L3 Mention Informatique
Parcours Informatique et MIAGE

Génie Logiciel Avancé - Advanced Software Engineering

Part IV : An Introduction to Test

Burkhart Wolff
wolff@lri.fr

Validation and Verification : A Clarification

- Validation :
  - Does the system meet the clients requirements ?
  - Will the performance be sufficient ?
  - Will the usability be sufficient ?

Validation and Verification : A Clarification

- Validation :
  - Does the system meet the clients requirements ?
  - Will the performance be sufficient ?
  - Will the usability be sufficient ?

Do we build the right system ?

Validation and Verification : A Clarification

- Validation :
  - Does the system meet the clients requirements ?
  - Will the performance be sufficient ?
  - Will the usability be sufficient ?

Do we build the right system ?

- Verification: Does the system meet the specification ?

Validation and Verification : A Clarification

- Validation :
  - Does the system meet the clients requirements ?
  - Will the performance be sufficient ?
  - Will the usability be sufficient ?

Do we build the right system ?

- Verification: Does the system meet the specification ?

Do we build the system right ?   Is it « correct » ?

How to do Validation ?

- Measuring customer satisfaction ...
  (well, that's post-hoc, and its difficult to predict)

- Interviews, inspections (again post-hoc)

- How to validate a system early?
  - Simulation Environments like Mathlab/Simuling (Embedded Systems).
  - Early prototypes, including performance analysis
    (for Software, but also Hardware-Processors)
  - Mock-ups (functionality, ergonomics of GUI's,,...)

  - Test and Animation on the basis of formal specifications

How to do Verification ?

- Test and Proof on the basis of formal specifications (e.g., à la MOAL !) against programs or system

How to do Verification ?

- Test and Proof on the basis of formal specifications (e.g., à la OCL !) against programs ...

  In the sequel, we concentrate on Testing for the purpose of Verification ... (not really validation)

  The "Testing-As-Model-Validation" technique is, however, very prominent in "reverse-engineering" processes.

Test vs. Proof

- **Note**:

  Some researcher consider "test" as opposite to "proof"! And they tend to apply the term "verification" only to proof and model-checking techniques…

  But:

  - Modern SE terminology uses the term "verification " to englobe both "test" and "proof" techniques
  - The prejudice is somewhat outdated; it goes back to Dijkstra's and van Dalens famous statement in 72:
    "A test can only reveal the presence of bugs, but not their absence …"
  - … but there is growing consensus nowadays that no technique can guarantee "the (total) absence of errors"
  - many test critics refer to unsystematic tests

Test vs. Proof

- **Note**:

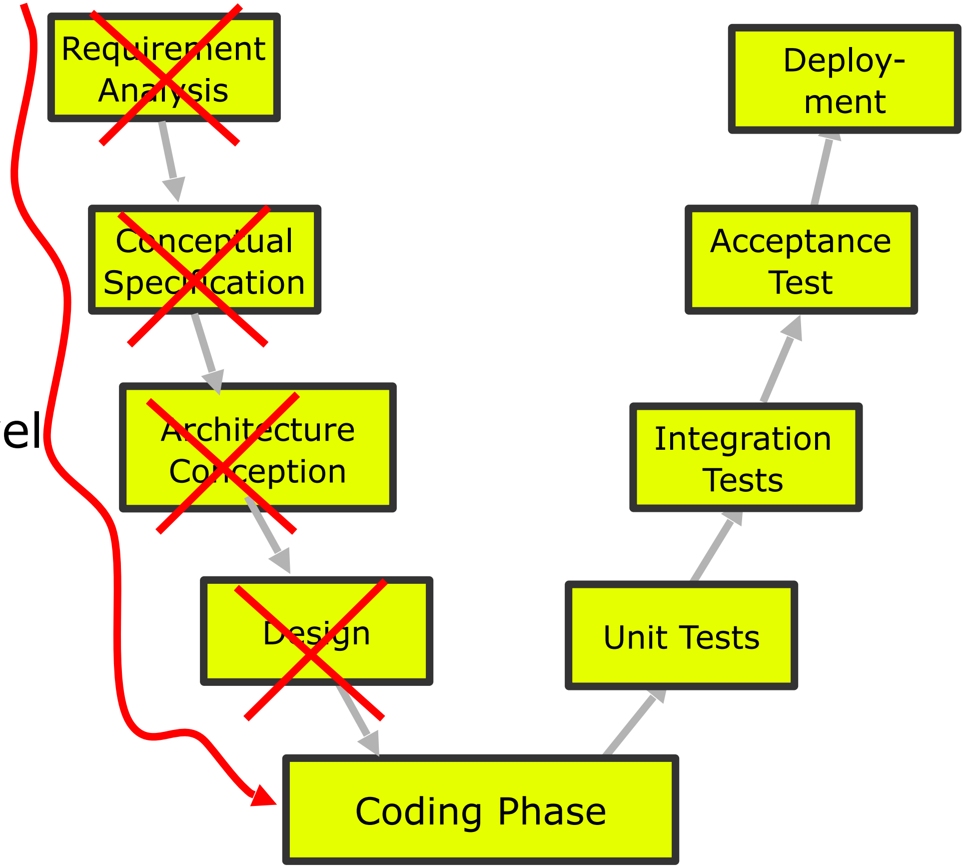
  We consider (systematic!) test more as an approximation to formal proof. Reasons:

  - The nature of the approximation can be made formally precise (via explicit test-assumptions ...)
  - both techniques, model-based tests and formal verification, share a lot of technologies ...
  - even full-blown proof attempts may profit from testing, since it can help to debug specs early and cost-effectively
  - Moreover, tests are based on different application hypothesis than other verification techniques, combining them increases confidence ...

Testing in the SE Process

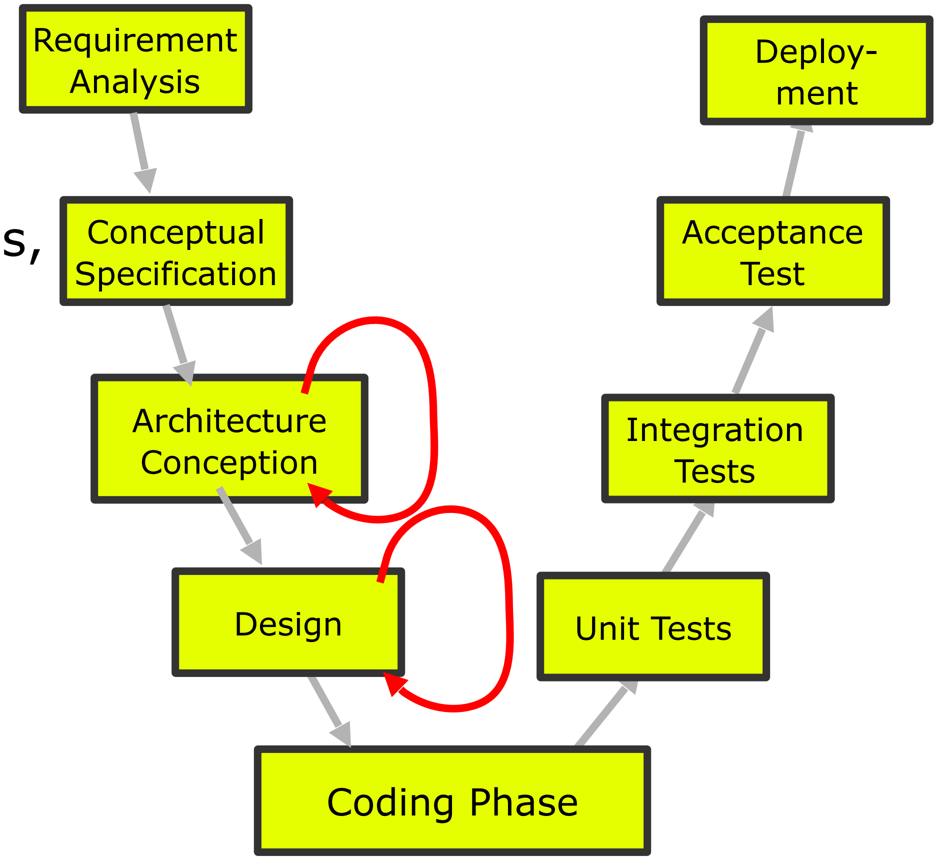
- Where are Test-activities integrated in the SE-Process:
  - Extreme Programming/ Agile Development:

    On the methodological level
  - Instead of requirements, models, specs, … avoiding “Upfront bureaucracy”, one writes and maintains test suites …



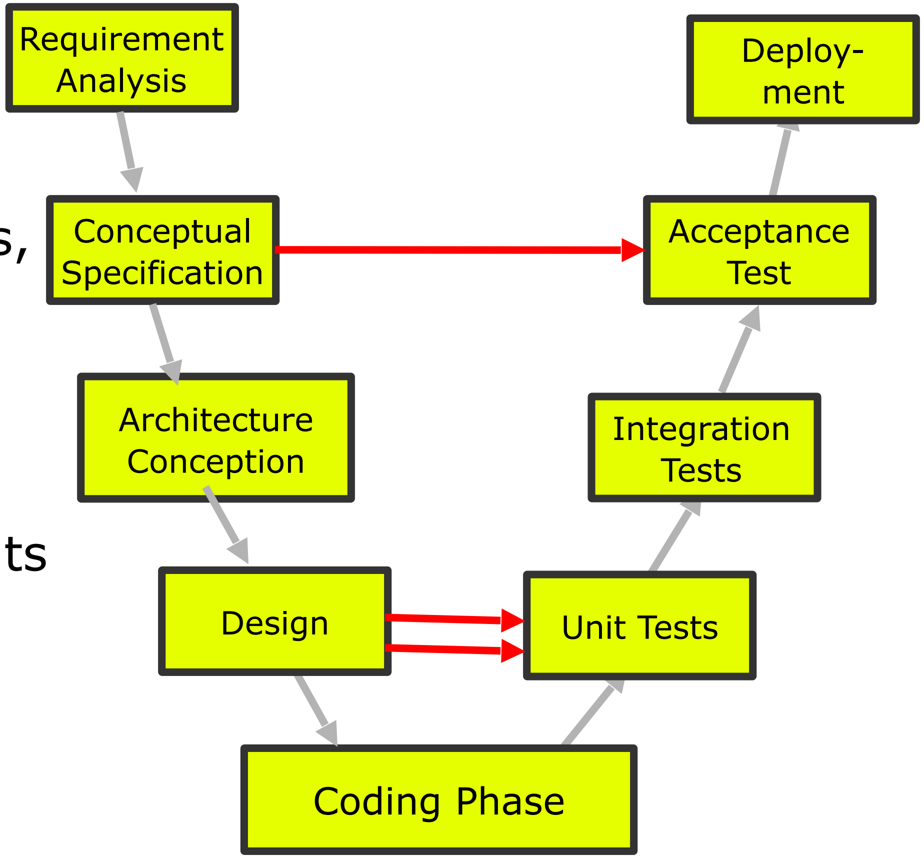
Testing in the SE Process

- Where are Test-activities integrated in the SE-Process:
  - On a conventional V process, (or RUP or CENELEC or …)
  - … in the early phases as validation technique for models / specs



Testing in the SE Process

- Where are Test-activities integrated in the SE-Process:
  - ➢ On a conventional V process, (or RUP or CENELEC or ...)
  - ➢ ... in the later phases as verification technique for code / modules / components against models/specs



Recall partI :

The Problem for Software-Quality

- A Very General Rule of Thumb:

  - Programming is not enough ! Overall,
    It is not even the most important cost-factor !!

