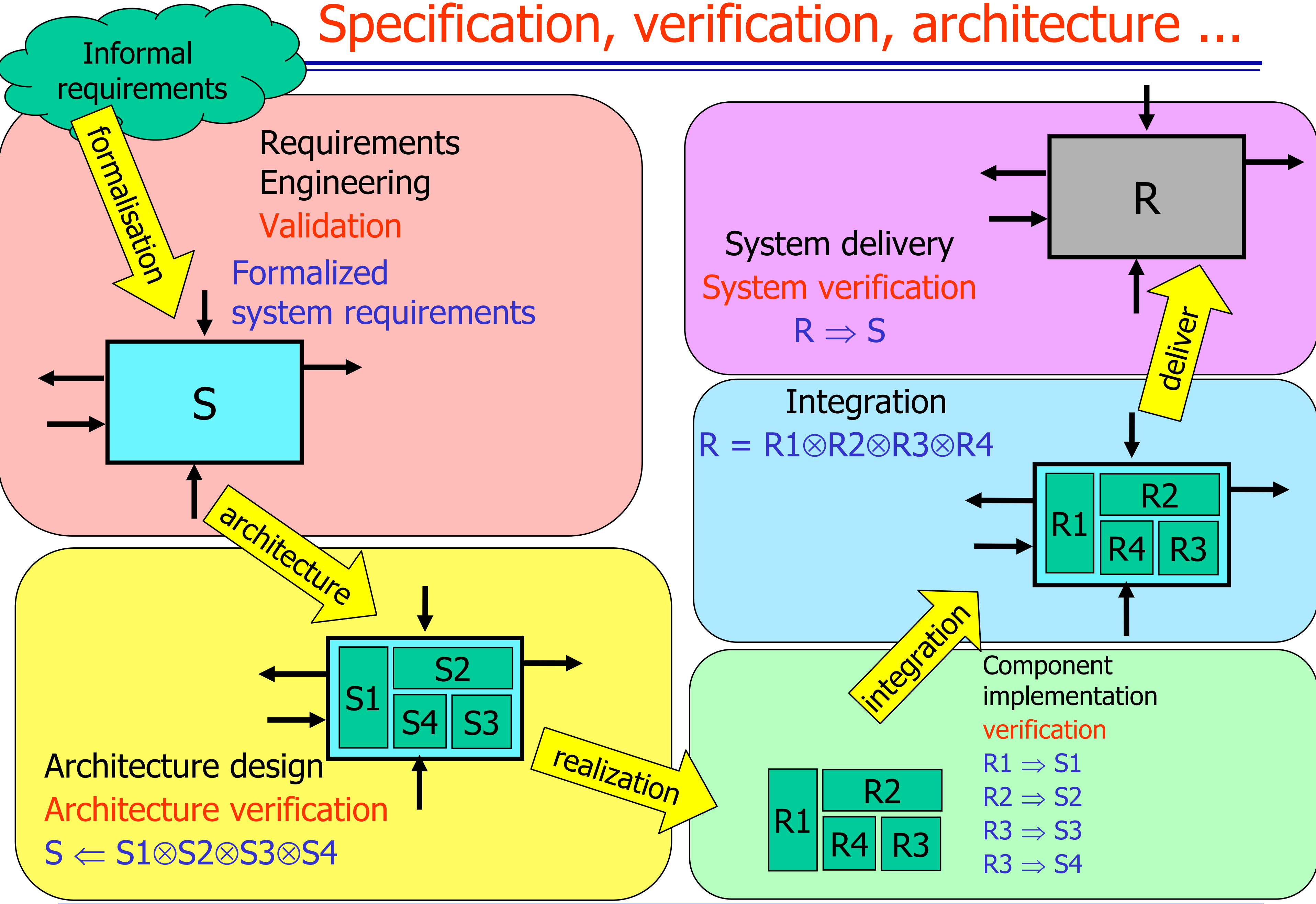
Import, OO Components

A formal framework for modular specification and verification of components and architectures

Manfred Broy

Technische Universität München
Institut für Informatik
D-80290 München, Germany

Specification, verification, architecture ...



Object-oriented Components and Interfaces

Classes as a components

- Needed concepts
  - ◇ Observations
  - ◇ Component: Class / Set of Classes
  - ◇ Composition of classes
  - ◇ Interface specification
- Specification by
  - ◇ Contract
  - ◇ State machines

Observations

In an OO software system

- which consists of a set of classes where
  - all sub-method calls are targeted to methods that are part of the system (this is the characterization of a system in contrast to a "component" that may rely on methods from the outside)

we may

- invoke methods (stimulus)

and observe

- values that the system returns (reactions)

  provided that the method call terminates
- Then the execution of a method invocation can be modelled as one large state transition (we call this the closed view)

Data types

- A type is either a *constant* type or a *variable* type.
- Constant types are basically sets of data values or class types (being names of classes used as types of the objects of that class).
- An identifier with constant type denotes a value of that set.
- A variable type is denoted by **Var** T where T is a constant type.
    - ◇ An identifier with variable type denotes a variable (an attribute) that has assigned a value out of the set of elements of type T.
- Every class name defines a type, the elements of which are object identifiers

Method header

- To keep our notation simple we consider only methods with one constant parameter w and one variable parameter v; headers read

  Method m (w : WT, v : Var VT)

  where WT and VT are constant types.

- The set of method invocations INVOC(m) for the method m is defined by:

  $$\text{INVOC}(m) = \{m(b_1,b_2,w,v,v'): w \in WT,\ v, v' \in VT,\ b_1, b_2 \in \text{Object}\}$$

  where phrase $p \in T$ expresses that p is a value of type T and $m(b_1, b_2, w, v, v')$ denotes a tuple of values.

- Here
  - ◇ b_1 denotes the caller and b_2 the callee,
  - ◇ v denotes the value of the variable parameter before and
  - ◇ v′ its value after the end of the execution of the method invocation.

Specification by Contract: States and their Attributes

- The states of the objects of a class are determined by the valuations of the attributes of that class.
- An attribute is a typed identifier.
- An attribute set V is a set of the form

  $$V = \{a_1 : T_1, \ldots , a_n : T_n\}$$

  where $a_1, \ldots , a_n$ are (distinct) identifiers and $T_1, \ldots , T_n$ are their types.
- A valuation of the attribute set V is a mapping

  $$\sigma: V \to UD$$

  where UD is universe of data values.

