Testing in context: framework and FSM based test derivation

Nina Yevtushenko, Natalia Kushik
Tomsk State University

TAROT 2015

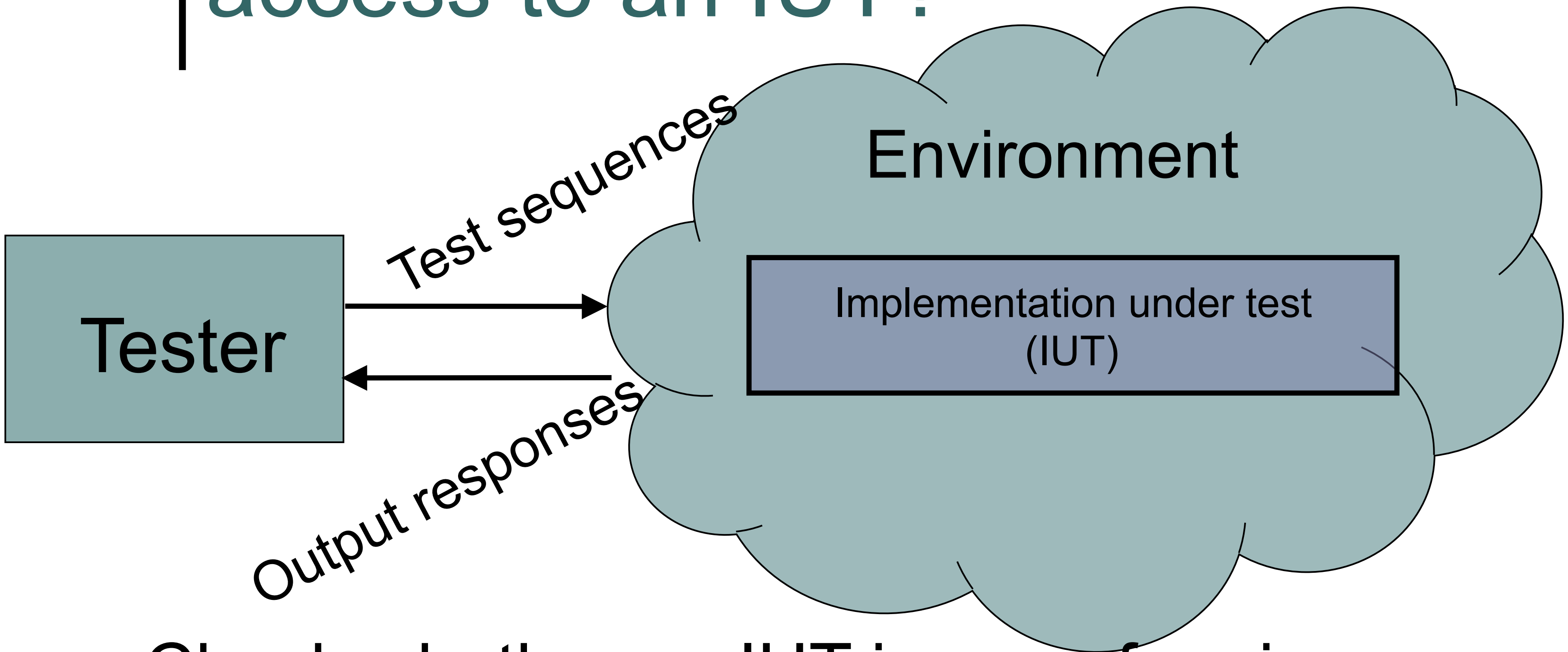
Outline

- Motivation
- Test architecture
  - Testing in context or embedded testing
- Deriving complete test suites based on the composed FSM when testing in context
  - Explicit enumeration
  - When the upper bound on the number of states is known
  - Mutation machine
- Deriving complete test suites based on the embedded component when testing in context
- Distinguishing sequences for deterministic and nondeterministic FSMs
- Complexity issues
- How to overcome these issues
- Conclusions

Motivation

- Some components cannot be tested in isolation separately from the overall system
- Complex systems have the hierarchical structure and sometimes it is simpler to test components than the overall system
- Formal methods for testing embedded components are different from those for testing in isolation

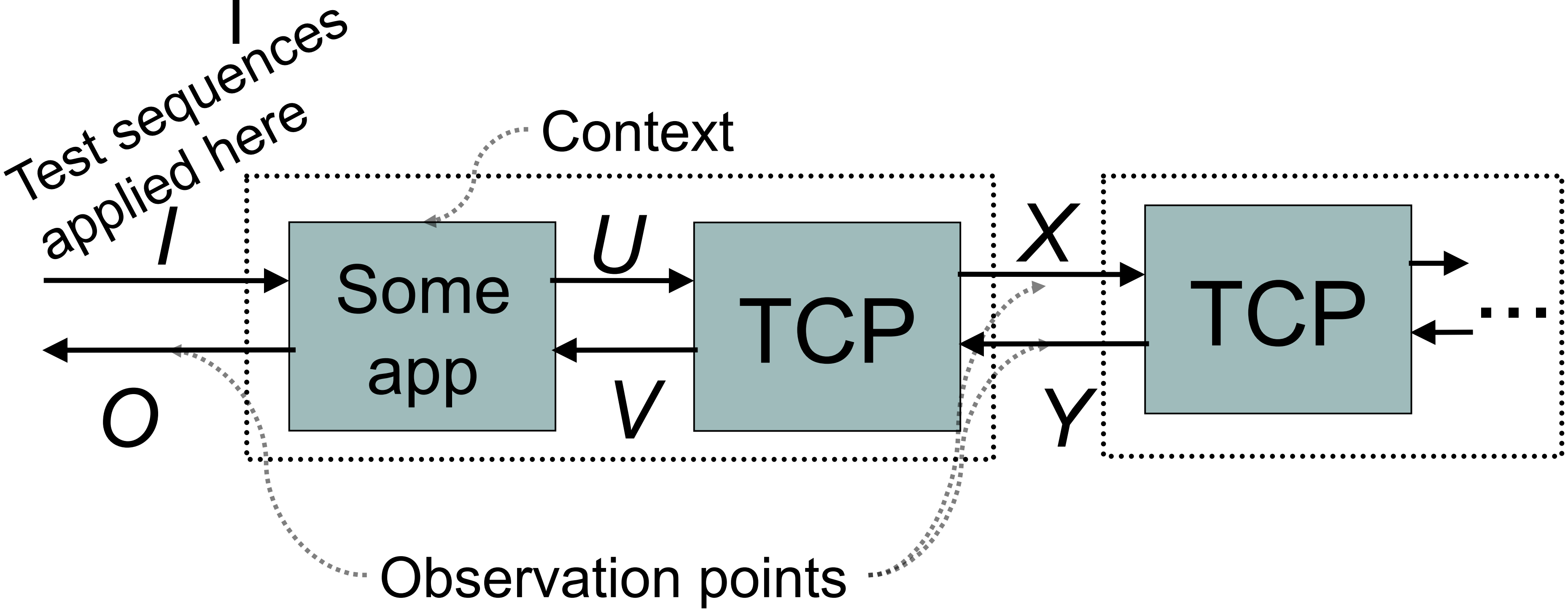
What if … not having a direct access to an IUT?



Check whether an IUT is a conforming implementation

Testing in context

TCP testing in context

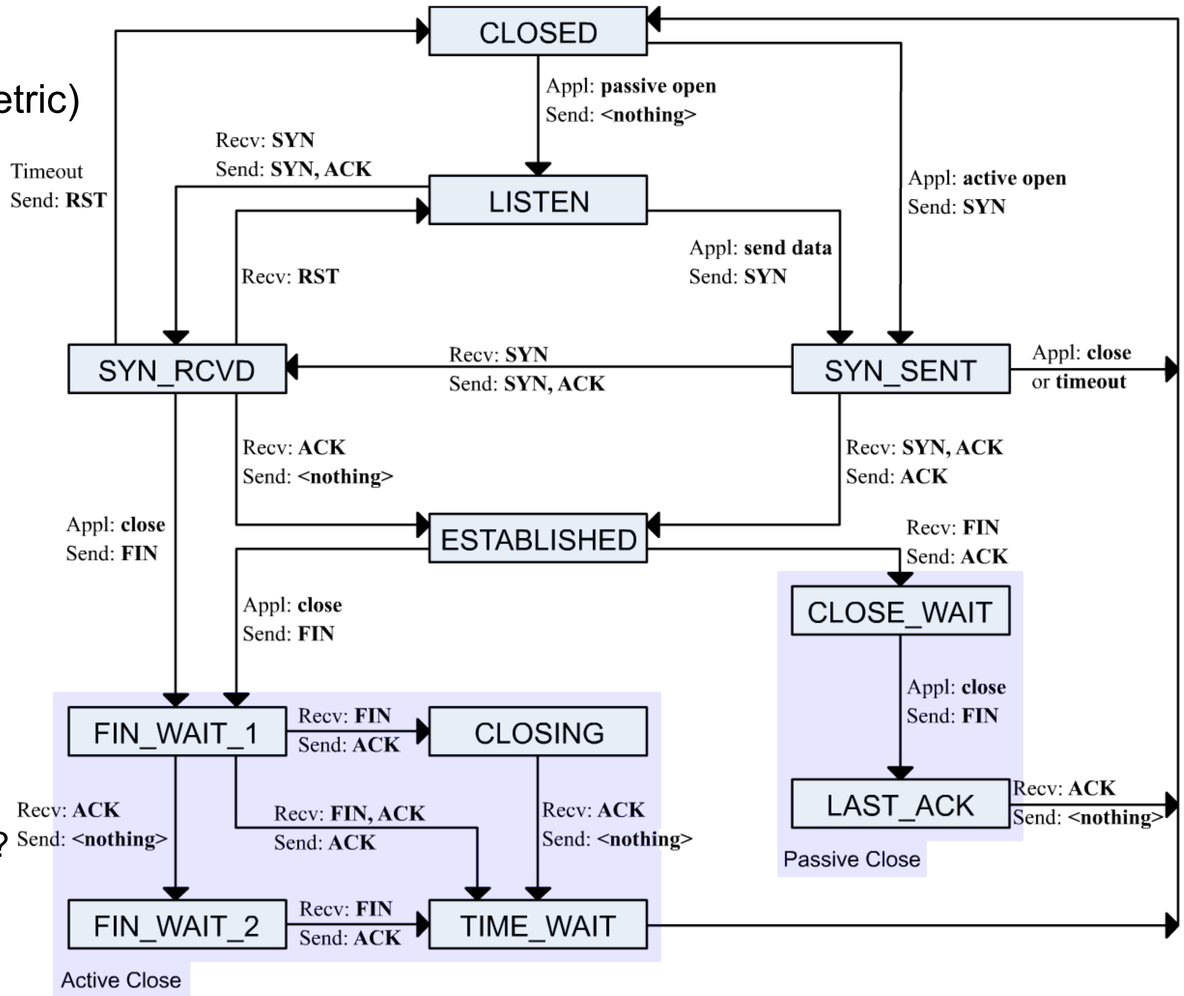


I = { *initialization, exit* }
O = { *successful_initialization, error, successful_exit* }
X = { *SYN, FIN, RST, ACK, SYNACK, FIN_ACK* }
Y = { *SYN, FIN, RST, ACK, SYNACK, FIN_ACK* }
U = { *passive_open, active_open, send_data, close* }
V = { *<nothing>* }

Transmission Control Protocol

RFC 793:
"...TCP is intended for use as
a highly reliable host-to-host (symmetric)
protocol between hosts in computer
communication networks..."

Used everywhere: Skype,
web browsers …



State model taken from
School of Computer Science
University of St Andrews
http://tcp.cs.st-andrews.ac.uk/index.shtml?page=tcp_fsm

FSM based testing

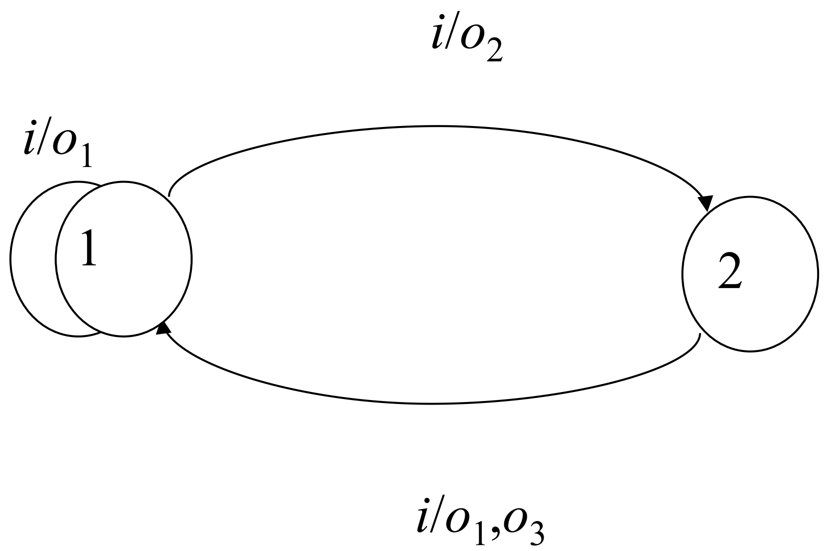
Our assumptions…

- All components are described by FSMs
- A composed system can be described by an FSM (complete or partial, deterministic or nondeterministic)
- Only one component can be faulty: all other components (the context) are fault free

Finite State Machine (FSM)

$\mathcal{S}$ $= (S, I, O, h_S, s_0)$ is an FSM

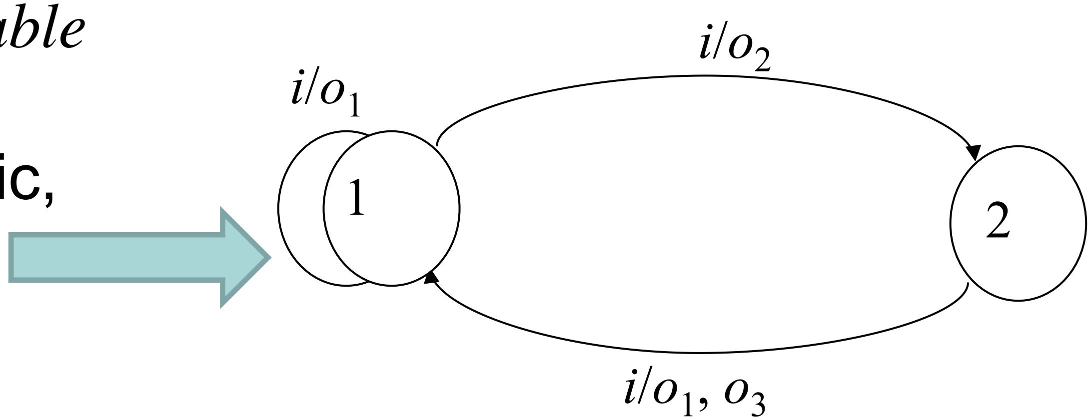
- S is a finite nonempty set of states with the initial state s_0
- I and O are finite input and output alphabets
- $h_S \subseteq S \times I \times O \times S$ is a behavior relation