  - A global estimate of project activities:

Percentage of «Coding» ?	15 - 20 %
Proportion of Validation et Verification ?	~20%
All others : (Analysis,Design, Certification, Maintenance, Management).	60 %

  - These figures may vary substantially in particular industries (Automotive, Railways, Medical…)

Verification Costs

- Conclusion:
  - verification by test or proof is vitally important, and also critical in the development
  - to do it cost-effectively, it requires
    - a lot of expertise on products and process
    - a lot of knowledge over methods, tools, and tool chains ...

Overview on the part on « Test »

- WHAT IS TESTING ?
- A taxonomy on types of tests
  - Static Test / Dynamic (*Runtime*) Test
  - Structural Test / Functional Test
  - Statistic Tests
- Functional Test; Link to UML/OCL
  - Dynamic Unit Tests, Static Unit Tests,
  - Coverage Criteria
- Structural Tests
  - Control Flow and Data Flow Graphs
  - Tests and executed paths. Undecidability.
  - Coverage Criteria

What is testing ?

- It is an approximation to verification by proof, based on different hypothesis
- Main Advantage: can be integrated into SE processes fairly easy.
- Main emphasis: finding bugs early,
  - either in the model
  - or in the program
  - or in both.

⇒ functional testing aka "black-box-testing"

⇒ structural testing aka "white-box-testing"

⇒ "grey-box-testing"

What is systematic (formal) testing ?

- A systematic test is:
  - process using programs and specifications
    to compute a set of test-cases
    under controlled conditions.
  - Objective: the set of test-cases is
    complete wrt. to a given adequacy criterion
    telling that we "tested enough" in a certain sense
  - Ideally: the process is tool-supported by a
    test-generation algorithm

Known Limits of Systematic Testing

- We said, test is an approximation to verification, usually easier (but less expensive)

- Note: Sometimes it is easier to verify by proof than by test. In particular:

  - low-level OS implementations like
    memory allocation, garbage collection
    memory virtualization, crypt-algorithms, ...
  - non-deterministic programs with
    no control over the non-determinism.

Taxonomy: Static / Dynamic Tests

- **static**: running a program before deployment on data carefully constructed by the tester
  - analyse the result on the basis of all components
  - working on some classes of executions symbolically = representing infinitely many executions
- **dynamic**: running the programme after deployment, on “real data” as imposed by the application domain
  - experiment with the “real” behaviour
  - essentially used for post-hoc analysis and debugging

Taxonomy: Unit / Sequence / Adaptive Tests

- **unit testing**: testing of a local component (function, module), typically only one step of the underlying state.
  (In functional programs, thats essentially all what
  you have to do!)

