Software quality assurance

- What are we assuring?
- Why are we assuring it?
- How do we assure it?
- How do we know we have assured it?

What are we assuring?

- Validation: building right system?
- Verification: building system right?
- Presence of good properties?
- Absence of bad properties?
- Identifying errors?
- Confidence in the absence of errors?

Why are we assuring it?

- Business reasons
- Ethical reasons
- Professional reasons
- Personal satisfaction
- Legal reasons
- Social reasons
- Economic reasons
- …
How do we assure it?

Product
[ex: prescription drugs]

Process
[ex: ISO 9000]

People
[ex: medical licensing]

How do we know we have assured it?

- Depends on "it"
- Depends on what we mean by “assurance”
- ...

Our focus

- Primarily on the product – testing, verification, etc.
  - And primarily on “built the system right?”
- Some on the process – walkthroughs, code reviews, etc.

Foundation: program correctness

- Relatively few programs are proven correct
  - Hard, expensive, and usually uni-dimensional
- The language and “way of thinking” is important, and many recent testing and anomaly checking technologies are heavily reliant on this foundation
Basics of program correctness

- Make precise the meaning of programs
- In a logic, write down (this is often called the specification)
  - the effect of the computation that the program is required to perform (the postcondition \( Q \))
  - any constraints on the input environment to allow this computation (the precondition \( P \))
- Associate precise (logical) meaning to each construct in the programming language (this is done per-language, not per-program)
- Reason (usually backwards) that the logical conditions are satisfied by the program \( S \)
- A Hoare triple is a predicate \( \{ P \} S \{ Q \} \) that is true whenever \( P \) holds and the execution of \( S \) guarantees that \( Q \) holds

Examples

- \( \{ \text{true} \} \)
  
  \[
  y := x * x; \\
  \{ y \geq 0 \}
  \]

- \( \{ x \neq 0 \} \)
  
  \[
  y := x * x; \\
  \{ y > 0 \}
  \]

- \( \{ x > 0 \} \)
  
  \[
  x := x + 1; \\
  \{ x > 1 \}
  \]

More examples

- \( \{ x = k \} \)
  
  \[
  \text{if } (x < 0) \ x := -x \text{ endif; } \\
  \{ \ ? \ }
  \]

- \( \{ \ ? \} \)
  
  \[
  x := 3; \\
  \{ x = 8 \}
  \]

Strongest postconditions

[example from Aldrich and perhaps from Leino]

The following are all valid Hoare triples

- \( \{ x = 5 \} x := x \times 2 \{ \text{true} \} \)
- \( \{ x = 5 \} x := x \times 2 \{ x > 0 \} \)
- \( \{ x = 5 \} x := x \times 2 \{ x = 10 \mid x = 5 \} \)
- \( \{ x = 5 \} x := x \times 2 \{ x = 10 \} \)

- Which is the most useful, interesting, valuable? Why?
**Weakest preconditions**

(example from Aldrich and perhaps from Leino)

Here are a number of valid Hoare Triples

- \( \{ x = 5 \land y = 10 \} \ z := x \div y \{ z < 1 \} \)
- \( \{ x < y \land y > 0 \} \ z := x \div y \{ z < 1 \} \)
- \( \{ y \neq 0 \land x \div y < 1 \} \ z := x \div y \{ z < 1 \} \)

- The last one is the most useful because it allows us to invoke the program in the most general condition
- It is called the weakest precondition, \( \text{wp}(S,Q) \) of \( S \) with respect to \( Q \)
  - \( \{ P \} S \{ Q \} \) and for all \( P' \) such that \( P' \Rightarrow P \), then \( P \) is \( \text{wp}(S,Q) \)

**Sequential execution**

- What if there are multiple statements
  - \( \{ P \} S1;S2 \{ Q \} \)
- We create an intermediate assertion
  - \( \{ P \} S1 (A) S2 \{ Q \} \)
- We reason (usually) backwards to prove the Hoare triples
- A formalization of this approach essentially defines the ; operator in most programming languages

**Conditional execution**

- \( \{ P \} \) if \( C \) then \( S1 \) else \( S2 \) endif
- \( \{ P \} \) if \( C \land x \geq y \) then max := \( x \) else max := \( y \) fi
- \( \{ P \land C \} S1 \{ Q \} \land \{ P \land \neg C \} S2 \{ Q \} \)

**Hoare logic rule: conditional**

\[
\{ P \} \text{ if } C \text{ then } S1 \text{ else } S2 \{ Q \} \\
\equiv \{ P \land C \} S1 \{ Q \} \land \{ P \land \neg C \} S2 \{ Q \}
\]
Be careful!

