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

JAVA

Program

Specification

JML

Translator

Logical Formula

Verifier

Automatic Theorem Prover

Valid

Invalid
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 execution finishes, B holds in final state.
A simple imperative language

- Atomic statements
  - Skip: `skip`
  - Assignment: `x := E`

- Control-flow statements
  - Sequential: `S1; S2`
  - Conditional: `if (cond) {S1} else {S2}`
  - Iteration: `while (cond) {S}`
Hoare Rules

\[
\begin{align*}
\{P\} \text{skip} \ {P} \\
\{A\} \ s1 \ \{C\} \ s2 \ {B} \\
\{A\} \ s1\,;\,s2 \ {B} \\
\{A \land \text{cond}\} \ s1 \ {B} \ \{A \land \neg \text{cond}\} \ s2 \ {B} \\
\{A\} \text{if}(\text{cond}) \ {s1} \text{ else } {s2} \ {B} \\
\{A \land \text{cond}\} \text{body} \ {A} \ \{A \land \neg \text{cond}\} => {B} \\
\{A\} \text{while}(\text{cond}) \ {\text{body}} \ {B}
\end{align*}
\]
Forward rule:

\{ A \} \ x := \ E \ \{ \exists x' | A[x \rightarrow x'] \ \&\& \ x == E[x \rightarrow x'] \}\n
- Intuition: \( x' \) is the previous value of \( x \). (\text{old}(x))

- Example:

\{ x \geq 3 \} \ x := \ x + 2 \ \{ \exists x' | (x \geq 3)[x \rightarrow x'] \ \&\& \ x == (x + 2)[x \rightarrow x'] \}\n
\{ x \geq 3 \} \ x := \ x + 2 \ \{ \exists x' | x' \geq 3 \ \&\& \ x == x' + 2 \}\n
\{ x \geq 3 \} \ x := \ x + 2 \ \{ \exists x' | x' \geq 3 \ \&\& \ x - 2 == x' \}\n
\{ x \geq 3 \} \ x := \ x + 2 \ \{ x - 2 \geq 3 \}\n
\{ x \geq 3 \} \ x := \ x + 2 \ \{ x \geq 5 \}
Hoare rules: assignment

Backward rule:

\[
\{ B[x \rightarrow E] \} \ x := E \ \{ B \}
\]

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

Example:

\[
\{ ? \} \ x := x + 2 \ \{ x \geq 5 \}
\]
\[
\{ x \geq 5[x \rightarrow x + 2] \} \ x := x + 2 \ \{ x \geq 5 \}
\]
\[
\{ x + 2 \geq 5 \} \ x := x + 2 \ \{ x \geq 5 \}
\]
\[
\{ x \geq 3 \} \ x := x + 2 \ \{ x \geq 5 \}
\]
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

WP

\[
X = 4 \\
Y = -1
\]

\[
X = 4 \\
Y = -15
\]

\[
X = 11 \\
Y = -3
\]

\[
X = 5 \\
Y = -1
\]

\[
X = 5 \\
Y = -15
\]

\[
X = 12 \\
Y = -3
\]

\[
X \geq 5 \land y < 0
\]

\[
X \geq 4 \land y < -2
\]

\[
X \geq 5 \land y < 0
\]
Proving correctness

{Weakest precondition (WP)}

\[ x := x + 1 \]
\[ \{ x \geq 5 \land y < 0 \} \]

- Since states \((x \geq 4 \land y < -2) \}\ subseteq \text{states(WP)}, then we have that

\[ \{ x \geq 4 \land y < -2 \} \]
\[ x := x + 1 \]
\[ \{ x \geq 5 \land y < 0 \} \]
Calculating the Weakest Precondition

- \( \text{WP}(\text{skip}, B) = \text{def } B \)
- \( \text{WP}(x := E, B) = \text{def } B[x \mapsto E] \)
- \( \text{WP}(s_1; s_2, B) = \text{def } \text{WP}(s_1, \text{WP}(s_2, B)) \)
- \( \text{WP}(\text{if}(E)\{s_1\} \text{else}\{s_2\}, B) = \text{def } \)
  
  \[
  E => \text{WP}(s_1, B) && \\
  !E => \text{WP}(s_2, B)
  \]
Given the following Hoare triple

\{Pre\}
Program
\{Post\}

The following formula is a Verification Condition (VC) for the triple:

- \( \text{Pre} \Rightarrow \text{WP}(\text{Program, Post}) \)

We call this a “backward” VC (in contrast with “forward” VC)
**Example**

- $WP(skip, B) = \text{def } B$
- $WP(x := E, B) = \text{def } B[x \rightarrow E]$

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

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

- $a \rightarrow WP(c = true, c == a || b)$
- $!a \rightarrow WP(c = b, c == a || b)$

= $(a \rightarrow \text{true} == a || b) \&\& (!a \rightarrow b == a || b)$

**Verification Condition:**

- $true \rightarrow WP(P, c == a || b)$
- $(a \rightarrow true == a || b) \&\& (!a \rightarrow b == a || b)$

- $true \Rightarrow (a \Rightarrow true == a || b) \&\& (!a \Rightarrow b == a || b)$
Problems with WP computation?

- **Loop iterations!**
- **WP\_k**\(_{\text{while}}(E) \ {S}, B)\)
  - **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(...))
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
Solutions:

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

\{\text{cond} \land \land \ A\} \ \text{body} \ \{\text{A}\}

(A \land \land \ !\text{cond}) \Rightarrow \text{B}

\text{while}(\text{cond}) \ \{\text{body}\} \ \{\text{B}\}
Hoare Rules for loop invariants

\[\{\text{cond } \land \land \text{ Inv}\}\text{ body } \{\text{Inv}\}\]

\[A \Rightarrow \text{Inv}\]

\[(A \land \!\! \text{cond}) \Rightarrow B\]

\[\{A\} \text{ while}(\text{cond}) \{\text{body}\} \{B\}\]
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:
  - $WP\ (havoc\ x,\ B)\ ==\ \forall\ x.\ B$
  - $WP\ (assume\ E,\ B)\ ==\ E\Rightarrow B$
  - $WP\ (assert\ E,\ B)\ ==\ E\ \&\&\ B$
We transform loop code following this rule:

While\((I,T) E\) do S endwhile ==

assert I  

havoc T

assume I

if (E) then
    S
    assert I
    assume false
endif

Check Invariant hold at loop entry

Check loop body preservers Invariant
Exercise!

- Complete the following Hoare Triple with the weakest precondition:

\[
\{???\} \\
\text{While}_{(x\geq 0, x)} \ x > 0 \ 	ext{do} \\
\quad \text{X} := x - 1 \\
\quad \text{EndWhile} \ \\
\{x = 0\}
\]
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
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

http://kindsoftware.com/products/opensource/ESCJava2/
- Programming language
- Specification Language
- Logical representation of correctness
- Automatic decision procedure
class Bag {
    int[] a;
    int n;
    int extractMin() {
        int minIndex = 0;
        int m = a[minIndex];
        int i = 1;
        for (i = 1; i < n; i++) {
            if (a[i] < m) {
                minIndex = i;
                m = a[i];
            }
        }
        n--; // Adjust to decrement correctly
        a[minIndex] = a[n];
        return m;
    }
}
//@ 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)