FSM $S = (S, I, O, h_S, s_0)$ can be

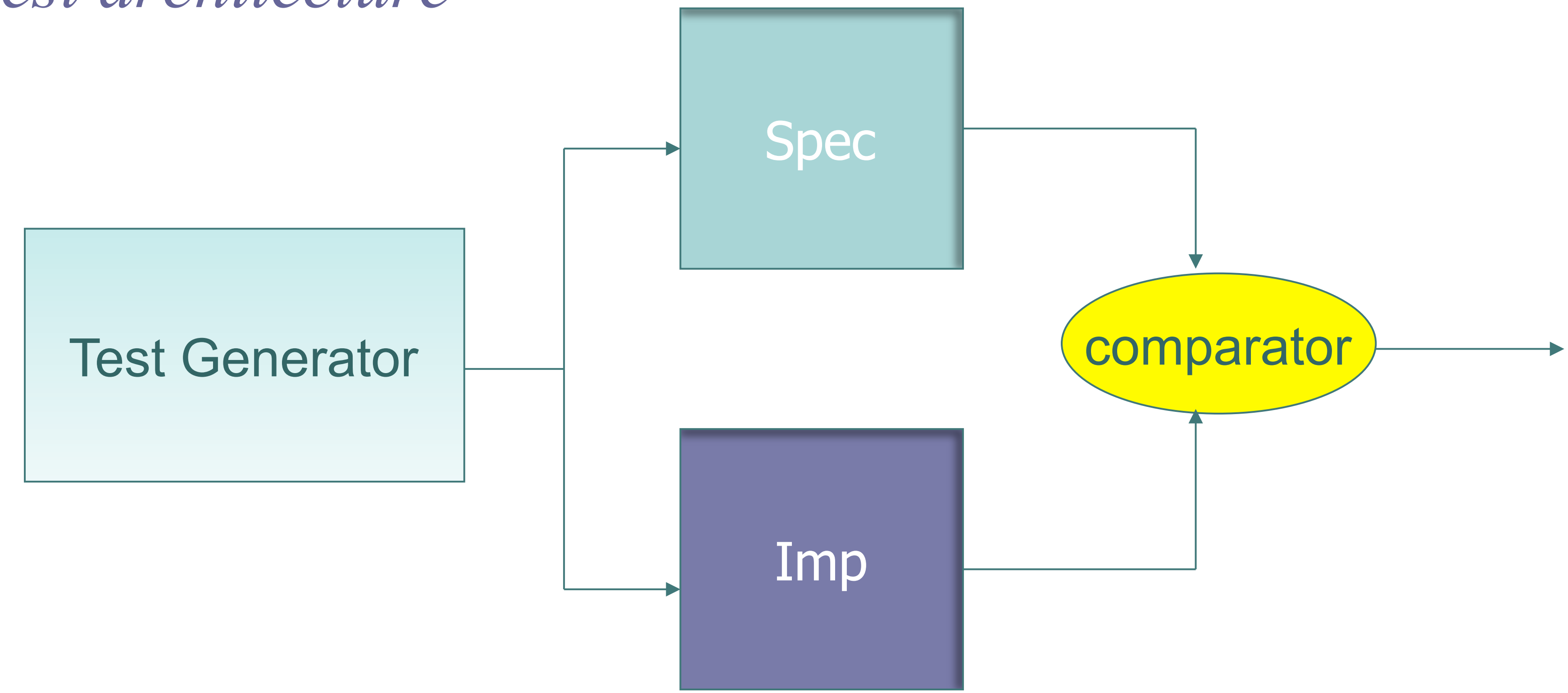
- *deterministic* if for each pair $(s, i) \in S \times I$ there exists at most one pair $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_S$
  otherwise, S is *nondeterministic*
- *complete* if for each pair $(s, i) \in S \times I$ there exists $(o, s') \in O \times S$ such that $(s, i, o, s') \in h_S$
  otherwise, S is *partial*
- *observable* if for each triple $(s, i, o) \in S \times I \times O$ there exists at most one state $s' \in S$ such that $(s, i, o, s') \in h_S$
  otherwise, S is *nonobservable*

This one is nondeterministic, complete and observable



Testing in isolation

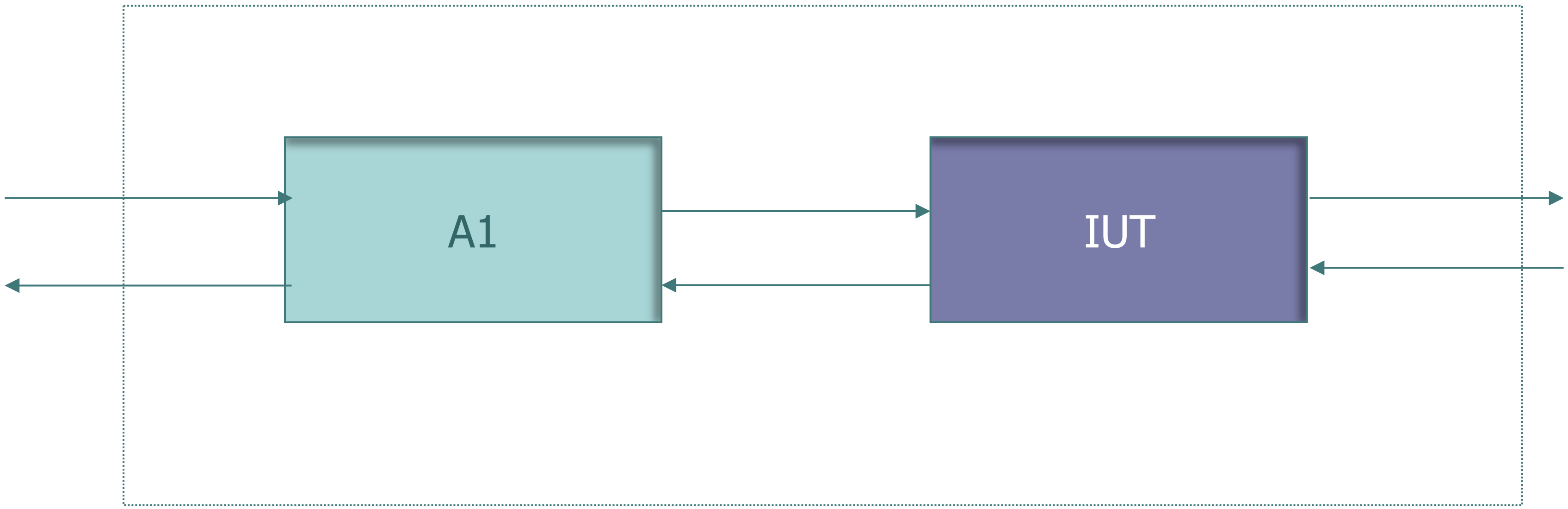
Test architecture



Conformance relation – the *equivalence*

Test architecture for testing embedded components or testing in context

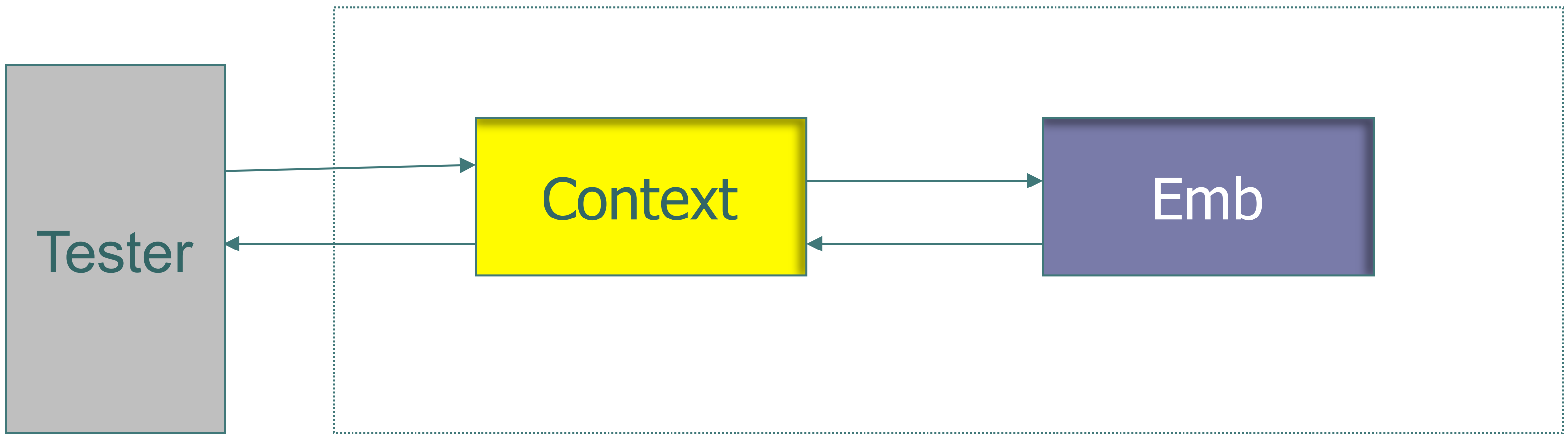
There are two FSMs in the system



There is a direct access only to some inputs and outputs of an IUT

Test architecture for testing in context

No direct access to the inputs and outputs of an IUT



Conformance relation – the *external* equivalence

How to derive a test suite for testing an embedded *IUT*?

Based on the specification of the overall composition

Derive the composed FSM

Test cases are derived for the composed FSM using FSM based testing methods

⇓

There can be the guaranteed fault coverage under specific conditions!

but

Tests are too long as there are many infeasible FSMs in the Fault Domain

Based on the specification of Emb

Use tests derived for the *Emb* in isolation

Use ordinary test methods for deriving corresponding external test suites

⇓

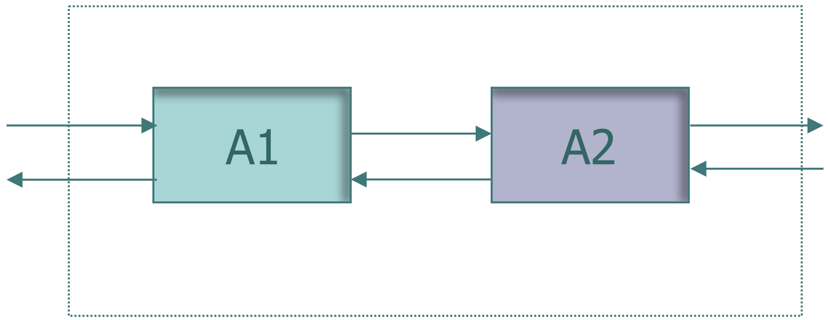
The problem with *partial controllability* and *observability*

Composed FSM based fault models

The composed FSM of two communicating FSMs

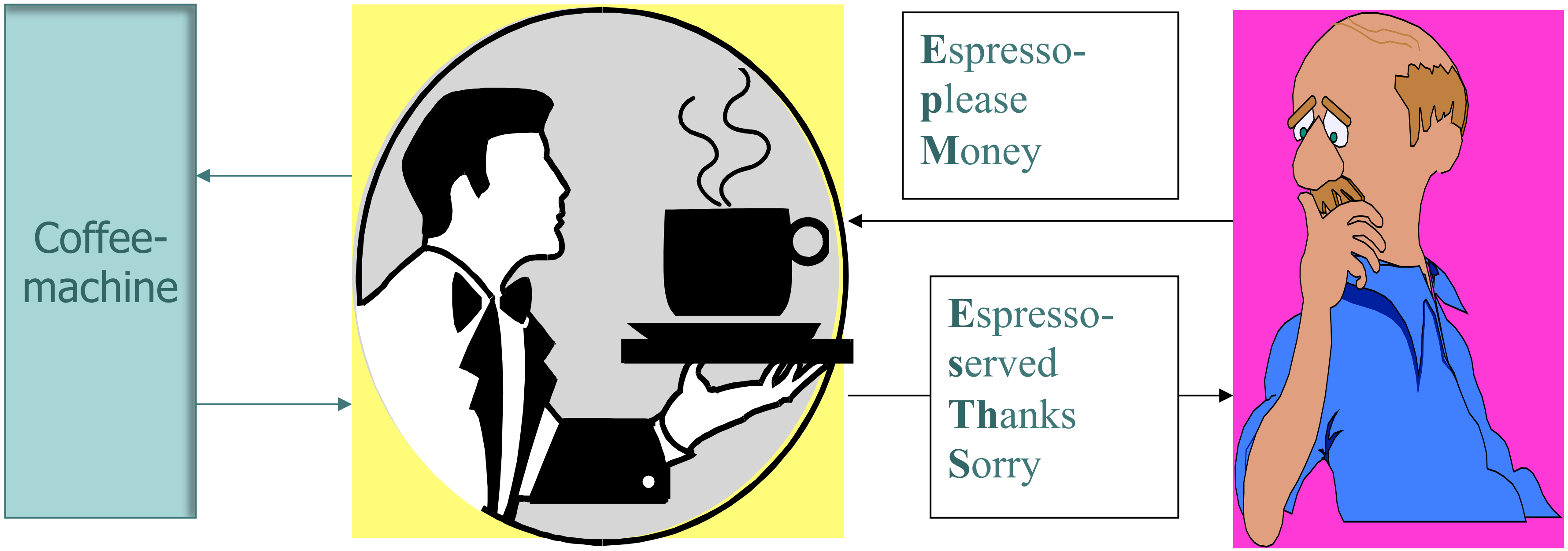
- Parallel (asynchronous) composition
- The component FSMs communicate in the dialogue mode
- One message in transit
- Slow environment

Communicating FSMs



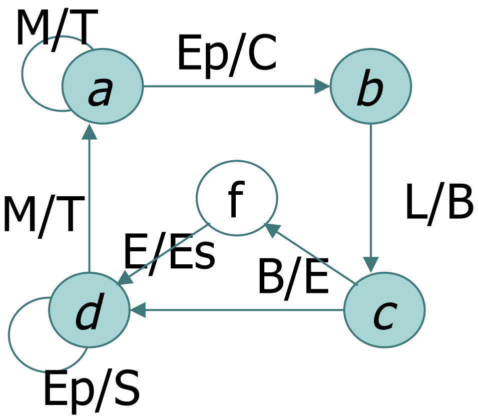
Parallel composition of FSMs (coffee-shop)

Coffee-shop

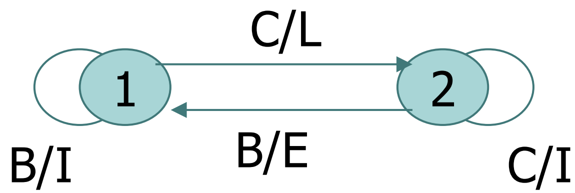


Parallel composition for a coffee-shop

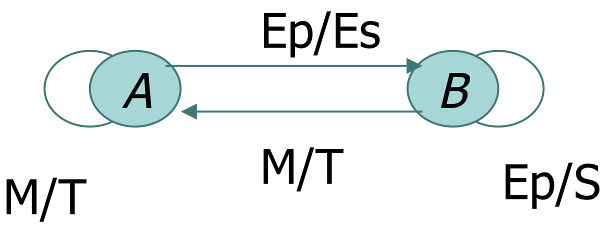
Waiter



Coffee machine



Coffee shop



Live-locks and dead-locks

The *live-lock* can occur when the dialogue becomes infinite

$$\Downarrow$$

A corresponding external input sequence is not allowed

$$\Downarrow$$

The composed FSM becomes partial

Detecting live-locks usually is based on timeouts

Dead-locks

The *dead-lock* can occur when components are partial and an unsafe (external or internal) input is applied to a component FSM

$\Downarrow$

A corresponding external input sequence is not allowed

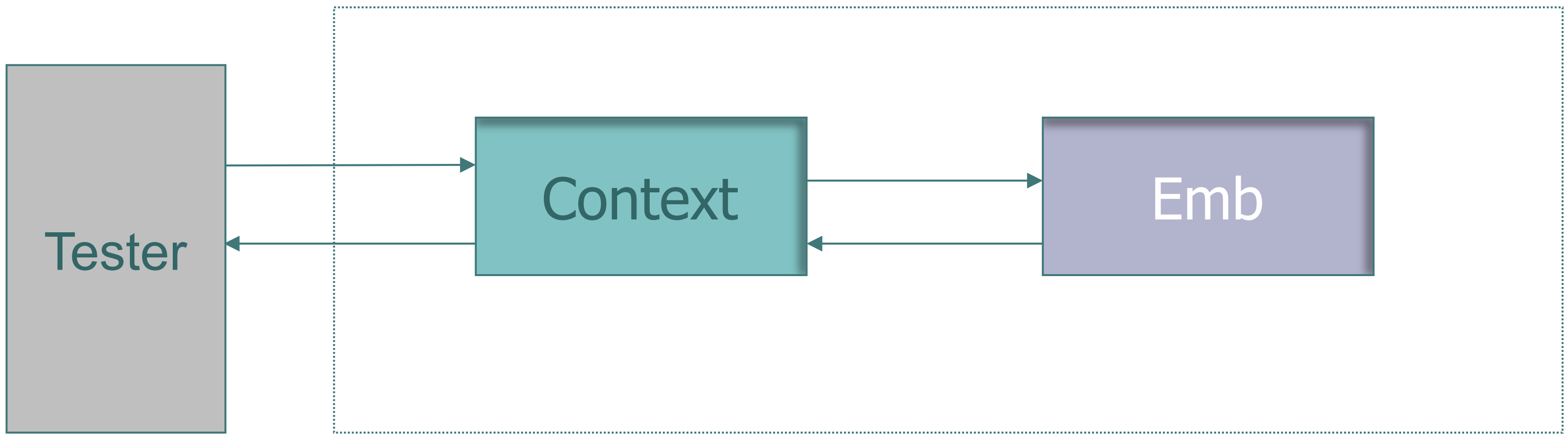
$\Downarrow$

The composed FSM becomes partial

For complete component FSMs there are no dead-locks

External equivalence

! The number of FSMs can replace the embedded component FSM preserving the external behavior of the overall system



