L3 Mention Informatique
Parcours Informatique et MIAGE

Génie Logiciel Avancé - Advanced Software Engineering

Part IV : Test Introduction

Burkhart Wolff
wolff@lri.fr

Validation and Verification

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

Validation and Verification

- 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

- 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

- 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 ?

- Mesuring customer satisfaction ...
  (well, that's afterwards, and its difficult)

- Interviews, inspections (again post-hoc)

- How to validate a system early?
  - early prototypes, including performance analysis ...
  - mock-ups (fonctionnality, ergonomics,...)

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

How to do Verification ?

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

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 and Proof Techniques ...

A Philosophical Position Statement : Test vs. Proof

- **Note**:

  Some researcher consider test as opposite to formal proof! Reasons:

  - “A test can only reveal the presence of bugs, but not their absence” (Dijkstra, v. Dalen)

  - ... these researchers referred to unsystematic tests ... (which are, addmittedly, still quite common in SE practice)

A Philosophical Position Statement : Test vs. Proof

- **Note**:

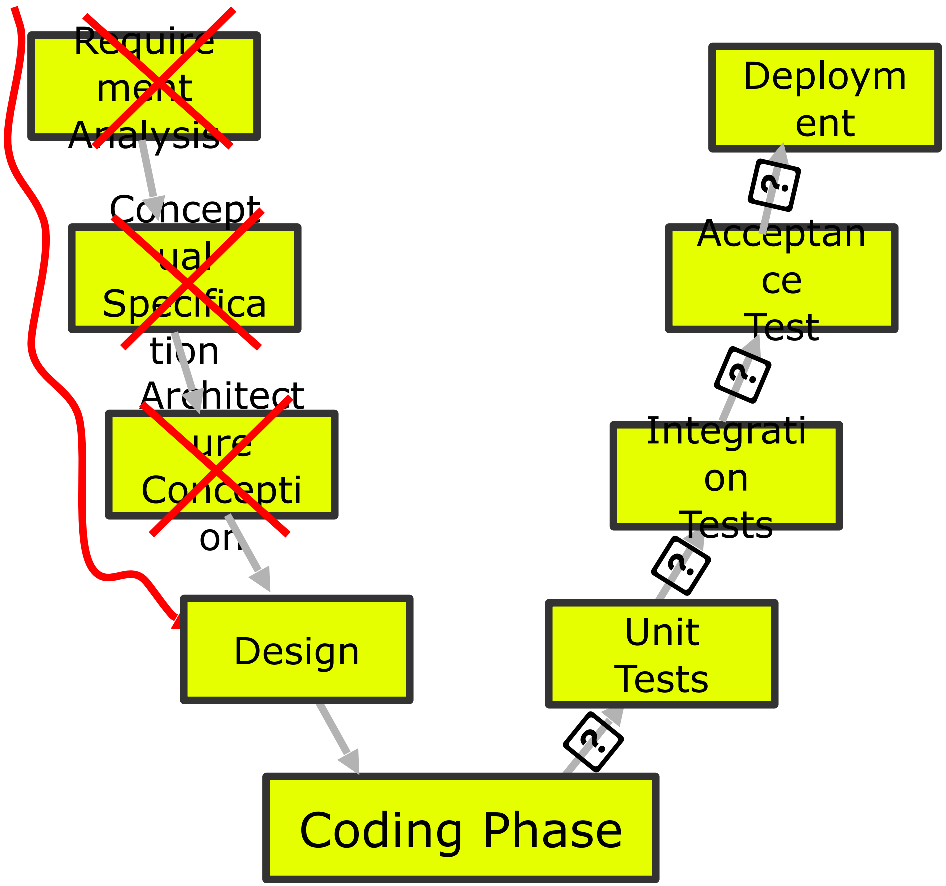
  We consider (systematic!) test more as
  an approximation to formal proof. Reasons:

  - The nature of the approximation can be made formally precise (via explicit test-hypothesis ...)

  - both techniques, model-based tests and formal verification, share a lot of technologies ...