Specification by contract for a Method

- Let V = {a : Var AT} be an attribute set.
- A *specification by contract* for a method with header

  method m (w : WT, v : Var VT)

  in a class with attribute set V is given by

  method m (w : WT, v : Var VT)
  pre P(w, v, a)
  post Q(w, v, a, v′, a′)
- Here P(w, v, a) and Q(w, v, a, v′, a′) denote predicates
  - ◇ v, a denote the values before and v′, a′ the values of the variables after the method invocation
- Two options: P guarantees termination or not

Example. Specification by Contract (SbC)

- We consider only one method here and assume only one attribute

  u : **Var** Seq Data

- Specification by contract for a method that gets access (“reads”) the ith element of sequence u:

  Method get (i : Nat, r : **Var** Data);

  pre $1 \leq i \leq \text{length}(u)$

  post $r' = \text{ith}(i, u) \wedge u' = u$

  Here we assume that the functions
  - ◇ length(s) (yielding the length of sequence s) and
  - ◇ ith(i, s) (yielding the i-th element of sequence s) are predefined for sequences, for instance, by an algebraic data

Specification of the data elements

SPEC SEQ =

{	**based_on** BOOL,		
	type Seq α,		
$\langle\rangle$:	Seq α,		empty sequence
$\langle\_\rangle$:	$\alpha \to$ Seq α,	Mixfix	*one-element sequence*
$\circ$:	Seq α, Seq $\alpha \to$ Seq α,	Infix	*concatenation*
iseseq:	Seq $\alpha \to$ Bool,		
first, last:	Seq $\alpha \to \alpha$,		
head, rest:	Seq $\alpha \to$ Seq α,		
index:	α, Seq $\alpha \to$ Nat,		
length:	Seq $\alpha \to$ Nat,		
ith:	Nat, Seq $\alpha \to \alpha$,		
drop:	α, Seq $\alpha \to$ Seq α,		
cut:	Seq α, Nat, Nat $\to$ Seq α		

Axioms

Seq α **generated_by** $\langle\rangle$, $\langle\_\rangle$, $\circ$,

$\text{iseseq}(\langle\rangle) = \text{true}$,

$\text{iseseq}(\langle a\rangle) = \text{false}$,

$\text{iseseq}(x \circ y) = \text{and}(\text{iseseq}(x), \text{iseseq}(y))$,

$\text{length}(\langle\rangle) = 0$,

$\text{length}(\langle a\rangle) = 1$,

$\text{length}(x \circ y) = \text{length}(x) + \text{length}(y)$,

$\text{ith}(1, \langle a\rangle \circ y) = a$,

$\text{ith}(n+1, \langle a\rangle \circ y) = \text{ith}(n, y)$,

$\text{index}(a, \langle\rangle) = 0$,

$\text{index}(a, \langle a\rangle) = 1$,

$a \neq b \Rightarrow \text{index}(a, \langle b\rangle \circ x) =$ **if** $\text{index}(a, x) = 0$ **then** 0 **else** $1 + \text{index}(a,x)$ **fi**

Axioms

$\mathrm{drop}(a, \langle a\rangle \circ x) = x,$
$a \neq b \Rightarrow \mathrm{drop}(a, \langle b\rangle \circ x) = \langle b\rangle \circ \mathrm{drop}(a, x),$

$\mathrm{cut}(s, i, 0) = \langle\rangle$
$\mathrm{cut}(s, 0, j+1) = \langle \mathrm{first}(s)\rangle \circ \mathrm{cut}(\mathrm{rest}(s), 0, j)),$
$\mathrm{cut}(s, i+1, j+1) = \mathrm{cut}(\mathrm{rest}(s), i, j),$

$x \circ \langle\rangle = x = \langle\rangle \circ x,$
$(x \circ y) \circ z = x \circ (y \circ z),$

$\mathrm{first}(\langle a\rangle \circ x) = a,$

$\mathrm{last}(x \circ \langle a\rangle) = a,$

$\mathrm{head}(\langle a\rangle \circ x) = \langle a\rangle,$

$\mathrm{rest}(\langle a\rangle \circ x) = x$

}

Simple Export Interfaces

A *syntactic export interface* consists of

a set types being classes (names) and

for each class a set M of method headers.

Specification by contract of classes

For a *syntactic export interface* consisting of

- a set of method headers and a set of class names
- a set of typed attributes defining the class state space and

a specification by contract is given by

- a specification by contract for each of its methods.
- initial assertions:

  initial P(a)

  expressing that initially the assertion holds
- In addition, state transition assertions R(a , a’) and invariants Q(a) may be given restricting the state changes for all methods.
- It is good to make invariants explicit, but there may be implicit invariants

Export Interfaces described by State Machines

- Given an interface with
  - ◇ an attribute set V and
  - ◇ a set of methods M

  the associated state transition function has the form

  $$\Delta: \Sigma(V) \times \text{INVOC}(M) \to (\Sigma(V) \cup \{\perp\})$$

- For m $\in$ INVOC(M) and s, s′ $\in \Sigma$(V) the equation
  - $\Delta(s, m) = s'$
- expresses that in state s method invocation m is enabled and leads to state s
- If
  - $\Delta(s, m) = \perp$
- this means that the method invocation m is not enabled in state s or that the method invocation does not terminate.
- In addition, we assume a set of initial states $I\Sigma \subseteq \Sigma(V)$.

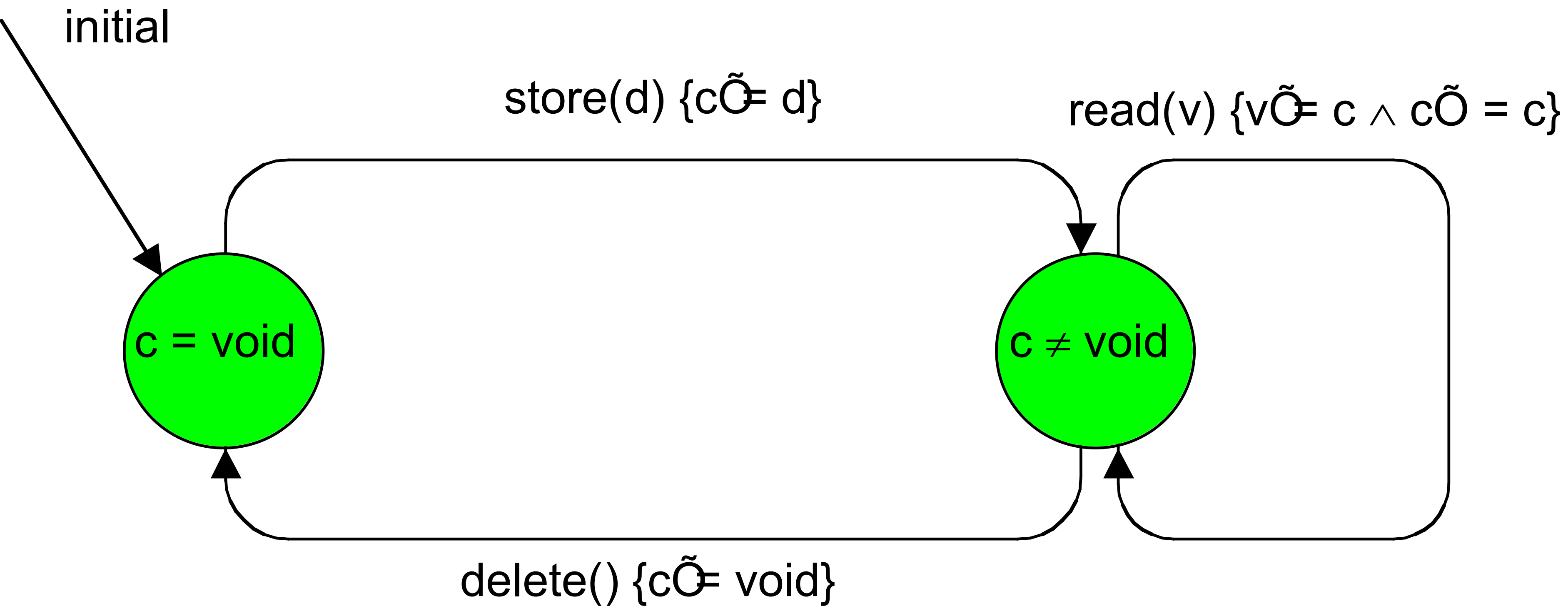
Example. Memory Cell

```
class Cell =
        { c: Var Data | {void}

          initial c = void

          method store (d: Data)
                  pre     $c = void$
                  post    $c' = d$

          method read (v: Var Data)
                  pre     $c \neq void$
                  post    $c' = c \wedge v' = c$

          method delete ()
                  pre     $c \neq void$
                  post    $c' = void$
        }
```

