Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

Lecture 10:

Verification Condition Generation

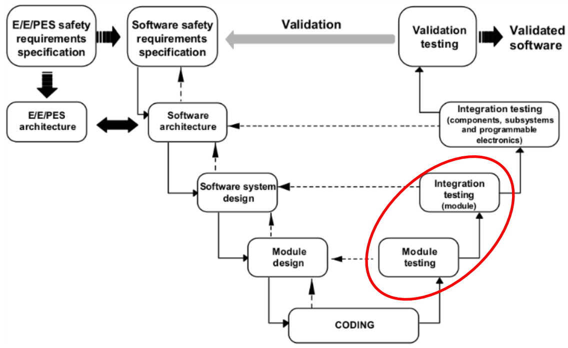
Christoph Lüth, Dieter Hutter, Jan Peleska

Frohes Neues Jahr!

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

VCG in the Development Cycle



Introduction

- In the last lecture, we introduced Hoare triples. They allow us to state and prove correctness assertions about programs, written as $\{P\}\,p\,\{Q\}$
- We introduced two notions, namely:
  - Syntactic derivability, $\vdash \{P\}\,p\,\{Q\}$ (the actual Floyd-Hoare calculus)
  - Semantic satisfaction, $\vDash \{P\}\,p\,\{Q\}$
- Question: how are the two related?
- The answer to that question also offers help with a practical problem: proofs with the Floyd-Hoare calculus are exceedingly long and tedious. Can we automate them, and how?

Correctness and Completeness

- In general, given a syntactic calculus with a semantic meaning, **correctness** means the syntactic calculus implies the semantic meaning, and **completeness** means all semantic statements can be derived syntactically.
  - Cf. also Static Program Analysis
- **Correctness** should be a basic property of verification calculi.
- **Completeness** is elusive due to Gödel's first incompleteness theorem:
  - Any logics which is strong enough to encode the natural numbers and primitive recursion* is incomplete.**

* Or any other notion of computation.
** Or inconsistent, which is even worse.

Correctness of the Floyd-Hoare calculus

> **Theorem** (Correctness of the Floyd-Hoare calculus)
>
> If $\vdash \{P\}\,p\,\{Q\}$, then $\vDash \{P\}\,p\,\{Q\}$.

- Proof: by induction on the derivation of $\vdash \{P\}\,p\,\{Q\}$.
- More precisely, for each rule we show that:
  - If the conclusion is $\vdash \{P\}\,p\,\{Q\}$, we can show $\vDash \{P\}\,p\,\{Q\}$
  - For the premisses, this can be assumed.
- Example: for the assignment rule, we show that

Completeness of the Floyd-Hoare calculus

- Predicate calculus is incomplete, so we cannot hope F/H is complete. But we get the following:

> **Theorem** (Relative completeness)
> If $\vDash \{P\}\,p\,\{Q\}$, then $\vdash \{P\}\,p\,\{Q\}$ *except* for the proofs occuring in the weakenings.

- To show this, we construct the **weakest precondition**.

> **Weakest precondition**
> Given a program c and an assertion P, the weakest precondition $wp(c, P)$ is an assertion W such that
> 1. W is a valid precondition $\vDash \{W\}\,c\,\{P\}$
> 2. And it is the weakest such: for any other Q such that $\vDash \{Q\}\,c\,\{P\}$, we have $W \rightarrow Q$.

Constructing the weakest precondition

- Consider a simple program and its verification:

$\{\, x = X \wedge y = Y \}$
$\leftrightarrow$
$\{\, y = Y \wedge x = X \}$
`z := y;`
$\{\, z = Y \wedge x = X \}$
`y := x;`
$\{\, z = Y \wedge y = X \}$
`x := z;`
$\{\, x = Y \ \wedge y = X \}$

- Note how proof is **constructed backwards systematically**.
- The idea is to construct the weakest precondition inductively.
- This also gives us a methodology to automate proofs in the calculus.