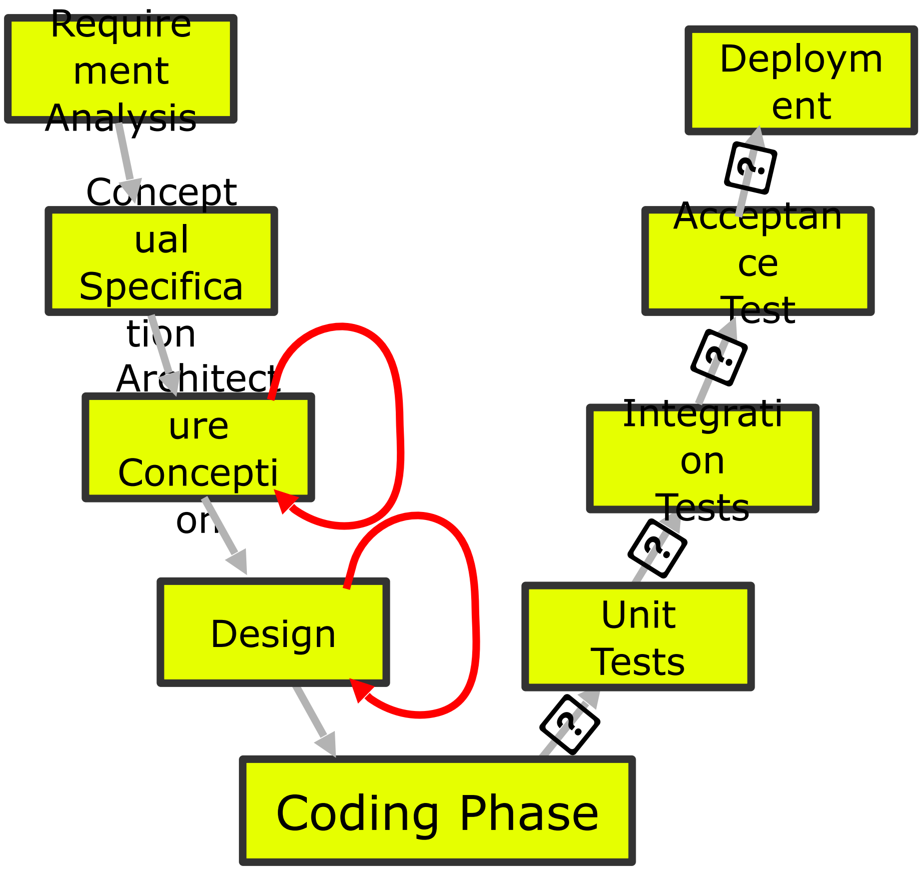
  - even full-blown proof attempts may profit from testing, since it can help to debug specs early and cost-effectively

Testing in the SE Process



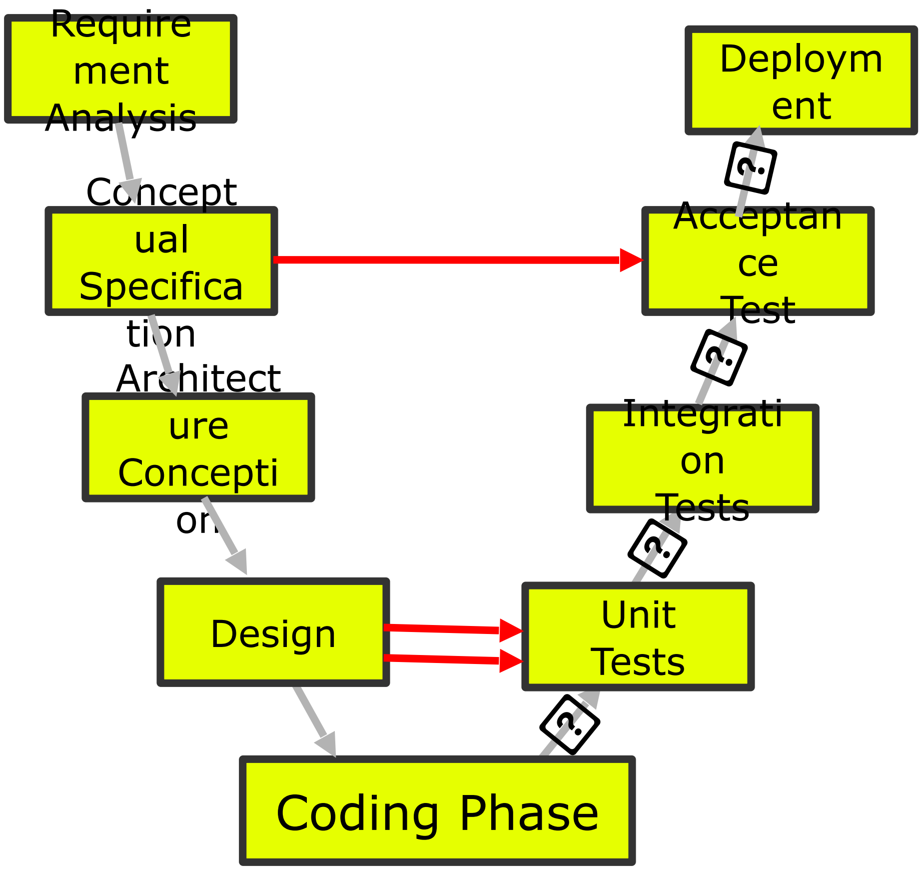
- Where to integrate Tests in the SE-Process:
  - On the methodological level, à la "Extreme Programming" (XP) ?
    No specs, instead writing test scenarios and test cases from the beginning ...

