Rigorous Methods for Software Engineering (F21RS-F20RS) Program Verification Part 1: Specification & Verification Condition Generation

Andrew Ireland
Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Overview

- Specifying correctness of imperative programs.
- Verification condition generation.

A Little History & Background

- Long history:
  - Goldstine & von Neumann 1947
  - Turing 1949
  - Floyd 1967
  - Hoare 1969
- Current applications:
  - SPARK (Ada subset)
  - Frama-C
  - eCv (MISRA-C)
  - Spec# (C#)
  - ...

Correctness Specifications

Partial correctness:

- $\{P\}\, C \{Q\}$
- P and Q are propositions and C is code.
- IF P is true before C is executed AND the execution of C terminates THEN Q is true after the execution terminates.

Total correctness:

- $[P]\ C\ [Q]$
- IF P is true before C is executed THEN the execution of C will terminate and Q will be true on termination.
- total correctness = partial correctness + termination

A Toy Programming Language

$$
\begin{array}{lll}
\langle \mathit{command} \rangle & ::= & \langle \mathit{variable} \rangle := \langle \mathit{term} \rangle \mid \\
 & & \langle \mathit{command} \rangle ; \ \ldots ; \ \langle \mathit{command} \rangle \mid \\
 & & \textbf{if } \langle \mathit{statement} \rangle \textbf{ then } \langle \mathit{command} \rangle \textbf{ end if} \mid \\
 & & \textbf{if } \langle \mathit{statement} \rangle \textbf{ then } \langle \mathit{command} \rangle \\
 & & \qquad\qquad\qquad \textbf{ else } \langle \mathit{command} \rangle \textbf{ end if} \mid \\
 & & \textbf{while } \langle \mathit{statement} \rangle \textbf{ loop } \langle \mathit{command} \rangle \textbf{ end loop}
\end{array}
$$

- Variables are represented by single upper-case letters.
- Terms denote integer values, *e.g.* 42, 2+3, `X+1`.
- Statements denote boolean values and can be built up using logical constants, *i.e.* *true* and *false*, terms and predicates, *e.g.* =, /=, >=, <=, *etc*, as well as the logical operators **and**, **or**, **not**.

Note: this “toy” programming language will enable us to cover the main aspects of imperative program proof – it would take much longer to cover the same ground using SPARK proof tools.

A Little Logic

- $P \wedge Q$: is true if P and Q are true.
- $P \vee Q$: is true if either P or Q are true, otherwise false.
- $P \rightarrow Q$: is true if whenever P is true Q is also true. If P is false, then $P \rightarrow Q$ is true.
- $\neg P$: is true if P is false, and false if P is true.

Some Examples

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

```
[$X \geq 0 \wedge even(X)$]
while X /= 0  loop
    X := X - 2;
end loop
[$X = 0$]
```

- Note that x and y denote variables that are used to name the initial values of X and Y respectively.
- Such variables are called **auxiliary** or **ghost** variables as they do not appear within the code. Lower-case letters will be used to denote auxiliary variables.

Which are Correct?

1. $\{X = x \wedge Y = y\}$
   ```
   X := Y;
   Y := X
   ```
   $\{X = y \wedge Y = x\}$
2. $\{true\}$
   ```
   if even(X)  then X := X + 1 end if
   ```
   $\{odd(X)\}$
3. $[true]$
   ```
   while X /= 0  loop
         X := X - 2
   end loop
   ```
   $[X = 0]$

Goal Oriented Program Verification

1. An annotated program C is developed where the annotations are assertions (logical formulae) that express desired conditions at various intermediate points.
2. A set of logical formulae, or **verification conditions** (VCs) is generated from the annotated program.
3. The verification conditions are proved by a theorem proving system.

Note that if all the generated VCs are proved then it follows that the program is correct with respect to the annotations.

Annotations

In order for VC generation to be automatic requires that the program is properly annotated, *i.e.* the program contains enough assertions. A program is properly annotated if there exists an assertion:

- Before each program command C_i (where $i > 1$) in a sequence $C_1; C_2; \ldots; C_n$ which is **not** an assignment command.
- After the reserved word **loop** within the **while** loop construct, *i.e.* the loop invariant.

Note that the generation of loop invariants is undecidable in general and remains a major bottle-neck in terms of fully automating the process of verifying programs.

Quotient-Remainder Example

```
{$true$}
R:= X;
Q:= 0;
{$R = X \wedge Q = 0$}
while Y<=R loop
    {$X = R + (Y * Q)$}        $\longleftarrow$ loop invariant
    R:= R - Y;
    Q:= Q + 1;
end loop;
{$X = R + (Y * Q) \wedge R < Y$}
```

A **loop invariant** is an assertion that is true before and after each execution of the loop body.

