Konzepte objektorientierter Programmierung
– Lecture 11 –

Prof. Dr. Peter Müller

Software Component Technology

Wintersemester 04/05

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Meeting the Requirements

Cooperating Program Parts with Well-Defined Interfaces

→ Objects (data + code)
→ Interfaces
→ Encapsulation

Classification and Specialization

→ Classification, subtyping
→ Polymorphism
→ Substitution principle

Inherently Concurrent Execution Model

→ Active objects
→ Message passing

Correctness

→ Interfaces
→ Encapsulation
→ Simple, powerful concepts

Agenda for Today

11. Interface Specifications

- 11.1 Contracts
- 11.2 Semantics of Interface Specifications
- 11.3 Model-Based Specifications
- 11.4 A Closer Look at Modifies Clauses

Objectives

- Specification techniques
- Description of component behavior

Correctness

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;
  …

  public void insert( int i ) {
    for ( int j = 0; j < next; j-- )
      if array[ j ] == i then return true;
    return false;
  }
}
```

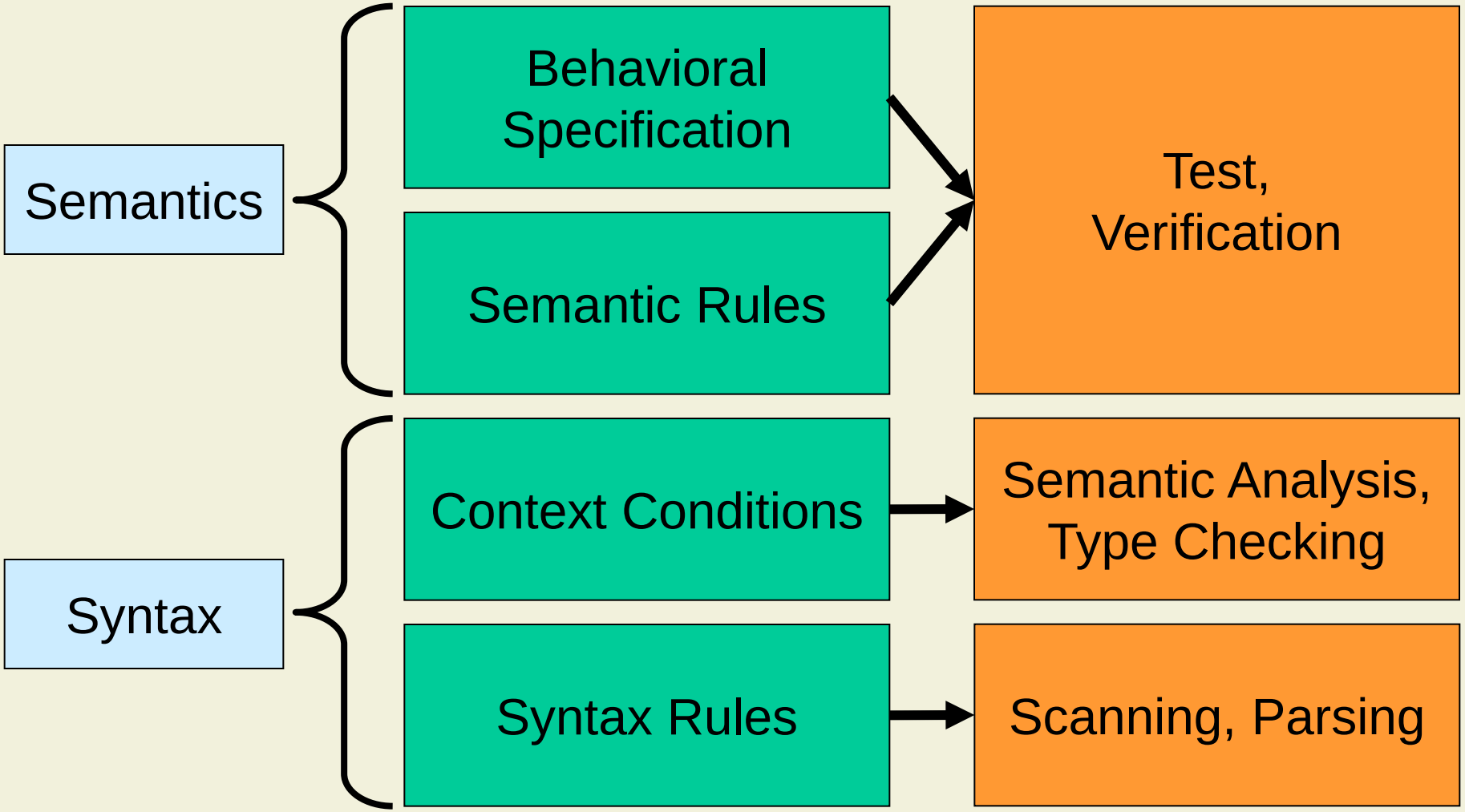
Behavioral Specification

Semantic Rules

Context Conditions

Syntax Rules

Aspects of Correctness



Test and Verification

Test

- Objective
  - Detect bugs
- Examples
  - White box test
  - Black box test
- Problems
  - Successful test does not guarantee correctness

Verification

- Objective
  - Prove correctness
- Examples
  - Formal verification based on a logic
  - Symbolic execution
- Problems
  - Expensive
  - Formal specification of behavior is required

11. Interface Specifications

Pre-Post Specifications

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;

  public boolean has( int i ) {
    for ( int j = 0; j < next; j++ )
      if ( array[ j ] == i ) return true;
    return false;
  }
  private void resize( ) {
    int[ ] tmp = new int[ array.length + 10 ];
    System.arraycopy( array, 0, tmp, 0,
      array.length );
    array = tmp;
  }
```

```
  // requires !has( i )
  // ensures has( old( i ) )
  public void insert( int i ) {
    if ( next == array.length )
      resize( );
    array[ next ] = i;
    next++;
  }

 …
}
```

