Introduction and Vocabulary

- Test case, test suite, test execution.
- Positive and negative outcomes.

Limits of Software Testing

- Glass-Box Testing
  - Statement-, branch-, term-coverage.

Other Approaches

- Model-based testing
- Program Verification
  - partial and total correctness
  - Proof System PD
- Runtime verification.

Review

VL 14...
VL 15...
VL 16...
VL 17...

Software Quality Assurance

- Project management
- Organisational software examination
- Analytic examination by humans
- Non-mechanical inspection
- Review
  - Manual proof
  - Computer-aided human examination
  - Semi-mechanical examination
  - e.g. interactive prover
  - Examination with computer
- Analyse and check against rules
- Consistency checks
- Quantitative examination
- Dynamic checking (test)
- Execute formal verification
- Prove constructive software engineering
- e.g. code generation (Ludewig and Lichter, 2013)

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
  - Modular reasoning
  - Return values / old values
  - Assertions
Example

Deterministic Programs

Example

Example

Example
We say $q$ wrt. partially correct $⊆ S$ if and only if $p$ $|= S$ correctness.

Syntax

Lemma.

Consider the program $x := y$, $y$, $x$ $|$ $x$ (starting in $S$).

(i) A $b$ $|$ $x$ $|$ $S$ $|$ $y$ $|$ $x$ is the function $\sigma$:

(ii) Let $x$ $|$ $y$ $|$ $S$ $|$ $τ$ $|$ $x$, $σ$ $|$ $S$ $|$ $τ$ $|$ $x$.

(iii) We use divergence $S$ $|$ $τ$ $|$ $x$.

(iv) While $x$ $|$ $y$ $|$ $S$ $|$ $τ$ $|$ $x$ $|$ $b$ $|$ $x$ $|$ $b$ and a $|$ $x$ $|$ $b$ $|$ $x$ $|$ $b$.

(v) Consider $E$, $x$ $|$ $y$ $|$ $S$ $|$ $τ$ $|$ $x$ $|$ $b$ $|$ $x$ $|$ $b$.

(vi) There is a $|$ $x$ $|$ $b$ $|$ $x$ $|$ $b$.

(vii) The $|$ $x$ $|$ $b$ $|$ $x$ $|$ $b$ is infinite or not extendible.
The correctness of deterministic programs can be formalized into a proof system known as PD, which is both correct and complete for partial correctness of deterministic programs. The correctness formula $\sigma$ is expressed as $\sigma = _{1 \leq i \leq n} x_i \delta x_i + (\sum_{i=1}^{n} x_i)^2 = x_1 \delta x_1 + x_2 \delta x_2 + \cdots + x_n \delta x_n + (\sum_{i=1}^{n} x_i)^2$.

Rule 6: While-Loop

Rule 5: Assignment

Rule 4: Skip-Statement

Example: Computing squares (of numbers...
In the following, we show derivability in PD: by showing example proof.
Expressions variables need to be treated specially:

- **\( \phi \)**: replace all (free) occurrences of (program or logical) variable \( p \), replace all (free) occurrences of (program or logical) variable \( q \).

- **\( \neg \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \equiv \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \land \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \lor \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \forall \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \exists \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \cdot \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{N} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{Z} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{Q} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{R} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{C} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{H} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{I} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{O} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{F} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{G} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{J} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{K} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{L} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{M} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{N} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{O} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{P} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{Q} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{R} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{S} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{T} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{U} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{V} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{W} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{X} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{Y} \)**: replace all (free) occurrences of (program or logical) variable \( p \).

- **\( \mathbb{Z} \)**: replace all (free) occurrences of (program or logical) variable \( p \).
Proof of (2)

Example Proof Cont'd

The rule '...

Substitution

Expressions

The rule '...

Substitution

Proof

Example Proof Cont'd

The rule '...

Substitution

Expressions

The rule '...

Substitution

Proof

Example Proof Cont'd

The rule '...

Substitution

Expressions
Example Proof Cont'd

Proof of (2)

\[
\begin{align*}
\text{Example Proof Cont'd} & \\
\text{Proof of (2)} & \\
\text{Example Proof Cont'd} & \\
\end{align*}
\]
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

Modular reasoning

Return values/old values

Assertions

The Verifier for Concurrent C (VCC) basically implements Hoare-style reasoning.

