Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

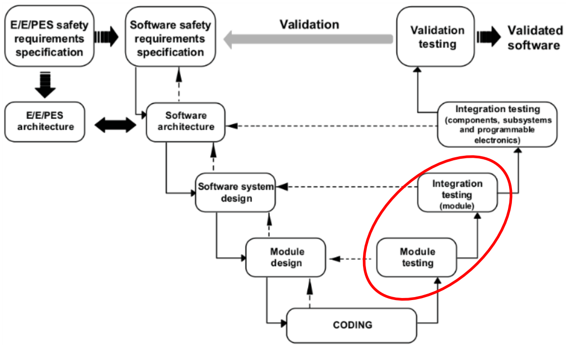
Lecture 09: Software Verification with Floyd-Hoare Logic

Christoph Lüth, Dieter Hutter, Jan Peleska

Where are we?

- 01: Concepts of Quality
- 02: Legal Requirements: Norms and Standards
- 03: The Software Development Process
- 04: Hazard Analysis
- 05: High-Level Design with SysML
- 06: Formal Modelling with OCL
- 07: Testing
- 08: Static Program Analysis
- 09: Software Verification with Floyd-Hoare Logic
- 10: Verification Condition Generation
- 11-12: Model Checking
- 13: Conclusions

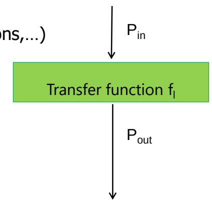
Software Verification in the Development Cycle



Static Program Analysis

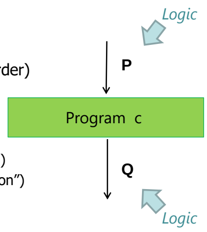
Transfer functions to propagate information along the execution path (i.e. from input to output, or vice versa)

- Information is encoded as a lattice $L = (M, \sqsubseteq)$.
- Transfer functions mapping information
  - $f_l : M \to M$ with l being a label
  - Knowledge transfer is monotone $\forall x, y.\ x \sqsubseteq y \Longrightarrow f_l(x) \sqsubseteq f_l(y)$
  - Restricted to a specific type of knowledge (Reachable Definitions, Available Expressions,…)
- What about a more general approach
  - Maintaining arbitrary knowledge ?
  - Knowledge representation ?



General Transfer Relations

- Transfer relations:
  - Knowledge P, Q is represented in logic (first-order)
  - {P} c {Q} denotes
    If P is known before executing c (and c terminates) then Q is known (P "precondition", Q "postcondition")
  - {P} c {Q} are called Floyd-Hoare triples



Charles Antony Richard Hoare: An axiomatic basis for computer programming (1969)
Robert W Floyd: Assigning Meanings to Programs (1967)

Software Verification

- Software Verification **proves** properties of programs. That is, given the basic problem of program P satisfying a property p we want to show that for **all possible inputs and runs** of P , the property p holds.
- Software verification is far **more powerful** than static analysis. For the same reasons, it cannot be fully automatic and thus requires user interaction. Hence, it is **complex to use**.
- Software verification does not have false negatives, only failed proof attempts. If we can prove a property, it holds.
- Software verification is used in **highly critical systems**.

The Basic Idea

- What does this program compute?
  - The index of the maximal element of the array a if it is non-empty.
- How to prove it?
  - (1) We need a language in which to **formalise** such **assertions**.
  - (2) We need a notion of meaning (**semantics**) for the program.
  - (3) We need to way to **deduce valid assertions**.
- Floyd-Hoare logic provides us with (1) and (3).

```
i := 0;
x := 0;
while (i < n) {
  if (a[i] ≥ a[x]) {
    x := i;
  }
  i := i + 1;
}
```

Formalizing correctness:

$\mathrm{array}(a, n) \wedge n > 0 \Longrightarrow$
$a[x] = \max(a, n)$
$\forall i.\ 0 \le i < n \Longrightarrow$
$a[i] \le \max(a, n)$
$\exists j.\ 0 \le j < n \Longrightarrow$
$a[j] = \max(a, n)$

