

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

- Special syntax:**
 - Atomic block `atomic {c;}`
 - (requires p) — pre-condition, p is basically a C expression
 - (ensures q) — post-condition, q is basically a C expression
 - (linearize invariant, `exp`) — (basically) a C expression
 - (linearize `exp`) — loop invariant, `exp` is (basically) a C expression
 - (linearize p) — (linearize invariant, p is (basically) a C expression
- (linearize p) — VCC considers concurrent programs, we need to declare for each procedure what global variables it is allowed to write to (also checked by VCC)
- Special expressions:**
 - `!linearize, !local(v)` — no other thread writes to variable v (in pre-conditions)
 - `!valid(v)` — the value of v when procedure was called (useful for post-conditions)
 - `!results` — return value of procedure (useful for post-conditions)

18₁₄

VCC Syntax Example

```

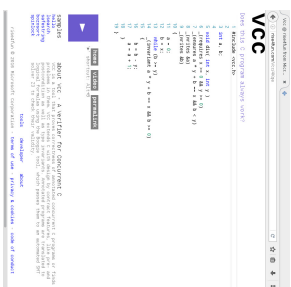
1  pthread_t t1, t2;
2  int a, b;
3
4  void div1 ( int x, int y )
5  {
6      _requires x >= 0 && y >= 0;
7      _ensures x >= 0 && b > y;
8      _writes (a)
9      {
10         a = x;
11         b = x;
12         while (b < y)
13             b = b + 1;
14         _invariant a <= y <= b && b >= 0;
15         a = a + 1;
16     }
17 }

```

$DIV \equiv a := 0; b := x; \text{ while } b \geq y \text{ do } b := b - y; a := a + 1 \text{ od}$
 $\{x \geq 0 \wedge y \geq 0\} DIV \{x \geq 0 \wedge y \geq 0\}$

19₁₄

VCC Web-Interface

20₁₄

Interpretation of Results

- VCC result: **"verification succeeded"**
 - We can **only** conclude that the tool — under its representation of the C-standard under its platform assumptions (32-bit), etc. — claims that there is a proof for $\models \{p\} DIV \{q\}$.
 - May be due to an error in the tool! (That's a false negative then)
 - Yet we can ask for a **proof** of the proof and check it manually (if really possible in practice) or with other tools like interactive theorem provers.
 - Note:** $\models \{false\} f \{q\}$ **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 (fined) counter-example from the hints or make loop-invariants (or pre-condition p) stronger, and try again.
 - Other case: **"timeout"** etc. — completely **inconclusive** outcome.

21₁₄

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 height field is always equal to the number of the nodes in the tree) or those that are **invariantly** be violated when updating the data structure
 - and much more.

22₁₄

Modular Reasoning

23₁₄

Modular Reasoning

We can add another rule for calls of functions $f : F$ (simplest case: only global variables):

$$(R) \frac{\{p\} F \{q\}}{\{p\} f() \{q\}} \quad \text{pre- and post-conditions}$$

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

- $\vdash \{true\} \text{read_number}() \leq \text{result} < 10^9$
- $\vdash (0 \leq x \wedge 0 \leq y) \text{ add}(\text{odd}(x) + \text{odd}(y) \pm \text{odd}(y)) \vee \text{result} \leq 0$
- $\vdash \{true\} \text{skipday}() (0 \leq \text{odd}(\text{sum}) < 10^9 \implies \text{"odd}(\text{sum})" \wedge (\text{odd}(\text{sum}) \leq 0 \implies \text{"-B-"}))$

we may be able to prove our pocket calculator correct.

+	+	+	+	+
+	+	+	+	+
+	+	+	+	+
+	+	+	+	+
+	+	+	+	+

```
int x, y, sum;
int read() {
    // ...
    return x;
}
int add(int x, int y) {
    // ...
    return x + y;
}
int skipday() {
    // ...
    return sum;
}
```

24/04

Return Values and Old Values

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

Can be defined using **auxiliary variables**:

Transform function

over variables $V = \{v_1, \dots, v_n\}$, where $\text{result}, v_i^{\text{old}} \notin V$ into

```
T f() { ...; return expr; }
into
T f() {
    ...;
    v1^old = v1; ...; v_n^old = v_n;
    result := expr;
    return result;
}
```

over $V' = V \cup \{v^{\text{old}} \mid v \in V\} \cup \{\text{result}\}$.

