ESC/Java

CS195
Lecture 17

Philosophy

- ESC
  - Stands for “Extended Static Checking”
- Extended over types
  - Catches more errors than typical type systems
- But not as rich as full verification
  - ESC/Java developers had verification background

A Digression on Verification

- Full verification requires
  - A complete specification
  - Theorem-proving machinery to show program meets specification
- This is much heavier than anything we have studied

A Verification Example

- Example: Sorting an array A of length n
- Convention: A is array on input, A’ on output
- Array is sorted in place
- Assumes 1-based indexing
- Element type has built-in operation

Extended Static Checking Again

- What sort of extended, but limited, properties does it check?
  - Index out of bounds
  - Null pointer dereferences
  - Race conditions
  - Deadlocks
  - . . .
- The properties we’ve been looking at . . .

Why the Name ESC/Java?

- Originally targeted at Modula-3
  - A pre-Java language
  - Had some influence on Java
- Retargeted after Java’s success
**Goals**

- Catch errors early
  - I.e., statically
- In a realistic setting
  - Multithreaded programs
  - Multiple compilation units
  - User-defined datatypes
  - With straightforward user annotations
- In contrast to earlier systems . . .

---

**In Increasing Sophistication & Expense**

- Syntactic Techniques
  - Lclint, low end of FindBugs
- Type systems
- Dataflow-based systems
  - CQual, most of Metal
- Model Checking
  - BLAST
- Theorem-proving
  - ESC/Java

---

**But . . .**

- ESC/Java is not a verification system
- False negatives
  - May miss some bugs it is designed to find
- False positives
  - May report some "bugs" that don't exist

---

**The Fix**

- Clearly no rational number has a 0 denominator
  - The denominator should always be non-zero
- This is a class invariant
  - A predicate that is always true about the class
- ESC/Java allows invariants to be declared

---

**Example**

```java
class Rational {
    int num, denom;
    Rational(int n, int d) {
        num = n;
        denom = d;
    }
    int toInteger() {
        return num/denom;
    }
}
```

```java```
public static void main(String[] a) {
    int n = readInt();
    int d = readInt();
    if (d == 0) return;
    Rational r = new Rational(d,n);
    print(r.toInteger());
}
```
Annotations

- Syntax
  - Made to look like comments
  - Ugly, but
    - Avoids compiler modifications
    - Commonly used
- Rich predicate language
  - Can go far beyond this simple example

Example

```java
public static void main(String[] a) {
    int n = readInt();
    int d = readInt();
    if (d == 0) return;
    Rational r = new Rational(d, n);
    print(r.toInteger());
}
```

Pre- and Post-Conditions

- A pre-condition is a fact that must hold before
  a function (method)
  - Here, the class invariant can only be established if
    input \( d \neq 0 \)
- A post-condition is a fact that must hold after
  a function (method)
  - Here, the post-condition is the class invariant

Example

```java
public static void main(String[] a) {
    int n = readInt();
    int d = readInt();
    if (d == 0) return;
    Rational r = new Rational(d, n);
    print(r.toInteger());
}
```

Why Pre- and Post-Conditions?

- Modular checking
  - Checking each procedure in isolation
  - Assume pre-condition holds
  - Verify
    - post-condition holds
    - Any pre-conditions of called methods hold
- If everything checks, then pre- and post-
  conditions are indeed correct

Another Example

```java
int extractMin() {
    int min = Integer.MAX_VALUE;
    for (int i = 1; i <= size; i++) {
        if (elem[i] < min) {
            min = elem[i];
            minIndex = i;
        }
    }
    size--;
    elem[minIndex] = elem[size];
    return min;
}
```
The Fix

- We probably don’t want the constructor to be called with a null argument.
- Add a precondition . . .

Another Example

```java
class Bag {
    int size;
    int[] elem;
    Bag(int[] input) {
        size = input.length;
        elem = new int[size];
        System.arraycopy(input, 0, elem, 0, size);
    }
    int extractMin() {
        int min = Integer.MAX_VALUE;
        int minIndex = 0;
        for (int i = 1; i <= size; i++) {
            if (elem[i] < min) {
                min = elem[i];
                minIndex = i;
            }
        }
        size--;
        elem[minIndex] = elem[size];
        return min;
    }
}
```

Warning: possible null dereference

Another Example

```java
class Bag {
    int size;
    int[] elem;
    Bag(int[] input) {
        size = input.length;
        elem = new int[size];
        System.arraycopy(input, 0, elem, 0, size);
    }
    int extractMin() {
        int min = Integer.MAX_VALUE;
        int minIndex = 0;
        for (int i = 1; i <= size; i++) {
            if (elem[i] < min) {
                min = elem[i];
                minIndex = i;
            }
        }
        size--;
        elem[minIndex] = elem[size];
        return min;
    }
}
```

Warning: possible null dereference

Warning: Array index possibly too large

A Bug!

