Softwaretechnik / Software-Engineering

Lecture 18: Wrapup & Questions

2018-07-16

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal
Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Code Quality Assurance: Content

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
  • Model checking

Program Verification

• partial and total correctness
  • Proof System PD

Review

• 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

• Do's and Don'ts in Code QA

Dependability

The Verifier for Concurrent C

VCC

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 ≡ a := 0; b := x; while b ≥ y do b := b − y; a := a + 1 od {x ≥ 0 ∧ y ≥ 0}
Example program

\begin{verbatim}
http://rise4fun.com/Vcc/4Kqe
\end{verbatim}

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 $| = \{ p \}$ \texttt{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: $| = \{ \text{false} \}$ \texttt{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 \texttt{p} and causing a violation of \texttt{q}.
  • → try to construct a (true) counter-example from the hints.
  • 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 \texttt{l} is always equal to the length of the string field \texttt{str}); those invariants may temporarily be violated when updating the data structure.
  • and much more.

Assertions

• Extend the syntax of deterministic programs by \texttt{S} ::= \ldots | assert (B)
• and the semantics by rule $⟨ \text{assert} (B), σ⟩ \rightarrow ⟨ E, σ⟩$ if $σ | = B$.
  (If the asserted boolean expression \texttt{B} does not hold in state \texttt{σ}, the empty program is not reached; otherwise the assertion remains in the first component: abnormal program termination).
• Extend PD by axiom:
  \begin{align*}
  \{ p \} \text{assert} (p) \{ p \}
  \end{align*}
  • That is, if \texttt{p} holds before the assertion, then we can continue with the derivation in PD.
  • If \texttt{p} does not hold, we "get stuck" (and cannot complete the derivation).
  • So we cannot derive \{ true \} \texttt{x} := 0; \texttt{assert}(\texttt{x} = 27) \{ true \}
    in PD.

Modular Reasoning
When running `fpcheck`, obtain computation paths like:

```c
int main() {
    int x, y, sum;
    sum = add();
    y = read_number();
    while (x + y > 99999999) {
        if (! (sum < 0)) {
            x = read_number();
        }
        y = read_number();
    }
    return sum;
}
```

Assume there are functions `n`, `m`, and `σ` with the following specification:

```c
{ v := x } n \leadsto \{ v \} m \sigma
```
With Proto-OCL constraint at runtime by using assertions:

```c
assert(rightChild == 4);
assert(leftChild == 3);
```

Example from Exercise Sheet 4.

More Complex Run-Time Verification: LSC Observers

In C code, the C standard library manual reads:

```c
#define NDEBUG
```

Disabled by default.

Available in standard libraries of many programming languages (C, C++, Java, . . . ).

For example, the C standard library manual reads:

```c
assert(0 < progress && progress < 100);
```

/* treats special cases 0 and 100*/

```c
if (true) {
    while (true) {
        display();
    }
}
```

In the code example, an assertion would be:

```c
assert(myVariable > 0);
```

The message "assertion failed in file foo.c, function bar()" verifies:

```c
void doSomething() {
    int myVariable;
    // code...
    assert(myVariable > 0);
    // code...
}
```

Run-Time Verification: Example

```c
void doSomething() {
    int myVariable;
    // code...
    assert(myVariable > 0);
    // code...
}
```

With the Proto-OCL constraint from Exercise Sheet 4.

### Assertions At Work

- In the C code example, an assertion is used to verify that the variable `myVariable` is greater than 0.
- The message "assertion failed in file foo.c, function bar()" indicates where and what the assertion is related to.
- This allows developers to quickly identify issues in their code at runtime.

### More Complex Run-Time Verification: LSC Observers

- LSC observers provide a way to verify complex runtime properties that are not easily expressible with simple assertions.
- For example, they can be used to ensure that certain conditions hold throughout the execution of a program.
- This is particularly useful in systems where runtime properties are critical for safety or correctness.

### Assertions In The Wild

- Assertions can be used in various scenarios to ensure the correctness of code at runtime.
- For instance, they can be employed in testing frameworks to validate the expected behavior of a system.
- In addition, they can be used to enforce invariants in critical sections of code where the state of the system needs to be preserved.

### More Complex Run-Time Verification: LSC Observers

- LSC observers extend the capabilities of simple assertions by allowing the expression of more complex and dynamic properties.
- They can be used to monitor the system's state over time and detect deviations from the expected behavior.
- This is especially valuable in real-time systems where timing constraints are crucial.

