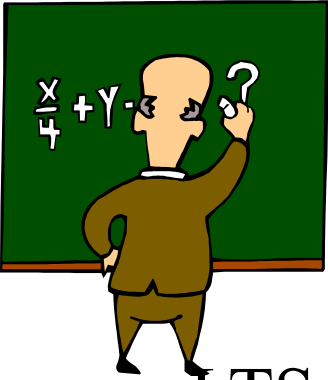


Test de Systèmes Informatiques

Partie V : Tests basé sur des modèles reactives
de comportement (Labelled Transition Systems with I/O)

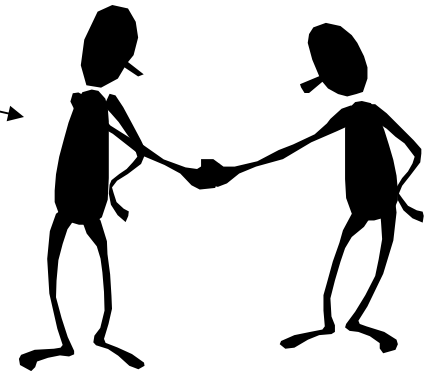
Burkhart Wolff, LRI
(basé sur Marie-Claude Gaudel)

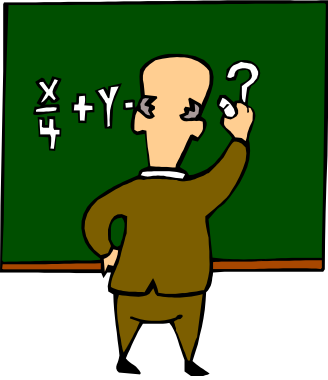


What is a LTS? (Without I/O as a beginning 😊)

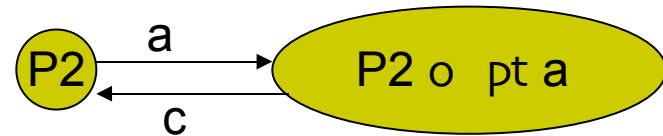
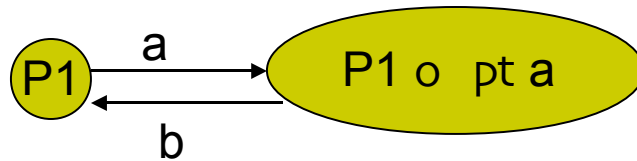


- LTS = Labelled Transition Systems: semantic basis for most process algebras (CCS, Lotos, etc)
- Emphasis on parallel compositions of processes (i.e. LTS)
 - Interleaving + “rendez-vous” model of concurrency and synchronisation
- $LTS : (S, L, T \subseteq S \times (L \cup \{i\}) \times S, s_0)$
 - S , finite set of states, with s_0 , initial state
 - L , finite set of labels (action names)
 - T , finite set of labelled transitions: $\langle s, a, s' \rangle$
 - i is a special, internal action

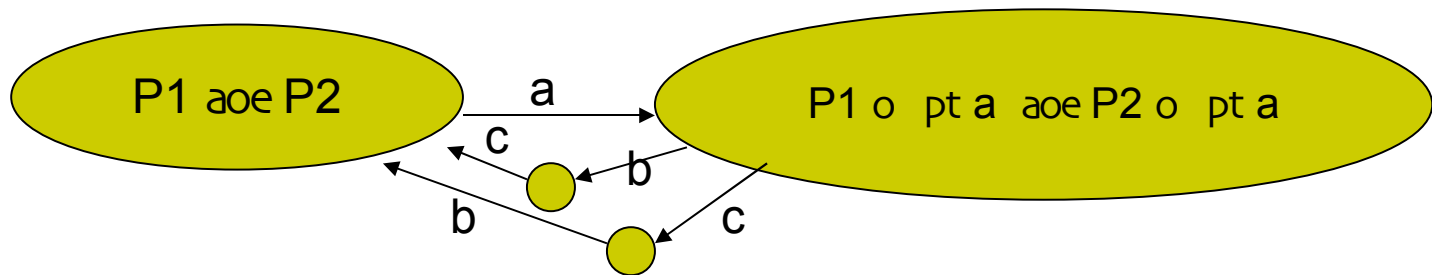




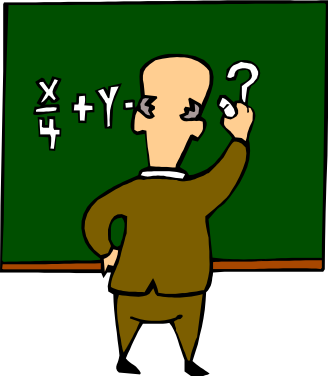
Example



Parallel composition of P1 and P2 with synchronisation on “a” :



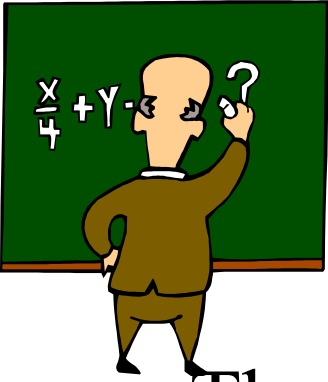
Remark: “ $P1 \parallel [a, b, c] \parallel P2$ ” would deadlock after “a”



The semantics of ! and ?



