In an OO software system
• which consists of a set of classes where
  ◦ all sub-method calls are 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 (reaction)
  provided that
  ◦ the call terminates
• Then the execution of a method invocation can be
  modelled as one large state transition (we all this the
  closed view)
Specification by contract for a Method

• Let \( V = \{ a : \text{Var} \ AT \} \) be an attribute set.
• A specification by contract for a method with header
  \[
  \text{Method} \ m (w : \text{WT}, v : \text{Var} \ VT)
  \]
  in a class with attribute set \( V \) is given by
  \[
  \begin{align*}
  \text{Method} & \ m (w : \text{WT}, v : \text{Var} \ VT) \\
  \text{pre} & \ P(w, v, a) \\
  \text{post} & \ Q(w, v, a, v', a')
  \end{align*}
  \]
• Here \( P(w, v, a) \) and \( Q(w, v, a, v', a') \) denote predicates
  \( \diamond v, a \) denote the values before and \( v', a' \) the values of the variables after the method invocation
• Precondition \( P \) guarantees termination

Example. Specification by Contract (SbC)

• We consider only one method here and assume only one attribute
  \[
  u : \text{Var} \ \text{Seq} \ \text{Data}
  \]
• Specification by contract for a method that gets access ("reads") the \( i \)-th element of sequence \( u \):
  \[
  \begin{align*}
  \text{Method} & \ \text{get} \ (i : \text{Nat}, r : \text{Var} \ \text{Data}); \\
  \text{pre} & \ 1 \leq i \leq \text{length}(u) \\
  \text{post} & \ r' = \text{ith}(i, u) \land u' = u
  \end{align*}
  \]
  Here we assume that the functions
  \( \diamond \text{length}(s) \) (yielding the length of sequence \( s \)) and
  \( \diamond \text{ith}(i, s) \) (yielding the \( i \)-th element of sequence \( s \)) are predefined for sequences, for instance, by an algebraic data
Specification of the data elements

**SPEC** \( \text{SEQ} = \{ \)

* based_on BOOL,
* type Seq \( \alpha \),

\( \circ : \) Seq \( \alpha \),
\( \langle \_ \rangle : \) \( \alpha \rightarrow \) Seq \( \alpha \),
\( \circ : \) Seq \( \alpha \), Seq \( \alpha \rightarrow \) Seq \( \alpha \),
\( \circ : \) Seq \( \alpha \), Seq \( \alpha \rightarrow \) Seq \( \alpha \),

**Mixfix** one-element sequence

**Infix** concatenation

**Axioms**

\( \text{Seq} \( \alpha \) \text{ generated by} \circ, \langle \_ \rangle, \circ, \)

* iseseq(\( \circ \)) = true,
* iseseq(\( \langle a \rangle \)) = false,
* iseseq(\( a \circ y \)) = and(iseseq(x), iseseq(y)),

\( \text{length}(\circ) = 0, \)
\( \text{length}(\langle a \rangle) = 1, \)
\( \text{length}(a \circ y) = \text{length}(x) + \text{length}(y), \)

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

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

drop(a, ⟨a⟩°x) = x,
a ≠ b ⇒ drop(a, ⟨b⟩°x) = ⟨b⟩°drop(a, x),

cut(s, i, 0) = ∅
cut(s, 0, j+1) = first(s) ° cut(rest(s), 0, j)),
cut(s, i+1, j+1) = cut(rest(s), i, j),

x°∅ = x = ∅°x,
(x°y)°z = x°(y°z),

first(⟨a⟩°x) = a,
last(x°⟨a⟩) = a,
head(⟨a⟩°x) = ⟨a⟩,
rest(⟨a⟩°x) = x

Simple Export Interfaces

• A syntactic export interface consists of a set of classes and their methods; more precisely of
  ◦ a set of class types (names) and
  ◦ for each class a set M of method headers.
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 ≠ void
    post c' = c ∧ v' = c

  method delete ()
    pre  c ≠ void
    post c' = void
}

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

```java
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:
```java
method credit = (x : Person, y : Var Amount, z : Var Account):
    pre  f(x) ≠ nil
    post z′ = f(x)
    ∧ f(x).d′ = f(x).d+y′
    ∧ (f(x).a-f(x).d ≥ y ⇒ y′ = y)
    ∧ (f(x).a-f(x).d ≤ y ⇒ y′ = f(x).a-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 a = d 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.

Example: Account manager (continued): Call forwarding

Note:
• 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
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.
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)
  ∧  f(x).d' = f(x).d+y'
  ∧  (f(x).a-f(x).d ≥ y ⇒ y' = y)
  ∧  (f(x).a-f(x).d ≤ y ⇒ 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

```java
{  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

• 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 for Export/Import Components

• Given E/I components \( c_i \) with \( i = 1, 2 \), and export signature \( \text{EX}(c_i) \) and import signature \( \text{IM}(c_i) \) where there are no name conflicts.

• Then export signature \( \text{EX} \) and import \( \text{IM} \) of the result of the composition \( c_1 \otimes c_2 \) is defined by

\[
\text{EX}(c_1 \otimes c_2) = (\text{EX}(c_1) \setminus \text{IM}(c_2)) \cup (\text{EX}(c_2) \setminus \text{IM}(c_1))
\]

\[
\text{IM}(c_1 \otimes c_2) = (\text{IM}(c_1) \setminus \text{EX}(c_2)) \cup (\text{IM}(c_2) \setminus \text{EX}(c_1))
\]

• The composed component \( c = c_1 \otimes c_2 \)
  
  ◦ 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