Then $\text{old}(x)$ is just an abbreviation for x^{old} .

25/04

Assertions

Assertions

- Extend the **syntax of deterministic programs** by

$S ::= \dots \mid \text{assert}(B)$

- and the **semantics** by rule

$\{\text{assert}(B), \sigma\} \rightarrow \{E, \sigma\}$ if $\sigma \models B$.

If the asserted boolean expression B does not hold in state σ , the empty program is not reached; otherwise the assertion remains in the first component, **abnormal** program termination.

Extend PD by axiom:

$\{A\} p \mid \text{assert}(p) \{p\}$

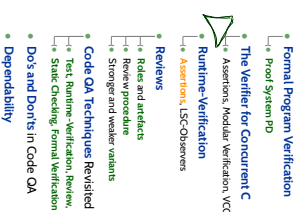
- That is, if p holds **before** the assertion, then we can **continue** with the derivation in PD.

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.

27/04

Content



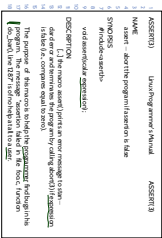
28/04

Run-Time Verification

29/04

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:



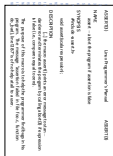
- In C code, `assert` can be disabled in production code (-D NDEBUG).
- Use `java -ea ...` to enable assertion checking (disabled by default).

30/4

Assertions At Work

- The abstract f -example from run-time verification: (specification: $\{p\} f \{q\}$)

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



- Compute the width of a progress bar:



```
int progress_bar_width() {
    // ...
    assert(p);
    // ...
    assert(q);
}
```

31/4

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

$\sigma_0 \xrightarrow{p} \sigma_1 \xrightarrow{q} \sigma_2 \dots \xrightarrow{p_{n-1}} \sigma_{n-1} \xrightarrow{q_{n-1}} \sigma_n \xrightarrow{f} \sigma_{n+1} \dots$



Software S

- Assume there are functions $check_p$ and $check_q$.
- **Idea:** create software S' by
 - extending S by implementations of $check_p$ and $check_q$.
 - call $check_q$ right after entering f , and which do not modify the program state (except for program counter).
- For S' , obtain computation paths like:

$$\sigma_0 \xrightarrow{p} \sigma_1 \xrightarrow{q} \sigma_2 \dots \xrightarrow{p_{n-1}} \sigma_{n-1} \xrightarrow{check_q} \sigma_n \xrightarrow{f} \sigma_{n+1} \dots$$
- If $check_p$ and $check_q$ notify us of violations of p or q , then we are notified of f violating its specification when running S' (= at run-time).

33/4

Run-Time Verification: Example



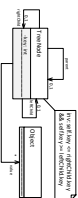
```
int x, y, sum;
int main() {
    while (true) {
        x = nextNumber();
        y = nextNumber();
        sum = add(x, y);
        verify_sum(x, y, sum);
        display();
    }
}
```

```
void verify_sum(int x, int y, int sum) {
    if (sum != (x + y) * 999999999) {
        printf("Error: sum is %d\n", sum);
        abort();
    }
}
```

34/4

Assertions At Work II

- Recall the structure model with Prop-OCL constant from Exercise Sheet 4/2012.
- Assume, we add a method `set_key()` to class `TreeNode`:



- We can check consistency with the Prop-OCL constraint at runtime by using assertions
- ```

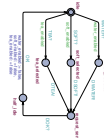
1 private int key;
2 TreeNode parent; leftChild; rightChild;
3 public void get_key() { return key; }
4 public void set_key(int new_key) {
5 key = new_key;
6 }
7
8 public void set_key(int new_key) {
9 // ...
10 // ...
11 // ...
12 // ...
13 // ...
14 // ...
15 // ...
16 // ...
17 // ...
18 // ...
19 // ...
20 // ...
21 // ...
22 // ...
23 // ...
24 // ...
25 // ...
26 // ...
27 // ...
28 // ...
29 // ...
30 // ...
31 // ...
32 // ...
33 // ...
34 // ...
35 // ...
36 // ...
37 // ...
38 // ...
39 // ...
40 // ...
41 // ...
42 // ...
43 // ...
44 // ...
45 // ...
46 // ...
47 // ...
48 // ...
49 // ...
50 // ...
51 // ...
52 // ...
53 // ...
54 // ...
55 // ...
56 // ...
57 // ...
58 // ...
59 // ...
60 // ...
61 // ...
62 // ...
63 // ...
64 // ...
65 // ...
66 // ...
67 // ...
68 // ...
69 // ...
70 // ...
71 // ...
72 // ...
73 // ...
74 // ...
75 // ...
76 // ...
77 // ...
78 // ...
79 // ...
80 // ...
81 // ...
82 // ...
83 // ...
84 // ...
85 // ...
86 // ...
87 // ...
88 // ...
89 // ...
90 // ...
91 // ...
92 // ...
93 // ...
94 // ...
95 // ...
96 // ...
97 // ...
98 // ...
99 // ...
100 // ...

