Systeme Hoher Qualität und Sicherheit
Vorlesung 10 vom 06.01.2014: Verification Condition Generation

Christoph Lüth & Christian Liguda

Universität Bremen

Wintersemester 2013/14

Frohes Neues Jahr!

Where are we?

- Lecture 1: Concepts of Quality
- Lecture 2: Concepts of Safety and Security, Norms and Standards
- Lecture 3: Quality of the Software Development Process
- Lecture 4: Requirements Analysis
- Lecture 5: High-Level Design & Formal Modelling
- Lecture 6: Detailed Specification, Refinement & Implementation
- Lecture 7: Testing
- Lecture 8: Program Analysis
- Lecture 9: Verification with Floyd-Hoare Logic
- **Lecture 10: Verification Condition Generation**
- Lecture 11: Model-Checking with LTL and CTL
- Lecture 12: NuSMV and Spin
- Lecture 13: Conclusions

Introduction

- In the last lecture, we learned about the **Floyd-Hoare calculus.**
- It allowed us to **state** and **prove** correctness assertions about programs, written as $\{P\}\, c\, \{Q\}$.
- The **problem** is that proofs of $\vdash \{P\}\, c\, \{Q\}$ are **exceedingly** tedious, and hence not viable in practice.
- We are looking for a calculus which reduces the size (and tediousness) of Floyd-Hoare proofs.
- The starting point is the **relative completeness** of the Floyd-Hoare calculus.

Completeness of the Floyd-Hoare Calculus

Relative Completeness

If $\models \{P\}\, c\, \{Q\}$, then $\vdash \{P\}\, c\, \{Q\}$ except for the weakening conditions.

- To show this, one constructs a so-called **weakest precondition**.

Weakest Precondition

Given a program c and an assertion P, the weakest precondition is an assertion W which

1. is a valid precondition: $\models \{W\}\, c\, \{P\}$
2. and is the weakest such: if $\models \{Q\}\, c\, \{P\}$, then $W \longrightarrow Q$.

- Question: is the weakest precondition **unique?**

Completeness of the Floyd-Hoare Calculus

Relative Completeness

If $\models \{P\}\, c\, \{Q\}$, then $\vdash \{P\}\, c\, \{Q\}$ except for the weakening conditions.

- To show this, one constructs a so-called **weakest precondition**.

Weakest Precondition

Given a program c and an assertion P, the weakest precondition is an assertion W which

1. is a valid precondition: $\models \{W\}\, c\, \{P\}$
2. and is the weakest such: if $\models \{Q\}\, c\, \{P\}$, then $W \longrightarrow Q$.

- Question: is the weakest precondition **unique?**
  Only up to logical equivalence: if W_1 and W_2 are weakest preconditions, then $W_1 \longleftrightarrow W_2$.

Constructing the Weakest Precondition

- Consider the following simple program and its verification:

  $\{X = x \land Y = y\}$

  ```
  Z:= Y;

  Y:= X;

  X:= Z;
  ```
  $\{X = y \land Y = x\}$

Constructing the Weakest Precondition

- Consider the following simple program and its verification:

$\{X = x \wedge Y = y\}$

Z:= Y;

Y:= X;
$\{Z = y \wedge Y = x\}$
X:= Z;
$\{X = y \wedge Y = x\}$

Constructing the Weakest Precondition

- Consider the following simple program and its verification:

  $\{X = x \land Y = y\}$

  Z:= Y;
  $\{Z = y \land X = x\}$
  Y:= X;
  $\{Z = y \land Y = x\}$
  X:= Z;
  $\{X = y \land Y = x\}$

Constructing the Weakest Precondition

- Consider the following simple program and its verification:

  $\{X = x \wedge Y = y\}$
  $\longleftrightarrow$
  $\{Y = y \wedge X = x\}$
  Z:= Y;
  $\{Z = y \wedge X = x\}$
  Y:= X;
  $\{Z = y \wedge Y = x\}$
  X:= Z;
  $\{X = y \wedge Y = x\}$

- The idea is to construct the weakest precondition **inductively**.

Constructing the Weakest Precondition

- There are four straightforward cases:

$$
\begin{array}{rcl}
\mathrm{wp}(\mathbf{skip}, P) & \overset{def}{=} & P \\
\mathrm{wp}(X := e, P) & \overset{def}{=} & P[e/X] \\
\mathrm{wp}(c_0; c_1, P) & \overset{def}{=} & \mathrm{wp}(c_0, \mathrm{wp}(c_1, P)) \\
\mathrm{wp}(\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1, P) & \overset{def}{=} & (b \wedge \mathrm{wp}(c_0, P)) \vee (\neg b \wedge \mathrm{wp}(c_1, P))
\end{array}
$$

- The complicated one is iteration. This is not surprising, because iteration gives us computational power (and makes our language Turing-complete). It can be given recursively:

  $$\mathrm{wp}(\mathbf{while}\ b\ \mathbf{do}\ c, P) \overset{def}{=} (\neg b \wedge P) \vee (b \wedge \mathrm{wp}(c, \mathrm{wp}(\mathbf{while}\ b\ \mathbf{do}\, c, P)))$$

  A closed formula can be given using Turing's β-predicate, but it is unwieldy to write down.