Frame Properties

```
// requires !has( i )
// ensures has( old( i ) )
public void insert( int i ) {
  array[ 0 ] = i;
  next = 1;
}
```

```
Set s = new ArraySet( );
s.insert( 42 );
s.insert( 1492 );
boolean b = s.has( 42 );
```

```
// requires !has( i )
// ensures has( old( i ) )  &&
//     ∀j: old( has( j ) ) ⇒ has( j )
public void insert( int i ) { … }
```

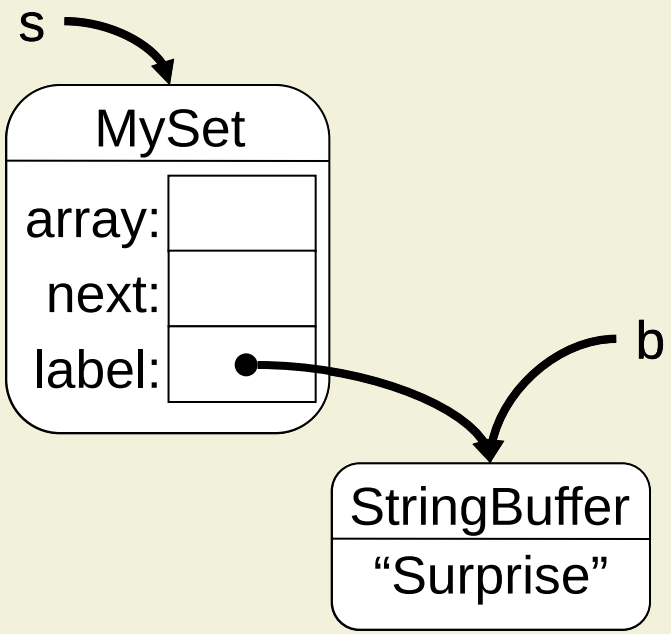
- Methods can have side-effects
- Frame properties describe **what is left unchanged** by a method execution

Frame Properties (cont'd)

```
// requires !has( i )
// ensures has( old( i ) )  &&
//     ∀j: old( has( j ) ) ⇒ has( j )
public void insert( int i ) { … }
```

```
void foo( Set s, StringBuffer b ) {
  b.insert( 0, "Hello" );
  s.insert( 1492 );
  System.out.println( b );
}
```

```
class MySet extends ArraySet {
  StringBuffer label;
  public void insert( int i ) {
    label.insert( 0, "Surprise" );
    super.insert( i );
  }
}
```



Modifies Clauses

- In **modular programs**, not all locations that remain unchanged are known
- **Modifies clauses** specify which locations **may be modified** by a method
- Locations **not mentioned** in the modifies clause **must remain unchanged**

```
// requires !has( i )
// ensures has( old( i ) )  &&
//    ∀j: old( has( j ) ) ⇒ has( j )
// modifies array[ next ], next,
//                 array
public void insert( int i ) {
  if ( next == array.length )
    resize( );
  array[ next ] = i;
  next++;
}
```