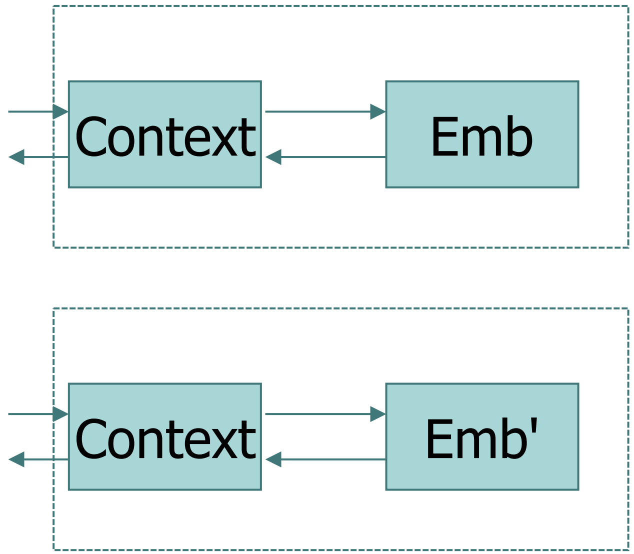
Conformance relation – the *external equivalence* $\cong_{ext}$

External equivalence (2)

FSMs *Emb* and *Emb'* are *externally equivalent* if *Context* ◊ *Emb* and *Context* ◊ *Emb'* are equivalent

! FSMs Emb and Emb' can be non-equivalent

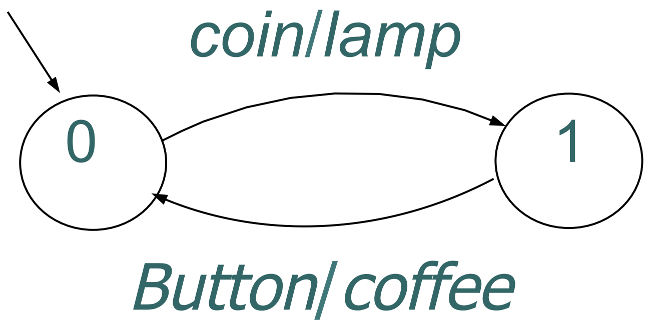
Specification and implementation systems



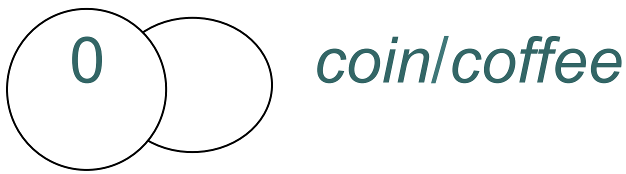
Externally equivalent coffee machines

If there is a waiter

Coffee-machine



Can be replaced with a reduced coffee-machine



Types of faults in *Emb* implementation

- *Output* faults: the output of a transition (s, i, o, s') is wrong compared with that of the specification embedded component FSM

- *Transfer* faults: the next state of the transition (s, i, o, s') is wrong compared with that of the specification embedded component FSM

- Mixed faults

Fault model for testing in context

Fault model $<S, \cong, Context \Diamond FD_{Emb}>$

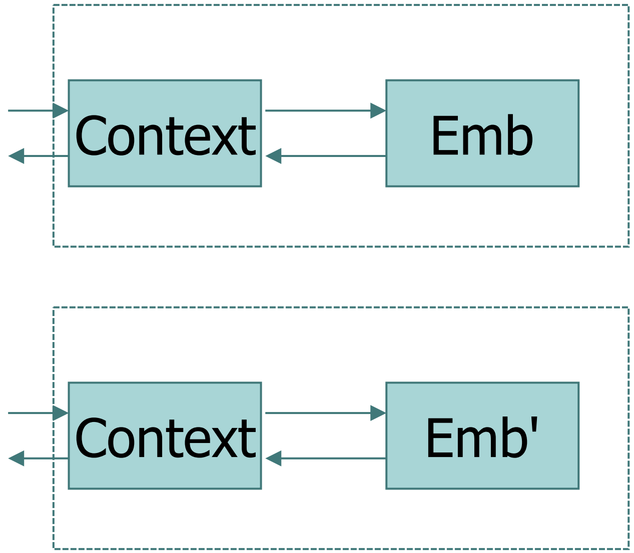
Our assumptions…

- $S = Context \Diamond Emb$ is a deterministic and complete FSM
- $\cong$ is the equivalence relation
- FD_{Emb} is the set of all possible implementation FSMs of *Emb* which have no live-locks when combined with the *Context*

Test suite reminder

A *test case* is a finite input sequence of the specification Context ◊ Emb. A *test suite* is a finite set of test cases

Specification and implementation under test



A test suite *TS* is *complete* w.r.t. the FM <*S*, ≅, *FD*> if for each FSM *Imp* ∈ *FD* that is not equivalent to *S*

there exists a test case α ∈ *TS* that kills *Imp*

We assume that each implementation system has a reliable reset *r* that takes the implementation from each state to the initial state

Explicit enumeration

Explicit enumeration can be used when the number of mutants of *Emb* is not big

Faults in the embedded component are explicitly enumerated

$S = Context \lozenge Emb$

$Imp = Context \lozenge Emb'$

! Imp can be partial if there are live-locks

Derive the intersection

$S \cap Imp$

If $S \cap Imp$ is not complete then

derive a distinguishing sequence (a test case that kills a faulty *Emb'*)

! If there are no live-locks when combining the context with the Emb' then a sequence distinguishing externally nonequivalent Emb and Emb' always exists

Explicit enumeration (2)

Advantage: Easy to implement

Disadvantage: Cannot be applied when the number of faults (the number of mutants) is huge

! Efficient algorithm for deriving distinguishing sequences for two FSMs should be developed

! There are no methods how to derive a complete test suite w.r.t. the FM $<S, \cong, Context \lozenge FD_{Emb}>$ without explicit enumeration

Using a bigger fault domain

Fault model <S, ≅, $\Im_m$>

Our assumptions…

- S = Context ◊ Emb is a deterministic and complete FSM
- ≅ is the equivalence relation
- *Fault domain* is a set of all complete deterministic FSMs with at most m states where $m = n_{context} \cdot n_{emb}$

$\Im_m \supseteq$ Context ◊ FD_{Emb} where FD_{Emb} is the set of all possible implementation FSMs of Emb with at most n_{emb} states which have no live-locks when combined with the context

A complete test suite w.r.t. <S, ≅, $\Im_m$> is complete w.r.t. <S, ≅, Context ◊ FD_{Emb}>

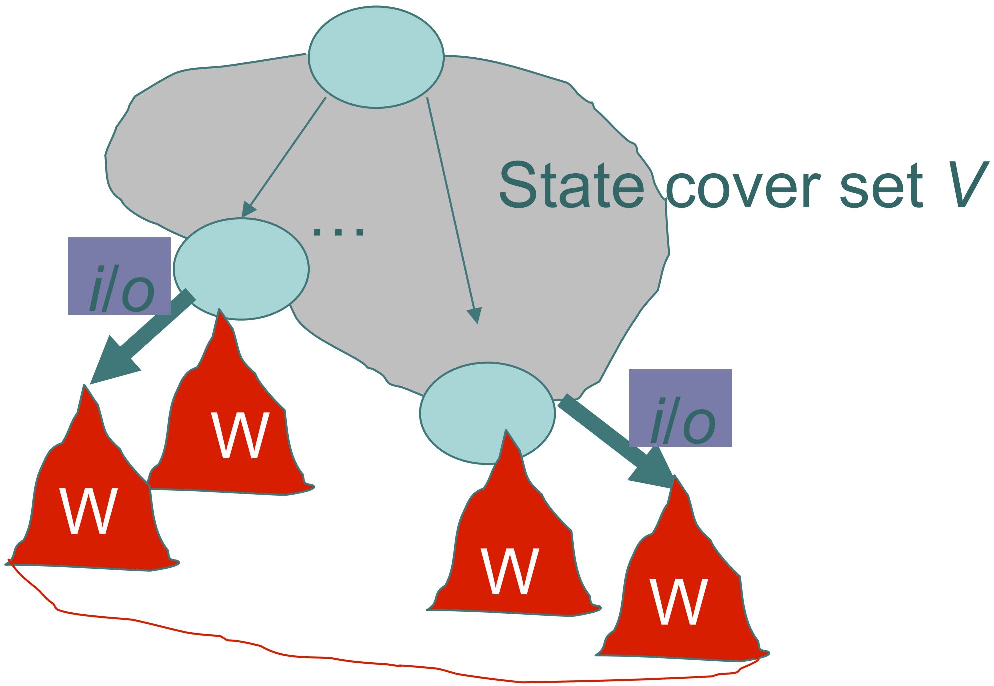
W-method

1. For each two states s_j and s_k of the specification FSM *Spec* derive a distinguishing sequence γ_{jk}
   Gather all the sequences into a set *W* that is called a *distinguishability* set
2. For each state s_j of the FSM *Spec* derive an input sequence that takes the FSM *Spec* to state s_j from the initial state
   Gather all the sequences into a set *CS* that is called a *state cover* set

W-method (2)

3. Concatenate each sequence of the state cover set *V* with the distinguishability set *W*: $TS_1 = V.W$

4. Concatenate each sequence of the state cover set *V* with the set *iW* for each input *i*: $TS_2 = V.I.W$



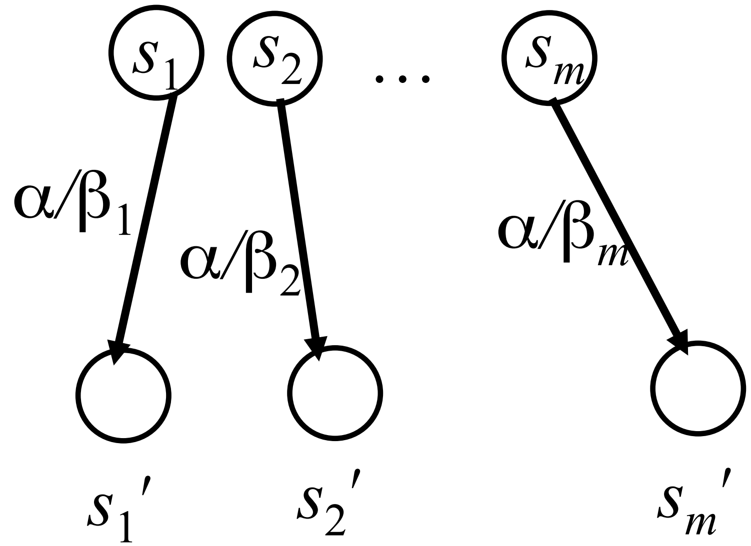
! The shortest test suites are derived when FSM has a *distinguishing* sequence

R. Dorofeeva, K. El-Fakih, S. Maag,R. Cavalli, N. Yevtushenko, "FSM-based conformance testing methods: A survey annotated with experimental evaluation," Inform. & Softw. Tech., vol. 52, no. 12, pp. 1286–1297, 2010

Distinguishing sequence

- Given two states of a deterministic complete FSM *Spec* and a distinguishing sequence α, there is a unique output response at each state of *Spec*
- After applying α at any state s_i and observing an output response β_i the initial state s_i before applying α becomes known

Distinguishing sequence α



$$\beta_1 \neq \beta_2 \neq \ldots \neq \beta_m$$

Using the W-method

The fault model <S, $\cong$, $\Im_m$>

Advantage: well developed

Disadvantages: what is *m*? If *m* is the product of the number of states of the *Context* and *Emb* then a test suite will be (extremely!!!) long

- Many machines are infeasible - not each machine with at most *m* states is a composition of the *Context* and some *Emb'*
- Does not take into account that the *Context* is fault-free

Using the W-method (2)

Disadvantages can be overcome by

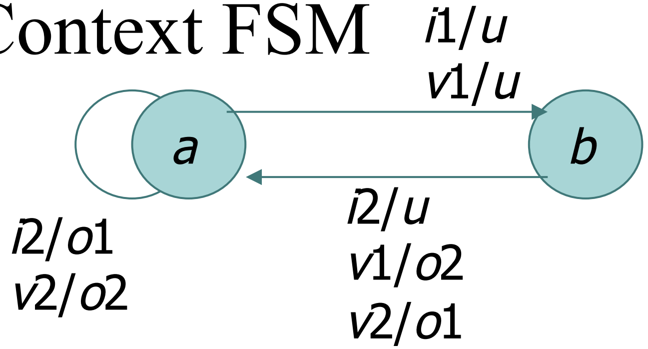
- *Using a mutation machine*
- *Tests can be shortened by deleting redundant transitions (as Ana mentioned yesterday…)*

1) L. P Lima and A. R. Cavalli, "A pragmatic approach to generating test sequences for embedded systems", *Proc. of the* 10*th* *International Workshop on Testing of Communicating Systems*, pp: 125-140, 1997

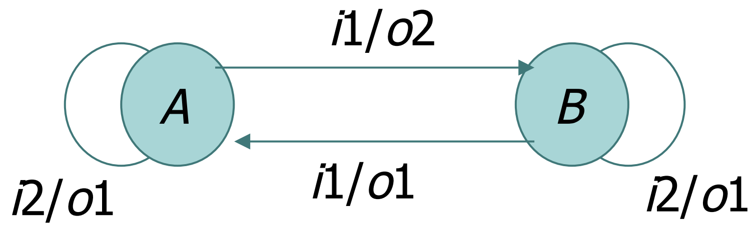
2) Yevtushenko, N., Cavalli, A.R., and Lima, L.P. (1998), *Test minimization for testing in context*. Proceedings of 11th IWTCS, pp: 127-145

Mutation FSM for *Emb*

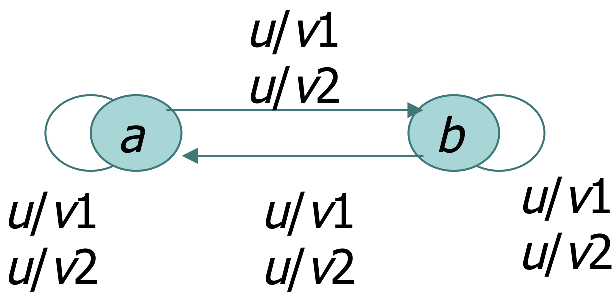
Context FSM



Composed FSM



Mutation FSM for
the embedded FSM



Mutation composed FSM is the composed FSM of Context ◊ MM_{Emb}

Mutation machine for testing in context

Fault model $<S, \cong, Sub(MM)>$

$$MM = Context \lozenge MM_{Emb}$$

There exist methods for deriving complete test suites w.r.t. such a fault model without explicit mutant enumeration

Known as: Grey box testing or Fault function or Mutation machine or Incremental testing

Mutation machine for testing in context (2)

Mutation machine *MM* is obtained by combining the Mutation Machine for the *Emb* with the context

