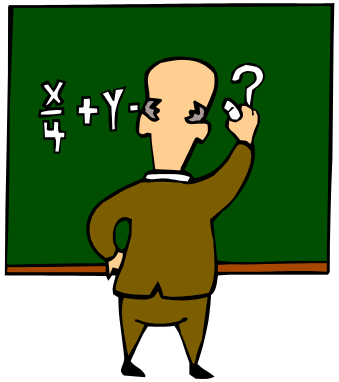
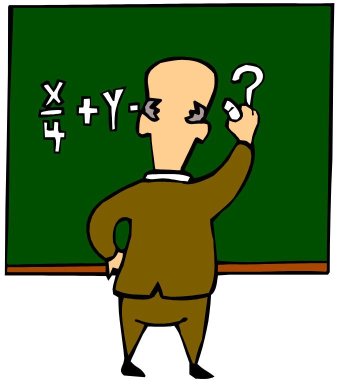


# Testing from Formal Specifications,
## *a unifying framework*

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



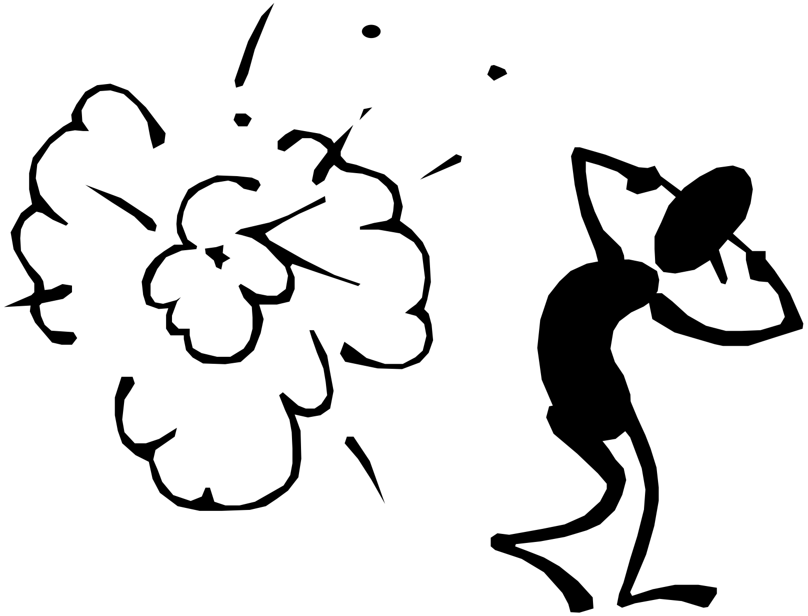
# Software Testing can be formal too

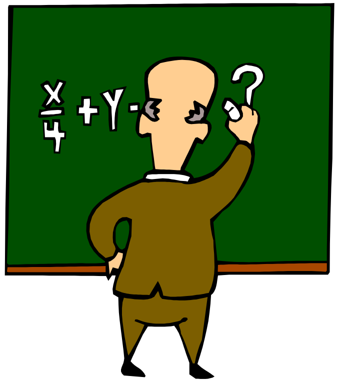
A pioneering paper:

- *« 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

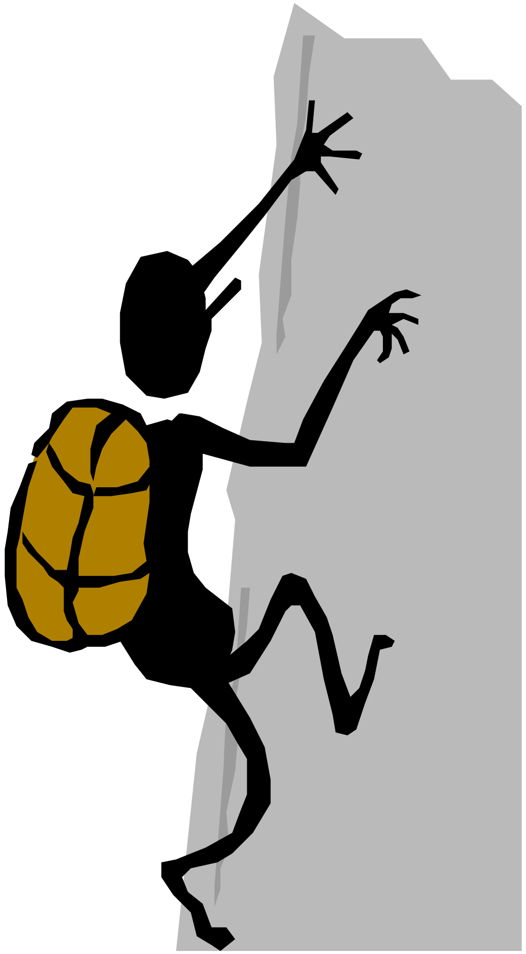


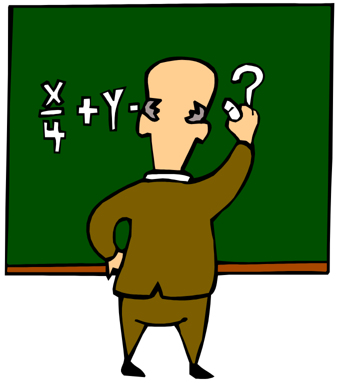


# There have been some progresses…

Outline of the course

- *Introduction Part*
  - Formal specifications
  - Testing
- Putting them together
- *Case splitting methods*
  - DNF, unfolding,…
- *Illustrations*
  - Axioms, FSM, CSP

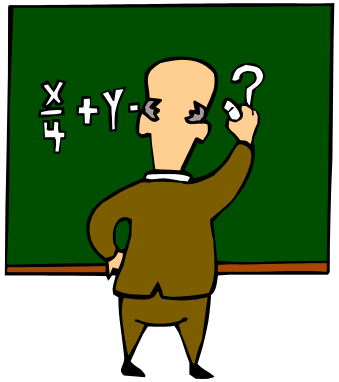




# INTRODUCTION PART

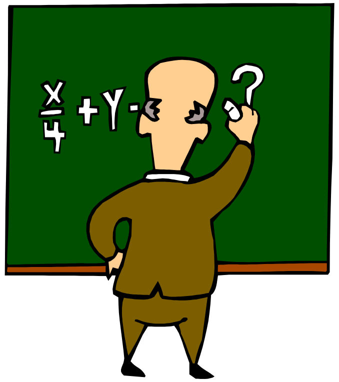
Formal specifications

Testing



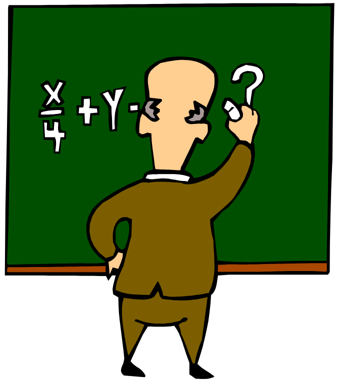
# 1 - Formal Specifications?