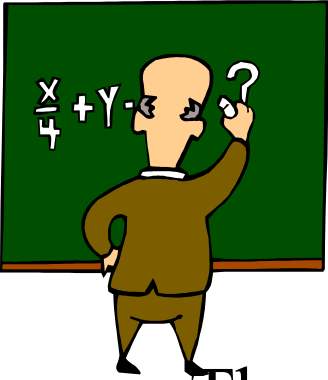
- Gate!value
 - Agreement to have “value” on gate “Gate”
- Gate? Variable:Type
 - Agreement to have any value of type “Type” on gate “Gate”
- No actual difference between output and input
 - $g?x:int$ is just a choice between $g!0, \dots g!i, \dots$



The testing problem, now



- There is a specification SP that is a LTS
- There is some SUT that, *possibly*, is an implementation of SP
- How to test the SUT to check that it satisfies (is conform to) SP ?
- How to decide whether the SUT passes the tests (beware to indeterminism!)
- What is the meaning of « is conform to » in this context?

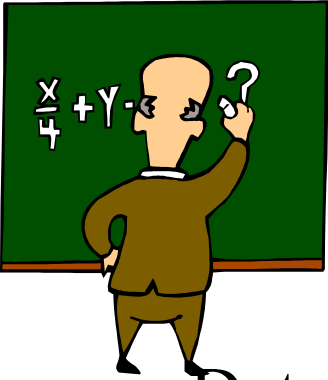


General Principles

- The SUT are tested by some « tester » processes that are built from the specification
- One runs the SUT and every tester in parallel

I || T

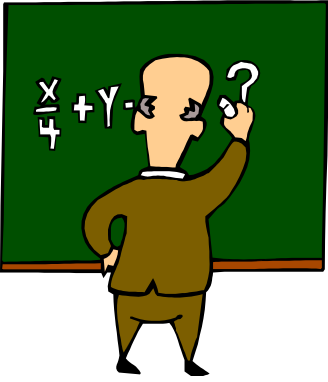
- And one observes :
 - The performed traces
 - Deadlocks (via time-out mechanisms)
- In this context,
 - test \Leftrightarrow tester specification
 - test experiment \Leftrightarrow concurrent execution of the SUT and the tester



LTS are not completely realistic but...



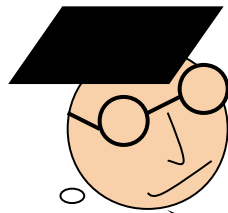
- Detection of deadlocks
- Example of a conformance relation: **the SUT must not deadlock when the specification may not deadlock**
- Example
 - An implementation of $P1 \mid [a] \mid P2$ must not refuse to perform « a » in its initial state, and after « a » it must refuse neither « b », nor « c », etc
- This relation is an instance of a “testing preorder” (Hennessy and De Nicola) between LTS. *Realistic conformance relations are not equivalences.*



Introduction of inputs \neq outputs

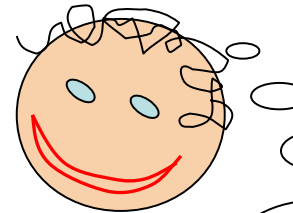


It is very simple:
One distinguishes **!a**, output of “a”,
from **?a**, input of “a”, and one
synchronises **!a** et **?a**

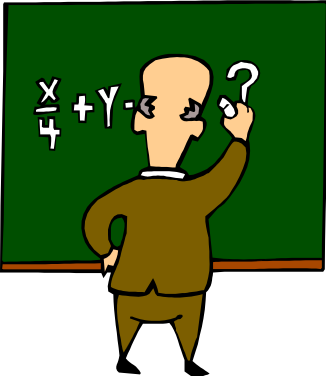


What a fool

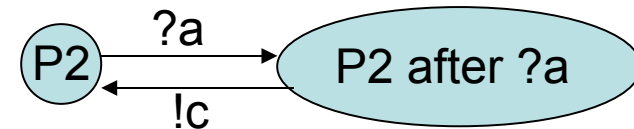
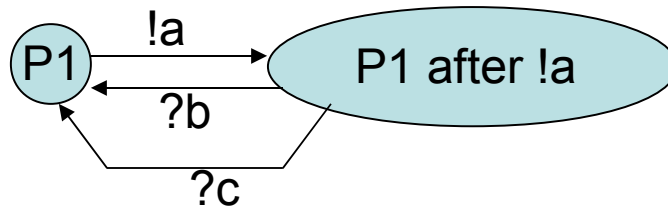
Are you sure?



What a fuss
maker



Some examples



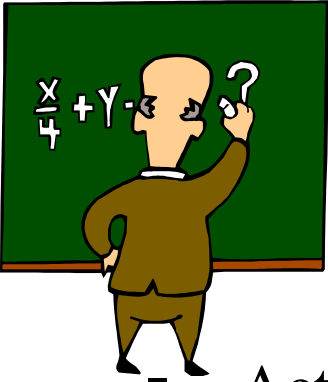
P1 || P2 behaviour :

- emission of “a” by P1, reception by P2
- emission of “c” by P2, reception by P1
- etc

First the decision is up to P1, then it is up to P2

The difference is a matter of control:

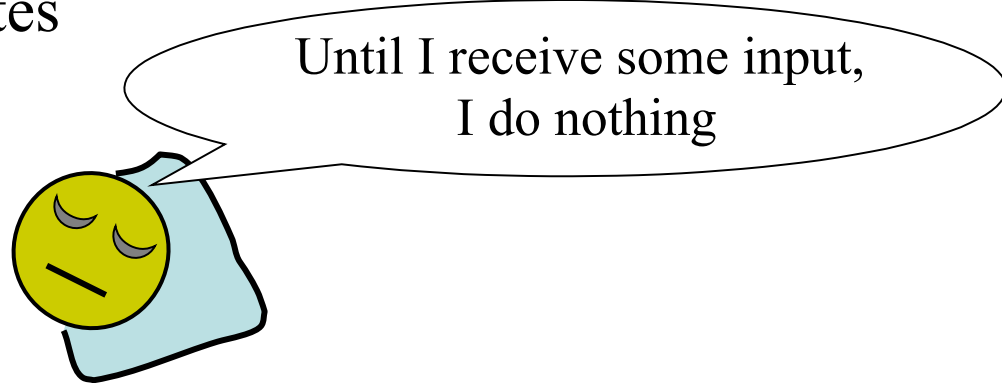
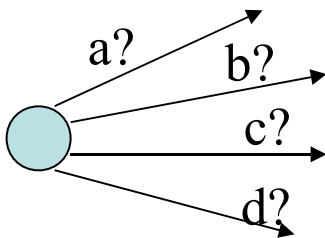
Who is the boss? 😊



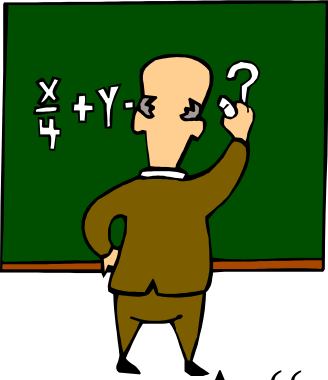
Specificities of IOTS



- Actions are divided into 2 sets: A set of input actions and a set of output actions
- Input actions synchronise with output actions and vice versa
- « Input enabled » : Inputs cannot be refused => *There is no more notion of deadlock*
- « Quiescent » states

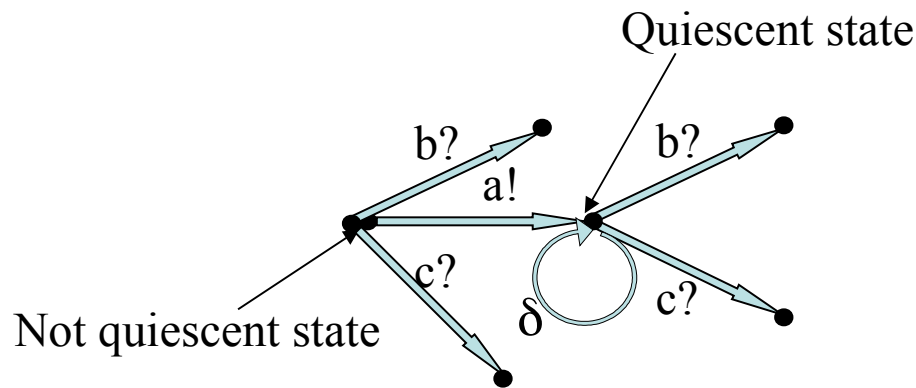


The system remains idle until some input arrives

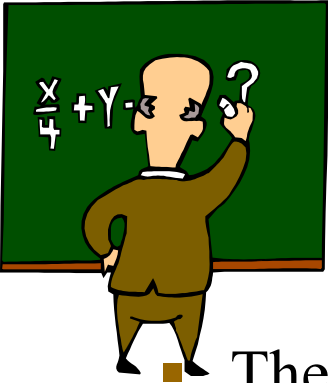


Modeling quiescence

- A “special virtual action” δ corresponds to the fact that no output action is fireable.

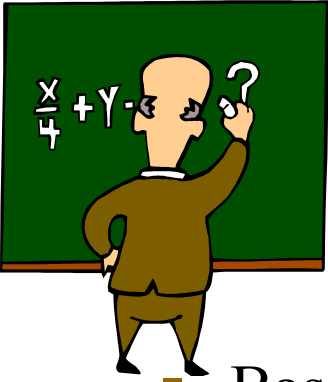


- \Rightarrow Notion of suspension traces (S-Traces):
Usual traces + traces with δ (i.e. including periods of inactivity)



What must be tested?

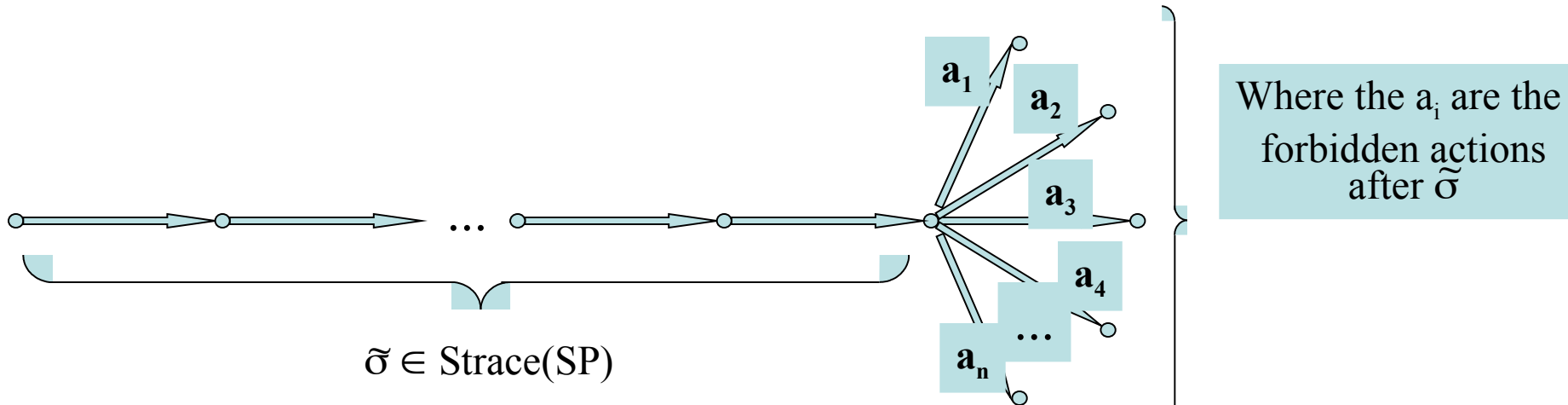
- The « ioco » conformance relation by Tretmans [55]:
 - $\forall \sigma \in \text{Straces}(SP), \text{out}(I \text{ after } \sigma) \subseteq \text{out}(SP \text{ after } \sigma)$
 - SP: specification (LTS), I: Implementation (IOTS)
- *After any S-trace of the specification, the set of output actions of the implementation must be included in the set of output actions of the specification.*
- Thus, the implementation can be more deterministic than the specification.
- **NB:** to ensure testability, the determinism of the implementation must be « balanced » : “complete testing assumption”
- To test quiescence, δ is considered as an output action. Its “mirror” input action is ... a time-out