Memory cell as a labelled state machine

Labelled state machines:

$$\Delta: \Sigma(V) \times \mathrm{INVOC}(M) \rightarrow \Sigma(V) \cup \{\perp\})$$



Forwarded calls

A method invocation may lead to a further method invocation; we speak of a

forwarded method call

Example. Account manager

We consider following three types:

Person	the type of individuals that may own accounts
Account	the type of accounts (a class)
Amount	the type of numbers representing amounts of money

For the class Accountmanager we consider only one method.
It uses a function f

Fct f = (x: Person) Account: …

that relates persons to their account numbers.

Class Accountmanager =
{…
    method credit = (x: Person, y: **Var** Amount, z: **Var** Account)
…
}
The method credit calls a method
    method balance = (y: **Var** Amount)

Example. Account manager

```
Class Accountmanager
{   Fct f = (x : Person) Account:…
     method  credit = (x : Person, y : Var Amount, z : Var Account):
           f(x).balance(y); z:= f(x)
}
Class Account
{   a, d : Var Nat;   {a denotes the state of the account, d what is bound
    by credit}

    invariant a ≥ d;

    method balance = (y : Var Amount)
    if      a-d ≥ y   then d := d+y
    else if a = d     then y := 0
                      else y := a-d; d := a
    fi fi
}
```

Specification by contract

In this example a call of the method credit

- ◇ leads to a call of the method balance,
- ◇ which may change the attribute d.

The specification by contract for credit reads as follows:

method credit = (x : Person, y : **Var** Amount, z : **Var** Account):

pre $f(x) \neq nil$

post $z' = f(x)$

$\wedge\ f(x).d' = f(x).d+y'$

$\wedge\ (f(x).a\text{-}f(x).d \geq y \Rightarrow y' = y)$

$\wedge\ (f(x).a\text{-}f(x).d \leq y \Rightarrow y' = f(x).a\text{-}f(x).d)$

- This shows that we have to refer to attributes of the object f(x) in the method credit.
- Here we use the notation b.a to refer to attribute a in the of the object b.

Example. Account manager (continued)

```
Class Account
{   a, d : Var Nat;

    invariant a ≥ d;

    method balance = (y : Var Amount)
    if      a-d ≥ y    then d := d+y
    else if a = d      then y := 0
                       else y := a-d; d := a
    fi fi
}
```

Replacement: d by e = a-d

```
Class Account'
{   a, e : Var Nat;

    invariant a ≥ e;

    method balance = (y : Var Amount)
    if      e ≥ y      then e := e-y
    else if e = 0      then y := 0
                       else y := e; e := 0
    fi fi
}
```

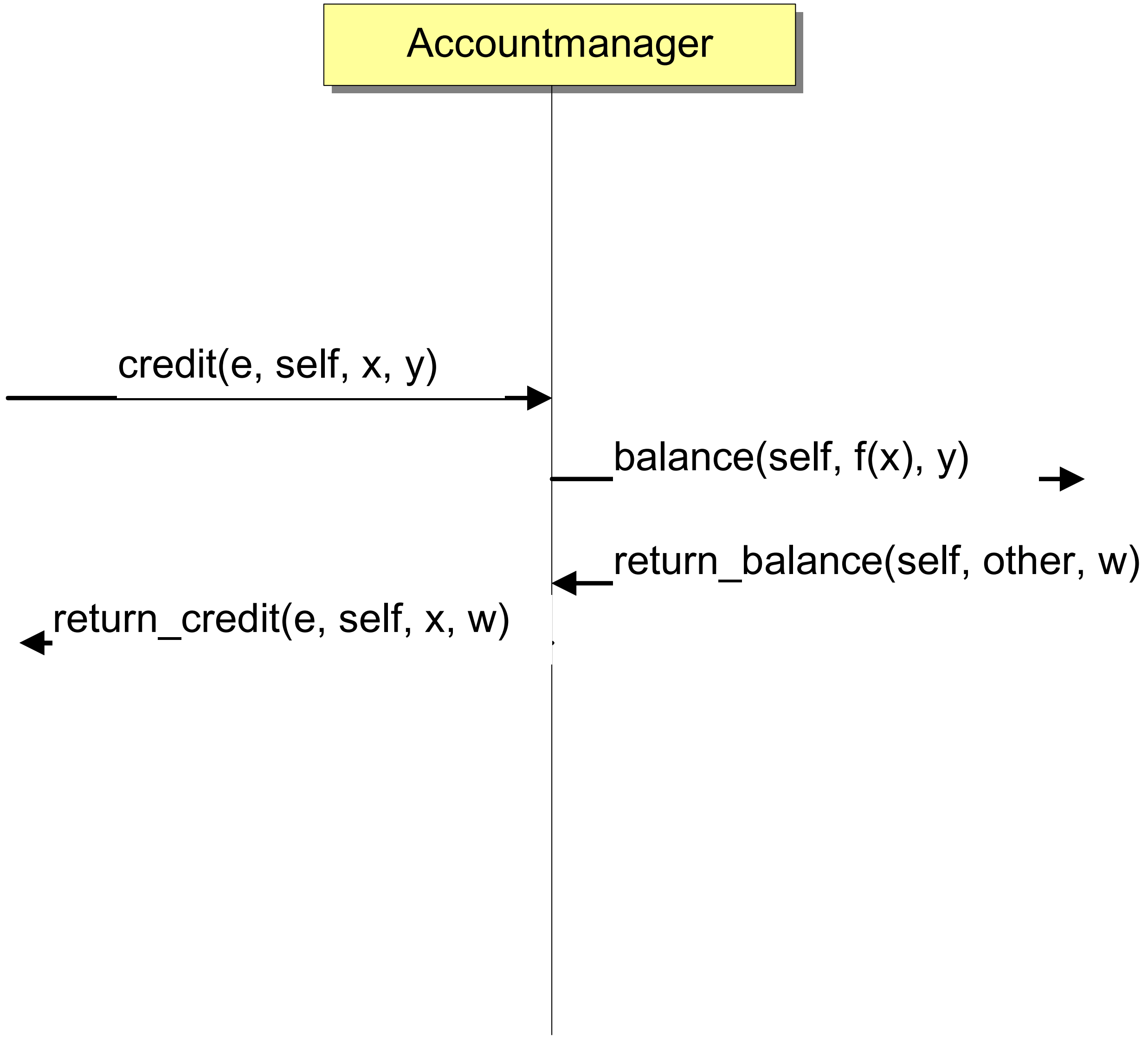
The classes Account and Account' are observable equivalent, but use different local attributes and thus cannot be replaced by each other in the context of SbC.

