Rigorous Software Engineering
Hoare Logic and Design by Contracts

Simão Melo de Sousa

RELEASE (UBI), LIACC (Porto)
Computer Science Department
University of Beira Interior, Portugal

2010-2011

Outline

Table of Contents

Floyd-Hoare Logic: Origins

- Robert W. Floyd. "Assigning meaning to programs"'. In Proc. Symp. in Applied Mathematics, volume 19, 1967.
- C. A. R. (Tony) Hoare. "An axiomatic basis for computer programming". Communications of the ACM, 12 (10): 576-585. October 1969.
- Edsger W. Dijkstra, "Guarded command, nondeterminacy and formal derivation of programs". Communication of the ACM, 18 (8):453-457, August 1975.

Main reference for this lecture

Rigorous Software Development, A Practical Introduction to Program Specification and Verification by José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, Simão Melo de Sousa. Series: Undergraduate Topics in Computer Science, Springer. 1st Edition., 2011, XIII, 307 p. 52 illus. ISBN: 978-0-85729-017-5.

Logical state vs. programs

Goal

- Provide means to prove the **correctness** and the **safety** of imperative programs.
- Notion of state: **logical** characterization of the set of values that are manipulated by the program, and their relation.
- Principles: consider each component of the analysed program as a **state transformer**.
- $\Longrightarrow$ Each program has a specific logical behaviour that can be characterized.

Logical state vs. programs

Goal

1. **What?** description of the task to be performed, of the problem to be solved (the relation between input and output).
2. **How?** The effective resolution method
3. The resulting **Artifact**: the program itself, the implementation of the algorithm

Need: **prove** that the *algorithm* (there may be several) and, furthermore, the *implementation* itself **solve** the target problem.

Thus: Hoare logic (and subsequent developments): logic, method and framework to establish the adequacy of the behaviour of a program against its logical specification .

Basic Notion: Hoare Triples

A first Definition

Let be S a program, and P and Q predicates over the variables manipulated by S.

- P: pre-condition, Q: post-condition
- Informal semantics:

  Total: $[P]\ S\ [Q]$ - if P holds in a given state and S is executed in that state, then execution of S will stop, and moreover Q will hold in the final state of execution.

  Partial: $\{P\}\ S\ \{Q\}$ - if P holds in a given state and S is executed in that state, then either execution of S does not stop, or if it does, Q will hold in the final state.

Triples and triples ...

For a given program S, there may be several valid triples.
In general we are interested in the triple $\{P\}\ S\ \{Q\}$ where P is **as weak as possible** and Q is **as strong as possible**.

Some Basics

Examples

- $\{x = 29\}$ `x:= x + 1` $\{x > 20\}$
- $\{x + 1 = 30\}$ `x:= x + 1` $\{x = 30\}$
- $\{x \neq 0\}$ `(x<0)?x:=-x:x:=x` $\{x > 0\}$
- $\{m \leq n\}$ `x := (m+n)/2` $\{m \leq x \leq n\}$

Program Variables and Logic Variables

No triplo $\{x + 1 = 30\}$`x:= x + 1`$\{x = 30\}$

- The variable x from $x + 1 = 30$ is a *logical* variable (its value is immutable)
- The variable x from $x := x + 1$ is a *program* variable (i.e. represents a memory block which contents can be changed)
- Thus, the variable x from the pre-condition represents a value that is not the value represented by the variable x from the post-condition: The former represents the contents of the memory block (represented by the program variable x) before the execution, and the later its value at the end of its execution.

Some Basics

Examples

- $\{x = 29\}$ `x:= x + 1` $\{x > 20\}$
- $\{x + 1 = 30\}$ `x:= x + 1` $\{x = 30\}$
- $\{x \neq 0\}$ `(x<0)?x:=-x:x:=x` $\{x > 0\}$
- $\{m \leq n\}$ `x := (m+n)/2` $\{m \leq x \leq n\}$

Precise Triples

For a given program S, there are a lot of pairs (P, Q) such that $\{P\}\ S\ \{Q\}$. But not all of them are interesting. The pertinent ones are such that P is as weak as possible and Q as strong as possible. (A is weaker than B or B is stronger than A, if $B \implies A$)

