Software Quality Assurance in a larger scope:
- Vocabulary
- Fault, error, failure
- Concepts of software quality assurance (next to testing)
- Formal Program Verification
- Deterministic Programs
  - Syntax
  - Semantics
  - Termination, Divergence
  - Correctness of deterministic programs
    - Partial correctness
    - Total correctness
- Proof System PD
- The Verifier for Concurrent C

Software Quality Assurance — See: Quality assurance.

IEEE 610.12 (1990)

Quality assurance — (1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements.
(2) A set of activities designed to evaluate the process by which products are developed or manufactured.

Note: In order to trust a product, it can be built well, or proven to be good (at best: both) — both is QA in the sense of (1).
A fault can manifest itself as an error within the considered element and the error can ultimately cause a failure. An error can arise as a result of unforeseen operating conditions or due to a failure, especially soft-errors. Example: A fault manifests itself as an error within the considered element, and the error can ultimately cause a failure.
Definition.

- \( x \equiv (x_1 + y, x_2) \equiv (x_3, (x_4 + x_5), x_6) \)

Correctness formula denoted by \( S \)

- \( x \equiv (x_1 + y, x_2) \equiv (x_3, (x_4 + x_5), x_6) \)

Post-condition \( \bullet \)

\( x \rightarrow y \) while \( \sigma \)

- \( x \equiv (x_1 + y, x_2) \equiv (x_3, (x_4 + x_5), x_6) \)

Transition sequence \( M \)

- \( x \equiv (x_1 + y, x_2) \equiv (x_3, (x_4 + x_5), x_6) \)

while \( \sigma \)

- \( x \equiv (x_1 + y, x_2) \equiv (x_3, (x_4 + x_5), x_6) \)

\( M \times S \rightarrow \sigma \)

- \( x \equiv (x_1 + y, x_2) \equiv (x_3, (x_4 + x_5), x_6) \)

Definition.

- \( x \equiv (x_1 + y, x_2) \equiv (x_3, (x_4 + x_5), x_6) \)

Correctness of while program

- \( x \equiv (x_1 + y, x_2) \equiv (x_3, (x_4 + x_5), x_6) \)

Another Example
The rule '[(A2)]

\[ x = a \land y = b \implies (x = y) \iff (a = b) \]

Proof of (1)
such that and $x \geq a$.

Then, $y = 0$ using $b = 0$.

We have
\[ x = x + 0 = x. \]

Thus, $PD$ is sound.

Therefore, partially correct, while $b = 0$. 

Back to the Example Proof.
The V erifier for Concurrent C

= return value of procedure (useful for post-conditions)

\( \old(v) \) — the value of \( \old(v) \) when procedure was called (useful for post-conditions)

\( \thread_local(&v) \) — procedure which global variables it is allowed to write to (also checked by VCC)

\( \result \) — return value of procedure (useful for post-conditions)

\( \expr \) — the value of \( \old(v) \) (in pre-conditions)

Interpreted C programs; we need to declare for each

Modular Reasoning

\( \text{assert} \) — (basically) a C expression

Transform function

\( \text{display}(\text{sum}) \);
\( \text{sum} = \text{add}(x,y) \);
\( x = \text{read_number}() \);
\( \text{main}() \) {
  \( \text{int} x \);
  \( \text{int} y \);
  \( \text{int} z \);
  \( \text{scanf}(%d, \&x) \);
  \( \text{scanf}(%d, \&y) \);
  \( \text{scanf}(%d, \&z) \);
  \( \text{printf}(\text{"%d\n"}, \text{sum}) \);
  \}

Extended PD by axiom:

\[ p \Rightarrow q \]

We can add another rule for calls of functions

Uses

Special syntax

Special expressions

Verify a C program with a BMC solver

The V erifier for Concurrent C

Modular reasoning

Generates \( \neg (\exists \sigma \in \Sigma) B \) from \( B \)

Computer science
Example program DIV:
http://rise4fun.com/Vcc/4Kqe

Interpretation of Results

• VCC says: "verification succeeded"
  We can only conclude that the tool — under its interpretation of the C-standard, under its platform assumptions (32-bit), etc. — "thinks" that it can prove
  \( \{ p \} \text{DIV} \{ q \} \).
  Can be due to an error in the tool!
  (That's a false negative then.)
  Yet we can ask for a printout of the proof and check it manually (hardly possible in practice) or with other tools like interactive theorem provers.
  Note:
  \( \{ false \} f \{ q \} \) always holds.
  That is, a mistake in writing down the pre-condition can make errors in the program go undetected.

• VCC says: "verification failed"
  May be a false positive. The tool does not provide counter-examples in the form of a computation path, it (only) gives hints on input values satisfying \( p \) and causing a violation of \( q \).
  \( \rightarrow \) try to construct a (true) counter-example from the hints.
  or:
  \( \rightarrow \) make pre-condition \( p \) or loop-invariant(s) stronger, and try again.

• Other case: "timeout" etc. — completely inconclusive outcome.

VCC Features

• For the exercises, we use VCC only for sequential, single-thread programs.
• VCC checks a number of implicit assertions:
  • no arithmetic overflow in expressions (according to C-standard),
  • array-out-of-bounds access,
  • NULL-pointer dereference,
  • and many more.
• VCC also supports:
  • concurrency: different threads may write to shared global variables; VCC can check whether concurrent access to shared variables is properly managed;
  • data structure invariants: we may declare invariants that have to hold for, e.g., records (e.g. the length field \( l \) is always equal to the length of the string field \( str \)); those invariants may temporarily be violated when updating the data structure.
  • and much more.
• Verification does not always succeed:
  • The backend SMT-solver may not be able to discharge proof-obligations (in particular non-linear multiplication and division are challenging);
  • In many cases, we need to provide loop invariants manually.

Tell Them What You've Told Them...