Testing in the SE Process



- Where to integrate Tests in the SE-Process:
  - On the methodological level, à la “Extreme Programming” (XP) ?
    No specs, instead writing test scenarios and test cases from the beginning ...
  - On the specification level for validation ...

Testing in the SE Process



- Where to integrate Tests in the SE-Process:
  - ➢ On the methodological level, à la “Extreme Programming” (XP) ?
    No specs, instead writing test scenarios and test cases from the beginning ...
  - ➢ On the specification level for validation ...
  - ➢ On the specification level against code

Some empirical data ...

- Size of Software ?
  - Peugeot 607 : 2 Mb embedded software
  - Windows 90: 10 Mb. LOC source, Win2000: 30 Mb.
  - Kernel Hyper V: 50000 LOC. (Highly complex, concurrent C)
  - Noyau RedHat 7.1 (2002) : ~2.4 M. LOC, XWindow ~1.8, Mozilla ~2.1 M.
  - Space Shuttle (and its environment) : ~50 MLOC

- Reminder: Development Cost ?
  - Percentage of «Coding» ? 15 - 20 %

Trend: Code is more and more generated (*CASE Tools*)

  - Proportion of Validation et Verification ? ~20% / ~20%

Verification Costs

- costs ? *35 - 50 % of the global effort ?*
- all “real” (large) software has remaining bugs …
- The cost of bug ?
  - the cost to reveal and fix it …

    or:

    the cost of a legal battle it may cause...

    or the potential damage to the image

    (difficult to evaluate, but veeeery real)

    or costs as a result to come later on the market

  - *on the other side – you can't test infinitely, and verification is again 10 times more costly than thoroughly testing !*

Verification Costs

- Conclusion:
  - verification 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 full verification (for ex. by proof)
- Main emphasis: finding bugs early,
  - either in the model
  - or in the program
  - or in both

- A systematic test is:
  - process programs and specifications
    and to compute a set of test-cases
    under controlled conditions.

  - *ideally*: testing is complete if a certain criteria,
    the adequacy criteria is reached.

Limits of testing ?

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

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

  - low-level OS implementations:
    memory allocation, garbage collection
    memory virtualization, ...
    crypt-algorithms, ...

  - non-deterministic programs with
    no control over the non-determinism.