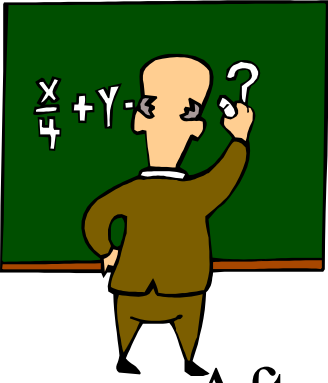
An exhaustive test set [LG 02]



- Based on *ioco* and the input enabled hypothesis, we defined the reduced test set $exhaust'_{ioco}$.



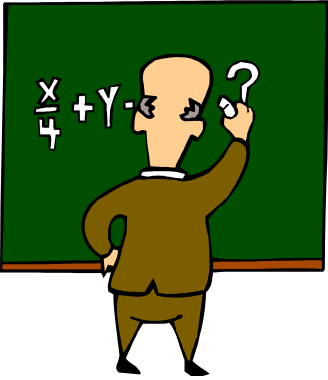
- The verdict of a test execution is: If the implementation blocks before the end of the test then *success*, else *fail*
- Or, if the implementation blocks before the end of σ , then *inconclusive*, if it blocks at the end of σ , then *success*, else *fail*



Intuition

- After every correct behaviour, the tester tries to provoke a forbidden output, (or a forbidden quiescence)
- \Rightarrow there is a failure only if such an output/quiescence takes place
- Directly inspired from the definition de *ioco*: no mandatory output/quiescence, just forbidden ones ...
- No needs to test inputs... They are enabled by hypothesis



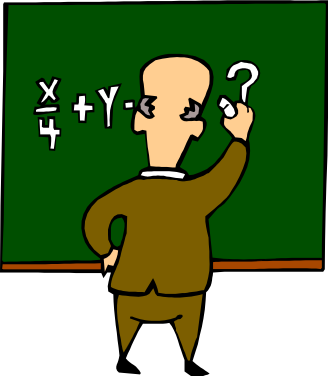


Testability hypotheses:

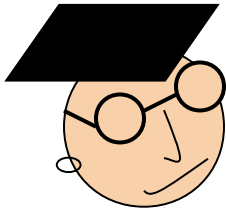
1. The SUT behaves like an IOTS with same alphabets I and O as the specification SP
 - same **atomic** actions
 - Quiescence is **detectable** by « time-out »
 - There is an upper bound $p(\sigma)$ of the **number of experiments** needed to decide whether a test based on σ is a success (complete testing assumption)
 - There is a reliable **reset** procedure for the SUT
 - Proof in [LG 02] that:

Testability Hypotheses \Rightarrow

(I passes exhaustive' $ioco$ (S) $\Leftrightarrow I ioco S$)

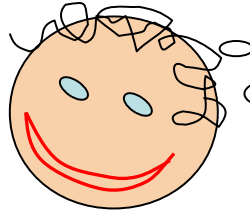


I agree: it was a bit more tricky than I expected. But everything is solved now, thanks to IOTS and ioco

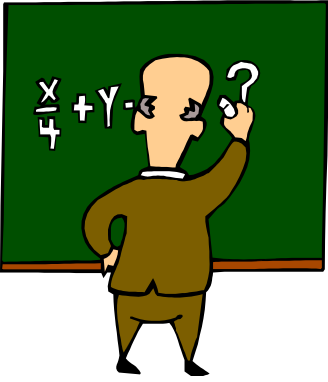


What a fool

Are you sure?



