Systeme Hoher Sicherheit und Qualität
Universität Bremen WS 2015/2016

Lecture 11 (11.01.2016)

Verification Condition Generation

Christoph Lüth Jan Peleska Dieter Hutter

Universität Bremen

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 SysML and OCL
- 07: Detailed Specification with SysML
- 08: Testing
- 09: Program Analysis
- 10: Foundations of Software Verification
- 11: Verification Condition Generation
- 12: Semantics of Programming Languages
- 13: Model-Checking
- 14: Conclusions and Outlook

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?
  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:

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

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

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

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

  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**)

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

Calculuation Verification Conditions

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

Calculating Verification Conditions

$$
\begin{aligned}
\mathrm{pre}(\mathbf{skip}, P) &\stackrel{def}{=} P\\
\mathrm{pre}(X := e, P) &\stackrel{def}{=} P[e/X]\\
\mathrm{pre}(c_0; c_1, P) &\stackrel{def}{=} \mathrm{pre}(c_0, \mathrm{pre}(c_1, P))\\
\mathrm{pre}(\mathbf{if}\ b\ \{c_0\}\ \mathbf{else}\ \{c_1\}, P) &\stackrel{def}{=} (b \wedge \mathrm{pre}(c_0, P)) \vee (\neg b \wedge \mathrm{pre}(c_1, P))\\
\mathrm{pre}(\mathbf{assert}\ Q, P) &\stackrel{def}{=} Q\\
\mathrm{pre}(\mathbf{while}\ b\ \mathbf{inv}\ I\ \{c\}, P) &\stackrel{def}{=} I\\
\\
\mathrm{vc}(\mathbf{skip}, P) &\stackrel{def}{=} \emptyset\\
\mathrm{vc}(X := e, P) &\stackrel{def}{=} \emptyset\\
\mathrm{vc}(c_0; c_1, P) &\stackrel{def}{=} \mathrm{vc}(c_0, \mathrm{pre}(c_1, P)) \cup \mathrm{vc}(c_1, P)\\
\mathrm{vc}(\mathbf{if}\ b\ \{c_0\}\ \mathbf{else}\ \{c_1\}, P) &\stackrel{def}{=} \mathrm{vc}(c_0, P) \cup \mathrm{vc}(c_1, P)\\
\mathrm{vc}(\mathbf{assert}\ Q, P) &\stackrel{def}{=} \{Q \longrightarrow P\}\\
\mathrm{vc}(\mathbf{while}\ b\ \mathbf{inv}\ I\ \{c\}, P) &\stackrel{def}{=} \mathrm{vc}(c, I)\ \cup \{I \wedge b \longrightarrow \mathrm{pre}(c, I)\}\\
&\qquad\qquad \cup \{I \wedge \neg b \longrightarrow P\}\\
\mathrm{vc}(\{P\}\, c\, \{Q\}) &\stackrel{def}{=} \{P \longrightarrow \mathrm{pre}(c, Q)\} \cup \mathrm{vc}(c, Q)
\end{aligned}
$$

Correctness of the VC Calculus

Correctness of the VC Calculus

For a annotated program c and an assertion P:

$$\mathrm{vc}(c, P) \implies \{\mathrm{pre}(c, P)\}\, c\, \{P\}$$

- 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} {
  P := P * C;
  C := C + 1
}
{P = N!}
```

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

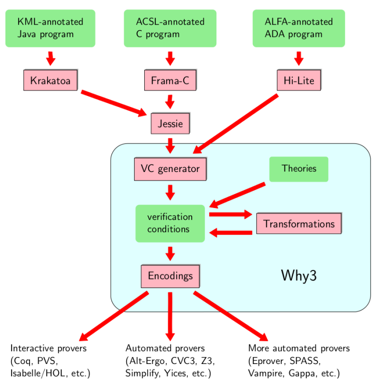
The Framing Problem

- 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 $x \in X$.

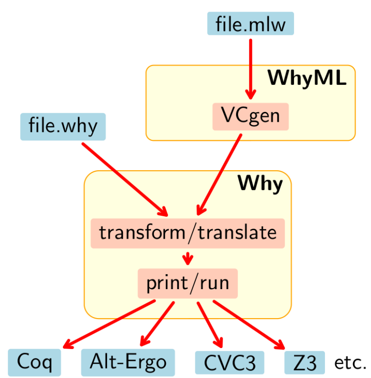
Verification Condition Generation Tools

- 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



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).