Automated Testing&Verification

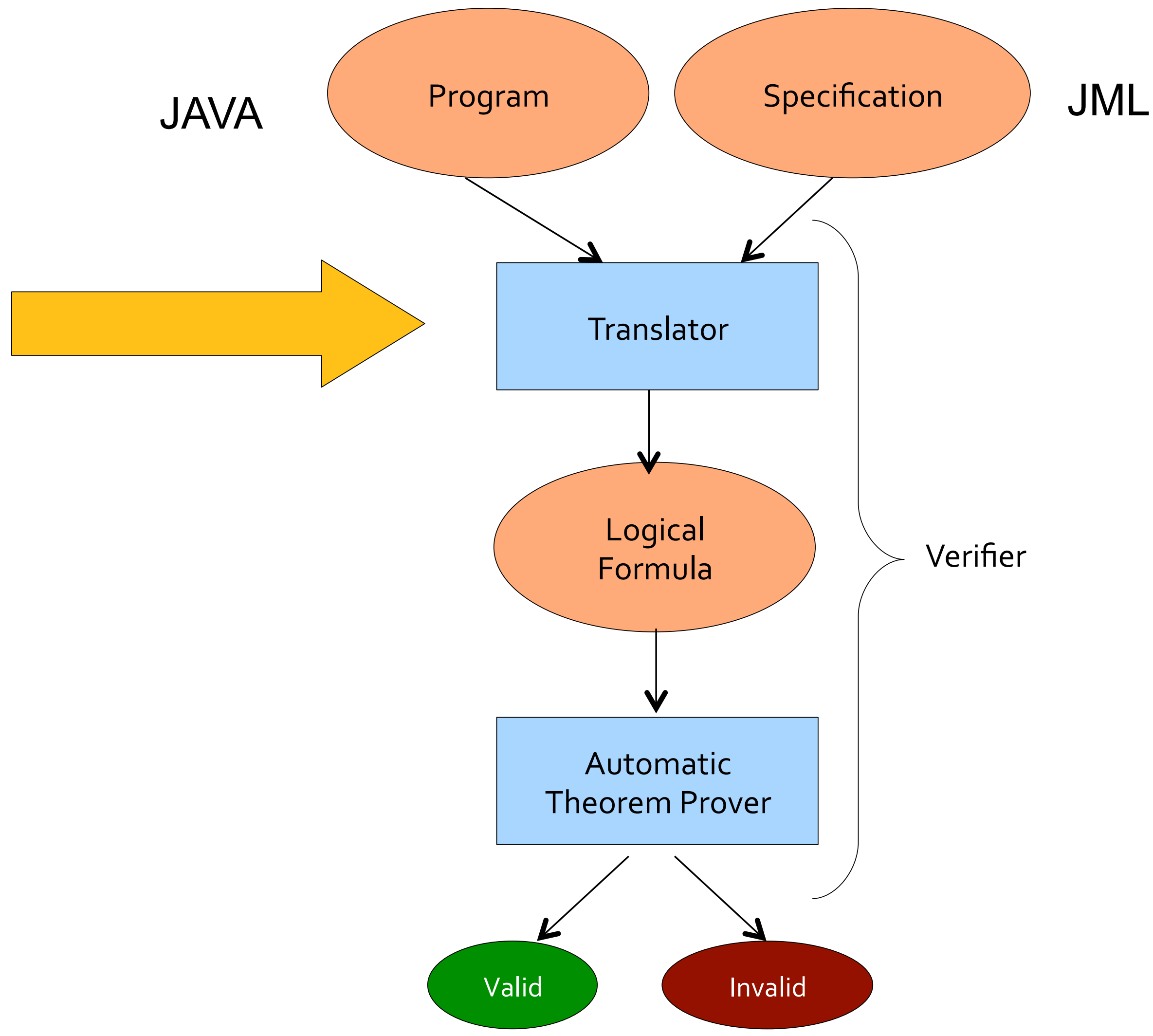
Verification Conditions

Juan Pablo Galeotti, Alessandra Gorla, Andreas Rau
Saarland University, Germany

Course Grading

- 30% projects (10% each)
  - At least 50% threshold for exam admittance
  - Groups of 2
- 70% final exam (see course schedule)
  - Closed-book
  - Allowed: one A4 page (both sides!)