```

35/4

## More Complex Run-Time Verification: LSC Observers

### Checksum



```

1 // ...
2 // ...
3 // ...
4 // ...
5 // ...
6 // ...
7 // ...
8 // ...
9 // ...
10 // ...
11 // ...
12 // ...
13 // ...
14 // ...
15 // ...
16 // ...
17 // ...
18 // ...
19 // ...
20 // ...
21 // ...
22 // ...
23 // ...
24 // ...
25 // ...
26 // ...
27 // ...
28 // ...
29 // ...
30 // ...
31 // ...
32 // ...
33 // ...
34 // ...
35 // ...
36 // ...
37 // ...
38 // ...
39 // ...
40 // ...
41 // ...
42 // ...
43 // ...
44 // ...
45 // ...
46 // ...
47 // ...
48 // ...
49 // ...
50 // ...
51 // ...
52 // ...
53 // ...
54 // ...
55 // ...
56 // ...
57 // ...
58 // ...
59 // ...
60 // ...
61 // ...
62 // ...
63 // ...
64 // ...
65 // ...
66 // ...
67 // ...
68 // ...
69 // ...
70 // ...
71 // ...
72 // ...
73 // ...
74 // ...
75 // ...
76 // ...
77 // ...
78 // ...
79 // ...
80 // ...
81 // ...
82 // ...
83 // ...
84 // ...
85 // ...
86 // ...
87 // ...
88 // ...
89 // ...
90 // ...
91 // ...
92 // ...
93 // ...
94 // ...
95 // ...
96 // ...
97 // ...
98 // ...
99 // ...
100 // ...

```



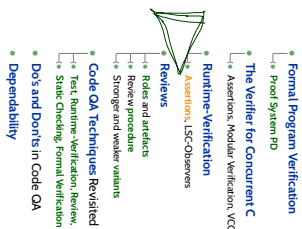
35/4

## Run-Time Verification: Discussion

Experience: Assertions for pre/post conditions and intermediate invariants are an extremely powerful tool with a very attractive gain/effort ratio (low effort, high gain).

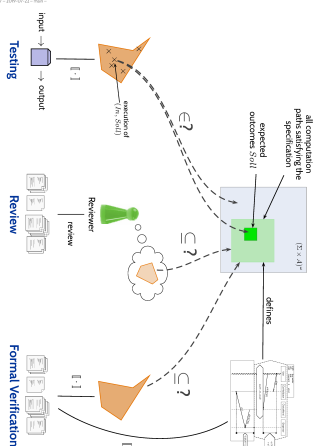
- Assertion effectively work as **safe-guard** against **unexpected use** of functions and **regression**, e.g. during later maintenance or efficiency improvement.
- Assertions can serve as **formal support of documentation**.  
 ▶ assert( expr );  
 means:  
 "To re-enter, at this point in the program, I expect condition expr to hold".  
 Be good to your readers: **add a comment** that explains the **assert**.

## Content

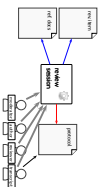


## Review

*Recall: Three Basic Directions*

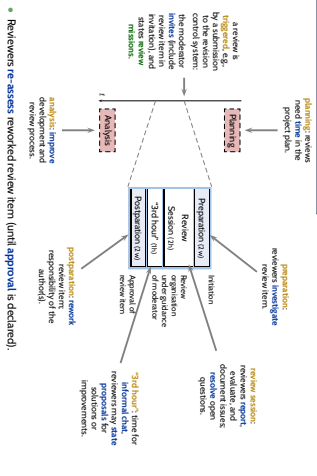


## Reviews



- **Input to Review Session:**
  - **Review Item:** can be every conceivable human-readable part of software documentation: models, test cases, code, etc.
  - **Goal:** *What is the purpose of this?* **What is the artifact?** **What is the context?** **What is examined, not the human (who created it).**
  - **Reference documents:** need to enable an assessor to find requirements specifications, guidelines, etc.
- **Review team:** consists of everybody but the author(s).

