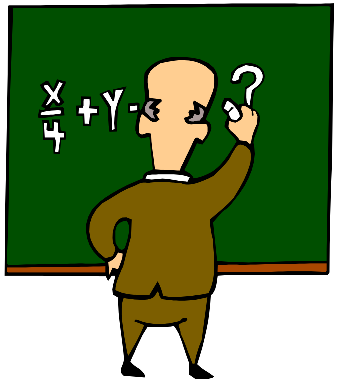
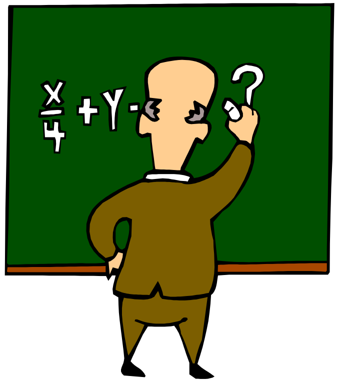
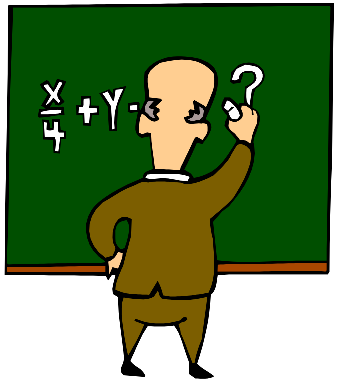


SOFTWARE TESTING BASED ON AXIOMS



Algebraic Specifications

- Abstract Data Types
- Description of required properties, independent of implementation
- Signature: sorts of values, opérations with profile
- + Axioms: equations, conditional equations (1st order formulas)
- (+ Constraints: hierarchy, finite generation)



A VERY basic example

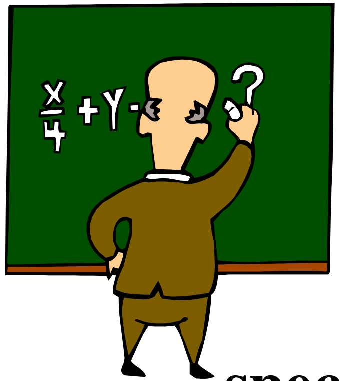
spec BOOL

free generated type *Bool* ::= true | false

op *not : Bool* → *Bool*

- not(true) = false
- not(false) = true

end



A more sophisticated one

spec CONTAINER = NAT, BOOL
then

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

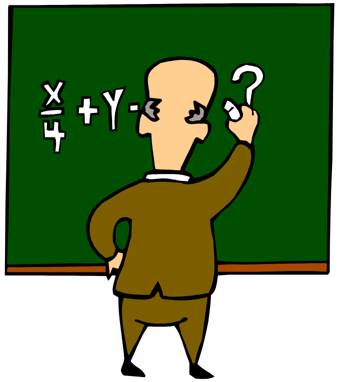
op *isin* : *Nat* × *Container* → *Bool*

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

∀ *x*, *y*:*Nat*; *c*:*Container*

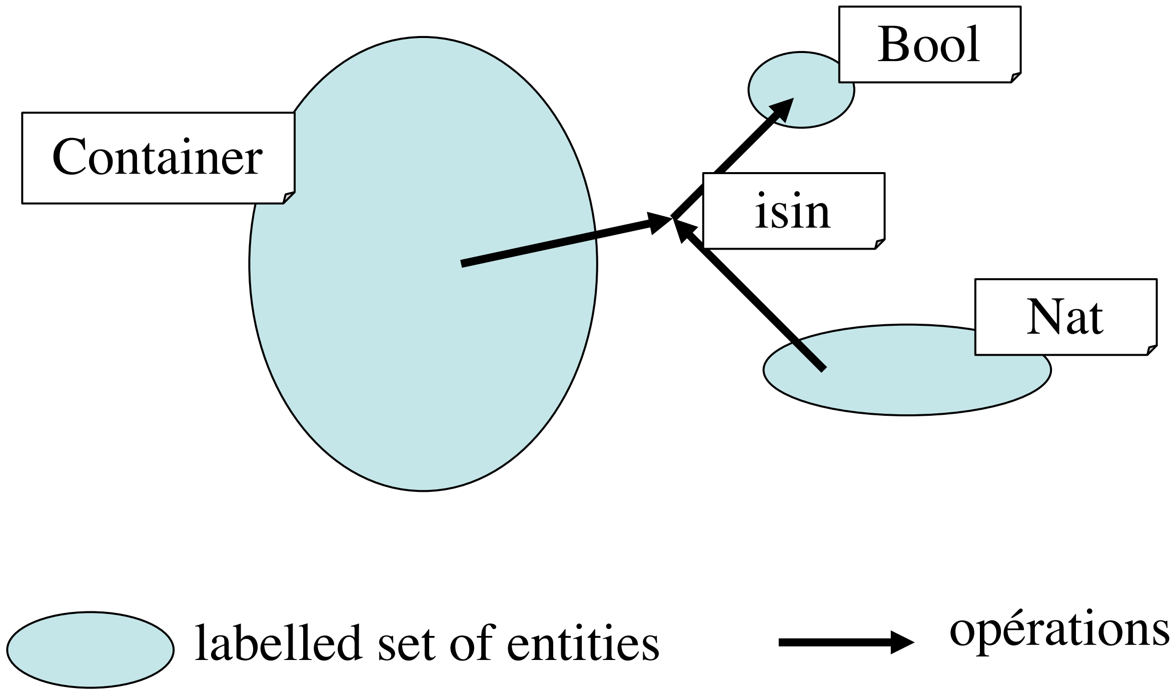
- *isin(x, [ ]) = false*
- *eq(x, y) = true ⇒ isin(x, y::c) = true*
- *eq(x, y) = false ⇒ isin(x, y::c) = isin(x,c)*
- *remove(x, [ ]) = [ ]*
- *eq(x, y) = true ⇒ remove(x, y::c) = c*
- *eq(x, y) = false ⇒ remove(x, y::c) = y::remove(x,c)*

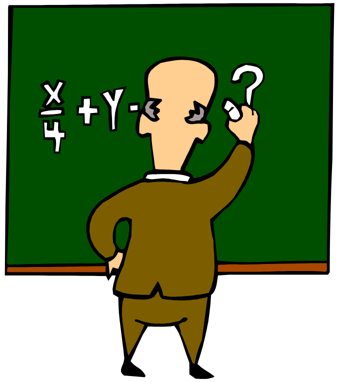
end



Formalities

- **Semantics**
  - Many-sorted algebras: sets of values and functions



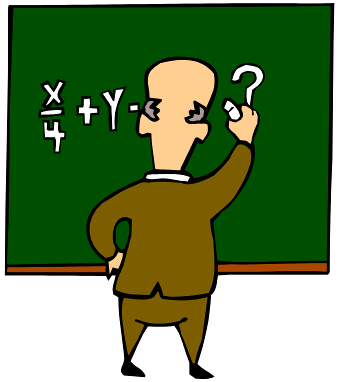


Specificities of testing based on Axioms

- It is not natural to test the operations with couples <input, output>
  - Note that several outputs may be acceptable…
- What must be verified is that *the constructs which implement the operations satisfy the axioms*
- Exercises :