Recall our simple programming language

- **Arithmetic** expressions:

  $a\ ::= x \mid n \mid a_1[a_2] \mid a_1\ op_a\ a_2$

  - Arithmetic operators: $op_a \in \{+, -, *, /\}$
- **Boolean** expressions:

  $b\ ::= \mathrm{true} \mid \mathrm{false} \mid \mathrm{not}\ b \mid b_1 op_b\ b_2 \mid a_1 op_r\ a_2$

  - Boolean operators: $op_b \in \{and, or\}$
  - Relational operators: $op_r \in \{=, <, \le, >, \ge, \neq\}$
- **Statements**:

  $\mathrm{S} ::= \mathrm{x} := \mathrm{a} \mid \mathbf{skip} \mid \mathrm{S1};\ \mathrm{S2} \mid \mathbf{if}\ (\mathrm{b})\ \ \mathrm{S1}\ \mathbf{else}\ \mathrm{S2} \mid \mathbf{while}\ (\mathrm{b})\ \mathrm{S}$

  - Labels from basic blocks omitted, only used in static analysis to derive cfg.
  - Note this abstract syntax, operator precedence and grouping statements is not covered.

Semantics of our simple language

- The semantics of an **imperative** language is state transition: the program has an ambient state, which is changed by assigning values to certain locations.
- Example:

x	?		x	5		x	5		x	6
y	12	x := 5 →	y	12	z := x + y →	y	12	x := x + 1 →	y	12
z	?		z	?		z	17		z	17

σ $\qquad \sigma_1 = \sigma$[x/5] $\qquad \sigma_2 = \sigma_1$[z/17] $= \sigma$[x/5, z/17] $\qquad \sigma_3 = \sigma_2$[x/6] $= \sigma$[x/6, z/17]

- Semantics in a nutshell:

> **Expressions** evaluate to values Val (for our language integers).
> **Locations** Loc are variable names.
> A **program state** maps locations to values: $\Sigma = Loc \rightharpoonup Val$
> A program maps an initial state to a final state, **if it terminates**.
> **Assertions** are predicates over program states.

Semantics in a nutshell

- There are three major ways to denote semantics.

(1) As a relation between program states, described by an abstract machine (**operational semantics**).
(2) As a function between program states, defined for each statement of the programming langauge (**denotational semantics**).
(3) As the set of all assertions which hold for a program (**axiomatic semantics**).

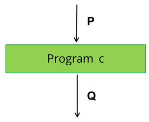
- Floyd-Hoare logic covers the third aspect, but it is important that all three semantics agree.
  - We will not cover semantics in detail here, but will concentrate on how to **use** Floyd-Hoare logic to prove correctness.

Extending our simple language

- We introduce a set Var of **logical variables.**
- **Assertions** are boolean expressions, which may not be executable, and arithmetic expressions containing logical variables.
- Arithmetic assertions
  $ae\ ::= x \mid X \mid n \mid ae_1[ae_2] \mid ae_1\ op_a\ ae_2 \mid f(ae_1, \dots, ae_n)$
  - where $x \in Loc, X \in Var, op_a \in \{+,-,*,/\}$
- Boolean assertions:
  $be\ \coloneqq \text{true} \mid \text{false} \mid \text{not}\ be \mid be_1 op_b\ be_2 \mid ae_1 op_r\ ae_2$
  $\mid p(ae_1, \dots, ae_n) \mid \forall X.\, be \mid \exists X.\, be$
  - Boolean operators: $op_b \in \{\wedge, \vee, \Longrightarrow\}$
  - Relational operators: $op_r \in \{=, <, \leq, >, \geq, \neq\}$

Floyd-Hoare Triples

The basic build blocks of Floyd-Hoare logic are Hoare triples of the form $\{P\}c\,\{Q\}$.



- P, Q are assertions using variables in Loc and Var
  - e.g. x < 5 + y, Odd(x), …
- A state σ satisfies P (written $\sigma \vDash P$) iff $P[^{\sigma(x)}/_x]$ is true for all $x \in Loc$ and all possible values for $X \in Var$:
  - e.g. let

x	5
y	12
z	17

$\sigma =$ (the state above), then σ satisfies x < 5 + y, Odd(x)

- A formula P describes a set of states, i.e. all states that satisfy the formula P.

