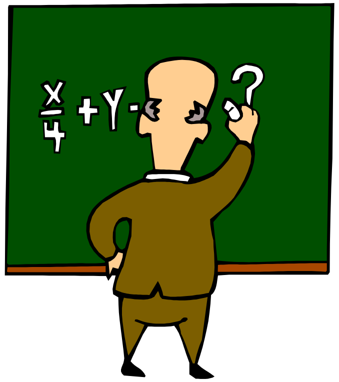
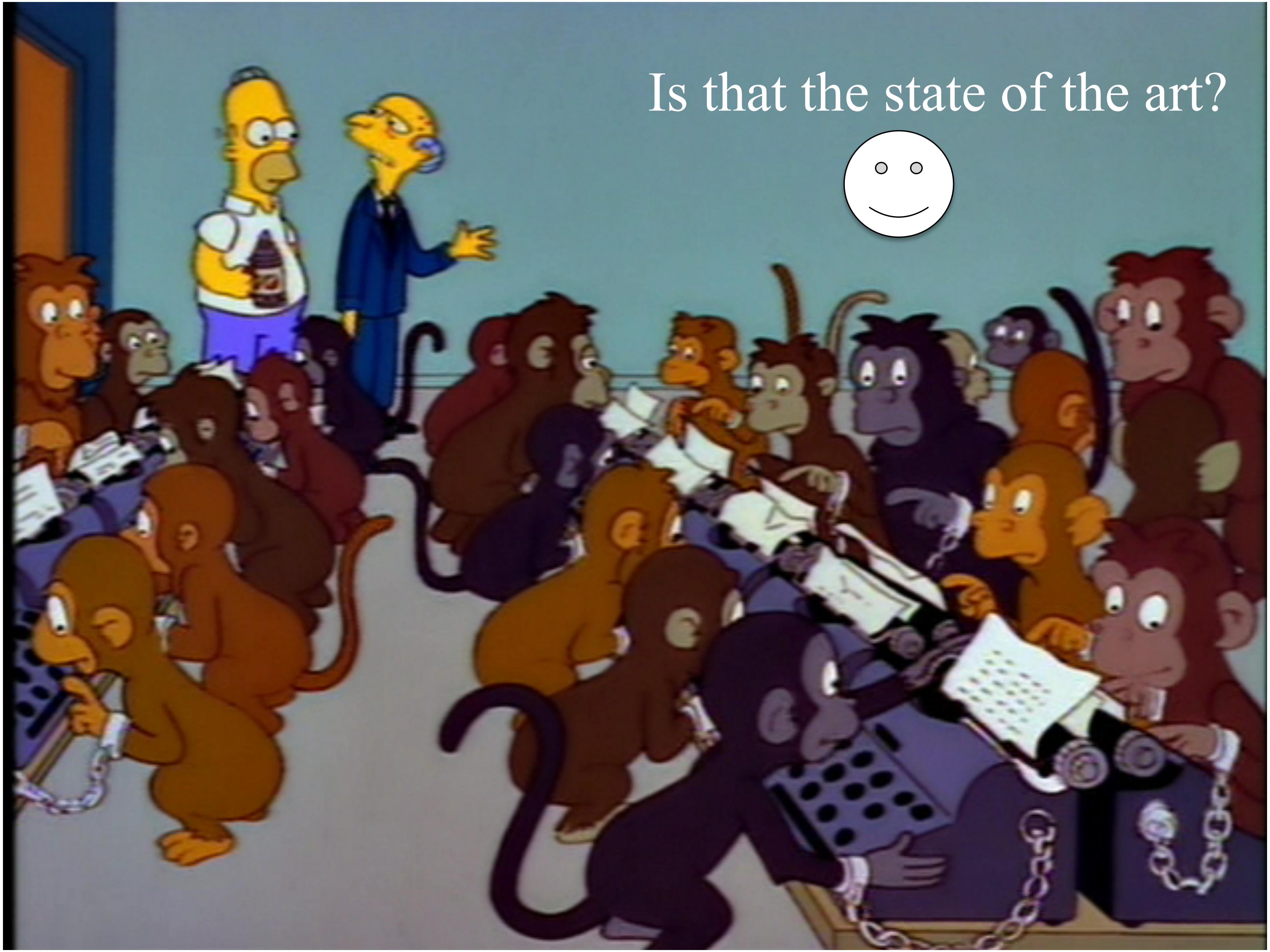
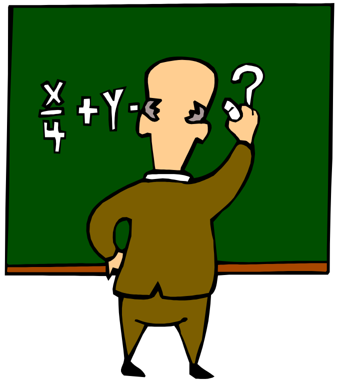


How can Formal Specifications benefit to Software Testing?

Marie-Claude Gaudel
Emeritus Professor
LRI, Univ Paris-Sud & CNRS


Is that the state of the art?

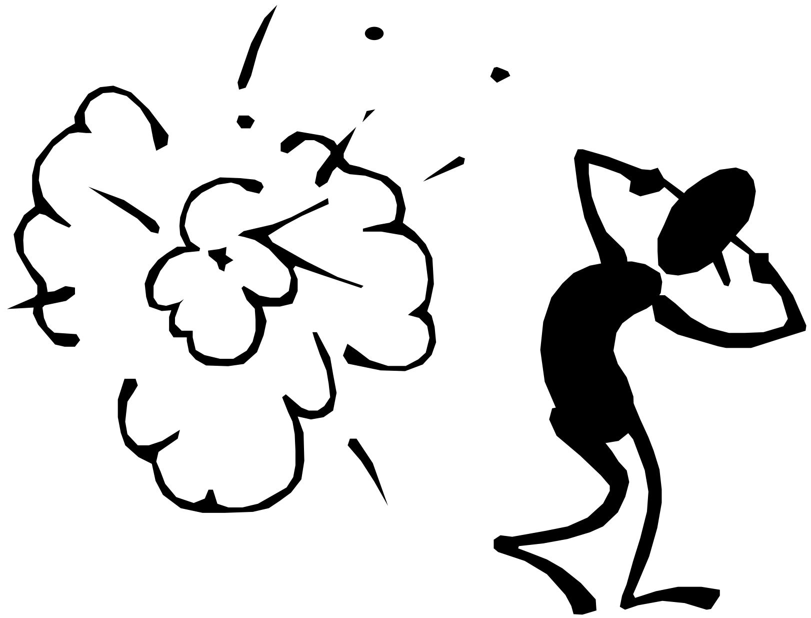


The long quest of a theory of software testing…

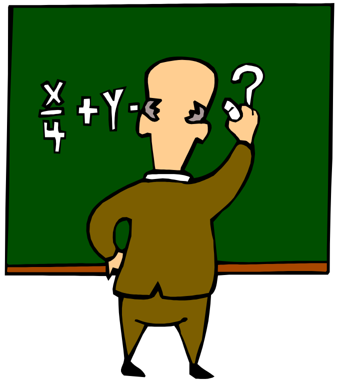
A pioneering paper:

- *« We know less about the theory of testing, which we do often, than about the theory of program proving, which we do seldom »*

Goodenough J. B., Gerhart S.,
IEEE Transactions on Software Engineering, 1975



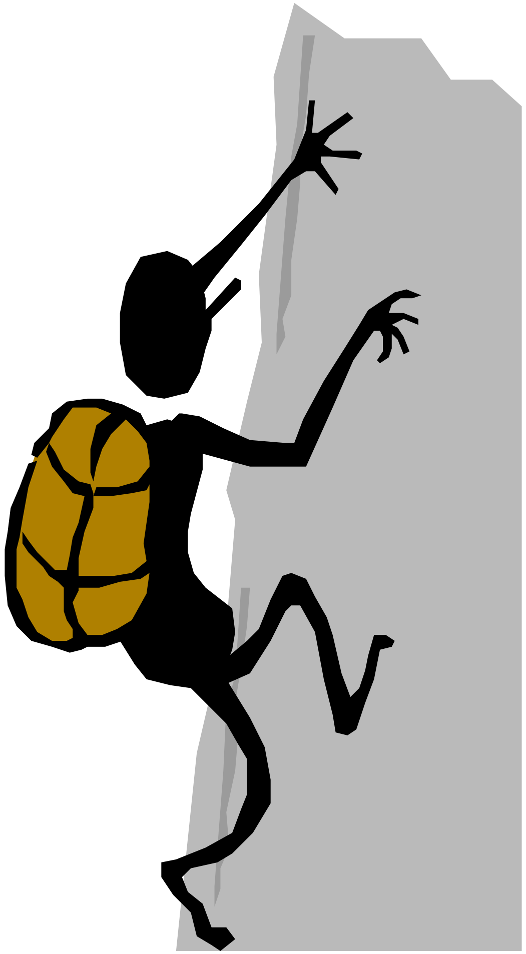
And then many others…

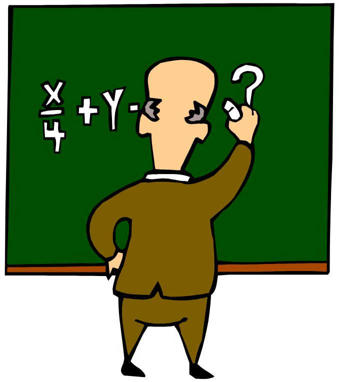


In this talk: formal methods and software testing

Outline of the talk:

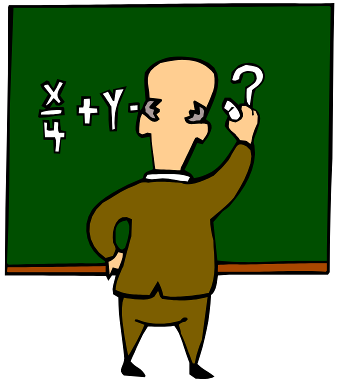
- *Generalities on specification-based testing (or model-based testing)*
- Specificities of formal specifications w.r.t. testing
- *Bridging the gap between testing and formalities:*
  - Testing hypotheses
  - Exploiting testing hypotheses



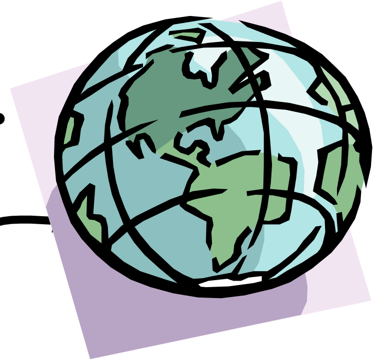


INTRODUCTION PART

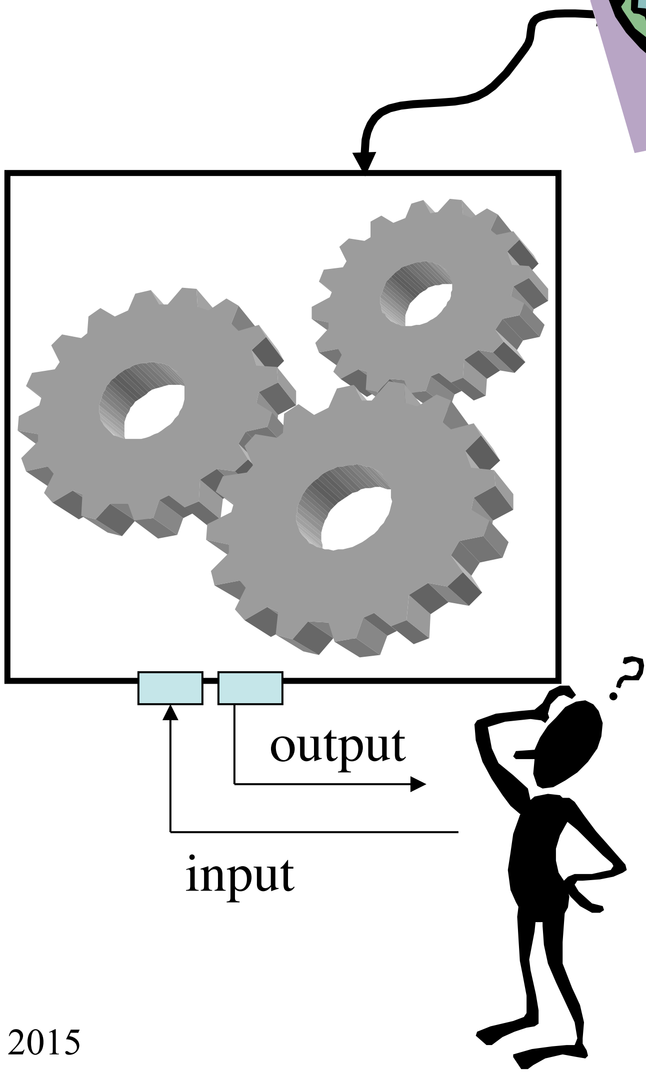
Preliminary considerations on specification-based testing

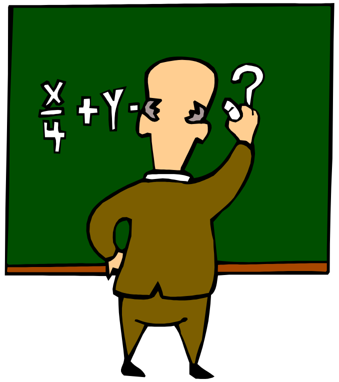


A few words on testing…

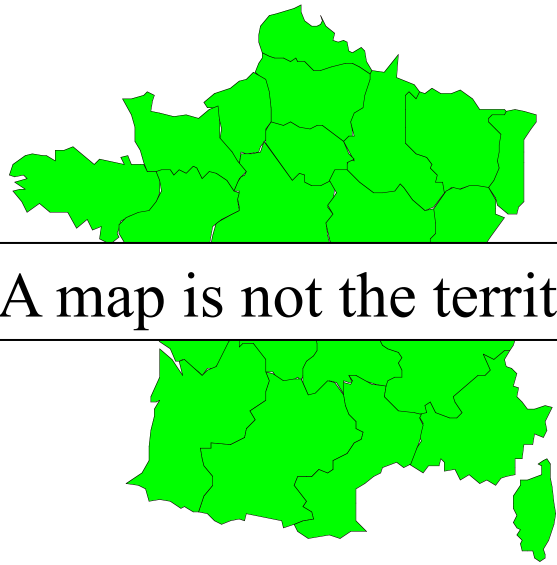


- One tests SYSTEMS
- A system is a dynamic entity, *embedded in the physical world*
- It is *observable* via some limited interface/procedure
- It is not always *controllable*
- It is quite different from a *piece of text* (formula, program) or a *diagram*





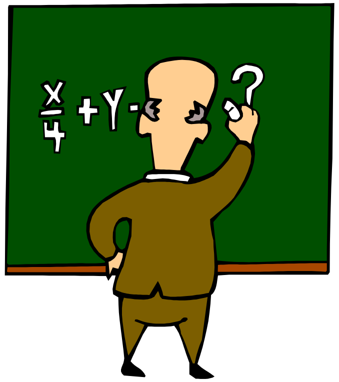
A philosophical interlude



"A map is not the territory"* Korzybski

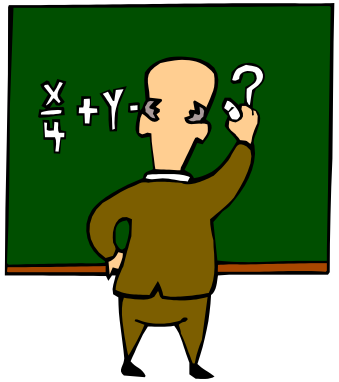
*A variant: "don't eat the menu…" ☺

A program text, or a specification text,
or a model, is not the system



Specification-based Testing

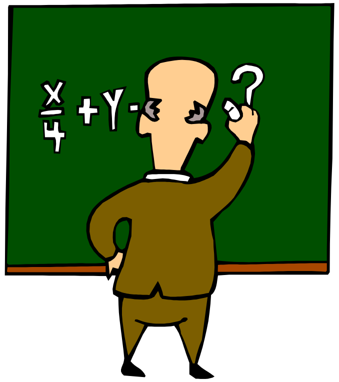
- The internal organisation of the SUT (System Under Test) is not considered
- There is some specified requirement expressed as a text, formula, diagram,…
- The aim is to detect deviations of the SUT w.r.t. the specified requirement



Specification-based Testing: underlying hypotheses

- The internal organisation of the SUT (System Under Test) is not considered, indeed…
- *However,*
  - Implicitly or explicitly, one considers a class of "testable implementations" =>
  - Notion of ***Testability Hypotheses*** on the SUT

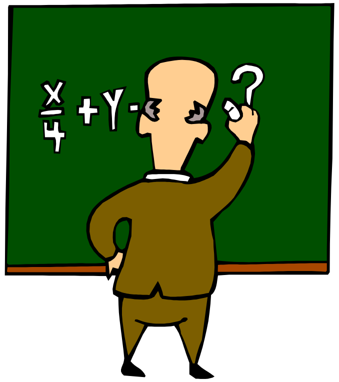
Often implicit, but always there!



Testability?