Verifying Programs



Translating a program to a formula

- Both program and its contract must be translated into the same formalism
- In order to do this, we need some way of encoding the program behavior in the logic we are using.
- Formal semantics for the programming language is needed:
  - Several approaches:
    - **Operational**: Simulation of the program execution in a "virtual" machine.
    - **Denotational**: Program is seen as mathematical function
    - **Axiomatic**: Program is seen as set of axioms and inference rules.

Axiomatic Semantics

- Hoare Triples
- Rule system aimed at the verification of imperative programs
- **Partial Correctness**: {A} `program` {B} if
  - Program starts in a state that satisfies **A**
  - In case exection finishes, **B** holds in final state.

A simple imperative language

- Atomic statements
  - **Skip:** `skip`
  - **Assigment**: `x:= E`
- Control-flow statements
  - **Sequential:** `S1; S2`
  - **Conditional:** `if (cond) {S1} else {S2}`
  - **Iteration:** `while (cond) {S}`

Hoare Rules

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}$$

$$\frac{\{A\}\ s1\ \{C\} \quad \{C\}\ s2\ \{B\}}{\{A\}\ \texttt{s1;s2}\ \{B\}}$$

$$\frac{\{A \mathbin{\&\&} cond\}\ s1\ \{B\} \quad \{A \mathbin{\&\&} !cond\}\ s2\ \{B\}}{\{A\}\ \texttt{if(cond) \{s1\} else \{s2\}}\ \{B\}}$$

$$\frac{\{A \mathbin{\&\&} cond\}\ body\ \{A\} \qquad (A \mathbin{\&\&} !cond) \Rightarrow B}{\{A\}\ \texttt{while(cond) \{body\}}\ \{B\}}$$

Hoare rules: assignment

Forward rule:

$$\{ A \}\ \mathtt{x := E}\ \{\exists x' | A[x \rightarrow x']\ \&\&\ x == E[x \rightarrow x']\}$$

- Intuition: x′ is the previous value of x. (\old(x))
- Example:

$$\{ x>=3 \}\ \mathtt{x := x+2}\ \{\exists x' | (x>=3)[x \rightarrow x']\ \&\&\ x == (x+2)[x \rightarrow x']\}$$

$$\{ x>=3 \}\ \mathtt{x := x+2}\ \{\exists x' | x'>=3\ \&\&\ x == x'+2\}$$

$$\{ x>=3 \}\ \mathtt{x := x+2}\ \{\exists x' | x'>=3\ \&\&\ x-2 == x'\}$$

$$\{ x>=3 \}\ \mathtt{x := x+2}\ \{x-2>=3\}$$

$$\{ x>=3 \}\ \mathtt{x := x+2}\ \{x>=5\}$$

Hoare rules: assignment

Backward rule:

$$\{ B[x \rightarrow E] \} \; \mathbf{x := E} \; \{B\}$$

- Intuition: Given **B(x)**, then **B(E)** should hold if **x:=E**

  - Example:

$$\{?\} \; \mathbf{x := x+2} \; \{x>=5\}$$
$$\{x>=5[x \rightarrow x+2]\} \; \mathbf{x:=x+2} \; \{x>=5\}$$
$$\{x+2>=5\} \; \mathbf{x:=x+2} \; \{x>=5\}$$
$$\{x>=3\} \; \mathbf{x:=x+2} \; \{x>=5\}$$

Verifying program behaviour

- **Verification condition** (VC)
  - A logical formula such that its validity means some aspect of program correctness

- Given the following Hoare triple:

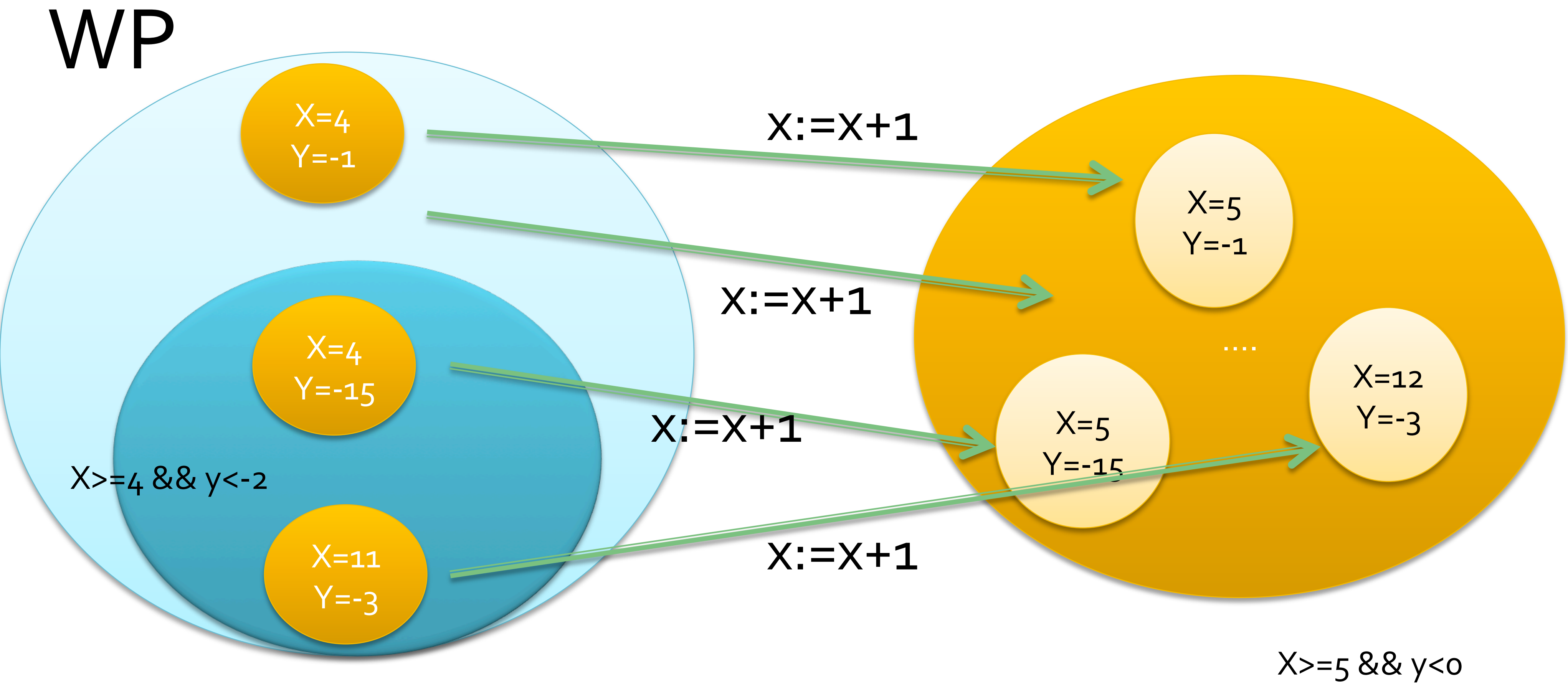
{ x>= 4 && y<-2}

x := x +1

{ x>=5 && y<0}

Program states

{ x>= 4 && y<-2}
x := x +1
{ x>=5 && y<0}



Proving correctness

{Weakest precondition (WP)}

x:=x+1

{ x>=5 && y<0}

- Since states(x>=4 && y<-2) \subsetof states(WP), then we have that

{x>=4 && y<-2}

x:=x+1

{x>=5 && y<0}

Calculating the Weakest Precondition

- $WP(\texttt{skip}, B) =_{def} B$
- $WP(\texttt{x:=E}, B) =_{def} B[x \rightarrow E]$
- $WP(\texttt{s1;s2}, B) =_{def} WP(s1, WP(s2, B))$
- $WP(\texttt{if(E)\{s1\}else\{s2\}}, B) =_{def}$

  E=> WP(s1,B) &&

  !E => WP(s2,B)

Verification Condition

- Given the following Hoare triple

  {Pre}
  Program
  {Post}

- The following formula is a Verification Condition (VC) for the triple:
  - **Pre => WP(Program, Post)**
- We call this a "backward" VC (in constrast with "forward" VC)