- As for any specification framework, there is a notation, for instance:
  - – 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.*



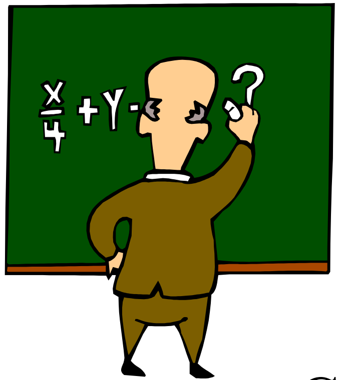
# Example 1:Pre/Post-conditions (à la VDM)

MAX (a:$\mathbb{Z}$, b:$\mathbb{Z}$)

**result** max:$\mathbb{Z}$

**pre** true

**post** $(max=a \lor max=b) \land max \geq a \land max \geq b$



# Example 1bis: axioms of a data type (à la CASL)

**spec** CONTAINER = NAT, BOOL

**then**

**generated type** *Container* ::= [] | _::_(*Nat* ; *Container*)

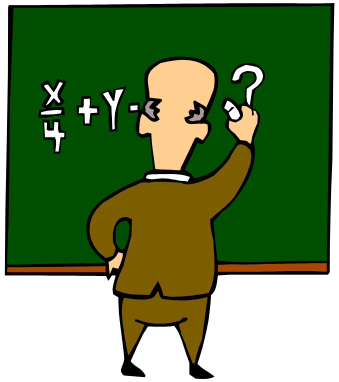
**op** *isin* : *Nat* × *Container* → *Bool*

**op** *remove*: *Nat* × *Container* → *Container*

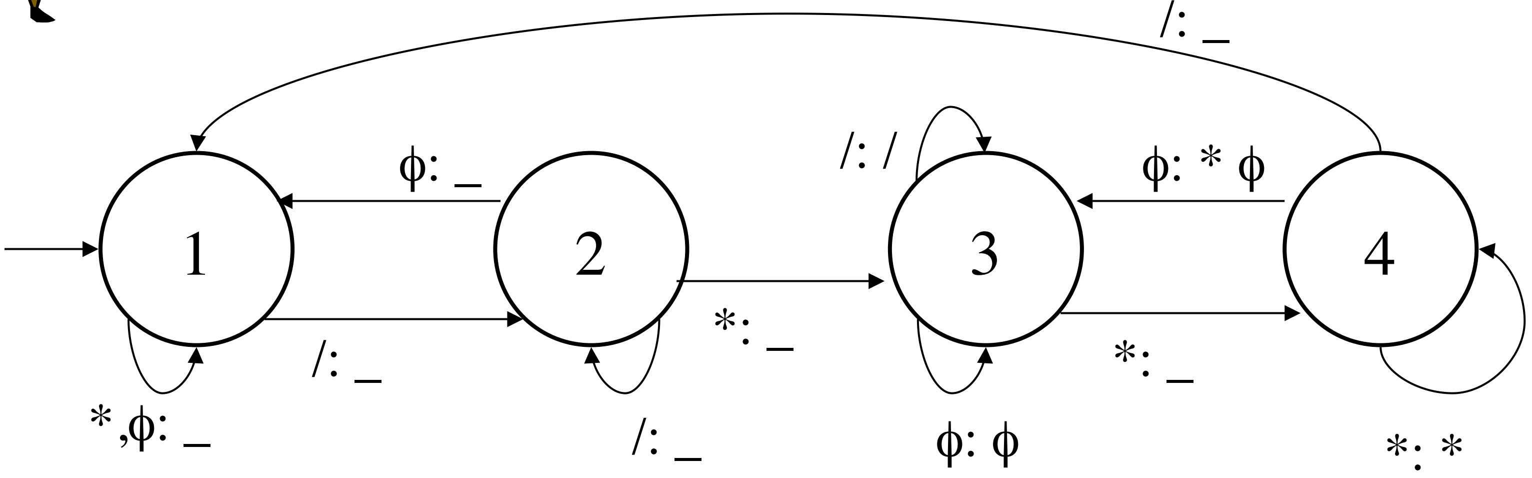
$\forall x, y{:}Nat;\ c{:}Container$

- $isin(x, []) = false$
- $eq(x, y) = true \Rightarrow isin(x, y{::}c) = true$
- $eq(x, y) = false \Rightarrow isin(x, y{::}c) = isin(x,c)$
- $remove(x, []) = []$
- $eq(x, y) = true \Rightarrow remove(x, y{::}c) = c$
- $eq(x, y) = false \Rightarrow remove(x, y{::}c) = y{::}remove(x,c)$

**end**



# Example 2: FSM



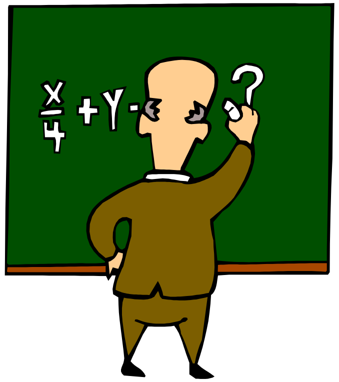
*This FSM removes from the input text all that is not a comment*

A comment is a string between /* and */

Examples:

This is not a comment /* all that / *is ** a comment */ this is no more a comment.

NB: ϕ is any character but * and /



# Example 3: CSP processes

$$Counter_2 = add \rightarrow C_1$$

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

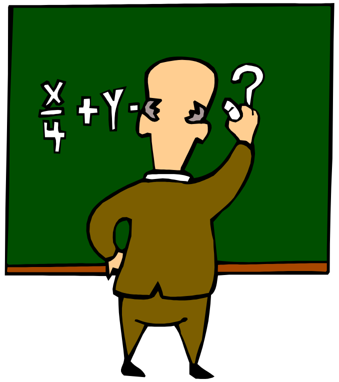
$$C_2 = sub \rightarrow C_1$$

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

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

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

with hidden synchronisation on $c$



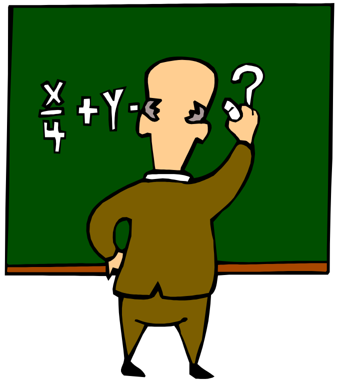
# Example 3bis: a *Circus* process

$$
\begin{array}{l}
\mathit{RANGE} == 0\,..\,59 \\
\mathbf{channel}\ \ \mathit{tick}, \mathit{time} \\
\mathbf{channel}\ \ \mathit{out} : \mathit{RANGE} \times \mathit{RANGE}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{process}\ \ \mathit{Chrono} \mathrel{\widehat=} \mathbf{begin} \\
\ \ \mathbf{state}\ \ \mathit{AState}\ \ ==\ \ [\, \mathit{sec}, \mathit{min} : \mathit{RANGE} \,] \\
\ \ \mathit{AInit}\ \ ==\ \ [\, \mathit{AState}\,' \mid \mathit{sec}' = \mathit{min}' \land \mathit{min}' = 0 \,] \\
\mathit{IncSec}\ \ ==\ \ [\, \Delta \mathit{AState} \mid \mathit{sec}' = (\mathit{sec} + 1) \bmod 60 \land \mathit{min}' = \mathit{min} \,] \\
\mathit{IncMin}\ \ ==\ \ [\, \Delta \mathit{AState} \mid \mathit{min}' = (\mathit{min} + 1) \bmod 60 \land \mathit{sec}' = \mathit{sec} \,] \\
\\
\mathit{Run} \mathrel{\widehat=}\ \ \mathit{tick} \longrightarrow \mathit{IncSec};\ ((\mathit{sec} = 0)\ \&\ \mathit{IncMin}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Box \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathit{sec} \neq 0)\ \&\ \mathbf{Skip})) \\
\qquad\qquad \Box \\
\qquad\qquad \mathit{time} \longrightarrow \mathit{out}\,!(\mathit{min}, \mathit{sec}) \longrightarrow \mathbf{Skip} \\
\bullet\ (\mathit{AInit};\ (\mu\ X \bullet (\mathit{Run};\ X))) \\
\\
\mathbf{end}
\end{array}
$$

