Softwaretechnik / Software-Engineering

Lecture 18: Wrapup & Questions

2018-07-16

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
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,
- Runtime verification.

Program Verification
- Partial and total correctness,
- Proof System PD.

Review
Content

- The Verifier for Concurrent C
  - Assertions, Modular Verification, VCC

- Runtime-Verification
  - Assertions, LSC-Observers

- Reviews
  - Roles and artefacts
  - Review procedure
  - Stronger and weaker variants

- Code QA Techniques Revisited
  - Test, Runtime-Verification, Review,
  - Static Checking, Formal Verification

- Dependability
The Verifier for Concurrent C
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

```
#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;
            }
    }
```

\[ DIV \equiv a := 0; b := x; \textbf{while } b \geq y \textbf{ do } b := b - y; a := a + 1 \textbf{ od } \]

\[ \{ x \geq 0 \land y \geq 0 \} DIV \{ x \geq 0 \land y \geq 0 \} \]
Example program \textit{DIV}: http://rise4fun.com/Vcc/4Kqe
Interpretation of Results

- VCC result: “verification succeeded”
  - We can only conclude that the tool
    – under its interpretation of the C-standard, under its platform assumptions (32-bit), etc. –
    claims that there is a proof for $\models \{p\} \text{DIV} \{q\}$.
  - May 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: $\models \{\text{false}\} \text{f} \{q\}$ always holds.
    That is, a mistake in writing down the pre-condition can make errors in the program go undetected!

- VCC result: “verification failed”
  - May be a false positive (wrt. the goal of finding errors).
    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 $q$.
  - → try to construct a (true) counter-example from the hints.
    or: make loop-invariant(s) (or pre-condition $p$) 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.

- 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.

- 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.
Assertions
Assertions

- Extend the **syntax** of deterministic programs by
  
  \[ S ::= \cdots | \text{assert}(B) \]
  
- and the **semantics** by rule
  
  \[ \langle \text{assert}(B), \sigma \rangle \rightarrow \langle E, \sigma \rangle \text{ if } \sigma \models B. \]

  (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:

\[(A7) \{p\} \text{assert}(p) \{p\}\]

- 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 \(\{true\} \ x := 0; \ \text{assert}(x = 27) \{true\}\) in PD.
Modular Reasoning
We can add another rule for calls of functions $f : F$ (simplest case: only global variables):

$$(R7) \quad \{p\} F \{q\} \quad \{p\} f() \{q\}$$

“If we have $\vdash \{p\} F \{q\}$ for the implementation of function $f$,
then if $f$ is called in a state satisfying $p$, the state after return of $f$ will satisfy $q$.”

$p$ is called pre-condition and $q$ is called post-condition of $f$.

**Example:** if we have

- $\{true\} \text{read}_\text{number} \{0 \leq \text{result} < 10^8\}$
- $\{0 \leq x \wedge 0 \leq y\} \text{add} \{(\text{old}(x) + \text{old}(y) < 10^8 \wedge \text{result} = \text{old}(x) + \text{old}(y)) \lor \text{result} < 0\}$
- $\{true\} \text{display} \{(0 \leq \text{old}(\text{sum}) < 10^8 \implies "\text{old}(\text{sum})") \wedge (\text{old}(\text{sum}) < 0 \implies "-E-")\}$

we may be able to prove our pocket calculator correct.
Return Values and Old Values

- For **modular reasoning**, it’s often useful to refer in the post-condition to
  - the **return value** as \( \text{result} \),
  - the **values** of variable \( x \) at calling time as \( \text{old}(x) \).

- Can be defined using **auxiliary variables**:
  - Transform function
    \[
    T \ f() \ \{ \ldots ; \text{return expr}; \}
    \]
    (over variables \( V = \{v_1, \ldots, v_n\}; \text{where result, } v_i^{old} \notin V \}) into
    \[
    \begin{align*}
    T \ f() \ {\}
    & \triangleright v_1^{old} := v_1; \ldots; v_n^{old} := v_n; \\
    & \ldots; \\
    & \text{result} := \text{expr} ; \\
    & \text{return result} ; \\
    \end{align*}
    \]
    over \( V' = V \cup \{v^{old} \mid v \in V\} \cup \{\text{result}\} \).
  - Then \( \text{old}(x) \) is just an abbreviation for \( x^{old} \).
Content

- The Verifier for Concurrent C
  - Assertions, Modular Verification, VCC

- Runtime-Verificati
cn
  - Assertions, LSC-Observers

- Reviews
  - Roles and artefacts
  - Review procedure
  - Stronger and weaker variants

- Code QA Techniques Revisited
  - Test, Runtime-Verification, Review,
  - Static Checking, Formal Verification

- Dependability
Run-Time Verification
Run-Time Verification: Idea

- Assume, there is a function $f$ in software $S$ with the following specification:
  
  - **pre-condition**: $p$, **post-condition**: $q$.