Forwarded Calls, Back-Calls, and Call Stack

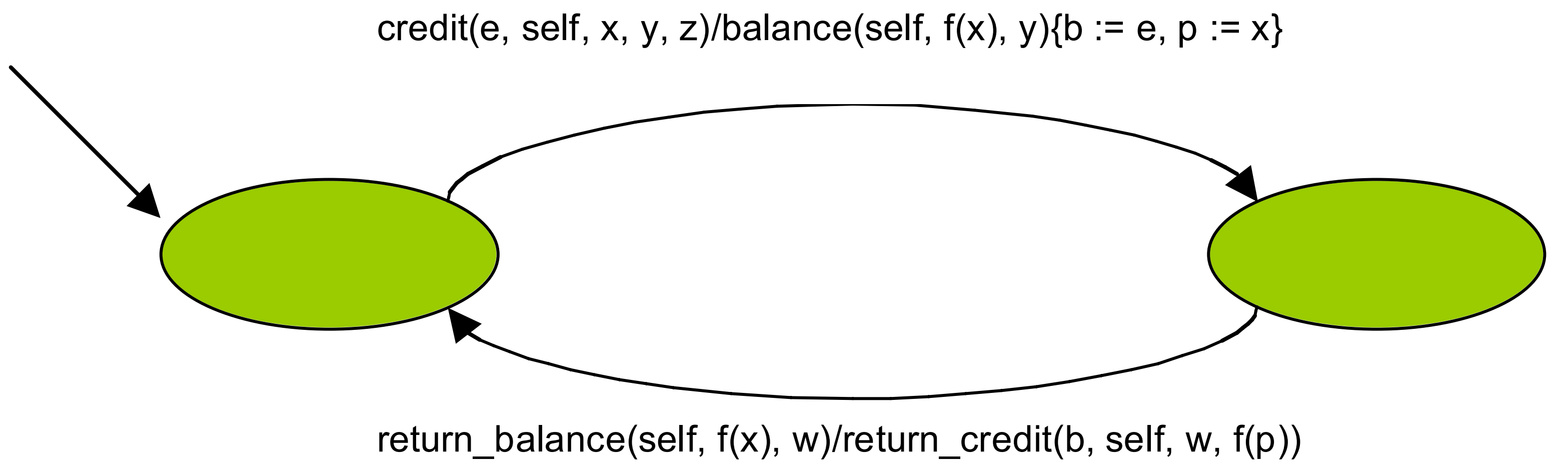
- When dealing with forwarded calls there may be call-backs, in general.
  - ◇ a method invocation for object b may lead to a forwarded call that in turn may lead to invocation of methods of object b.

  We speak of a call-back.

Account manager (continued)



Example: Account manager (continued): Call forwarding


credit(e, self, x, y, z)/balance(self, f(x), y){b := e, p := x}
return_balance(self, f(x), w)/return_credit(b, self, w, f(p))

Here we split each method invocation in two messages:

- The invocation message
- The return message

This models asynchronous method calls

Note that

- the state machine requires additional attributes that are not the attributes that we use in the class Accountmanager such as

  b: **Var** Object

  p: **Var** Person

Why simple (export only) classes are not enough

Conventional OO has the following deficiencies:

- Synchronous method invocation inadequate concept for large distributed software
  - Modelling of forwarded method calls of methods outside the considered system part
  - for system with varying availability and QoS
  - Inherently sequential
- Interface specifications insufficient
  - Design by contract breaks principle of encapsulation
  - Forwarded calls and call backs need to make stack discipline explicit
- Appropriate notion of component missing
- Concept of composition missing/unclear/too complicated
- No support of hierarchical composition/decomposition
- No build-in concept of real time/concurrency

A way out: Export/Import interfaces

Open View: Components with Export and Import

- We treat methods that can be called in forwarded method calls to the outside of the considered subsystem explicit:
- We use export and import in specifications and classes
- The imported methods are thus that are used in forwarded method calls to the outside

This leads

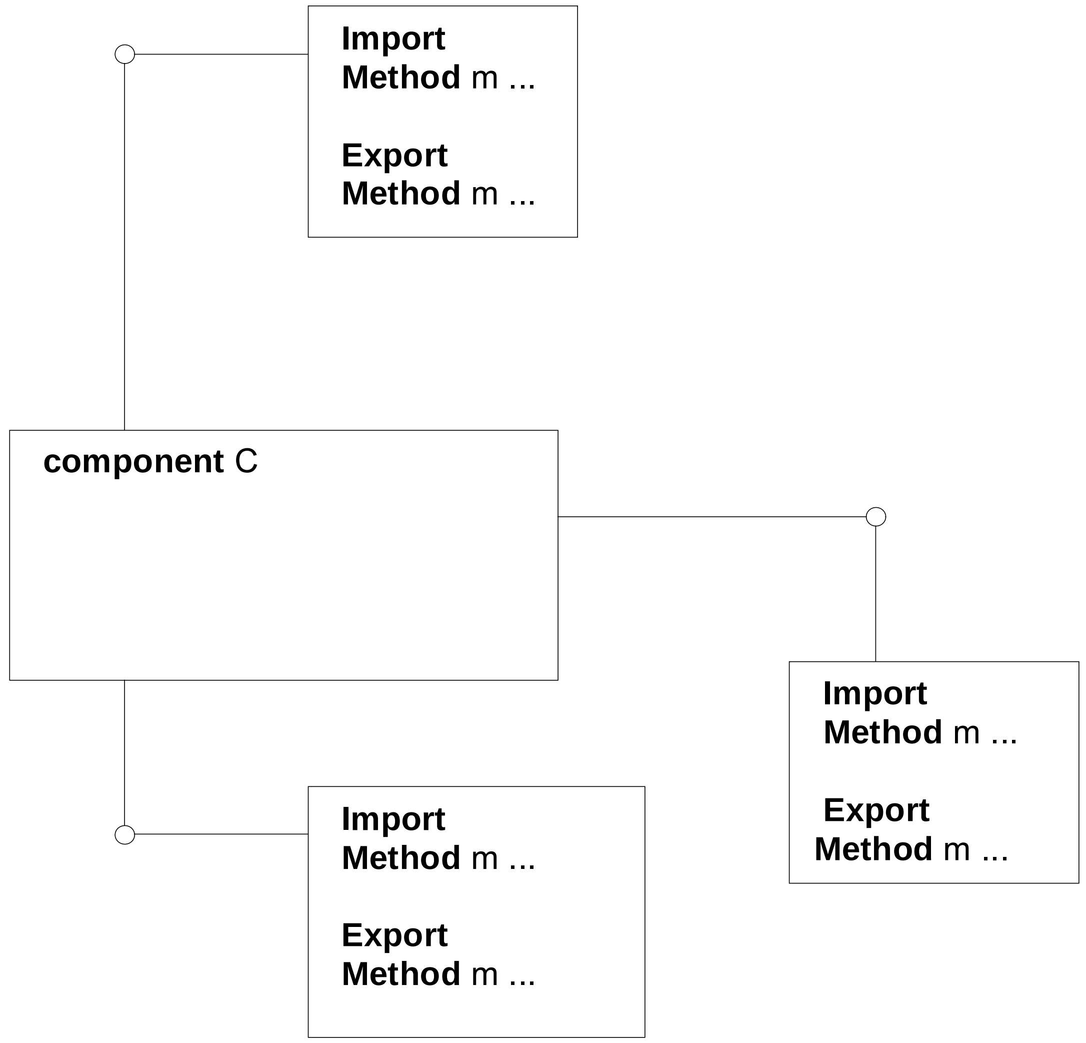
- to what we call an open view onto sets of classes