- Hence, $\mathrm{wp}(c, P)$ is not an effective way to **prove** correctness.

Verfication Conditions: Annotated Programs

- **Idea**: invariants specified in the program by **annotations**.
- Arithmetic and Boolean Expressions (**AExp**, **BExp**) remain as they are.
- Annotated Statements (**ACom**)

$$\begin{aligned} c ::= {} & \textbf{skip} \mid \textbf{Loc} := \textbf{AExp} \mid \textbf{assert}\ P \mid \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \\ & \mid \textbf{while}\ b\ \textbf{inv}\ I\ \textbf{do}\ c \mid c_1; c_2 \mid \{c\} \end{aligned}$$

Calculation Verification Conditions

- For an annotated statement $c \in \mathbf{ACom}$ and an assertion P (the postcondition), we calculuate a **set** of verification conditions $\mathsf{vc}(c, P)$ and a precondition $\mathsf{pre}(c, P)$.
- The precondition is an auxiliary definition — it is mainly needed to compute the verification conditions.
- If we can prove the verification conditions, then $\mathsf{pre}(c, P)$ is a proper precondition, i.e. $\models \{\mathsf{pre}(c, P)\}\, c\, \{P\}$.

Calculating Verification Conditions

$$\begin{aligned}
\mathrm{pre}(\textbf{skip}, P) &\overset{def}{=} P \\
\mathrm{pre}(X := e, P) &\overset{def}{=} P[e/X] \\
\mathrm{pre}(c_0; c_1, P) &\overset{def}{=} \mathrm{pre}(c_0, \mathrm{pre}(c_1, P)) \\
\mathrm{pre}(\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, P) &\overset{def}{=} (b \wedge \mathrm{pre}(c_0, P)) \vee (\neg b \wedge \mathrm{pre}(c_1, P)) \\
\mathrm{pre}(\textbf{assert}\ Q, P) &\overset{def}{=} Q \\
\mathrm{pre}(\textbf{while}\ b\ \textbf{inv}\ I\ \textbf{do}\ c, P) &\overset{def}{=} I
\end{aligned}$$

$$\begin{aligned}
\mathrm{vc}(\textbf{skip}, P) &\overset{def}{=} \emptyset \\
\mathrm{vc}(X := e, P) &\overset{def}{=} \emptyset \\
\mathrm{vc}(c_0; c_1, P) &\overset{def}{=} \mathrm{vc}(c_0, \mathrm{pre}(c_1, P)) \cup \mathrm{vc}(c_1, P) \\
\mathrm{vc}(\textbf{if}\ b\ \textbf{then}\ c_0\ \textbf{else}\ c_1, P) &\overset{def}{=} \emptyset \\
\mathrm{vc}(\textbf{assert}\ Q, P) &\overset{def}{=} \{Q \longrightarrow P\} \\
\mathrm{vc}(\textbf{while}\ b\ \textbf{inv}\ I\ \textbf{do}\ c, P) &\overset{def}{=} \mathrm{vc}(c, I) \cup \{I \wedge b \longrightarrow \mathrm{pre}(c, I)\} \\
&\qquad \cup \{I \wedge \neg b \longrightarrow P\}
\end{aligned}$$

Correctness of the VC Calculus

Correctness of the VC Calculus

For an annotated program c and an assertion P, let $\mathrm{vc}(c, P) = \{P_1, \ldots, P_n\}$. If $P_1 \wedge \ldots \wedge P_n$, then $\models \{\mathrm{pre}(c, P)\}\, c\, \{P\}$.

- Proof: By induction on c.

Example: Faculty

Let *Fac* be the annotated faculty program:

```
{0 ≤ N}
P := 1;
C := 1;
while C ≤ N inv {P = (C − 1)! ∧ C − 1 ≤ N} do {
  P := P × C;
  C := C + 1
}
{P = N!}
```

$$\begin{aligned} &\mathrm{vc}(\mathit{Fac}) = \\ &\quad \{\ 0 \leq N \longrightarrow 1 = 0! \wedge 0 \leq N, \\ &\qquad P = (C-1)! \wedge C-1 \leq N \wedge C \leq N \longrightarrow P \times C = C! \wedge C \leq N, \\ &\qquad P = (C-1)! \wedge C-1 \leq N \wedge \neg(C \leq N) \longrightarrow P = N!\ \} \end{aligned}$$