Quotient-Remainder VCs

Given the annotated Quotient-Remainder program on the previous slide, the process of **verification condition generation** (VCG) would produce the following VCs:

$$\begin{aligned}
\mathit{true} &\rightarrow (X = X \wedge 0 = 0) \\
(R = X \wedge Q = 0) &\rightarrow (X = R + (Y * Q)) \\
(X = R + (Y * Q)) \wedge \neg(Y \leq R) &\rightarrow (X = R + (Y * Q) \wedge R < Y) \\
(X = R + (Y * Q)) \wedge Y \leq R &\rightarrow (X = (R - Y) + (Y * (Q + 1)))
\end{aligned}$$

Note that VCs are logical formulae — they contain no references to programming constructs.

VCG - How it Works

- VCG is a recursive process, where the given annotated program is decomposed repeatedly into simpler verification tasks – each of which can be proved independently.
- This recursive process is defined for our toy programming language by the following VC generation rules ...

Generating VCs for Assignment Commands

The VC generated by

$$\{P\}V := E\{Q\}$$

is

$$P \to Q[E/V]$$

where $Q[E/V]$ denotes the expression constructed by replacing all occurrences of V by E within Q - known as the **weakest precondition**.
Here the **weakest precondition** represents the set of all the states such that if $V := E$ is executed in anyone of the states then on termination Q will be true.

Example: The VC generated by

$$\{X = 0\}X := X + 1\{X = 1\}$$

is

$$(X = 0) \to (X + 1) = 1$$

Generating VCs for **if-then** Conditionals

The VCs generated by

$$\{P\}\textbf{if}\ \ S\ \ \textbf{then}\ \ C\,\textbf{end if}\{Q\}$$

are

1. $(P \wedge \neg S) \rightarrow Q$
2. the VCs generated from:

$$\{P \wedge S\}C\{Q\}$$

Generating VCs for **if-then-else** Conditionals

The VCs generated by

$$\{P\}\textbf{if}\ \ S\ \ \textbf{then}\ \ C_1\ \ \textbf{else}\ \ C_2\,\textbf{end if}\{Q\}$$

are

1. the VCs generated from:

$$\{P \wedge S\}C_1\{Q\}$$

2. the VCs generated from:

$$\{P \wedge \neg S\}C_2\{Q\}$$

Generating VCs for Sequences

1. The VCs generated by

$$\{P\}C_1; C_2; \ldots; C_{n-1}; \{R\}C_n\{Q\}$$

(where C_n is **not** an assignment) are:

   1.1 the VCs generated from:

   $$\{P\}C_1; C_2; \ldots; C_{n-1}; \{R\}$$

   1.2 the VCs generated from:

   $$\{R\}C_n\{Q\}$$

2. The VCs generated by

$$\{P\}C_1; C_2; \ldots; C_{n-1}; V := E\{Q\}$$

are the VCs generated from:

$$\{P\}C_1; C_2; \ldots; C_{n-1}; \{Q[E/V]\}$$

Generating VCs for **while**-loop Commands

The VCs generated by

$$\{P\}\textbf{while}\ \ S\ \ \textbf{loop}\ \{R\}\, C\, \textbf{end loop}\{Q\}$$

are

1. $P \to R$
2. $R \wedge \neg S \to Q$
3. the VCs generated from:

$$\{R \wedge S\}C\{R\}$$

- Note that the assertion R appears as both a precondition and a postcondition to the body of the loop, *i.e.* it is *true* before and after each iteration of a loop. Such an assertion is called the **loop invariant**.
- A loop has many invariants. However, the selection of an "appropriate" invariant is crucial to the successful construction of a verification proof.

Quotient-Remainder Revisited

```
(1)        {$true$}
           R:= X;
           Q:= 0;
           {$R = X \wedge Q = 0$}
           while  Y<=R   loop
                {$X = R + (Y * Q)$}
                R:= R - Y;
                Q:= Q + 1;
           end loop;
           {$X = R + (Y * Q) \wedge R < Y$}
```

Apply sequence generation to (1) giving ...

Quotient-Remainder Revisited

```
(2)     {$true$}
        R:= X;
        Q:= 0;
        {$R = X \wedge Q = 0$}

(3)     {$R = X \wedge Q = 0$}
        while  Y<=R   loop
                {$X = R + (Y * Q)$}
                R:= R - Y;
                Q:= Q + 1;
        end loop;
        {$X = R + (Y * Q) \wedge R < Y$}
```

Quotient-Remainder Revisited

```
(2)        {$true$}
           R:= X;
           Q:= 0;
           {$R = X \wedge Q = 0$}
```

Apply sequence generation to (2):

```
(4)        {$true$}
           R:= X;
           {$R = X \wedge 0 = 0$}
```

Apply assignment generation to (4) giving VC1:

$$\mathit{true} \rightarrow (X = X \wedge 0 = 0)$$

Quotient-Remainder Revisited

```
(3)    {$R = X \wedge Q = 0$}
       while Y<=R loop
           {$X = R + (Y * Q)$}
           R:= R - Y;
           Q:= Q + 1;
       end loop;
       {$X = R + (Y * Q) \wedge R < Y$}
```

Apply **while** generation to (3) giving rise to VC2 and VC3:

$$(R = X \wedge Q = 0) \quad \rightarrow \quad (X = R + (Y * Q))$$
$$(X = R + (Y * Q)) \wedge \neg(Y \leq R) \quad \rightarrow \quad (X = R + (Y * Q) \wedge R < Y)$$

and the VCs derived from (5) ...