Syntax of export/import interface

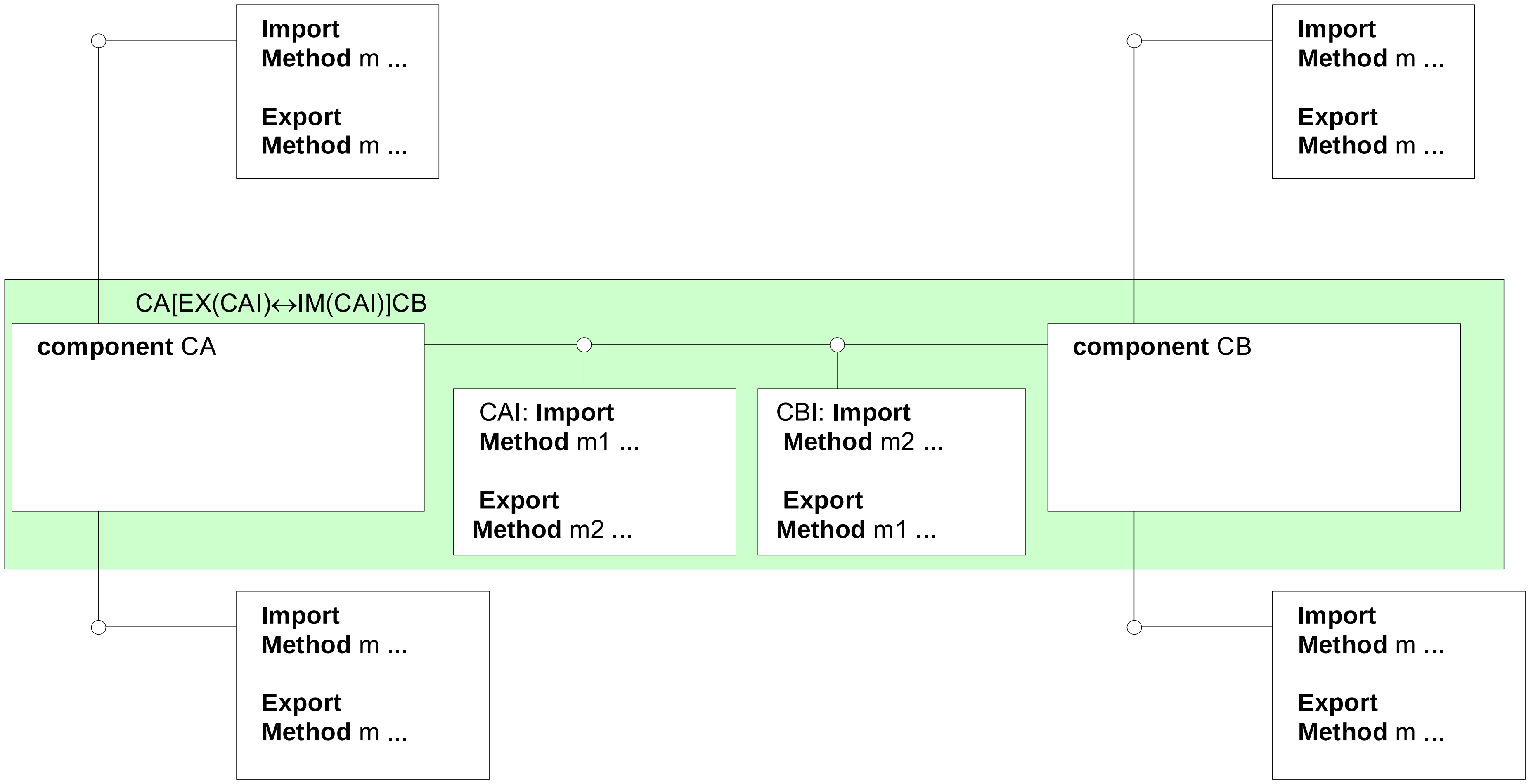
A syntactic export/import interface consists of

- two syntactic interfaces represented by
  - ◇ two sets of class names,
  - ◇ sets of method headers associated with each class name, which define the set of export and the set of import methods.
- Methods in the set of export methods can be called from the environment,
- Methods in the set of import methods are provided by the environment and can be called by the subsystem.

Components in OO with Multiple Sub-Interfaces



Composition



Design By Contract: Example. Account manager (ctd)

```
Class Accountmanager
{  Fct f = (x : Person) Account: …
{export
method  credit = (x: Person, y: Var Amount, z: Var Account):
        pre     $f(x) \neq nil$
        post    $z' = f(x)$
        $\wedge$       $f(x).d' = f(x).d+y'$
        $\wedge$       $(f(x).a-f(x).d \geq y \Rightarrow y' = y)$
        $\wedge$       $(f(x).a-f(x).d \leq y \Rightarrow y' = f(x).a-f(x).d)$
        body    f(x).balance(y); z:= f(x)
}
```

Import part

```
import
{  a, d : Var Nat;
   invariant a ≥ d;

   method balance = (y : Var Amount):
        pre    true
        post   d′ = d+y′
        ∧      (a-d ≥ y ⇒ y′ = y)
        ∧      (a-d ≤ y ⇒ y′ = a-d)
}}
```