Example Revisited

```
// requires !has( i )
// ensures has( old( i ) )  &&
//  ∀j: old( has( j ) ) ⇒ has( j )
// modifies array[ next ], next, array
public void insert( int i ) { … }
```

```
void foo( Set s, StringBuffer b ) {
  b.insert( 0, “Hello” );
  s.insert( 1492 );
  System.out.println( b );
}
```

```
class MySet extends ArraySet {
  StringBuffer label;
  public void insert( int i ) {
    label.insert( 0, “Surprise” );
    super.insert( i );
  }
}
```

- **Behavioral subtyping:** Subtype methods have to satisfy modifies clauses of overridden supertype methods

Object Invariants

- Object invariants describe **consistency criteria** for objects
- **Invariants** have to hold in all states, in which an object can be accessed by other objects
- Invariants refer to **one execution state**

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;

  // invariant array != null      &&
  //    0 <= next <= array.length


  public void insert( int i ) {
    if ( next == array.length )
      resize( );
    array[ next ] = i;
    next++;
  }
 ...
}
```

History Constraints

- History constraints relate **two succeeding execution states**
- History constraints enable **datatype induction**

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;
  // constraint old( array.length ) <= array.length
  private void resize( ) {
    int[ ] tmp = new int[ array.length + 10 ];
    …  }
 …
}
```

```
class Person {
  private int age;
  // constraint old( age ) <= age
  …
}
```

11. Interface Specifications

Hoare Triples

- Like programs, specifications need a well-defined semantics
- Semantics of specifications can be defined in terms of **Hoare triples**

$$\{ \mathbf{P} \}\ m(\ \ldots\ )\ \{ \mathbf{Q} \}$$

- **Total correctness**: If **P** holds in the prestate of an execution of method m, then m terminates in a state in which **Q** holds
- **Partial correctness**: If **P** holds in the prestate of an execution of method m and m terminates, then **Q** holds in the poststate of the execution

Assuming Invariants

- Methods have to **assume the invariant of this** to hold in the prestate

```
// requires !has( i )
public void insert( int i ) {
  if ( next == array.length )
    resize( );
  array[ next ] = i;
  next++;
}
```

- Methods have to **assume that invariant of all allocated objects** to hold in the prestate

```
class Client {
  ArraySet as;
  // requires !as.has( 5 )
  public void foo( ) {
    as.insert( 5 );
  }
}
```

Notation

- $inv_T(o)$ denotes the invariant of class T for object o

```
class Redundant {
  private int a, b;
  // invariant this.a == this.b
  …
}
```

$$inv_{Redundant}(o) \equiv o.a == o.b$$

- INV_T denotes the conjunction of the invariant of class T for all allocated T-objects

$$INV_T \equiv \forall o: allocated(o) \wedge type(o) <: T \Rightarrow inv_T(o)$$

Semantics of Pre-Post Specifications

- Each method m may **assume** in the prestate of its executions
  - **The invariants of all allocated objects**
  - **The conjunction of m's preconditions**, **P**
- The method must **guarantee** in the poststate
  - **The conjunction of its postconditions**, **Q**

$$\{ \mathbf{P} \wedge \forall S: INV_S \}\ m(\ \ldots\ )\ \{ \mathbf{Q} \}$$

Semantics of Modifies-Clauses

- Each method m may **modify**
  - **Locations mentioned in its modifies clause**, **M**
  - **Locations of newly allocated objects**
- All other locations have to be left unchanged
  - Temporary modifications are possible

$$\{ \mathbf{P} \wedge \forall S: INV_S \}$$
$$m(\ldots)$$
$$\{ \forall x,f: [x,f] \in \mathbf{M} \vee \ !old(allocated(x)) \vee x.f == old(x.f) \}$$

Modifies Clauses: Example

```
// modifies array[ next ], next,
//                 array
public void insert( int i ) {
  if ( next == array.length )
    resize( );
  array[ next ] = i;
  next++;
}
```

```
// modifies array
private void resize( ) {
    int[ ] tmp = new int[ array.length + 10 ];
    System.arraycopy( array, 0, tmp, 0,
                                    array.length );
    array = tmp;
}
```

- insert modifies directly
  - this.array[this.next], this.next
- insert modifies indirectly via invocation of resize
  - this.array
  - Locations of newly allocated array