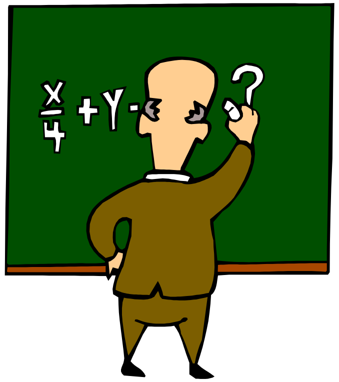
SUT?

- If the SUT can be *any demonic system*, there is no sensible way of testing it ☹
- Fortunately, *some basic assumptions are feasible* (example: correct implementation of booleans and bounded integers, determinism, …)
- Some others can be *verified in another way*: static checks on the program, preliminary tests, a priori knowledge of the environment…

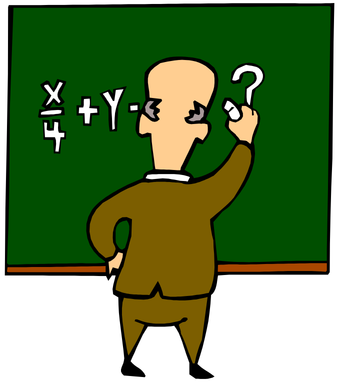


Specification-based testing:
for what sort of faults?

- Are the properties expressed by the specification satisfied?
- One tests the SUT against what is expressed by the specification.
- Strongly dependent on the kind of specification/model considered

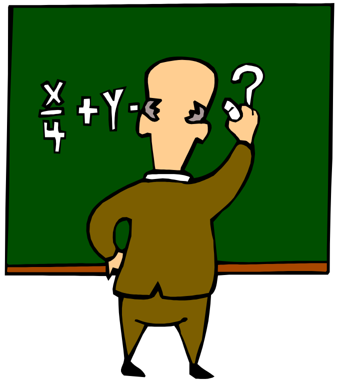


FORMAL SPECIFICATIONS AND TESTING



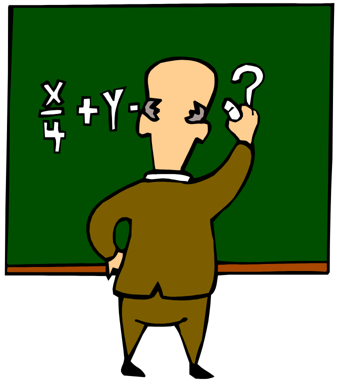
Formal Specifications?

- As for any specification framework, there is a notation:
  - Formulas
    - Pre/Post-conditions, 1st order logic, JML, SPEC# …
    - Algebraic Spec (CASL), Z, VDM, B,
  - Processes definitions
    - CSP, CCS, Lotos, Circus …
  - Annotated diagrams
    - Automata, Finite State Machines (FSM), Petri Nets…
- But there is more than a syntax…



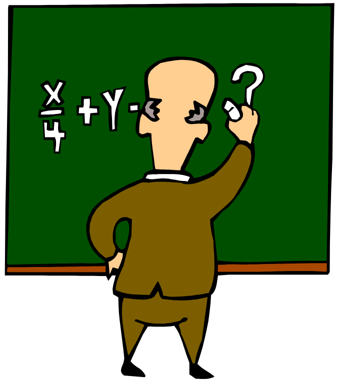
What makes a specification method formal?

- *There is a formal semantics*
  - Algebras, Predicate transformers, Sets, Labelled Transition Systems (LTS), Traces and Failures…
- There is a *formal system* (proofs) or a *verification method* (model-checking), or both.
- *Thus*
  - *Formal specifications can be analysed to guide the identification of appropriate test cases.*
  - *They may contribute to the definition of oracles.*



Relations between formal specifications

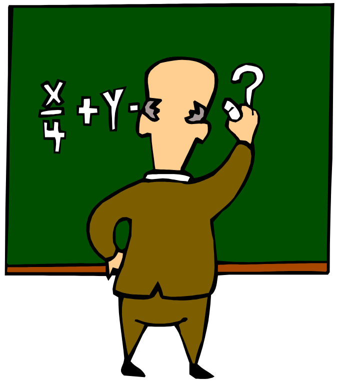
- In addition to syntax, semantics, deduction system, formal specifications come with notions of
  - *Equivalences (behavioural, observational,…)*
  - *Refinements*
  - *Conformance*
  - *Satisfaction*
- That are essential for testing
- That are semantically or/and logically defined



Required: a satisfaction/ conformance relation

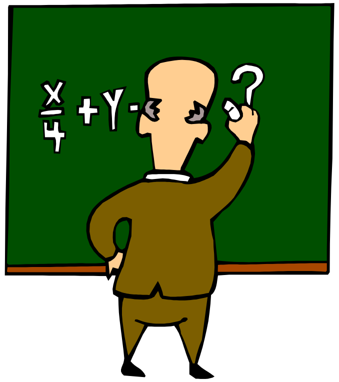
?

- Given some "testable" *SUT*, what does it mean that it satisfies *SP*?
- What is the correctness reference? Is there an "exhaustive" (or "complete") set of tests?
- *SP* is some sort of *model or formula*; *SUT* is some sort of *system*; how to define *"SUT sat SP"* or *"SUT conf SP"* in such an heterogeneous context?

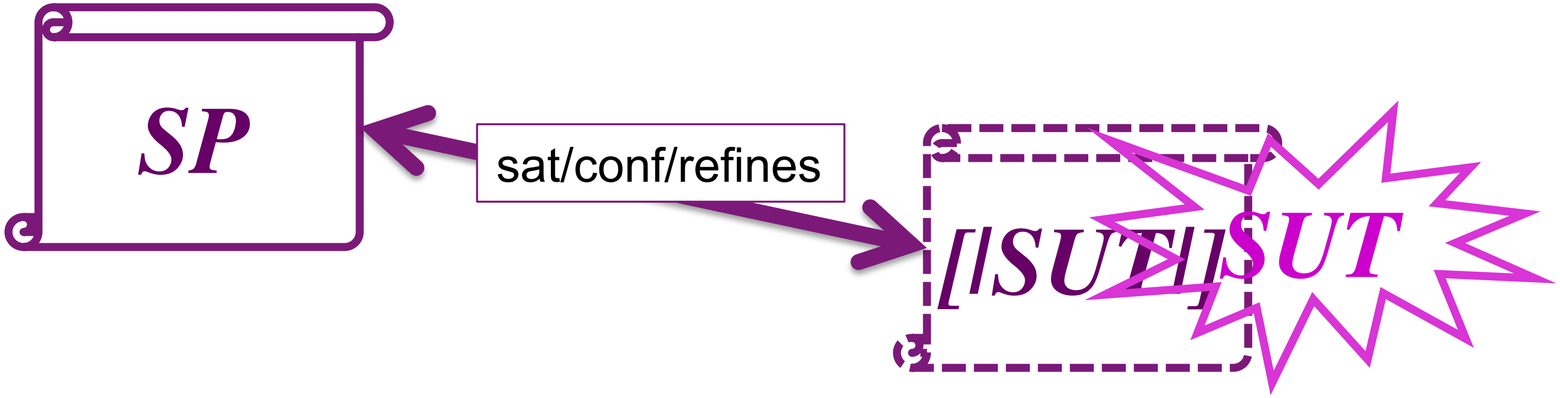


A generic testability hypothesis

- *"The SUT corresponds to some unknown formal specification in the same formalism as specification SP"*
  - If *SP* is a *FSM*, *SUT* behaves like some *FSM*
  - If *SP* is a formula, the symbols of the formula can be interpreted/computed by *SUT*
  - If *SP* is a process, *SUT* can be observed as a process, with traces and deadlocks
- Notation: *⟦SUT⟧*