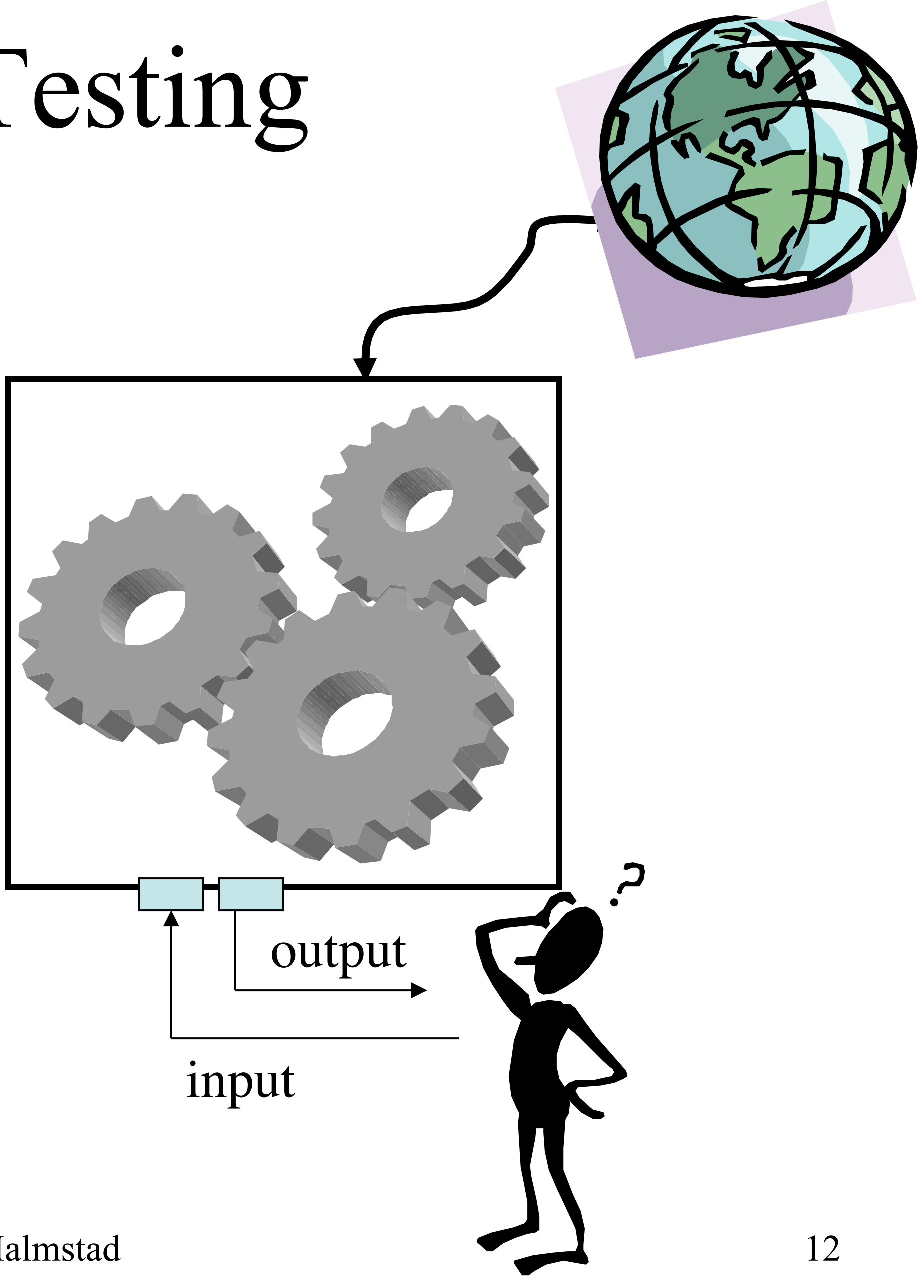
The *Chrono* process

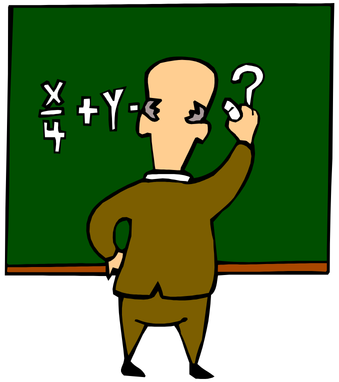
$$
\begin{array}{l}
\mathbf{process}\ \ \mathit{Clock} \mathrel{\widehat=}\ \ \mathbf{begin}\ \bullet \mu\ X \bullet \mathit{tick} \longrightarrow X\ \ \mathbf{end} \\
\mathbf{process}\ \ \mathit{TChrono} \mathrel{\widehat=}\ \ (\mathit{Chrono} \llbracket \{\!|\ \mathit{tick}\ |\!\} \rrbracket \mathit{Clock}) \setminus \{\!|\ \mathit{tick}\ |\!\}
\end{array}
$$



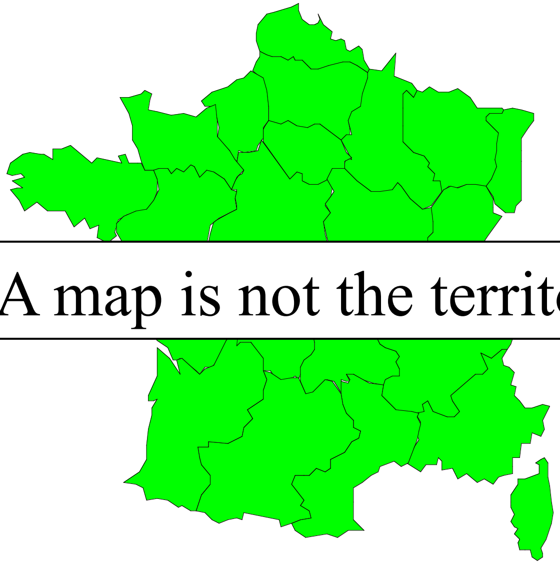
# 2 - Testing

- 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





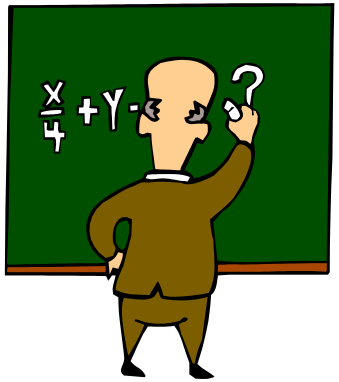
# A philosophical interlude



"A map is not the territory"* Korzybski

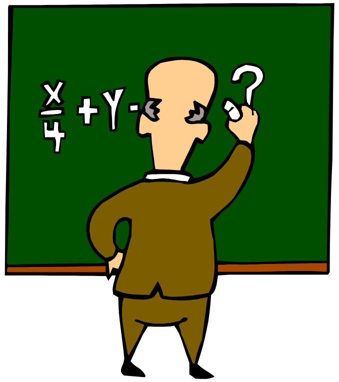
*A variant: "don't eat the menu…" ☺

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



# Black-Box Testing

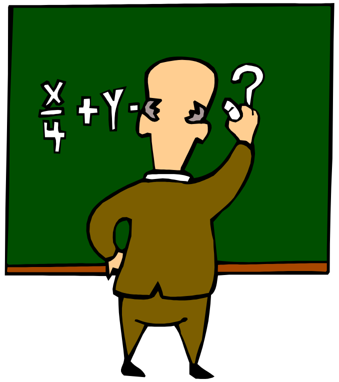
- *Black-Box Testing:*
  - the internal organisation of the SUT (System Under Test) is not known
- *However,*
  - Implicitely or explicitely, one considers a class of “testable implementations” => notion of ***Testability Hypotheses*** on the SUT



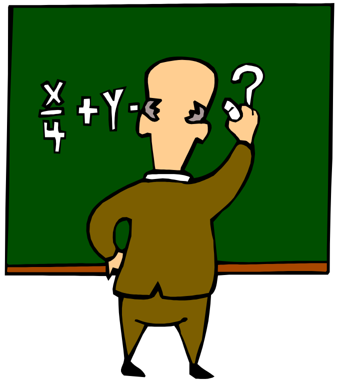
# Testability?