Design By Contract: Example. Account manager (ctd)

```
Class Accountmanager
{   Fct f = (x : Person) Account: …
export
{   method  credit = (x : Person, y : Var Amount, z : Var Account):
            pre       f(x) ≠ nil
            post      z′ = f(x)
              ∧       post.f(x).balance(y)
            body      f(x).balance(y); z := f(x)
}
import
{   a, d : Var Nat;
    invariant a ≥ d;

    method balance = (y : Var Amount):
            pre       true
            post      …
}}
```

DbC for Export/Import components

- Step 1: Specify: SbC: We give SbC for all methods
- Step 2: Design: Component implementation
  - We provide a body for each exported method
  - Only method calls are allowed that are either in the export or import parts (no calls of "undeclared" methods)
  - The body is required to fulfil the pre/postconditions
- Step 3: Verify: Component verification
  - Verify the pre/post-conditions for each implementation of an export method
  - We refer to the SbCs for the imported (and the exported) methods use in nested calls in the bodies when proving the correctness of each exported method w.r.t. its pre/postconditon

Remarks

- There is some similarity to Lamport's TLA where systems are modelled by
  - ◇ The set of actions a system can do
  - ◇ The set of actions the environment can do
  - ◇ Actions are represented by relations on states
  - ◇ Fairness/lifeness properties by temporal logic on system runs
  - ◇ Difference: actions are atomic - method calls are not

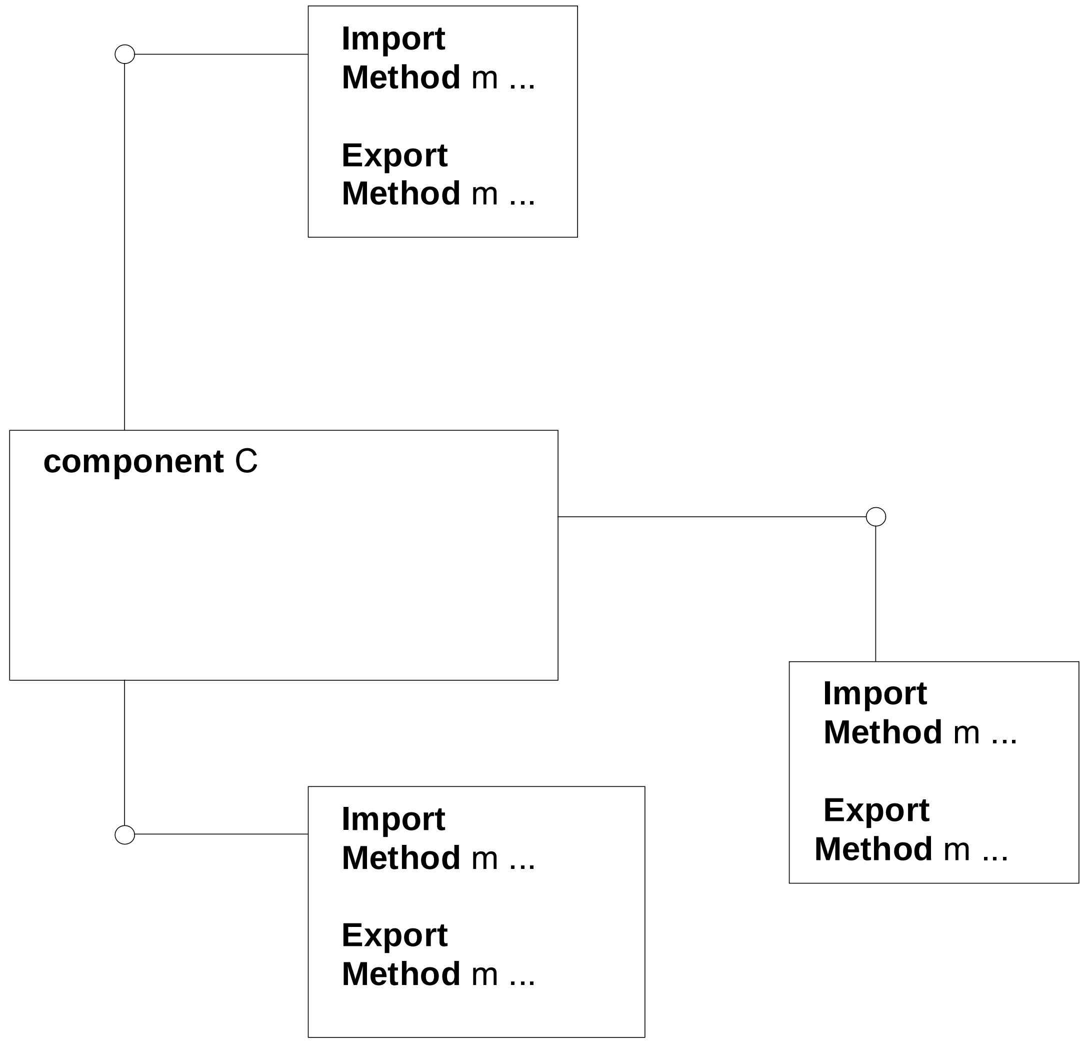
An example in TLA - taken from Leslie’s book

Zur Anzeige wird der QuickTime™
Dekompressor „TIFF (LZW)“
benötigt.

