Specification-Only Class Members

A Formal Specification and Verification Lecture

Model Fields

Specification as Abstraction

TECHNISCHE UNIVERSITÄT DARMSTADT

Specification

- Higher abstraction level than implementation
- Concerned with **what** is computed, not how it is computed

Example

- Sortedness of a collection

  (**forall** int i; i >= 0 && i<coll.size()-1; coll.get(i) <= coll.get(i+1));

- Searching an element in a collection, etc.

Are we sufficiently abstract with our specifications?

Examples of Suboptimal Specifications

```
class Decimal {

    public static final short PRECISION = (short) 1000;
    /*@ spec_public @*/ private short intPart = (short) 0;
     //@ public invariant decPart >= 0 && decPart < PRECISION;
    /*@ spec_public @*/ private short decPart = (short) 0;

    /*@ public normal_behavior
      @ requires \invariant_for(other)
      @ ensures intPart * PRECISION + decPart ==
      @       \old( (intPart + other.intPart) * PRECISION + (decPart + other.decPart));
      @*/
     public void add(Decimal other) {
        …
     }
}
```

Examples of Suboptimal Specifications:
Matrix Implementation

```
public class MatrixImplem {

   private final int x;
   private final int y;
   private int[] matrix_implem;

    //@ ensures \result == matrix_implem[i*columns + j];
    public /*@ pure @*/ int get (int i, int j) {
       return matrix_implem[columns * j + i];
    }
}
```

- Implementation encodes (rows$\times$columns)-matrix into one-dimensional array
- Specification uses “low-level” data structure to specify e.g. get()

Observations

Specification uses data representation from implementation

Unnatural specifications

- Complicated and longer than necessary
  - Implementation-level data structures optimized for e.g. performance and not for comprehensibility

Close to implementation

- Greater probability of same bugs (in spec and code)
- Easily affected by code design changes

Model Fields:
Bringing Abstraction to the Specification

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class A {
  //@ model T name;
  //@ model static T name;
}
```

```
interface I {
  //@ model T name;
  //@ model instance T name;
}
```

- Resemble fields
  - Declared using JML modifier **model**

    (type T can be any type defined in the logic, e.g., LocSet)
  - Can be declared in interfaces and classes
    - In classes: instance fields by default (use **static** otherwise)
    - In interfaces: static by default (use **instance** otherwise)
  - Used in specifications like fields
- Specification-Only (not accessible from implementation)

Decimal Example: Adding a Model Field

```
class Decimal {
   public static final short PRECISION = (short) 1000;
   /*@ spec_public @*/ private short intPart  = (short) 0;
   /*@ spec_public @*/ private short decPart = (short) 0;
   //@ public model short value;

   /*@ public normal_behavior
     @ ensures value == \old(other.value + value);
     @*/
   public void add(Decimal other) { … }
}
```

Matrix Example: Adding a Model Field

```
public class MatrixImplem {
  //@ public model int[][] matrix;

  private final int x;
  private final int y;
  private int[] matrix_implem;

  //@ ensures \result == matrix[i][j];
  public /*@ pure @*/ int get (int i, int j) {
    return matrix_implem[x * j + i];
  }
}
```

How to connect model fields with their implementation?

Connecting Model Fields and Implementation
Special case using ‘=‘

Use **represents** clause to connect model fields and implementation

```
class A {
  //@ model T field;
  //@ represents field = <JML expression of type T>
}
```

“The value of field is equal to the value of the JML expression”

Model fields

- are not assigned
- value depends on current state (similar to queries)

Examples of Suboptimal Specifications

```
class Decimal {
    public static final short PRECISION = (short) 1000;
    /*@ spec_public @*/ private short intPart = (short) 0;
    /*@ spec_public @*/ private short decPart = (short) 0;

    //@ model short value;
    //@ represents value = intPart * PRECISION + decPart;

    /*@ public normal_behavior
      @ ensures value == \old(other.value) + value;
      @*/
    public void add(Decimal other) { … }
}
```

Connecting Model Fields and Implementation:
The General Case

Often not possible to specify relation between abstraction and implementation using ‘=‘, e.g., matrix example.

To specify a relational correspondence between abstraction and implementation use:

//@ **model** *T field*;

//@ **represents** *field* **\such_that** <*JML* ***Boolean*** *expression*>

Meaning:

“Value of *field* satisfies at any time the provided JML Boolean expression”

Matrix Example: Relating Model and Code

```
public class MatrixImplem {
  //@ public model int[][] matrix;
  private final int x; // nr columns
  private final int y; // nr rows
  private int[] matrix_implem;

  //@ ensures \result == matrix[i][j];
  public /*@ pure @*/ int get (int i, int j) {
    return matrix_implem[x * j + i];
  }
```

We want to express that in any state the following holds:

For any i,j with i >= 0, i < x, j >= 0 and j < y

“matrix[i][j] has the same value as matrix_impl[i*x + j]”

Matrix Example: Relating Model and Code

```
public class MatrixImplem {
  //@ public model int[][] matrix;
  private final int x; // nr columns
  private final int y; // nr rows
  private int[] matrix_implem;
```

```
  /*@ represents matrix \such_that
    @ (\forall int i; i >= 0 && i < x;
    @   (\forall int j; j >= 0 && j < y;
    @       matrix[i][j] == matrix_implem[x * j + i]));
    @*/
```

Property that must hold for model field matrix.
Specifies relation between model field and implementation