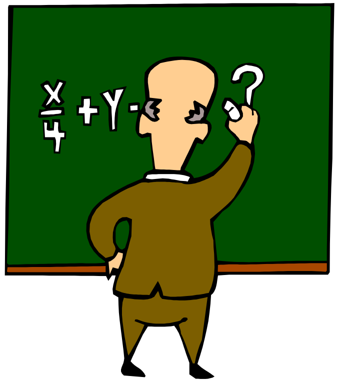
$x + y = y + x$

$eq(x, y) = true \Rightarrow isin(x, y{::}c) = true$



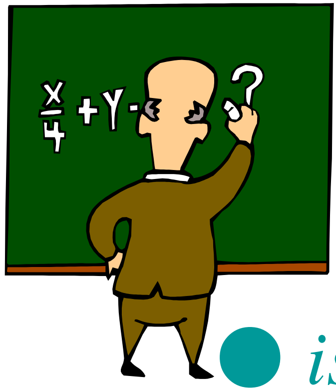
Link between the specification and the SUT

- The SUT provides some procedures, functions, methods, for executing the operations of the signature
  - (example : Java class, Ada package, ML structure…)
- Let note $\boldsymbol{op_{SUT}}$ the implementation of $\boldsymbol{op}$
- Let $\boldsymbol{t}$ an expression without variable written with some operations and constants of the signature,
- we note $\boldsymbol{t_{SUT}}$ the result of its computation by the SUT,



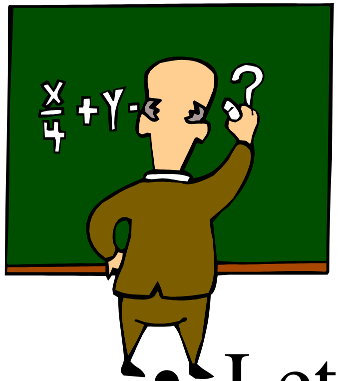
What is a test?

- Let ε some equation written with the operations of the signature (and, may be, some variables)
  - Test of ε : any close instantiation $t = t'$ of ε
  - Test experiment of *SUT* against $t = t'$: evaluations of t_{SUT} and t'_{SUT} and comparison of the resulting values
  - **NB** : oracle $\Leftrightarrow$ test of equality
- Straightforward generalisation to conditional equations; less straightforward for some 1st order formulas ($\forall$, $\exists$) (cf. Machado 1998, Aiguier et al. 2016, etc).



Examples of tests (simplified for conditional axioms)

- *isin(x, [ ]) = false*
  - isin (0, []) = false
- *eq(x, y) = true ⇒ isin(x, y::c) = true*
  - isin(1, 1::2:: []) = true
- *eq(x, y) = false ⇒ isin(x, y::c) = isin(x,c)*
  - isin(1, 0::3:: []) = false
- *remove(x, [ ]) = [ ]*
  - remove(5, []) = []
- *eq(x, y) = true ⇒ remove(x, y::c) = c*
  - remove(0, 0::3::4:: []) = 3::4::[]
- *eq(x, y) = false ⇒ remove(x, y::c) = y::remove(x,c)*
  - remove(1, 7::5::3:: []) = 7::5::3:: []



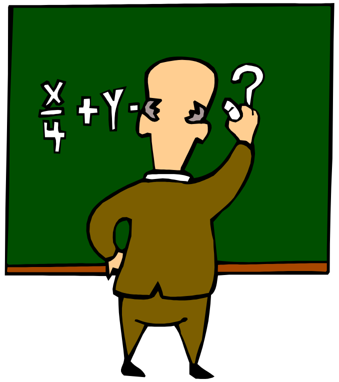
Exhaustive test set

- Let $SP = (\Sigma, Ax)$
- The exhaustive test set of SP, noted $Exhaust_{SP}$ is the set of all the closed well-sorted instances of all the axioms of SP:

$$Exhaust_{SP} = \{ \Phi\sigma \mid$$

$$\Phi \in Ax,\ \sigma = \{\sigma_s : var(\Phi)_s \rightarrow (T_\Sigma)_s \mid s \in S\} \}$$

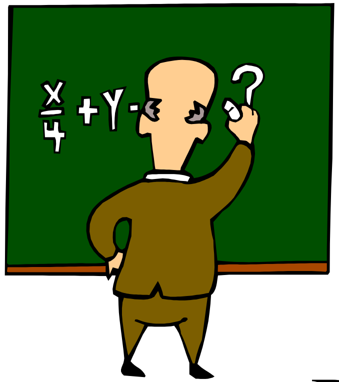
  - **NB1** : definition derived from the classical notion of axiom satisfaction
  - **NB2** : some tests are inconclusive and can be removed



Testability Hypotheses

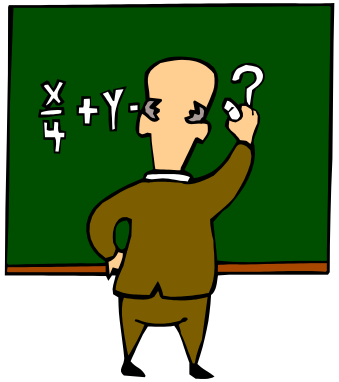
- Unavoidable : when testing a system, it is impossible not to make assumptions on its behaviour and its environment

- **Remark :** $Exhaust_{SP}$ is exhaustive w.r.t. the specification, not always w.r.t. the implementation ☹
- Here : a SUT is Σ-*testable* if:
  - The operations of Σ are implemented in a deterministic way
  - All the values are specified by Σ (no junks, Σ-generation)



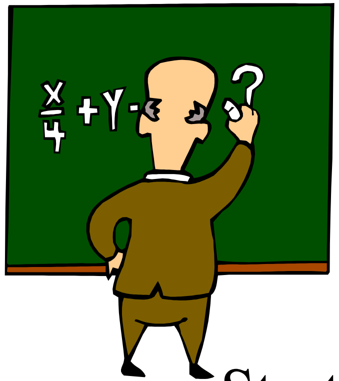
Another exhaustivity

- Based on a different (operational) semantics
  - $\{t = t{\downarrow} \,/\, T_{\Sigma}\}$
  - T_{Σ} is the (sorted) set of ground Σ-terms
  - $t{\downarrow}$ is the normal form of t, when using the axioms as conditional rewriting rules
- Restriction on the class of specifications
  - The axioms must define a convergent term rewriting system
- Weakening of the testability hypothesis
  - Finite generation is no more required



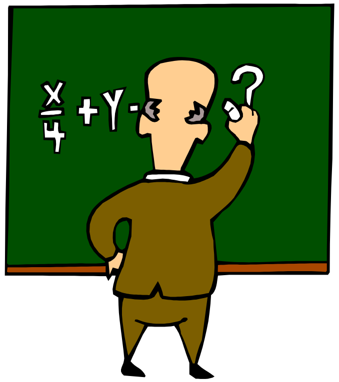
Selection Hypotheses

- *Uniformity Hypothesis*
  - $\Phi(X)$ formula, *SUT* system, *D* sub-domain
  - $(\forall t_0 \in D)(\llbracket SUT \rrbracket \models \Phi(t_0) \Rightarrow (\forall t \in D)\ (\llbracket SUT \rrbracket \models \Phi(t)))$
  - Determination of sub-domains ? *guided by the axioms, see later…*
- *Regularity Hypothesis*
  - $((\forall t \in T_\Sigma)\ (\,|t| \leq k \Rightarrow \llbracket SUT \rrbracket \models \Phi(t)\,)) \Rightarrow (\forall t \in T_\Sigma)\ (\llbracket SUT \rrbracket \models \Phi(t))$
  - Determination of |t|? *guided by the axioms or… by necessity* 😕