Simples Examples

Max

$\{ x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge z \in \mathbb{N} \}$ `(x>y)?z:=x:z:=y` $\{ (x > y \implies z = x) \wedge (x \leq y \implies z = y) \}$

or, alternatively

$\{ x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge z \in \mathbb{N} \}$ `(x>y)?z:=x:z:=y` $\{ (x \leq z \wedge y \leq z \wedge (\forall t \in \mathbb{N},\ (x \leq t \wedge y \leq t) \implies z \leq t))) \}$

Next even

$\{ x \in \mathbb{N} \wedge x = x_0 \}$ `(x mod 2 = 0)?x:=x+2:x:=x+1` $\{ x > x_0 \wedge \forall z \in \mathbb{N}, (z > x_0 \wedge ((par\ z)) \implies x \leq z) \}$

Product as sums

$\{a, b \in \mathbb{N} \wedge b > 0\}$

```
res:=0; i:= 0; while (i!=b) {i:=i+1; res := res + a}
```

$\{res = a \times i \wedge i = b\}$

greatest element

$\{N \in \mathbb{N} \wedge N > 0\}$

```
res:=a[0]; i:=1;
while (i<=N)
   {(a[i]>res)?res:=a[i]:skip; i:=i+1;}
```

$\{N \in \mathbb{N} \wedge N > 0 \wedge \forall x \in \{0 \cdots N\}, res \geq a[x]\}$

Design by Contracts

Principles: Hoare triples as Software Contracts

- Introduced by Bertrand Meyer for Eiffel, as a systematic approach to specifying and implementing object-oriented software components
- Interaction between these components is based on precisely defined specifications of their mutual obligations: the contracts
- $\{C_{caller}\}\ Component\ \{C_{called}\}$: The component *Component* **requires** from its caller the respect of the contract C_{caller}, in order to **ensures** that its execution, when it ends, delivers the contract C_{called}

Contracts allow

- To record details of methods' responsibilities and assumptions
- To document the intended behaviour (specification) of software components
- To avoid having to repeatedly check arguments
- To assign blame across interfaces

Design by Contracts

Contracts are:

- more abstract than code (e.g. sqrt might be implemented using linear search, Newton's method, ...)
- not necessarily checkable (e.g. quantified over infinite types, or just textual strings ...)
- ... but in most cases it is possible to automatically generate verification condition
- (can be) always up-to-date with implementations during development
- Contracts avoid performing time-consuming defensive checks inside production code
- dynamic assertion checking can be automatically generated by a special compiler and turned off at any point

Design by Contracts

Modularity of Reasoning

- As far as verification is concerned: a program is fully verified if each annotated component is individually and successfully verified
- In order to understand this code it suffices to
  - read the methods' contracts
  - instead of looking at "all" the code.

Other features

- Refine design by refining contracts
- Reverse DBC can be used to understand, document, improve, and maintain an existing code base
- Evaluate system quality
  - through rigorous testing (specification-driven)
  - or, through the formal verification of key subsystems

Design by Contracts

- DbC $\Longrightarrow$ Annotating Program Components
- Components = class, methods, modules, functions, ...
- Kinds of contracts:

- safety
  - array index or range
  - overflow (buffer, integer, ...)
  - pointer (null pointer deferencing)
- Functional requirements
- Frame conditions
- Concurrency and timing behaviour
- Abrupt and exceptional behaviour
- static properties (invariants)
  - class
  - module
  - loops

Design by Contracts

What can be done with the contracts?

- Static analysis
- Proof based verification:
  - Model Checking
  - SMT, theorem provers,
  - Proof assistants
- Runtime checks
- Test generation based on contracts

Designing contracts?

- The source code can be obtained by traditional software engineering processes.
- The annotation can be manually introduced, or derived from *protection profiles*, or systematically derived from a requirement analysis.

Design by Contracts

Tools or platforms. Examples:

Algorithm	Why, Dafny
C:	(ACSL), Frama-C, VCC, Havoc
Java:	(JML), LOOP tool, ESC/JAVA, Krakatoa, Key Tools, Bandera, Jive, Jack etc...
C♯:	Spec♯, code contracts
ADA:	Spark, RavenSpark

This lecture

- Our point of view: Program Verification.
- Hoare Logic is at the core of the deductive approach of the DbC.
- An overview of the (classical) foundation of the Hoare Logic and its algorithmic counterpart.
- For a small language. Unfortunately, realistic feature are out of scope in this short introduction.