Advantage: Tests are derived w.r.t. external inputs and outputs

Disadvantages: a) MM is big enough

b) MM still has infeasible machines

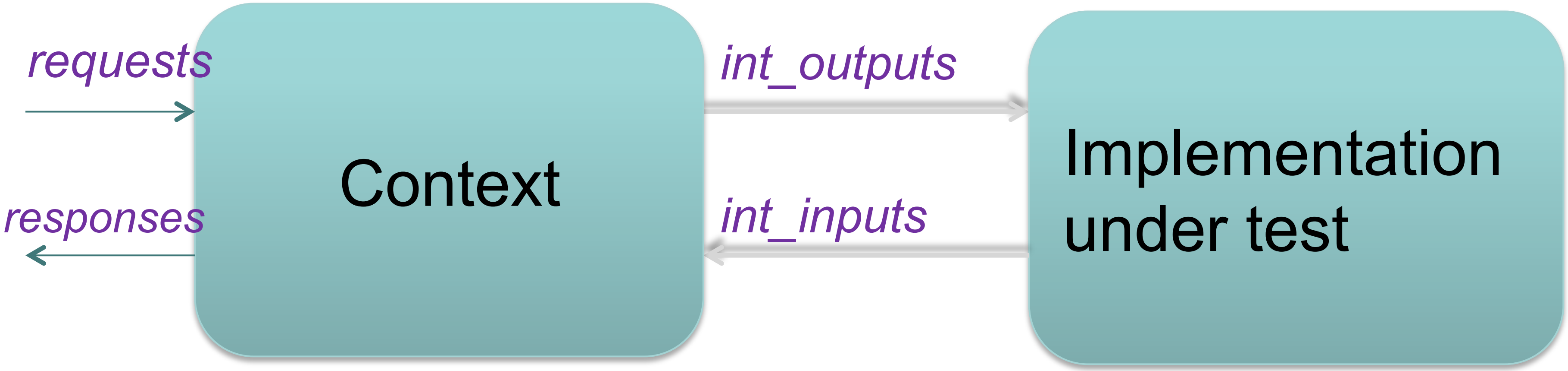
! The number of infeasible submachines can be reduced if several mutation machines are used

! Tests can be shortened by deleting redundant transitions

! Tests can be shortened if the composed mutation FSM has a separating (distinguishing) sequence

Embedded component based fault models

The idea behind testing in context



Context can be…

- Environment

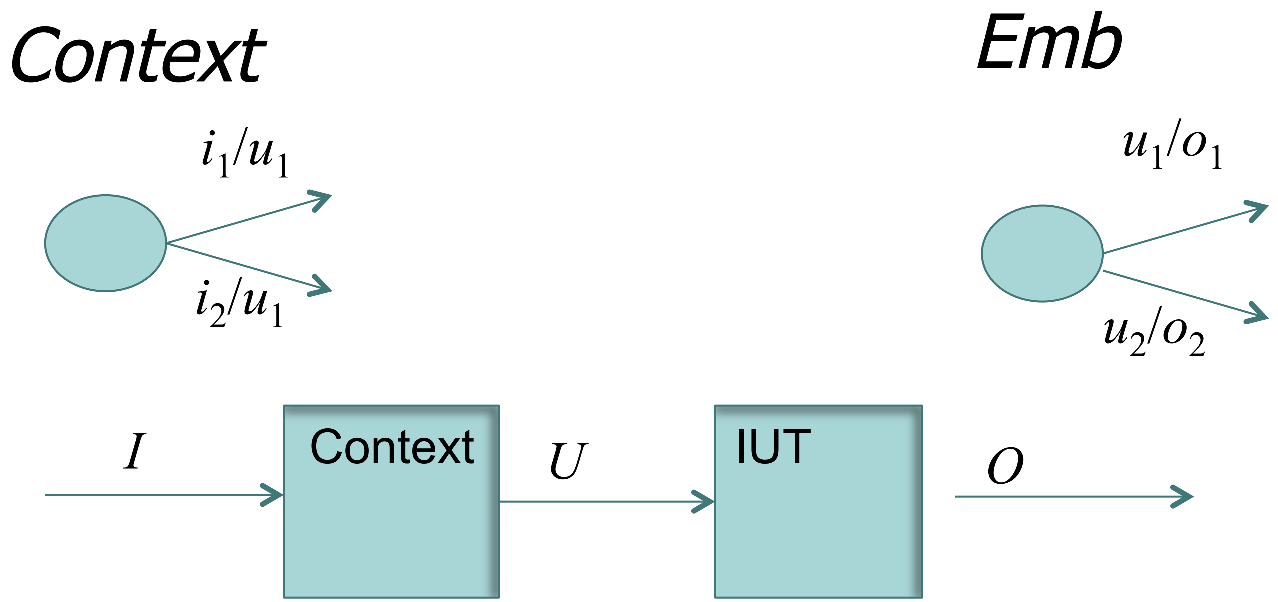
- Another implementation

The problem…

- No access is granted to internal channels

Partial controllability and partial observabilty of an IUT

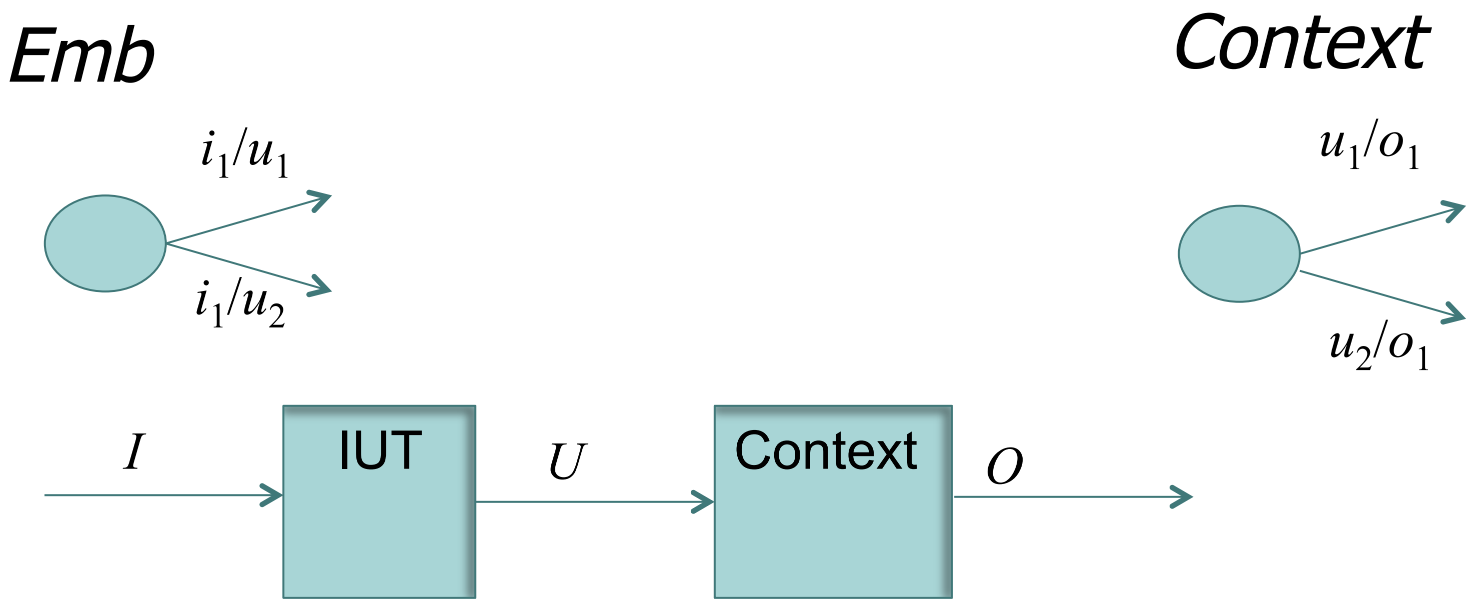
Partial controllabilty when testing in context: a transition tour for an embedded component



How to check a transition under u_2 in the embedded component? Or we cannot check it at all?

It is much harder to derive a transition tour when there is no direct access ☹

Partial observability when testing in context



Is this fine that at the initial state internal outputs cannot be distinguished or we still need to do this but after an appropriate external input sequence?

And a transition tour is not enough for checking transfer faults …

As a small example let's consider…

Password Authentication Protocol (PAP)

- Authentication protocol that uses a password
- Two entities share a password in advance and use the password as the basis of authentication
- Considered to be unsecure, but that's another business ☺

How it works…

- A client sends a username and a password
- The server sends authentication: Ack (when OK!) or Nack (when not OK!)

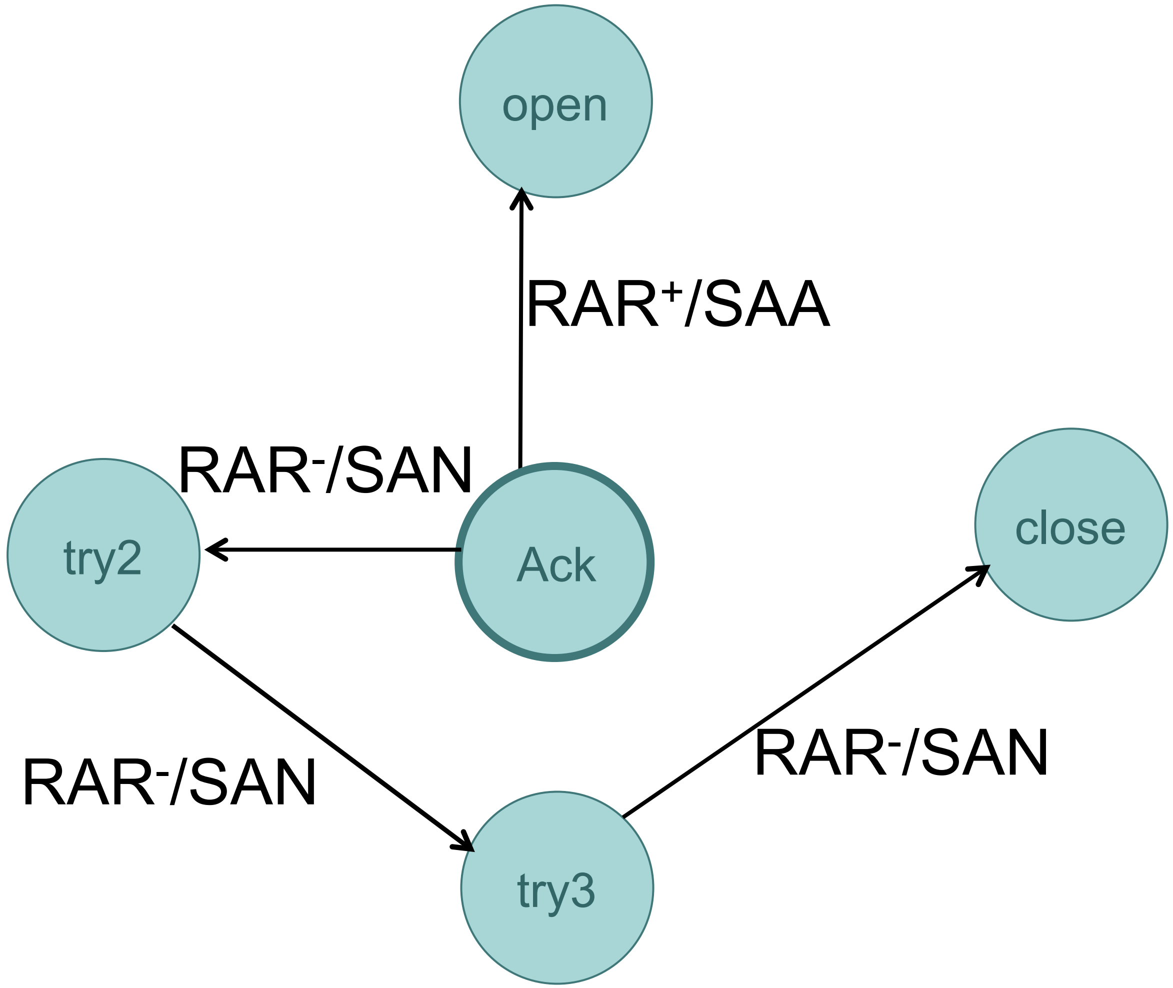
One of FSMs for PAP

RAR^+ - «good» login

RAR^- - «bad» login

SAA - Ack

SAN – Nack



Deriving tests

Under assumption…

- We can ‘build’ an FSM that simulates a faulty implementation
- There can be faults of two types:

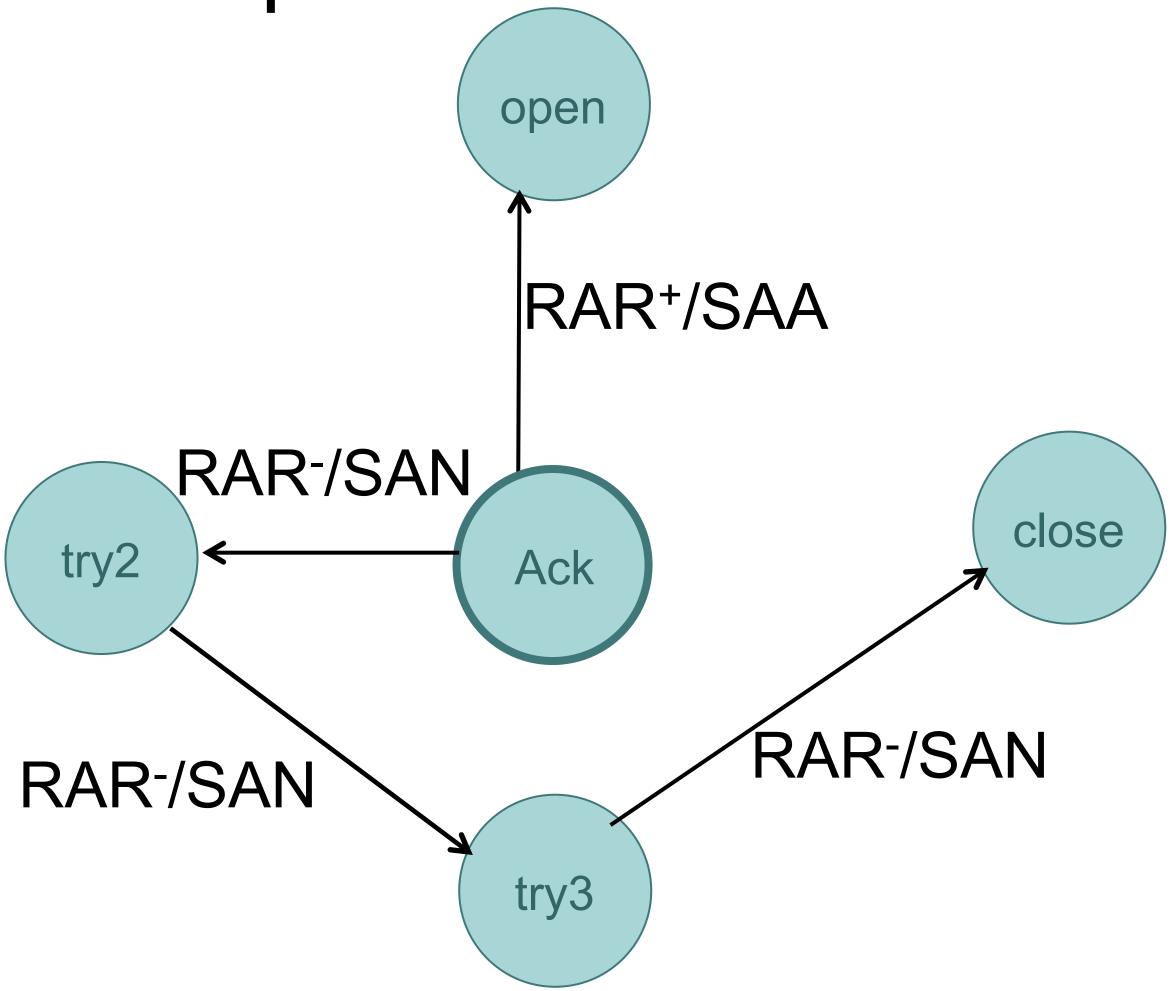
-Transition faults

-Output faults

Let’s rely on a transition tour

- *Idea*: to traverse each FSM transition at least once
- Theory: transition tour is known to detect all output faults

Transition tour for the PAP model



Test suite:

RAR^{+}

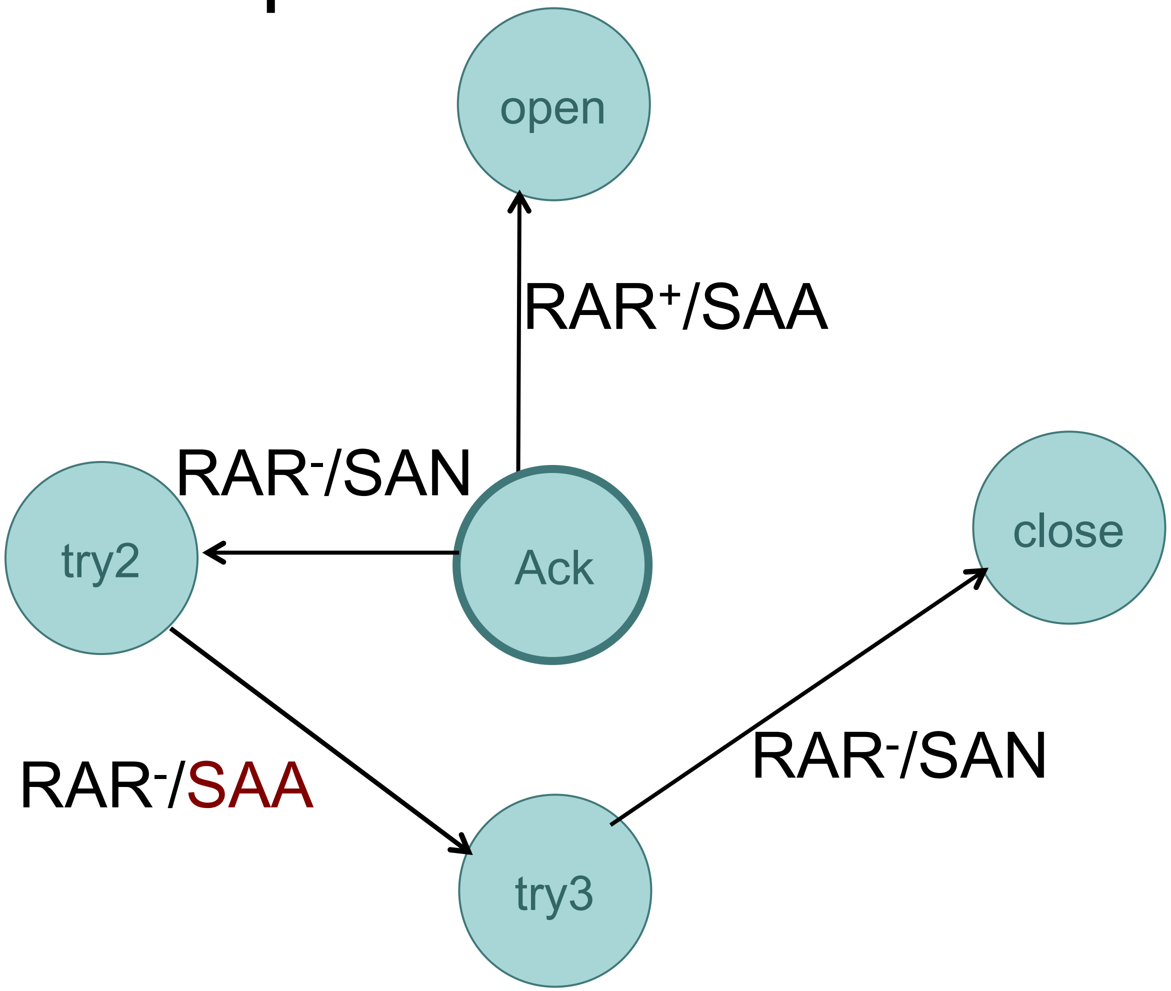
$RAR^{-}RAR^{-}RAR^{-}$

Expected output reactions:

SAA

SAN SAN SAN

Detecting an output fault



Test suite:

RAR^+

$RAR^-RAR^-RAR^-$

Expected:

SAA

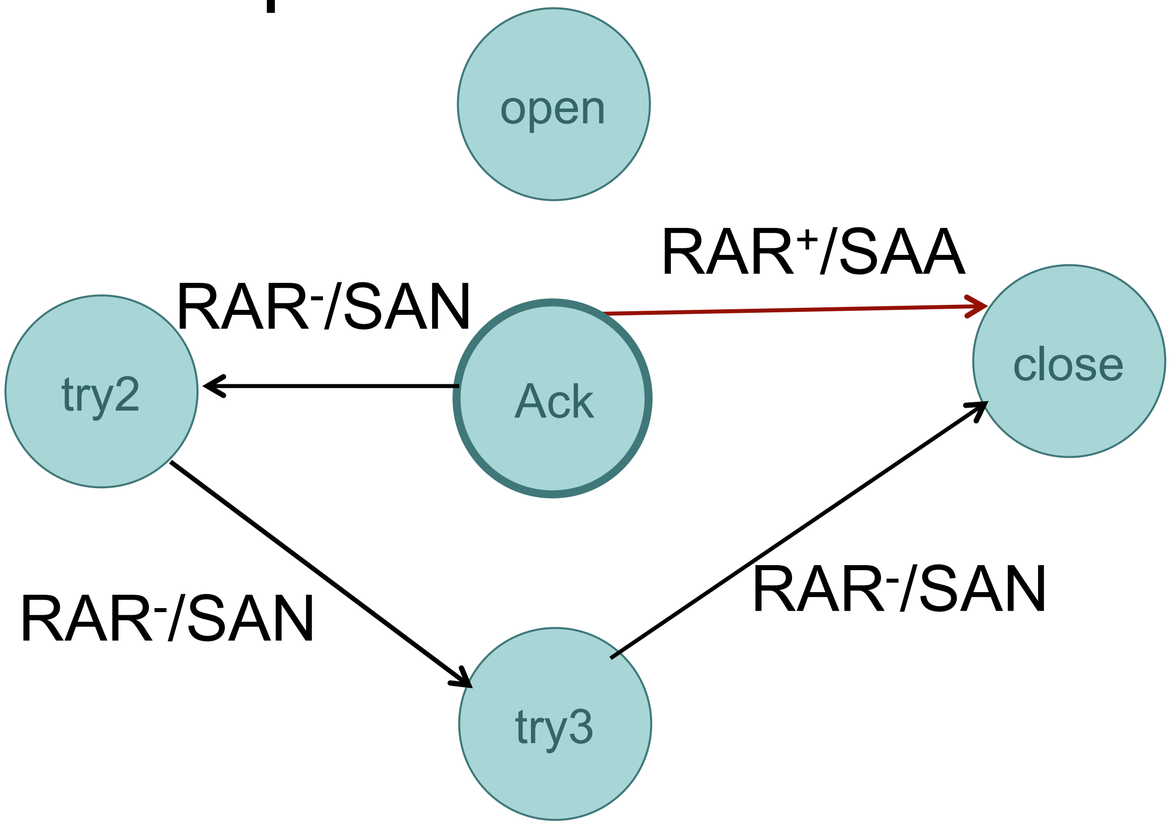
SAN SAN SAN

Observed:

SAA

SAN SAA SAN

Trying to detect a transfer fault



Test suite:

RAR^{+}

$RAR^{-}RAR^{-}RAR^{-}$

Expected:

SAA

SAN SAN SAN

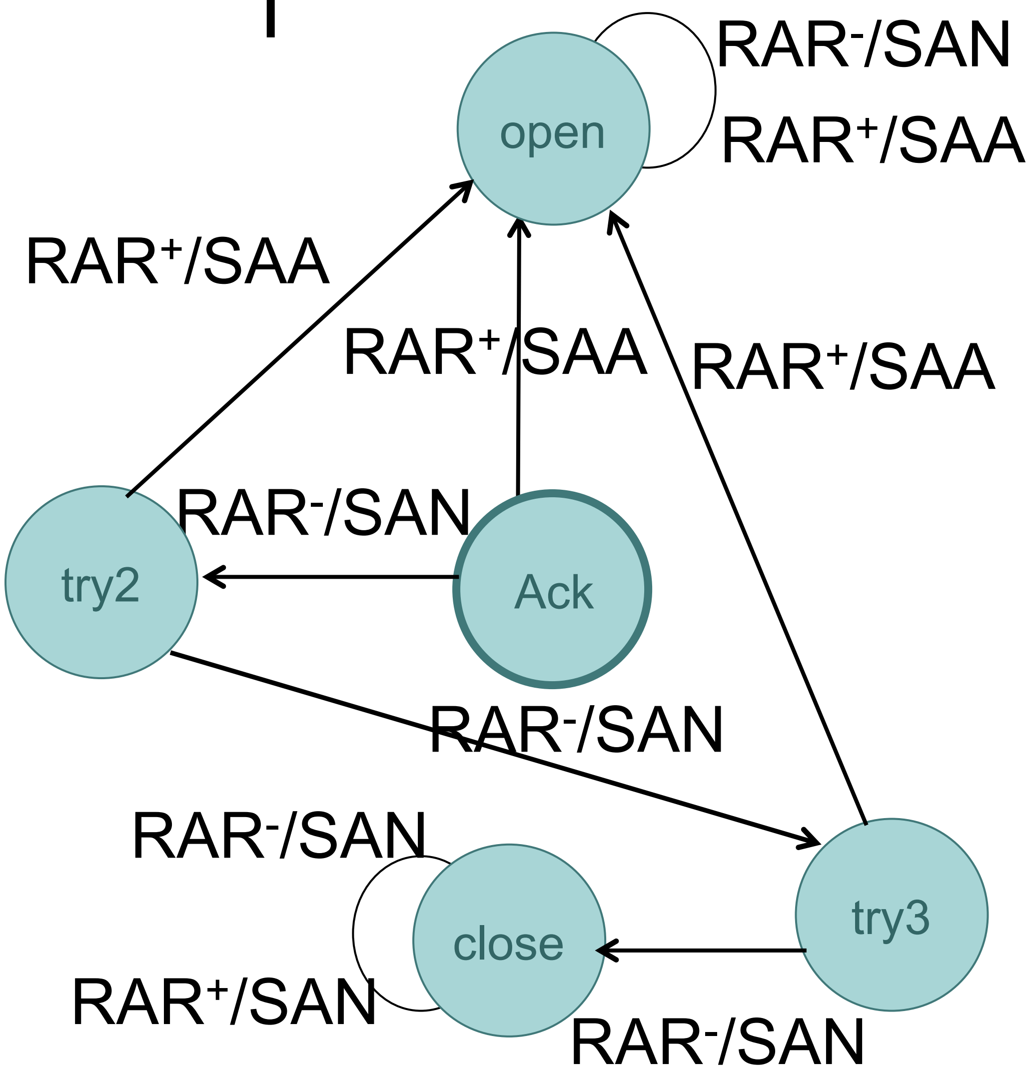
Observed:

SAA

SAN SAN SAN

A transition fault cannot be detected by a transition tour!!!

Let's make the model complete first



Define the undefined transitions…

- *Whenever the access is prohibited, the reply is SAN,*
- *Whenever, the access is given, the reply is SAA*

Distinguishing sequences for state pairs in the running example

(Ack, open) : $RAR^- RAR^- RAR^- RAR^- RAR^+$

(Ack, try2) : $RAR^- RAR^- RAR^+$

(Ack, try3) : $RAR^- RAR^+$

(Ack, close) : RAR^+

(open, try2) : $RAR^- RAR^- RAR^+$

(open, try3) : $RAR^- RAR^+$

(open, close) : RAR^+

(try2, try3) : $RAR^- RAR^+$

(try2, close) : RAR^+

(try3, close) : RAR^+

Deriving a test suite by W-method

Idea : to reach each state and then to distinguish this state from any other

Initial state Ack: $RAR^- RAR^- RAR^- RAR^- RAR^+$

…

RAR^+

state Open: $RAR^+ RAR^- RAR^- RAR^- RAR^- RAR^+$

…

$RAR^+ RAR^+$

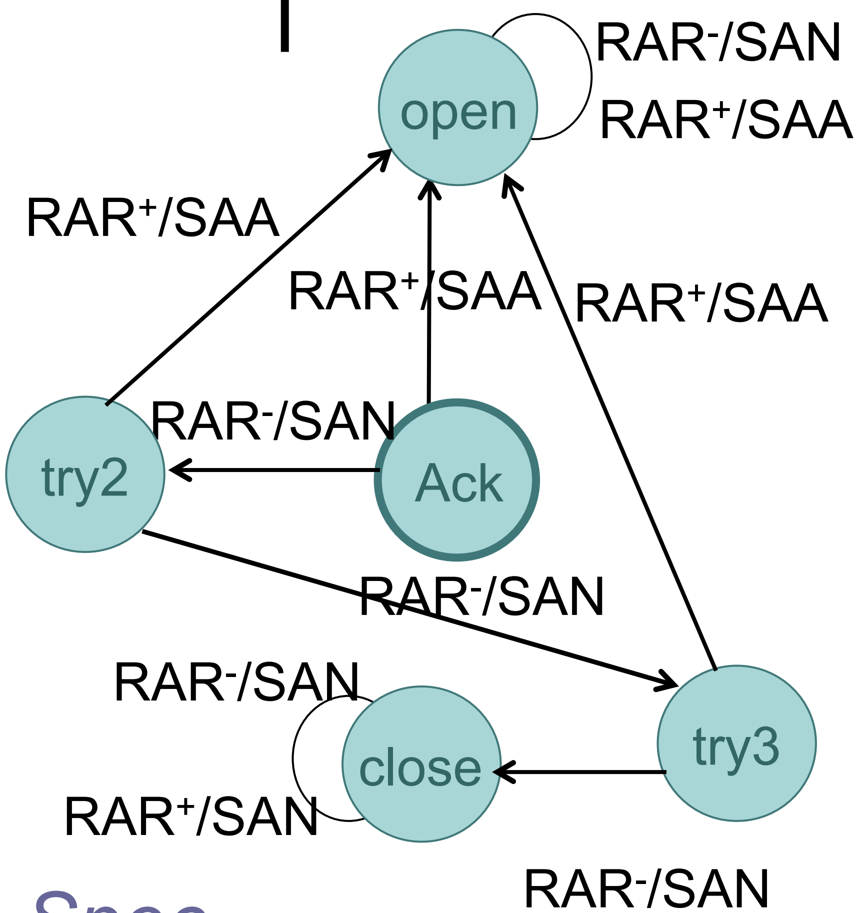
state try2: $RAR^+ RAR^- RAR^- RAR^- RAR^- RAR^+$