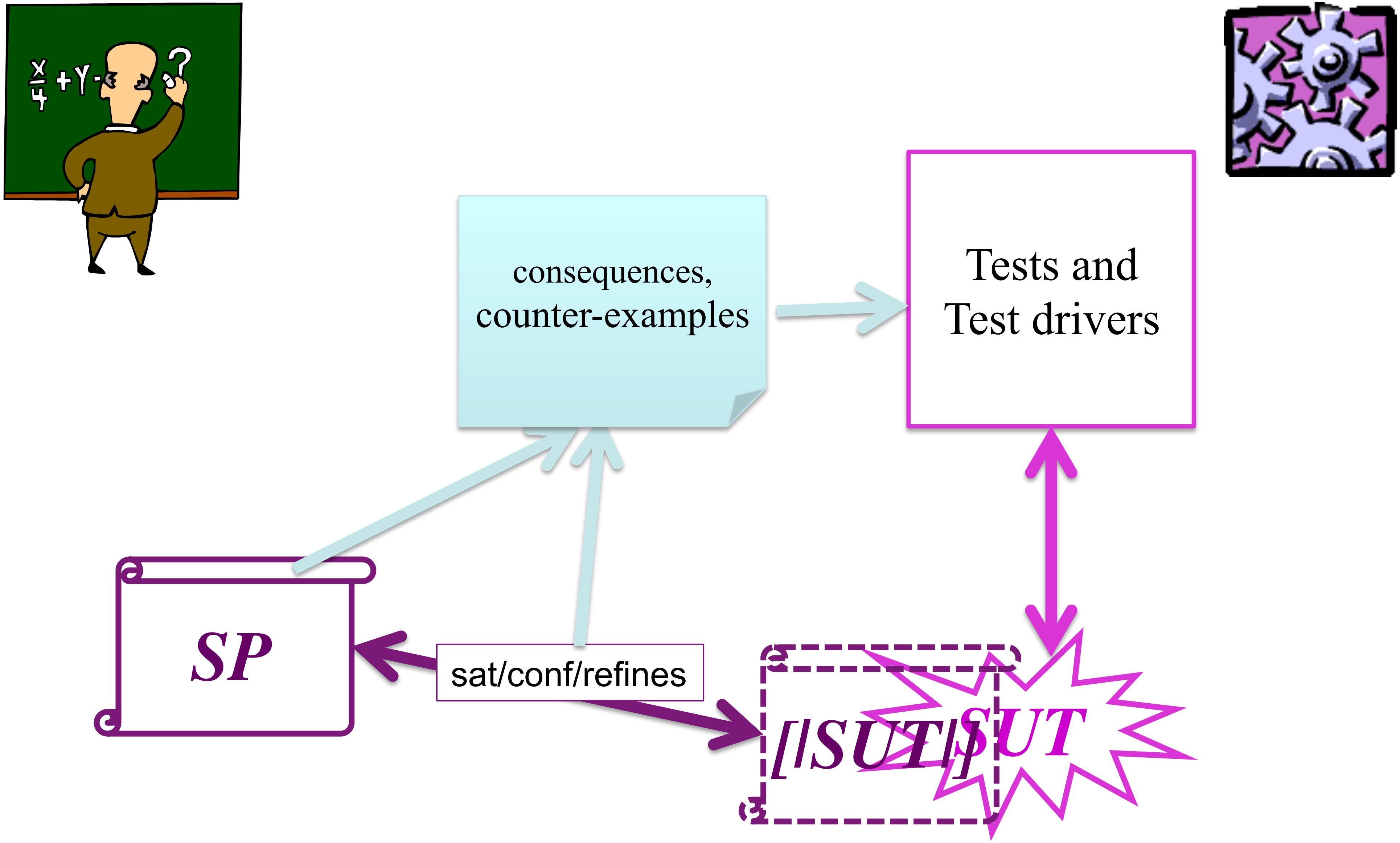


Back to well-established relations



For instance, the *satisfaction/conformance* relation is

- equivalence for FSM,
- logical satisfaction for formulas,
- Traces refinement, deadlock reduction (*conf*) for processes,
- *ioco* for LTS…


consequences,
counter-examples
Tests and
Test drivers
SP
sat/conf/refines
[|SUT|]
SUT

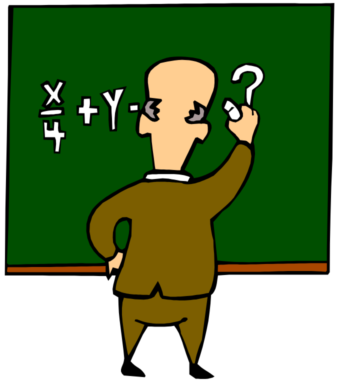


Illustration: testing against *traces refinement* in CSP

$$Counter_2 = add \rightarrow C_1$$
$$C_1 = add \rightarrow C_2 \,[\,]\, sub \rightarrow Counter_2$$
$$C_2 = sub \rightarrow C_1$$

Traces of $Counter_2$

<>

<*add*>

<*add,add*>

<*add,sub*>

<*add,add,sub*>

…

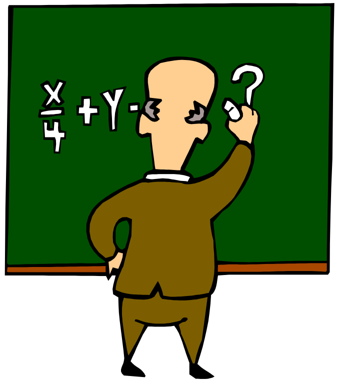


Illustration: testing against traces refinement in CSP

$$Counter_2 = add \rightarrow C_1$$
$$C_1 = add \rightarrow C_2 \,[\;]\, sub \rightarrow Counter_2$$
$$C_2 = sub \rightarrow C_1$$

Traces of $Counter_2$

<>

<*add*>

<*add,add*>

<*add,sub*>

<*add,add,sub*>

…

Forbidden traces

<*sub*>

<*add,add,add*>

<*add,sub,sub*>

…

$$test1 = pass \rightarrow sub \rightarrow fail \rightarrow STOP$$
$$test2 = inc \rightarrow add \rightarrow inc \rightarrow add \rightarrow pass \rightarrow add \rightarrow fail \rightarrow STOP$$
$$test3 = inc \rightarrow add \rightarrow inc \rightarrow sub \rightarrow pass \rightarrow sub \rightarrow fail \rightarrow STOP$$

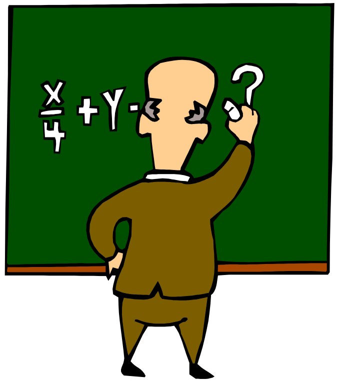


Illustration: testing against traces refinement in CSP

$$Counter_2 = add \rightarrow C_1$$
$$C_1 = add \rightarrow C_2 [\,] sub \rightarrow Counter_2$$
$$C_2 = sub \rightarrow C_1$$

Traces of $Counter_2$

<>

<*add*>

<*add,add*>

<*add,sub*>

<*add,add,sub*>

…

Forbidden traces

<*sub*>

<*add,add,add*>

<*add,sub,sub*>

…

$$test1 = pass \rightarrow sub \rightarrow fail \rightarrow STOP$$
$$test2 = inc \rightarrow add \rightarrow inc \rightarrow add \rightarrow pass \rightarrow add \rightarrow fail \rightarrow STOP$$
$$test3 = inc \rightarrow add \rightarrow inc \rightarrow sub \rightarrow pass \rightarrow sub \rightarrow fail \rightarrow STOP$$

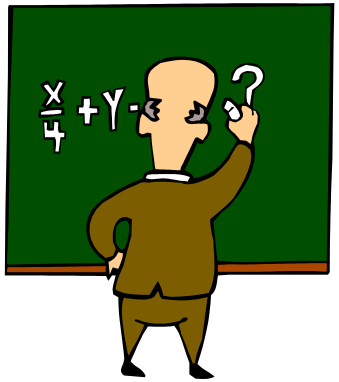
Test submissions

SUT |[*add,sub*]| *test1* \ [*add,sub*]

SUT |[*add,sub*]| *test2* \ [*add,sub*]

SUT |[*add,sub*]| *test3* \ [*add,sub*]

Oracle: the last observed event is not *fail*



Exhaustive test set for traces refinement of CSP

Let us consider the Test Set:

$\boldsymbol{Exhaust_T(SP) = \{T_T(s, a) \mid s \in traces(SP) \wedge \neg\, a \in initials(SP / s)\}}$

where

$T_T(s, a) = inc \rightarrow a_1 \rightarrow inc \rightarrow a_2 \rightarrow inc \ldots a_n \rightarrow pass \rightarrow a \rightarrow fail \rightarrow STOP$

for $s = \langle a_1, a_2, \ldots, a_n \rangle$.