### Review Procedure Over Time



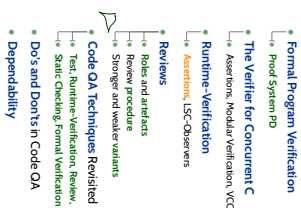


- (a) The moderator organises the review, issues invitations, supervises the review.
  - (b) The moderator may terminate the review if continuation is not possible, e.g. due to no participation or problem solving.
  - (c) The review session is **limited to 1 hour**, if needed, organize more sessions.
  - (d) The review team is under review, not the author(s).
  - (e) The review team may discuss accordingly. Authors neither defend themselves for the review.
  - (f) Rules are **not fixed**, e.g. the moderator does not act as reviewer. (Exception: author may write team opt.)
  - (g) **Style sheets** (outside fixed conventions) are **not discussed**.
- 
- (a) The review team is **not** supposed to **write** **recommendations** for the author(s).
  - (b) Each reviewer gets the opportunity present their findings/opinionally.
  - (c) Reviewers need to reach **consensus** on scores, comments is noted down.
  - (d) Issues are classified as:
    - critical review (useful for proposal)
    - major (substantive) review (e.g. need for additional effort)
    - minor (point for problem)
  - (e) The review team declares:
    - accept without changes,
    - accept with changes,
    - do not accept.
  - (f) The protocol is signed by all participants.

### Stronger and Weaker Review Variants



## Content



## Techniques Revisited

| Test                 | algorithmic | proof transformer | toolchain considered | evaluation | proof correct | partial results | entry cost |
|----------------------|-------------|-------------------|----------------------|------------|---------------|-----------------|------------|
| Runtime Verification | ✓           | ✓                 | ✗                    | ✗          | ✓             | ✓               | ✓          |
| Static Checking      |             |                   |                      |            |               |                 |            |
| Review               |             |                   |                      |            |               |                 |            |
| Verification         |             |                   |                      |            |               |                 |            |

### Strengths

- can be fully automatic (yet not easy for GUI program)
- negative test proves "program not completely broken on" "can run" (for 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 proof of correctness
- creating test cases for complex functions for complex conditional 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)

## Code Quality Assurance Techniques Revisited

## Techniques Revisited

|                      |            |                 |                    |            |               |                 |       |
|----------------------|------------|-----------------|--------------------|------------|---------------|-----------------|-------|
| Test                 | auto-matic | prove "can run" | hooshin considered | extra-five | prove correct | partial results | entry |
| Runtime-Verification | ✓          | (✓)             | ✓                  | (X)        | X             | ✓               | (✓)   |
| Review               |            |                 |                    |            |               |                 |       |
| Static-Checking      |            |                 |                    |            |               |                 |       |
| Verification         |            |                 |                    |            |               |                 |       |

### Strengths

- fully automatic force observers are in place;
- provides counter-example
- (really) final product is examined, thus toothbin and platform considered
- one can stop at any time and take partial results;
- as a result, 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 **very incomplete**, so no correctness proofs
- constructing observers for complex properties may be **difficult**, one needs to learn how to construct observers.

Techniques Revisited

|                          | auto-<br>matic | prove<br>turnout | toolchain<br>considered | exhaustive | prove<br>correct | partial<br>results | entry<br>cost |
|--------------------------|----------------|------------------|-------------------------|------------|------------------|--------------------|---------------|
| Test                     | ✓              | ✓                | ✓                       | ✗          | ✗                | ✓                  | ✓             |
| Runtime-<br>Verification | ✓              | ✓                | ✓                       | ✗          | ✗                | ✓                  | ✓             |
| Static Checking          | ✗              | ✗                | ✗                       | ✓          | ✓                | ✓                  | ✓             |
| Verification             |                |                  |                         |            |                  |                    |               |

- 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/cost ratio achievable.