$RAR^+ RAR^- RAR^- RAR^+$

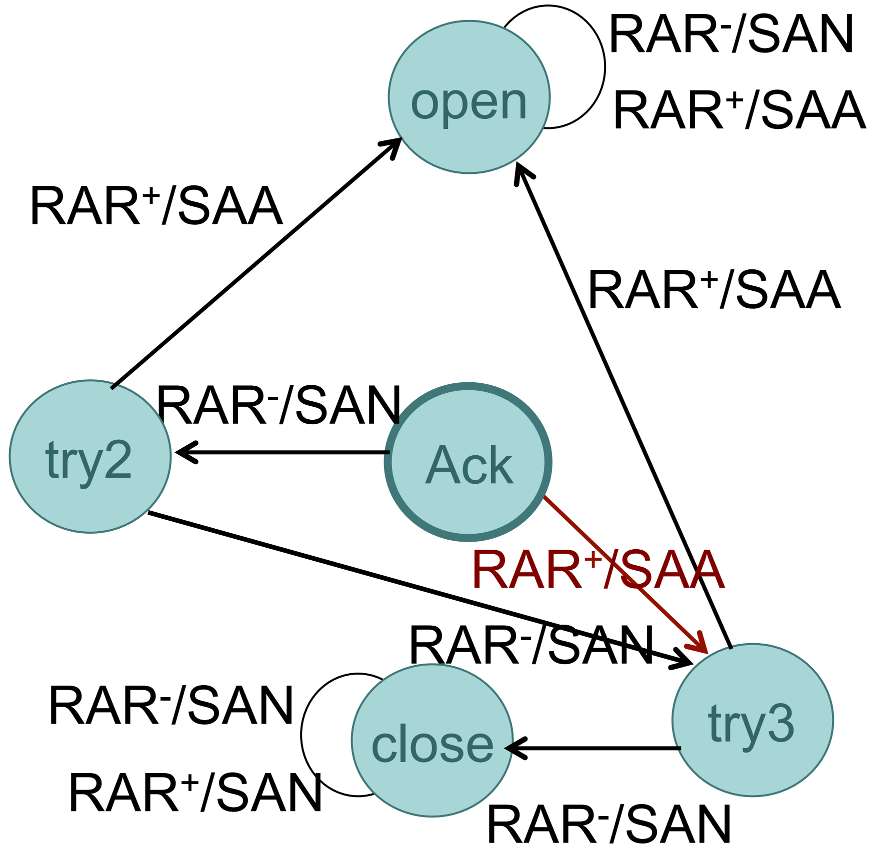
$RAR^+ RAR^- RAR^+$

$RAR^+ RAR^+$ …

Detecting a transfer fault



Spec



Imp

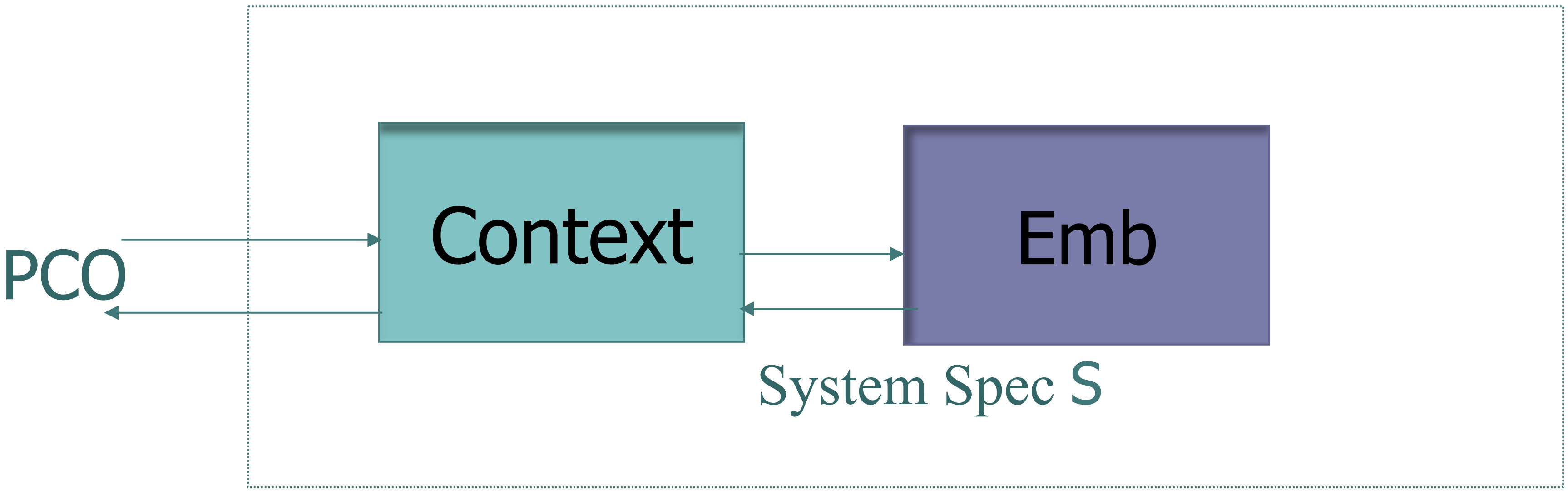
Test sequence RAR^+ RAR^- RAR^- RAR^- RAR^- RAR^+

Spec reaction : SAA SAN SAN SAN SAN SAA

Imp reaction : SAA SAN SAN SAN SAN SAN

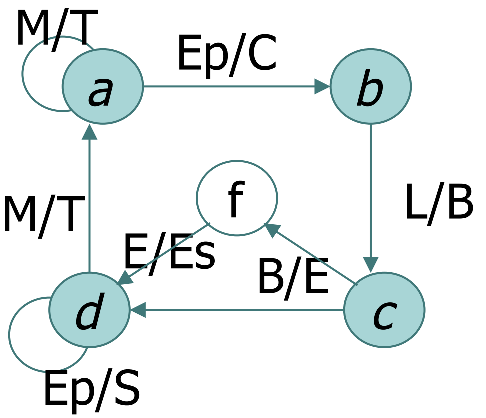
Using tests derived for the isolated embedded component

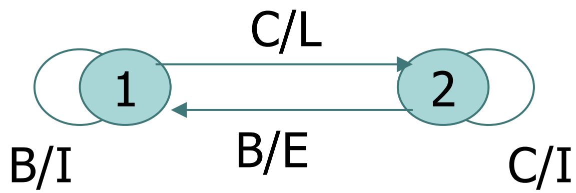
- Derive a complete test suite for Emb in isolation
- Translate it into external inputs and outputs

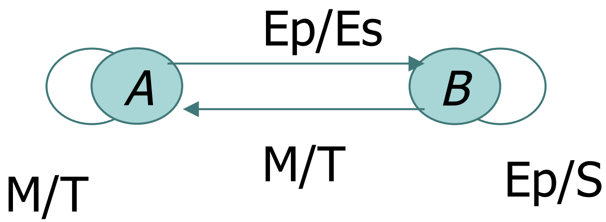


! Not all internal test cases can be translated
What is the fault coverage of what is left?

Testing the coffee machine





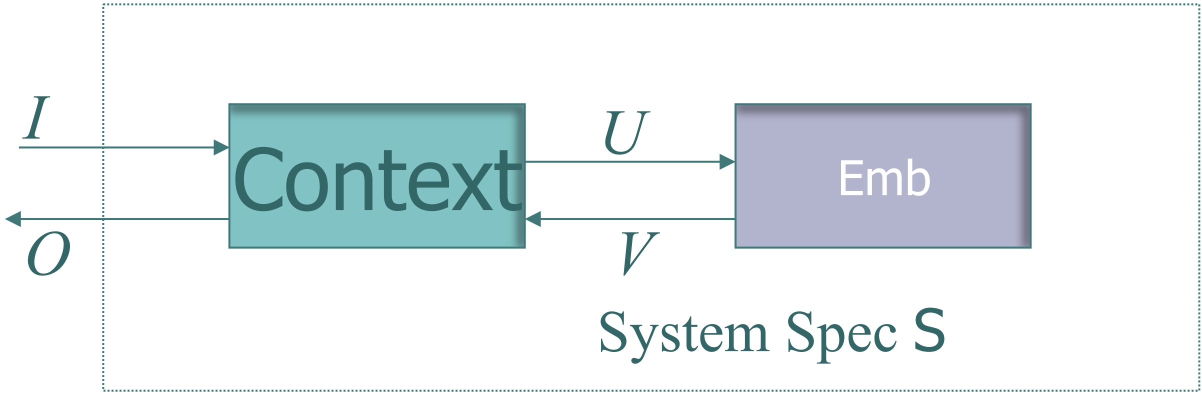


Input sequence B…. cannot be applied to the coffee machine since the Waiter always starts with C

Internal and external tests

An *internal test case* is an input sequence of the *Emb* that has traces over $(UV)^*$

An *external test case* is an input sequence of the *S* that has traces over $(IO)^*$



What can be detected by external test cases

A faulty implementation *Emb*' of the *Emb* can be detected by some external test case iff *Emb*' and *Emb* are externally non-equivalent

Solution: To derive an *Embedded Equivalent* (*EE*) of the *Emb* that contains the behavior of each FSM *Emb*' that is externally equivalent to *Emb* and only them

Then an internal test suite that detects each non-reduction *Emb*' of such *EE* can be translated into an external test case that detects (kills) the faulty *Emb*'

How to test the embedded component separately

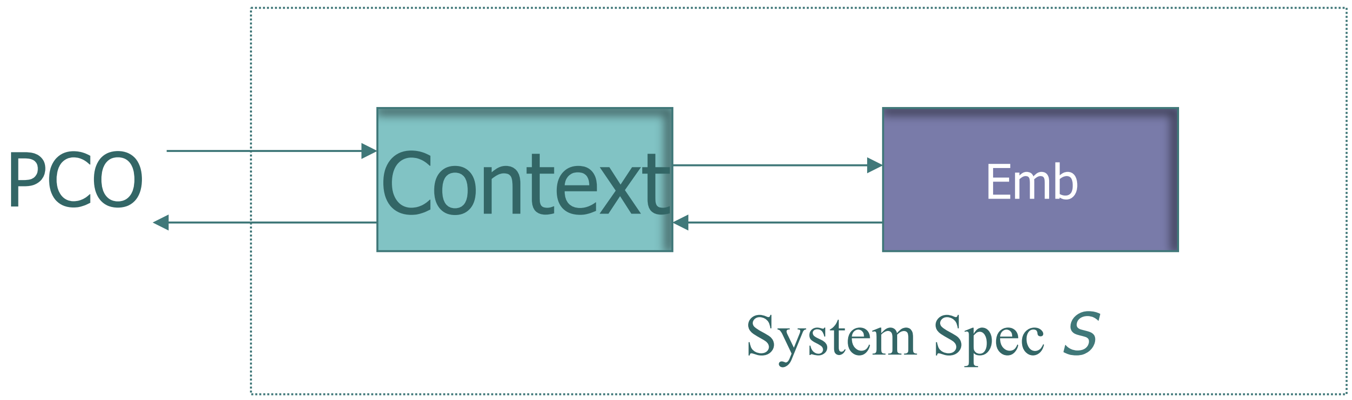
- Derive an *embedded equivalent* *EE* of the *Emb*
- Derive a complete internal test suite TS_{int} w.r.t. the fault model <*EE*, ≤, FD_{Emb}>
- Translate the internal test suite TS_{int} into an external test suite TS_{ext}

Note: the test translation problem arises

Testing by solving an FSM equation

All possible *permissible* behaviors of the *Emb* that do not change the external behavior of the composition can be captured by the general solution to the equation $Context \lozenge X \cong S$

- *EE* is the largest solution to the equation
- *EE generally is a **nondeterministic** FSM*

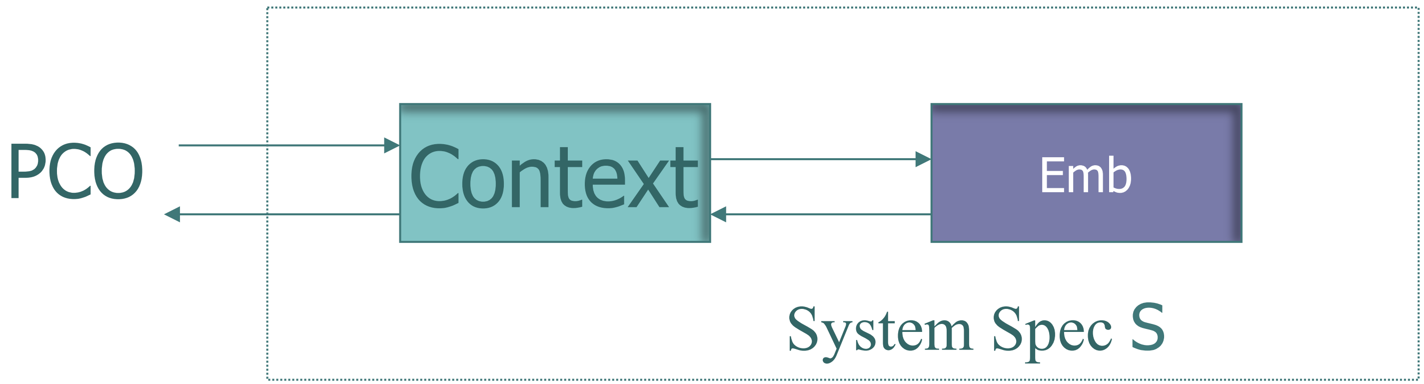


Testing by solving an FSM equation (2)

Let *EE* be the largest solution to the equation $Context \Diamond X \cong S$

An FSM Emb' can replace the FSM Emb in the composition without violating expected external outputs iff Emb' is a reduction of EE

! The largest solution describes how precisely the *Emb* behavior can be tested



Testing by solving an FSM equation

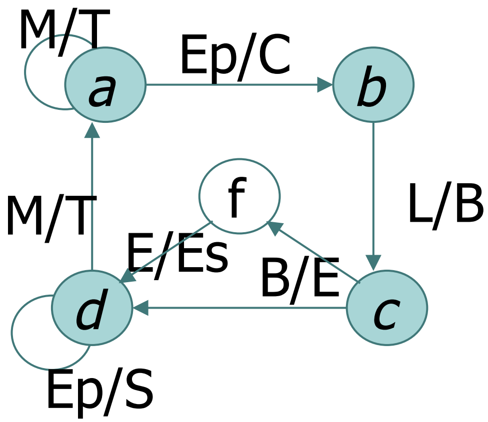
Fault model: <*EE*, ≤, FD_{Emb}> where *EE* is the largest solution to *Context* ◊ *X* ≤ *S* that contains all conforming behaviors of *Emb*

- Derive a complete test suite w.r.t. <*EE*, ≤, FD_{Emb}>
- Each internal test case can be translated it into external inputs and outputs

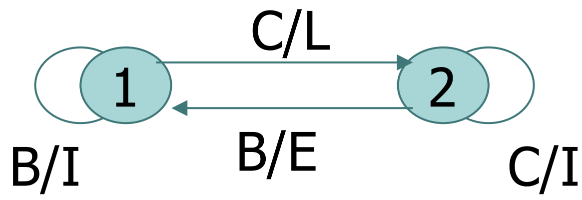
Shorter tests w.r.t. <*EE*, ≤, FD_{Emb}> are derived when *EE* has a separating (distinguishing) sequence

Parallel composition for a coffee-shop

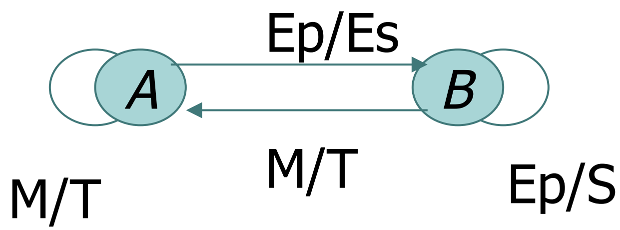
Waiter



Coffee machine



Coffee shop

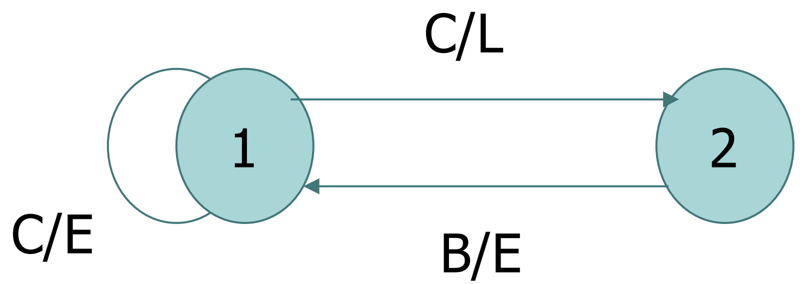


Solving the equation for the coffee machine

Solve the equation for the coffee machine

Waiter ◊ *X* ≅ *Coffee-shop*

The largest solution

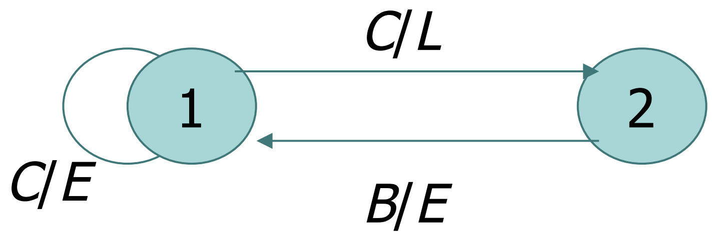


All other input sequences take the largest solution to the *DNC* state, since they cannot be applied due to the waiter behavior