Semantics of Invariants

- Invariants have to **hold in pre- and poststates** of method executions, but can be temporarily violated in between

```
class Redundant {
  private int a, b;
  // invariant a == b
  public void set( int v ) {
    // invariant holds
    a = v;
    // invariant violated
    b = v;
    // invariant holds
  }
}
```

$$\{ \mathbf{P} \wedge \forall S: INV_S \}\ m(\ \ldots\ )\ \{ \forall S: INV_S \}$$

Verification of Invariants

- Assumption: The invariant of class T refers to **private attributes of T-objects only**
- We have to prove that each method or constructor of class T **guarantees the invariant of class T for all allocated T-objects** in the poststate

$$\{ \mathbf{P} \wedge \forall S: INV_S \}\ m(\ \ldots\ )\ \{ INV_T \}$$

- Invariants are difficult for object structures with aliasing (see Lecture 4)

11. Interface Specifications

Observer Methods

```
// requires   ∀j. 0 <= j < next: array[ j ] != i
// ensures    array[ old( next ) ] == old( i )  &&
//            ∀j. 0 <= j < next: old( array[ j ] ) == array[ j ]
public void insert( int i ) { … }
```

```
// requires   !has( i )
// ensures    has( old( i ) ) && ∀j: old( has( j ) ) ⇒ has( j )
public void insert( int i ) { … }
```

- Observer methods enable **implementation-independent specifications**
  - Information hiding
  - Exchange of implementations

Problems of Observer Methods

- Observer methods can be specified
  - By other observer methods or
  - By referring to the implementation

```
class ArraySet implements Set {
  // requires   true
  // ensures    ∃j. 0 <= j < next:
  //                array[ j ] == i
  public boolean has( int i ) { … }
  …
}
```

- No well-defined implementation-independent specification for observer methods

```
interface Set {
  // requires   true
  // ensures    ??
  public boolean has( int i );
  public void insert( int i );
}
```

Excursus: Abstract Datatypes

- Abstract datatypes describe datatypes without giving an implementation
- Laws (axioms) can be used to define the semantics of the datatype

```
datatype ▫
functions:
 0:     → ▫
 succ:  ▫ → ▫
laws:
 L1: succ(n)=succ(m) ⇒ n=m
 L2: succ(n)≠0
 L3: 0∈S ∧ (n∈S ⇒ succ(n)∈S)⇒
     ∀x: x∈S
end
```

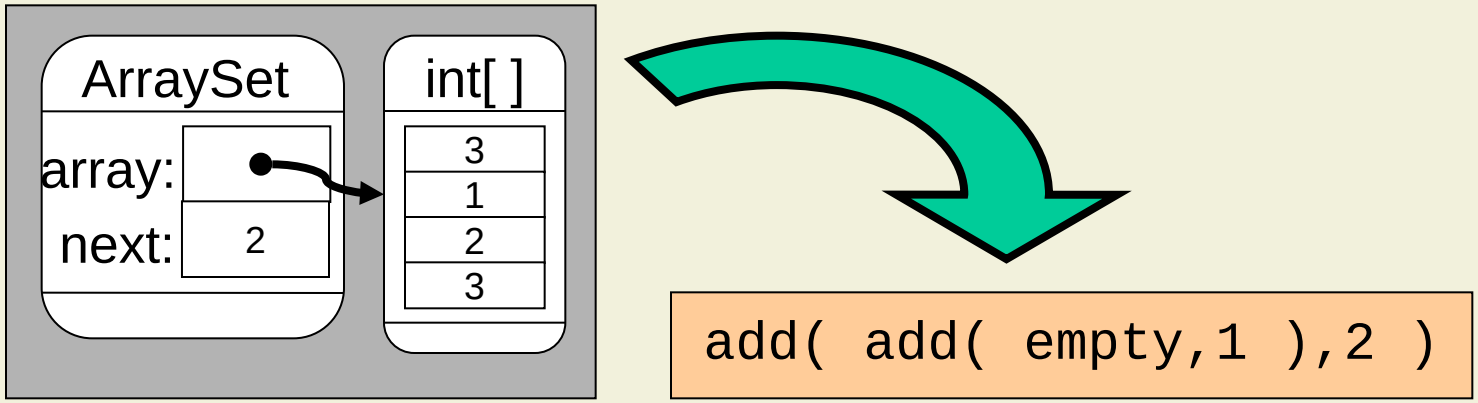
- See M. Wirsing: *Algebraic Specification*. In: J. van Leeuwen: *Handbook of Theoretical Computer Science*, Volume B. Elsevier, 1990.