**SUT?**

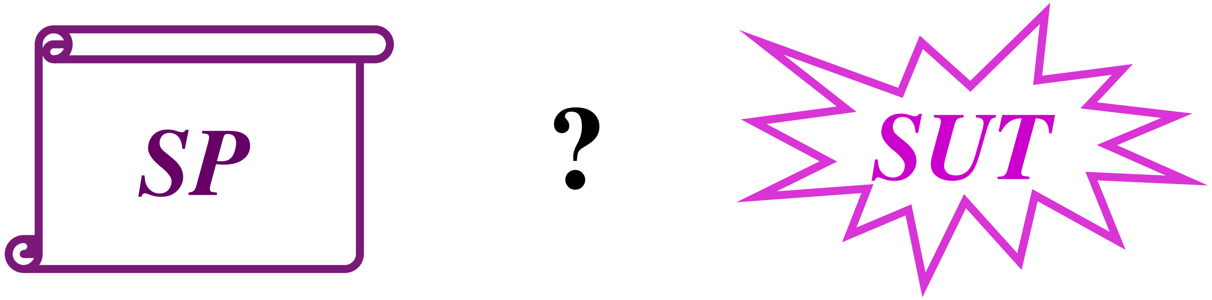
- If the SUT can be *any demonic system*, there is no sensible way of testing it ☹
- Fortunately, *some basic assumptions are feasible* (example: correct implementation of booleans and bounded integers, determinism, …)
- Some others can be *verified in another way*: static checks on the program, preliminary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



# FORMAL SPECIFICATIONS AND TESTING

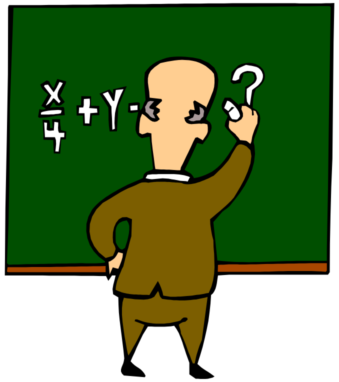


# Wanted: a satisfaction/ conformance relation



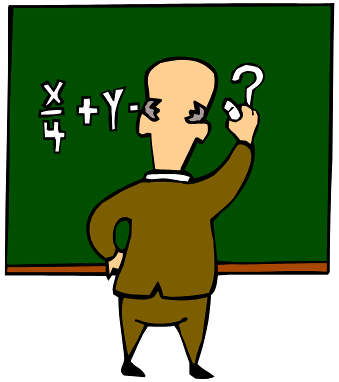
***SP*** **?** ***SUT***

- 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?

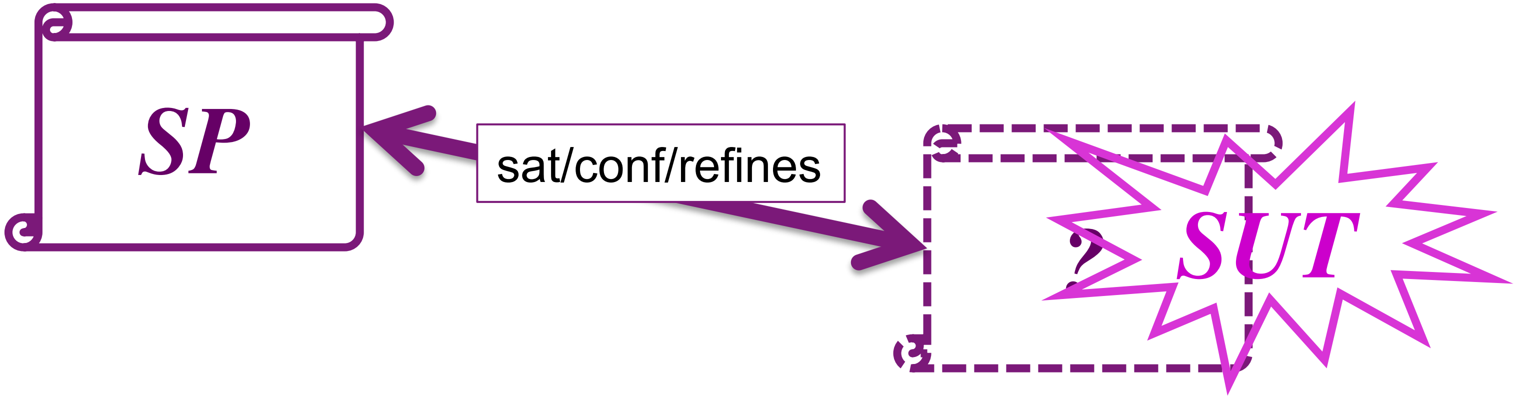


# A generic testability hypothesis

- *"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 by *SUT*
  - If *SP* is a process, *SUT* can be observed as a process, with traces and deadlocks
- Notation: $[\![SUT]\!]$

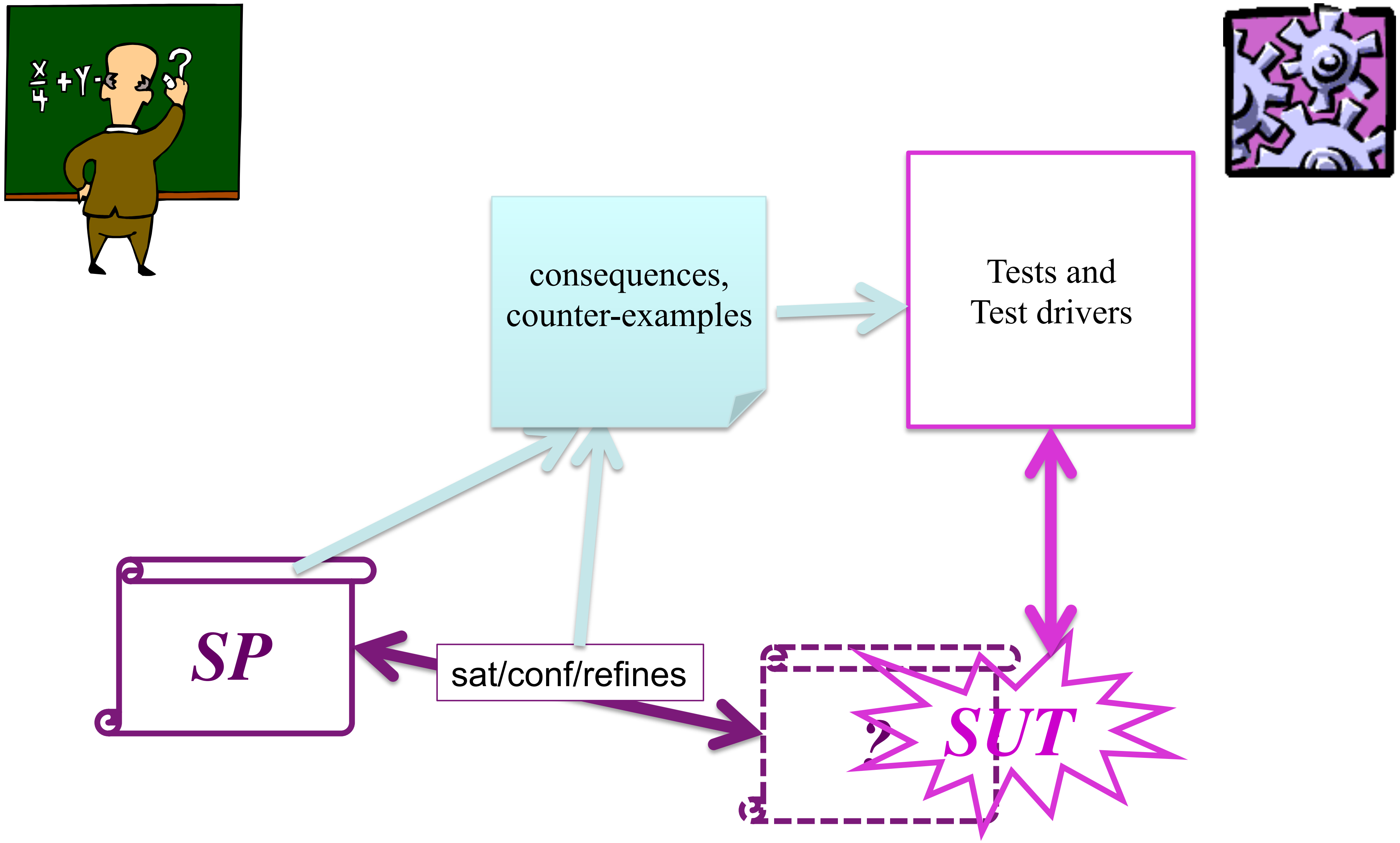


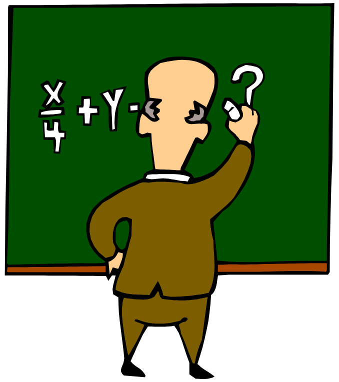
# Back to well-established relations



For instance, the *satisfaction/conformance* relation is

- equivalence for FSM,
- logical satisfaction for formulas,
- refinement for processes,
- *ioco* for LTS…