Test suite derivation for a coffee machine

A complete internal test suite w.r.t the fault model <*EE*, ≤, FD_{Emb}>

The largest solution *EE*



An external test case

$E_p.E_p$

Internal test suite ($m \leq 2$)

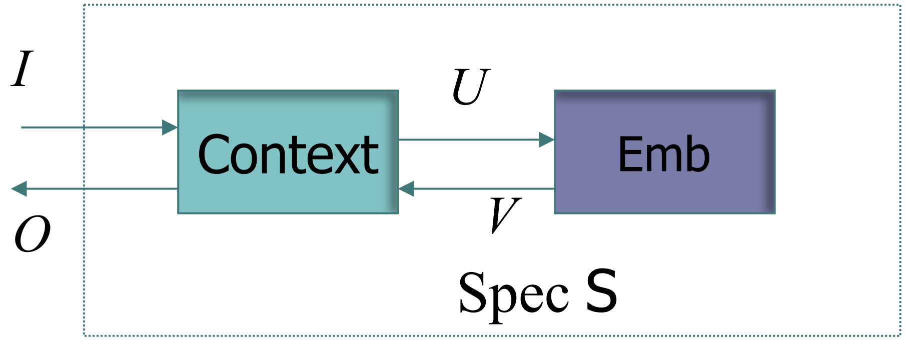
$C.C$

$C.B$

Test translation problem (formally)

- FSMs Context and Emb
- Internal test case γδ over $(UV)^*$ s.t. γ detects each embedded component implementation Emb' with the trace γδ
- We should derive an external test case α over I^* s.t. each embedded component FSM Emb' with the trace γδ is killed by α

Testing the embedded component



! Is not optimized yet

Conclusions about FSM based testing in context

- Tests which check all detectable output and transfer faults in an embedded component, can be derived as tests for a composed FSM or for a nondeterministic embedded equivalent of the component
- In both cases, tests become shorter when deterministic and nondeterministic FSMs have a **distinguishing** sequence
- One should compromise between the preciseness and testing abilities of the model
- One should consider proper classes and heuristics

Some references for FSM based testing in context

- Petrenko, A., Yevtushenko, N., Bochmann, G. v., Dssouli, R.: Testing in context: framework and test derivation, Computer communications 19, 1236-1249 (1996)
- Petrenko, A., Yevtushenko, N., Bochmann, G. v.: Fault models for testing in context. Proc. International Conference on Formal Techniques for Networked and Distributed Systems, 125-140 (1996)
- Lima, L. P.: A pragmatic method to generate test sequences for embedded systems, Ph.D. Thesis, Institute National des Telecommunications, Evry, France (1998)
- Anido, R., Cavalli, A. R., Lima, L., Yevtushenko, N. Test suite minimization for testing in context. Soft. Test. Verif. Reliab., 13 (3), 141-155 (2003)
- K. El-Fakih and N. Yevtushenko, "Fault propagation by equation solving", *Proc. of the IFIP* 24th *International Conference on Formal Techniques for Networked and Distributed Systems*, Madrid, Spain, LNCS 3235, pp. 185-198 (2004)
- El-Fakih, K., Petrenko A., Yevtushenko, N: "FSM Test Translation Through Context". 18th International Conference on Testing of Communicating Systems- TestCom 2006, New York, USA, Lecture Notes in Computer Science 3964, 245-258 (2006)

Complexity of related problems and how to decrease it…

Some primitive complexity into…

Time **Space**

…This is what it counts for an algorithm *A*…

n is the size of the input of a problem **P**

1) ***Time*** – can be considered as the number of primitive operations, in the worst case, to solve the problem

// number of transitions of the corresponding Turing machine

2) ***Space*** – can be considered as the size of memory to be used, in the worst case, to solve the problem

// the length of a tape in use of the corresponding Turing machine

What is good and what is bad?

When the time is polynomial

- There exists an algorithm that solves the problem in a polynomial time
- The problem is in P then

When the time is not polynomial

- Maybe, there exists an algorithm that verifies the solution in a polynomial time?

Then the problem is in NP

- Or maybe there exists an algorithm that solves the problem using a polynomial space?

Then the problem is in PSPACE

P is good, for small degrees of the polynomials

NP and PSPACE – not really

Is it me, who didn't fine a nice algorithm or nobody can?

The problem **P** is NP-hard / PSPACE-hard if each problem from NP / PSPACE can be reduced to it

P is not harder than any problem from NP / PSPACE

The problem **P** is NP-complete / PSPACE-complete if

- It is in NP / PSPACE
- It is NP-hard / PSPACE-hard

Completeness justifies the problem complexity

SAT is a classical example of NP-complete problem

SAT problem

The decision problem is being considered

Input: Conjunctive Normal Form (CNF) formula $f(x_1, \ldots, x_n)$
Output: Does there exist an input vector **i**, such that
$f(\mathbf{i}) \neq (0, \ldots, 0)$, i.e. f is satisfiable?

$$f(x_1, x_2, x_3) = (x_1 \vee \overline{x_2})(x_2 \vee x_3)(x_1 \vee \overline{x_3})x_1$$

is satisfiable, as $f(1, 0, 1) = 1$

The SAT problem is NP-complete

It will be used to derive distinguishing sequences…

! Is also used for proving NP-completeness of a number of problems

When testing against FSMs…

1) Reaching each FSM state *s*

↓

2) Distinguishing *s* from any other FSM state

↓

3) Traversing a single transition to check the output and final state

- 1) can be solved via an application of a homing / synchronizing sequence
- 2) can be solved via an application of a distinguishing sequence

Does there exist a distinguishing sequence?

The decision problem is being considered

DISTINUISHING problem

Input: complete deterministic FSM $S = (S, I, O, h)$, $|S| = n$

Output: Does there exist a distinguishing sequence for S?

The problem of checking the existence of a distinguishing sequence for deterministic machines is PSPACE-complete

Lee, D., Yannakakis, M., 1994

Does there exist a distinguishing sequence? (2)

The decision problem is being considered

DISTINUISHING problem
Input: complete nondeterministic FSM $S = (S, I, O, h)$, $|S| = n$
Output: Does there exist a distinguishing sequence for S?

The problem of checking the existence of a distinguishing sequence for nondeterministic machines is PSPACE-complete

Bad… very bad ‘news’

Most of the problems in Model based testing are PSPACE-complete

In particular…

The problem of checking the existence of a distinguishing sequence for complete deterministic FSMs

The problem of checking the existence of a distinguishing sequence for complete nondeterministic FSMs

The problem of checking the existence of a homing / synchronizing sequence for complete (non-)deterministic FSMs

Test sequences and checking sequences are somewhat hard to derive…

In context, it is even harder!

How to decrease the complexity?

Utilizing scalable representations allows to 'hide' the complexity
Research groups of R. Brayton, R. Jiang, A. Mishchenko, T. Villa, J. Tretmans, W. Kunz

Considering specific types of bugs in the software, i.e. specific fault models
Research groups of J. Offutt, F. Wotawa, N. Yevtushenko

Providing **effective heuristics**
Research groups of A. Zakrevskiy, H. Yenigün, R. Brayton, A. Cavalli

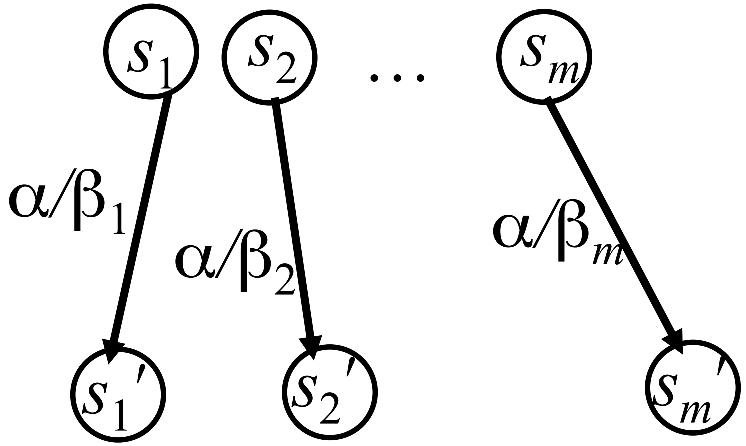
Switching **from preset to adaptive** test derivation strategy
Research groups of M. Yannakakis, A.K. Petrenko, N. Yevtushenko, A. Petrenko, R. Hierons

Each of those is good for distinguished FSM classes

Distinguishing sequence

- Distinguishing = separating for nondeterministic machines
- The sequence α allows to detect the initial state of the machine under experiment
- After applying α at any state s_i and observing an output response β_i the initial state s_i becomes known

Separating sequence α



$out(s_i, \alpha) \cap out(s_j, \alpha) = \varnothing$

Deriving a distinguishing sequence for nondeterministic FSM

$S' = \{s_1, s_2\}$

- Derive a truncated successor tree (TST)

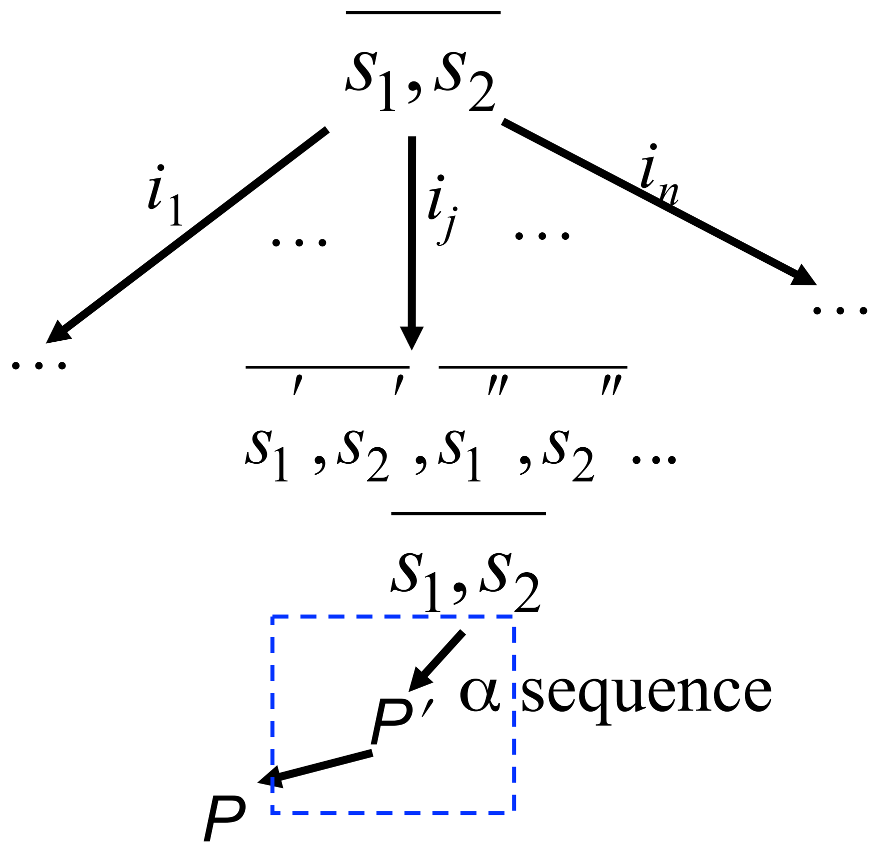
$\exists\, o_1\ ((s_1, i_j, o_1, s_1') \in h_S\ \&\ (s_2, i_j, o_1, s_2') \in h_S\ \&\ s_1' \neq s_2')$

- Truncating rules

Rule 1 *P* is the empty set

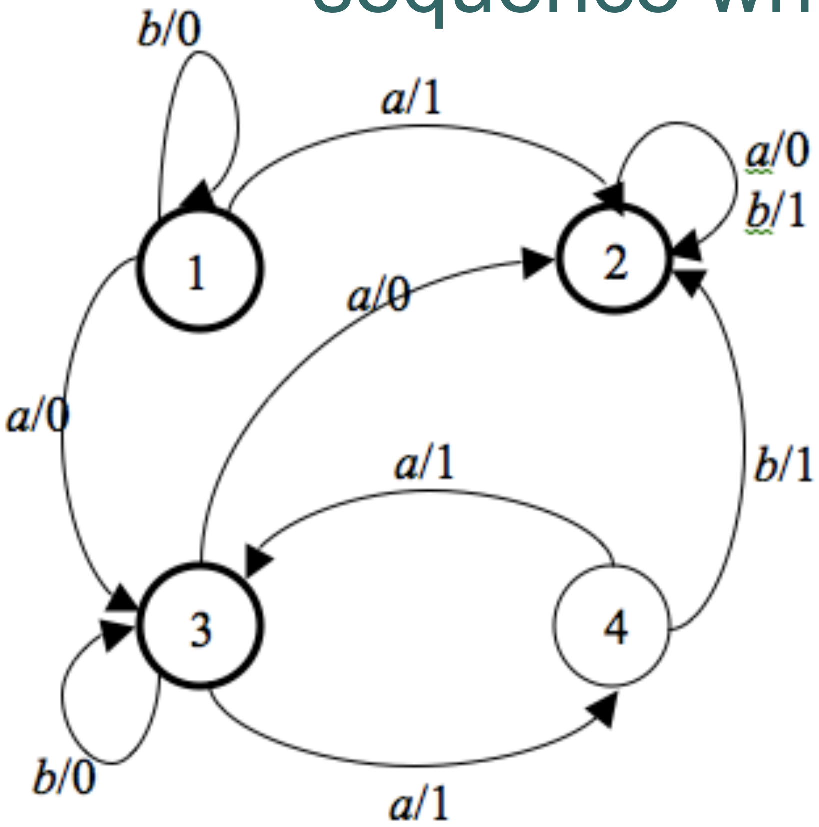
Rule 2 Set *P* contains a subset that labels another node of the path from the root to the node labeled by the set *P*

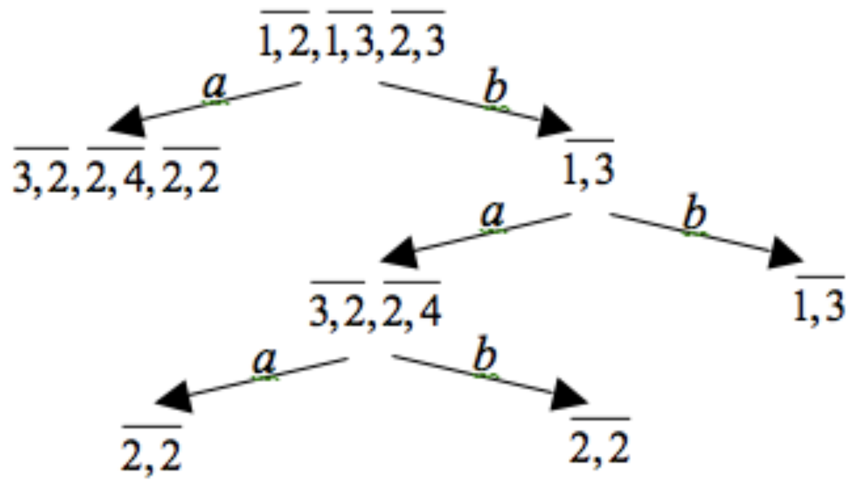
Rule 3 *P* contains singleton



α is a distinguishing sequence iff it labels the path truncated by Rule 1

Let's derive some distinguishing sequence when testing in context…





An FSM S = ({1, 2, 3, 4}, {a, b}, {0, 1}, h_S, {1, 2, 3}) and its truncated successor tree

There does not exist a distinguishing sequence for S

The length of a separating sequence

Theoretically: The length of the separating sequence has length of the order 2^{n^2}

$\Downarrow$

Very huge (More than exponential !!!) complexity, in general

$\Downarrow$

We still do not know if this upper bound is reachable (???)

$\Downarrow$

However, can we reduce the corresponding test suite (???)