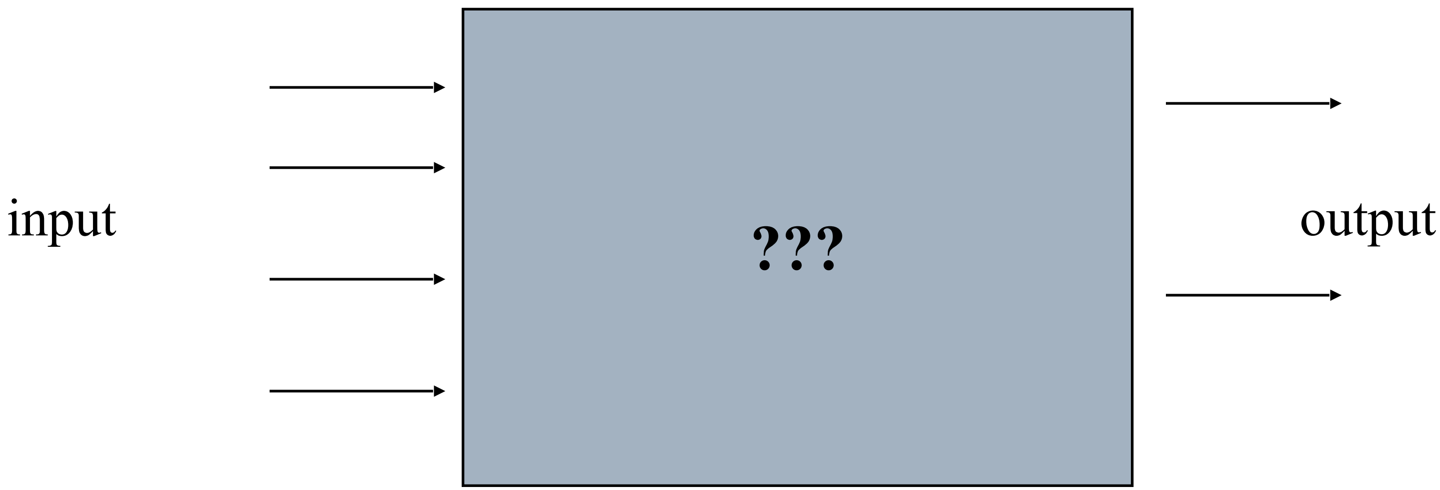
- **sequence testing**: testing of a local component (function, module), but typicallY sequences of executions,
  which typically depend on internal state

- **adaptive testing**: testing components by sequences
  of steps, but these sequences represent communication where later parts
  in the sequence depend on what has
  been earlier communicated

- **random/statistical testing**: not treated here.

Functional ("Black-box") Unit Test

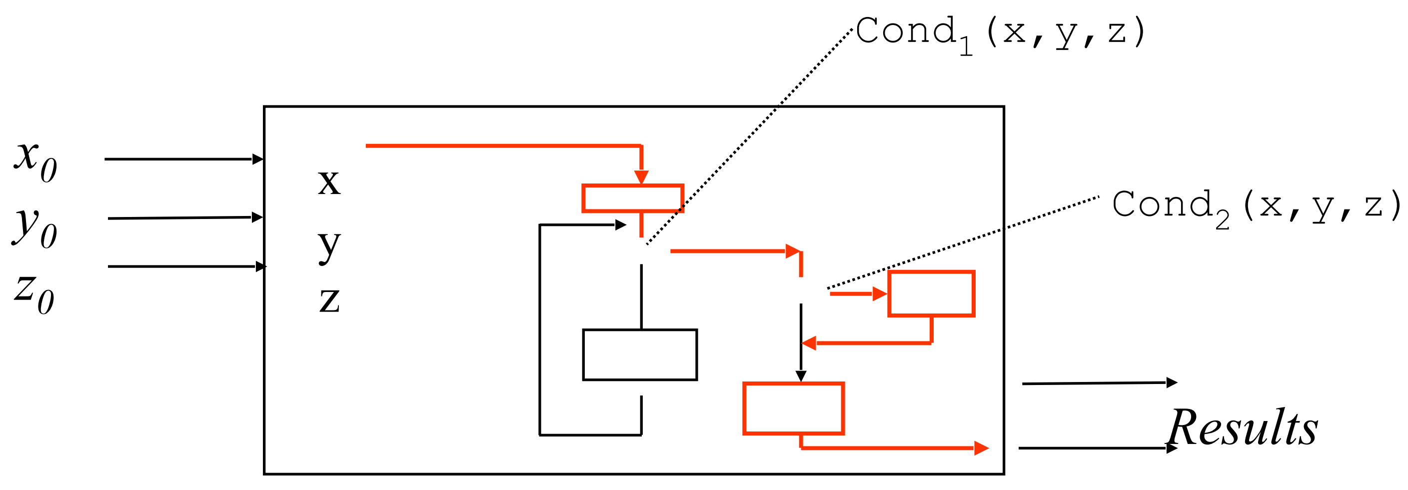
- We got the spec, but not the program, which is considered a black box:



we focus on what the program *should* do !!!