- Computation paths of $S$ may look like this:

  $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \xrightarrow{\alpha_{n-1}} \sigma_n \xrightarrow{\text{call } f} \sigma_{n+1} \cdots \sigma_m \xrightarrow{\text{f returns}} \sigma_{m+1} \cdots$

- Assume there are functions $\text{check}_p$ and $\text{check}_q$, which check whether $p$ and $q$ hold at the current program state, and which do not modify the program state (except for program counter).

- **Idea**: create software $S'$ by

  (i) extending $S$ by implementations of $\text{check}_p$ and $\text{check}_q$,

  (ii) call $\text{check}_p$ right after entering $f$,

  (iii) call $\text{check}_q$ right before returning from $f$.

- For $S'$, obtain computation paths like:

  $\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \xrightarrow{\alpha_{n-1}} \sigma_n \xrightarrow{\text{call } f} \sigma_{n+1} \cdots \sigma_m \xrightarrow{\text{check}_p} \sigma'_{n+1} \cdots \sigma_m \xrightarrow{\text{check}_q} \sigma'_m \xrightarrow{\text{f returns}} \sigma_{m+1} \cdots$

- If $\text{check}_p$ and $\text{check}_q$ notify us of violations of $p$ or $q$, then we are notified of $f$ violating its specification when running $S'$ (at run-time).
Run-Time Verification: Example

```c
int x, y, sum;

int main() {
    while (true) {
        x = read_number();
        y = read_number();
        sum = add(); // add 'x' and 'y'
        display(); // display 'sum'
    }
}

void verify_sum(int x, int y, int sum) {
    if (sum != (x+y) || (x + y > 99999999 && !(sum < 0)))
        printf(stderr, "verify_sum: error\n");
        abort();
}
```
Run-Time Verification: Example

```
int x, y, sum;

int main() {
    while (true) {
        x = read_number();
        y = read_number();
        sum = add(x, y);
        // verify_sum(x, y, sum);
        display();
    }
}

void verify_sum(int x, int y, int sum) {
    if (sum != (x+y) || (x + y > 99999999 && !(sum < 0))) {
        printf(stderr, "verify_sum: error\n");
        abort();
    }
}
```
A Very Useful Special Case: Assertions

- Maybe the simplest instance of runtime verification: Assertions.
- Available in standard libraries of many programming languages (C, C++, Java, ...).

- For example, the C standard library manual reads:

```
NAME
   assert — abort the program if assertion is false

SYNOPSIS
   #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.
```

- In C code, assert can be disabled in production code (-DNDEBUG).
- Use java -ea ... to enable assertion checking (disabled by default).

(cf. https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html)
The abstract $f$-example from **run-time verification**:
(specification: $\{p\} f \{q\}$)

```
void f(...) {
  assert(p);
  ...  
  assert(q);
}
```

- Compute the width of a progress bar:

```
int progress_bar_width(int progress, int window_left, int window_right)
{
  assert(window_left <= window_right);  /* pre-condition */
  ...  
  assert(0 < progress && progress < 100);  // extremal cases already treated
  ...  
  return r;
}
```
Recall the **structure model** with Proto-OCL constraint from Exercise Sheet 4/2017.

Assume, we add a method `set_key()` to class **TreeNode**:

```java
class TreeNode {
  private int key;
  TreeNode parent, leftChild, rightChild;

  public int get_key() { return key; }
  public void set_key(int new_key) {
    key = new_key;
  }
}
```

We can check consistency with the Proto-OCL constraint at runtime by using assertions:

```java
public void set_key(int new_key) {
  assert (parent == null || parent.get_key() <= new_key);
  assert (leftChild == null || new_key <= leftChild.get_key());
  assert (rightChild == null || new_key <= rightChild.get_key());
  key = new_key;
}
```
More Complex Run-Time Verification: LSC Observers