Example: Abstract Datatype Set

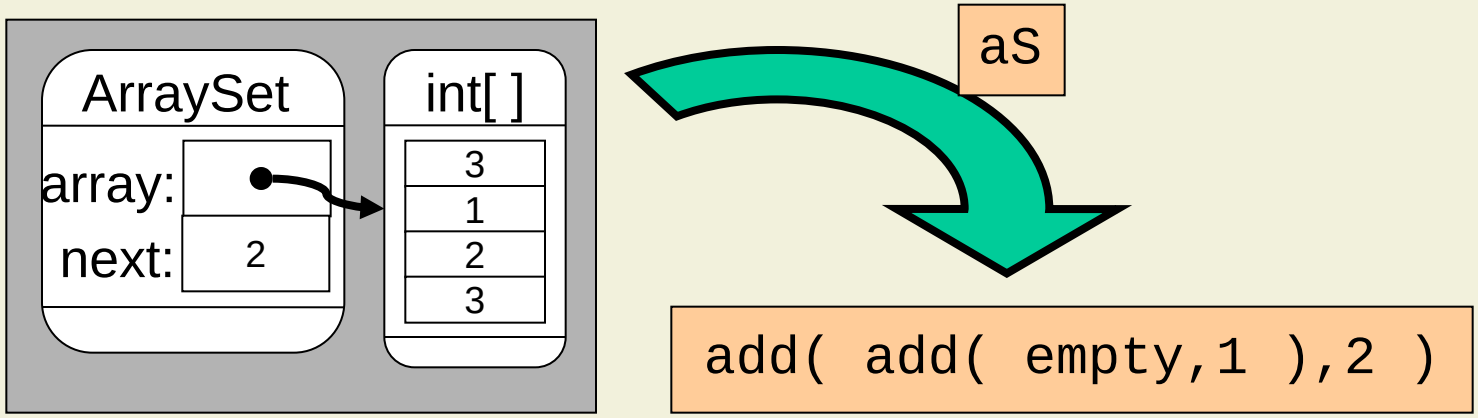
```
datatype Set
 uses: ▫, Bool
 functions:
  empty:   → Set
  member:  Set × ▫ → Bool
  add:     Set × ▫ → Set
 laws:
  L1: ¬member( empty,i )
  L2: member( add( S,i ),i )
  L3: add( add( S,i ),i ) = add( S,i )
  L4: add( add( S,i ),j ) = add( add( S,j ),i )
end
```

Abstraction Functions

- Specify interfaces in terms of the **mathematical vocabulary** provided by abstract datatypes
  - Abstract datatype serves as **model** for the implementation (**two-tiered specifications**)
- Relate object (structures) to terms of the abstract datatypes by **abstraction functions** [Hoare 72]



Example: Abstraction Function




```
aS: Object → Set
  aS( X ) = aA( X.array,0,X.next )

aA: Object × ▫ × ▫ → Set
  n ≥ l ⇒ aA( X,n,l ) = empty
  n < l ⇒ aA( X,n,l ) = add( aA( X,n+1,l ),X[n] )
```

Example: Interface Specification

```
interface Set {
  // requires   true
  // ensures    result = member( aS( this ),i )
  public boolean has( int i );

  // requires   ¬member( aS( this ),i )
  // ensures    aS( this ) = add( old( aS( this ) ),old( i ) )
  public void insert( int i );
}
```

- Abstraction functions omitted for **boolean** and **int**
  - `aB: boolean → Bool`
  - `aI: int → ▫`

Discussion

- Abstract datatypes
  - Enable **implementation-independent** specifications, especially of interfaces and observer methods
  - Preserve **information hiding** in specifications
  - Are mathematically **precise**
- Problems of two-tiered specifications
  - Programmers have to know programming and specification language
  - Specifications are **not executable** (for instance for testing)

Model Classes

- **Description** of abstract datatypes **in the programming language**
- Very **simple semantics**
  - No side-effects
  - No sharing
  - Performance is not important
- Real implementations are **specified based on model classes**
- Approach is used in the Java Modeling Language JML (see www.jmlspecs.org)

Model Class Example

```
public /*@ pure @*/ abstract class ModelSet {

  /** Is the argument equal to one of the values in the set. */
  public abstract boolean member( int elem ) ;

  /** Returns a new set that contains all the elements of
   *  this and also the given argument. */
  public abstract ModelSet add( int elem );
}
```

