5.5 Behavioral Subtyping

Subtyping of programming languages enforces that
- no type errors occur, and
- there is a method implementation for each
  method invocation.

It does not guarantee that subtype objects behave like supertype objects.

Example: (no specialization)

D-objects do not behave similar to C-objects:

```java
class C {
    int a;
    int getA() { return a; }
    void incr() { a++; }
}

class D extends C {
    int getA() { return a+2; }
    void incr() {
        throw new NullPointerException();
    }
}
```

Problem:
What does it mean that a subtype behaves like the supertype or is a specialization of the supertype? In particular, how do we handle
- abstract types/classes?
- extensions of the state space?

The operational semantics provides a bad basis for defining behavioral subtyping.

Approach:
Define the behavioral subtype relation based on specified properties:
→ Each subtype object has to satisfy the specification of the supertype.

For pre- and post-specifications this means:
- pre[Supertype] => pre[Subtype]
- post[Subtype] => post[Supertype]

Explanation: (Behavioral subtyping)
A subtype S is called a behavioral subtype of T iff S-objects behave according to the specification of T (auf Deutsch: S ist ein konformer Subtyp von T).
Remark:

- Behavioral subtyping depends on the specification.
- Main goal is to understand the relation between sub- and supertypes.
- The described techniques should be applied to informal specifications/documentation as well.

Kinds of Specialization:

- Refining implementation
- Refining nondeterminism
- Adding methods and extending state

Overview:

- Relation between Specifications & Implementations
- Pre- and postspecifications without abstract variables and without abstraction (concrete specs)
- Pre- and postspecifications with abstract variables and with abstraction (abstract specs)
- Treatment of invariants
- Treatment of frame properties
- Illustrating example
Relation between Specifications & Implementation

Instead of relating an implementation of class D to the specifications of all its supertypes, we check that:
- the specification of D, spec(D), is a behavioral subtype of its direct supertype,
- the implementation of D, impl(D), satisfies spec(D).

Illustration:

```java
class C {
    pre  PMC(this,p)
    post QMC(this,p,result)
    R m( T p ) { .../* impl(m,C) */ }
}

class D extends C {
    pre  PMD(this,p)
    post QMD(this,p,result)
    R m( T p ) { .../* impl(m,D) */ }
}
```

Check:

a)  \{ PMD(this,p) \}  impl(m,D)  \{ QMD(this,p,result) \}
b)  PMC(this,p)  \Rightarrow  PMD(this,p)
c)  QMD(this,p,result)  \Rightarrow  QMC(this,p,result)
Remark:

Specification languages usually support specification inheritance in a form that establish (b) and (c) or refined versions of it.

Example: (Spec. inheritance in JML)

```java
class C {
   /*@ public normal_behavior
      @   requires PREMC
      @   ensures POSTMC ; @*/
   R m( T p ){ ... }
}

class D extends C {
   /*@ also
      @   public normal_behavior
      @   requires PREMD
      @   ensures POSTMD ; @*/
   R m( T p ){ ... }
}
```

The specification of D is an abbreviation for:

```java
class D extends C {  // not JML
   /*@ public normal_behavior
      @   requires PREMC || PREMD
      @   ensures ( \old(PREMC) => POSTMC )
      @   && ( \old(PREMD) => POSTMD ) ; 
      @*/
   R m( T p ){ ... }
}
```
Concrete Pre-Post-Specifications

Pre- and postconditions of methods in supertypes are valid pre-/postconditions for subtype methods:

→ Inheritance of specification

Subtypes without Additional Attributes:

Inherited method:
- Specification is inherited without change.

Overriding method:
- Precondition inherited or new weaker precondition
- Postcondition inherited and possibly strengthened by additional properties

Example:

```java
class C {
    /*@ public normal_behavior
        @ requires P ;
        @ ensures  Q ;
        @*/
    C m(){ ... }
}
```
class D extends C {
  /*@ also
     @   public normal_behavior
     @   requires P ;
     @   ensures \result instanceof D ;
     @}*/
  C m(){ ... }
}

Additional method:
- no constraints (but see about invariants).

Remark:
Similar restrictions have to be applied to the specification of exceptional behavior.