ChoicePanel:

\[
\text{st} : \{ \text{idle, wsel, ssel, tsel, reqs, half} \}
\]

\[
\text{take_event}(E : \{ \text{TAL, WATER, SOFT, TEA, ... } \}) \{
\text{bool stable} = 1;
\text{switch} (\text{st}) \{
\text{case idle :}
\text{switch} (E) \{
\text{case WATER :}
\text{if} (\text{water_enabled}) \{ \text{st} := \text{wsel}; \text{stable} := 0; \}
\text{;;}
\text{case SOFT :}
\text{...}
\text{case wsel :}
\text{switch} (E) \{
\text{case TAL :}
\text{send_DWATER(); st} := \text{reqs};
\text{hey_observer_I_just_sent_DWATER();}
\text{;;}
\text{)}}
\text{)}}
\text{}}
\text{}}}
• **Experience:**
  During development, *assertions* for pre/post conditions and intermediate invariants are an extremely powerful tool with a very attractive gain/effort ratio (low effort, high gain).

• Assertions effectively work as **safe-guard against unexpected use** of functions and **regression**, e.g. during later maintenance or efficiency improvement.

• Can serve as **formal** (support of) **documentation**:
  “Dear reader, at this point in the program, I expect condition *expr* to hold, because…”.
By the Way: Development vs. Release Versions

- **Development- vs. Release Versions:**
  - Common practice:
    - development version *with* run-time verification enabled (cf. `assert(3)`),
    - release version *without* run-time verification.

If run-time verification is enabled in a release version,
- software should **terminate as gracefully as possible** (e.g. try to save data),
- save information from assertion failure if possible for future analysis.

**Risk:** with bad luck, the software only behaves well *because of* the run-time verification code. Yet disabling run-time verification “breaks” the software. Yet very complex run-time verification may significantly slow down the software, so needs to be disabled...
Content

- The Verifier for Concurrent C
  - Assertions, Modular Verification, VCC

- Runtime-Verification
  - Assertions, LSC-Observers

- Reviews
  - Roles and artefacts
  - Review procedure
  - Stronger and weaker variants

- Code QA Techniques Revisited
  - Test, Runtime-Verification, Review,
  - Static Checking, Formal Verification

- Dependability
Review
Recall: Three Basic Directions

all computation paths satisfying the specification

expected outcomes \( S_{\text{oll}} \)

\[(\Sigma \times A)^{\omega}\]

defines

\(\in\) ?

\(\subseteq\) ?

\(\subseteq\) ?

execution of \((\text{In}, S_{\text{oll}})\)

execution of \((\text{In}, S_{\text{oll}})\)

Reviewer

input → output

Review

Testing

prove \( S \models \mathcal{I} \), conclude \([S] \in \mathcal{I}\)

Formal Verification
Input to Review Session:

- **Review item**: can be every closed, human-readable part of software (documentation, module, test data, installation manual, etc.).

  **Social aspect**: it is an **artefact** which is examined, not the **human** (who created it).

- **Reference documents**: need to enable an assessment (requirements specification, guidelines (e.g. coding conventions), catalogue of questions (“all variables initialised?”), etc.)

Roles:

- **Moderator**: leads session, responsible for properly conducted procedure.

- **Author**: (representative of the) creator(s) of the artefact under review; is present to listen to the discussions; can answer questions; does not speak up if not asked.

- **Reviewer(s)**: person who is able to judge the artefact under review; maybe different reviewers for different aspects (programming, tool usage, etc.), at best experienced in detecting inconsistencies or incompleteness.

- **Transcript Writer**: keeps minutes of review session, can be assumed by author.

The **review team** consists of everybody but the author(s).
Review Procedure Over Time

**planning:** reviews need **time** in the project plan.

a review is **triggered**, e.g., by a submission to the revision control system:

the moderator **invites** (include review item in invitation), and states **review missions**.

**preparation:** reviewers **investigate** review item.

**review session:** reviewers **report**, evaluate, and document issues; **resolve** open questions.

**postparation:** **rework** review item; responsibility of the author(s).

**analysis:** improve development and review process.

- Reviewers **re-assess** reworked review item (until **approval** is declared).
Review Rules (Ludewig and Lichter, 2013)

(i) The moderator organises the review, issues invitations, supervises the review session.

(ii) The moderator may terminate the review if conduction is not possible, e.g., due to inputs, preparation, or people missing.

(iii) The review session is limited to 2 hours. If needed: organise more sessions.

(iv) The review item is under review, not the author(s). Reviewers choose their words accordingly. Authors neither defend themselves nor the review item.

(v) Roles are not mixed up, e.g., the moderator does not act as reviewer. (Exception: author may write transcript.)

(vi) Style issues (outside fixed conventions) are not discussed.