Taxomomy: Static / Dynamic Tests

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

Taxonomy: Unit / Sequence / Reactive Tests

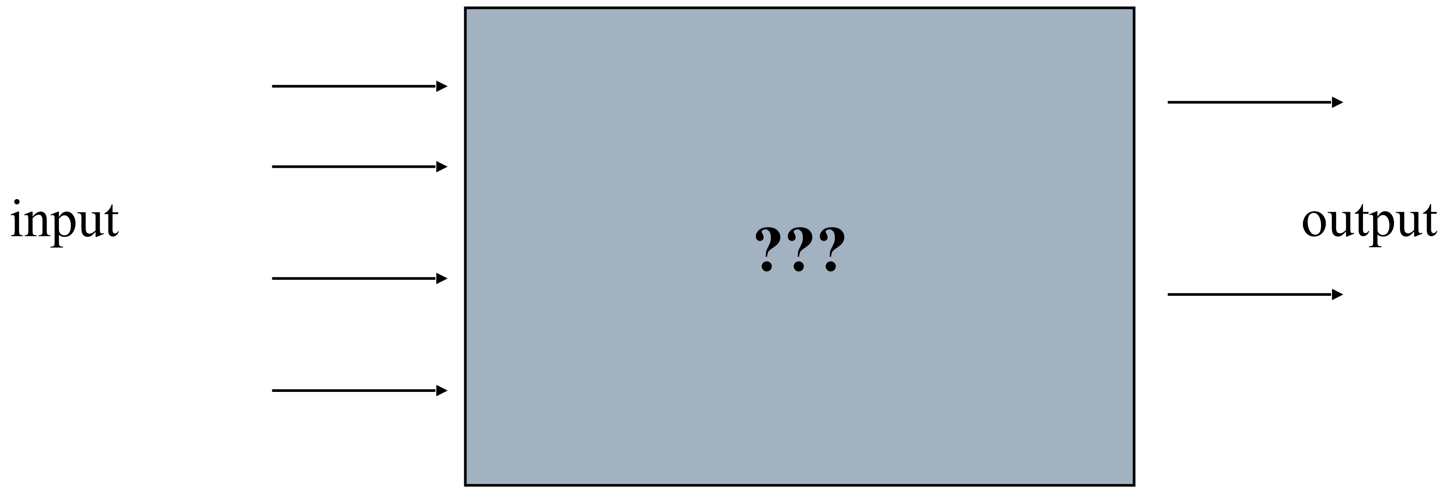
- **unit**: testing of a local component (function, module), typically only one step of the underlying state.
  (In functional programs, thats essentially all what you have to do!)
- **sequence**: testing of a local component (function, module), but typicallY sequences of executions, which typically depend on internal state
- **reactive sequence**: testing components by sequences of steps, but these sequences represent communication where later parts in the seqience depend on what has been earlier cummunicated

Taxonomy: Functional / Structural Test

- **functional**: (also: black-box tests). Tests were generated on a specification of the component, the test focusses on input output behaviour.
- **structural**: (also: white-box tests). Tests were generated on the basis of the structure or the program, i.e. using control-flow, data-flow paths or by using symbolic executions.
- **both**: (also: grey-box testing).

Functional ("Black-box") Unit Test

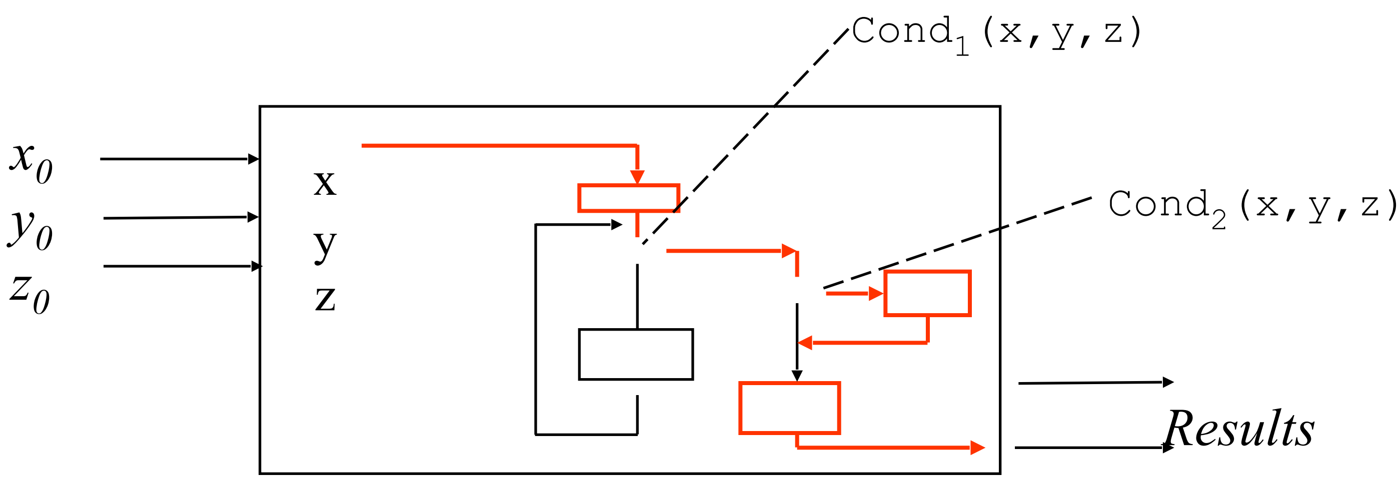
- We got the spec, but not the program, which is considered a black box:



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.