Table of Contents

The *While* language

Two base types		τ	::=	**bool** | **int**
Phrase types		θ	::=	$\mathbf{Exp}_{\tau}$ | **Comm** | **Term** | **Assert**
$\mathbf{Exp}_{\mathbf{int}}$	$\ni$	e	::=	$\ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x$
				$\mid -e \mid e + e \mid e - e \mid e * e \mid e$ **div** $e \mid e$ **mod** e
$\mathbf{Exp}_{\mathbf{bool}}$	$\ni$	b	::=	**true** | **false** | $!b \mid b$ && $b \mid b \parallel b$
				$\mid e == e \mid e < e \mid e <= e \mid e > e \mid e >= e \mid e != e$
Comm	$\ni$	C	::=	**skip** | C ; $C \mid x := e$ | **if** b **then** C **else** $C \mid$
				| **while** b **do** $\{A\}$ C
				(*We may use the short notation* $(b)?C : C$
				(*for simple conditionals*)
Term	$\ni$	t	::=	$\ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid x$
				$\mid -t \mid t + t \mid t - t \mid t * t \mid t$ **div** $t \mid t$ **mod** t
				$\mid f(t_1, \ldots, t_{\alpha(f)})$
Assert	$\ni$	A	::=	**true** | **false** | $!A \mid A$ && $A \mid A \parallel A$
				$\mid A \rightarrow A \mid$ `Forall` $x.\ A \mid$ `Exists` $x.\ A$
				$\mid t == t \mid t < t \mid t <= t \mid t > t \mid t >= t \mid t != t$
				$\mid p(t_1, \ldots, t_{\alpha(p)})$

First approach: the *While* language

Operational Semantics

- *program state* = a function $s \in \Sigma$ that maps program variables into their current values

$$\Sigma = \mathcal{V} \to \mathbb{Z}$$

$$\begin{array}{rcl} [\![\cdot]\!]_{\mathbf{Exp_{int}}} & : & \mathbf{Exp_{int}} \to \Sigma \to \mathbb{Z} \\ [\![\cdot]\!]_{\mathbf{Exp_{bool}}} & : & \mathbf{Exp_{bool}} \to \Sigma \to \{\mathit{true}, \mathit{false}\} \end{array}$$

- Expressions are interpreted as functions from states to the corresponding domain of interpretation
- Operators have the obvious interpretation
- For our first approach:
  - expression: side effect free
  - expression evaluation is total: for every state, *expression evaluation does not go wrong*

First approach: the *While* language

Natural style operational semantics

• Evaluation relation: $\Downarrow \subseteq \textbf{Comm} \times \Sigma \times \Sigma$.
• $(C, s) \Downarrow s'$ = if C is executed in the initial state s then its execution will terminate, and the final state will be s'.

- if $C = \textbf{skip}$ then $(C, s) \Downarrow s$
- if $C = x := e$ then $(C, s) \Downarrow s[x : [\![e]\!](s)]$
- if $C = C_1 \,;\, C_2$, $(C_1, s) \Downarrow s'$, and $(C_2, s') \Downarrow s''$, then $(C, s) \Downarrow s''$
- if $C = \textbf{if } b \textbf{ then } C_t \textbf{ else } C_f$ then there are two cases:
  - if $[\![b]\!](s) = \mathit{true}$ and $(C_t, s) \Downarrow s_t$, then $(C, s) \Downarrow s_t$
  - if $[\![b]\!](s) = \mathit{false}$ and $(C_f, s) \Downarrow s_f$, then $(C, s) \Downarrow s_f$
- if $C = \textbf{while } b \textbf{ do } \{I\}\ C$ then there are two cases:
  - if $[\![b]\!](s) = \mathit{true}$, $(C, s) \Downarrow s'$, and $(\textbf{while } b \textbf{ do } \{I\}\ C, s') \Downarrow s''$, then $(C, s) \Downarrow s''$
  - if $[\![b]\!](s) = \mathit{false}$ then $(C, s) \Downarrow s$

A simple example

$$C = \textbf{if } x > 0 \textbf{ then } x := 2 * x \,;\, \textbf{while } x < 10 \textbf{ do } \{x <= 10\}\, x := 2 * x \textbf{ else skip}$$

$$\Downarrow$$

$$(C, \{x \mapsto 10\}) \Downarrow \{x \mapsto 20\}$$

Evaluation of terms and interpretation of assertions

$$\begin{array}{rcl} [\![\cdot]\!]_{\textbf{Term}} & : & \textbf{Term} \to \Sigma \to \mathbb{Z} \\ [\![\cdot]\!]_{\textbf{Assert}} & : & \textbf{Assert} \to \Sigma \to \{\mathit{true}, \mathit{false}\} \end{array}$$

