Contents of the Block “Quality Assurance”

(i) Introduction and Vocabulary
• correctness illustrated
• vocabulary: fault, error, failure
• three basic approaches

(ii) Formal Verification
• Hoare calculus
• Verifying C Compiler (VCC)
• over- / under-approximations

(iii) (Systematic) Tests
• systematic test vs. experiment
• classification of test procedures
• model-based testing
• glass-box tests: coverage measures

(iv) Runtime Verification

(v) Review

(vi) Concluding Discussion
• Dependability

Last Lecture:
• Completed the block “Architecture & Design”

This Lecture:
• Educational Objectives:
 Capabilities for following tasks/questions.
• When do we call a software correct?
• What is fault, error, failure? How are they related?
• What is formal and partial correctness?
• What is a Hoare triple (or correctness formula)?
• Is this program (partially) correct?
• Prove the (partial) correctness of this WHILE-program using PD.
• What can we conclude from the outcome of tools like VCC?

Content:
• Introduction, Vocabulary
• WHILE-program semantics, partial & total correctness
• Correctness proofs with the calculus PD.
• The Verifying C Compiler (VCC)
Definition. A software specification is a finite description $S$ of a (possibly infinite) set $\llbracket S \rrbracket$ of softwares, i.e. $\llbracket S \rrbracket = \{ (S_1, \llbracket \cdot \rrbracket_1), \ldots \}$. The (possibly partial) function $\llbracket \cdot \rrbracket : S \mapsto \llbracket S \rrbracket$ is called interpretation of $S$.

We define: Software $S$ is correct wrt. software specification $S$ if and only if $(S, \llbracket \cdot \rrbracket) \in \llbracket S \rrbracket$.

• Note: no specification, no correctness. Without specification, $S$ is neither correct nor not correct — it’s just some software then.
fault — abnormal condition that can cause an element or an item to fail.

Note: Permanent, intermittent and transient faults (especially soft-errors) are considered.

Note: An intermittent fault occurs time and time again, then disappears. This type of fault can occur when a component is on the verge of breaking down or, for example, due to a glitch in a switch. Some systematic faults (e.g. timing marginalities) could lead to intermittent faults.

ISO 26262 (2011)

error — discrepancy between a computed, observed or measured value or condition, and the true, specified, or theoretically correct value or condition.

Note: An error can arise as a result of unforeseen operating conditions or due to a fault within the system, subsystem or, component being considered.

Note: A fault can manifest itself as an error within the considered element and the error can ultimately cause a failure.

ISO 26262 (2011)

failure — termination of the ability of an element, to perform a function as required.

Note: Incorrect specification is a source of failure.

ISO 26262 (2011)

We want to avoid failures, thus we try to detect faults, e.g. by looking for errors.

• If we are lucky, the requirement specification is a constraint on computation paths.
• LSC 'buy water' is such a software specification $S$.
• It denotes all controller softwares which "faithfully" sell water.
  (Or which refuse to accept C50 coins, or block the 'WATER' button).
• Formally \[
\llbracket \text{buy water} \rrbracket \text{spec} = \{ S | \llbracket S \rrbracket \text{satisfies 'buy water'} \}.
\]
• In pictures:
  \[
(\Sigma \times A) \omega \text{all computation paths satisfying 'buy water' of one acceptable software } S
\]
  \[
(\Sigma \times A) \omega \text{all computation paths satisfying 'buy water' of one not acceptable software } S
\]
• Then we can check correctness of a given software $S$ by examining its computation paths $\llbracket S \rrbracket$.
(iv) We use $\sigma$ since

$$\langle 0 \rangle := 1;$$

and $a \neq 0$.

Example

Note that $\langle 1 \rangle := 0$.

Correctness Formulae ("Hoare Triples")

Given a Boolean expression $\ast$ and a transition sequence $\llbracket \sigma \rrbracket$, the final state of that computation satisfies $\ast$ if and only if $\llbracket \sigma \rrbracket \in \ast$.
The program $\langle E, \sigma \rangle$ is defined as:

- $E$ is an expression or a statement.
- $\sigma$ is the initial state.