Constructing the weakest precondition

- There are four straightforward cases:
  - (1) $wp(\mathbf{skip}, P) = P$
  - (2) $wp(X \coloneqq e, P) = P\,[e \,/\, X]$
  - (3) $wp(c_0; c_1, P) = wp(c_0, wp(c_1, P))$
  - (4) $wp(\mathbf{if}\ b\ \{c_0\}\ \mathbf{else}\ \{c_1\}, P) = (b \wedge wp(c_0, P)) \vee (\neg\, b \wedge wp(c_1, P))$
- The complicated one is iteration (unsurprisingly, since it is the source of the computational power and Turing-completeness of the language). It can be given recursively:
  - (5) $wp(\mathbf{while}\ b\ \{c\}, P) = (\neg\, b \wedge P) \vee wp\big(c, wp\ (\mathbf{while}\ b\ \{c\}, P)\big)$
- A closed formula can be given, but it can be infinite and is not practical. It shows the relative completeness, but does not give us an effective way to automate proofs.
- Hence, $wp(c, P)$ is not effective for proof automation, but it shows the right way: we just need something for iterations.

Verification Conditions: Annotations

- The idea is that we have to give the invariants manually by annotating them.
- We need a language for this:
  - Arithmetic expressions and boolean expressions stays as they are.
  - Statements are augmented to **annotated statements**:
    - S ::= x := a | **skip** | S1; S2 | **if** (b) S1 **else** S2
    - | **assert** P | **while** (b) **inv** P S
  - Each while loop needs to its invariant annotated.
    - This is for partial correctness, total correctness also needs a **variant**: an expression which is strictly decreasing in a well-founded order such as $(<, \mathbb{N})$ after the loop body.
  - The assert statement allows us to force a weakening.

Preconditions and Verification Conditions

- We are given an annotated statement c, a precondition P and a postcondition Q.
  - We want to know: when does $\vDash \{P\}\, c\, \{Q\}$ hold?
- For this, we calculate a **precondition** $pre(c, Q)$ and a **set** of **verification conditions** $vc(c, Q)$.
  - The idea is that if all the verification conditions hold, then the precondition holds:
$$\bigwedge_{R \in vc(c,Q)} R \Rightarrow \vDash \{pre(c, Q)\}c\,\{Q\}$$
  - For the precondition P, we get the additional weaking $P \Rightarrow pre(c, Q)$.

Calculation Verification Conditions

- Intuitively, we calculate the verification conditions by stepping through the program backwards, starting with the postcondition Q.
- For each of the four simple cases (assignment, sequencing, case distinction and ***skip***), we calculate new current postcondition Q
- At each iteration, we calculate the precondition R of the loop body working backwards from the invariant I, and get two verification conditions:
  - The invariant I and negated loop condition implies Q.
  - The invariant I and loop condition implies R.
- Asserting R generates the verification condition $R \Rightarrow Q$.
- Let's try this.

Example: deriving VCs for the factorial.

{ 0 <= n }
{ 1 == (1-1)! && (1- 1) <= n }
p := 1;
{ p == (1-1)! && (1- 1) <= n }
c := 1;
{ p == (c-1)! && (c- 1) <= n }
while (c <= n)
inv (p == (c-1)! && c-1 <= n) {
{ p*c == ((c+1)-1)! &&
((c+1)- 1) <= n }
p := p* c;
{ p == ((c+1)-1)! && ((c+1)- 1) <= n }
c := c+1;
{ p == (c-1)! && (c- 1) <= n }
}
{ p = n! }

VCs (unedited):
1. p == (c-1)! && (c- 1) <= n && ! (c <= n) ==> p= n!
2. p == (c-1)! && c-1 <= n && c<= n ==> p* c= ((c+1)-1)! && ((c+1)-1) <= n
3. 0 <= n ==> 1= (1-1)! && 1-1 <= n

