Property Specifications - Lecture 2

Assertions
  Basics
  JML
Verification Conditions,
Hoare Logics,
Assertions/Specifications

Precise, formal specifications concerning the behavior of some unit of code
Usually written in a language separate from programming language.
Used for documentation, verification, runtime monitoring, testing
Assertion - Types

**Invariants** (from Wikipedia) - A predicate that will always keep its truth value throughout a specific sequence of operations, is called (an) invariant to that sequence.

- State Invariants
- Loop Invariants

**Pre-conditions/Post-Conditions** - Pre- and post-conditions are constraints that define a contract that an implementation of the operation has to fulfill. A precondition must hold when an operation is called, a postcondition must be true when the operation returns.
Invariants

• Definition
  – An invariant is a property that is always true of an object’s state (when control is not inside the object’s methods).

• Invariants allow you to define:
  – Acceptable states of an object, and
  – Consistency of an object’s state.

//@ public invariant !name.equals("") && weight >= 0;
Pre and Postconditions

• Definition
  – A method or function *precondition* says what must be true to call it.
  – A method or function *normal postcondition* says what is true when it returns normally (i.e., without throwing an exception).
  – A method or function *exceptional postcondition* says what is true when a method throws an exception.

//@ signals (IllegalArgumentException e) x < 0;
Relational Model of Methods

- Can think of a method as a relation:
  Inputs ↔ Outputs

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

precondition

postcondition

Input

Output
```
Assertions – How Used

Program annotated with invariants, pre/post-conditions

- Verification condition generator
- Source to source Compiler
- Test generation compiler

- Theorem Prover
- Runtime Monitor
- Testing Environment
Relationship to Temporal Logic

Temporal logic predicates are same as assertion/specification predicates.
Assertion specifications are local with respect to some code unit (composed by Hoare logic rules)
Temporal logic predicates apply to states during execution of some code unit and are defined on paths or structures of paths
Relationship to Temporal Logic

Temporal logic properties for code units can be composed into properties for larger code units.

System level temporal logic can be decomposed into component level properties.

Component level temporal logic properties can be translated into invariants, preconditions and postconditions.
Relationship to Temporal Logic

- System Level Temporal Logic Properties
- Temporal Logic Properties for Components
- Environment Specifications
- Invariants, Preconditions and Postconditions for Components
- Automatable

Automatable?
Temporal Logic Composition

System
  Properties and Environment
  Component E
    Properties and Environment
      Component A
      Properties and Environment
      Component B
      Properties and Environment
      Component C
      Properties and Environment
      Component D
      Properties and Environment
Decomposition of I/P/P Specifications

System Level Preconditions == Environment Specifications

Automatable?

Component Level Preconditions

Component Level Invariants and Postconditions
Composition of I/P/P Specifications

System Level Preconditions == Environment Specifications

Component Level Preconditions

Component Level Invariants and Postconditions
Composition of I/P/P Specifications

Hoare Rules for composition should apply. Automatable??

System
Invariants, P/P
Conditions

Component E
Invariants, P/P
Conditions

Component E
Invariants, P/P
Conditions

Component A
Invariants, P/P
Conditions

Component B
Invariants, P/P
Conditions

Component C
Invariants, P/P
Conditions

Component D
Invariants, P/P
Conditions
Tools for JML-Based Verification

JML Annotated Java

- Web pages
- jmltest
- Unit tests
- jmlunit
- Class file
- jmic
- XVP
- Model checking
- Bogor
- Correctness proof
- JACK, Jive, Krakatoa, KeY, LOOP
- Data trace file
- Daikon
- Warnings
- ESC/Java2
Java Modeling Language

Illustrate Assertions with Java Modeling Language

– Hoare-style (Contracts).
– Method pre- and postconditions.
– Invariants.
Java Modeling Language

• JML Annotations/Assertions
• Top-level in classes and interfaces:
  – invariant
  – spec_public
  – nullable
• For methods and constructors:
  – requires
  – ensures
  – assignable
  – pure
public class ArrayOps {
    private /*@ spec_public @*/ Object[] a;
    //@ public invariant 0 < a.length;
   /*@ requires 0 < arr.length; @*/
    @ ensures this.a == arr;
    //@*
    public void init(Object[] arr) {
        this.a = arr;
    }
}
spec_public, nullable, and invariant

- **spec_public**
  - Public visibility.
  - Only public for specification purposes.

- **nullable**
  - Field (and array elements) may be null.
  - Default is **non_null**.

- **invariant** must be:
  - True at end of constructor.
  - Preserved by each method.
requires and ensures

• **requires** clause:
  – Precondition.
  – Obligation on callers, after parameter passing.
  – Assumed by implementor.

• **ensures** clause:
  – Postcondition.
  – Obligation on implementor, at return.
  – Assumed by caller.
assignable and pure

- **assignable**
  - Frame axiom.
  - Locations (fields) in pre-state.
  - New object fields not covered.
  - Mostly checked statically.
  - Synonyms: modifies, modifiable

- **pure**
  - No side effects.
  - Implies assignable \ nothing
  - Allows method’s use in specifications.
Redundant Clauses

• ensures_redundantly
  – Alerts reader.
  – States something to prove.
  – Must be implied by:
    • ensures clauses,
    • assignable clause,
    • invariant, and
    • JML semantics.

• Also requires_redundantly, etc.
Formal Specifications

• Formal assertions are written as Java expressions, but:
  – Can’t have side effects
    • No use of =, ++, --, etc., and
    • Can only call pure methods.
  – Can use some extensions to Java:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\result</td>
<td>result of method call</td>
</tr>
<tr>
<td>a ==&gt; b</td>
<td>a implies b</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>b implies a</td>
</tr>
<tr>
<td>a &lt;=&gt; b</td>
<td>a iff b</td>
</tr>
<tr>
<td>a &lt;= !=&gt; b</td>
<td>!(a &lt;=&gt; b)</td>
</tr>
<tr>
<td>old(E)</td>
<td>value of E in the pre-state</td>
</tr>
</tbody>
</table>
BoundedStack’s Data and Invariant

```java
public class BoundedStack {
    private /*@ spec_public_nullable @*/ Object[] elems;
    private /*@ spec_public @*/ int size = 0;
    //@ public invariant 0 <= size;
   /*@ public invariant elems != null @ && (forall int i; @ size <= i && i < elems.length; @ elems[i] == null); @*/
```
BoundedStack’s Constructor

@ requires 0 < n;
@ assignable elems;
@ ensures elems.length == n;
@*

public BoundedStack(int n) {
  elems = new Object[n];
}
BoundedStack’s push Method

BoundedStack’s push Method
/*@ requires size < elems.length1;
@ assignable elems[size], size;
@ ensures size == \old(size+1);
@ ensures elems[size1] == x;
@ ensures_redundantly
@ (\forall int i; 0 <= i && i < size1;
@ elems[i] == \old(elems[i]));
@*/

public void push(Object x) {
    elems[size] = x;
    size++;
}
BoundedStack’s pop Method

BoundedStack’s pop Method
/*@ requires 0 < size;
@ assignable size, elems[size1];
@ ensures size == \old(size1);
@ ensures_redundantly
@ elems[size] == null
@ && (\forall int i; 0 <= i && i < size1;
@ elems[i] == \old(elems[i]));
@*/
public void pop() {
    size;
    elems[size] = null;
}
BoundedStack’s top Method

BoundedStack’s top Method
/*@ requires 0 < size;
@ assignable \nothing;
@ ensures \result == elems[size1];
@*/
public /*@ pure @*/ Object top() {
    return elems[size1];
}
}