CIS 771: Software Specifications

Lecture 18: Specifying and Checking Partial Properties of Java Code

Extended Static Checking

- Theorem-prover based technology for statically reasoning
  - about annotated programs
  - with respect to run-time errors
- Long-term project at DEC/Compaq
  - Started in early 90s for Modula-3
  - Now supports Java
Philosophy

- Can’t check everything
- Sacrifice generality and thoroughness for speed
- ESC is incomplete
  - Error reports may not be real defects
- ESC is unsound
  - Defective programs may not be reported
- Provide a useful tool with little investment
  - Without any specifications
  - Incremental gain for added specifications
- Focus on common or hard to find errors
  - Null pointer dereference
  - Lock related
ESC/Java Annotations

- Embedded in comments
- Single line comments
  ```java
  // @ ...
  ```
- Multi-line comments
  ```java
 /*@ */
  ```

Classes of errors

- Detects potential run-time errors
- JVM exceptions
  - `NullPointerException`, `IndexOutOfBoundsException`, `ClassCastException`, `ArrayStoreException`, `ArithmeticException`, `NegativeArraySizeException`
  - Errors reported even if exception is caught
- Programmer annotations
  - invariants, precondition, postconditions, and assertions
ESC/Java Expressions
- ESC/Java specifications have as a core language Java’s boolean expression syntax
- ESC restricts some legal Java expressions
  - Side-effect free Java expressions with no calls
  - E.g., cannot use “=”, “++”, “new”
- ESC introduces some special Java expressions
- ESC/Java specifications are less expressive than OCL

ESC/Java
- Expression evaluation errors
  - e.g., NullPointerException
  - Do not give rise to error reports
  - Yield undefined values \(\Rightarrow\) failure to prove annotation in which expression is embedded
ESC/Java Operators (excerpts)

- `\type(E)` : construct spec type from Java type
- `\typeof(E)` : returns dynamic type of E
- `\elemtype(E)` : returns element type of an array
- `S <: T` : S is a subtype of T (or equal)

```java
void storeObject(T[] a, int I, T x) {
    a[i] = x;
}
```

What happens if the dynamic type of a is “S[]” where S is a subtype of T?

`ArrayStoreException`

ESC/Java Operators (excerpts)

```java
void storeObject(T[] a, int I, T x) {
    a[i] = x;
}
```

There are multiple ways to specify that this cannot happen:

```java
//@ requires \elemtype(\typeof(a)) == \type(T);
//@ requires \typeof(a) == \type(T[]);
//@ requires x == null ||
//@ \typeof(x) <: \elemtype(\typeof(a));
```
ESC/Java Operators (excerpts)

- \old : like "@pre" in OCL (see 3.2.15)
- \result : return value
- \fresh(e) : e is non-null in post-state and unallocated in pre-state
- ==> : implies
- \exists : existential quantification
- \forall : universal quantification
- \nonnullelements : custom \forall

- A rich set of lock querying operators

---

ESC/Java Operators (excerpts)

- (\forall T V; E)
  - E is true for all substitutions of values of T bound to variables V
  - For reference types T, quantifier ranges over allocated instances (excluding null)
  - For integral types T, values range over mathematical integers NOT computer-based integers
- (\exists T V; E)
  - E is true for some substitution of values of T bound to variables V
- \nonnullelements(A)
  - A != null &&
    - (\forall int i; 0 <= i && i < A.length ==> A[i] != null)
Modifies

- Modifies clauses used by caller
- Implementing method is not checked to see that it does not modify unmentioned variables
- $\text{\textbackslash ol d}(f)$ refers to post-state value (!)
  - Unless you include: $\text{\textbackslash modifies f}$
- Explanation is a bit subtle
  - (see the manual)

For You to Do (pause here)

Consider an implementation of the “academia” system with the following class that implements the attributes and associations of “Course” in our OCL model:

class Course {
    String name;
    int number;
    Instructor teacher;
    Student[] enrolled;
    int numEnrolled;
    Student[] waiting;
    int numWaiting;
    Course[] prereqs;
    numPrereqs;
}
For You to Do (pause here)

Express the following OCL invariants in ESC/Java (assume that you are in the context of the Course class):

-- NoWaitingUnlessEnrolled:
-- .. no one is waiting for a course unless someone is enrolled.
context c:Course
inv NoWaitingUnlessEnrolled:
  c.waitList->notEmpty implies c.enrolled->notEmpty