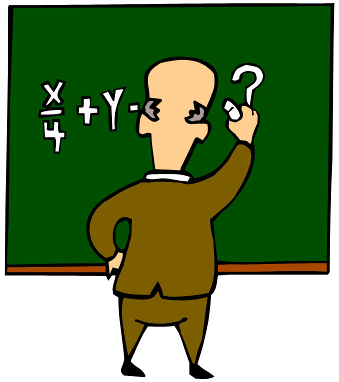
For any test T, its execution against SUT is specified as:

$$\boldsymbol{Execution_{SP,SUT}(T) = (SUT \; |[\, \alpha SP \,]| \; T) \backslash \alpha SP}$$

Theorem (Cavalcanti Gaudel 2007) :

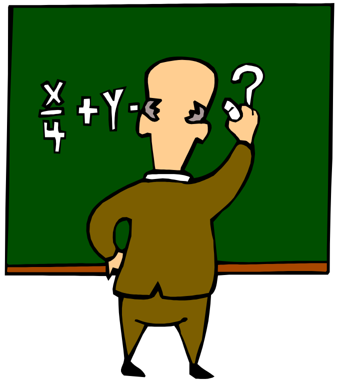
$[|SUT|]$ is a traces refinement of SP iff

$\boldsymbol{\forall\, T_T(s, a) \in Exhaust_T(SP), \quad \forall\, t \in traces(Execution_{SP,SUT}(T_T(s, a))),}$
$\boldsymbol{\neg\, last(t) = fail}$

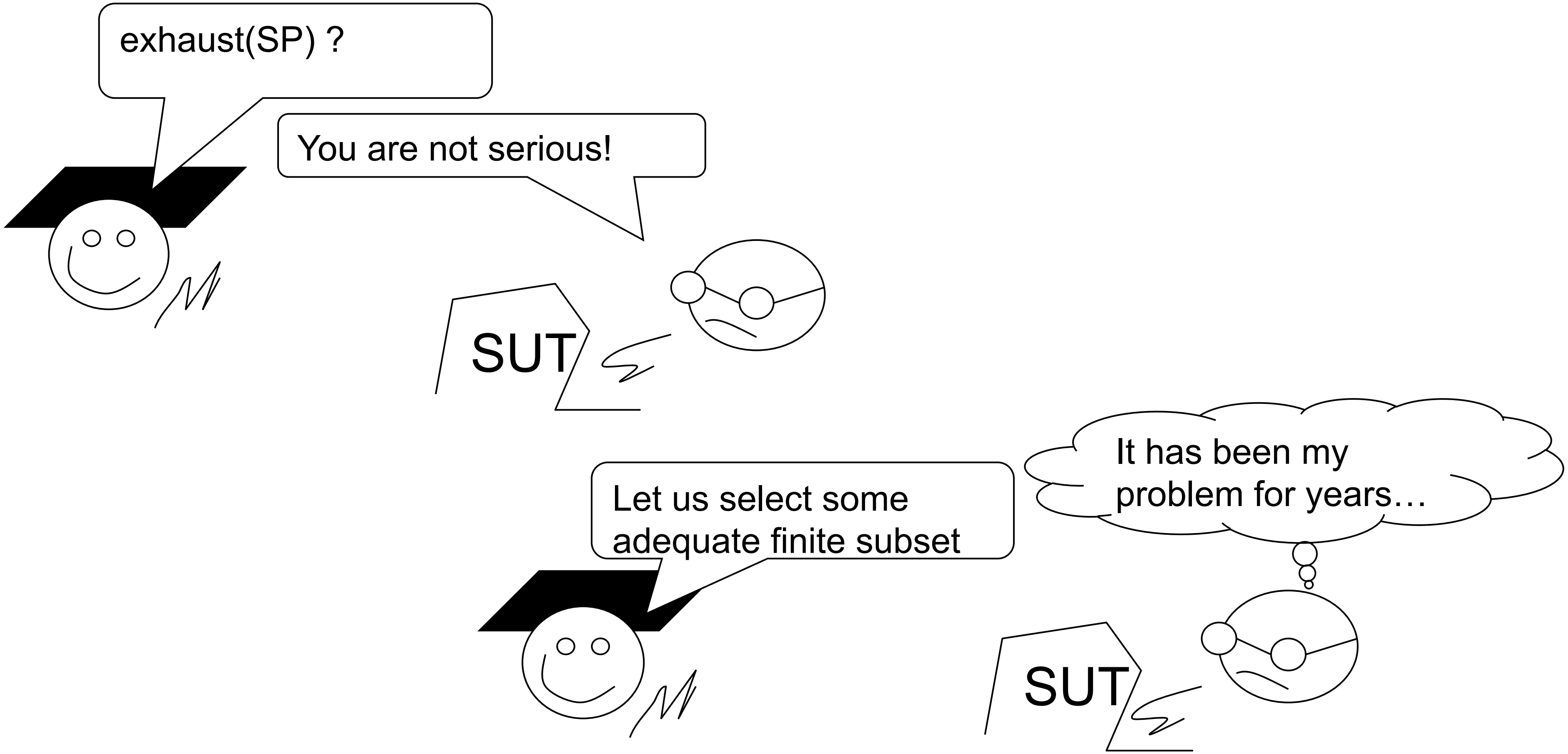


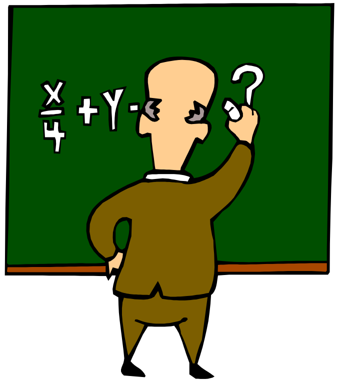
The corresponding testability hypotheses

- *SUT* behaves like a CSP process
  - With the same alphabet of actions as *SP*
  - The *actions and events are atomic*
- If *SUT* is non-determinist, it satisfies the classical *complete testing assumption*…
    - *(after a sufficient number of executions all the possible behaviours are covered)*
  - *Which can be ensured by some adequate scheduler/test driver (f.i. CHESS…)*



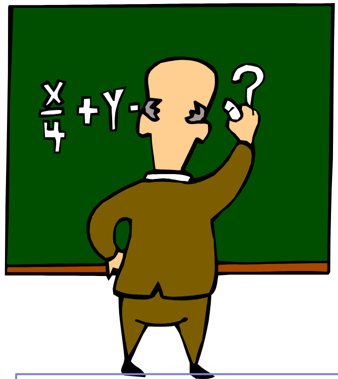
Its nice to have some theorems, but exhaustivity is not practicable…





Selection

- How to select finite subsets of $Exhaust_{SP}$?
- *Test Set Selection* is based on the specification (of course, it's Black Box Testing!)
- Among the solutions:
  - Uniformity hypotheses
  - Regularity hypotheses
  - Others …



Another example from CSP

$$Replicator = c?x:Int \rightarrow d!x \rightarrow Replicator$$

$$FreshInt(n:Int) = c!n \rightarrow FreshInt(n+1)$$

$(FreshInt(0)\,|[c]|\,Replicator)\setminus c$ parallel composition with hidden synchronisation on c

Traces of Replicator

<>

<c.0> <c.1> …

<c.0,d.0> <c.1,d.1>…

<c.0,d.0,c.7> …

…

Forbidden symbolic traces of Replicator

$<d.v>\ \forall\ v \in Int$

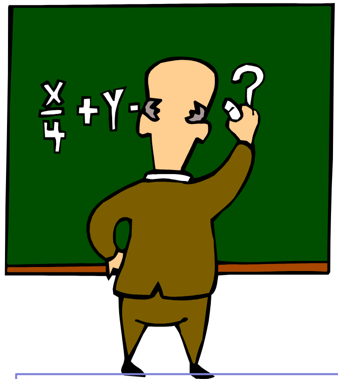
$<c.v,\ d.w>\ \forall\ v,w \in Int,\ v \neq w$

$<c.v,\ c.w>\ \forall\ v,w \in Int$

$<c.v,\ d.v,\ d.w>\ \forall\ v,w \in Int$

$<c.v,\ d.v,\ c.w,\ d.u>\ \forall\ v,w,u \in Int,\ w \neq u$

…



An example from CSP

$$Replicator = c?x:Int \rightarrow d!x \rightarrow Replicator$$

$$FreshInt(n:Int) = c!n \rightarrow FreshInt(n+1)$$

$(FreshInt(0) \,|[c]|\, Replicator) \setminus c$ parallel composition with hidden synchronisation on c

Traces of Replicator
<>
<c.0> <c.1>
<c.0,d.0> <c.1,d.1>
<c.0,d.0,c.7>
…

Forbidden symbolic traces

$<d.v> \ \forall\, v \in Int$

$<c.v, d.w> \ \forall\, v,w \in Int,\ v \neq w$

$<c.v, c.w> \ \forall\, v,w \in Int$

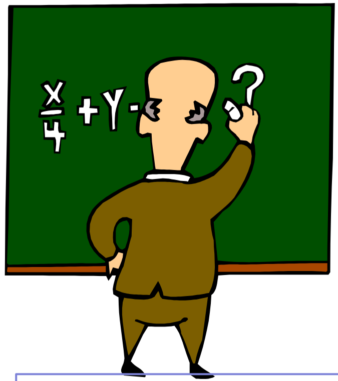
$<c.v, d.v, d.w> \ \forall\, v,w \in Int$

