FORMAL CONCEPTS OF IMPLEMENTATION AND TESTING

VIEWS OF IMPLEMENTATION

(Brinksma, Scollo, van Steenbergen)

A) *implementation* as a synonym of the **real/physical system** that is the subject of conformance requirements and conformance testing [view not to be dealt with further - **not formalizable**].

B) *implementation* as a (deterministic) **reduction** of a given specification. In this context specification and implementation are relative notions in a hierarchy of system descriptions, where one description is viewed as an implementation of another description, the specification, if the former results from the latter by **resolving choices that were left open in that specification** (consider *reduction of non-determinism*).

C) *implementation* as an **extension** of a given specification. Again specification and implementation are to be regarded as relative notions in a hierarchy. An implementation **adds information that is consistent with the original specification**. Unlike refinement however, (proper) extension involves additional information about the observable behavior of the system described. This notion is especially relevant in the context of partial specification (consider *resolution of deadlocks*).

D) *implementation* as a **refinement** of a given specification. Also working with relative notions of implementation and specification, in this case an implementation provides more detail on the subdivision of the specification itself into smaller components. Both descriptions are <u>extensionally</u> equivalent, i.e. their observable behavior cannot be distinguished. The <u>intension</u> of both descriptions is not the same, as the implementation gives more details about the internal structure of the object of specification (see **style transformations** later in course).

Testing: Checking correspondence
implementation $\leftrightarrow$ specification

Black-Box Testing vs. White-Box Testing:

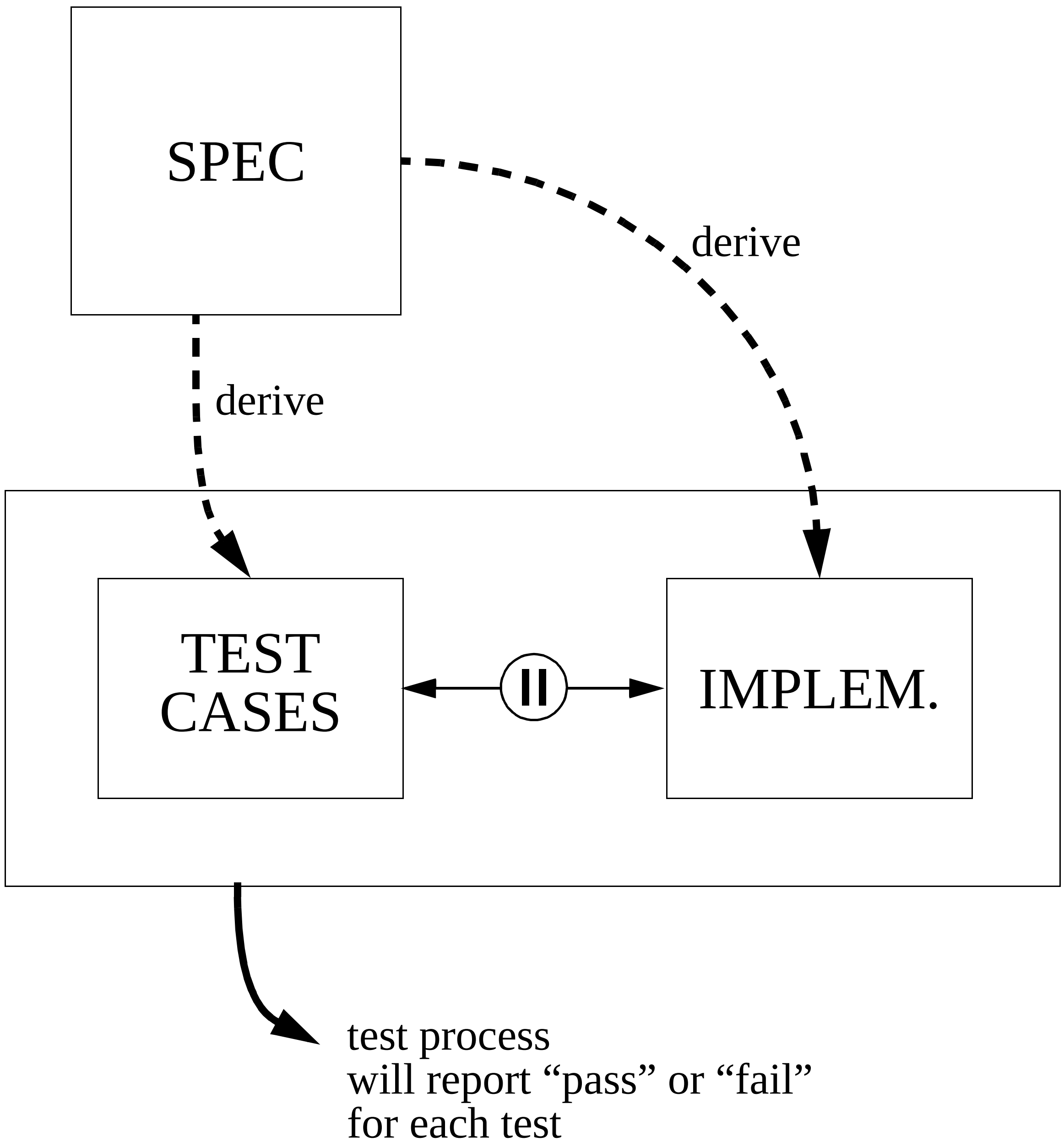
In **Black-Box Testing**, one cannot assume any information on the structure of the system being tested (one works on the basis of the specification, which could have different structure from implementation). In the case of protocol standards, this is called *Conformance Testing*.

In **White-Box Testing**, one can assume complete information of the internal structure of the system being tested.

We deal with Black-Box Testing.