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

460.4

Techniques Revisited

|                          | auto-<br>matic | prove<br>turnout | toolchain<br>considered | exhaustive | prove<br>correct | partial<br>results | entry<br>cost |
|--------------------------|----------------|------------------|-------------------------|------------|------------------|--------------------|---------------|
| Test                     | ✓              | ✓                | ✓                       | ✗          | ✗                | ✓                  | ✓             |
| Runtime-<br>Verification | ✓              | ✓                | ✓                       | ✗          | ✗                | ✓                  | ✓             |
| Static Checking          | ✗              | ✗                | ✗                       | ✓          | ✓                | ✓                  | ✗             |
| Verification             |                |                  |                         |            |                  |                    |               |

- Strengths:**
- three are (commercially, fully automatic tools (Coventry, 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 insecure concerning (if few false positives warned);
  - 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 tool (to limit false positives) can be challenging.

460.4

Techniques Revisited

|                          | auto-<br>matic | prove<br>turnout | toolchain<br>considered | exhaustive | prove<br>correct | partial<br>results | entry<br>cost |
|--------------------------|----------------|------------------|-------------------------|------------|------------------|--------------------|---------------|
| Test                     | ✓              | ✓                | ✓                       | ✗          | ✗                | ✓                  | ✓             |
| Runtime-<br>Verification | ✓              | ✓                | ✓                       | ✗          | ✗                | ✓                  | ✓             |
| Static Checking          | ✗              | ✗                | ✗                       | ✓          | ✓                | ✓                  | ✗             |
| Verification             | ✓              | ✗                | ✗                       | ✓          | ✓                | ✗                  | ✗             |

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

- Weaknesses:**
- no results on actual execution, toolchain not reviewed;
  - can be very insecure concerning (if few false positives warned);
  - entry cost high; significant tuning is useful to know how to deal with tool limitations;
  - proving things "challenging" (trying to find a proof does not allow any useful conclusion);
  - false negative (broken program "proves" correct) hard to detect.

460.4





Techniques Revisited

|                          | auto-<br>matic | prove<br>turnout | toolchain<br>considered | exhaustive | prove<br>correct | partial<br>results | entry<br>cost |
|--------------------------|----------------|------------------|-------------------------|------------|------------------|--------------------|---------------|
| Test                     | ✓              | ✓                | ✓                       | ✗          | ✗                | ✓                  | ✓             |
| Runtime-<br>Verification | ✓              | ✓                | ✓                       | ✗          | ✗                | ✓                  | ✓             |
| Static Checking          | ✗              | ✗                | ✗                       | ✓          | ✓                | ✓                  | ✗             |
| Verification             | ✓              | ✗                | ✗                       | ✓          | ✓                | ✗                  | ✗             |

Some Final, General Guidelines

460.4

Do's and Don'ts in Code Quality Assurance

-  **Avoid using special examination versions for examination.**  
(Fish-tummers, subls, etc. may have errors which may cause false positives and (!) negatives)
-  **Avoid to stop examination when the first error is detected.**  
Clear: Examination should be aborted if the examined program is not executable at all.
-  **Do not modify the artefact under examination during examinatin.**  
otherwise, it is unclear what exactly has been examined ("moving target").  
(examination results need to be uniquely traceable to one artefact version)
  - fundamental flaws are sometimes easier to detect
  - with a complete picture of unsuccessful/dysfunctional tests;
  - changes are particularly error-prone; should not happen "en passant" in examination;
  - being flaws during examination may cause them to go uncounted in the statistics
-  **Do not switch (line gained) between examination and debugging.**  
an examiner fixing flaws would violate the role assignment.

460.4

Dependency Case

49/4

Looking Back:  
17 Lectures on Software Engineering

52/4

Proposed: Dependency Cases (Jackson, 2009)

- A **dependable** system is one you can **depend** on – that is, you can place your trust in it
- Develops [should] **express the critical properties** and **make an explicit argument** that the system satisfies them."

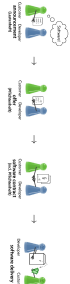
Proposed Approach

- Identify the critical requirements, and determine what level of confidence is needed (Most systems do also have non-critical requirements.)
- Construct a **dependency case**, i.e. an **argument**, that the software, in concert with other components, establishes the **critical properties**.
- The **dependency case** should be:
  - **auditable**: can (easily) be evaluated by third-party certifier.
  - **complete**: no holes in the argument. (e.g. assumptions on complex, unproven depend 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.