- The loop can iterate up to elem[size] - But Java arrays are zero-based - elem[size] is out of range
- Correct the line of code . . .

Hmmm . . .

- The problem is more complicated than we thought
- There is nothing that forces size and the length of elem to be the same
- Add a class invariant . . .
Another Example

```java
class Bag {
  int size;
  int[] elem;

  Bag(int[] input) {
    size = input.length;
    elem = new int[size];
    System.arraycopy(input, 0, elem, 0, size);
  }
  int extractMin() {
    int min = Integer.MAX_VALUE;
    int minIndex = 0;
    for (int i = 0; i < size; i++) {
      if (elem[i] < min) {
        min = elem[i];
        minIndex = i;
      }
    }
    size--;
    elem[minIndex] = elem[size];
    return min;
  }
}
```

//@ input != null
//@ elem != null
//@ invariant 0 <= size <= elem.length

---

What's Wrong?

- We know size >= 0 on entry to the method from the class invariant
- But the line size--; implies size may be negative on this line of code
- Fix by requiring size > 0
  - Guarantees size >= 0 at point of dereference

Done!

- No more warnings are reported
  - ESC/Java finds no more potential bugs
- ESC/Java has found all of the bugs that it can

The Rest of the Lecture

- More philosophy
  - The rationale behind the design
- More design
  - Kinds of annotations
- Sketch of system architecture
  - A bit on how it works
- Experience

Why Annotations?

- Currently, most research avoids annotations
  - Try to infer needed information if possible
  - Deal with simpler properties where inference is tractable
- But ESC/Java emphasizes annotations
  - Believe annotations document important facts
  - But better than documentation
    - Because annotations are checkable
Expressiveness

• ESC/Java deals with things we have seen
  - E.g., Null dereferences

• And things we haven’t
  - Array references
    - Requires symbolic arithmetic
  - Deadlocks in multithreaded systems
    - Not discussed in these notes

• Emphasis on finding sophisticated bugs

Kinds of Annotations

• There are four classes of annotations
  - Lexical
  - Statement
  - Declaration
  - Modifier

• Different classes appear in different syntactic contexts
  - See the manual

You cannot mix annotations of different classes in the same specification comment!

Examples

• Requires
  States a pre-condition for a method

• Ensures
  States a post-condition for a method

• Exsures
  States a post-condition for abnormal termination of a method

More Examples

• Loop_invariant
  State a fact that must be true on every iteration of a loop

• Assert
  State a fact that must be true at a particular program point

• Unreachable
  Equivalent to assert(false)

Escapes

• Modifies
  - List variables/fields the method may modify
  - Assumed to be true---not checked!
    - A source of unsoundness

• Nowarn
  - Supress warnings

• Assume
  - Believe some fact even if it can’t be proven

Design Issues for Java

• When are invariants maintained?
  - Not all the time
  - Methods often temporarily violate invariants

• A design decision
  - Default behavior affects number of annotations
  - Not an easy problem...
    - ESC/Java checks/assume invariants at method boundaries
Method Override

- Overridden methods inherit annotations of the parent
- Can add restrictions to the postcondition
  - Like subtyping

How It Works

- Steps
  - Reduce program to a simple intermediate language
    - Very close to a logic
  - Generate verification conditions
    - Really, translate program into logical formula
    - Together with invariants, assertions, etc.
  - Feed the VC’s to a theorem prover
    - Prover says “yes”, program is OK
    - Prover says “no”, generate counterexample

A Little More Detail

- Consider a method with a
  - Precondition A
  - Body B
  - Postcondition C
- Translate B into a formula $B'$
- Try to satisfy the formula $\neg((A \land B) \Rightarrow C)$
  - Can't satisfy: program is safe
  - Can satisfy: satisfying assignment is counterexample

Pros and Cons

- Experience is that annotation burden is significant
  - Especially for legacy code
  - Easier if writing new code with annotations
- Up to 10% of program is annotations
  - Most common annotations are the simplest
    - E.g., non-null
- Time-consuming to check and remove each warning

Houdini

- These observations led to an effort to reduce annotation burden
- Idea
  - Write a system to infer annotations
  - Use feedback from error messages to infer what annotation to add
  - Straightforward when annotations involve reasoning only about dataflow
  - Harder for more complex annotations

Summary

- ESC/Java is interesting
  - Handles many familiar issues
  - And some unfamiliar ones
    - E.g., array bounds checking
- Has not found widespread use
  - Presumably because of annotation burden
  - But nevertheless an interesting design point