VCs (simplified):
1. p == (c-1)! && c- 1 == n ==> p= n!
2. p == (c-1)! && c-1 <= n && c<= n ==> p* c= c!
3. p == (c-1)! && c-1 <= n && c<= n ==> c <= n
4. 0 <= n ==> 1= 0!
5. 0 <= n ==> 0 <= n

Formal Definition

- Calculating the precondition:

$pre(\mathbf{skip}, Q) = Q$
$pre(X \coloneqq e, Q) = Q\,[e \,/\, X]$
$pre(c_0; c_1, Q = pre(c_0, pre(c_1, Q))$
$pre(\mathbf{if}\ (b)\ c_0\ \mathbf{else}\ c_1, Q) = \big(b \wedge pre(c_0, Q)\big) \vee \big(\neg\, b \wedge pre(c_1, Q)\big)$
$pre(\mathbf{assert}\ R, Q) = R$
$pre\ (\mathbf{while}\ (b)\mathbf{inv}\ I\ c, Q) = I$

- Calculating the verification conditions:

$vc(skip, Q) = \emptyset$
$vc(X \coloneqq e, Q) = \emptyset$
$vc(c_0; c_1, Q) = vc\big(c_0, pre(c_1, Q)\big) \cup vc(c_1, Q)$
$vc(\mathbf{if}\ (b)\ c_0\ \mathbf{else}\ c_1, Q) = vc(c_0, Q) \cup vc(c_1, Q)$
$vc(\mathbf{while}\ (b)\ \mathbf{inv}\ I\ c, Q) = vc(c, I) \cup \{I \wedge b \Rightarrow pre(c, I), I \wedge \neg b \Rightarrow Q\}$
$vc(\mathbf{assert}\ R, Q) = \{R \Rightarrow Q\}$

- The main definition:

$$vcg(\{P\}\, c\, \{Q\}) = \{P \Rightarrow pre(c, Q)\} \cup vc(c, Q)$$

Another example: integer division

{ 0 <= a && 0 <= b }
{ 1 }
r := a;
{ 2 }
q := 0;
{ 3 }
while (b <= r)
inv (a == b* q + r && 0 <= r) {
{ 4 }
r := r- b;
{ 5 }
q := q+1;
{ 6 }
}
{ a == b* q + r && 0 <= r && r < b }

Correctness of VC

- The correctness calculus is correct: if we can prove all the verifcation conditons, the program is correct w.r.t to given pre- and postconditions.
- Formally:

Theorem (Correctness of the VCG calculus)
Given assertions P and Q (with P the precondition and Q the postcondition), and an annotated program, then

$$\bigwedge_{R \in vcg(c,Q)} R \Rightarrow \models \{P\}\, c\, \{Q\}$$

- Proof: by induction on c.

Using VCG in Real Life

We have just a toy language, but VCG can be used in real life. What features are missing?

- **Modularity**: the language must have modularity concepts, e.g. functions (as in C), or classes (as in Java), and we must be able to verify them separately.
- **Framing**: in our simple calculus, we need to specify which variables stay the same (e.g. when entering a loop). This becomes tedious when there are a lot of variables involved; it is more practical to specify which variables may change.
- **References**: languages such as C and Java use references, which allow aliasing. This has to be modelled semantically; specifically, the assignment rule has to be adapted.
- **Machine arithmetic**: programs work with machine words and floating point representations, not integers and real numbers. This can be the cause of insidious errors.