(vii) The review team is not supposed to develop solutions. Issues are not noted down in form of tasks for the author(s).

(viii) Each reviewer gets the opportunity to present her/his findings appropriately.

(ix) Reviewers need to reach consensus on issues, consensus is noted down.

(x) Issues are classified as:
   • critical (review unusable for purpose),
   • major (usability severely affected),
   • minor (usability hardly affected),
   • good (no problem).

(xi) The review team declares:
   • accept without changes,
   • accept with changes,
   • do not accept.

(xii) The protocol is signed by all participants.
**Stronger and Weaker Review Variants**

- **Design and Code Inspection** *(Fagan, 1976, 1986)*
  - deluxe variant of review,
  - approx. 50% more time, approx. 50% more errors found.

- **Review**

  - **Structured Walkthrough**
    - simple variant of review:
      - developer moderates walkthrough-session,
      - developer presents artefact(s),
      - reviewer poses (prepared or spontaneous) questions,
      - issues are noted down,
    - Variation point: do reviewers see the artefact before the session?
    - less effort, less effective.
    
    → disadvantages: unclear responsibilities; “salesman”-developer may trick reviewers.

- **Comment** (‘Stellungnahme’)
  - colleague(s) of developer read artefacts,
  - developer considers feedback.
  
  → advantage: low organisational effort;
  → disadvantages: choice of colleagues may be biased; no protocol; consideration of comments at discretion of developer.

- **Careful Reading** (‘Durchsicht’)
  - done by developer,
  - recommendation: “away from screen” (use print-out or different device and situation)
Content

- The Verifier for Concurrent C
  - Assertions, Modular Verification, VCC

- Runtime-Verification
  - Assertions, LSC-Observers

- Reviews
  - Roles and artefacts
  - Review procedure
  - Stronger and weaker variants

- Code QA Techniques Revisited
  - Test, Runtime-Verification, Review,
  - Static Checking, Formal Verification

- Dependability
Code Quality Assurance Techniques Revisited
## Strengths:
- can be fully automatic (yet not easy for GUI programs);
- negative test proves "program not completely broken", "can run" (or positive scenarios);
- final product is examined, thus toolchain and platform considered;
- one can stop at any time and take partial results;
- few, simple test cases are usually easy to obtain;
- provides reproducible counter-examples (good starting point for repair).

## Weaknesses:
- (in most cases) vastly incomplete, thus no proofs of correctness;
- creating test cases for complex functions (or complex conditions) can be difficult;
- maintenance of many, complex test cases be challenging.
- executing many tests may need substantial time (but: can sometimes be run in parallel);
Techniques Revisited

<table>
<thead>
<tr>
<th></th>
<th>automatic</th>
<th>prove “can run”</th>
<th>toolchain considered</th>
<th>exhaustive</th>
<th>prove correct</th>
<th>partial results</th>
<th>entry cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Runtime-Verification</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Review</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Strengths:**
- **fully automatic** (once observers are in place);
- **provides counter-example**;
- (nearly) **final product is examined**, thus toolchain and platform considered;
- one can stop at any time and take **partial results**;
- **assert-statements have a very good effort/effect ratio**.

**Weaknesses:**
- counter-examples **not necessarily reproducible**;
- may negatively affect **performance**;
- code is changed, program may only run **because of** the observers;
- completeness depends on usage, may also be **vastly incomplete**, so no correctness proofs;
- constructing observers for complex properties may be **difficult**, one needs to learn how to construct observers.
# Techniques Revisited

<table>
<thead>
<tr>
<th></th>
<th>automatic</th>
<th>prove “can run”</th>
<th>toolchain considered</th>
<th>exhaustive</th>
<th>prove correct</th>
<th>partial results</th>
<th>entry cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>Runtime-Verification</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>Review</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Static Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Strengths:**

- human readers can understand the code, may spot point errors;
- reported to be highly effective;
- one can stop at any time and take partial results;
- intermediate entry costs;
  - good effort/effect ratio achievable.