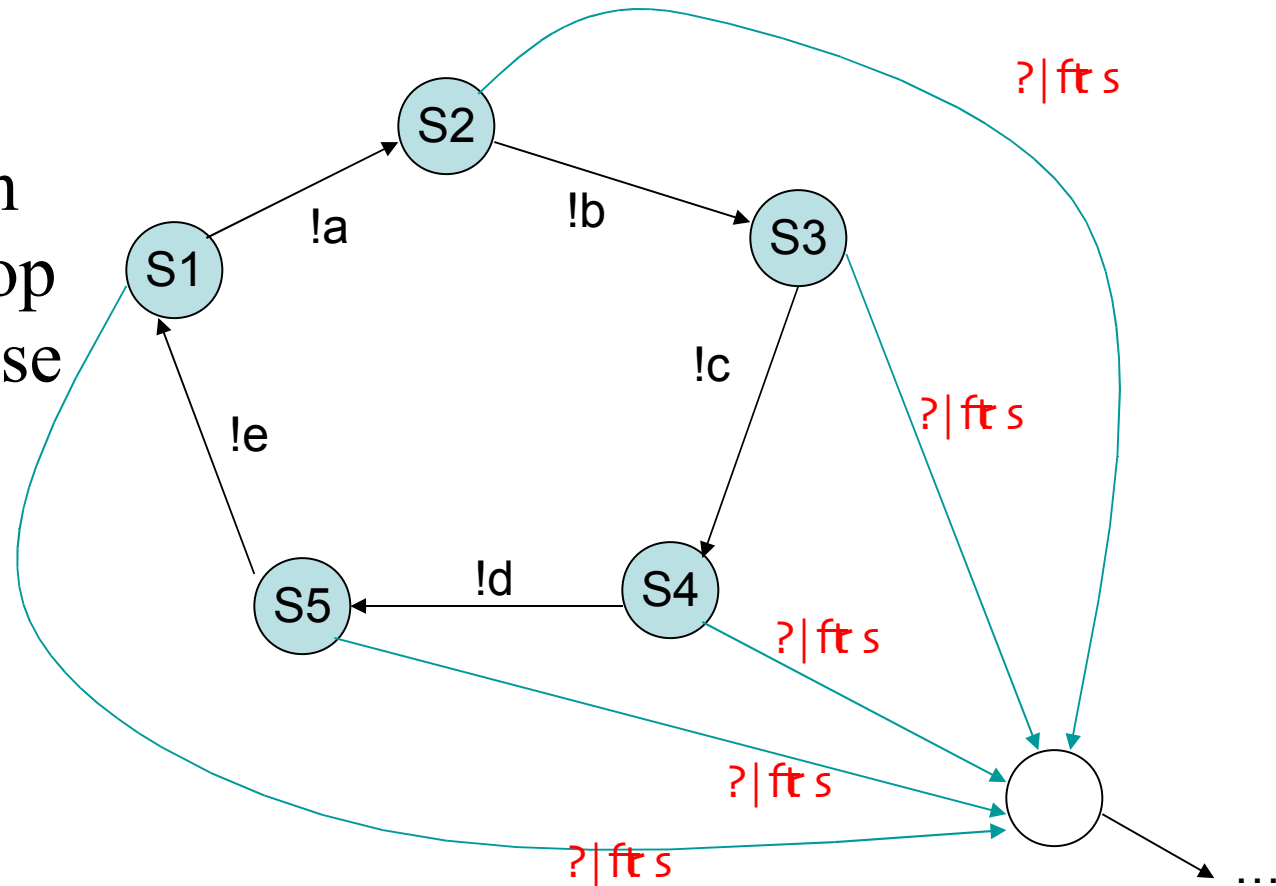
What a fuss maker



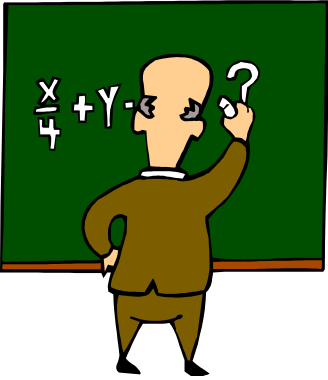
Some comments on IOTS and *ioco*



What about a screen saver that “may” stop when you want to use your computer? 😊



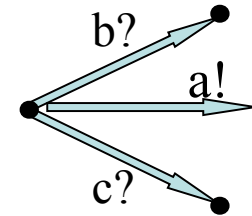
<http://www.lri.fr/Rapports-internes/2006/RR1434.pdf> (in french..., sorry)



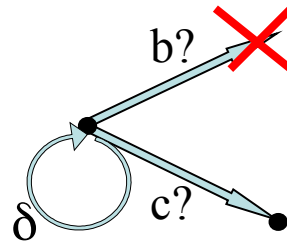
Some more comments on IOTS and *ioco*



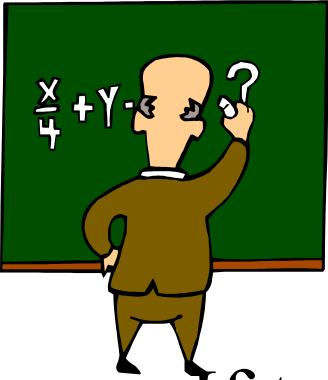
- Strong non-determinism: Inputs can be prevented by the SUT (as in the previous slide); Outputs can be prevented by the environment



- In real life, some inputs are forbidden in some states... This needs to be modelled, because it must be tested