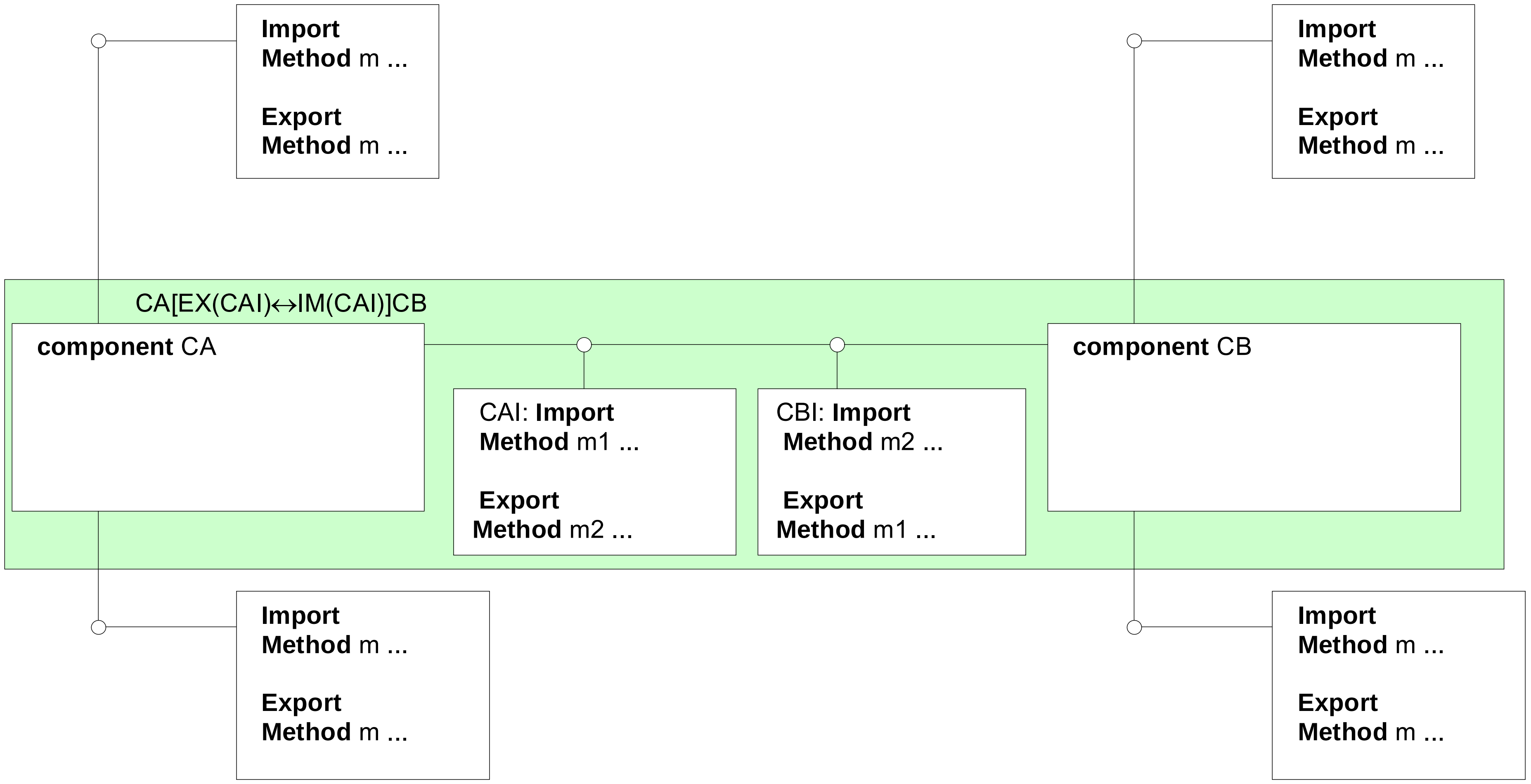
Remarks

- We may in addition structure the export and import part into
  - a set of pairs of export and import signatures that are sub-signatures of the overall export and import interfaces
- This pairs may be called sub-interfaces
- This leads in the direction of connectors

Components in OO with Multiple Sub-Interfaces



Composition



Composition for Export/Import Components

- Given E/I components ci with i = 1, 2, and export signature EX(ci) and import signature IM(ci)
- $\Re(\{c1, c2\})$ holds, if there are no name conflicts.
- Then export signature EX and import IM of the result of the composition $c1 \otimes c2$ is defined by

  $$EX(c1 \otimes c2) = (EX(c1) \backslash IM(c2)) \cup (EX(c2) \backslash IM(c1))$$

  $$IM(c1 \otimes c2) = (IM(c1) \backslash EX(c2)) \cup (IM(c2) \backslash EX(c1))$$

- The composed component $c = c1 \otimes c2$
  - ◇ exports what is exported by one of the components and not imported by the other one and
  - ◇ imports what is imported by one of the component and not exported by the other one.
- Methods that imported by one component and exported by the other one are bound this way and made local

Actually we get local (hidden) methods that way!

We ignore that to keep notation simple!

Verification of composed components

Let all definitions as before and assume SbC for all methods

For proving the correctness of composition we prove

- for each exported method m with pre-condition P_{ex} and post-condition Q_{ex}
- that is bound by some imported method m with pre-condition P_{im} and post-condition Q_{im} that

$$P_{im} \Rightarrow P_{ex}$$

$$Q_{ex} \Rightarrow Q_{im}$$

DbC for architectures export/import components

Design by contract for the export/import case:

- Step S: Specify system: Export only SbC
- Step A: Develop the architecture
  - ◇ Step AD: Design architecture: List components and their export/import methods
  - ◇ Step AS: Specify architecture: Give Export/Import SbC for all components
  - ◇ Step AV: Verify architecture
- Step I: Component implementation
  - ◇ Step ID: Design: We provide a body for each exported method

    Only calls are allowed that are either in the export or import parts (no calls of "undeclared" methods)
  - ◇ Step IS: Specification taken from architecture: The body is supposed to fulfil the pre/post-conditions
  - ◇ Step IV: Component verification: SbCs for imported methods are used when proving the correctness of each exported method for its pre/postconditon
- Step G: Component composition - integration: correctness for free

A fresh approach

- Forget about methods as atomic state changes
- Split message execution into two messages:
  - Calls
  - Returns

This means we go from

- State oriented specification to
- Communication (message exchange) oriented specification

In- and Out-Messages for a method header

- A method invocation consists of two interactions of messages called *the method invocation message* and the *return message*.
- Given a method header (for explanations see above)

  method m (w : WT, v : **Var** VT)

  the corresponding set of invocation messages is defined by

  $$\text{SINVOC}(m) = \{m(b_1,b_2,w,v): w \in \text{WT}, v \in \text{VT}, b_1, b_2 \in \text{Object}\}$$

  The return message has the type (where v' is the value of the variable after the execution of the method invocation)

  $$\text{RINVOC}(m) = \{\text{return\_m}(b_1,b_2,v'): v' \in \text{VT}, b_1, b_2 \in \text{Object}\}$$
- With each method we associate this way two types of messages, the invocation message and the return message.

Sets of messages

- Given a set of methods M we define

$$\text{SINVOC}(M) = \{ c \in \text{SINVOC}(m) : m \in M\}$$

$$\text{RINVOC}(M) = \{ c \in \text{RINVOC}(m) : m \in M\}$$

This way we denote the sets of all possible invocation and return messages of methods that are in the set of methods M.

Example. Account manager (continued)