Example

- WP(`skip`, B) $=_{def}$ B
- WP(`x:=E`, B) $=_{def}$ B[x→E]
- WP(`s1;s2` , B) $=_{def}$ WP(s1, WP(s2, B))
- WP(`if(E){s1}else{s2}`, B) $=_{def}$
  E=> WP(s1,B) &&
  !E => WP(s2,B)

```
bool P(bool a, bool b)
requires true
ensures c==a || b
{
  if (a)
    c=true
  else
    c=b
}
```

WP(`if(a)` ..., c==a||b) =

a=> WP(`c=true`, c==a||b) &&

!a => WP(`c=b`, c==a||b)

= (a => true==a||b) && (!a => b== a||b)

Verification Condition:

true => WP(P, c==a||b)

true =>(a=> true==a||b) && (!a => b==a||b) ✓

Problems with WP computation?

- **Loop iterations!**
- WP_k(`while(E) {S}`, B)
  - WP_0(…) $=_{def}$!E => B
  - WP_1(…) $=_{def}$!E => B && E => WP (S,B)
    = WP_0(…) && E => WP(S,B)
  - WP_2(…) $=_{def}$ WP_1(…) && E=>WP(S, WP_1(…))
  - ….
  - WP_i+1(…) $=_{def}$ WP_i && E=>WP(S,WP_i(…))

Problems with WP computation?

- WP_k(`while(E) {S}`, B) ==
  - glb{WP_k(…) | for all k>=0)
  - glb means "greatest lower bound"

- Compute a precise WP might be impossible in some cases
  - An extremely expensive in other cases

Dealing with loops

- Solutions:
  - **Unroll loops**: Verify a fixed set of execution traces
  - Add loop invariants to programs

Hoare Rules for loops

{cond && A} body {A}

(A && !cond)=>B

{A} `while(cond) {body}` {B}

Hoare Rules for loop invariants

{cond && Inv} body {Inv}

A=>Inv

(A && !cond)=>B

{A} `while(cond) {body}` {B}

Handling Loops

- We extend our programming language with these new sentences
  - Assume E
  - Assert E
  - Havoc x (assign any non-deterministic value to x)
  - While_(I,T) E do S endwhile
    - Where:
      - I is the loop invariant
      - T is the set of modified locations, variables

Handling Loops

- We extend our WP definition for the new language constructs:
  - WP (havoc x, B) == \forall x. B
  - WP (assume E, B) == E=>B
  - WP (assert E, B) == E && B

Verifying Loops

- We transform loop code following this rule:

```
While_(I,T) E do S endwhile ==
        assert I          Check Invariant hold at loop entry
        havoc T
        assume I
        if (E) then
                S
                assert I          Check loop body preservers
                                  Invariant
                assume false
        endif
```

Exercise!

- Complete the following Hoare Triple with the weakest precondition:

{???}

While_(x>=0,x) x>0 do

X:=x-1

EndWhile

{x=0}

Procedure calls

- Options:
  - Inlining the procedure call
  - Replace procedure call with callee contract

- Given a Procedure “Proc” with precondition pre, postcondition post and a set of touched locations M, the statement Call Proc(x) is modelled as:
  - Assert pre
  - Havoc M
  - Assume post

Recap

- Axiomatic semantics using Hoare rules
- Computing a formula that captures the weakest precondition for a pair <program,postcondition>.
- Using WP for checking Hoare triples correctness
- How to use loop invariants for checking correctness