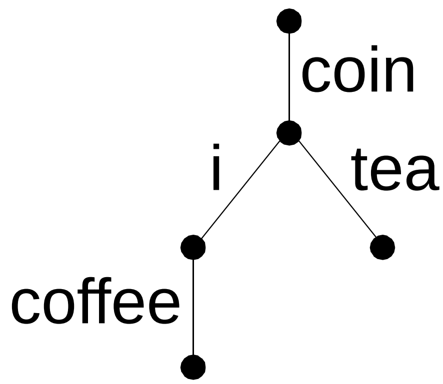
Note that (in general) no finite number of finite tests applied on a black box will be able to ensure that the box *implements* the specification (this is a well-known results from automata theory).

SYNCHRONOUS TESTING



Note however that much real testing is asynchronous, i.e. there are queues between tester and implementation.

The basic concepts of implementation and testing of non-deterministic processes go back to C.A.R. Hoare



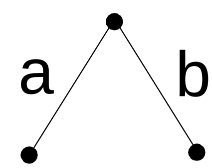
This specifies a machine that may or may not give you **tea**, but will always give you **coffee**.

Thus the **tea** part is optional and does not need to be implemented.

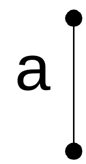
∴ A tester for this machine won't report failure if **tea** is not implemented.

Example:

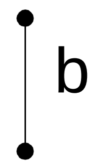
Suppose we want to test an implementation of:



We need two testers:

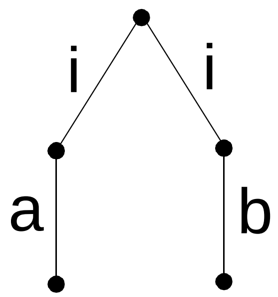


and



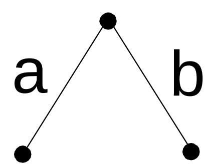
Successive runs of the implementation must satisfy both.

Non-Deterministic Example:



The tested entity could go either way!

The tester should be prepared for either possibility at each run, so tester is:



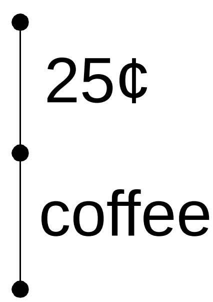
This example shows that, in the case of nondeterministic implementations, even repeated runs of the test may not be able to determine conformance.

Note that applying a and b by themselves in turn will prove nothing because the tester can always decide to go the other way.

Testing nondeterministic processes is practicallly infeasible because it is the nondeterministic process that has the initiative and can refuse what the tester proposes.

Note also that, strictly speaking, it is not sufficient to test *acceptance*; it is also necessary to test *refusals* (*robustness* testing)

E.g. for LTS



It is not sufficient to test that it accepts

25¢; coffee

it should also be tested that it refuses **10¢** at beginning, **25¢** after **25¢**, everything after **coffee**.

The testing theory presented here does not consider robustness tests.

Basic Notation

$$P = \sigma \Rightarrow \quad =\text{def} \quad \exists\, P' \,|\, P = \sigma \Rightarrow P'$$

$$P = a \Rightarrow \quad = \quad \exists P' \,|\, P = a \Rightarrow P'$$

$$P \; after \; \sigma \quad = \quad \{P' \,|\, P = \sigma \Rightarrow P'\}$$

$$Tr(P) \quad = \quad \{\sigma \,|\, P = \sigma \Rightarrow \;\}$$

Where P, P' are behaviors, $a \in L$, $\sigma \in L^*$

P, Q trace equivalent: Tr(P) = Tr(Q)

The empty set $\emptyset$ will be written {}

Refusal sets *are sets of sets of actions*

$\mathrm{Ref}(P,\sigma) =_{def}$

$\{X \subseteq L \mid \exists\, P' \in P \text{ after } \sigma,$

$\quad \text{such that } \neg(P' \overset{a}{\Rightarrow}),\ \forall\, a \in X\}$

A set $X \subseteq L$ belongs to $\mathrm{Ref}(P,\sigma)$ iff P may engage in trace σ and, after doing so, refuse every event in set X

Understand well this definition!