- \{true\}
  \texttt{max := abs(x)+abs(y);}
  \{max \geq x \wedge max \geq y\}

- This predicate holds, but we don’t “want” it to
  - The postcondition is written in a way that permits satisfying programs that don’t compute the maximum
  - In essence, every specification is satisfied by an infinite number of programs and vice versa

- The “right” postcondition is
  \{- (max = x \vee max = y)
   \wedge (max \geq x \wedge max \geq y)\}

Assignment statements

- We’ve been highly informal in dealing with assignment statements
- What does the statement \texttt{x := E} mean?
  \(- \{Q(E)\} x := E \{Q(x)\}\)
  \(- \text{If we knew something to be true about } E \text{ before the assignment, then we know it to be true about } x \text{ after the assignment (assuming no side-effects)}\)

Examples

- \{y > 0\}
  \texttt{x := y}
  \{x > 0\}

- \{x > 0\}
  \[Q(E) \equiv x + 1 > 1 \equiv x > 0\]
  \texttt{x := x + 1;}
  \{x > 1\}
  \[Q(x) \equiv x > 1\]

More examples

- \{qd\}
  \texttt{x := y + 5}
  \{x > 0\}

- \{x = A \wedge y = B\}
  \texttt{t := x;}
  \texttt{x := y;}
  \texttt{y := t}
  \{x = B \wedge y = A\}
Loops

- \{P\} while B do S \{Q\}
- We can try to unroll this into
  \{P \land \neg B\} S \{Q\} \lor
  \{P \land B\} S \{Q \land \neg B\} \lor
  \{P \land B\} S \{Q \land B\} S \{Q \land \neg B\} \lor ...
- But we don’t know how far to unroll, since we don’t know how many times the loop can execute.
- The most common approach to this is to find a loop invariant, which is a predicate that
  - is true each time the loop head is reached (on entry and after each iteration)
  - and helps us prove the postcondition of the loop
  - It approximates the fixed point of the loop

Loop invariant for \{P\} while B do S \{Q\}

- Find I such that
  - P \Rightarrow I
  - B \land I \Rightarrow S \{I\}
  - \neg B \land I \Rightarrow Q
- Loop termination proves Q

Example
(n > 0)
x := a[1];
i := 2;
while i <= n do
  if a[i] > x then x := a[i];
i := i + 1;
end;
{x = \text{max}(a[1],...,a[n])}

Termination

- Proofs with loop invariants do not guarantee that the loop terminates, only that it does produce the proper postcondition if it terminates — this is called weak correctness.
- A Hoare triple for which termination has been proven is strongly correct.
- Proofs of termination are usually performed separately from proofs of correctness, and they are usually performed through well-founded sets
  - In this example it’s easy, since i is bounded by n, and i increases at each iteration
- Historically, the interest has been in proving that a program does terminate, but many important programs now are intended not to terminate.

Correctness of data structures

- Primarily due to Hoare; figures from Wulf et al.
- Prove the specifications on the abstract operations (e.g., Push_a).
- Prove the specifications on the concrete operations (e.g., Push_c).
- Prove the relation between abstract and concrete operations (e.g., \{S_a\}).

Example
\{\text{full}(S_c)\} \Rightarrow \{\text{full}(R(S_c))\}
S_a := \langle x \rangle \Rightarrow S_c := \langle x \rangle
R(S_c) := \langle x \rangle
R(S_a) := \langle x \rangle
S_a := \langle x \rangle
\Rightarrow S_c := \langle x \rangle
So what?

- It lays a foundation for
  - Thinking about programs more precisely
  - Applying techniques like these in limited, critical situations
  - Development of some modern design, specification and analysis approaches that seem to have value in more situations
  - Basis for many testing and analysis approaches

Testing vs. proving

- Dynamic
  - Builds confidence
    - Can only show the presence of bugs, not their absence
    - Used widely in practice
    - Costly

- Static
  - It's a proof
    - Proofs are human processes that aren't foolproof
    - Applicability is practically limited
    - Extremely costly

Brief (and informal) aside

- Dynamic techniques are unattractive because they are "unsound" — you can believe something is true when it's not
- Static techniques are unattractive because they are often very costly — and they may lead you to confuse the checked property for other desirable properties
- The truth is that they should be considered to be complementary, not competitive

Testing

- In any case, testing is by far the dominant approach to assessing software products
Two kinds of improvements

• One goal is to improve testing to increase the quality of the software that is produced
• Another goal is to reduce the costs of testing while maintaining the current quality of the software that is produced

Terminology

• A failure occurs when a program doesn’t satisfy its specification
• A fault occurs when a program's internal state is inconsistent with what is expected (usually an informal notion)
• A defect is the code that leads to a fault (and perhaps to a failure)
• An error is the mistake the programmer made in creating the defect

More terminology

• A test case is a specific set of data that exercises the program
• A test suite is a set of test cases
• Old terminology
  – A test case (suite) fails if it demonstrates a problem
• New terminology
  – A test case (suite) succeeds if it demonstrates a problem

Root cause analysis

• Tries to track a failure to an error
• Identifying errors is important because it can
  – help identify and remove other related defects
  – help a programmer (and perhaps a team) avoid making the same or a similar error again
Kinds of testing

- Unit
- White-box
- Black-box
- Gray-box
- Bottom-up
- Top-down
- Boundary condition
- Syntax-driven

- Big bang
- Integration
- Acceptance
- Stress
- Regression
- Alpha
- Beta
- Fuzz

In groups

- The program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is isosceles, equilateral, or scalene.
- Write a set of test cases that you feel would adequately test this program

In practice

- 13 kinds of errors were found in actual programs
- When highly experienced programmers are given this example, on the average they figure out about half of the kinds of errors

The lucky thirteen...

- Valid scalene triangle
- Valid equilateral triangle
- Valid isosceles triangle
- Three cases that represent valid isosceles triangles in all permutations
- One side is zero
- One side is negative
- 3 positive integers where two sum to the third
- All permutations of the previous case
The remaining ones

- 3 positive integers where two sum to less than the third
- 3 permutations of the previous case
- All sides are zero
- A non-integer side
- An incorrect number of inputs

Questions?