Partial and Total Correctness

- **Partial correctness**: $\vDash \{P\}c\{Q\}$
  - c is partial correct with precondition P and postcondition Q iff, for all states σ which satisfy P and for which the execution of c terminates in some state σ' then it holds that σ' satisfies Q:
    $\forall \sigma.\, \sigma \vDash P \wedge \exists \sigma'.\, \langle \sigma, c \rangle \to \sigma' \implies \sigma' \vDash Q$
- **Total correctness**: $\vDash [P]c[Q]$
  - c is total correct with precondition P and postcondition Q iff, for all states σ which satisfy P the execution of c terminates in some state σ' which satisfies Q:
    $\forall \sigma.\, \sigma \vDash P \implies \exists \sigma'.\, \langle \sigma, \quad c \rangle \to \sigma' \wedge \sigma' \vDash Q$
- Examples: $\vDash \{true\} while(true)\ skip\ \{true\}$,
  $\nvDash [true]\ while(true) skip\ \ [true]$

Reasoning with Floyd-Hoare Triples

- How do we know that $\vDash \{P\}c\{Q\}$ in practice ?
- Calculus to derive triples, written as $\vdash \{P\}c\{Q\}$
  - Rules operate along the constructs of the programming language (cf. operational semantics)
  - Only one rule is applicable for each construct (!)
  - Rules are of the form

$$\frac{\vdash \{P_1\}c_1\{Q_1\}, \dots, \vdash \{P_n\}c_n\{Q_n\}}{\vdash \{P\}c\,\{Q\}}$$

meaning we can derive $\vdash \{P\}c\{Q\}$ if all $\vdash \{P_i\}c_i\{Q_i\}$ are derivable.

Floyd-Hoare Rules: Assignment

- Assignment rule:

$$\frac{}{\vdash \{P[^e/_x]\}\ x \coloneqq\ e\ \{P\}}$$

- $P[^e/_x]$ replaces all occurrences of the program variable x by the arithmetic expression e.
- Examples:
  - $\vdash\ \{0\ <\ 10\}\, x \coloneqq\ 0\, \{x\ <\ 10\}$
  - $\vdash \{\underbrace{x - 1\ <\ 10}_{x\ <\ 11}\}\, x \coloneqq\ x\ - 1\, \{x\ <\ 10\}$
  - $\vdash \{\underbrace{x\ +\ 1\ +\ x\ +\ 1\ <\ 10}_{x\ +\ x\ <\ 8}\}\ x \coloneqq\ x\ +\ 1\, \{x\ +\ x\ <\ 10\}$

Rules: Sequencing and Conditional

- Sequence:

$$\frac{\vdash \{P\}\ c_1\ \{Q\}\ \ \vdash \{Q\}\ c_2\ \{R\}}{\vdash \{P\}\ c_1; c_2\ \{R\}}$$

  - Needs an intermediate state predicate Q.
- Conditional:

$$\frac{\vdash \{P \wedge b\}\ c_1\ \{Q\}\ \ \vdash \{P \wedge \neg b\}\ c_2\ \{Q\}}{\vdash \{P\}\ \textbf{if}(\mathrm{b})\ c_1 \textbf{else}\ c_2\ \{Q\}}$$

  - Two preconditions capture both cases of b and $\neg\, b$.
  - Both branches end in the same postcondition Q.

Rules: Iteration and Skip

$$\frac{\vdash \{P \wedge b\}\ c\ \{P\}}{\vdash \{P\}\ \mathbf{while}\ (b)\ c\ \{P \wedge \neg\, b\}}$$

- P is called the **loop invariant**. It has to hold both before and after the loop (but not necessarily in the whole body).
- Before the loop, we can assume the loop condition b holds.
- After the loop, we know the loop condition b does not hold.
- In practice, the loop invariant has to be **given**– this is the creative and difficult part of working with the Floyd-Hoare calculus.

$$\frac{}{\vdash \{P\}\ \mathbf{skip}\ \{P\}}$$

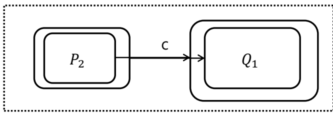
- **skip** has no effect: pre- and postcondition are the same.