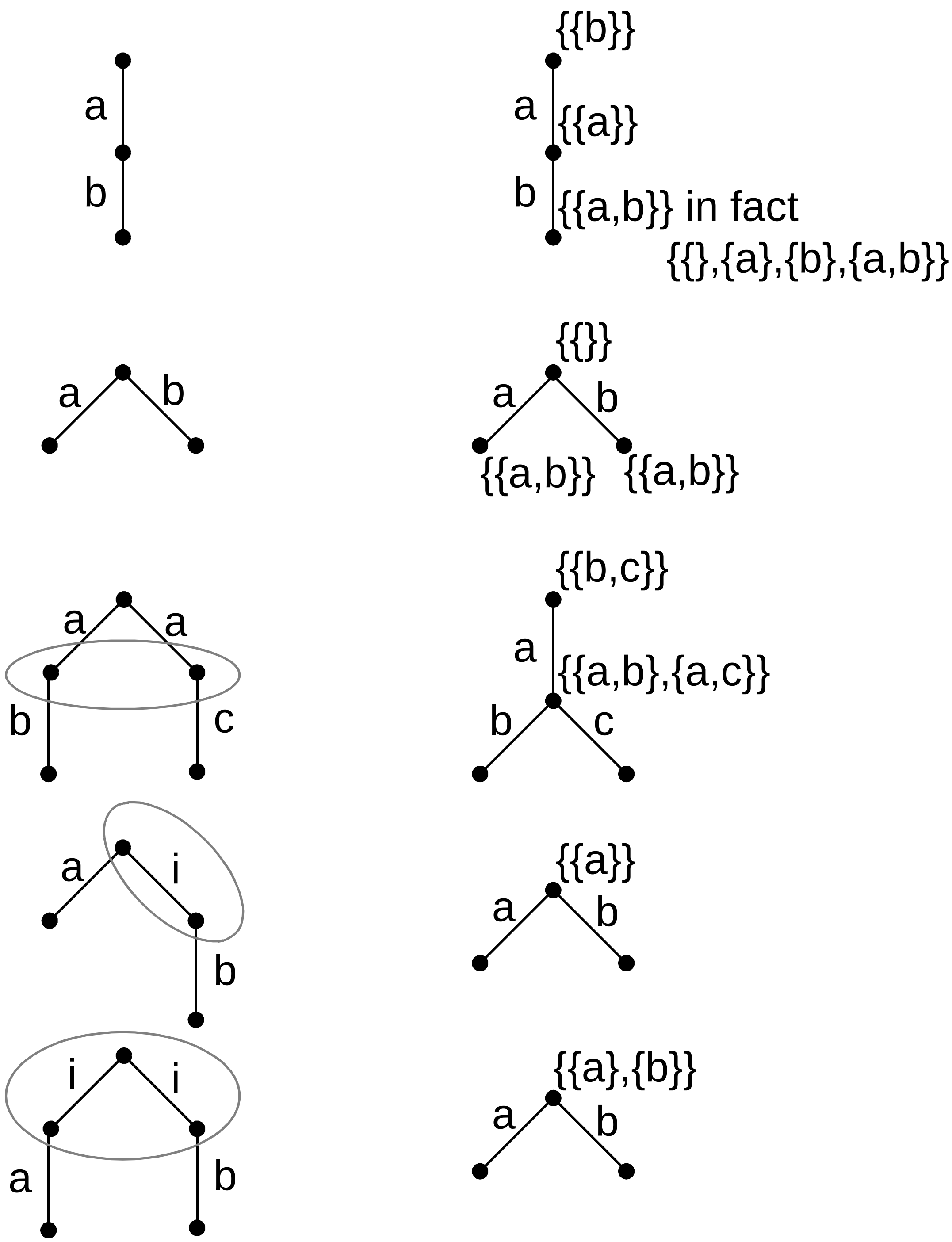
Note: $\mathrm{Ref}(P,\sigma) = \{\}$ if $\sigma \notin \mathrm{Tr}(P)$

Observe difference w.r.t. $\mathrm{Ref}(P,\sigma) = \{\{\}\}$

(nothing is refused)

Refusal Trees:

LTS REF.TREE



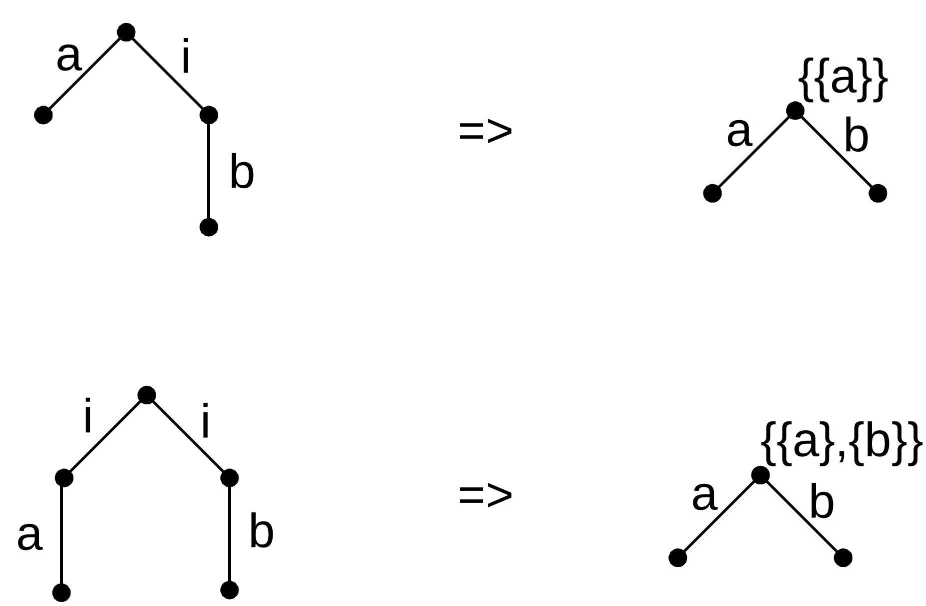
Note: all subsets are included (e.g. {}) but not shown usually

Notes on refusal trees

An action can show *both* in the refusal set for a node, *and* in the label of an edge outgoing that node.

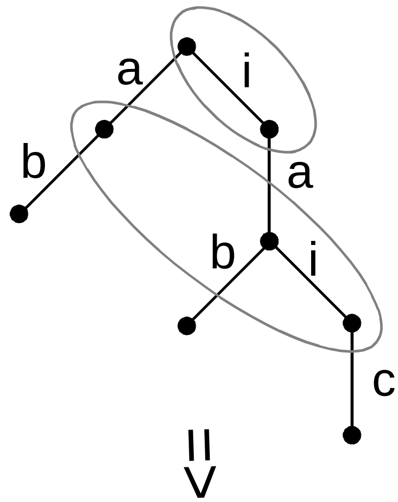
This means that the action *can be* refused, depending on a non-deterministic choice

BEH. TREE REF. TREE



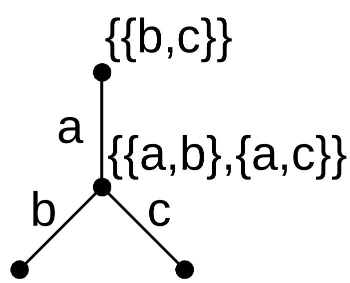
Note that this implies that a behavior tree can be obtained from a refusal tree.

more complicated example



S

L = {a,b,c}



←Note that b and c can be both refused and accepted. Not so a.