- Terms are interpreted very similarly to (program) integer expressions (but functions remain uninterpreted)
- Assertions are interpreted similarly to (program) boolean expressions (but predicates remain uninterpreted)

$$\begin{array}{rcl} I(+) & = & + \\ I(-) & = & - \\ I(*) & = & \times \\ I(\ \textbf{div}\ ) & = & \div \\ I(\ \textbf{mod}\ ) & = & \text{mod} \\ I(==) & = & = \\ I(\ !=) & = & \neq \\ I(<) & = & < \\ I(<=) & = & \leq \\ I(>) & = & > \\ I(>=) & = & \geq \end{array}$$

$$\begin{array}{rcl} [\![c]\!](s) & = & c \\ [\![x]\!](s) & = & s(x) \\ [\![-e]\!](s) & = & -[\![e]\!](s) \\ [\![e_1 \Box e_2]\!](s) & = & [\![e_1]\!](s) \boxdot [\![e_2]\!](s), \text{ where } \boxdot = I(\Box) \\ & & \\ [\![\textbf{true}]\!](s) & = & \mathit{true} \\ [\![\textbf{false}]\!](s) & = & \mathit{false} \\ [\![e_1 \Box e_2]\!](s) & = & [\![e_1]\!](s) \boxdot [\![e_2]\!](s), \text{ where } \boxdot = I(\Box) \\ [\![\ !A]\!](s) & = & \neg [\![A]\!](s) \\ [\![A_1 \ \&\&\ A_2]\!](s) & = & [\![A_1]\!](s) \wedge [\![A_2]\!](s), \\ [\![A_1 \parallel A_2]\!](s) & = & [\![A_1]\!](s) \vee [\![A_2]\!](s), \\ [\![A_1 \to A_2]\!](s) & = & [\![A_1]\!](s) \Longrightarrow [\![A_2]\!](s), \\ [\![\texttt{Forall}\ x.\ A]\!](s) & = & \forall v \in \mathbb{Z}.\ [\![A]\!](s[x \mapsto v]), \quad \text{with } v \text{ fresh} \\ [\![\texttt{Exists}\ x.\ A]\!](s) & = & \exists v \in \mathbb{Z}.\ [\![A]\!](s[x \mapsto v]), \quad \text{with } v \text{ fresh} \end{array}$$

Validity

- Let consider a model $(\mathbb{Z}, I)$, where I is the interpretation function that maps constants to their obvious interpretations as integer numbers, and maps functions and predicates to the corresponding arithmetic functions and comparison predicates (see Logic lectures!)
- Let A be an assertion and $\mathcal{M}$ be a set of assertions. In the context of $(\mathbb{Z}, I)$:

  A *valid*, written $\models A$ if $[\![A]\!](s) = \mathit{true}$ for all states $s \in \Sigma$.
  $\mathcal{M}$ *valid*, written $\models \mathcal{M}$ if $\models A$ holds for every $A \in \mathcal{M}$.

- We assume the existence of *external* means for checking validity of assertions
- These tools should allow us to define theories, i.e. to write axioms concerning the behaviour of the uninterpreted functions and predicates

Specification and Hoare Triples

$$\textbf{Spec} \quad \ni \quad S \quad ::= \quad \{A\}\, C\, \{A\} \mid [A]\, C\, [A]$$

$$[\![\cdot]\!]_{\textbf{Spec}} \quad : \quad \textbf{Spec} \rightarrow \{true, false\}$$

$$[\![\{P\}\, C\, \{Q\}]\!] \quad = \quad \forall s, s' \in \Sigma.\ [\![P]\!](s) \wedge (C, s) \Downarrow s' \Longrightarrow [\![Q]\!](s')$$

$$[\![[P]\, C\, [Q]]\!] \quad = \quad \forall s \in \Sigma.\ [\![P]\!](s) \Longrightarrow (\exists s' \in \Sigma.(C, s) \Downarrow s' \wedge [\![Q]\!](s'))$$

The case for loops

$$\textbf{while}\,(b)\,\textbf{do}\ C \quad \equiv \quad \textbf{if}\ b\ \textbf{then}\ C\ \textbf{else skip};\textbf{while}\,(b)\,\textbf{do}\ C$$