Final Rule: Weakening

- Weakening is crucial, because it allows us to change pre- or postconditions by applying rules of logic.

$$\frac{P_2 \Longrightarrow P_1 \qquad \vdash \{P_1\}\ c\ \{Q_1\} \qquad Q_1 \Longrightarrow Q_2}{\vdash \{P_2\}\ c\ \{Q_2\}}$$

- We can **weaken** the precondition and **strengthen** the postcondition:
  - $P \Longrightarrow Q$ means that all states in which P holds, Q also holds.
  - $\vDash \{P\}c\{Q\}$ means whenever c starts in a state in which P holds, it ends in a state in which Q holds.
  - So, we can reduce the starting set, and enlarge the target set.



How to derive and denote proofs

// $\{P\}$
// $\{P_1\}$
x:= e;
// $\{P_2\}$
// $\{P_3\}$
while (x< n) {
  // $\{P_3 \wedge x < n\}$
  // $\{P_4\}$
  z := a
  // $\{P_3\}$
  }
// $\{P_3 \wedge \neg(x < n)\}$
// $\{Q\}$

- The example shows $\vdash \{P\}c\{Q\}$
- We annotate the program with valid assertions: the precondition in the preceding line, the postcondition in the following line.
- The sequencing rule is applied implicitly.
- Consecutive assertions imply weaking, which has to be proven separately.
  - In the example:
    $P \Longrightarrow P_1$,
    $P_2 \Longrightarrow P_3$,
    $P_3 \wedge x < n \Longrightarrow P_4$,
    $P_3 \wedge \neg(x < n) \Longrightarrow Q$

More Examples

P ==
 p ≔ 1;
 c ≔ 1;
 while (c ≤ n) {
  p ≔ p ∗ c;
  c ≔ c + 1
 }

Specification:
 $\vdash \{\, 1 \le n \}$
  P
 $\{\, p = n! \,\}$

Invariant:
 $p = (c - 1)!$

Q ==
 p ≔ 1;
 while (0 < n) {
  p ≔ p ∗ n;
  n ≔ n − 1
 }

Specification:
 $\vdash \{\, 1 \le n \wedge n = N \}$
  Q
 $\{\, p = N! \,\}$

Invariant:
 $p = \prod_{i=n+1}^{N} i$

R ==
 r ≔ a;
 q ≔ 0;
 while (b ≤ r) {
  r ≔ r − b;
  q ≔ q + 1
 }

Specification:
 $\vdash \{\, a \ge 0 \wedge b \ge 0 \,\}$
  R
 $\{\, a = b * q + r \wedge 0 \le r \wedge r < b \}$

Invariant:
 $a = b * q + r \wedge 0 \le r$

How to find an Invariant

- Going backwards: try to split/weaken postcondition Q into negated loop-condition and „something else" which becomes the invariant.
- Many while-loops are in fact for-loops, i.e. they count uniformly:

```
i ≔ 0;
while (i < n) {
  …;
  i ≔ i + 1
 }
```

- In this case:
  - If post-condition is $P(n)$, invariant is $P(i) \wedge i \le n$.
  - If post-condition is $\forall j.\, 0 \le j < n.\, P(j)$ (uses indexing, typically with arrays), invariant is $\forall j.\, j \le 0 < i.\, i \le n \wedge\ P(j)$.

Summary

- Floyd-Hoare-Logic allows us to **prove** properties of programs.
- The proofs cover all possible inputs, all possible runs.
- There is **partial** and **total correctness**:
  - Total correctness = partial correctness + termination.
- There is one rule for each construct of the programming language.
- Proofs can in part be constructed automatically, but iteration needs an **invariant** (which cannot be derived mechanically).
- Next lecture: correctness and completeness of the rules.