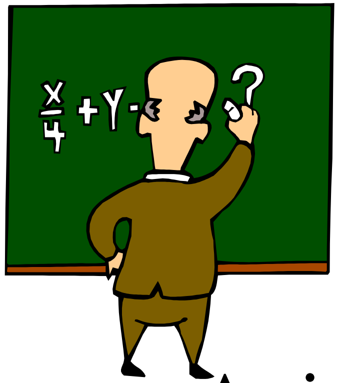
A Method

- Starting point : axioms coverage (one test by axiom)
- => Strong uniformity hypotheses on the sorts of the variables or on the validity domain of the premisses
- Example : 6 tests for CONTAINER
  - isin (0, []) = false
  - isin(1, 1::2:: []) = true
  - isin(1, 0::3:: []) = false
  - remove(1, []) = [],
  - remove(0, 0::3:: []) = 3:: []
  - remove(1, 3:: []) = 3:: []
- Uniformity on *Nat*, on pairs of *Nat* such that *eq(x,y) = true*, on pairs of *Nat* such as *eq(x, y) = false*
- Uniformity on *Container*



Weakening of hypotheses

- Successive weakening using the axioms of the specification
- A natural way for discovering sub-domains is to perform *some case analysis of the specification*
- Example : the *isin* function is defined by 3 axioms
  - $isin(x, [\,]) = false$
  - $eq(x, y) = true \Rightarrow isin(x, y::c) = true$
  - $eq(x, y) = false \Rightarrow isin(x, y::c) = isin(x,c)$
- => 3 tests. But one may want to go further
  – The occurrences of *isin(,)* can be decomposed into these 3 subcases



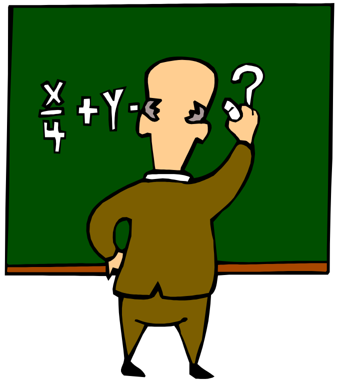
2 main techniques for weakening uniformity hypotheses

- Axioms Composition
  - For instance, given the axioms:

    $eq(x,y) = true \Rightarrow le\ (x, y) = true$

    $lt(x,y) = true \Rightarrow le\ (x, y) = true$

    $lt(x,y) = false \wedge eq(x, y) = false \Rightarrow le\ (x, y) = false$
  - any occurrence of le(,) in an axiom can be decomposed into 3 sub-cases (or 2, or 1… depending on its context)
- Unfolding of recursive occurrences,
  - see next slide

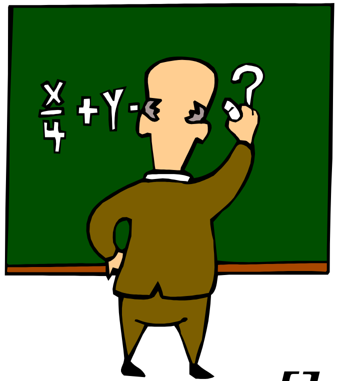


Unfolding *isin*

The definition of *isin* is:

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

- Thus any term $isin(t1,t2)$ may correspond to three subcases
  - $t2=[\,]$: $isin(t1,t2)$ can be replaced by *false*
  - $t2= y::c$, and $eq(t1, y)=true$: it can be replaced by *true*
  - $t2= y::c$, and $eq(t1, y)=false$: it can be replaced by $y::isin(t1,c)$

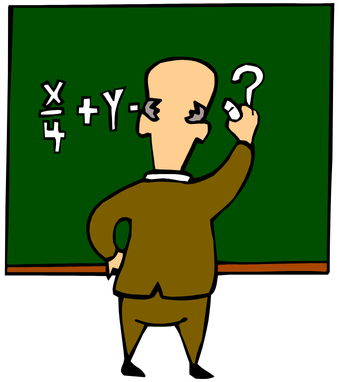


Infolding the 3rd (red) axiom

eq(x, y) = false ⇒ isin(x, y::c) = isin(x,c)

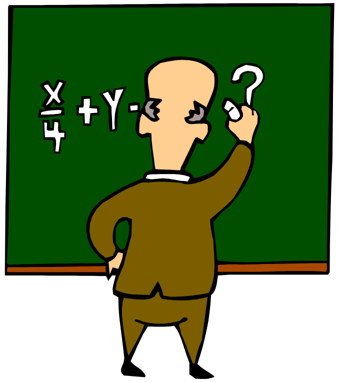
3 new test cases

- ***c=[]:***
  - *eq(x, y) = false ∧ c=[ ] ⇒ isin(x, y::c) = isin(x,c)*
  - *eq(x, y) = false ⇒ isin(x, y::[ ]) = false*
- ***c= y'::c'*, and *eq(x, y')=true***
  - *eq(x, y) = false ∧ c= y'::c' ∧ eq(x, y')=true ⇒ isin(x, y::y'::c') = isin(x, y'::c')*
  - *eq(x, y) = false ∧ eq(x, y')=true ⇒ isin(x, y::y'::c') =true*
- ***c= y'::c'*, and *eq(x, y')=false*:**
  - *eq(x, y) = false ∧ c= y'::c' ∧ eq(x, y')=false ⇒ isin(x, y::y'::c') = isin(x, y'::c')*
  - *eq(x, y) = false ∧ eq(x, y')=true ⇒ isin(x, y::y'::c') =false*



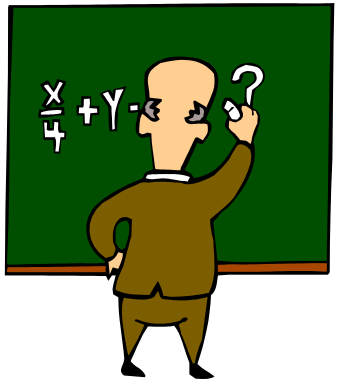
When and how to stop

- Depending on the context (risk, cost, schedule, …), one chooses for each specification:
  - What boolean functions or predicates to decompose (le, or, and, …)
  - What operations to unfold and how many times (rarely more than once, but there are counter examples)
- Some good standard strategy : composition of all pairs of sub-cases
  - NB : There may be unfeasible compositions



The oracle problem

- Decision that t_{SUT} and t'_{SUT} are “equal”
- The simple case :
  - the sort *s* of *t* and *t’* corresponds to some type of the programming language with a built-in equality (observable sort)
- “Weak oracle hypothese”: the built-in equality on the types of the programming language, and the booleans, are correctly implemented

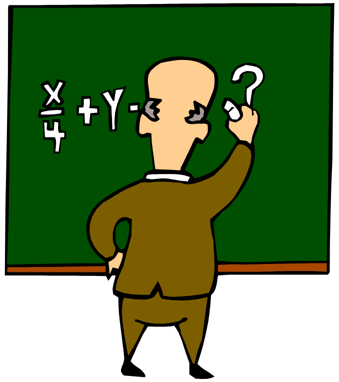


The other cases

- How to test that

  $eq(x, y) = false \Rightarrow remove(x, y::c) = y::remove(x,c)$?