Class Accountmanager =

{…

  export

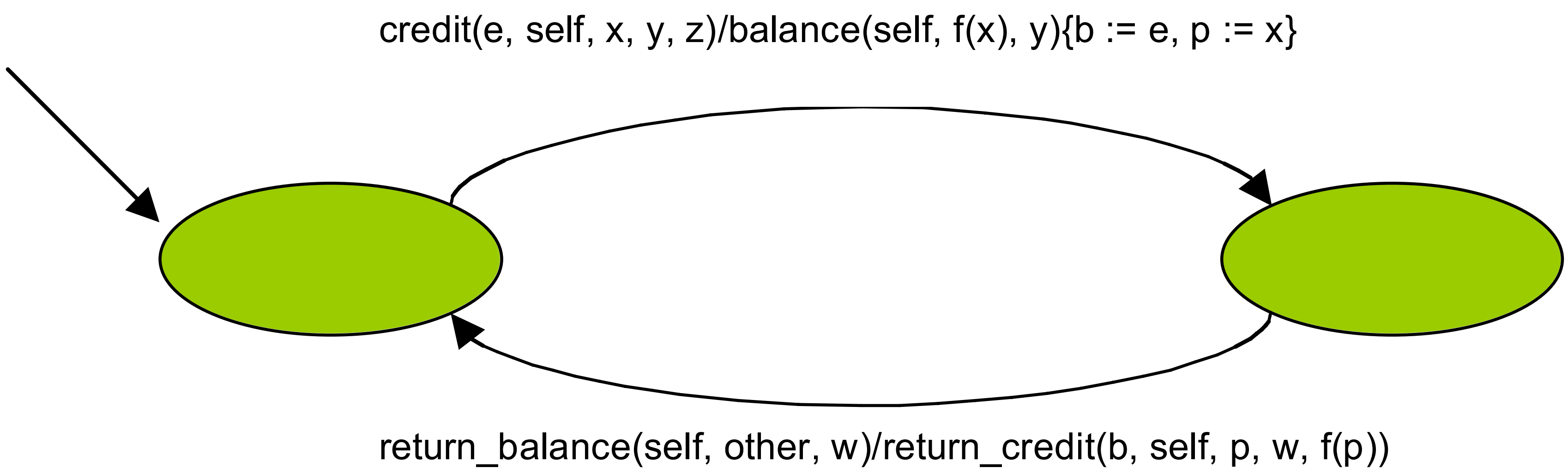
  method credit = (x: Person, y: **Var** Amount, z: **Var** Account)

…

  import method balance = (y: **Var** Amount)

}

Again the state machine


credit(e, self, x, y, z)/balance(self, f(x), y){b := e, p := x}
return_balance(self, other, w)/return_credit(b, self, p, w, f(p))

Modelling Export/Import Interfaces by I/O Machines

In- and Out-Messages of a syntactic class interface

- Let c be a syntactic export/import interface with
  - ◇ set EX(c) of export class names and their methods and
  - ◇ set IM(c) of import class names and methods.

  They define a set In(c) of ingoing messages

$$In(c) = SINVOC(EX(c)) \cup RINVOC(IM(c))$$

and a set of outgoing messages Out(c) specified by

$$Out(c) = SINVOC(IM(c)) \cup RINVOC(EX(c))$$

Export/import state machine

- Given an interface c with an attribute set V and a set of methods, the associated state machine has the form

  $$\Delta: \text{State} \times \text{In}(c) \to ((\text{State} \times \text{Out}(c)) \cup \{\perp\})$$

  For $m \in \text{In(IF)}$ the equation $\Delta(s, m) = \perp$ expresses that the method invocation does not terminate.

  The state space State is defined by the equation

  $$\text{State} = \Sigma(V) \times \text{CTS}$$

- Here CTS is the control state space. Its members can be understood as representations of the control stack. Since we do not want to go deeper into the very technical discussion of control stacks, we do not further specify CTS. Again, we assume that a set of initial states $\text{IState} \subseteq \text{State}$ is given.

Composition of the two state machines

Consider machines associated with the components c_i (i = 1, 2):

Δ_i: $State_i \times In(c_i) \to (Statei \times Out(c_i)) \cup \{\perp\}$

We define the composed state machine

Δ: $State \times In(c) \to (State \times Out(c)) \cup \{\perp\}$

as follows

$State = State_1 \times State_2$

for $x \in In(c)$ and $(s1, s2) \in State$ we define:

$$
\begin{aligned}
& x \in In(c1) \wedge (s1', y) = \Delta1(s1, x) \quad \Rightarrow \\
& \qquad y \in In(c2) \Rightarrow \Delta((s1, s2), x) = \Delta((s1', s2), y) \\
& \quad \wedge \quad y \notin In(c2) \Rightarrow \Delta((s1, s2), x) = ((s1', s2), y) \\
& x \in In(c1) \wedge \Delta1(s1, x) = \perp \Rightarrow \Delta((s1, s2), x) = \perp
\end{aligned}
$$

In analogy we define the case of input to the second component:

$$x \in In(c2) \wedge (s2', y) = \Delta 2(s2, x) \quad \Rightarrow$$

$$y \in In(c1) \Rightarrow \Delta((s1, s2), x) = \Delta((s1, s2'), y)$$

$$\wedge \quad y \notin In(c1) \Rightarrow \Delta((s1, s2), x) = ((s1, s2'), y)$$

$$x \in In(c2) \wedge \Delta 2(s2, x) = \perp \Rightarrow \Delta((s1, s2), x) = \perp$$

This gives a recursive definition for state transition function Δ.

We define

$$\Delta = \Delta 1 \| \Delta 2$$

Actually, this way of definition results in a classical least fixpoint characterization of the composed transition relation Δ.

Interface Abstraction by Functions on Streams

Given a state machine

$$\Delta: State \times In(c) \to (State \times Out(c)) \cup \{\perp\}$$

we specify a function called interface abstraction

$$\alpha_\Delta: State \to (In(c)^* \to Out(c)^*)$$

by (let $i \in In(c)$, $x \in In(c)^*$,

$\langle i \rangle \hat{}\, x$ denotes the concatenation of the

one element sequence $\langle i \rangle$ with the stream x)

$$(\sigma', o) = \Delta(\sigma, i) \Rightarrow \alpha_\Delta(\sigma)(\langle i \rangle \hat{}\, x) = \langle o \rangle \hat{}\, \alpha_\Delta(\sigma')(x)$$

$$\Delta(\sigma, i) = \perp \Rightarrow \alpha_\Delta(\sigma)(\langle i \rangle \hat{}\, x) = \langle\rangle$$

Obviously $\alpha_\Delta(\sigma)$ is prefix monotonic.

$\alpha_\Delta(\sigma)$ is the abstract interface for the state machine (Δ, σ),

- which is the state machine with the initial state σ
- and the state transition function Δ.

The interface abstraction gets rid of the state space (information hiding)

Observable Equivalence

- Two components c1 and c2 are observably equivalent, if and only
- if their state machines $(\Delta 1, \sigma 1)$ and $(\Delta 2, \sigma 2)$ fulfil the equation

$$\alpha_{\Delta 1}(\sigma 1) = \alpha_{\Delta 2}(\sigma 2)$$

Account manager (continued)

- We define the associated function

  $\alpha_\Delta(\sigma)$

  for the component Accountmanager with initial state σ by one equation:

$$\alpha_\Delta(\sigma)(\langle \text{credit}(e, \text{self}, x, y, z)\rangle \hat{} \langle \text{return\_balance}(\text{self}, \text{other}, w)\rangle \hat{} x) = \langle \text{balance}(\text{self}, f(x), y)\rangle \hat{} \langle \text{return\_credit}(e, \text{self}, x, w, f(x))\rangle \hat{} \alpha_\Delta(\sigma')(x)$$

- In this case the specification fairly simple due to the simple structure of the class.
- In particular, the problem of making the stack explicit disappears.

Concluding Remarks

- Export/import view
- Call are split into to messages
- Classes and object can be modelled state machines with input and output
- This leads to a message switching view onto export/import components
- Concurrency can be included

Further issues

- Why not go to full message switching then
- How would a programming language look like based on this paradigm