Structural ("white-box") Tests

- we select "critical" paths
- specification used to verify the obtained results



what the program does and how ...

Functional Unit Test : An Example

The (informal) specification:

Read a "Triangle Object" (with three sides of integral type), and test if it is isoscele, equilateral, or (default) arbitrary.

Each length should be positive.

Let's give it a formal specification,
and develop a test set ...

Functional Unit Test : An Example

The specification in UML/MOAL:

```
Triangles
---------------------------------------------
a, b, c: Integer
---------------------------------------------
- mk(Integer,Integer,Integer):Triangle
- is_Triangle(): {equ (*equilateral*),
                  iso (*isosceles*),
                  arb (*arbitrary*)}
```

Functional Unit Test : An Example

We add the constraints:

```
Triangles
----------------------------------------
a, b, c: Integer
----------------------------------------
- mk(Integer,Integer,Integer):Triangle
- is_Triangle(): {equ (*equilateral*),
                  iso (*isosceles*),
                  arb (*arbitrary*)}
```

inv $0<a \wedge 0<b \wedge 0<c$

inv $c \leq a+b \wedge a \leq b+c \wedge b \leq c+a$

operation t.is_Triangle():

post $t.a=t.b \wedge t.b=t.c \longrightarrow result=equ$

post $(t.a \neq t.b \vee t.b \neq t.c) \wedge (t.a=t.b \vee t.b=t.c \vee t.a=t.c)) \longrightarrow result=iso$

post $(t.a \neq t.b \vee t.b \neq t.c \vee t.a \neq t.c)) \longrightarrow result=arb$

Revision: Boolean Logic + Some Basic Rules

- $\neg(a \wedge b) = \neg a \vee \neg b$ (* deMorgan1 *)
- $\neg(a \vee b) = \neg a \wedge \neg b$ (* deMorgan2 *)
- $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
- $\neg(\neg a) = a$
- $a \wedge b = b \wedge a; \quad a \vee b = b \vee a$
- $a \wedge (b \wedge c) = (a \wedge b) \wedge c$
- $a \vee (b \vee c) = (a \vee b) \vee c$
- $a \longrightarrow b = (\neg a) \vee b$
- $(a=b \wedge P(a)) = P(b)$ (* one point rule *)

- let $x = E$ in $C(x) = C(E)$ (* let elimination *)
- if c then C else $D = (c \wedge C) \vee (\neg c \wedge D) = (c \longrightarrow C) \wedge (\neg c \longrightarrow D)$

Intuitive Test-Data Generation

- Consider the test specification (the "Test Case"):

  mk(x,y,z).isTriangle() ≡ X

  i.e. for which input (x,y,z) should an implementation of our contract yield which X ?

  Note that we define mk(0,0,0) to invalid, as well as all other invalid triangles …

Intuitive Test-Data Generation

- an arbitrary valid triangle: (3, 4, 5)
- an equilateral triangle: (5, 5, 5)
- an isoscele triangle and its permutations :

  (6, 6, 7), (7, 6, 6), (6, 7, 6)
- impossible triangles and their permutations :

  (1, 2, 4), (4, 1, 2), (2, 4, 1) -- x + y > z

  (1, 2, 3), (2, 4, 2), (5, 3, 2) -- x + y = z (necessary?)
- a zero length : (0, 5, 4), (4, 0, 5),
- . . .
- Would we have to consider negative values?

Intuitive Test-Data Generation

- Ouf, is there a systematic and automatic way to compute all these tests ?
- Can we avoid hand-written test-scripts ? Avoid the task to maintain them ?
- And the question remains:

When did we test „enough" ?