- Suppose that containers are represented by hash-tables, or ordered trees, or …
- Solution: *observable contexts*
  - Test that all the possible "observations" on the two results are equal
  - **Observation** : (minimal) composition of operations of the signature that yields an observable results

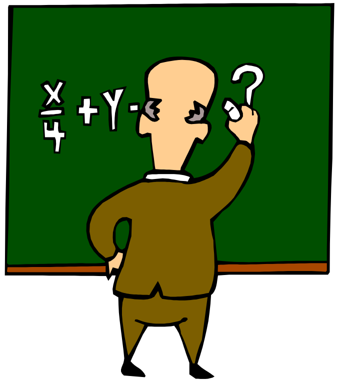


Observable contexts

- The CONTAINER example
  - *isin(n, _),* for all n: Nat

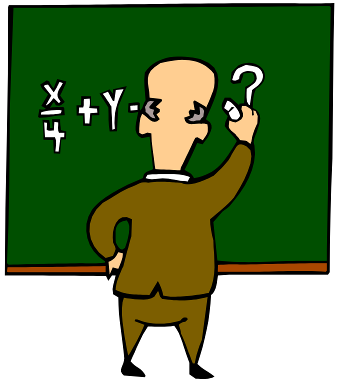
    $eq(x, y) = false \Rightarrow isin(n, remove(x, y::c)) =$
    $isin(n, y::remove(x,c)\ )$

  As in this case, there is often an infinity of observable contexts ☹
- Need for selection strategies
  - Either among the observable contexts => partial oracle
  - Either among a new observable exhaustive test set (see Gaudel Le Gall 2007)

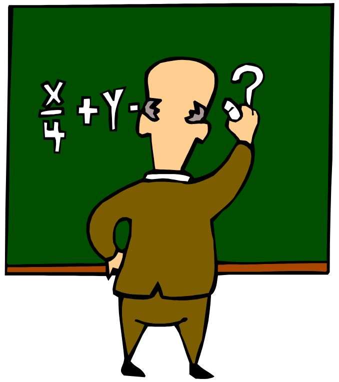


Some applications of testing based on axioms

- Onboard part of the driving system of an automatic subway (line D, Lyon)
- pieces of software written in C, parts of a nuclear safety shutdown system.
- EPFL library of Ada components
- *Validation* of a transit node specification
- test of an implementation of the Two-Phase-Commit protocol
- JML, SPEC#, are derived from algebraic specifications

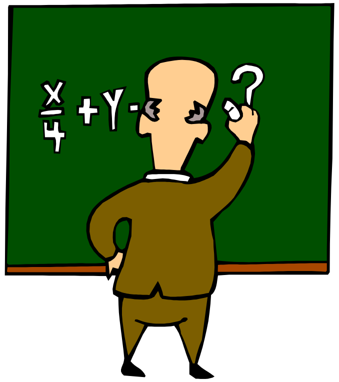


SOFTWARE TESTING BASED ON FINITE STATE MACHINES



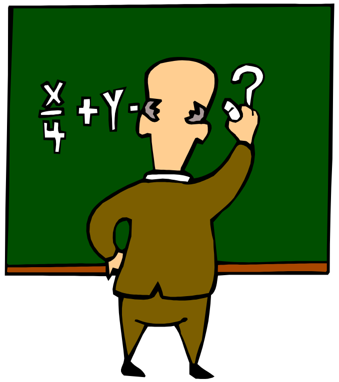
Back in history: FSM-based testing

- Originally invented in the sixties for testing circuits, *thus there is a finite number of states*
- First applied to software by Chow in 78
- Big corpus of knowledge, with a lot of variants on the kind of considered FSM
- The "Bible" on the subject: [Lee & Yannakakis 1996]

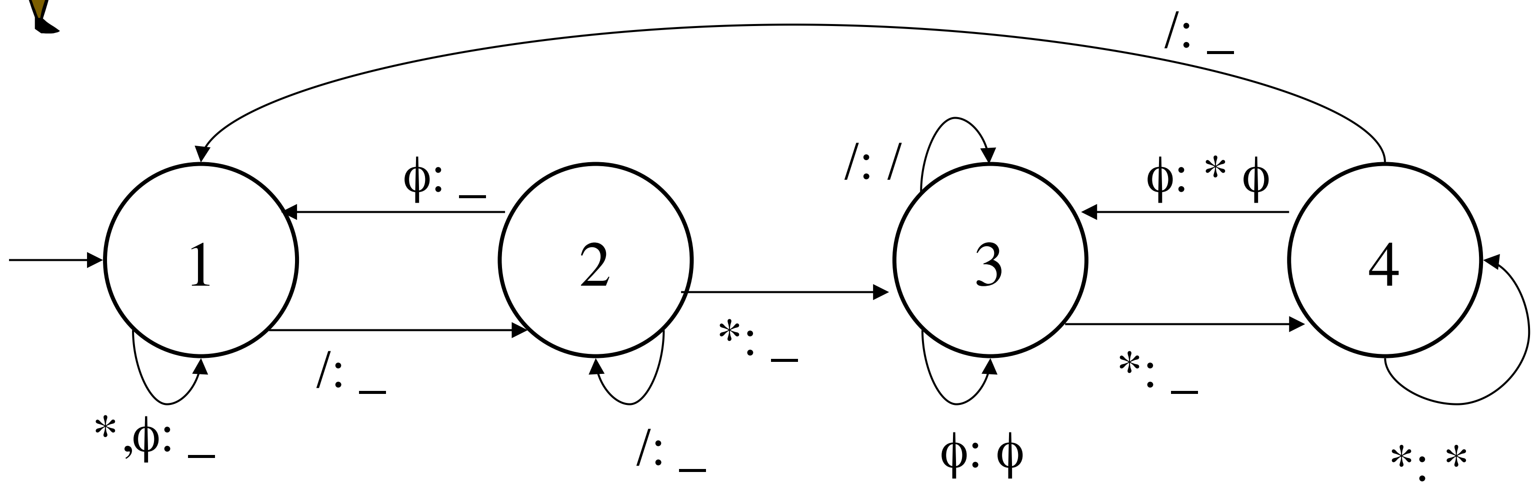


What is an FSM?

- S: finite set of states, I: input alphabet, O: output alphabet
- T: finite set of transitions:
  - **s –x:y-> s'** $\in$ **T**, s, s'$\in$S, x $\in$I, y $\in$O
  - **Notations**: $\lambda(s,x)=y$, $\lambda^*(s,w)=w'$, w $\in$I*, w' $\in$O*
- ***Equivalent states* : $\forall$w, $\lambda^*(s,w) = \lambda^*(s',w)$**
- Here, the considered FSM are: deterministic, complete *($\forall$ s$\in$S, $\forall$x$\in$I, $\exists$ s –x:y-> s' $\in$T)*, minimal *(no equivalent states)*, and all states are reachable.

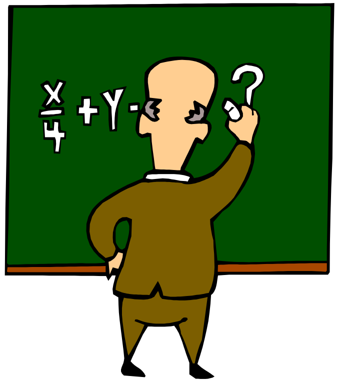


The same example



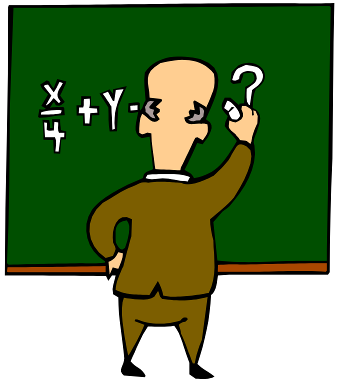
This FSM removes from the input text all that is not a comment

ϕ is any character but * and /
This is not a comment **/*** all that / * is ** a comment ***/** this is no more a comment.