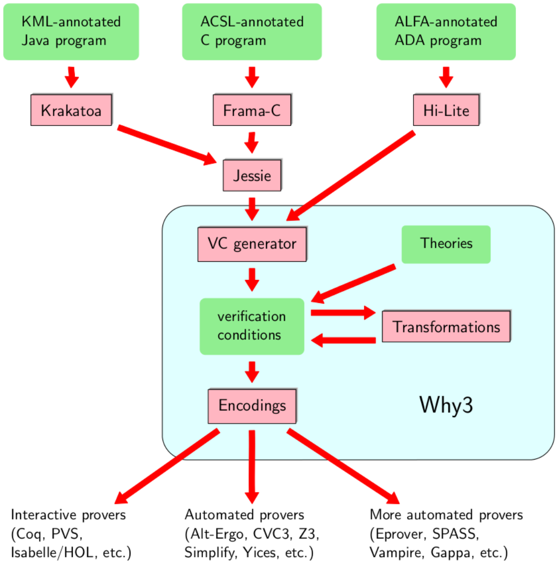
The Framing Problem

- One problem with the simple definition from above is that we need to specify which variables stay the same (**framing problem**).
  - Essentially, when going into a loop we use lose all information of the current precondition, as it is replaced by the loop invariant.
  - This does not occur in the faculty example, as all program variables are changed.
- Instead of having to write this down every time, it is more useful to modify the logic, such that we specify which variables are **modified**, and assume the rest stays untouched.
- Sketch of definition: We say $\models \{P, X\}\, c\, \{Q\}$ is a Hoare-Triple with **modification set** X if for all states σ which satisfy P if c terminates in a state σ', then σ' satisfies Q, and if $\sigma(x) \neq \sigma'(x)$ then $x \in X$.

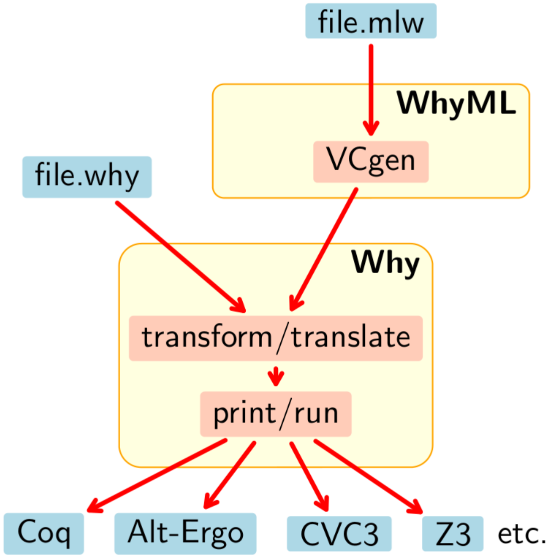
Verification Condition Generation Tools

- The Why3 toolset (`http://why3.lri.fr`)
  - The Why3 verification condition generator
  - Plug-ins for different provers
  - Front-ends for different languages: C (Frama-C), Java (Krakatoa)
- The Boogie VCG (`http://research.microsoft.com/en-us/projects/boogie/`)
- The VCC Tool (built on top of Boogie)
  - Verification of C programs
  - Used in German Verisoft XT project to verify Microsoft Hyper-V hypervisor

Why3 Overview: Toolset


KML-annotated Java program
ACSL-annotated C program
ALFA-annotated ADA program
Krakatoa
Frama-C
Hi-Lite
Jessie
VC generator
Theories
verification conditions
Transformations
Encodings
Why3
Interactive provers (Coq, PVS, Isabelle/HOL, etc.)
Automated provers (Alt-Ergo, CVC3, Z3, Simplify, Yices, etc.)
More automated provers (Eprover, SPASS, Vampire, Gappa, etc.)

Why3 Overview: VCG



Why3 Example: Faculty (in WhyML)

```
let fac(n: int): int
  requires { n >= 0 }
  ensures  { result = fact(n) } =
  let p = ref 0 in
  let c = ref 0 in
  p := 1;
  c := 1;
  while !c <= n do
    invariant { !p= fact(!c-1) /\ !c-1 <= n }
    variant { n- !c }
    p:= !p* !c;
    c:= !c+ 1
    done;
 !p
```

Why3 Example: Generated VC for Faculty

```
goal WP_parameter_fac :
 forall n:int.
  n >= 0 ->
   (forall p:int.
     p = 1 ->
      (forall c:int.
        c = 1 ->
         (p = fact (c - 1) /\ (c - 1) <= n) /\
          (forall c1:int, p1:int.
            p1 = fact (c1 - 1) /\ (c1 - 1) <= n ->
             (if c1 <= n then forall p2:int.
                                p2 = (p1 * c1) ->
                                 (forall c2:int.
                                   c2 = (c1 + 1) ->
                                    (p2 = fact (c2 - 1) /\
                                     (c2 - 1) <= n) /\
                                     0 <= (n - c1) /\
                                     (n - c2) < (n - c1))
              else p1 = fact n))))
```

Summary

- Starting from the **relative completeness** of the Floyd-Hoare calculus, we devised a **Verification Condition Generation** calculus which makes program verification viable.
- Verification Condition Generation reduces an **annotated** program to a set of logical properties.
- We need to annotate **preconditions**, **postconditions** and **invariants**.
- Tools which support this sort of reasoning include **Why3** and **Boogie**. They come with front-ends for **real programming languages**, such as C, Java, C#, and Ada.
- 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).