Ref(S,ε) = {{},{b},{c},{b,c}}
Ref(S,a) = {{},{a},{b},{c},{a,b},{a,c}}

For simplicity, only maximal sets are shown usually, so we can say

Ref(S,ε) = {{b,c}}
Ref(S,a) = {{a,b},{a,c}}

The ***conformance*** relation tries to capture the concept of *implementation* of a specification

- An implementation may not involve some *options* (actions which can be refused) so it can be a ***reduction*** of a specification

  e.g. a specification of a coffee machine that may or may not give tea can be implemented by omitting tea altogether: reduction

- An implementation can resolve some deadlocks, by adding transitions (***extension***)

  e.g. a specification of a phone system may state that it is not possible to dial before offhook
  (->refusal = deadlock)
  an implementation may resolve the deadlock by saying that such a dial will cause the system to return to initial state: extension.
  (deadlock as abstraction from implementation details)

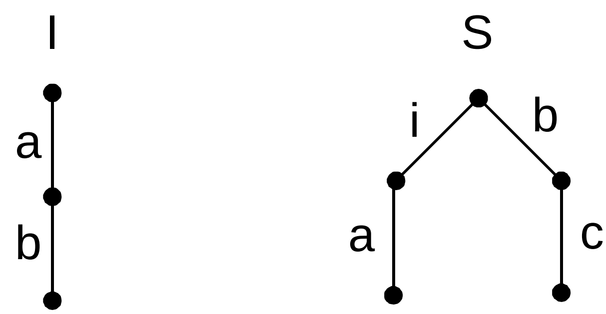
The *conformance* relation

I conf S $=_{def}$ $\forall\ \sigma \in$ Tr(I) $\cap$ Tr(S) (for all common traces σ)

Ref(I, σ) $\subseteq$ Ref(S, σ) (not symmetric)

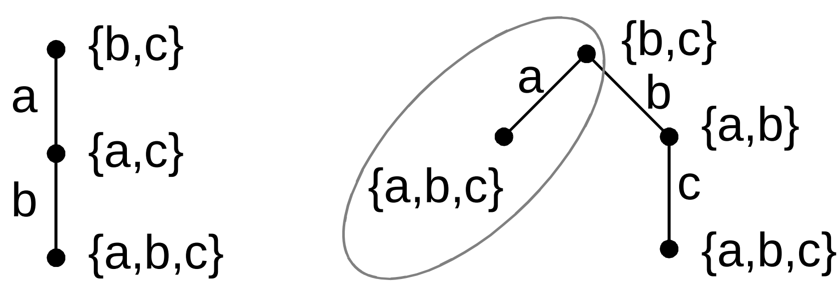
so I conf S iff, placed in an environment whose traces are limited to those common to I and S, I can only deadlock when S can. (I will deadlock *in fewer cases*)

e.g.



L = {a,b,c}

Refusal trees:



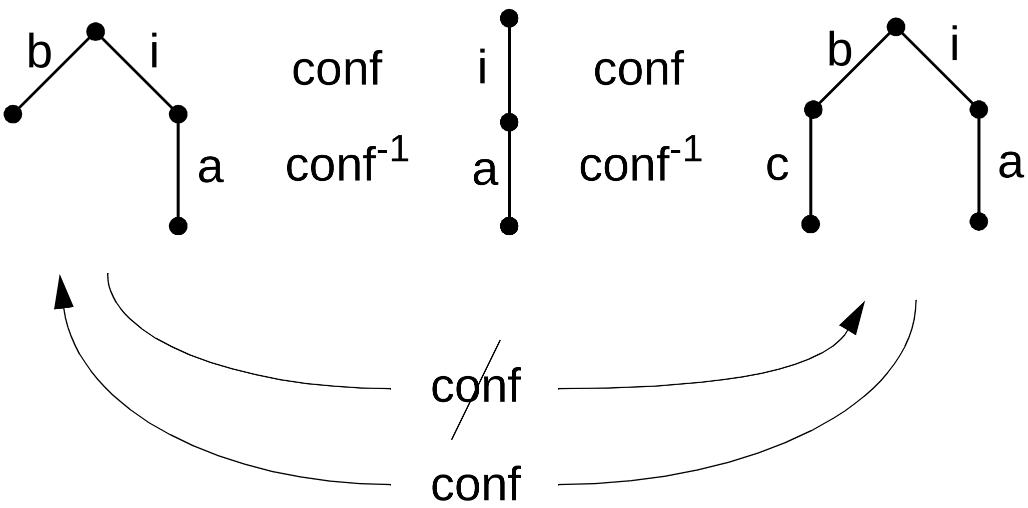
Tr(I) $\cap$ Tr(S) = {ε, a}

Ref(I,ε) = {b,c} $\subseteq$ Ref(S,ε) = {b,c}

Ref(I,a) = {a,c} $\subseteq$ Ref(S,a) = {a,b,c}

so I conf S

The conformance relation is unfortunately not transitive - nor symmetric
(it is not an equivalence, nor a preorder)



check these!