Model Classes: Laws

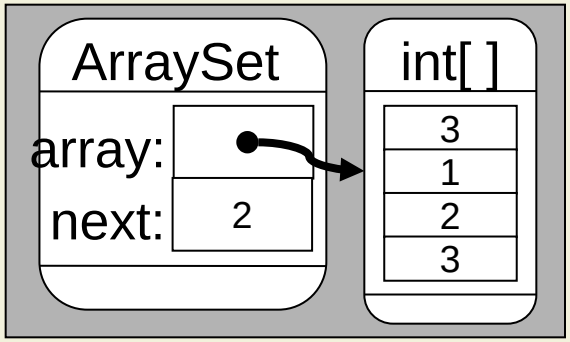
```
public /* pure */ abstract class ModelSet {

  // invariant ∀int e1, e2. equational_theory( this, e1, e2 )

  /* ensures result ⇔ !(new ModelSet( )).member( e1 )
       && s.add( e1 ).member( e1 )
       && s.add( e1 ).add( e2 ).equals( s.add( e2 ).add( e1 ) )
       && s.add( e1 ).add( e1 ).equals( s.add( e1 ) )          */
  static public pure model boolean
    equational_theory( ModelSet s, int e1, int e2 );
}
```

Model Attributes

- Link implementations to models
- Are not part of the state of an object



- May only be used in specifications

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;

  // public model ModelSet abs

  // requires  true
  // ensures  result == abs.member( i )
  public boolean has( int i ) { … }

  // requires !abs.member( i )
  // ensures abs.equals( old(abs.add( i )) )
  public void insert( int i ) { … }
…
}
```

Abstraction Functions for Model Classes

- Abstraction functions can be expressed in Java
- represents clauses define the values of model attributes

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;

  // public model ModelSet abs
  // private represents abs ← aS( )

  /* private model ModelSet aS( ) {
    ModelSet res = new ModelSet( );
    for ( int j = 0; j < next; j++ )
      res = res.add( array[ j ] );
    return res;
  } */

  …
}
```

Specification of Interfaces

- Model attributes may be declared in interfaces
- represents clauses are added in subclasses

```
interface Set {
  // public model ModelSet abs

  // requires  true
  // ensures  result == abs.member( i )
  public boolean has( int i );

  // requires  !abs.member( i )
  // ensures  abs.equals( old( abs.add( i ) ) )
  public void insert( int i );
}
```

11. Interface Specifications

Frame Properties Revisited

- Enumerating modifiable locations **violates information hiding**

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;
  // modifies array[ next ], next, array
  public void insert( int i ) { … }
  …
}
```

- Enumerating modifiable locations **does not work for interfaces**

```
interface Set {
  // modifies ??
  public void insert( int i );
}
```

Extended State Problem

- **Behavioral subtyping** requires subtype methods to satisfy modifies clauses of supertype methods

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;

  // modifies array[ next ], next, array
  public void insert( int i ) { … }
  …
}
```

- **But:** Subtype methods must have the right to modify the **extended state**

```
class MaxSet extends ArraySet {
  private int max;

  public void insert( int i ) {
    if ( i > max ) max = i;
    super.insert( i );
}
```

Modifies Clauses with Model Attributes

- Model attributes can be used in modifies clauses
- **Rule**: Right to modify model location includes right to modify all normal locations that are needed to compute the value of the model location

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;

  // public model ModelSet abs

  // modifies abs
  public void insert( int i ) { … }
  …
}
```

Dependencies

- "*locations that are needed to compute the value of the model location*" are declared by **depends clauses**
- Depends clauses can be private

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;

  // public model ModelSet abs
  // private depends abs ←
  //                 array[ * ], next, array

  // modifies abs
  public void insert( int i ) { … }
  …
}
```

Solution

- **Interfaces:** Model attributes can be declared in interfaces
- **Information hiding:** Model attributes are public and do not reveal implementation details
- **Extended state:** Subclasses can add new depends clauses for attributes of extended state

```
class MaxSet extends ArraySet {
  private int max;
  // private depends abs ← max

  …
}
```

Summary

- Interface specification
  - For objects: Invariants and history constraints
  - For methods: Pre-post specifications and modifies clauses
- Model-based specifications
  - Two-tiered: Implementation + algebraic specification
  - One-tiered: Implementation + model classes
  - Abstraction functions or model attributes to relate implementations to models