Loop invariants

- Any property whose validity is preserved by executions of the loop's body.
- Since these executions may only take place when the loop condition is true, an invariant of the loop $\textbf{while}\,(b)\,\textbf{do}\ C$ is any assertion I such that $\{I \,\&\&\, b\}\, C\, \{I\}$ is valid, in which case, of course, it also holds that $\{I\}\,\textbf{while}\,(b)\,\textbf{do}\ C\,\{I\}$ is valid
- The validity of $[I \,\&\&\, b]\, C\, [I]$ *does not* however imply the validity of $[I]\,\textbf{while}\,(b)\,\textbf{do}\ C\,[I]$. Proving such a total correctness specification implies proving the *termination* of the loop
- The required notion here is a quantitative one: a loop variant is any *positive integer* expression that is *strictly decreasing* from iteration to iteration.
- The existence of a valid variant for a given loop implies the termination of all possible executions of the loop

Axioms and Rules of Hoare Logic

$$(skip)\quad \frac{}{\{P\}\ \mathbf{skip}\ \{P\}} \qquad (assign)\quad \frac{}{\{Q[x \mapsto e]\}\ x := e\ \{Q\}}$$

$$(seq)\quad \frac{\{P\}\ C_1\ \{R\} \qquad \{R\}\ C_2\ \{Q\}}{\{P\}\ C_1\ ;\ C_2\ \{Q\}}$$

$$(while)\quad \frac{\{I\ \&\&\ b\}\ C\ \{I\}}{\{I\}\ \mathbf{while}\ b\ \mathbf{do}\ \{I\}\ C\ \{I\ \&\&\ !b\}}$$

$$(if)\quad \frac{\{P\ \&\&\ b\}\ C_t\ \{Q\} \qquad \{P\ \&\&\ !b\}\ C_f\ \{Q\}}{\{P\}\ \mathbf{if}\ b\ \mathbf{then}\ C_t\ \mathbf{else}\ C_f\ \{Q\}}$$

$$(conseq)\quad \frac{\{P\}\ C\ \{Q\}}{\{P'\}\ C\ \{Q'\}}\quad \text{if } \models P' \rightarrow P \text{ and } \models Q \rightarrow Q'$$

$$(while_{wf})\quad \frac{[I\ \&\&\ b\ \&\&\ V == n]\ C\ [I\ \&\&\ V < n] \qquad I\ \&\&\ B \longrightarrow V >= 0}{[I]\ \mathbf{while}\ b\ \mathbf{do}\ \{I\}\ C\ [I\ \&\&\ !b]}$$

Derivability in HL denoted by $\vdash$

Soundness of Hoare logic

If $\vdash \{P\}\, C\, \{Q\}$**, then** $[\![\{P\}\, C\, \{Q\}]\!] = \mathit{true}$**.**

Proof.
By induction on the derivation of $\vdash \{P\}\, C\, \{Q\}$. For the *while* case we also proceed by induction on the definition of the evaluation relation.

$\{x >= -100 \;\&\&\; x <= 100\}$
if $x < 0$ **then** $x = x + 100$ **else skip** ; $y = 2 * x$
$\{y >= 0 \;\&\&\; y <= 300\}$

(*seq*)

1. $\{x >= -100 \;\&\&\; x <= 100\}$ **if** $x < 0$ **then** $x = x + 100$ **else skip** $\{x >= 0 \;\&\&\; x <= 150\}$ (*if*)
   - 1.1 $\{x >= -100 \;\&\&\; x <= 100 \;\&\&\; x < 0\}\, x = x + 100\, \{x >= 0 \;\&\&\; x <= 150\}$ (*conseq*)
     - 1.1.1. $\{x >= -100 \;\&\&\; x <= 50\}\, x = x + 100\, \{x >= 0 \;\&\&\; x <= 150\}$ (*assign*)
   - 1.2. $\{x >= -100 \;\&\&\; x <= 100 \;\&\&\; !(x < 0)\}$ **skip** $\{x >= 0 \;\&\&\; x <= 150\}$ (*conseq*)
     - 1.2.1. $\{x >= 0 \;\&\&\; x <= 150\}$ **skip** $\{x >= 0 \;\&\&\; x <= 150\}$ (*skip*)
2. $\{x >= 0 \;\&\&\; x <= 150\}\, y = 2 * x\, \{y >= 0 \;\&\&\; y <= 300\}$ (*assign*)