Special syntax:

#include <vcc.h>

_(requires p)—pre-condition, p is (basically) a C expression

_(ensures q)—post-condition, q is (basically) a C expression

_(invariant expr)—loop invariant, expr is (basically) a C expression

_(assert p)—intermediate invariant, p is (basically) 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 Syntax Example

```c
#include <vcc.h>

int a, b;

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

DIV

≡

DIV

The VCC Web-Interface

Example program DIV:

http://rise4fun.com/Vcc/4Kqe

Interpretation of Results

• VCC result: "verification succeeded"

• VCC result: "verification failed"

• Other case: "timeout" etc.
Interpretation of RESULTS

As we've seen, proving properties can be inconclusive. We may need to try again.

Other case: timeout

→ Other case: timeout

The tool may be a false positive.

May be due to an error in the tool!

→ May be due to an error in the tool!

→ May be due to an error in the tool!

We can only conclude that the tool claims that there is a proof for...

Yet we can ask...

and check it manually... for a printout of the proof...

Note: Church's theorem does not provide counter-examples in the form of a computation path...

That is, a mistake in writing down the pre-condition can make errors in the program go undetected!

And causing a violation of... satisfies hints on input values...

→ And causing a violation of... satisfies hints on input values...

→ And causing a violation of... satisfies hints on input values...

→ And causing a violation of... satisfies hints on input values...

Interpretation of Results

That's a false negative. (That's a false negative.)

That's a false negative. (That's a false negative.)

That's a false negative. (That's a false negative.)

As we've seen, proving properties can be inconclusive. We may need to try again.

Other case: timeout

→ Other case: timeout

The tool may be a false positive.

May be due to an error in the tool!

→ May be due to an error in the tool!

→ May be due to an error in the tool!

We can only conclude that the tool claims that there is a proof for...

Yet we can ask...

and check it manually... for a printout of the proof...

Note: Church's theorem does not provide counter-examples in the form of a computation path...

That is, a mistake in writing down the pre-condition can make errors in the program go undetected!

And causing a violation of... satisfies hints on input values...

→ And causing a violation of... satisfies hints on input values...

→ And causing a violation of... satisfies hints on input values...

→ And causing a violation of... satisfies hints on input values...

Interpretation of Results

That's a false negative. (That's a false negative.)

That's a false negative. (That's a false negative.)

That's a false negative. (That's a false negative.)

As we've seen, proving properties can be inconclusive. We may need to try again.

Other case: timeout

→ Other case: timeout

The tool may be a false positive.

May be due to an error in the tool!

→ May be due to an error in the tool!

→ May be due to an error in the tool!

We can only conclude that the tool claims that there is a proof for...

Yet we can ask...

and check it manually... for a printout of the proof...

Note: Church's theorem does not provide counter-examples in the form of a computation path...

That is, a mistake in writing down the pre-condition can make errors in the program go undetected!

And causing a violation of... satisfies hints on input values...

→ And causing a violation of... satisfies hints on input values...

→ And causing a violation of... satisfies hints on input values...

→ And causing a violation of... satisfies hints on input values...

Interpretation of Results

That's a false negative. (That's a false negative.)

That's a false negative. (That's a false negative.)

That's a false negative. (That's a false negative.)
The sum is just an abbreviation for \( \sum \). To compute it, we use the `read_number` function to read an integer and then compute the sum using a while loop:

\[
\text{result} = \sum \text{read_number}\; y + (\text{result} + y) \leq \text{result} \lor \text{result} < \text{read_number}
\]

According to the C-standard, all variables must be declared before they are used, and all expressions are evaluated according to specific rules. VCC checks a number of rules, including:

- Shared variables are properly managed;
- Null-pointer dereferences are properly handled;
- No arithmetic overflows are allowed;
- No integer overflows are allowed;
- All expressions are handled according to the C-standard;
- For the exercises, we use VCC only for sequential, single-thread programs.

We can add another rule for calls of functions:

\[
\text{result} = \text{sum} (x, y) + \text{result} \leq \text{result} \lor \text{result} < \text{sum}(x, y)
\]

The post-condition \( R_7 \) of the function `sum`:

\[
\text{result} = \text{sum}_1 (x, y) + \text{result} \leq \text{result} \lor \text{result} < \text{sum}_1 (x, y)
\]
Assert:

- Extend the syntax of deterministic programs by \( S := \cdot \cdot \cdot | assert(B) \)

- and the semantics by rule \( \langle assert(B), \sigma \rangle \rightarrow \langle E, \sigma \rangle \) if \( \sigma |\|= B \).

  (If the asserted boolean expression \( B \) does not hold in state \( \sigma \), the empty program is not reached; otherwise the assertion remains in the first component: abnormal program termination).

  Extend PD by axiom:

  \[
  \{ p \} assert(p) \{ p \}
  \]

  • That is, if \( p \) holds before the assertion, then we can continue with the derivation in PD. If \( p \) does not hold, we "get stuck" (and cannot complete the derivation).

  So we cannot derive \( \{ true \} x := 0; assert(x = 27) \{ true \} \) in PD.

**Formal Verification:**

- Program verification is another approach to software quality assurance.
- Proof System PD can be used to prove that a given program is correct wrt. its specification. This approach considers all inputs inside the specification!
- Tools like VCC implement this approach.

**References**