### Assertions At Work

- In the context of real-time systems, assertions can be crucial for ensuring that deadlines are met and that the system operates as intended.
- They help in early detection of runtime issues, allowing for timely corrective actions.
- For example, in a证券交易系统 (stock trading system), assertions can be used to ensure that transactions are processed within the specified time limits.
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 very complex run-time verification may significantly slow down the software, so needs to be disabled.

---

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
- Do's and Don'ts in Code QA
- Dependability

---

Reviews
- 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.
- Analysis: improve development and review process.
- Preparation: reviewers investigate review item.
- Review Session: reviewers report, evaluate, and document issues; resolve open questions.
- "3rd hour": time for informal chat, reviewers may state proposals for solutions or improvements.
- Postparation: rework review item; responsibility of the author(s).
- Analysis: reviewers re-assess reworked review item (until approval is declared).
one needs to learn how to construct observers. Constructing observers for complex properties may be challenging, so no correctness proofs; vast completeness depends on usage, may also be not necessarily reproducible counter-examples: weak strengths: easy to obtain partial results one can stop at any time and take (nearly) automatic (yet not easy for GUI programs); fully automatic proves “program not completely broken” can be difficult creating test cases for complex functions (or complex conditions); performance may negatively affect, thus no proofs of correctness; (in most cases) can sometimes be run in parallel; substantial time executing many tests may need; 

Weaknesses

reproducible counter-examples provides

Careful Reading

→: low organisational effort; considers feedback.

Do’s and Don’ts

• developers present her/his findings appropriately.

Formal Verification, Static Checking, Review

• assertions, modular verification, VCC

The Verifier for Concurrent C 1986, Fagan

Content

Review Rules

Strengths

Variation point

Notice: DELI

• the review item. Issues are noted down, moderator declares: “the moderator declares:” no problem.

ix) The issue is under review, moderator presents her/his findings appropriately. If needed: organise more sessions.

x) The review is not possible, e.g., due to inputs, preparation, or people missing.

xi) The issue is discussed, moderator declares: “the moderator declares” (usability hardly affected), (usability severely affected), major issues are noted down, (review unusable for purpose), not the author(s).

xii) The issue is not discussed, e.g., the moderator declares: “the moderator declares” (no problem).

• the review team moderates walkthrough session, (xvi) The developer submits artefact for review, moderator moderates walkthrough session, (vii) The developer only has feedback and may terminate the review if need to reach consensus.

• the review team moderates walkthrough session, (xv) The developer submits artefact for review, moderator moderates walkthrough session, (vi) The developer only has feedback and may terminate the review if need to reach consensus.

xviii) The developer only has feedback and may terminate the review if need to reach consensus.

xix) The developer only has feedback and may terminate the review if need to reach consensus.
Dependability Case

• Establishes the critical properties, that the software, in concert with other components, that the system satisfies them. That is, you can place your trust in it.

Proposed Approach

Dependability Case

Technology Revisited

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Weaknesses:
  - Partial results may not be considered useful.
  - Tool support may not be available.
  - Some toolchains are reviewed.
  - Entry costs may be high.
  - Results may not be on actual execution.

- Strengths:
  - Correctness proofs can be automatic.
  - Faster than testing.
  - Can provide some tools are:
    - Highly effective
    - Good effort/effect ratio achievable
    - Reporting to be good effort/effect ratio achievable
    - Partial results may be taken at any time.

- Challenges:
  - Many false positives can be very annoying to developers.
  - False negatives can be hard to detect.
  - False positives can be challenging.
  - Tool support may not be available.
  - Errors overlooked.

- Approaches:
  - Proving "can" check partial results faster than testing.
  - Run "can" proof with partial results.
  - Partial results may be taken at any time.
  - Some tools are available (few commercial tools).
  - Review toolchain.
  - Configuring the tools can be challenging.
  - False positives may be very annoying.
  - False negatives can be hard to detect.

- Analysis:
  - Entry cost high
  - No results on actual execution.
  - No toolchain reviewed.
  - Many false positives can be very annoying to developers.
• Topic Area: Software Quality Assurance

and structural CFA XP, Scrum limits of testing completeness etc.

role, artefact, activity

(know how to interpret the indications)
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 (L TL, CTL, CTL∗))
• Cyber-Physical Systems - Hybrid Models (Modelling and analysis of cyber-physical systems with hybrid automata)
• Program Verification
• 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

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.