-- NoSelfPrerequisite
-- .. no course has itself as a prerequisite
context c:Course
inv NoSelfPrerequisite:
  c.prerequisites->excludes(c)

Annotation Based Checking

- ESC/Java supports many forms of annotation that we are familiar with
- Data annotations
  - invariant
- Method annotations
  - requires [pre-condition]
  - modifies [frame-condition]
  - ensures [post-condition]
  - alsoRequires, ... [for sub-typing]
  - ensures [for exceptions]
ESC Specific Annotations

- Basic annotations (pragmas)
  - nowarn, assert, assume, unreachable
- Data annotations
  - non_null, axiom, loop_invariant
- Abstraction support
  - spec_public, ghost, set
- Synchronization specific pragmas

For You to Do (pause here)

Consider an implementation of the “academia” system with the following class that implements the attributes, associations and operations of the “Course” and “Student” in our OCL model:

class Course {
class Student {
    String name;
    int number;
    Instructor teacher;
    Student[] enrolled;
    int numEnrolled;
    Student[] waiting;
    int numWaiting;
    Course[] prereqs;
    numPrereqs;

    public void addPreReq(Course c) {...}
    }
    void dropCourse(Course c) {...}
    void newId(n : Integer) {...}
    }
}
For You to Do (pause here)

Express the following OCL operation specifications in ESC (assume that you are in the context of the Course class):

```esc
-- newId pre/post-conditions
-- .. pre-conditions
-- - n is greater than 100
-- .. post-conditions
-- - Id object is new and its number is equal to the supplied parameter
context Student::newId(n: Integer)
pre GE100: n >= 100
post Newld: id.oclIsNew
post IdNumber: id.number = n

-- dropCourse pre/post-conditions
-- .. pre-conditions
-- - currently taking course
-- .. post-conditions
-- - taking same as old taking minus given course
context Student::dropCourse(c: Course)
pre NowTaking: taking->includes(c)
post NotTaking: taking = taking@pre->excluding(c)
```

Checked Annotations

- ESC Java will analyze the program to see if the annotations hold
- Local annotations
  - `assert` E
  - `unreachable` [assert false]
  - `loop_invariant E` [do, while loops]
- Method annotations
  - `requires, ensures`
    [specific method entry and exit]
- Global annotations
  - `invariant E` [every method entry and exit]
  - `nonnull V` [checked at every assignment]
Unchecked Annotations

- User supplied information about program behavior
- Suppress warnings at a statement
  - `nowarn` [parameterized by error]
- Assumptions about data
  - `assume E` [local assumption]
  - `axiom E` [global assumption]

Unchecked Annotations

- Are the main source of unsoundness in ESC Java
  - The user can tell the system that something is true about a variable when it is not the case
- Are the main mechanism for reducing spurious error reports
  - Due to incompleteness of theorem-prover
- Sometimes ESC Java warns you that assumptions may invalidate a result
Annotation Philosophy

- Include checked annotations in your program
  - That express the properties you want the code to have
- Include assumptions only when you obtain an error report
- When possible use checked annotations rather than assumptions
  - e.g., invariant versus axiom

Modular Checking

- ESC Java analyzes programs one method at a time
  - Performance is improved
  - Accuracy is a problem
- No information about calling context
  - Possible parameter or field values
- Does not analyze called methods
  - Use annotations to represent effects of method call
Spec Files

- ESC/Java uses source annotations
- What if the sources are not available?
  - e.g., libraries
- Generate simple spec from .class files
  - Constraints that enforce proper typing
- Use .spec files
  - Routine bodies may be omitted via “;” or “{ }”
  - Comes with specs for Java libraries

java.util.Stack .spec file

```java
public class Stack extends Vector {

    //@ requires typeof(item) <: elementType || item==null
    //@ requires containsNull || item!=null
    //@ modifies elementCount
    //@ ensures elementCount == old(elementCount)+1
    //@ ensures \result==item
    public Object push(/*@ non_null*/ Object item) {    }

    //@ requires elementCount > 0
    //@ modifies elementCount
    //@ ensures elementCount == old(elementCount)-1
    //@ ensures typeof(\result) <: elementType || \result==null
    //@ ensures !containsNull ==> \result!=null
    public synchronized Object pop() {    }

}
```
For You to Do (pause here)

What ESC/Java annotations would you use for the “top” method of the stack? i.e., the method that returns the top element, but does not “pop” it

Try to be provide annotations at the same level of detail as the one’s just presented.