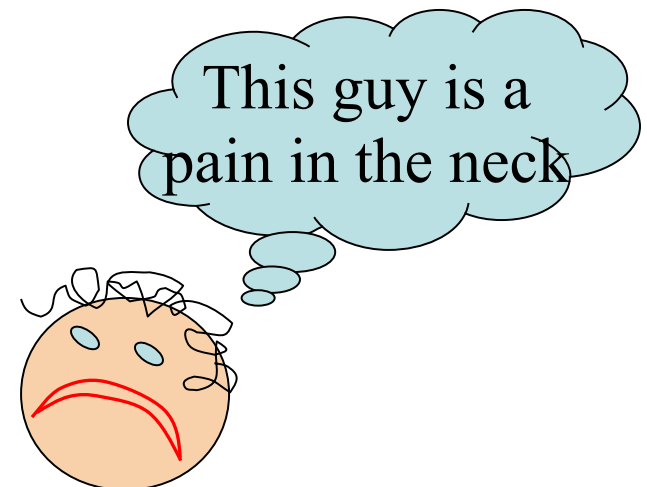
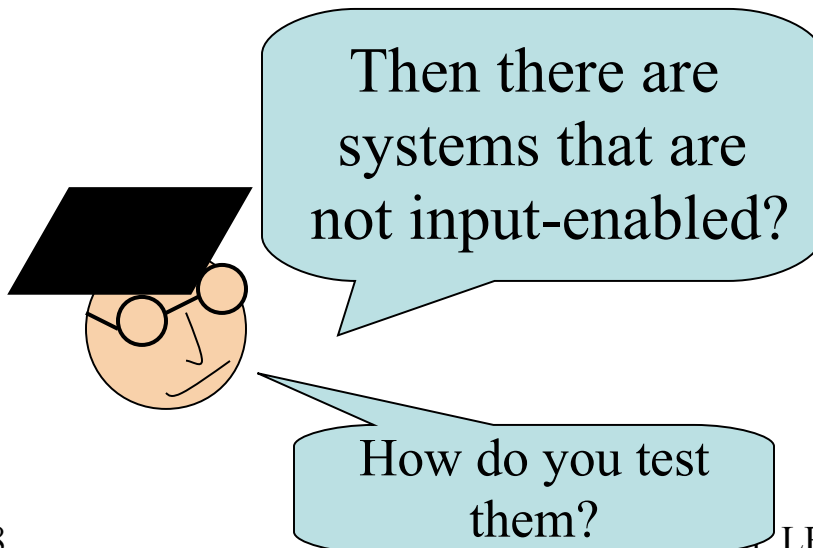
RI-OLTS
(english report available)

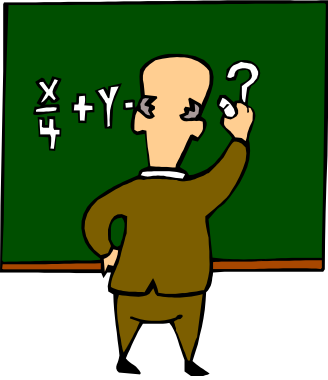


Some last(?) comments on IOTS and *ioco*



- If testers are « input-enabled », they can be controlled by the SUT...
- Current solution (*ioco*, Tretmans & C^o) : the testers are not « input-enabled » (\neq TGV)

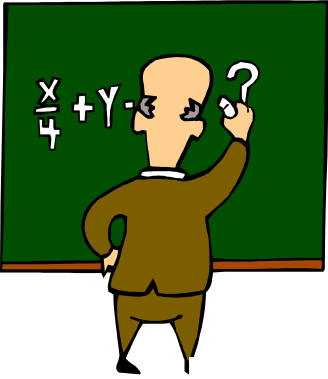




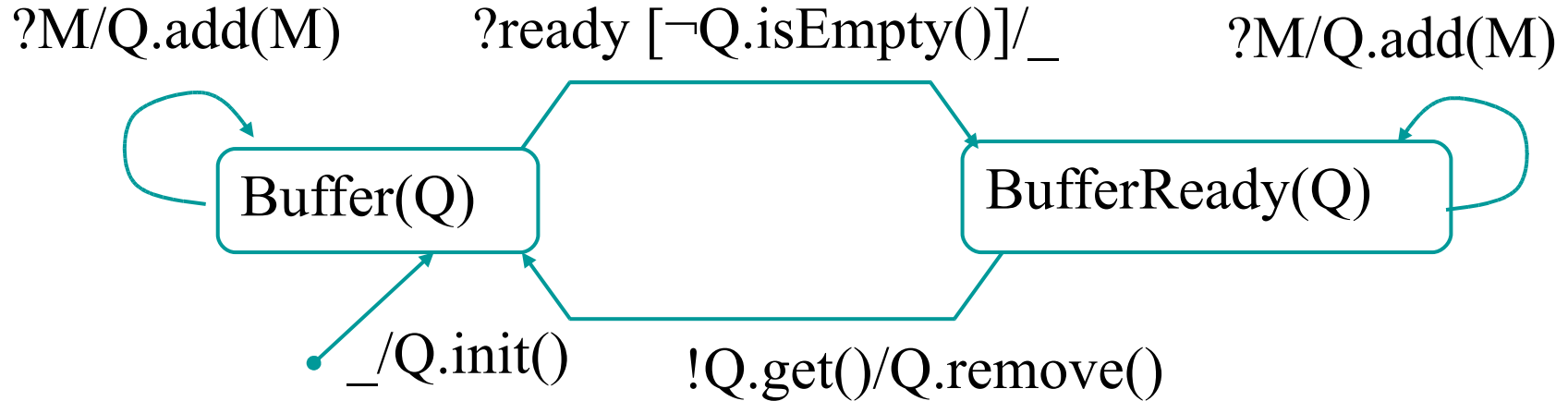
Introduction of data types



- Values and variables in the I/O actions (numbers, messages...)
- Guards (conditional actions)
- Parameterised states (by data structures)
- \Rightarrow Implementations are infinite IOTS with infinite number of actions and infinite number of states
- This leads to a test set exhaustive' _{ioco} even more infinite! 😊
- \Rightarrow Design of adequate selection methods



Buffer with priorities



M: Message, couples of text and priority

Q: Queue of Messages, with init, add, remove and get operations

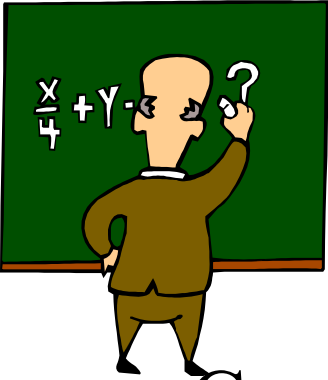
process

BufferReady[outGate, inGate](Q:Queue):=

[] outGate!get(Q);Buffer(remove(Q))