Side condition to be proved (straightforwardly)

1.1. $\models x >= -100 \;\&\&\; x <= 100 \;\&\&\; x < 0 \rightarrow x >= -100 \;\&\&\; x <= 50$ and
$\models x >= 0 \;\&\&\; x <= 150 \rightarrow x >= 0 \;\&\&\; x <= 150$

1.2. $\models x >= -100 \;\&\&\; x <= 100 \;\&\&\; !(x < 0) \rightarrow x >= 0 \;\&\&\; x <= 150$ and
$\models x >= 0 \;\&\&\; x <= 150 \rightarrow x >= 0 \;\&\&\; x <= 150$

Table of Contents

But...

- Two desirable properties for backward proof construction are missing:
  - Sub-formula property
  - Unambiguous choice of rule
- The consequence rule causes ambiguity. Its presence is however necessary to make possible the application of rules for *skip*, assignment, and while
- An alternative is to distribute the side conditions among the different rules

Goal directed version of HL

Axioms and Rules of Hoare Logic

$$\frac{}{\{P\}\,\textbf{skip}\,\{Q\}} \quad \text{if } \models P \rightarrow Q \qquad \frac{}{\{P\}\,x := e\,\{Q\}} \quad \text{if } \models P \rightarrow Q[e/x]$$

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

$$\frac{\{I \;\&\&\; b\}\,C\,\{I\}}{\{P\}\,\textbf{while}\ b\ \textbf{do}\,\{I\}\,C\,\{Q\}} \quad \text{if } \models P \rightarrow I \text{ and } \models I \;\&\&\; !b \rightarrow Q$$

$$\frac{\{P \;\&\&\; b\}\,C_t\,\{Q\} \qquad \{P \;\&\&\; !b\}\,C_f\,\{Q\}}{\{P\}\,\textbf{if}\ b\ \textbf{then}\ C_t\ \textbf{else}\ C_f\,\{Q\}}$$

Goal directed Version of HL

Derivability in goal oriented HL denoted by $\vdash_g$

soundness properties

- If $\vdash_g \{P\}\, C\, \{Q\}$ and both $\models P' \rightarrow P$ and $\models Q \rightarrow Q'$ hold, then $\vdash_g \{P'\}\, C\, \{Q'\}$.
  (Proof. By induction on the derivation of $\vdash_g \{P\}\, C\, \{Q\}$).
- For any program C and assertions P, Q,$\vdash_g \{P\}\, C\, \{Q\}$ iff $\vdash \{P\}\, C\, \{Q\}$.

But...

The *seq* rule

In the goal directed strategy, one has still to guess the intermediary assertion in the *seq* rule

A strategy for proofs

- Focus on the command and postcondition; guess an appropriate precondition
- In the sequence rule, we obtain the intermediate condition from the postcondition of the second command
- We do this by always choosing the **weakest precondition** (for the given postcondition)
- i.e., in rules for *skip*, *assignment*, and *while*, the precondition is determined by looking at the side condition and choosing the weakest condition that satisfies it
- A note: the dual approach (forwardly, from pre-conditions to post-conditions) is also possible.

Mechanization: Program Verification Architecture

How can a proof tool be used for verifying programs with Hoare Logic using the Weakest Preconditions strategy? Two possibilities:

- Encode Hoare Logic directly in proof tool and reason about program constructs
- Two-phase architecture:
  - use Hoare Logic to construct a set of verification conditions
  - use a general-purpose proof tool to discharge verification conditions

Second approach is much more flexible

Verification Conditions

- VCs are purely first-order, not containing program constructs.
- Can be checked / discharged using any standard proof tool (theorem prover or proof assistant) with support for the data types of the language.
- Modifications in the language are only reflected in the first component, not in the proof tool
- Moreover it is possible to use a multi-prover approach