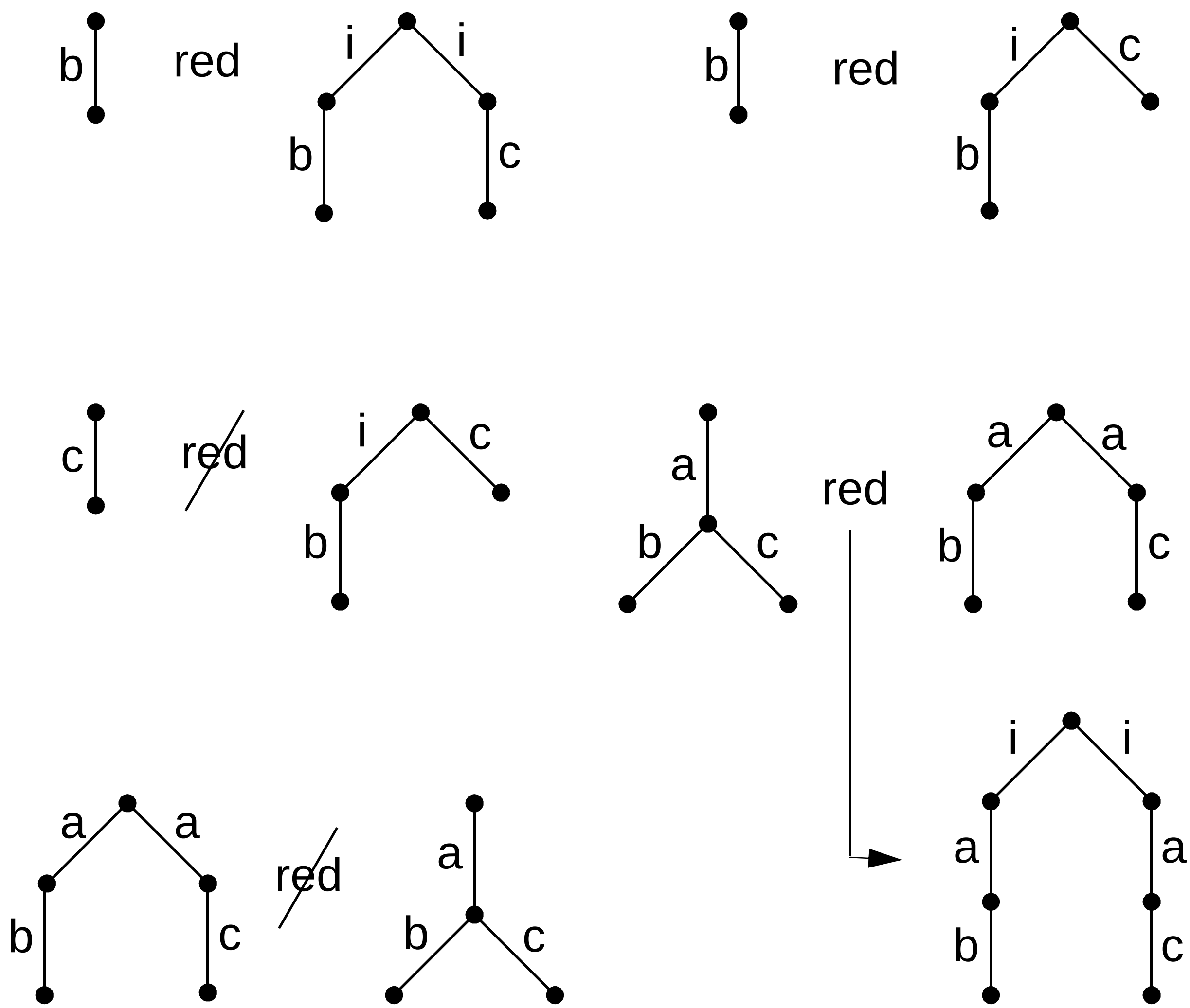
conf becomes transitive for sets of processes with the same tace sets.