Functional *Dynamic* Unit Test

Can we exploit the Spec so far ?

How to perform Runtime-Test?

Well, we compile:

```
context X:
inv l_1 : C_1, ...,
inv    l_n : C_n
```

to some checking code (with *assert* as in Junit, ACSL, ...)

```
check_X() = assert(C_1);  ... ; assert(C_n)
```

Functional *Dynamic* Unit Test

How to perform Runtime-Test?

Moreover, compile:

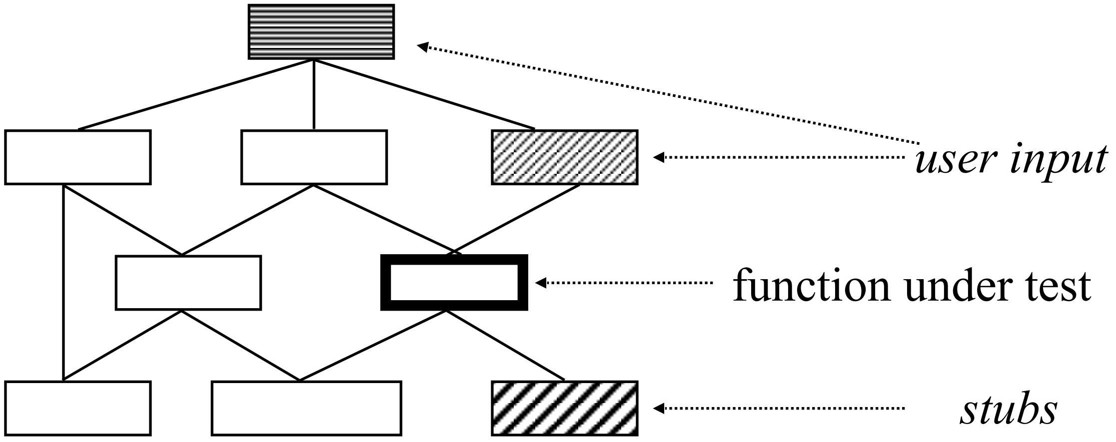
```
context C::m(a_1:C_1,...,a_n:C_n)
pre: P(self,a_1,...,a_n)
post  : Q(self,a_1,...,a_n,result)
```

to some checking code (with assert as in Junit, VCC, ACSL, …)

```
check_C(); check_C_1(); ... ; check_C_n();
assert(P(self,a_1,...,a_n));
result=run_m(self,a_1,...,a_n);
assert(Q(self,a_1,...,a_n,result));
```

Functional *Dynamic* Unit Test in Context

- *Obviously, systematic stimuli of functions is problematic in runtime testing*
- *... there may be a lot of dead code (libraries) (technical problem to measure code coverage)*
- *... there may be an enormous amount of rarely executed code ...*
- *Runtime testing requires a complete program*



Conclusion: Functional Dynamic Tests

- Advantage: any violation of an invariant, a pre-condition or a post-condition is detected for "real" data

- If a violation occurs within an execution of a method, the error is locally reported.

