Checking Program Properties with ESC/Java

17-654/17-765
Analysis of Software Artifacts
Jonathan Aldrich

ESC/Java

- A checker for Java programs
  - Finds null pointers, array dereferences…
  - Checks Hoare logic specifications
    - Expressed in Java Modeling Language (JML)
- Goal:
  - Find errors
  - Increase confidence in correctness
    - Unlike a Hoare Logic proof, not a guarantee of correctness
- Developed at Compaq SRC in the 90s
  - Now open sourced as ESC/Java 2

ESC/Java Uses JML Specifications

```
/*@ requires len >= 0 && array != null && array.length == len;
  @
  @ ensures \result == (\sum int j;  0 <= j && j < len;  array[j]);
  @*/
int sum(int array[], int len) {
    int sum = 0;
    int i = 0;
    /*@ loop_invariant sum == (\sum int j;  0 <= j && j < i;  array[j]); */
    while (i < len) {
        sum = sum + array[i];
        i = i + 1;
    }
    return sum;
}
```


Modular Checking with Hoare Logic

$$\frac{p : \{P\}\ S\ \{Q\}}{\{P\}\ p();\ \{Q\}}$$

- Simple procedure rule for Hoare Logic
  - Assumes all variables are global
- Can verify p independent of caller
  - No need to look at S when verifying caller of p
    - Just use post-condition and check precondition
  - No need to look at caller of p to check S
    - Just assume pre-condition

Modular Checking Example (1)

procedure multiply

{ n > 0 }
result := 0;
i := 0;
*{ invariant: result = m*i && 0 ≤ i ≤ n && n > 0 }*
while i < n do
add();
i := i + 1;
*{ result = m*n }*

procedure add

*{ result = m*i }*
result = result + m
*{ result = m*i + m }*

- Loop precondition
  n > 0
  *⇒ 0 = m * 0 && 0 ≤ 0 ≤ n && n > 0*
- Precondition of add
  *result = m*i && 0 ≤ i ≤ n && n > 0 && i < n*
  *⇒ result = m*i*
- Re-establish loop invariant
  *result = m*i + m*
  ⇒ result = m(i+1) && 0 ≤ i+1 ≤ n && n > 0*
- Does not hold!
  - Need to strengthen add specification

Modular Checking Example (2)

procedure multiply

{ n > 0 }
result := 0;
i := 0;
*{ invariant: result = m*i && 0 ≤ i ≤ n && n > 0 }*
while i < n do
add();
i := i + 1;
*{ result = m*n }*

procedure add

*{result = m*i && 0 ≤ i < n && n > 0 }*
result = result + m
*{result = m*i+m && 0 ≤ i < n && n > 0 }*

- Loop precondition
  n > 0
  *⇒ 0 = m * 0 && 0 ≤ 0 ≤ n && n > 0*
- Precondition of add
  *result = m*i && 0 ≤ i ≤ n && n > 0 && i < n*
  *⇒ result = m*i && 0 ≤ i < n && n > 0*
- Re-establish loop invariant
  *result = m*i+m && 0 ≤ i < n && n > 0*
  ⇒ result = m(i+1) && 0 ≤ i+1 ≤ n && n > 0*
- Establish postcondition
  *result = m*i && 0 ≤ i ≤ n && n > 0 && i ≥ n*
  *⇒ result = m*n*

A Better Rule

$$\frac{p : \{P\}\ S\ \{Q\} \quad \text{p does not modify variables in I}}{\{P \text{ and } I\}\ p();\ \{Q \text{ and } I\}}$$

- Increases independence of functions
  - No need to specify invariant of caller in specification of P

Modular Checking Example (3)

procedure multiply
{ n > 0 }
result := 0;
i := 0;
*{ invariant: result = m*i*
&& 0 ≤ i ≤ n && n > 0 }
while i < n do
add();
i := i + 1;
*{ result = m*n }*

// modifies result
procedure add
*{ result = m*i }*
result = result + m
*{ result = m*i+m }*

Add specification still mentions i.

- Verification condition for loop:

*{ invariant: result = m*i*
&& 0 ≤ i ≤ n && n > 0 }
while i < n do
*{result = m*i && 0 ≤ i ≤ n*
&& n > 0 && i < n}
*{result = m*i && 0 ≤ i+1 ≤ n && n>0}*
add();
{result = m(i+1) && 0 ≤ i+1 ≤ n && n>0}*
i := i + 1;
*{result = m*i && 0 ≤ i ≤ n && n>0 }*

Modular Checking Example (4)

procedure multiply

```
{ n > 0 }
result := 0;
i := 0;
{ invariant: result = m*i
&& 0 ≤ i ≤ n && n > 0 }
while i < n do
    add();
    i := i + 1;
{ result = m*n }
```

```
// modifies result
int add(int n, int m)
    { true }
    return n + m
    { \result = n+m }
```

- Verification condition for loop:

```
{ invariant: result = m*i
&& 0 ≤ i ≤ n && n > 0 }
while i < n do
   {result = m*i && 0 ≤ i ≤ n
     && n > 0 && i < n}
   {result = m*i && 0 ≤ i+1 ≤ n && n>0}
   add();
   {result = m*(i+1) && 0 ≤ i+1 ≤ n && n>0}
   i := i + 1;
   {result = m*i && 0 ≤ i ≤ n && n>0 }
```

Add specification still mentions i.
A better spec works the same way but uses variable renamings.
Used in ESC/Java; formal semantics beyond scope of this course

Demo: multiply in ESC/Java

Demo: SimpleSet in ESC/Java

ESC/Java's Limitations

- Does not check for some errors
  - Infinite loops, arithmetic overflow
  - Functional properties not stated by user
  - Non-functional properties
- May miss some errors
  - Unsound: describes an analysis that can miss errors
  - Only checks one iteration of loops
  - @modifies is unchecked
  - Assumptions about invariants in referred-to objects
  - Several others as well!

Loops in ESC/Java

- The loop:

  //@ loop_invariant *E*;
  while (*B*) {
    S
  }

- Is treated as:

  //@ assert *E*;
  if (*B*) {
    S
    //@ assert *E*;
    //@ assume !B;
  }

- Can optionally increase # iterations with *-loop n*

ESC/Java's Limitations (con't)

- May report false positives
  - Often can be solved with an extra precondition or invariant
  - Spurious warnings can also be disabled

ESC/Java Tradeoffs

- Attempts to automate Hoare-logic style checking
- Benefits
  - Easier than manual proof
- Drawbacks
  - Unsound
  - Still quite labor-intensive
- Applicability
  - Checking of critical code
    - When it's worth the extra effort to get it right
    - When you can't do a complete Hoare-logic proof
  - Still must use other analysis techniques
    - ESC/Java is unsound
    - The spec must also be validated!