Subtypes with Extended State:
If the state space is extended in the subtype, it can be necessary to strengthen the precondition of overriding methods w.r.t. properties concerning the additional attributes. However, this would violate:

pre[Supertype] => pre[Subtype]
Solution for closed programs:
Include subtype properties in the specification of the supertype and use the slightly weaker restrictions:
- \((\text{pre}[\text{Supertype}] \land \text{this instanceof Subtype}) \Rightarrow \text{pre}[\text{Subtype}]\)
- \((\text{post}[\text{Subtype}] \land \text{this instanceof Subtype}) \Rightarrow \text{post}[\text{Supertype}]\)

Remark:
- This technique can only be applied if supertypes know their subtypes.
- A more practical and elegant solution for handling extended state is to use abstraction.

Abstract Pre-Post-Specifications

The basic idea is to write specifications in terms of abstract states and to provide abstraction functions:
- from concrete objects to their abstract states
- from abstract states of subtype objects to abstract states of supertype objects.

The abstraction functions in particular provide the flexibility to handle extended state.
Here, we only consider an example:
Example:

class C {
    //@ public model boolean valid;
    //@ public model AS state;

    //@ public normal_behavior
    @  requires valid && r(state) ;
    @  ensures  q(state) ;
    @*/
    void m(){ ... }
}

class D extends C {
    private BD d;
    //@ private depends valid <- d;
    //@ private represents valid <- CD.pd(d) ;
    //@ private depends state <- d;
    //@ private represents state <- CD.f(d) ;
    ...
}

class E extends C {
    private BE e;
    //@ private depends valid <- e;
    //@ private represents valid <- CE.pe(e) ;
    //@ private depends state <- e;
    //@ private represents state <- CE.g(e) ;
    ...
}
- \( pd \) is a predicate expressing when attribute \( d \) has a valid state.
- \( f \) is an abstraction function mapping values of concrete type \( BD \) to the abstract type \( AS \).

The definitions of \( pd \) and \( f \) can be tailored to the needs of class \( D \). The same holds for \( pe \), \( g \) and class \( E \).

Remarks:

• Often one assumes an explicit abstraction function that maps values of subtype objects into the state space of the supertype.

• Abstract/model variables enable two kinds of state extensions:
  - overriding the representation function
  - additional abstract variables

• The variations in the subtypes can be captured by representation functions.
Treatment of Invariants

In principle, invariants can be expressed by pre- and postconditions. However, as a specification construct they allow to express restrictions on subtype behavior already in supertypes:

→ Invariants of supertypes have to be satisfied by additional subtype methods.

Example:

```java
class C {
    public int a = 0;
    //@ public invariant a >= 0;
    ...
    // no declaration of m
}

class D extends C {
    ...
    void m(){ a = -1; } // violates invariant
    ...
}
```
General Approach:

Invariants of supertypes have to be satisfied by all subtype methods. This can be formulated as:

\[ \text{inv[Subtype]} \implies \text{inv[Supertype]} \]

This can be achieved by invariant inheritance:

→ The subtype invariant is the conjunction of the supertype invariant and additional invariant clauses specified in the subtype.

Problems:

- What is the precise meaning of an invariant? When should invariants hold?
- How is subtyping and dynamic binding handled for invariants?

There is no well-established answer to these problems. We discuss existing solutions.
Semantical variants:

1. Invariants have to hold in visible states:
   JML: „Invariants have to hold in each state outside of a public method‘s execution and at the beginning and end of such execution.“

2. Transformation into pre- and postconditions:
   Invariants have to hold in poststates of constructors.
   If they hold in the prestate of a method, they have to hold in the poststate as well.

For verification, both variants are problematic:
Let S be a subtype of T, m be a method in T and x a variable of type T holding an S-object:
   ... x.m(...) ...

For the verification we need inv[S] as precondition.
But: S may not be known at the invocation site.
Example:

class C {
    public int a = 0;
    //@ public invariant a >= 0;
    ...
    void m(){
        ... // maintains invariant
    }
}

class Foo {
    void mfoo() {
        ... x.m() ...
    }
}

class D extends C {
    public int b = 0;
    //@ public invariant a > 1;
    //@ public invariant b >= 0;
    ...
    void m(){
        b = 4 / a ;
        ... // maintains both invariants
    }
}

Problem: What has to be shown for the prestate of a method invocation x.m()
Possible solution:

Consider as invariant of type T the following formula \( \text{INV}[T] \):
\[
\forall \text{Type S: } S \leq T \Rightarrow (\text{type(this)}=S \Rightarrow \text{inv}[S])
\]
where \( \text{inv}[S] \) is the invariant specified for the implementation of type S. Then we have for all subtypes \( U \) of \( T \):

\[
\text{type(this)} = U \Rightarrow (\text{inv}[U] \iff \text{INV}[T])
\]

### Treatment of Frame Properties

The specification of frame properties has to cope with two problems:

- **Information hiding**: not all assignable variables can be named in the specification.
- **Extended state**: The supertype specification cannot capture the state of additional attributes in subtypes.
Example:

```java
class C {
    public int a = 0;
    private int b = 0;
    public static int c = 123;
    ...

    /**
     * public normal_behavior
     * @ assignable a;
     */
    public void m(){ a++; b++ }
}

class Foo {
    void mfoo() {
        ...
        x.m() ...
    }
}

class D extends C {
    public int d = 0;
    ...
    public void m(){
        super.m();
        d = 87;
        C.c = 4 ;
    }
}
```

18.06.2007 © A. Poetzsch-Heffter, TU Kaiserslautern
Possible solution:

- Use abstract attributes/variables, depends- and representation clauses.
- Information Hiding: Abstract attributes can depend on non-accessible attributes.
- Extended state: Depends relation can be extended in subtypes.

(Example is given in the following subsection)

Using the Techniques Together

Specification techniques for OO-programs have two goals:
- Specification of properties by annotating programs.
- Complete specification of types as basis for behavioral subtyping.

We illustrate this by a larger example.
Example:

The following example is a Java-version of the central example given in:


The example demonstrates:
- state extensions
- behavioral subtyping
public interface Reader {

    //@ public model instance boolean valid;
    //@ public model instance Object state;

    //@ public normal_behavior
    @ requires    valid;
    @ assignable  state;
    @ ensures     -1 <= \result
    @             && \result < 65535 ;
    @*/
    public int getChar();

    //@ public normal_behavior
    @ requires    valid;
    @ assignable  valid, state;
    @*/
    public void close();

}
public abstract
class BuffReader implements Reader {

    protected /*@ spec_public @*/ int lo, cur, hi;
    protected /*@ spec_public @*/ char[] buff;

    //@ public model boolean svalid;

    //@ public represents valid <-
    // @ this != null &&
    // @ 0 <= lo && lo <= cur && cur <= hi &&
    // @ buff != null && hi-lo <= buff.length &&
    // @ svalid ;
    //@

    public int getChar() {
        if ( cur == hi ) refill();
        if ( cur == hi ) return -1;
        cur++;
        return buff[cur-lo-1];
    }

    //@ public normal_behavior
    // @ requires valid;
    // @ assignable state;
    // @ ensures cur == \old(cur) ;
    //@
    public abstract void refill();

    //@ depends valid <- lo, cur, hi, buff, svalid;
    //@ depends state <- lo, cur, hi, buff, buff[*];
    //@ depends svalid <- lo, hi, buff;
}

18.06.2007 © A. Poetzsch-Heffter, TU Kaiserslautern
public interface BlankReader extends Reader {

    /*@ public normal_behavior
    @  requires    0 <= n;
    @  assignable  valid, state;
    @  ensures     valid  && \result == this ;
    @*/
    public BlankReader init( int n );
}

A non-trivial application of BlankReader and BlankReaderImpl (s. next slide):

public class ReaderTest {

    public static void main( String[] args ) {
        BlankReader br = new BlankReaderImpl();
        br.init(1000000);

        int count = 0;
        int chr;
        do {
            chr = br.getChar();
            count++;
        } while( chr != -1 );
        br.close();
        System.out.println(count);
    }
}
public class BlankReaderImpl
        extends BuffReader
        implements BlankReader
{
    private int num;
    //@ private represents svalid <- hi <= num ;

    public BlankReader init( int n ) {
        num = n;
        buff = new byte[ Math.min(n,8192) ];
        lo = 0;
        cur = 0;
        hi = buff.length;
        for( int i = 0; i < hi; i++ ) {
            buff[i] = 32;
        }
        return this;
    }

    public void refill() {
        lo = cur;
        hi = Math.min( lo+buff.length, num );
    }

    public void close() {};

    //@ private depends state <- num ;
    //@ private depends svalid <- num ;
    // representation of state not
    // considered
}