**Weaknesses:**

- no tool support;
- no results on actual execution, toolchain not reviewed;
- human readers may overlook errors; usually not aiming at proofs.
- does (in general) not provide counter-examples, developers may deny existence of error.
## Techniques Revisited

<table>
<thead>
<tr>
<th></th>
<th>automatic</th>
<th>prove “can run”</th>
<th>toolchain considered</th>
<th>exhaustive</th>
<th>prove correct</th>
<th>partial results</th>
<th>entry cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Runtime-Verification</strong></td>
<td>✔</td>
<td>(✔)</td>
<td>✔</td>
<td>(✘)</td>
<td>✘</td>
<td>✔</td>
<td>(✔)</td>
</tr>
<tr>
<td><strong>Review</strong></td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>(✔)</td>
<td>(✔)</td>
<td>✔</td>
<td>(✔)</td>
</tr>
<tr>
<td><strong>Static Checking</strong></td>
<td>✔</td>
<td>(✘)</td>
<td>✘</td>
<td>✔</td>
<td>(✔)</td>
<td>✔</td>
<td>(✘)</td>
</tr>
<tr>
<td><strong>Verification</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Strengths:
- there are (commercial), **fully automatic** tools (lint, Coverity, Polyspace, etc.);
- some tools are **complete** (relative to assumptions on language semantics, platform, etc.);
- can be **faster than testing**;
- one can stop at any time and take **partial results**.

### Weaknesses:
- no results on actual execution, **toolchain not reviewed**;
- can be very **resource consuming** (if few false positives wanted), e.g., code may need to be “designed for static analysis”;
- many false positives can be very **annoying to developers** (if fast checks wanted);
- distinguish **false from true positives** can be challenging;
- **configuring the tools** (to limit false positives) can be challenging.
## Techniques Revisited

<table>
<thead>
<tr>
<th></th>
<th>automatic</th>
<th>prove “can run”</th>
<th>toolchain considered</th>
<th>exhaustive</th>
<th>prove correct</th>
<th>partial results</th>
<th>entry cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>☒</td>
<td>☒</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Runtime-Verification</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>☒</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Review</td>
<td>☒</td>
<td>☒</td>
<td>☒</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Static Checking</td>
<td>✔</td>
<td>☒</td>
<td>☒</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>☒</td>
</tr>
<tr>
<td>Verification</td>
<td>✔</td>
<td>☒</td>
<td>☒</td>
<td>✔</td>
<td>✔</td>
<td>☒</td>
<td>☒</td>
</tr>
</tbody>
</table>

### Strengths:
- some **tool support** available (few commercial tools);
- **complete** (relative to assumptions on language semantics, platform, etc.);
- thus can provide **correctness proofs**;
- can prove correctness for **multiple language semantics and platforms** at a time;
- can be **more efficient than other techniques**.

### Weaknesses:
- no results on actual execution, **toolchain not reviewed**;
- not many **intermediate results**: “half of a proof” may not allow any useful conclusions;
- **entry cost high**: significant training is useful to know how to deal with tool limitations;
- proving things is challenging: failing to find a proof does not allow any useful conclusion;
- **false negatives** (broken program “proved” correct) hard to detect.
## Techniques Revisited

<table>
<thead>
<tr>
<th></th>
<th>automatic</th>
<th>prove “can run”</th>
<th>toolchain considered</th>
<th>exhaustive</th>
<th>prove correct</th>
<th>partial results</th>
<th>entry cost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test</strong></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✘</td>
<td>✘</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>Runtime-</strong></td>
<td>✔</td>
<td>(✔)</td>
<td>✔</td>
<td>(✘)</td>
<td>✘</td>
<td>✔</td>
<td>(✔)</td>
</tr>
<tr>
<td><strong>Verification</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Review</strong></td>
<td>✘</td>
<td>✘</td>
<td>✘</td>
<td>(✔)</td>
<td>(✔)</td>
<td>✘</td>
<td>(✔)</td>
</tr>
<tr>
<td><strong>Static Checking</strong></td>
<td>✔</td>
<td>(✘)</td>
<td>✘</td>
<td>✔</td>
<td>(✔)</td>
<td>✘</td>
<td>(✘)</td>
</tr>
<tr>
<td><strong>Verification</strong></td>
<td>(✔)</td>
<td>✘</td>
<td>✘</td>
<td>❌</td>
<td>✘</td>
<td>(✘)</td>
<td>✘</td>
</tr>
</tbody>
</table>
Dependability Case
Proposal: Dependability Cases (Jackson, 2009)

- A **dependable** system is one you can **depend** on – that is, you can place your trust in it.

  “Developers [should] express the critical properties and **make an explicit argument** that the system satisfies them.”

**Proposed Approach:**

- Identify the **critical requirements**, and determine what **level of confidence** is needed.
  (Most systems do also have **non-critical** requirements.)

- Construct a **dependability case**, i.e. an **argument**, that the software, in concert with other components, establishes the **critical properties**.