The *reduction* relation

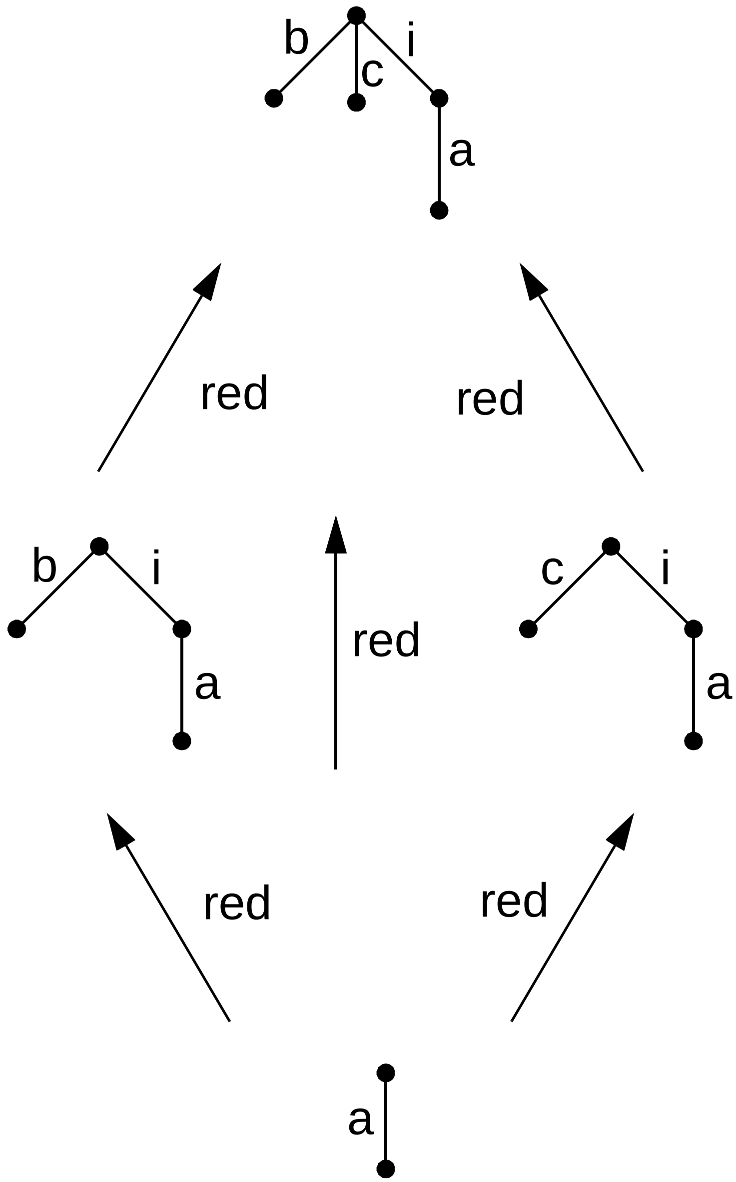
I **red** S iff

(i) Tr(I) $\subseteq$ Tr(S)

(ii) I conf S



Hierarchies of reductions



b and c are *options*

The *extension* relation

I **ext** S iff

(i) $Tr(I) \supseteq Tr(S)$

(ii) I conf S

Theorem: conf = red ext

(a conf is the red of an ext)

There are interesting relationships between these relations, e.g.

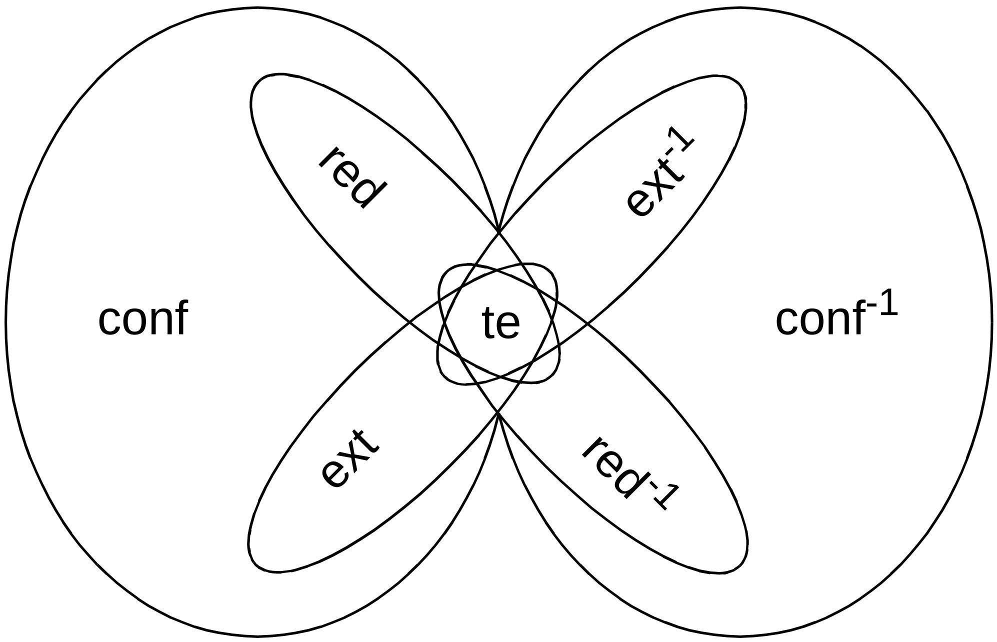
red = ext = conf
    for processes with equal trace sets

P1 **te** P2 iff P1 red P2 and P2 red P1
P1 ext P2 and P2 ext P1
P1 conf P2 and P2 conf P1 (*)

$$\begin{aligned} te &= red \cap red^{-1} \\ &= ext \cap ext^{-1} \\ &= conf \cap conf^{-1} \text{ (*)} \end{aligned}$$

(*) true only for processes with equal trace sets, because otherwise conf is not transitive

The LOTOS Flower



(Note: true for conf only for processes with equal trace sets) red and ext are reflexive, transitive

Also: $\approx \subset$ te

i.e. weak bisim. eq. implies testing eq..

($\therefore$ non te implies non wbe)

(try to prove this...)

Canonical Tester (E. Brinksma)

T is a **canonical tester** of S iff (def.):

(i) $Tr(T) = Tr(S)$

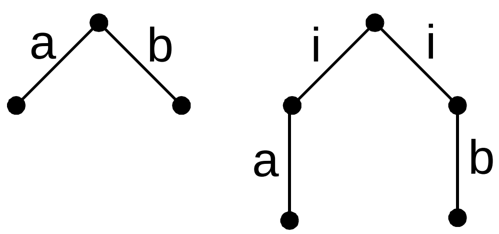
(ii) $\forall\, I$: I conf S iff

$\forall\, \sigma \in L^*$,

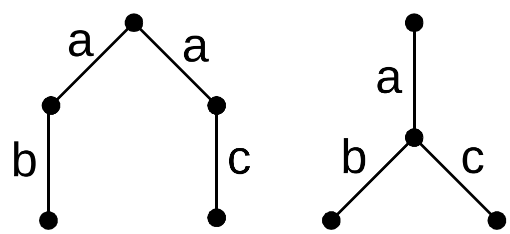
$I \,\|\, T \stackrel{\sigma}{\Longrightarrow} stop$

implies $T \stackrel{\sigma}{\Longrightarrow} stop$

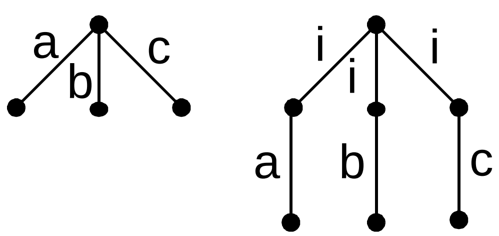
(i.e. the composition of the tester T and any given process I can reach a deadlock before T stops iff I does not conform to spec S)

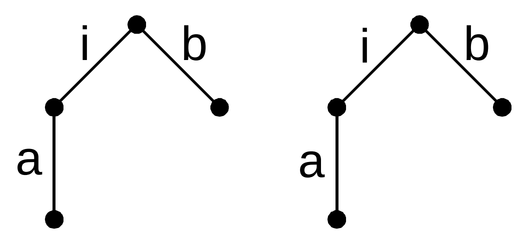


are mutal c.t.