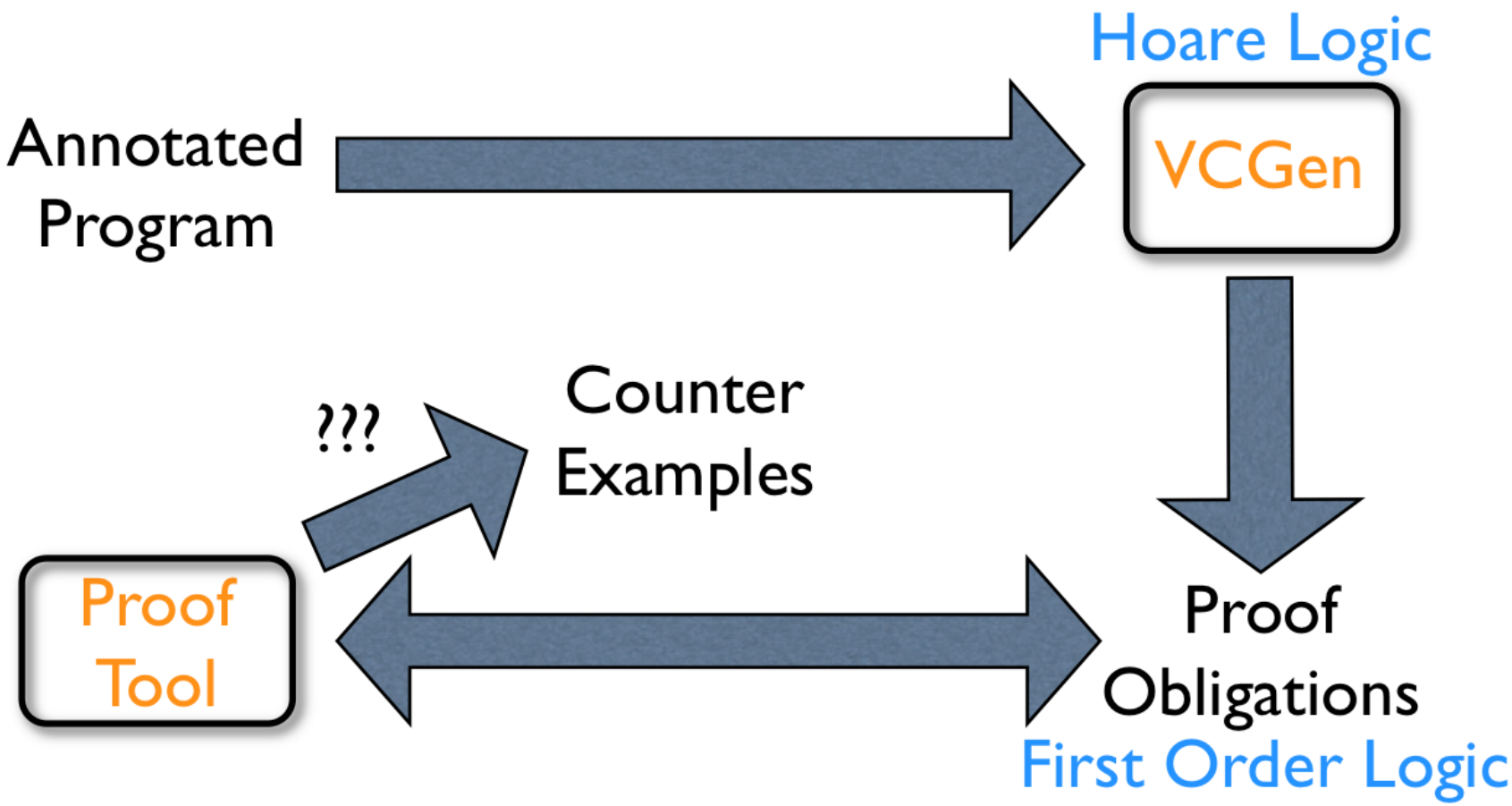
Two-phase Architecture

1. Given a Hoare triple $\{P\}\, C\, \{Q\}$, we mechanically produce a derivation with P C Q as conclusion, assuming that all its side conditions are valid.
2. Each side condition of the form $\models A \rightarrow B$ generated in step 1 must now be checked. To that effect the first-order formula $A \rightarrow B$ is exported to a proof tool. Such a formula is called a **verification condition** (VC).
3. If all verification conditions can be proved by a sound proof tool, then $\{P\}\, C\, \{Q\}$ is a valid Hoare triple. If at least one condition is shown not to be valid, then this is evidence that the triple is also not valid.