Quotient-Remainder Revisited

(5)

$$\{X = R + (Y * Q) \land Y \leq R\}$$

```
R:= R - Y;
Q:= Q + 1;
```

$$\{X = R + (Y * Q)\}$$

Apply sequence generation to (5):

(6)

$$\{X = R + (Y * Q) \land Y \leq R\}$$

```
R:= R - Y;
```

$$\{X = R + (Y * (Q + 1))\}$$

Apply assignment generation to (6) giving rise to VC4:

$$(X = R + (Y * Q)) \land Y \leq R) \rightarrow (X = (R - Y) + (Y * (Q + 1)))$$

[next lecture addresses how-to-prove VCs]

Learning Outcomes

- Understand assertion based program specification.
- Understand the distinction between partial and total correctness.
- Understand the relationship between verification conditions and program proof.
- Be able to derive verification conditions for a simple imperative programming language.

Recommended Reading

- "High Integrity Software: The SPARK Approach to Safety and Security" Barnes, J. Addison-Wesley 2003 **[chapters 3 & 11]**
- "Assigning meanings to programs", Floyd, R.W. Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19, 1967.
- "Programming Language Theory and its Implementation", M.J.C.Gordon, Prentice Hall, 1988.
- "An axiomatic basis for computer programming", Hoare, C.A.R. Communications of the ACM, 12, 1969.

Appendix - Inference Rules for Imperative Program Proof

The proceeding VCG rules are underpinned by the following set of inference rules. Note that the rules that follow can be used to verify programs, however, a special purpose theorem prover would be required. In contrast the VCG approach requires a general purpose theorem prover.

The Assignment Axiom

$$\{P[E/V]\}\ V := E\ \{P\}$$

- Where V is an arbitrary variable, E is an expression and P is a statement. Note that $P[E/V]$ denotes the result of substituting E for all occurrences of V in P.
- Some example instances of the assignment axiom:
  $\{Y = 2\}\, X := 2\{Y = X\}$
  $\{X + 1 = N + 1\}\, X := X + 1\{X = N + 1\}$
  $\{A = B\}\, X := A\{X = B\}$

The Sequence Rule

$$\frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}}$$

Given:

$\{X = m \wedge Y = n\}$
`R:=X; X:=Y`
$\{R = m \wedge X = n\}$

$\{R = m \wedge X = n\}$
`Y:=R`
$\{Y = m \wedge X = n\}$

we can use the Sequence-rule to get:

$\{X = m \wedge Y = n\}$
`R:=X; X:=Y; Y:=R`
$\{Y = m \wedge X = n\}$

Conditional Rules

- **If-Then rule:**

$$\frac{\{P \wedge S\}\ C\ \{Q\} \qquad P \wedge \neg S \to Q}{\{P\}\ \textbf{if}\ \ S\ \ \textbf{then}\ \ C\ \textbf{end if}\ \{Q\}}$$

- **If-Then-Else rule:**

$$\frac{\{P \wedge S\}\ C_1\ \{Q\} \qquad \{P \wedge \neg S\}\ C_2\ \{Q\}}{\{P\}\ \textbf{if}\ \ S\ \ \textbf{then}\ \ C_1\ \ \textbf{else}\ \ C_2\ \textbf{end if}\ \{Q\}}$$

The While Rule

$$\frac{\{P \rightarrow R\} \quad \{R \land S\}\ C\ \{R\} \quad \{\neg S \land R \rightarrow Q\}}{\{P\}\ \textbf{while}\ \ S\ \ \textbf{loop}\ \{R\}\ C\ \textbf{end loop}\ \{Q\}}$$

Logical Rules

- Precondition strengthening:

$$\frac{\vdash P \rightarrow P' \qquad \vdash \{P'\}\ C\ \{Q\}}{\vdash \{P\}\ C\ \{Q\}}$$

- Postcondition weakening:

$$\frac{\vdash \{P\}\ C\ \{Q'\} \qquad \vdash Q' \rightarrow Q}{\vdash \{P\}\ C\ \{Q\}}$$

Note: these rules are typically used in conjunction with the proceedings program oriented inference rules.