$<c.v, d.v, c.w, d.u> \forall\, v,w,u \in Int,\ w \neq u$

…

No condition on *v*: an arbitrary value will do => Uniformity Hypothesis

There is one condition on *w*: $v \neq w$. Any value satisfying it will do => Uniformity Hypothesis, etc



An example from CSP

$$Replicator = c?x:Int \rightarrow d!x \rightarrow Replicator$$

$$FreshInt(n:Int) = c!n \rightarrow FreshInt(n+1)$$

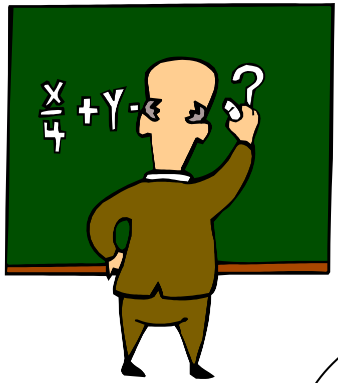
$(FreshInt(0)\,|[c]|\,Replicator)\setminus c$ parallel composition with hidden synchronisation on c

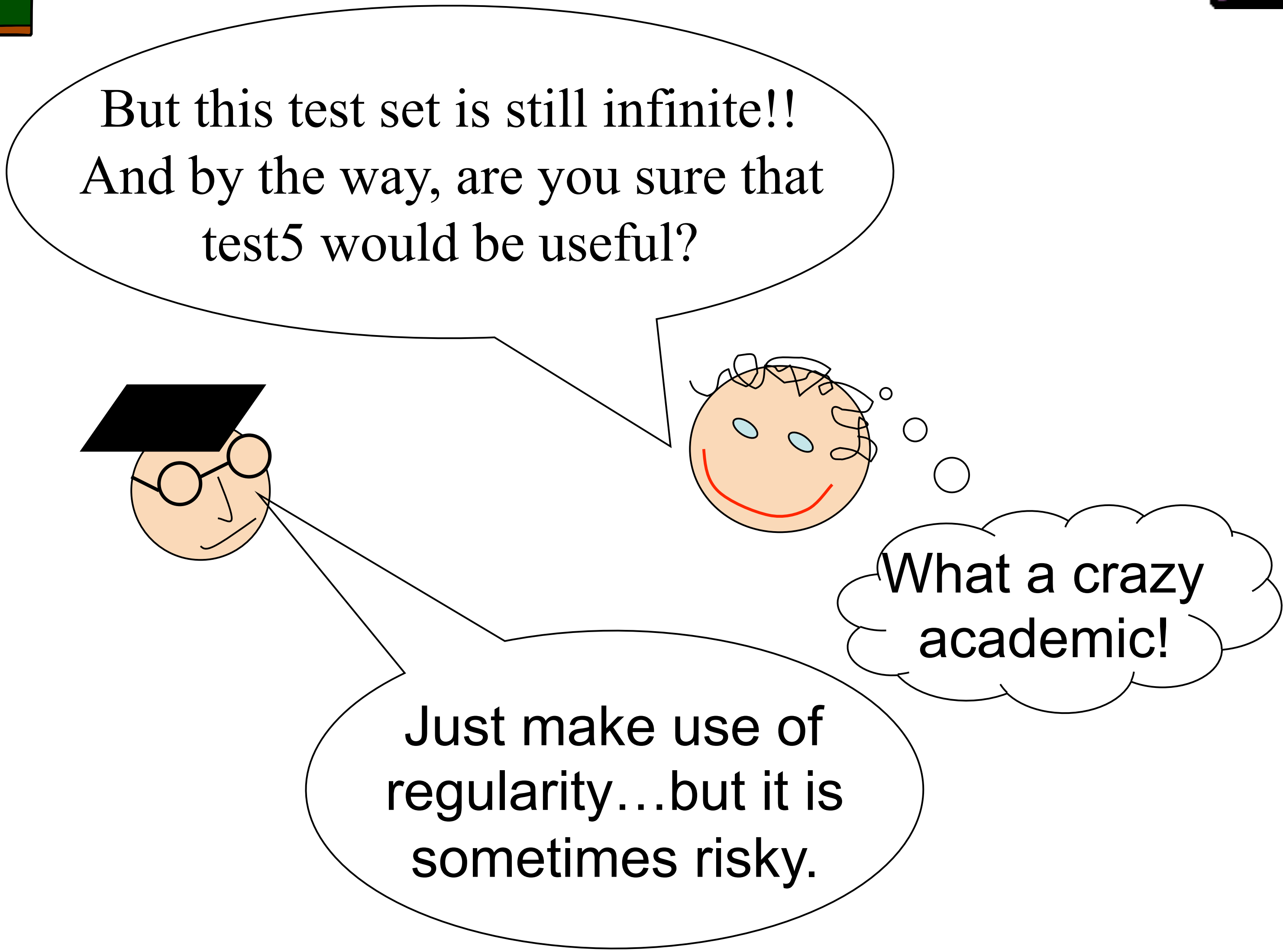
Traces of Replicator
<>
<c.0> <c.1>…
<c.0,d.0> <c.1,d.1>…
<c.0,d.0,c.7>…
…

Forbidden symbolic traces
<*d.v*> ∀ *v* ∈ *Int*
<*c.v, d.w*> ∀ *v,w* ∈ *Int, v≠w*
<*c.v, c.w*> ∀ *v,w* ∈ *Int*
<*c.v, d.v, d.w*> ∀ *v,w* ∈ *Int*
<*c.v, d.v, c.w, d.u*> ∀ *v,w,u* ∈ *Int, w≠u*
…

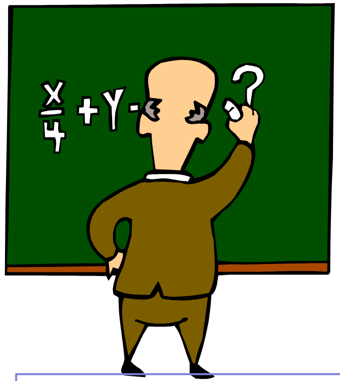
No condition on *v*: an arbitrary value will do => Uniformity Hypothesis => test1
There is one condition on *w*: *v≠w* .Any value satisfying it will do => Uniformity Hypothesis => test2, etc

$test1 = pass \rightarrow d.127 \rightarrow fail \rightarrow STOP$
$test2 = inc \rightarrow c.0 \rightarrow pass \rightarrow d.17 \rightarrow fail \rightarrow STOP$
$test3 = inc \rightarrow c.4 \rightarrow pass \rightarrow c.1024 \rightarrow fail \rightarrow STOP$
$test4 = inc \rightarrow c.78 \rightarrow inc \rightarrow d.78 \rightarrow pass \rightarrow d.46 \rightarrow fail \rightarrow STOP$
$test5 = \ldots$





But this test set is still infinite!! And by the way, are you sure that test5 would be useful?
What a crazy academic!
Just make use of regularity…but it is sometimes risky.



An example of regularity hypothesis

$$Replicator = c?x:Int \rightarrow d!x \rightarrow Replicator$$

$$FreshInt(n:Int) = c!n \rightarrow FreshInt(n+1)$$

$$(FreshInt(0) \,|[c]|\, Replicator) \setminus c$$ parallel composition

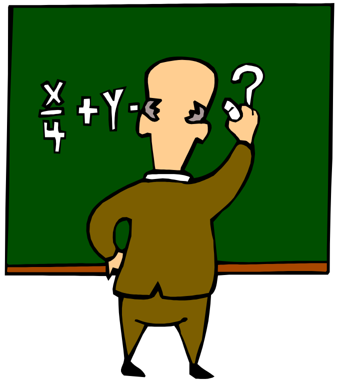
with hidden synchronisation on *c*

Traces of Replicator
<>
<c.0>
<c.1>…
<c.0,d.0>
<c.1,d.1>…
<c.0,d.0,c.7>…

Forbidden symbolic traces
<d.v> $\forall\ v \in Int$
<c.v, d.w> $\forall\ v,w \in Int,\ v \neq w$
<c.v, c.w> $\forall\ v,w \in Int$
<c.v, d.v, d.w> $\forall\ v,w \in Int$
<c.v, d.v, c.w, d.u> $\forall\ v,w,u \in Int,\ w \neq u$
…