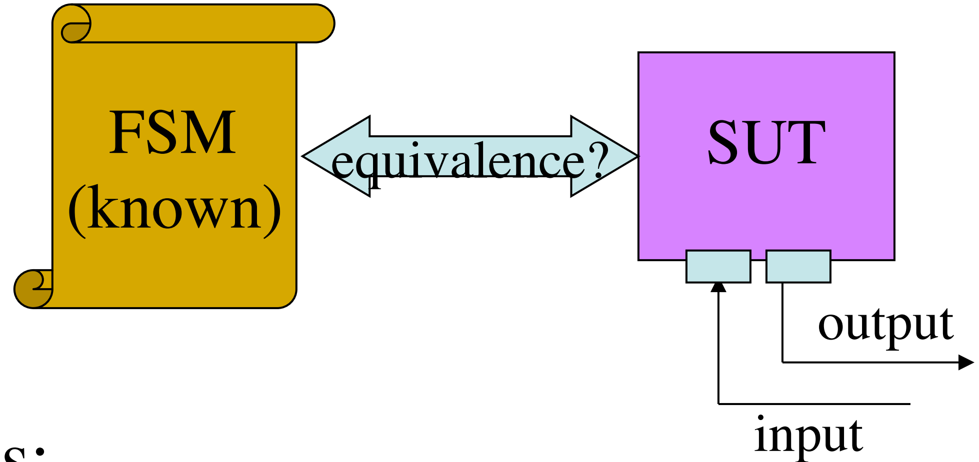
Formalities

- A FSM is a “regular transducer”
- It defines a function from I* into O*
- *There is no memory*: given an input, the output depends only on the current state and not on the way it has been reached.



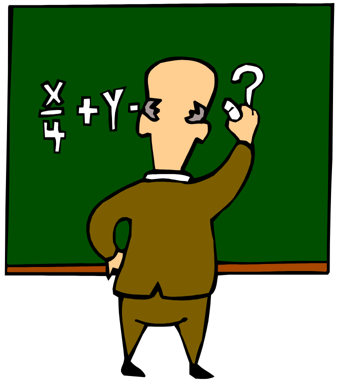
Revising history

- System under test
  – unknown, but…



- Testability Hypothesis:
  – the SUT behaves like some (unknown) FSM with the same* number of states as the description
  – Whatever the trace leading to some state s, the execution of transition *s –x:y-> s'* has the same effect (output, change of state)

*or more but this number is known

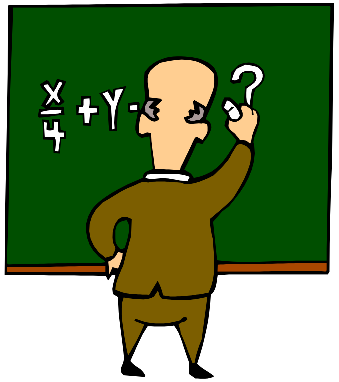


Back in history: control and observation

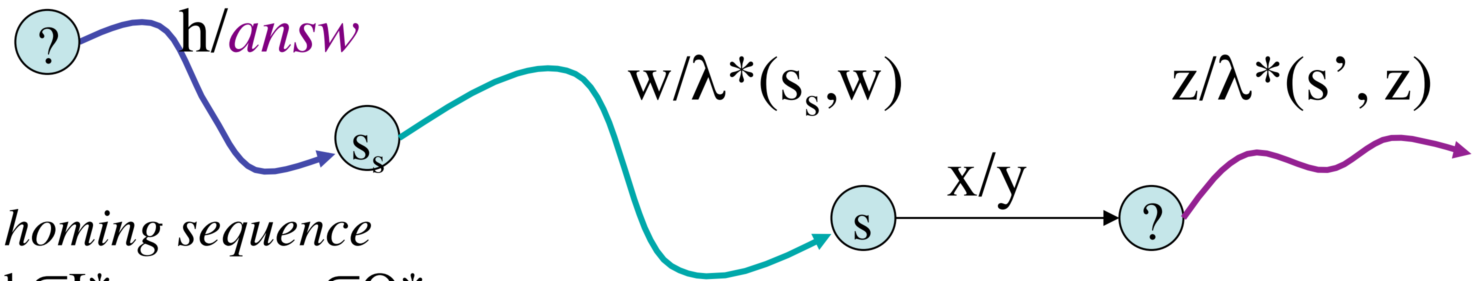
- test strategy: *transition coverage*

  s –x:y-> s'

- Questions
  - **control**: how to put the SUT into a state equivalent to s?
    - solution 1: if there is an initial state, perform a "**reliable reset**", and then some adequate input sequence
    - solution 2: "**homing sequence**", and then some adequate input sequence
  - **observation:** how to check that after receiving x and issuing y, the SUT is in a state equivalent to s'?
    - "**separating family** " : collection $\{Z_i\}_{i=1,\ldots,n}$ of sets of input sequences whose output sequences make it possible to distinguish s_i from any other state



One of the tests for $s \xrightarrow{x/y} s'$

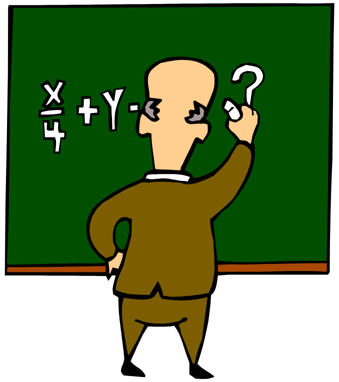


homing sequence
$h \in I^* \Rightarrow$ answ $\in O^*$
(then the SUT state should be equivalent to s_s)
Or
reset: $s_s = s_0$

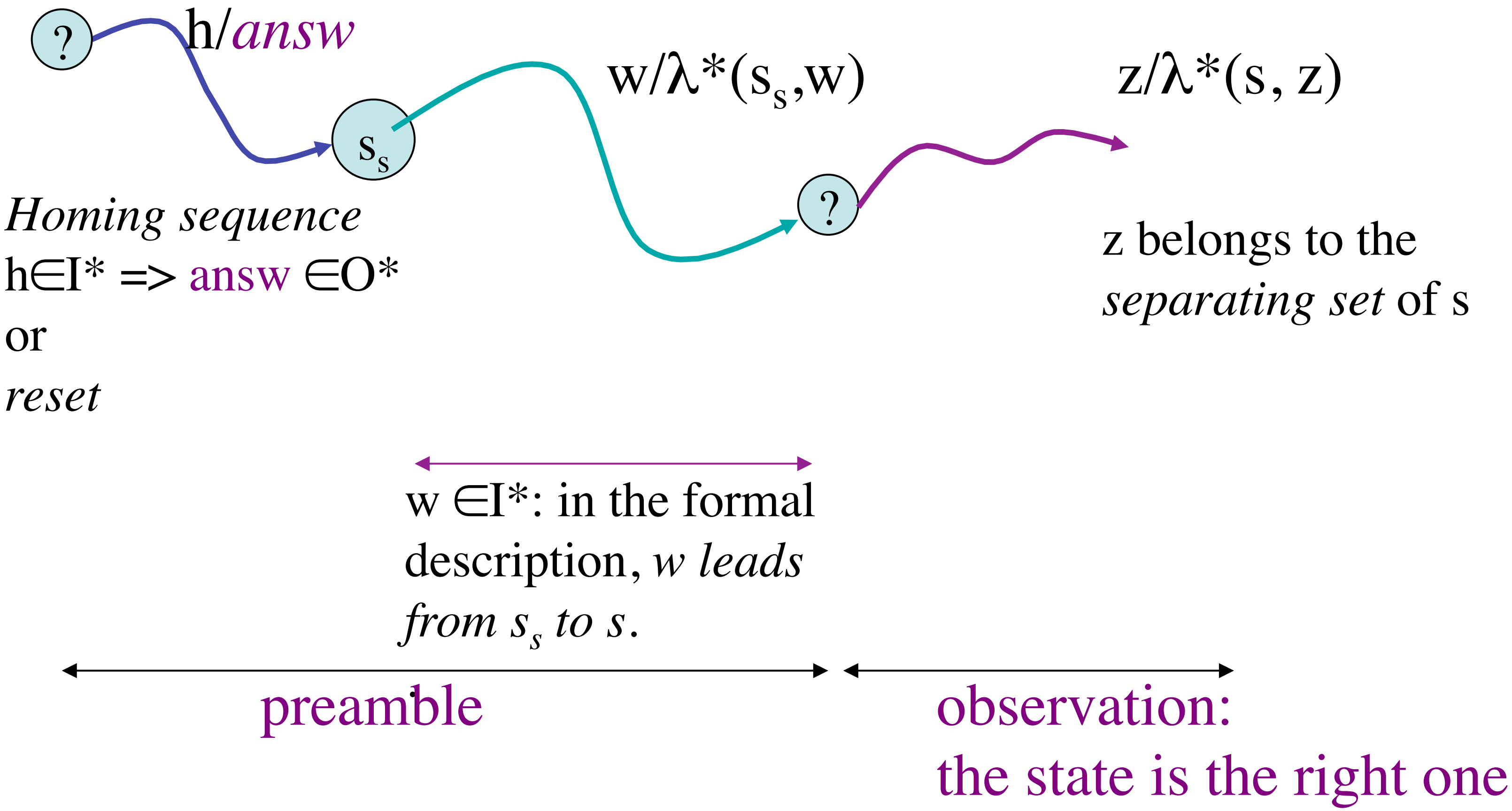
$w \in I^*$: in the formal description, *w leads from* s_s *to* s.