- The **dependability case** should be
  - **auditable**: can (easily) be evaluated by third-party certifier.
  - **complete**: no holes in the argument; any assumptions that are not justified should be noted (e.g. assumptions on compiler, on protocol obeyed by users, etc.)
  - **sound**: e.g. should not claim full correctness [...] based on nonexhaustive testing; should not make unwarranted assumptions on independence of component failures; etc.
Tell Them What You’ve Told Them...

- **Runtime Verification**
  - (as the name suggests) checks properties at program run-time,
  - generous use of `assert`'s can be a valuable safe-guard against
    - `regressions`, usage `outside specification`, etc.
  - and serve as formal documentation of (intermediate) assumptions.
  - Very attractive effort / effect ratio.

- **Review** (structured examination of artefacts by humans)
  - (mild variant) advocated in the XP approach,
  - **not uncommon:** lead programmer reviews all commits from team members,
  - literature reports good effort/effect ratio achievable.

- All approaches to code quality assurance have their
  - advantages and drawbacks.
  - Which to use? It depends!

- Overall: Consider Dependability Cases
  - an (auditable, complete, sound) argument, that a software has the critical properties.
Looking Back:
18 Lectures on Software Engineering
Contents of the Course

- Introduction L 1: 16.4., Mon
- Scales, Metrics, Costs, L 2: 19.4., Thu
- L 3: 23.4., Mon
- T 1: 26.4., Thu
- Development Process L 4: 30.4., Mon
- L 5: 3.5., Thu
- L 6: 7.5., Mon
- T 1: 10.5., Thu
- Requirements L 7: 14.5., Mon
- T 2: 17.5., Thu
- L 8: 28.5., Mon
- T 3: 7.6., Thu
- Engineering L 9: 4.6., Mon
- T 4: 21.6., Thu
- Arch. & Design, Software- L 10: 11.6., Mon
- L 11: 14.6., Thu
- L 12: 18.6., Mon
- T 4: 21.6., Thu
- Modelling, Patterns L 13: 25.6., Mon
- L 14: 28.6., Thu
- QA L 15: 2.7., Mon
- T 5: 5.7., Thu
- (Testing, Formal Verification) L 16: 9.7., Mon
- L 17: 12.7., Thu
- Wrap-Up L 18: 16.7., Mon
- T 6: 19.7., Thu
**Topic Area: Project Management**

- **measure**, know what you measure (scales, pseudo-metrics)
- estimate, measure, improve estimation – it’s about experience
- describe processes in terms of artefact, activity, role, etc. – and risk

**Topic Area: Requirements Engineering**

- is about humans (“was mit Menschen machen”)
- there are methods and tools to get indications of issues (know how to interpret the indications)
- precise requirements have a precise notion of being satisfied

**Topic Area: Architecture & Design**

- “Nobody builds a house without a plan.” (L. Lamport)
- software has structural and behavioural aspects
- there are methods and tools to get indications of issues (know how to interpret the indications)

**Topic Area: Software Quality Assurance**

- testing is almost always incomplete; testing is necessary (know how to interpret the outcomes: true/false positive/negative)
- there are methods and tools to prove code correct (correctness is relative: correct wrt. specification (and assumptions))
That’s Today’s Software Engineering — More or Less...
Questions?
Advertisements
Further studies:

- **Real-Time Systems** (not in 2018/19)  
  (specification and verification of real-time systems)

- **Software Design, Modelling, and Analysis in UML** (not in 2018/19)  
  (a formal, in-depth view on structural and behavioural modelling)

- **Cyber-Physical Systems I - Discrete Models**  
  (more on variants of CFA and queries (LTL, CTL, CTL*)

- **Cyber-Physical Systems - Hybrid Models**  
  (Modelling and analysis of cyber-physical systems with hybrid automata)

- **Program Verification**  
  (the theory behind tools like VCC)

- **Formal Methods for Java**  
  (JML and “VCC for Java”)

- **Decision Procedures** → **Program Verification, Tue, 2018-07-17, 16:00**  
  (the basis for program verification)

→ [https://swt.informatik.uni-freiburg.de/teaching](https://swt.informatik.uni-freiburg.de/teaching)
• **Individual Projects**
  (BSc/MSc project, Lab Project, BSc/MSc thesis)
  • **formal modelling** of industrial case studies
  • **improving analysis techniques**
  • **own topics**
  → **contact us** (3–6 months before planned start).

• Want to be a **tutor**, e.g. Software Engineering 2019,
  → **contact us** (around September / March).

• Want to be a **scientific student assistant**?
  → **contact us**.
Thanks For Your Participation...
References
References