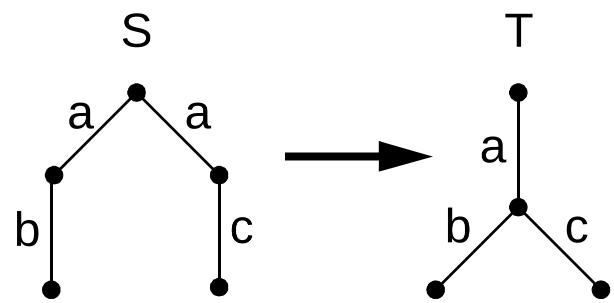
are also mutual c.t.

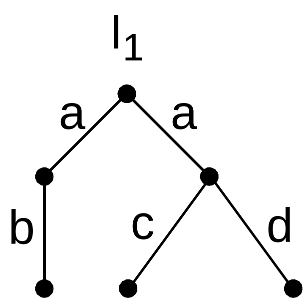


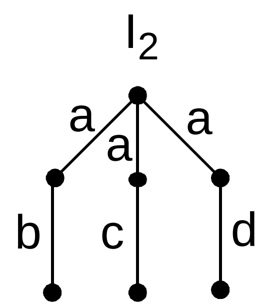


etc.

Example







This conforms to S.
It passes the tester T

This does not conform to S.
It hangs up with T on right branch.

A CONSTRUCTIVE DEFINITION OF CANONICAL TESTER (G. LEDUC)

The following alternative definition of canonical tester is a bit of a brain-twister, but it implies an algorithm for constructing it.

T(S) is defined thus:

(i) Tr(T(S)) = Tr(S)

(ii) $\forall\ \sigma \in$ Tr(S), $\forall$ A $\subseteq$ L, we have:

A $\in$ Ref(T(S),σ) iff
(L $\in$ Ref(S,σ) $\Leftrightarrow$ L-A $\in$ Ref(S,σ))

It constructs the refusal tree of the canonical tester.

Try it!

Recall: L = set of all labels (obs. actions)
L $\in$ Ref(S,σ) = σ leads to deadlock

Example

Here is a small example of use of the 'constructive' definition of canonical tester.

Let's take S= i;a;stop [] i;b;stop
L = {a,b}, Tr(S) = {ε, a, b}

The set of all subsets A of L is {{},{a},{b},{a,b}}
Ref(S,ε) ={{},{a},{b}}
Ref(S,a) = Ref(S,b) = {{}, {a},{b},{a,b}}

What is the refusal set of T(S) after ε?
Applying the definition systematically:
{} $\in$ Ref(T(S),ε) iff {a,b} $\in$ Ref(S,ε) <=> {a,b} $\in$ Ref(S,ε)

The RHS of the iff reduces to True (False <=> False)
so {} $\in$ Ref(T(S))

{a} $\in$ Ref(T(S),ε) iff {a,b} $\in$ Ref(S,ε) <=> {b} $\in$ Ref(S,ε)
The RHS is False (False <=> True), so {a} is not ...

{b} etc. is similar

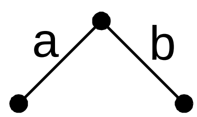
{a,b} $\in$ Ref(T(S),ε) iff {a,b} $\in$ Ref(S,ε) <=> {} $\in$ Ref(S,ε)
RHS is False (False <=> True), so...

The rest is similar, for one last example let's try
{a} $\in$ Ref(T(S),a) iff {a,b} $\in$ Ref(S,a) <=> {b} $\in$ Ref(S,a)
RHS is True, so {a} is in.

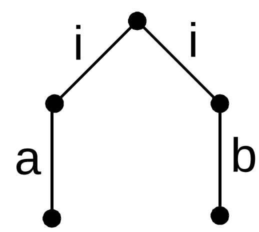
So T(S) is a;stop [] b;stop

The set of **testers** is the set of all possible *reductions* of a C.T. It includes the C.T. itself. **Basic** testers cannot be reduced.

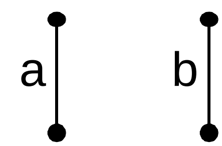
E.g.



S



C.T.



Basic tests of implems. of S

The testing of an implementation I of S will involve applying the test cases to I, and looking for *premature deadlocks*.

(if I can be nondeterministic, test cases will have to be applied repeatedly and we know that no number of tests can assure conformance)

Why introduce the concept of tester and not always use directly the CT as tester?

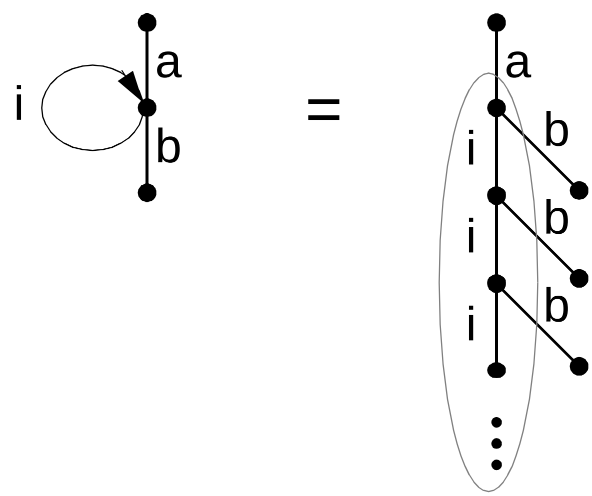
Because the CT can contain nondeterminism, while one would like testing to be a deterministic process.