VCG Tools

- Often use an intermediate language for VCG and front-ends for concrete programming languages.
- The Why3 toolset (http://why3.lri.fr)
  - A verification condition generator
  - Front-ends for different languages:
    C (Frama-C), Java (defunct?)
- Boogie (Microsoft Research)
  - Frontends for programming languages such C, C#, Java.
- VCC – a verifying C compiler built on top of Boogie
  - Interactive demo:
    https://www.rise4fun.com/Vcc/

VCC Example: Binary Search

- A correct (?) binary search implementation:

```
#include <limits.h>

unsigned int bin_search(unsigned int a [], unsigned int a_len, unsigned int key)
{
  unsigned int lo= 0;
  unsigned int hi= a_len;
  unsigned int mid;

  while (lo <= hi)
     {
      mid= (lo+ hi)/2;
      if (a[mid] < key) lo= mid+1;
      else hi= mid;
    }

  if (!(lo < a_len && a[lo] == key)) lo= UINT_MAX;

  return lo;
}
```


VCC: Correctness Conditions?

- We need to annotate the program.
- Precondition:
  - `a` is an array of length `a_len`;
  - The array `a` is sorted.
- Postcondition:
  - Let `r` be the result, then:
  - if `r` is `UINT_MAX`, all elements of `a` are unequal to `key`;
  - if `r` is not `UINT_MAX`, then `a[r] == key`.
- Loop invariants:
  - `hi` is less-equal to `a_len`;
  - everything „left" of `lo` is less then `key`;
  - everything „right" of `hi` is larger-equal to `key`.

VCC Example: Binary Search

- Source code as annotated for VCC:

```
#include <limits.h>
#include <vcc.h>
unsigned int bin_search(unsigned int a [], unsigned int a_len, unsigned int key)
  _(requires \thread_local_array(a, a_len))
  _(requires \forall unsigned int i, j; i < j && j < a_len ==> a[i] <= a[j])
  _(ensures \result != UINT_MAX ==> a[\result] == key)
  _(ensures \result == UINT_MAX ==> \forall unsigned int i; i < a_len ==> a[i] !=  key)
{
  unsigned int lo= 0;
  unsigned int hi= a_len;
  unsigned int mid;

  while (lo <= hi)
    _(invariant hi <= a_len)
    _(invariant \forall unsigned int i; i < lo ==> a[i] <  key)
    _(invariant \forall unsigned int i; hi <= i && i < a_len ==>a[i] >= key)
    {
      mid= (lo+ hi)/2;
      if (a[mid] < key) lo= mid+1;
      else hi= mid;
    }
  if (!(lo < a_len && a[lo] == key)) lo= UINT_MAX;
  return lo;
}
```


Binary Search: the Corrected Program

- Corrected source code:

```
#include <limits.h>
#include <vcc.h>
unsigned int bin_search(unsigned int a [], unsigned int a_len, unsigned int key)
  _(requires \thread_local_array(a, a_len))
  _(requires \forall unsigned int i, j; i < j && j < a_len ==> a[i] <= a[j])
  _(ensures \result != UINT_MAX ==> a[\result] == key)
  _(ensures \result == UINT_MAX ==> \forall unsigned int i; i < a_len ==> a[i] !=  key)
{
  unsigned int lo= 0;
  unsigned int hi= a_len;
  unsigned int mid;

  while (lo < hi)
    _(invariant hi <= a_len)
    _(invariant \forall unsigned int i; i < lo ==> a[i] <  key)
    _(invariant \forall unsigned int i; hi <= i && i < a_len ==>a[i] >= key)
    {
      mid= (hi-lo)/2+ lo;
      if (a[mid] < key) lo= mid+1;
      else hi= mid;
    }
  if (!(lo < a_len && a[lo] == key)) lo= UINT_MAX;
  return lo;
}
```


Summary

- Starting from the relative completeness of the Floyd-Hoare calculus, we devised a verification condition generation (vcg) calculus which makes program verification viable.
- Verification condition generation reduces the question whether the given pre/postconditions hold for a program to the validity of a set of logical properties.
  - We do need to annotate the while loops with invariants.
  - Most of these logical properties can be discharged with automated theorem provers.
- To scale to real-world programs, we need to deal with framing, modularity (each function/method needs to be verified independently), and machine arithmetic (integer word arithmetic and floating-points).