consequences,
counter-examples
Tests and
Test drivers
SP
sat/conf/refines
?
SUT



# Illustration: testing against *traces refinement* in CSP

$$\begin{gathered} Counter_2 = add \rightarrow C_1 \\ C_1 = add \rightarrow C_2 \,[\;]\, sub \rightarrow Counter_2 \\ C_2 = sub \rightarrow C_1 \end{gathered}$$

Traces of $Counter_2$

<>

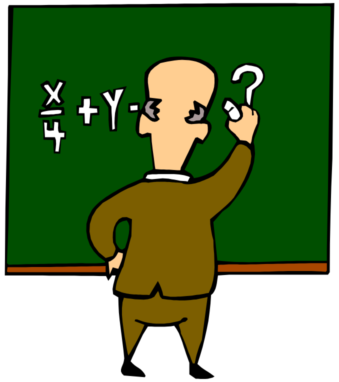
<*add*>

<*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*>

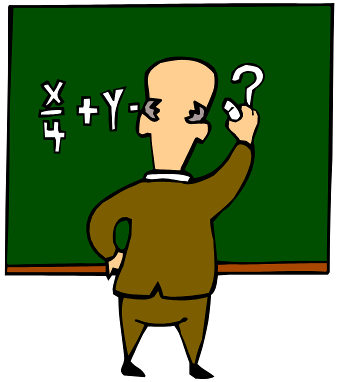
<*add,sub,sub*>

…

$test1 = pass \rightarrow sub \rightarrow fail \rightarrow STOP$

$test2 = inc \rightarrow add \rightarrow inc \rightarrow add \rightarrow pass \rightarrow add \rightarrow fail \rightarrow STOP$

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



# Illustration: testing against traces refinement in CSP

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

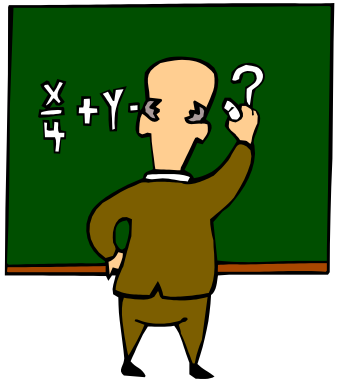
Traces of $Counter_2$
$\langle\rangle$
$\langle add \rangle$
$\langle add, add \rangle$
$\langle add, sub \rangle$
$\langle add, add, sub \rangle$
…

Forbidden traces
$\langle sub \rangle$
$\langle add, add, add \rangle$
$\langle add, sub, sub \rangle$
…

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

Test submissions
$SUT \,|[add,sub]|\, test1 \setminus [add,sub]$
$SUT \,|[add,sub]|\, test2 \setminus [add,sub]$
$SUT \,|[add,sub]|\, test3 \setminus [add,sub]$

Oracle: the last observed event is not *fail*



# Exhaustive test set for traces refinement of CSP

Let us consider the Test Set:

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

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

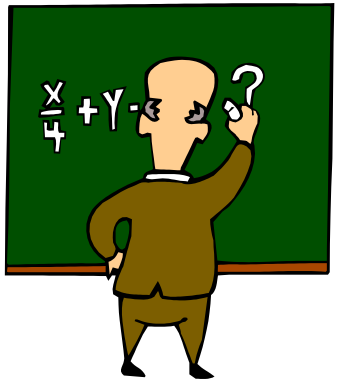
For any test *T*, its execution against *SUT* is specified as:

$\mathbf{Execution_{SP,SUT}(T) = (SUT \mid[ \alpha SP ]\mid T) \backslash \alpha SP}$

**Theorem (Cavalcanti Gaudel 2007)** :

*SUT* is a traces refinement of *SP* iff

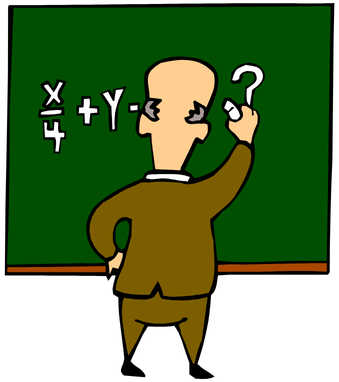
$\mathbf{\forall T_T(s, a) \in Exhaust_T(SP), \forall t \in traces(Execution_{SP,SUT}(T_T(s, a))), \neg last(t) = fail}$



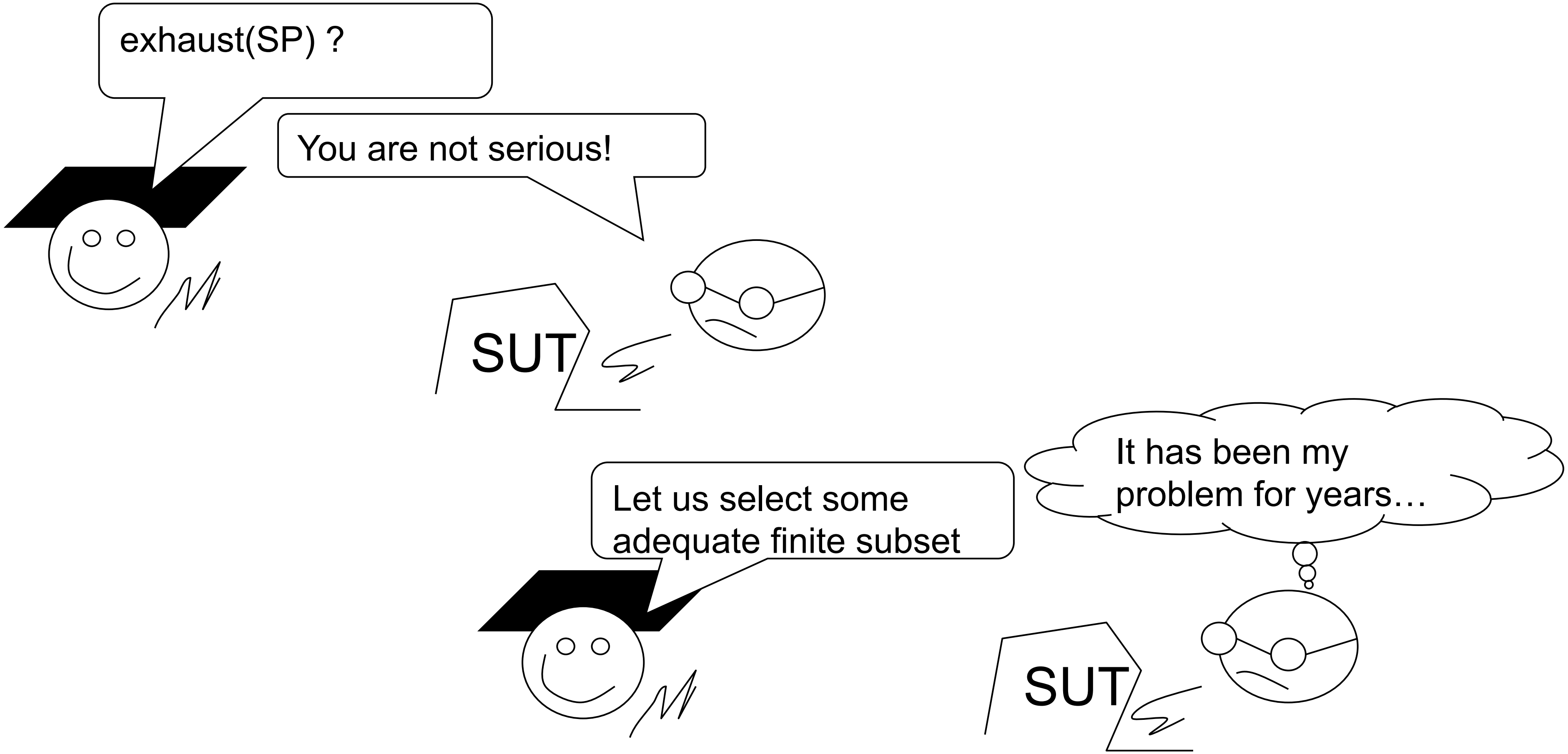
# The corresponding testability hypotheses