By reducing the nondeterminism in a CT, one obtains a set of deterministic tests which then can be applied in sequence by a test harness.

Extension to LTS

This method can be extended to LTSs
(with appropriate changes in the defs)

The only conditions is that there should be no loops of **i**, because then we can no longer compute refusals finitely



Note the infinite sequence of i

Loops of internal actions should be removed first by using congruence laws.

(this is always possible; see Milner 1989, p.148 and 167)

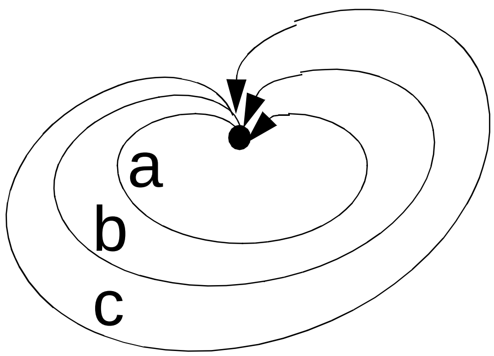
Problematic aspects of Conf

$\forall$ I, I conf stop (anything conforms to stop)

$\forall$ I, stop conf I is false because of the empty trace in stop

However, for any S, one can construct easily an I such that I conf S.

E.g. if L = {a,b,c}, such an I is



Note:
I never deadlocks!

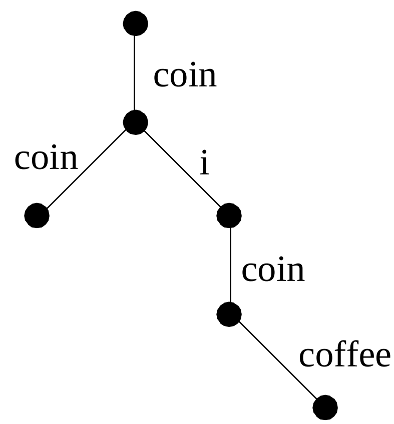
Is conf *too loose*?

This suggests that the extension process needs constraints, otherwise anything can be added.

Counterintuitive Example 1

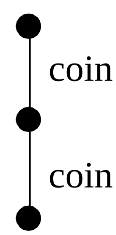
Suppose we have a coffee dispenser in which you have to insert two coins before collecting a cup of coffee. However, you should not insert the second coin <u>too quickly</u>, or you are left with nothing. This dispenser can be modeled by:

B_1 =



Now take the following machine in which you insert two coins and get nothing:

B_2 =



Rather surprisingly, B2 red B1 . With B2, no matter how long you wait before entering the second coin, you will never get coffee, and still it is considered a valid implementation of B1! This anomaly results from the assumption that an internal event will eventually occur.

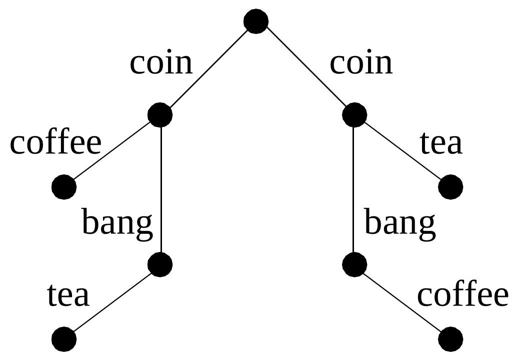
B_2 red B_1:

because B_2 can only engage in traces that are possible for B_1 and only refuse actions that are refused by B_1

Counterintuitive Example 2

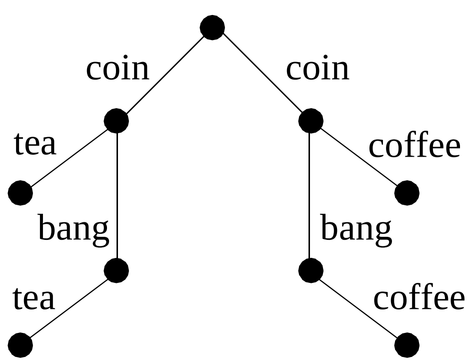
Consider this coffee and tea dispenser:

$B_1 =$



Suppose you want to drink coffee. First insert a coin, then try the coffee button. If it won’t work, hit the machine, and try again: this time you will be successful. So with this dispenser you can always get what you want. Now consider this dispenser:

$B_2 =$



Obviously, with this dispenser the above procedure will not always work: it might happen that you will have to settle for tea, even if coffee is what you want. However, it turns out that $\mathbf{B_1}$ **te** $\mathbf{B_2}$. So testing equivalence identifies the two processes, whereas intuitively they display a different behavior.

Note that $\mathbf{B_1} \not\approx \mathbf{B_2}$ suggesting that in this case w.b.e. is not too strong.

LOTOS testing theory is a nice formalization of what it means to test a nondeterministic system, something which perhaps is done less well in testing theory based on Finite State Machines.

However it has conceptual problems and it does not deal well with the ‘stimulus-response’ concepts which are at the basis of much testing practice. Also it does not deal with asynchronous testing.

Recent research has been addressing these problems. One way to solve the first problem is to add the concept of ‘direction’ of actions (whether they are initiated by the env’t or by the system).

Useful applications of this theory were found to the area of design:

- development of a detailed design from an abstract design
- testing of conformance of a design w.r.t. requirements