```
   //@ ensures \result == matrix[i][j];
  public /*@ pure @*/ int get (int i, int j) {
    return matrix_implem[x * j + i];
  }
```

Represent Clauses: General

- Model field declaration and represents clause do **not** need to be present in the same class or interface
  - **model** field declared in supertype (e.g., abstract class or interface), and
  - **represents** clause added in implementing classes

Demo: ModelSimple

Matrix Example:
One Abstraction — Many Implementations

```
interface Matrix {
  //@ protected model instance int[][] matrix;
  …
}
public class MatrixAs2DimArray implements Matrix {
  private /*@ spec_protected @*/ int [][] matrixArray;
  //@ represents matrix = matrixArray;
}
public class MatrixImplem implements Matrix {
  private /*@ spec_protected @*/ int [] matrix_implem;
  //@ represents matrix \such_that …
```

Represents Clause: Corner Cases

TECHNISCHE UNIVERSITÄT DARMSTADT

Problem: What does it mean if relation is (or at least becomes in some states) unsatisfiable?

For instance:

```
//@ model int value;
//@ represents value \such_that false;
```

(At least) Two possible options to provide a meaning:

1. In a state where the defined relation is unsatisfiable, the value of the model field is unspecified (i.e., except of it being of the declared type nothing is known).
2. Specification is inconsistent. Consequences similar to a precondition being **false**.

KeY supports both options

Configurable in Options | Taclet Options | modelFields

Select showSatisfiability for option 1 and treatAsAxiom for option 2

Software Engineering Group

Representation of Model Fields in the Logic

Model fields depend on state, but cannot be assigned values.

For each model field declaration //@ **model** (**static** | **instance**) *T f*;

declared in a type *S*, a function symbol

T S::$*f*(Heap, *S*)   (instance model field)

T S::$f(Heap)   (static model field)

is added to the signature.

Representation of Model Fields in the Logic
Example

Let self be a program variable of type $\mathrm{Decimal}$.

The term

$$\mathrm{value}(\mathrm{heap}, \mathrm{self})$$

represents the value of the

instance model field value of object self.

KeY:

For readability reasons the term is pretty printed as $\mathrm{self}.\mathrm{value}$

Translating **represents**

For a represents clause declared in type T

//@ **represents** *f* **\such_that** *Prop*

of a model field *f* (of type *S* declared in *U*) the following axiom $Ax^T{}_f$ (h,o) is added

- if showSatisfiability is chosen (model field uninterpreted if *Prop not satisfiable)*

  $(T\text{::exactInstance}(o) = \text{TRUE} \rightarrow$

  $(\text{\textbackslash exists } S \text{ x}; \mathcal{F}(Prop)(h,o)[U\text{::\$}f(h,o) / x]) \rightarrow \mathcal{F}(Prop)(h,o)))$

- otherwise: $T\text{::exactInstance}(o) = \text{TRUE} \rightarrow \mathcal{F}(Prop)(h,o)$

where $\mathcal{F}(Prop)$ is the DL formula resulting from the translation of *Prop*

Using represents in the Calculus

The axiom is introduced by application of the rule

$$\frac{\Gamma, Ax^{T}{}_{f}(h1,o1) \implies \Phi[T::\$f(h1, o1)], \Delta}{\Gamma \implies \Phi[T::\$f(h1, o1)], \Delta}$$

where o1 is of type T and the occurrence of T::$f(h1, o1) in Φ is not below a modality or update.

Demo: ModelSimple (just translation)

GHOST FIELDS AND GHOST VARIABLES

How to specify that acquire() must be called before free() ?

```
public class ResourceCtrl {

  private void acquire() {
     …
  }
                              private void free() {
                                 …
                              }
                            }
```

How to specify that acquire() must be called before free() ?

```
public class ResourceCtrl {
  private boolean acquired = false;
  //@ requires !acquired;
  //@ ensures acquired;
  private void acquire() {
    …
    acquired = true;
  }
  //@ requires acquired;
  //@ ensures !acquired;
  private void free() {
    …
    acquired = false;
  }
}
```

Problems

- Field introduced for specification-only purposes: Clutters implementation
- Might influence program execution

Ghost Variables and Ghost Fields

```
public class A {
  //@ ghost T field = <JML expression of type T>;
  public void m() {
     …
    //@ ghost T localVar = <JML expression of type T>;
    …
    //@ set localVar = <JML expression of type T>;
  }
}
```

- Ghost field/variables are treated exactly like normal fields and local variables.
- Type might be any type (i.e., not just a Java type)
- Special assignment statement for ghost fields/variables
  (right-hand side must be side-effect free; object creation is fine, but use with care)

How to specify that acquire() must be called before free() ?

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class ResourceCtrl {
  //@ private ghost boolean acquired = false;
  //@ requires !acquired;
  //@ ensures acquired;
  private void acquire() {
    …
    //@ set acquired = true;
  }
  //@ requires acquired;
  //@ ensures !acquired;
  private void free() {
    …
    //@ set acquired = false;
  }
}
```


Software Engineering Group

Ghost Fields

TECHNISCHE UNIVERSITÄT DARMSTADT

- Extend the state (not the case for model fields)
- Must be assigned values
  - Increased responsibility for specifier
  - Overriding methods must be aware of ghost elements
- Specification gets operational flavour (no longer just descriptive)
- Seamless integration
  - Like normal fields/variable ➔ no additional concepts needed
  - Easy to use
- Increases kind of properties that can be specified
- Useful for local/loop specifications

Rule of Thumb: Use model fields for abstraction purposes and ghost fields for class local specifications or specific specification properties