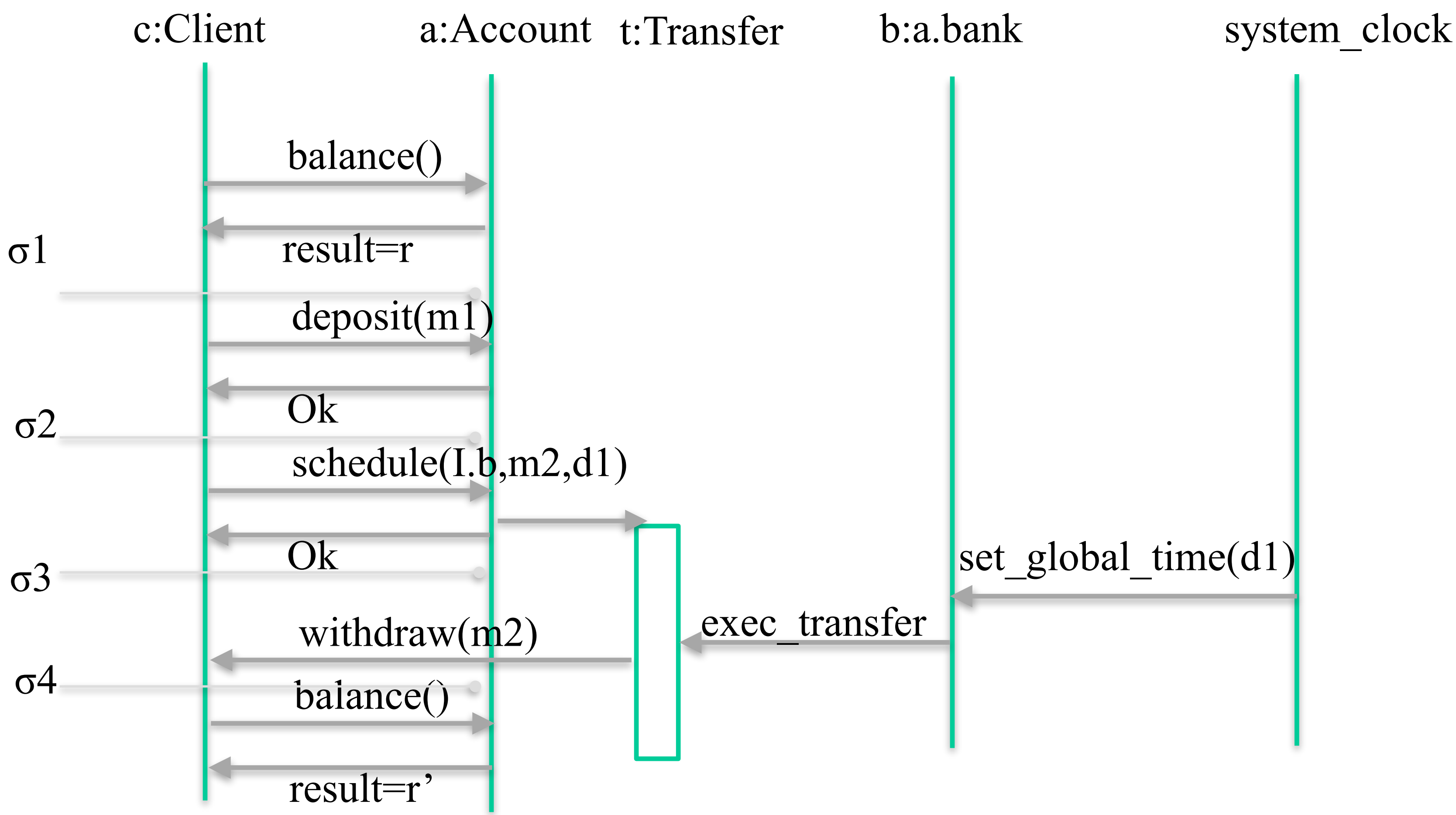
- On the other hand – it is post-hoc. Only when a problem occurred, we know where. And we need complete program.

- Inefficiencies can be partly overcome by optimised compilations, but restricts the technique to very important, easy-to-compute properties

Conclusion: Test in the SE Process

- General questions for verification in a process:
  - How to select test-data ? To which purpose ?
  - How to focus verification activities?
    Where to verify formally, and
    where to test, and when did we test enough?

    Note: The quality of a test is not necessarily increased by the number of test-cases !
  - Automation ? Tools ?


c:Client
a:Account
t:Transfer
b:a.bank
system_clock
balance()
result=r
σ1
deposit(m1)
Ok
σ2
schedule(I.b,m2,d1)
Ok
σ3
set_global_time(d1)
exec_transfer
withdraw(m2)
σ4
balance()
result=r'