### Examples

#### Example 1

- $E = x + y$
- $\sigma = \{ x = 1 \}$

#### Example 2

- $E = \text{if } x > 0 \text{ then } y := x \text{ else } z := x \text{ fi}$
- $\sigma = \{ x = 1 \}$

### Correctness of Programs

- The correctness of a program is denoted by $\langle E, \sigma \rangle \rightarrow \langle \tau, \sigma' \rangle$.
- $\tau$ is the final state.
- $\sigma'$ is the updated state.

### Semantics of Total Correctness

- $\langle E, \sigma \rangle \rightarrow^{*} \langle \tau, \sigma' \rangle$ means that there is a sequence of steps leading from $\langle E, \sigma \rangle$ to $\langle \tau, \sigma' \rangle$.

### Example

- $E = \text{while } x > 0 \text{ do } x := x - 1 \text{ od}$
- $\sigma = \{ x = 1 \}$

- The program diverges if $x = 0$.

### Conclusion

- The total correctness of a program is defined as $\langle E, \sigma \rangle \rightarrow^{\text{tot}} \langle \tau, \sigma' \rangle$.
- $\langle E, \sigma \rangle \rightarrow^{\text{tot}} \langle \tau, \sigma' \rangle$ means that $E$ is total with respect to $\sigma$.

### Fun Facts

- The function $\sigma$ has exactly one element, $S$.
- In the sense of $S \perp \{ a \}$, $\sigma$ diverges from $a$.

### Notes

- The state $S$ is the function $\langle 0 \rangle := \langle 1 \rangle$.
The proof needs a loop invariant, which we shall exhibit by hand.

The loop invariant is that $r < y$ and $x + q ≥ r$ while

```
while (A2);

r < y ∧ ¬B2
```

and

```
if 1
```

by (A2).

By (R4)

```
1
```

and

```
2
```

1

hence

```
3
```

 loops.

We have $A3$.

The proof is therefore complete.

Example Proof

Example Proof

Example Proof
Proof: (1) by (R6) using

\[ x \geq y + (0 \lor x \land y) \]

...
Why Assertions?

• Available in standard libraries of many programming languages, e.g. C:

```c
#include <assert.h>

void assert(scalar expression);
```

DESCRIPTION

[...] the macro assert() prints an error message to standard error and terminates the program by calling abort(3) if expression is false (i.e., compares equal to zero).

The purpose of this macro is to help the programmer find bugs in his program. The message "assertion failed in file foo.c, function do_bar(), line 1287" is of no help at all to a user.

VCC Syntax Example

```c
#include <vcc.h>

int q, r;

void div(int x, int y)
{
    (requires x >= 0 && y >= 0)
    (ensures q * y + r == x && r < y)
    (writes &q)
    (writes &r)
    {
        q = 0;
        r = x;
        while (r >= y)
        {
            r = r - y;
            q = q + 1;
        }
    }
}
```

DIV ≡ q := 0; r := x; while r ≥ y do r := r − y; q := q + 1 done

The Verifying C Compiler

• The Verifying C Compiler (VCC) basically implements Hoare-style reasoning.

• Special syntax:
  - `#include <vcc.h>`
  - `(requires p)` — pre-condition, `p` is a C expression
  - `(ensures q)` — post-condition, `q` is a C expression
  - `(invariant expr)` — loop invariant, `expr` is a C expression
  - `(assert p)` — intermediate invariant, `p` is a C expression
  - `(writes &v)` — VCC considers concurrent C programs; we need to declare for each procedure which global variables it is allowed to write to (also checked by VCC)

• Special expressions:
  - `\thread local(&v)` — no other thread writes to variable `v` (in pre-conditions)
  - `\old(v)` — the value of `v` when procedure was called (useful for post-conditions)
  - `\result` — return value of procedure (useful for post-conditions)

VCC Web-Interface
• 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.
• 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.

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!
  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: $|p| = \{false\} \{q\}$ always holds — so a mistake in writing down the pre-condition can provoke a false negative.
• VCC says: "verification failed"
  One case: "timeout" etc. — completely inconclusive outcome.
  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 $q$. May be a false negative if these inputs are actually never used. Make pre-condition $p$ stronger, and try again.