Copyright 1997-2009, MIT

50/4

Contents of the Course



|                                 |       |           |
|---------------------------------|-------|-----------|
| Introduction                    | 1-2   | 22.4, Mon |
| Metrics, Costs, Design Patterns | 3-6   | 22.4, Mon |
| Requirements Engineering        | 7-11  | 22.4, Mon |
| Software Design                 | 12-15 | 22.4, Mon |
| Modeling                        | 16-18 | 22.4, Mon |
| Testing                         | 19-22 | 22.4, Mon |
| Deployment                      | 23-24 | 22.4, Mon |

Copyright 1997-2009, MIT

53/4

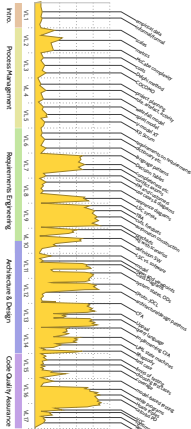
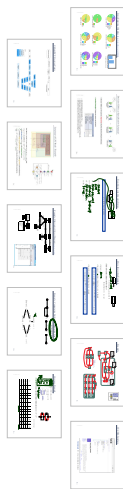
Tell Them What You've Told Them...

- **Runtime Verification**
  - (as the course suggests) checks properties of **program run-time**
  - generates use of **assertions** can be a valuable safe-guard against **runtime errors** (e.g. **runtime errors**, **runtime errors**, etc.)
  - and some at formal documentation of (intermediated) assumptions. Very attractive effort / effect ratio.
- Review (structured examination of artifacts by humans)
- (mid-variant) advocated in the XP approach.
- **not uncommon**:
  - lead programmer reviews all commits from team members.
  - literature reports good effect/ratio into achievable.
- All approaches to code quality assurance have their **advantages and drawbacks**.
- Which to use? It depends!
- Overall: Consider **Dependency Cases**
  - an **auditable** coming along with an **argument**, that it satisfies the **critical properties**.

Copyright 1997-2009, MIT

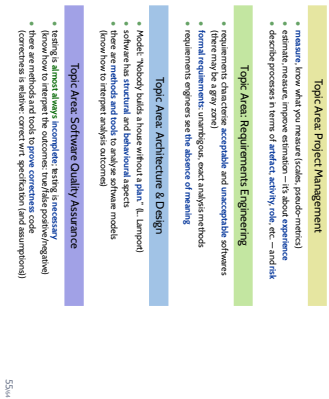
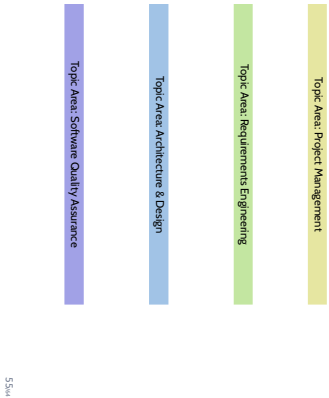
54/4

What Did We Do?



Copyright 1997-2009, MIT

54/4



That's Today's Software Engineering — More or Less...



Questions?

Advertisements

Advertisement

- **Further studies:**
  - **Real-Time Systems** (incl. 2019/20)  
(specification and verification of real-time systems)
  - **Software Design, Modelling, and Analysis in UML** (incl. 2019/20)  
(a formal, in-depth view on structural and behavioural modelling)
  - **Cyber-Physical Systems I - Discrete Models**  
(more on variants of C&A and aspects UML, 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**  
(the basis for program verification)
- <https://svt.informatik.uni-freiburg.de/teaching>

60/64

Advertisement

- **Individual Projects**  
(BS, MSc, project, Lab Project, BS, 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 2020.
  - **contact us** (around early September / early March).
- Want to be a **scientific student assistant**?
  - **contact us**.

61/64



## References

**References**

Fagan, M. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 5(3):182-211.

Fagan, M. (1986). Advances in software inspections. *IEEE Transactions On Software Engineering*, 12(7):744-751.

Heane, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576-590.

Jackson, D. (2009). *A direct path to dependable software*. Comm. ACM, 52(4).

Ludewig, J. and Uthair, H. (2013). *Software Engineering*. fourth edition, 3. edition.