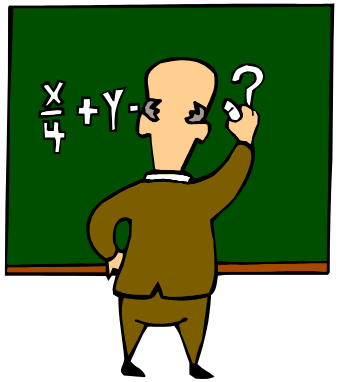
There is no dependency between the recursive calls of *Replicator.*
There is no shared state.
$\Rightarrow$ If the SUT is determinist, one execution is sufficient => Regularity Hypothesis =>
Finite Test Set

$$test1 = pass \rightarrow d.127 \rightarrow fail \rightarrow STOP$$
$$test2 = inc \rightarrow c.0 \rightarrow pass \rightarrow d.17 \rightarrow fail \rightarrow STOP$$
$$test3 = inc \rightarrow c.4 \rightarrow pass \rightarrow c.1024 \rightarrow fail \rightarrow STOP$$
$$test4 = inc \rightarrow c.78 \rightarrow inc \rightarrow d.78 \rightarrow pass \rightarrow d.46 \rightarrow fail \rightarrow STOP$$



Selection Hypotheses

- Addition to Testability Hypotheses: *Selection Hypotheses* on the SUT
- *Uniformity Hypothesis*
  - $\Phi(X)$ is a property, *SUT* is a system, *D* is a sub-domain of the domain of *X*
  - $(\forall t_0 \in D)\ (\,[\![SUT]\!]\ sat\ \Phi(t_0) \Rightarrow (\forall t \in D)\ ([\![SUT]\!] \models \Phi(t))\,)$
  - Determination of sub-domains ? *guided by the specification, see later…*
- *Regularity Hypothesis*
  - $(\ (\forall t \in Dom(X),\ |t| \leq k \Rightarrow [\![SUT]\!]\ sat\ \Phi(t)\ )) \Rightarrow$
    $(\forall t \in Dom(X)\ )\ ([\![SUT]\!]\ sat\ \Phi(t))$
  - Determination of |t|? *cf. specification*



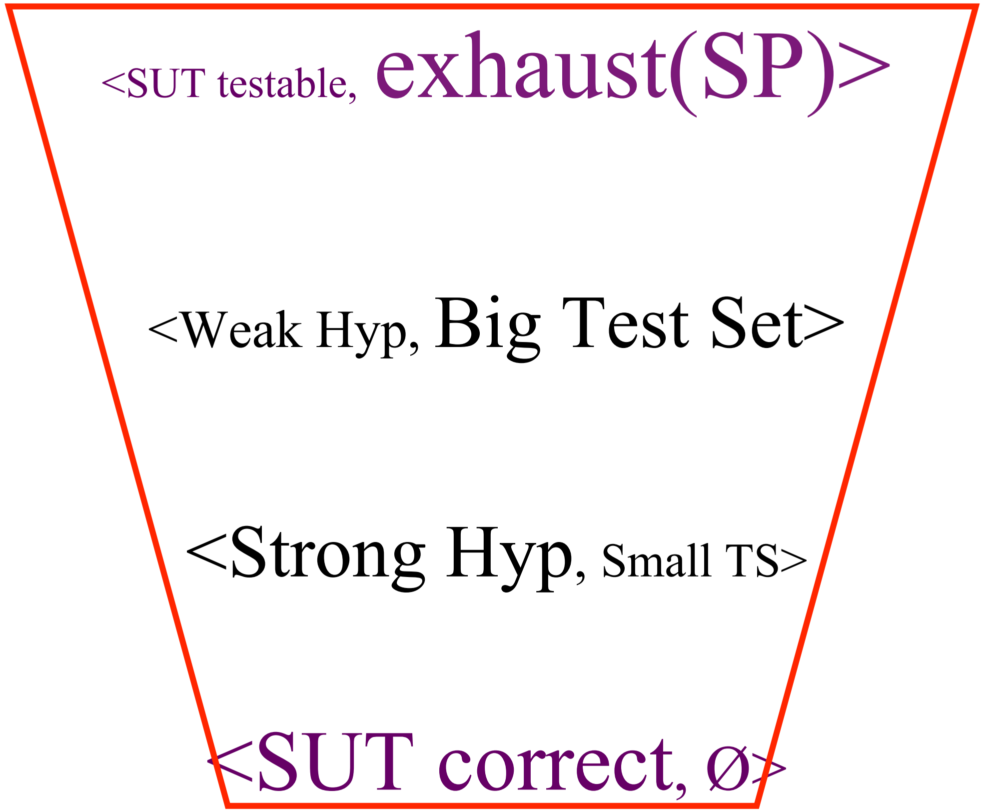
Selection of finite test sets

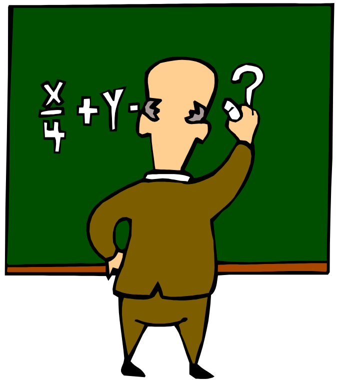
- “Selection Hypotheses” *H* on *SUT*, and construction of practicable test sets *T* such that:

H holds for SUT =>

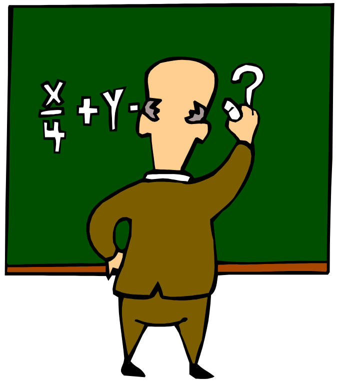
*(SUT **passes** T <=> [SUT] **sat** SP)*

- *<H, T>* is a valid and unbiased Test Context
- or: *T* is complete w.r.t. *H*





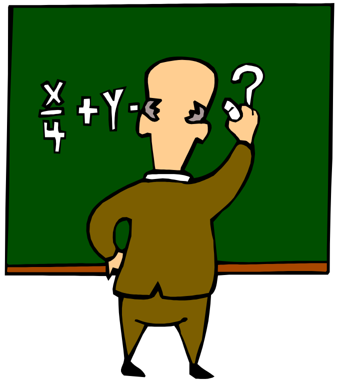
INVENTING AND EXPLOITING TESTING HYPOTHESES



"Invention" of selection hypotheses

Several possibilities:

- Guided by the conditions that appear in the specification : case analysis, case splitting
- Or guided by some knowledge of the operational environment
- Or guided by some fault model
- Or guided by the syntax (coverage criteria)



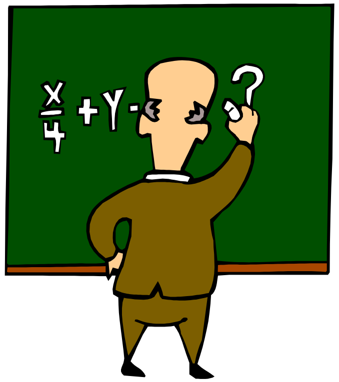
Case splitting

Two main techniques:

- Reduction of formulas into Disjunctive Normal Form (DNF) *[Dick & Faivre 1993]*
- Unfolding of recursive definitions *[Burstall & Darlington 1977]*

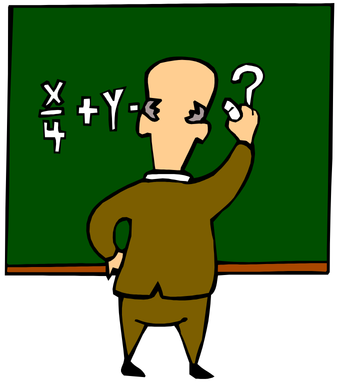
Implementations:

- Conditional rewriting, Narrowing
- Symbolic evaluation



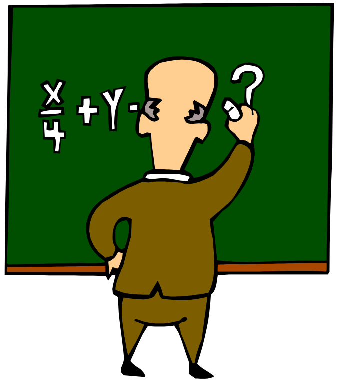
Non-termination of case splitting?

- Regularity hypotheses again, or
- Interpolation – Inference of invariants => use of proof assistants
- An advanced prototype: HOL-TestGen:
  - Developed by Brucker-Wolff-Brügger-Krieger
  - Test case generator for specification based unit testing
  - Built-on top of the HOL/Isabelle theorem proving environment



HOL-TestGen in a nutshell

- In HOL-TestGen you can:
  - write test specifications in Higher-order logics (HOL)
  - (semi-) automatically partition the input space, resulting in abstract test cases
  - automatically select concrete test data
  - automatically generate test scripts (in SML)
  - using a foreign language interface, implementations in arbitrary languages (e.g. C) can be tested.