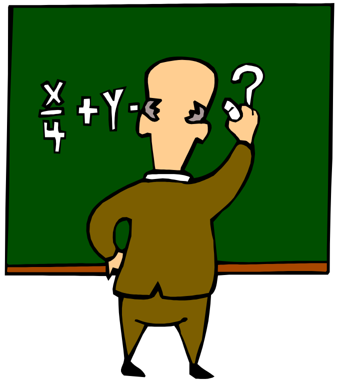
- *SUT* behaves like a CSP process
  - – With the same alphabet of actions as *SP*
  - – The *actions and events are atomic*

- If *SUT* is non-determinist, it satisfies the classical *complete testing assumption*



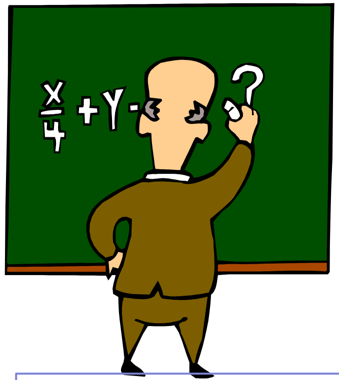
# Exhaustivity is not practicable





# 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 …



# An example from CSP

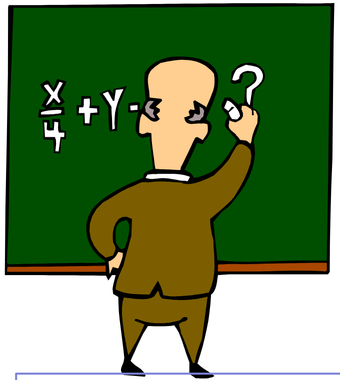
$$Replicator = c?x:Int \rightarrow d!x \rightarrow Replicator$$

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

$(FreshInt(0)\,|[c]|\,Replicator) \setminus c$ 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
$<d.v>\ \forall\ v \in Int$
$<c.v,\ d.w>\ \forall\ v,w \in Int,\ v \neq w$
$<c.v,\ c.w>\ \forall\ v,w \in Int$
$<c.v,\ d.v,\ d.w>\ \forall\ v,w \in Int$
$<c.v,\ d.v,\ c.w,\ d.u> \forall\ v,w,u \in Int,\ w \neq u$
…



# 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

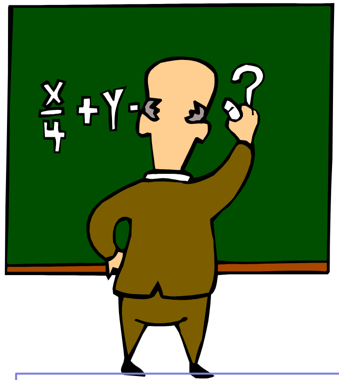
with hidden synchronisation on *c*

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>* ∀ $v \in Int$
*<c.v, d.w>* ∀ $v,w \in Int,\ v \neq w$
*<c.v, c.w>* ∀ $v,w \in Int$
*<c.v, d.v, d.w>* ∀ $v,w \in Int$
*<c.v, d.v, c.w, d.u>*∀ $v,w,u \in Int,\ w \neq u$
…

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

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

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

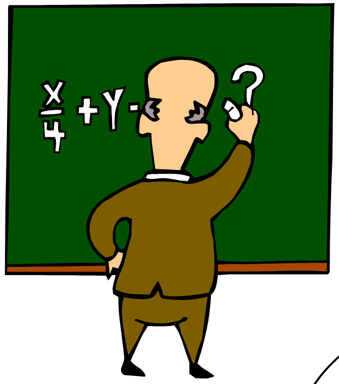
$(FreshInt(0)\,|[c]|\,Replicator)\setminus c$ parallel composition with hidden synchronisation on $c$

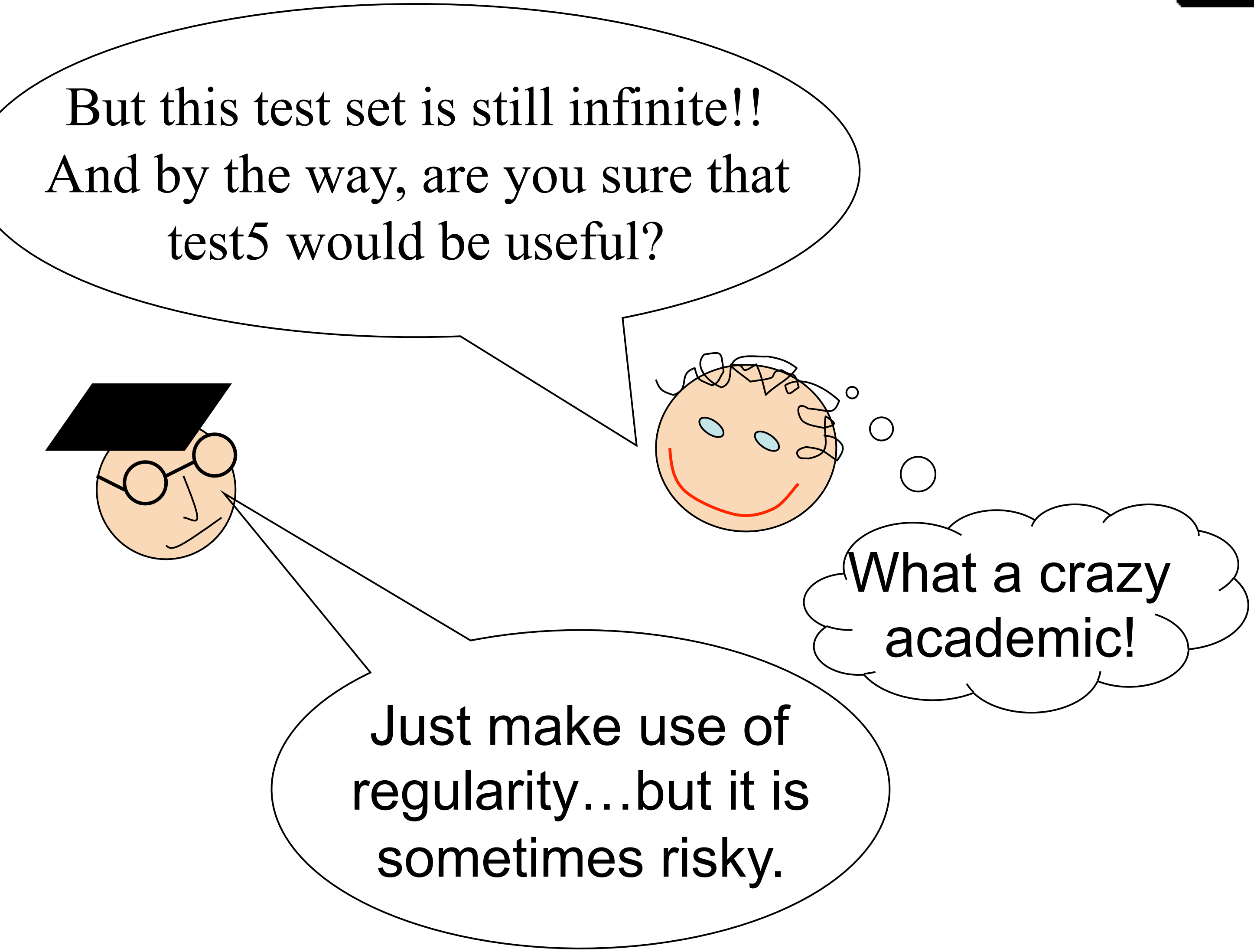
Traces of Replicator
<>
<c.0>  <c.1>…
<c.0,d.0> <c.1,d.1>…
…<c.0,d.0,c.7>…