Some notes on this architecture

- Note that the HL *proof tree* can always be constructed (explicitly or virtually)
- But the VCs may not all be dischargeable: automatic prover may be able to find a counter-example... or interactive proof may not succeed
- What does it mean when at least one VC is not valid?(the verification of the program has failed) $\Longrightarrow$ Errors in program, specification, or annotations

General Architecture of a Program Verification System



An Architecture for Verification

- Our next step is then to mechanize the construction of a derivation, following the WP strategy.
- The result will be an algorithm (called a Verification Conditions Generator, VCGen) that does not even explicitly construct the proof tree; it just outputs the set of verification conditions

Weakest Preconditions Mechanized

Principles

- $\{P\}\, C\, \{Q\}$ Given program C and a postcondition Q, we can calculate an assertion $\mathrm{wp}\,(C, Q)$ such that $\{\mathrm{wp}\,(C, Q)\}\, C\, \{Q\}$ is valid
- and moreover
  if $\{P\}\, C\, \{Q\}$ is valid for some P then P is stronger than $\mathrm{wp}\,(C, Q)$ (i.e. $P \rightarrow \mathrm{wp}\,(C, Q)$).
- Thus $\mathrm{wp}\,(C, Q)$ is the **weakest precondition** that grants the truth of postcondition Q after execution of C.

VCGen algorithm

$$
\begin{aligned}
\mathrm{wp}\,(\textbf{skip}, Q) &= Q \\
\mathrm{wp}\,(x := e, Q) &= Q[e/x] \\
\mathrm{wp}\,(C_1; C_2, Q) &= \mathrm{wp}\,(C_1, \mathrm{wp}\,(C_2, Q)) \\
\mathrm{wp}\,(\textbf{if } b \textbf{ then } C_t \textbf{ else } C_f, Q) &= (b \rightarrow \mathrm{wp}\,(C_t, Q)) \;\&\&\; (\;!b \rightarrow \mathrm{wp}\,(C_f, Q)) \\
\mathrm{wp}\,(\textbf{while } b \textbf{ do } \{I\}\; C, Q) &= I
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{VC}(\textbf{skip}, Q) &= \emptyset \\
\mathrm{VC}(x := e, Q) &= \emptyset \\
\mathrm{VC}(C_1; C_2, Q) &= \mathrm{VC}(C_1, \mathrm{wp}\,(C_2, Q)) \;\cup\; \mathrm{VC}(C_2, Q) \\
\mathrm{VC}(\textbf{if } b \textbf{ then } C_t \textbf{ else } C_f, Q) &= \mathrm{VC}(C_t, Q) \;\cup\; \mathrm{VC}(C_f, Q) \\
\mathrm{VC}(\textbf{while } b \textbf{ do } \{I\}\; C, Q) &= \{(I \;\&\&\; b) \rightarrow \mathrm{wp}\,(C, I), (I \;\&\&\; !b) \rightarrow Q\} \\
&\quad \cup\, \mathrm{VC}(C, I)
\end{aligned}
$$

$$
\mathrm{VCG}(\{P\}\, C\, \{Q\}) = \{P \rightarrow \mathrm{wp}\,(C, Q)\} \cup \mathrm{VC}(C, Q)
$$

VCGen algorithm

Soundness of VCGen

Let $C \in$ **Comm** and $P, Q \in$ **Assert** such that $\models \mathrm{VCG}(\{P\}\, C\, \{Q\})$.
Then $\vdash_g \{P\}\, C\, \{Q\}$
Proof.
It suffices to prove that there exists a derivation in the goal directed HL that has as side conditions exactly the set given by $\mathrm{VCG}(\{P\}\, C\, \{Q\})$.

An example

Consider

$$\begin{array}{l}\textbf{fact} \doteq \\ \quad f := 1\,;\; i := 1\,; \\ \quad \textbf{while}\ i <= n\ \textbf{do}\ \{f == fact(i-1)\ \&\&\ i <= n+1\} \\ \quad\quad f := f * i\,; \\ \quad\quad i := i+1\end{array}$$

The universal closures of

$$\mathrm{VCG}(\{n >= 0\}\ \textbf{fact}\ \{f == fact(n)\})$$

are

1. $\texttt{Forall}\ n.\ (n >= 0 \rightarrow 1 == fact(1-1)\ \&\&\ 1 <= n+1)$
2. $\texttt{Forall}\ i, n.\ (f == fact(i-1)\ \&\&\ i <= n+1\ \&\&\ i <= n \rightarrow f * i == fact(i+1-1)\ \&\&\ i+1 <= n+1)$
3. $\texttt{Forall}\ i, f, n.\ (f == fact(i-1)\ \&\&\ i <= n+1\ \&\&\ i > n \rightarrow f == fact(n))$