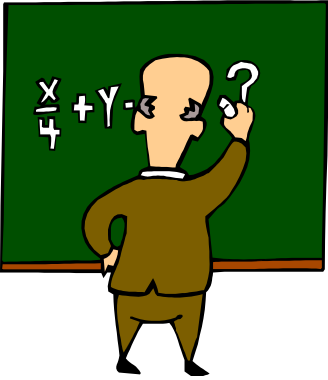
[] inGate?M:Message;BufferReady(add(M,Q))

endproc

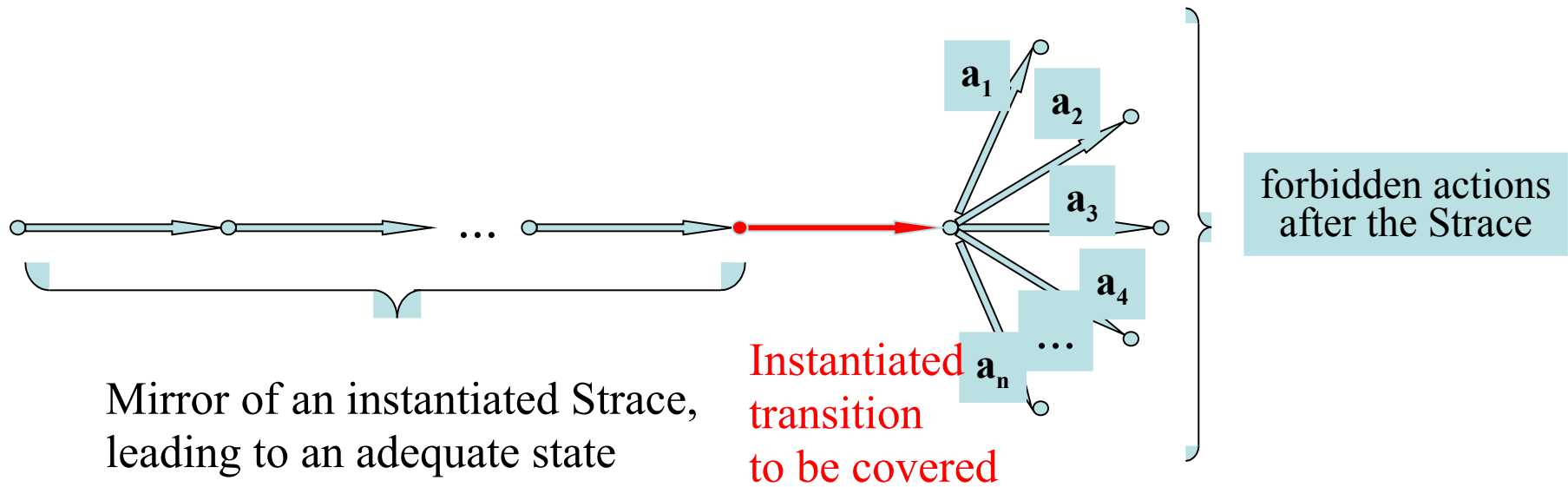


Selection

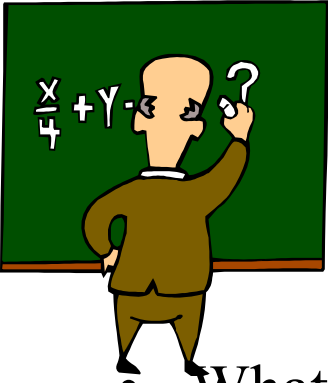
- Coverage criteria based on the symbolic description
- Choice of a subset of feasible Straces that reaches every symbolic transitions (the good old W criterion)
 - uniformity on the activation domain of the transitions
 - may be too weak: data type properties must be taken into account



A selectable test

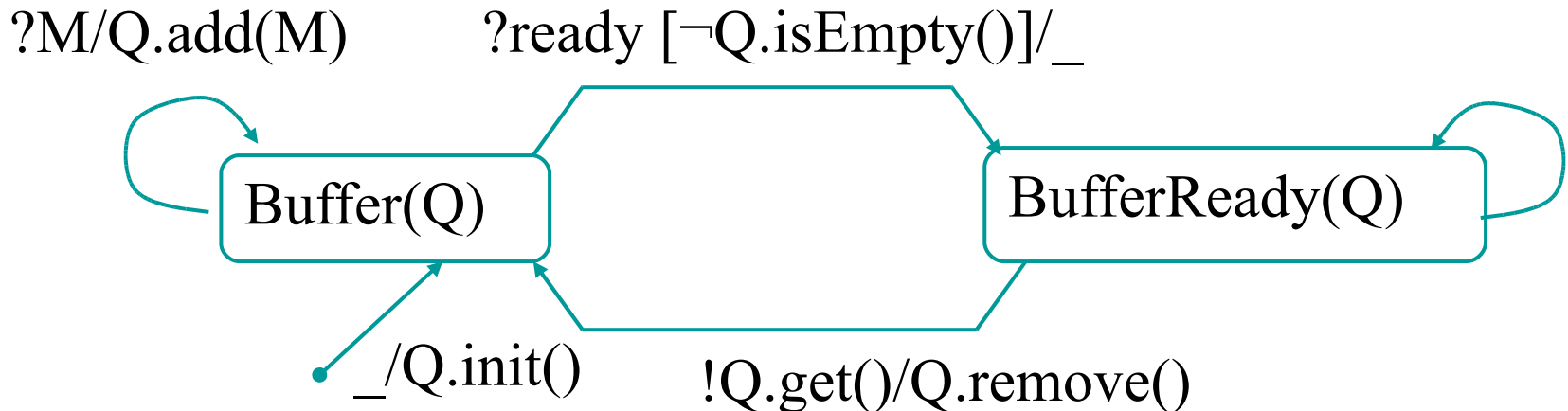


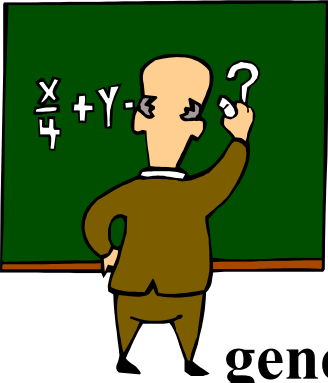
PB... struggle with a constraint solver to get an adequate feasible Strace