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

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

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

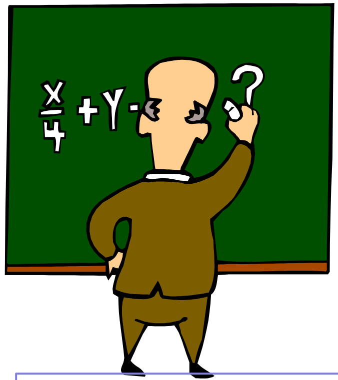




But this test set is still infinite!!
And by the way, are you sure that
test5 would be useful?

What a crazy
academic!

Just make use of
regularity…but it is
sometimes risky.



# An example of regularity hypothesis

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

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

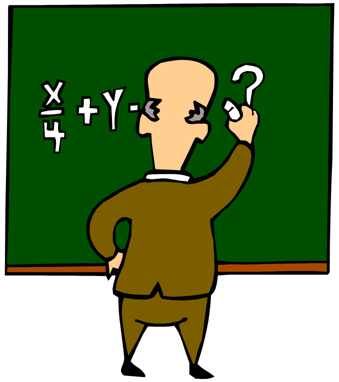
$$(FreshInt(0)\,|[c]|\,Replicator) \setminus c$$ parallel composition with hidden synchronisation on *c*

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>* $\forall v,w,u \in Int,\ w \neq u$
…

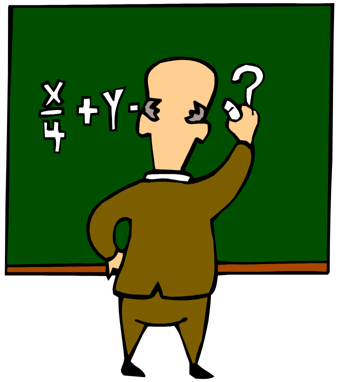
There is no dependency between the recursive calls of *Replicator.*
There is no shared state.
⇒ If the SUT is determinist, one execution is sufficient => Regularity Hypothesis => **Finite Test Set**

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



# 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*



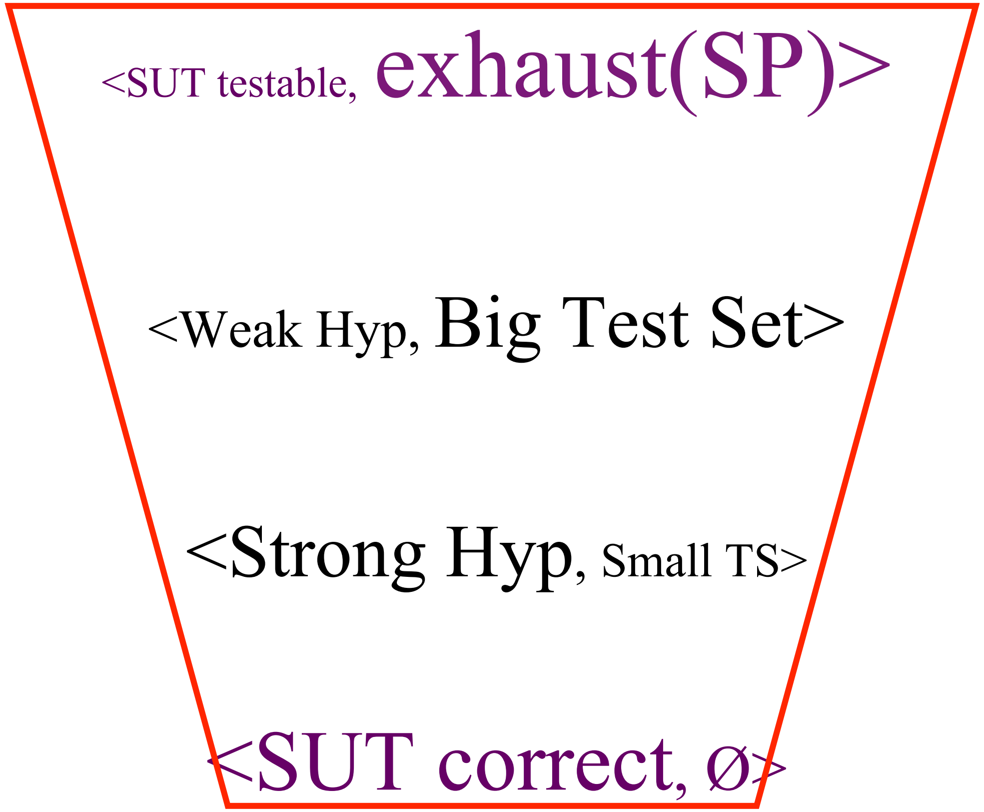
# Selection of finite test sets

- "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*



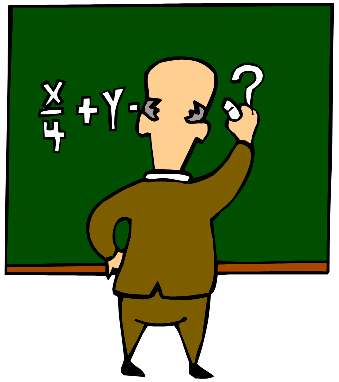
<SUT testable, exhaust(SP)>

<Weak Hyp, Big Test Set>

<Strong Hyp, Small TS>

<SUT correct, Ø>

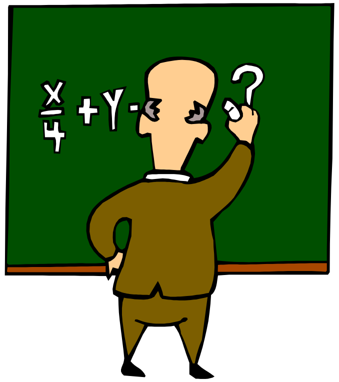
# SOME BASIC TECHNIQUES FOR CASE SPLITTING



# “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)



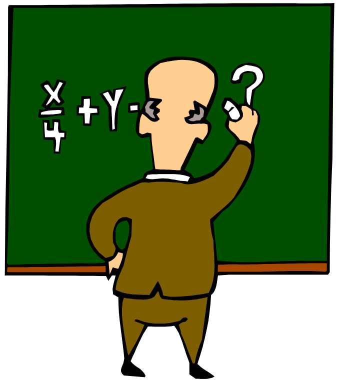
# Case splitting

Two main techniques:

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

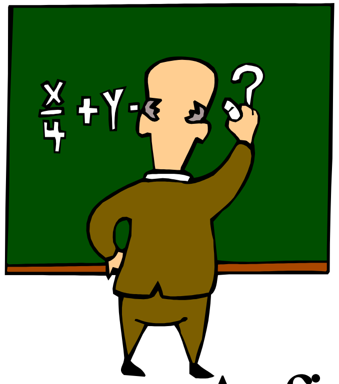
Implementations:

- Conditional rewriting, Narrowing
- Symbolic evaluation



# DNF?

- It is a disjunction (sequence of ORs) consisting of one or more disjuncts, each of which is a conjunction (AND) of one or more literals (i.e., statement letters and negations of statement letters; Mendelson 1997, p. 30)
- ∧, ∨, and ¬ are the only logical operators, ¬ is the most internal, then ∧, then ∨
- Intuitively, this gives a list of disjoint test cases.