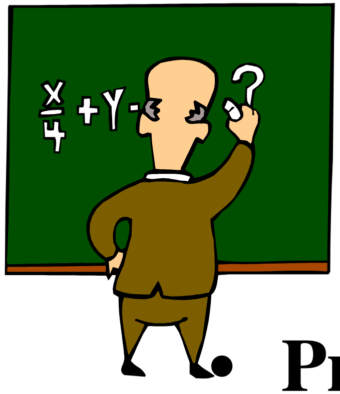
z belongs to the *separating set* of s'

preamble | transition execution | observation



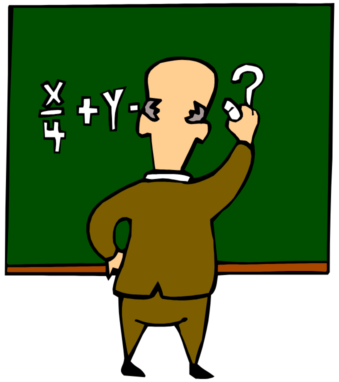
One of the tests of the preamble of s -x/y-> s’



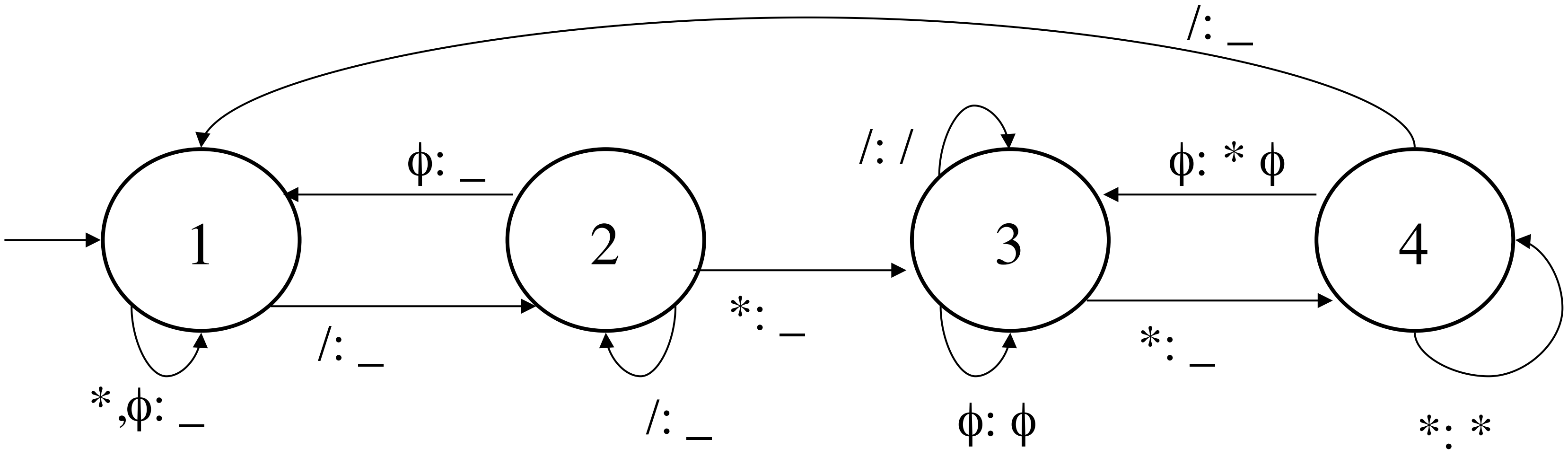


Back in history [Chow 78]

- **Preliminary theorem**: every such FSM has a *characterizing set* W= $\{w_1, \ldots, w_m\} \subseteq I^+$, which allows to distinguish the states
  - **$s \neq s' \Rightarrow \exists w_i \in W$ such that $\lambda^*(s,w_i) \neq \lambda^*(s',w_i)$**
- **Test sequences**: ***p.z***, *where $p \in P$, $z \in Z$*
  - P: for every transition s –x:y-> s', there are two sequences in P, **p and p.x**, such that p leads from the initial state to s
  - Z = W (or W^k if there are k more states)
  - i.e., coverage of transitions, with observation of the origin and destination states
- The FSM has an initial state, and the SUT provides a *reliable reset*