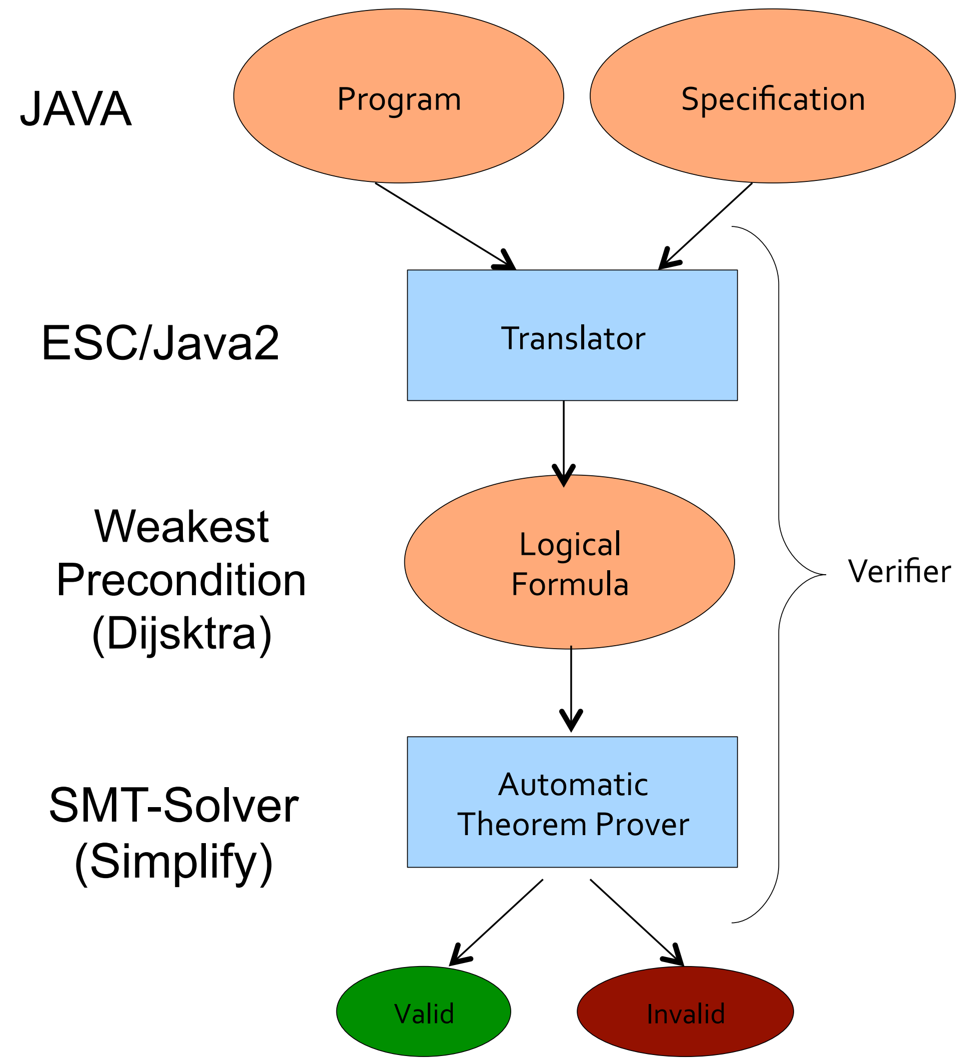
Tools! Tools! Tools!

- ESC/Java2: the formula is built using Dijsktra's Weakes precondition. Automatic theorem prover: Simplify SMT Solver.

http://kindsoftware.com/products/opensource/ESCJava2/

ESC/Java2

- Programming language
- Specification Language
- Logical representation of correctness
- Automatic decision procedure



Demo ESC/Java2

```
class Bag {
        int[] a;
        int n;
        int extractMin() {
                int mindex=0;
                int m=a[mindex];
                int i=1;
                for (i=1;i<n;i++) {
                        if (a[i]<m) {
                                mindex=i;
                                m = a[i];
                        }
                }
                n--;
                a[mindex]=a[n];
                return m;
        }
```

JML annotations for extractMin

```
//@ requires n>0;
//@ ensures (\forall int j; 0<=j && j<n ; \result<=a[j])
int extractMin() {
        int mindex=0;
        int m=a[mindex];
        int i=1;
        //@ loop_invariant i>=1;
        //@ loop_invariant i<=n;
        //@ loop_invariant mindex>=0;
        //@ loop_invariant mindex<i;
        //@ loop_invariant m==a[mindex];
        //@ loop_invariant (\forall int j; 0<=j && j<i; m<=a[j]);
        for (i=1;i<n;i++) {
                if (a[i]<m) {
                        mindex=i;
                        m = a[i];
                }
        }
        n--;
        a[mindex]=a[n];
        return m;
}
```

Lab Session on Thursday

- Bring your computer!
- Groups of 2
- Please install:
  - A Java IDE
  - At least JDK 1.6
  - CVC3 (http://www.cs.nyu.edu/acsys/cvc3/download.html)