The example again

- What does it mean to cover:
 - $\langle \text{BufferReady}(Q), !Q.get()/Q.remove(), \text{Buffer}(Q) \rangle$
 - noted $\langle \text{BufferReady}(Q), !Q.get(), \text{Buffer}(remove(Q)) \rangle$
- Choice of any value for Q (such that the corresponding transition is fireable) \Rightarrow “weak coverage”





Properties of “get”

generated type $Queue ::= \text{init} \mid \text{add}(\text{Message} ; \text{Queue})$

ops

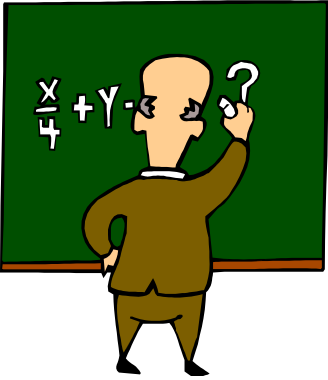
$\text{isEmpty}: \text{Queue} \rightarrow \text{Bool}$

$\text{get}: \text{Queue} \rightarrow \text{Message}$

$\text{remove}: \text{Queue} \rightarrow \text{Queue}$







$\forall M:\text{Message}; Q:\text{Queue}$

- $\text{get}(\text{init}) = (0.<>)$;
- $\text{get}(\text{add}(M, \text{init})) = M$;
- $\text{isEmpty}(Q) = \text{false} \wedge \text{priority}(\text{get}(Q)) \geq \text{priority}(M) \Rightarrow \text{get}(\text{add}(M, Q)) = \text{get}(Q)$;
- $\text{isEmpty}(Q) = \text{false} \wedge \text{priority}(\text{get}(Q)) < \text{priority}(M) \Rightarrow \text{get}(\text{add}(M, Q)) = M$; ...

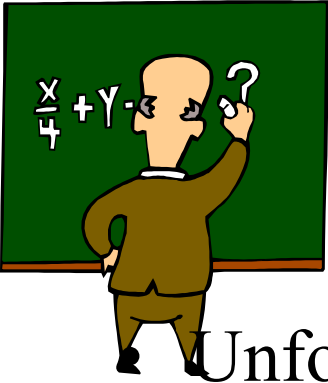


Intuition

This specification defines the operations *get* and *remove* according to the following 4 cases:

- [-] : The queue is empty
- [] : The queue contains only 1 message
- [ |  | ] : There are at least 2 messages and the last added one has not the highest priority
- [ | ] : There are at least 2 messages and the last added one has the highest priority

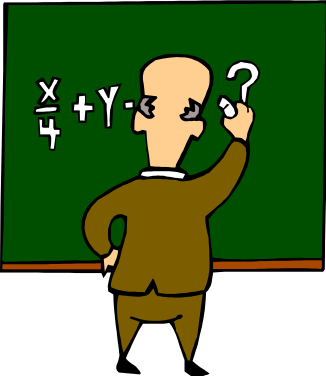
These 4 cases will serve as basis for unfolding the symbolic transition.



Unfolding symbolic transitions

• Unfolding *get...* \Rightarrow 3 feasible instances

- ~~$\langle \text{BufferReady}(\text{init}), (0, \langle \rangle), \text{Buffer}(\text{init}) \rangle$~~
- ~~$\langle \text{BufferReady}(\text{add}(M, \text{init})), !M, \text{Buffer}(\text{init}) \rangle$~~
- $\langle \text{BufferReady}(\text{add}(M, Q)), !\text{get}(Q), \text{Buffer}(\text{remove}(\text{add}(M, Q))) \rangle$
 - when $\text{isEmpty}(Q) = \text{false} \wedge \text{priority}(\text{get}(Q)) \geq \text{priority}(M)$
- $\langle \text{Buffer}(\text{add}(M, Q)), !M, \text{Buffer}(Q) \rangle$
 - when $\text{isEmpty}(Q) = \text{false} \wedge \text{priority}(\text{get}(Q)) < \text{priority}(M)$
- Unfolding \geq in the 3rd sub-case \Rightarrow 2 sub-sub-cases
 - $!\text{get}(Q)$ when $\text{priority}(\text{get}(Q)) = \text{priority}(M)$
 - $!\text{get}(Q)$ when $\text{priority}(\text{get}(Q)) > \text{priority}(M)$
- ***Stronger coverage: the sub-cases hidden in the event, guard, and actions are covered***



Conclusion



- Generic framework/formal specification methods
- **Semantics matters**
- **Satisfaction/conformance relation is basic**
- Both are used to
 - Define *tests*
 - Define *test experiments* and *verdicts*
 - Define *exhaustiveness* and the class of *testable* systems
- Proof system or verification tools (model checker) are used to
 - Optimise tests and test sets
 - Implement *selection strategies*