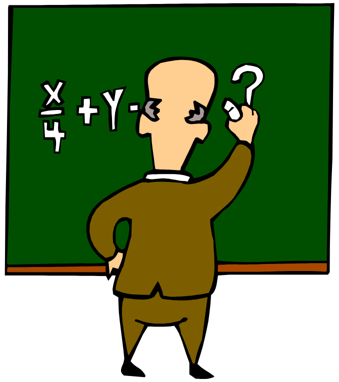


“W” for the example

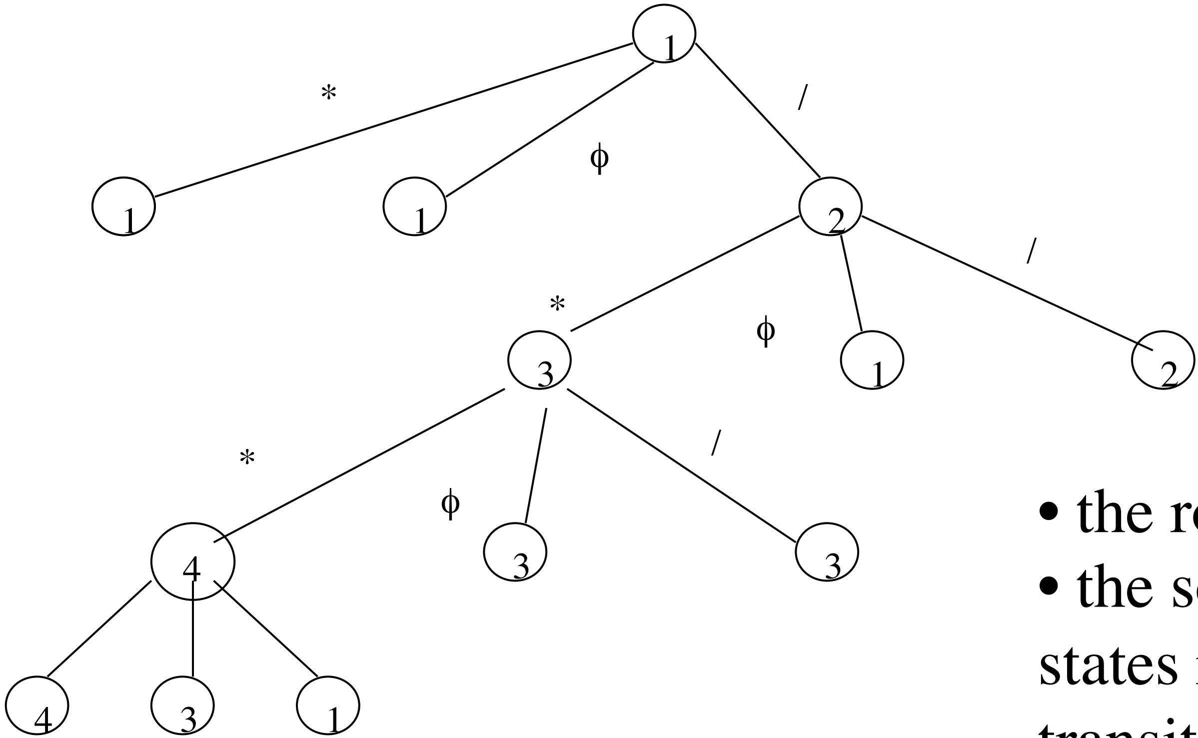


Characterizing set : **W ={*ϕ}**
Note: it is a destructive observation

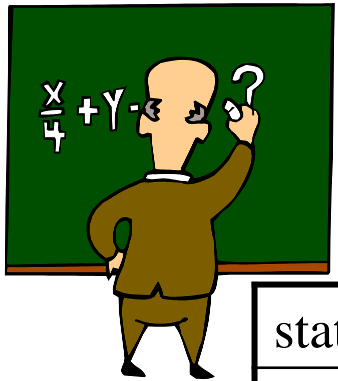
state	*ϕ
1	_
2	ϕ
3	*ϕ
4	**ϕ



Construction of some P (covering tree)



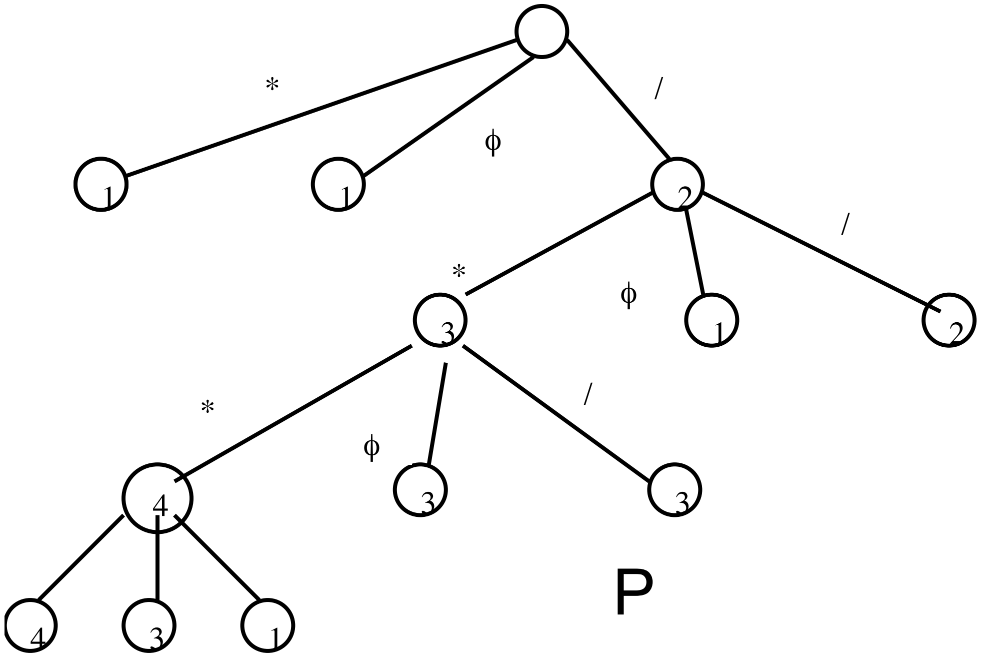
- the root is the initial state
- the sons of a node are those states reachable by some transition
- if a state is already in the tree it is terminal



Tests and expected results

state	*φ
1	_
2	φ
3	*φ
4	**φ

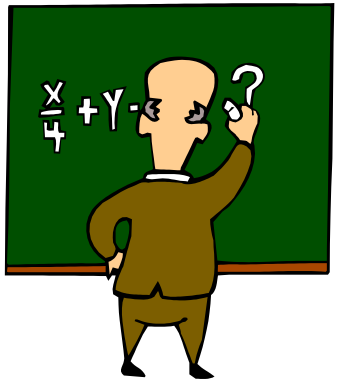
W



P

*φ	nothing
**φ	nothing
φ*φ	nothing
/*φ	φ
//*φ	φ
/φ*φ	nothing
/**φ	*φ

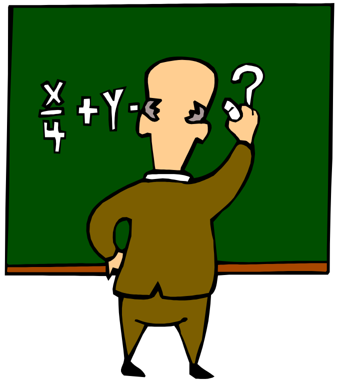
/*/*φ	/*φ
/*φ*φ	φ*φ
/***φ	**φ
/****φ	***φ
/**φ*φ	*φ*φ
/**/*φ	nothing