Give a 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 of the analysis:

```
inv  0<a ∧ 0<b ∧ 0<c
inv  c≤a+b ∧ a≤b+c ∧ b≤c+a
```

Triangles

```
a, b, c: Integer
```

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

```
operation t.is_Triangle():
  post  t.a=t.b ∧ t.b=t.c ⟶ result=equ
  post (t.a≠t.b ∨ t.b≠t.c) ∧
        (t.a=t.b ∨ t.b=t.c ∨ t.a=t.c))⟶ result=iso
  post (t.a≠t.b ∨ t.b≠t.c ∨ t.a≠t.c))⟶ 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 *)
- $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
- $\neg(\neg a) = a$
- $a \wedge b = b \wedge a$; $a \vee b = b \vee a$
- $a \wedge (b \wedge c) = (a \wedge b) \wedge c$
- $a \vee (b \vee c) = (a \vee b) \vee c$
- $a \rightarrow b = (\neg a) \vee b$
- $(a=b \wedge P(a)) = P(b)$ (* one point rule *)

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

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?

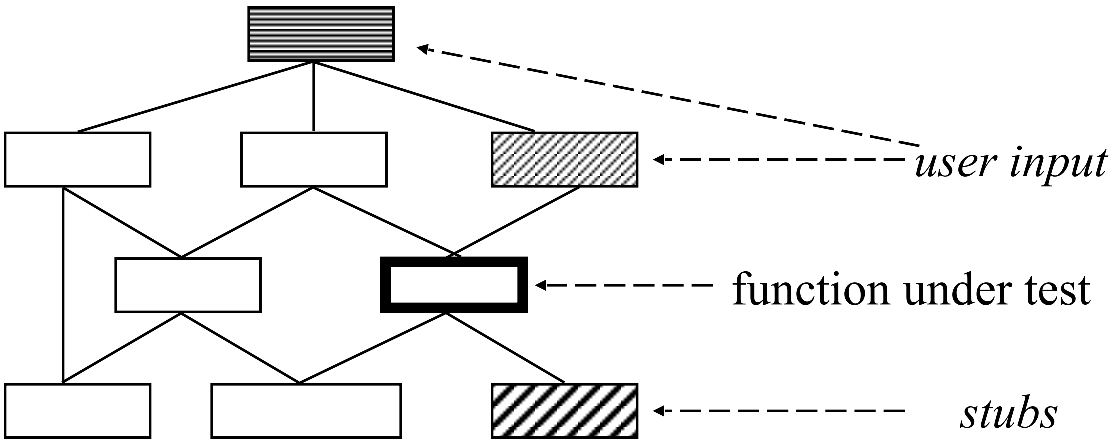
Moreover, compile:

$$
\begin{array}{lll}
\textbf{context} & \multicolumn{2}{l}{\texttt{C::m}(a_1:C_1,\ldots,a_n:C_n)} \\
\textbf{pre} & : & P(\mathit{self},a_1,\ldots,a_n) \\
\textbf{post} & : & Q(\mathit{self},a_1,\ldots,a_n,\mathit{result})
\end{array}
$$

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

$$
\begin{array}{l}
\texttt{check\_C}(); \ \texttt{check\_C}_1(); \ \ldots \ ; \ \texttt{check\_C}_n(); \\
\texttt{assert}(P(\mathit{self},a_1,\ldots,a_n)); \\
\mathit{result}=\texttt{run\_m}(\mathit{self},a_1,\ldots,a_n); \\
\texttt{assert}(Q(\mathit{self},a_1,\ldots,a_n,\mathit{result}));
\end{array}
$$

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 ...*

Conclusion: Functional Dynamic Unit Test : Problems

- Thus, any violation of an invariant, a pre-condition or a post-condition is detected.

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

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

- Inefficiencies can be partly overcome by optimized compilations.

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 does not increase necessarily by the number of test-cases !

  - Automation ? Tools ?