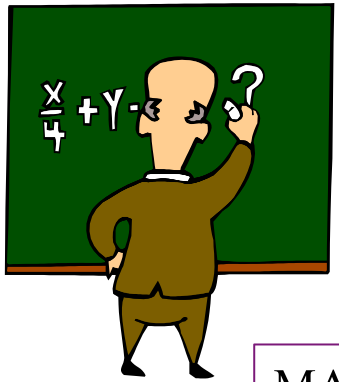


# More on DNF

- A first example of DNF decomposition:

$(p \vee q) \rightarrow \neg r \;\Leftrightarrow\; (p \wedge q \wedge \neg r) \vee (p \wedge \neg q \wedge \neg r) \vee (\neg p \wedge q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge \neg q \wedge \neg r)$

- Basic rules:
  - $(p \vee q)$ is decomposed into 3 disjoint cases: $p \wedge q$, $p \wedge \neg q$, $\neg p \wedge q$
  - $(A \rightarrow B)$ is decomposed into $\neg A$ and $A \wedge B$
  - $\neg\neg$ are eliminated
- Not very difficult, but…exponential explosion



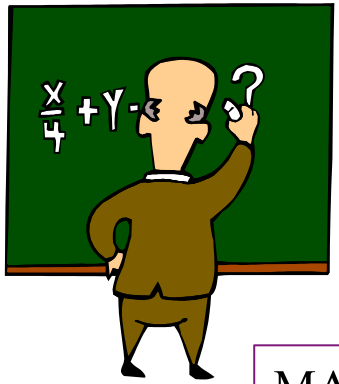
# Example of the reduction of pre/post-conditions

MAX (a:ℤ, b:ℤ)
**result** max:ℤ
**pre** true
**post** (max=a ∨ max=b) ∧ max≥a ∧ max≥b

(conjunction of pre-condition, post-condition and state Invariant, if any) =>

true ∧(max=a ∨ max=b)∧ max≥a ∧ max≥b

Thanks to Jeremy Dick and Alain Faivre



# Example of the reduction of pre/post-conditions

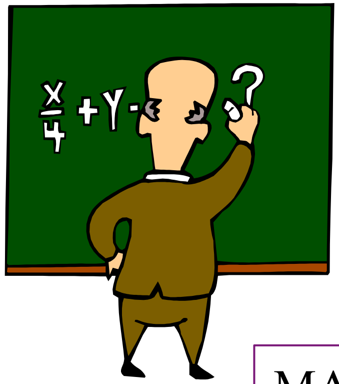
MAX (a:ℤ, b:ℤ)
**result** max:ℤ
**pre** true
**post** (max=a ∨ max=b) ∧ max≥a ∧ max≥b

(conjunction of pre-condition, post-condition and state Invariant, if any) =>

true ∧(max=a ∨ max=b)∧ max≥a ∧ max≥b

(simplification of “true ∧…”) =>

(max=a ∨ max=b)∧ max≥a ∧ max≥b



# Example of the reduction of pre/post-conditions

MAX (a:$\mathbb{Z}$, b:$\mathbb{Z}$)
**result** max:$\mathbb{Z}$
**pre** true
**post** $(max=a \lor max=b) \land max \geq a \land max \geq b$

(conjunction of pre-condition, post-condition and state Invariant, if any) =>

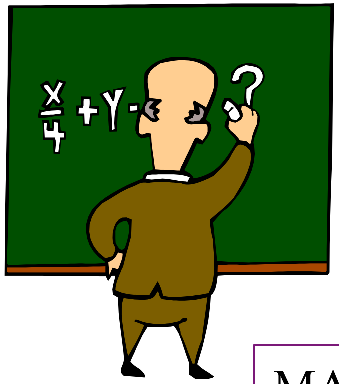
$true \land (max=a \lor max=b) \land max \geq a \land max \geq b$

(simplification of “true $\land$…”) =>

$(max=a \lor max=b) \land max \geq a \land max \geq b$

(distribution of $\lor$ ) =>

$(max=a \land max \geq a \land max \geq b) \lor (max=b \land max \geq a \land max \geq b)$



# Example of the reduction of pre/post-conditions

MAX (a:$\mathbb{Z}$, b:$\mathbb{Z}$)
**result** max:$\mathbb{Z}$
**pre** true
**post** $(max=a \lor max=b) \land max \ge a \land max \ge b$

(conjunction of pre-condition, post-condition and state Invariant, if any) =>

$true \land (max=a \lor max=b) \land max \ge a \land max \ge b$

(simplification of “true ∧…”) =>

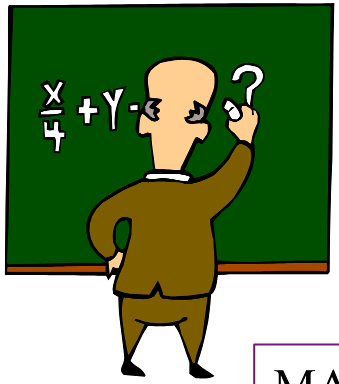
$(max=a \lor max=b) \land max \ge a \land max \ge b$

(distribution of ∨ ) =>

$(max=a \land max \ge a \land max \ge b) \lor (max=b \land max \ge a \land max \ge b)$

(decomposition of ∨) =>

$(max=a \land max=b \land max \ge a \land max \ge b) \lor$
$(max=a \land max \ne b \land max \ge a \land max \ge b) \lor$
$(max \ne a \land max=b \land max \ge a \land max \ge b)$



# Example of the reduction of pre/post-conditions

MAX (a:$\mathbb{Z}$, b:$\mathbb{Z}$)
**result** max:$\mathbb{Z}$
**pre** true
**post** $(max=a \vee max=b) \wedge max \geq a \wedge max \geq b$

(conjunction of pre-condition, post-condition and state Invariant, if any) =>

$true \wedge (max=a \vee max=b) \wedge max \geq a \wedge max \geq b$

(simplification of “true ∧…”) =>

$(max=a \vee max=b) \wedge max \geq a \wedge max \geq b$

(distribution of ∨ ) =>

$(max=a \wedge max \geq a \wedge max \geq b) \vee (max=b \wedge max \geq a \wedge max \geq b)$

(decomposition of ∨) =>

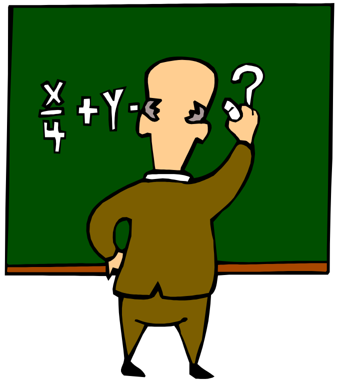
$(max=a \wedge max=b \wedge max \geq a \wedge max \geq b) \vee$
$(max=a \wedge max \neq b \wedge max \geq a \wedge max \geq b) \vee$
$(max \neq a \wedge max=b \wedge max \geq a \wedge max \geq b)$

(simplifications) =>

$(max=a \wedge max=b) \vee$
$(max=a \wedge max>b) \vee$
$(max=b \wedge max>a)$

3 test cases: {(a=b, max=a=b), (a>b, max = a), (b>a, max =b)}

Thus, *3 uniformity sub-domains + oracles*.



# Unfolding

- Unfolding is a classical technique for transforming (and understanding) recursive definitions
- It is just replacement of *f(op(x))* by the definition(s) of *f*, with adequate renaming of variables
  - *fact(n)* $=_{def}$ ***if*** *n=0* ***then*** *1* ***else*** *n*fact(n-1)* becomes:
  - *fact(n)* $=_{def}$ ***if*** *n=0* ***then*** *1* ***else if*** *(n-1)=0* ***then*** *n*1* ***else*** *n*(n-1)*fact(n-2)*
    - i.e. *fact(n)* $=_{def}$ ***if*** *n=0* ***then*** *1* ***else if*** *n=1* ***then*** *1* ***else*** *n*(n-1)*fact(n-2)*
  - etc
  - Going on, the definition of the *fact* function is replaced by its graph, i.e. its *exhaustive test set* ☺…