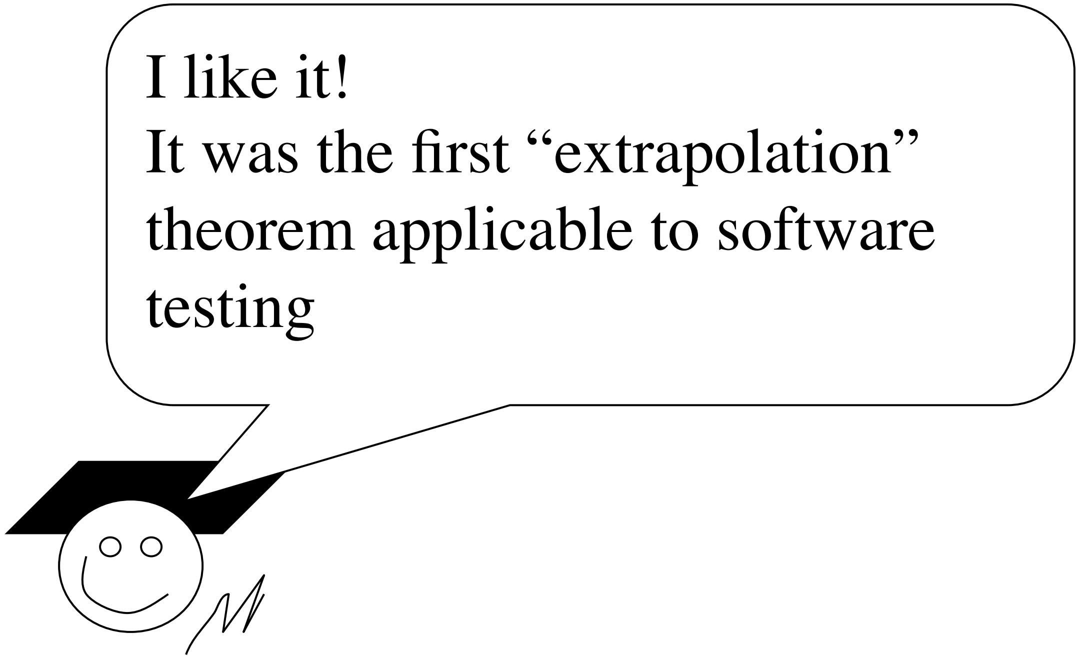


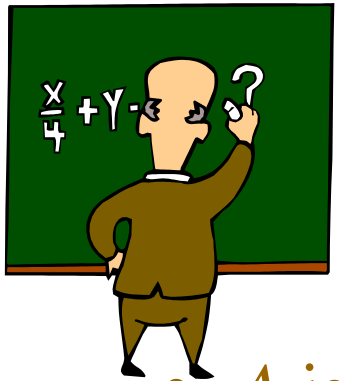
Exhaustivity of P.Z

- Let A, B, two FSM with the same I and O ;
  - Let $V \subseteq I^*$;
  - s_A is a state of A, s_B is a state of B;
  - s_A and s_B are V-equivalent iff
  - $\forall w \in V, \lambda^*(s_A,w) = \lambda^*(s_B,w)$
- A and B are V-equivalent <=> their initial states are V-equivalent
- **Chow's theorem**:

A and B are equivalent ⇔ **A and B are P.Z-equivalent**

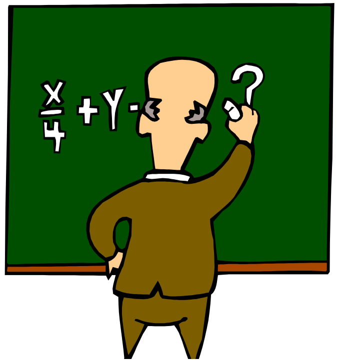



I like it!
It was the first “extrapolation” theorem applicable to software testing



Application to testing

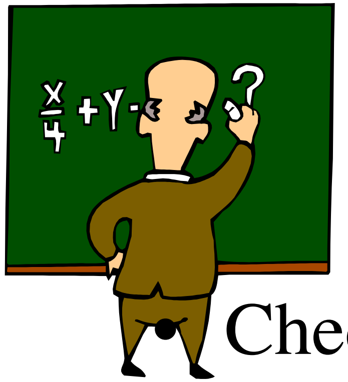
- A is a description/specification/model
- B is a system under test (SUT) that behaves like some FSM
  - One knows that I and O are the same sets for A and B.
  - One knows that B has the same number of states as A, or a known number of additional states
  - There is a reliable reset of B
  - It is all that is known about B
- From A one builds $P.Z$
  - One tests B against the sequences of $P.Z$
  - If all the output results are the same as for A, B is equivalent to A



Testability Hypotheses and all that…

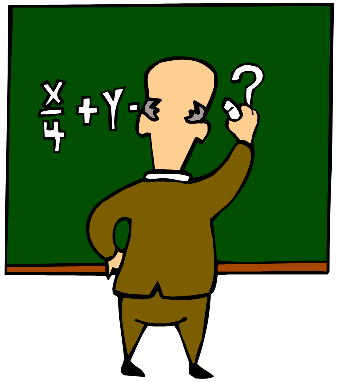
- Hypotheses:
  - *the SUT behaves like some FSM with the same* number of states as the specification*
  - *the SUT provides a reliable reset*
- Under these testability hypotheses, the success of a test set P.Z ensures equivalence of the SUT and the specification FSM
- Here the satisfaction relation is *equivalence*
- P.Z is exhaustive given these hypotheses and this relation, i.e.:
  - ***SUT behaves like some FSM with a known nb of states and it provides a reliable reset***

=> (SUT passes P.Z <=> SUT equiv SP)

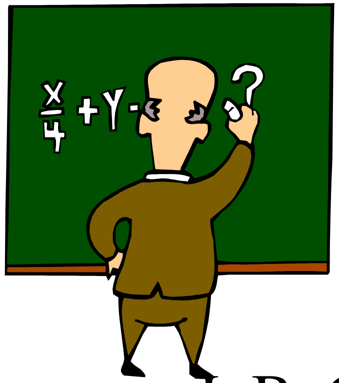


Before or after Chow

- Checking sequence:
  - covers every transition and its separating set; *distinguishes the description FSM from any other FSM with the same number of states,* no need of reset
  - Finite, *but may be exponential/nb of states*… in length, in construction
- Exhaustivity
  - Transition (+ separating set) coverage
- Control
  - homing sequence, or reliable reset
  - Non-determinism => adaptive test sequences
- Observation
  - distinguishing sets, UIO, or variants (plenty of them!)

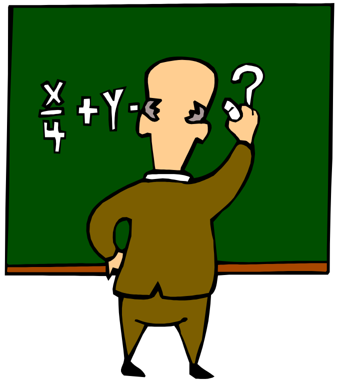


BIBLIOGRAPHY



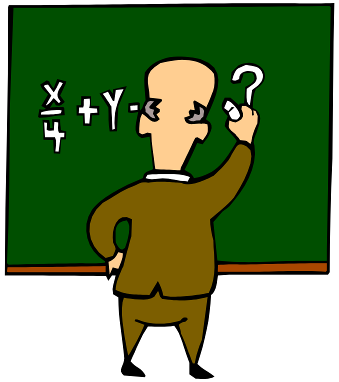
Historical Monuments

- J. B. Goodenough and S. L. Gerhart, 1975. “Toward a theory of test data selection”, *IEEE Transactions on Software Engineering*, SE-1(2): 156-173.
- T. Chow, 1978. “Testing software design modeled by finite-state machines”, *IEEE Transactions on Software Engineering*, SE-4(3):178–187.
- G. Bernot, M.-C. Gaudel, and B. Marre, 1991. “Software testing based on formal specifications: a theory and a tool”, *Software Engineering Journal*, 6(6):387-405.
- Dick J., Faivre A., 1993. “Automating the generation and sequencing of test cases from model-based specifications”, *LNCS* 670, pp. 268-284.



Recommended Surveys

- D. Lee and M. Yannakakis, 1996. “Principles and methods of testing finite state machines-a survey”, *Proceedings of the IEEE*, 84(8): 1090-1123.
- R. M. Hierons, K. Bogdanov, J. P. Bowen, et al., 2009. “Using formal specifications to support testing”, *ACM Comput. Surv*. 41(2), 76 pages.



Publications ± directly related to this tutorial

- Cavalcanti and M.-C. Gaudel, 2007. “Testing for refinement in CSP”, *LNCS* 4789, pp 151-170.
- M.-C. Gaudel and P. Le Gall, 2007. “Testing data types implementations from algebraic specifications”, *LNCS* 4949, 209-239.
- Cavalcanti and M.-C. Gaudel, 2011. “Testing for refinement in *Circus*”, *Act. Inf.*, 48(2):97-147.
- M.-C. Gaudel, 2011. “Checking models, proving programs, and testing systems”, *LNCS* 6706, pp 1-13.
- Cavalcanti, M.-C. Gaudel, 2015. “Test selection for traces refinement”, *TCS* 563:1-42.