How to use it? Step 1

- Writing a test-theory: *properties of the context*
- Example: Sorting in HOL

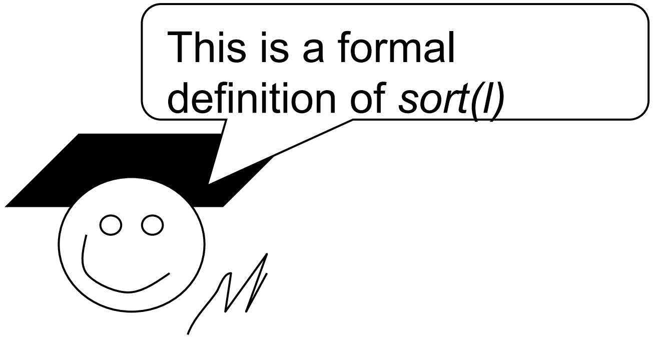
  - fun ins :: "('a::linorder) ⇒'a list ⇒ 'a list"

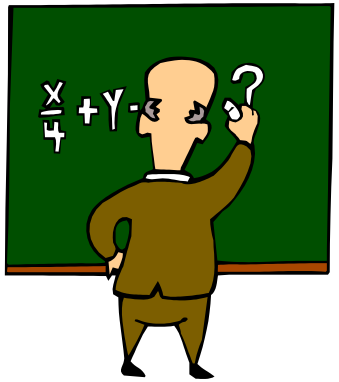
  where "ins x [] = [x] " |

  "ins x (y#ys) = (if (x < y) then x#y#ys else (y#(ins x ys)))"

  - fun sort:: "('a::linorder) list ⇒ 'a list"

  where "sort [] = [] " | "sort (x#xs) = ins x (sort xs)"

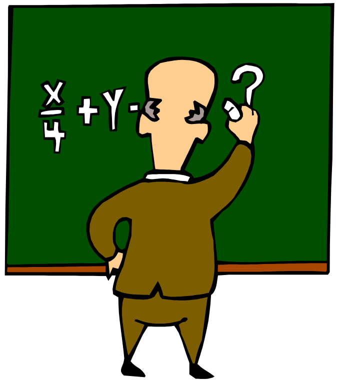




How to use it? Step 2

- Writing a test-theory: *properties of the context*
- Writing a test-specification TS: *what do you want to test?*
- Example:

test_spec "sort(l) = prog(l)"



How to use it? Step 3

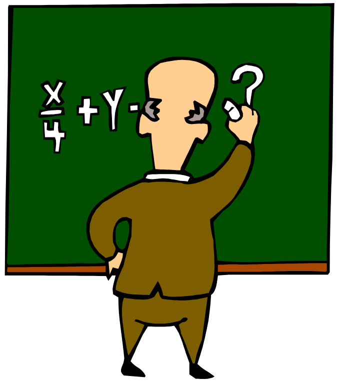
- Writing a test-theory: *properties of the context*
- Writing a test-specification TS: *what do you want to test?*
- Conversion into some test-theorem: *case-splitting (big parameterised test case generation macro)*

$TC_1 \Rightarrow \ldots \Rightarrow TC_n \Rightarrow THYP(H_1) \Rightarrow \ldots \Rightarrow THYP(H_m) \Rightarrow TS$

- where test cases TC_i have the form

$Constraint_1(x) \Rightarrow \ldots \Rightarrow Constraint_k(x) \Rightarrow P(prog\ x)$

- where $THYP(H_i)$ are test-hypotheses
- where TS is the Test Specification



How to use it? Step 3

- Writing a test-theory: *properties of the context*
- Writing a test-specification TS: *what do you want to test?*
- Conversion into some test-theorem: *case-splitting via some test case generation macro*

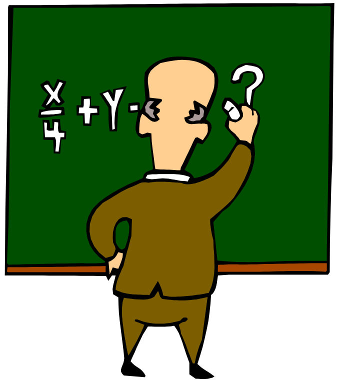
  Example : apply(gen_test_cases 3 1 “prog”) yields as constraints, i.e. as test cases

[] = prog([])

[?X1] = prog([?X1])

[?X1 ≤?X2] ⇒ [?X1, ?X2] = prog([?X1, ?X2])

[?X1 > ?X2] ⇒ [?X2, ?X1] = prog([?X1, ?X2])



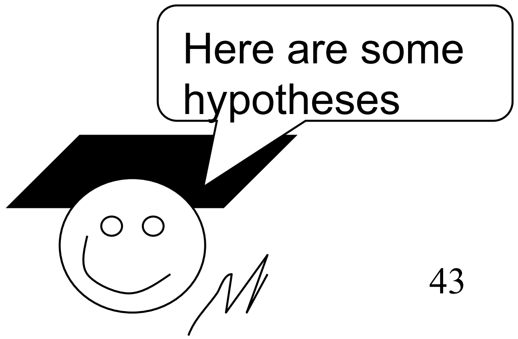
How to use it? Step 3

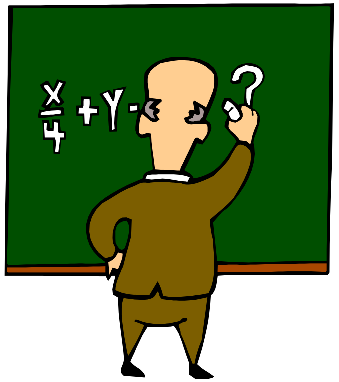
- Writing a test-theory: *properties of the context*
- Writing a test-specification TS: *what do you want to test?*
- Conversion into some test-theorem: *case-splitting via some test case generation macro*

  Example : apply(gen_test_cases 3 1 “prog”) yields among the hypotheses:

  – THYP($\exists$ x y. y < x $\rightarrow$ [y,x] = sort(PUT [x,y]) $\rightarrow$
  $\forall$ x y. y < x $\rightarrow$ [y,x] = sort(PUT [x,y]))

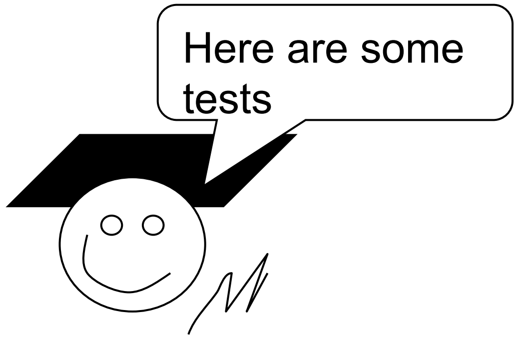
  – THYP(3 < |l| $\rightarrow$ is_sorted(SUT l))

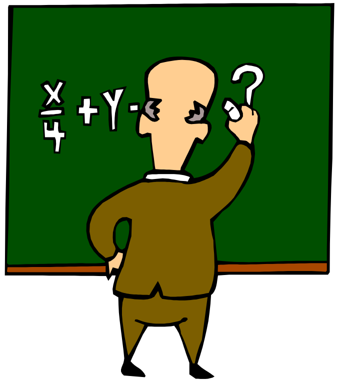




How to use it? Step 4

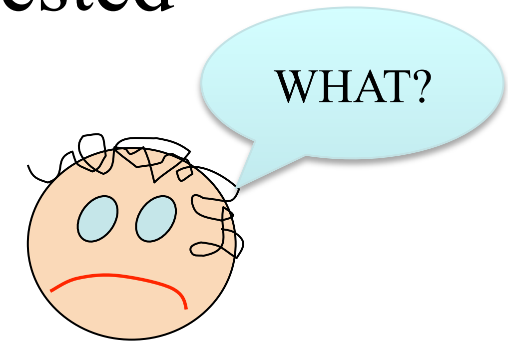
- Writing a test-theory: *properties of the context*
- Writing a test-specification TS: *what do you want to test?*
- Conversion into test-theorem: *case-splitting*
- Generation of test-data: using some SMT solver (Z3, Alt-Ergo)
  - [] = prog []
  - [3] = prog [3]
  - [6,8] = prog [6, 8]
  - [0,19] = prog [19, 0]

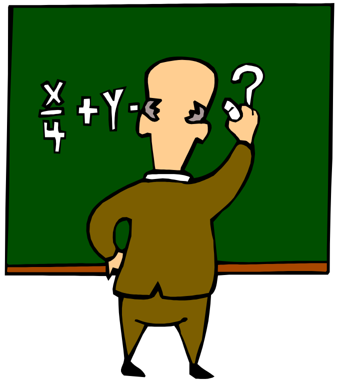




How to exploit the test-theorems?

- In addition to test data generation, hypotheses are useful:
- As static properties of the program, to be proved
- As new test specifications, to be tested
- As warning to the developers…





IT WAS MY CONCLUSION!
SOME QUESTIONS?