Now, let's decrease the complexity

Simplifying a derivation of test sequences

1) Using scalable representations
We will see how sequential circuits and their HDL descriptions can be effectively used

2) Considering specific types of faults
We will see how specific types of FSM mutants and their HDL descriptions can simplify the thing

3) Switching from preset to adaptive test derivation strategy
We will see how some problems get into P

Scalable representations for deriving distinguishing sequence

o FSMs can be represented by sequential circuits

⇓

o FSM inputs, states and outputs are Boolean vectors

Transition relation is described by Boolean transition and output functions

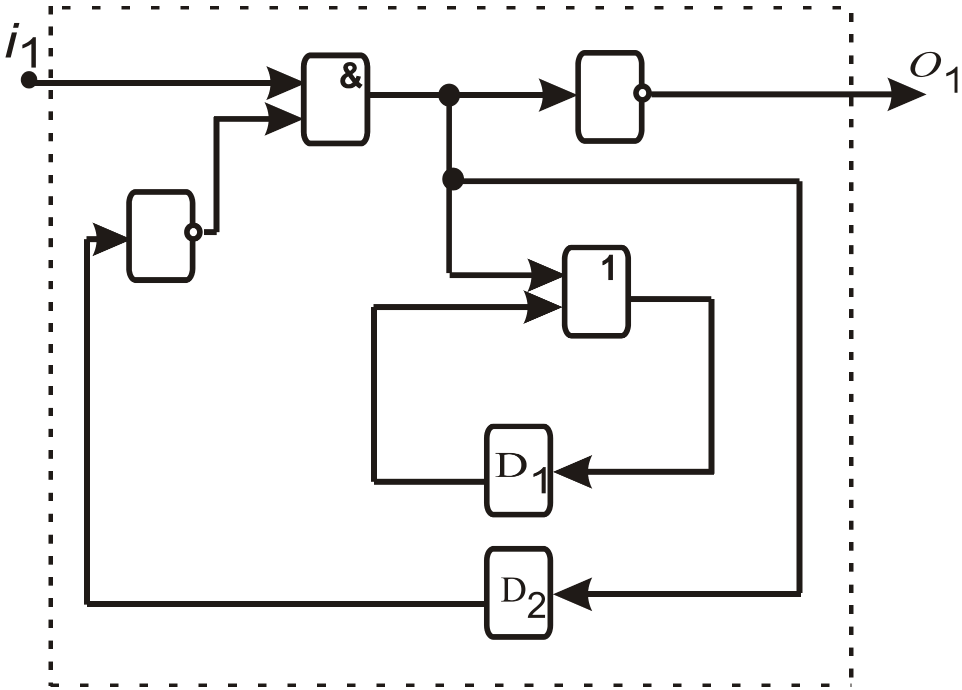
o Combinational circuits correspond to FSMs with a single state

⇓

Idea : to build a distinguishing sequence for two (or more) sequential circuits

Scalable representations for FSMs

A sequential circuit



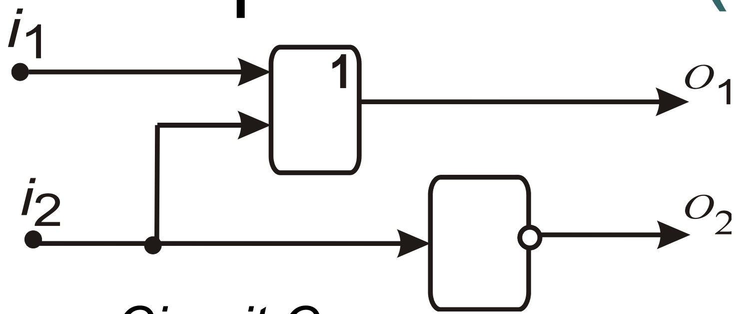
FSM state = set of latch states

Corresponding FSM

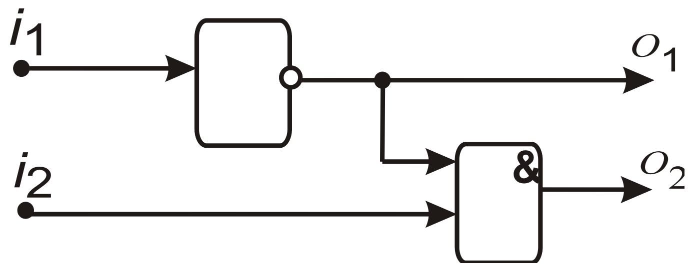
i \ s	00	01	10	11
0	00/1	00/1	10/1	10/1
1	11/0	00/1	11/0	10/1

FSM input/output = PI/PO

Miter based circuit equivalence checker (as an example)

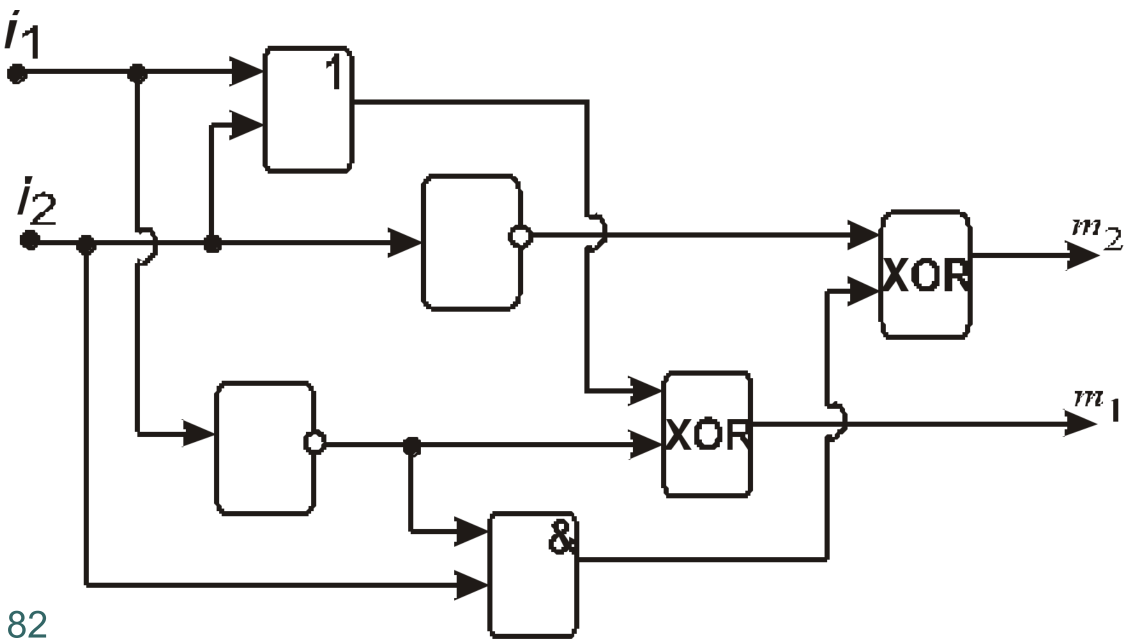


Circuit C



Circuit C′

Miter M combines both circuits, connecting the outputs with a XOR gate



Circuits C and C′ are equivalent if CNFs, corresponding to m_1 and m_2 are UNSAT

Checking the equivalence of two FSMs

o Given two FSMs S_1 and S_2, represent them as corresponding sequential circuits C_1 and C_2

o Derive combinational equivalents of length l for C_1 and C_2 (correspond to l-equivalents of the machines S_1 and S_2)

o Derive a miter for these l-equivalents

o Solve the SAT problem for the outputs of the miter

$\Downarrow$

If the answer is UNSAT then circuits are equivalent

Otherwise, a counter example is produced

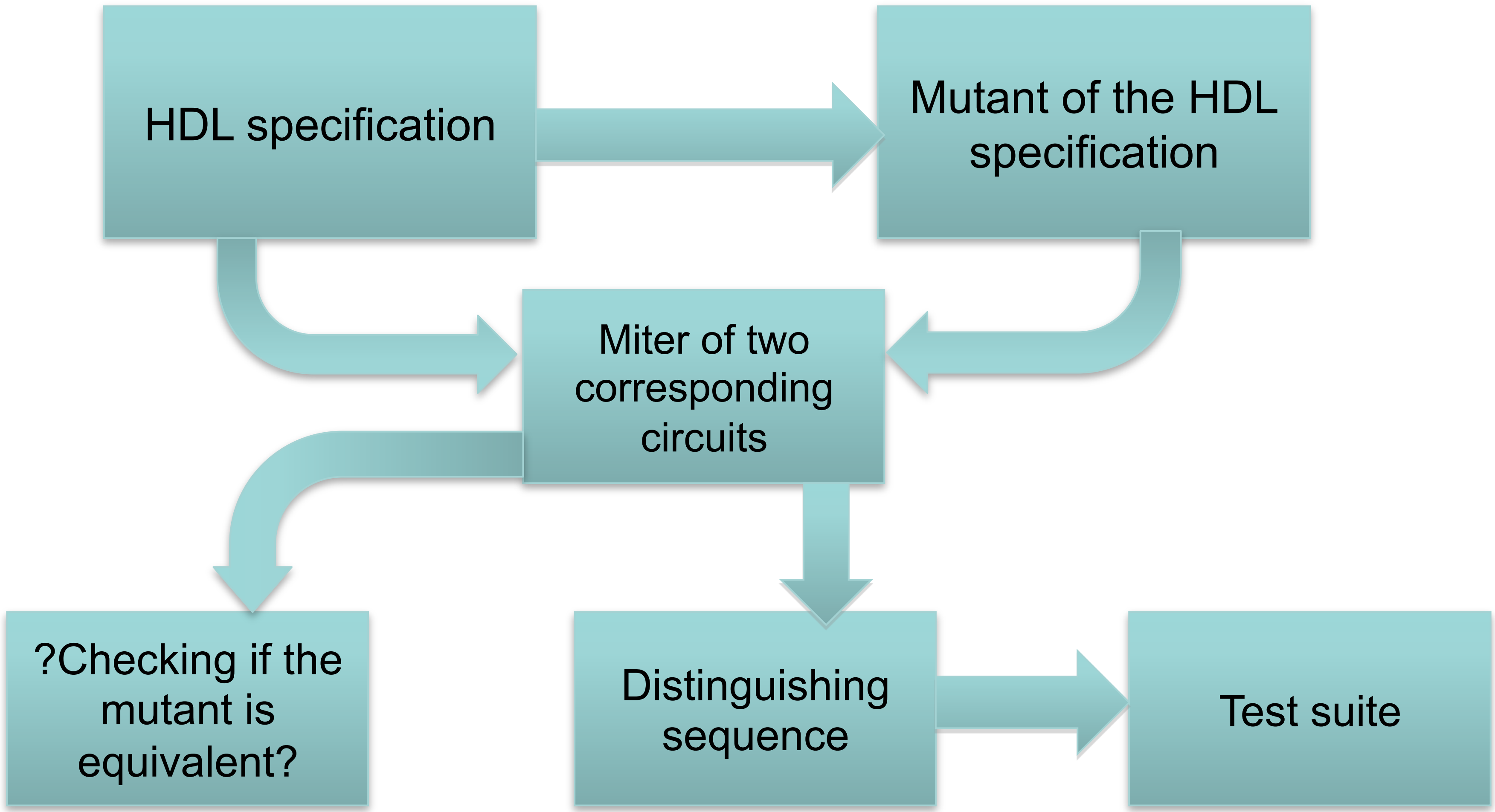
NOTE : counter example is a sequence that distinguishes S_1 and S_2

Idea for effective test derivation

- Given an embedded component *Emb*
- Describe the *Emb* behavior in some Hardware Description Language (*HDL*)
- Derive most probable HDL mutants for *Emb*
- Locate all the mutants in the fault domain *FD*
- Distinguish each mutant from the Emb HDL specification
- Add the corresponding distinguishing sequence into the test suite

We derive tests with the guaranteed fault coverage w.r.t. the fault model <S, ≈ , FD>

Deriving distinguishing sequences based on HDL specifications



Considering specific faults / mutants

- When testing in context, under a white box assumption: specification FSM *S* can be <u>initialized</u>, but

<u>nondeterministic, partial, possibly non-observable</u>

- Let's enumerate only those faults that are more likely to appear in implementation (!!!that is how we decrease the complexity!!!)

- By changing an output or a transition in *S*, one obtains a mutant $\mathcal{M}$

- Set of all mutants is a fault domain *FD*

We derive tests with the guaranteed fault coverage w.r.t. the fault model <S, ≈ , FD>

How to derive tests

We derive tests with the guaranteed fault coverage w.r.t. the fault model <S, ≈, FD>

$\Downarrow$

- One has to know how to distinguish between

  S and $\mathcal{M} \in$ ***FD***

- A distinguishing sequence for a direct sum $\boldsymbol{S} \oplus \mathcal{M}$ needs to be derived

- The direct sum $\boldsymbol{S} \oplus \mathcal{M}$ can be nondeterministic, partial, and possibly non-observable

Deriving an input sequence distinguishing (separating) two initial states of $S \oplus M$

$S' = \{s_1, s_2\}$

- Derive a truncated successor tree (TST)

Transitions under i_j are defined at both states s_1 and s_2

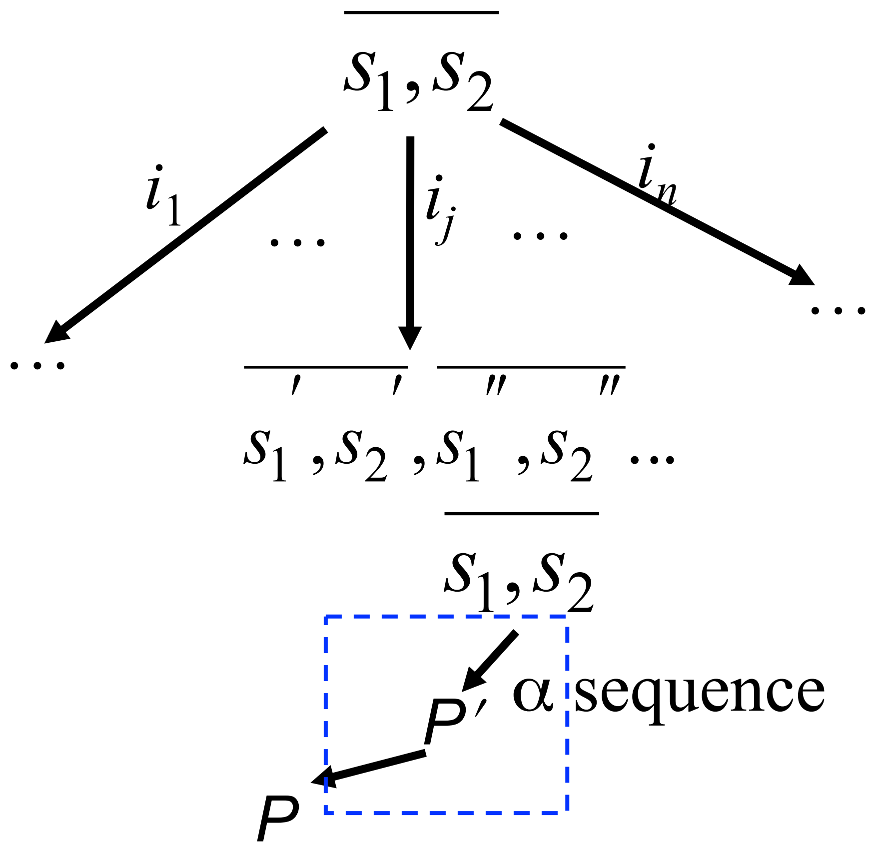
$\exists\, o_1\ ((s_1, i_j, o_1, s_1',) \in h_S\ \&\ (s_2, i_j, o_1, s_2') \in h_S\ )$

- Truncating rules

Rule 1 *P* is the empty set

Rule 2 Set *P* contains a subset that labels another node of the path from the root to the node labeled by the set *P*

Rule 3 *P* contains singleton



α is distinguishing ⇔ it labels the path truncated by Rule 1

Deriving a test suite *TS* w.r.t. <*S*, ≈, *FD*>

Input: FSM S that can be partial and non-observable

Output: A test *TS* for *S* or a corresponding message

Step 1 $i = 0$

Step 2

Derive a mutant M_i in the lexicographical order for the FSM *S*

Derive a separating sequence α for an FSM $M_i \oplus S$ // direct sum

If there is no separating sequence for the FSM $M_i \oplus S$, then

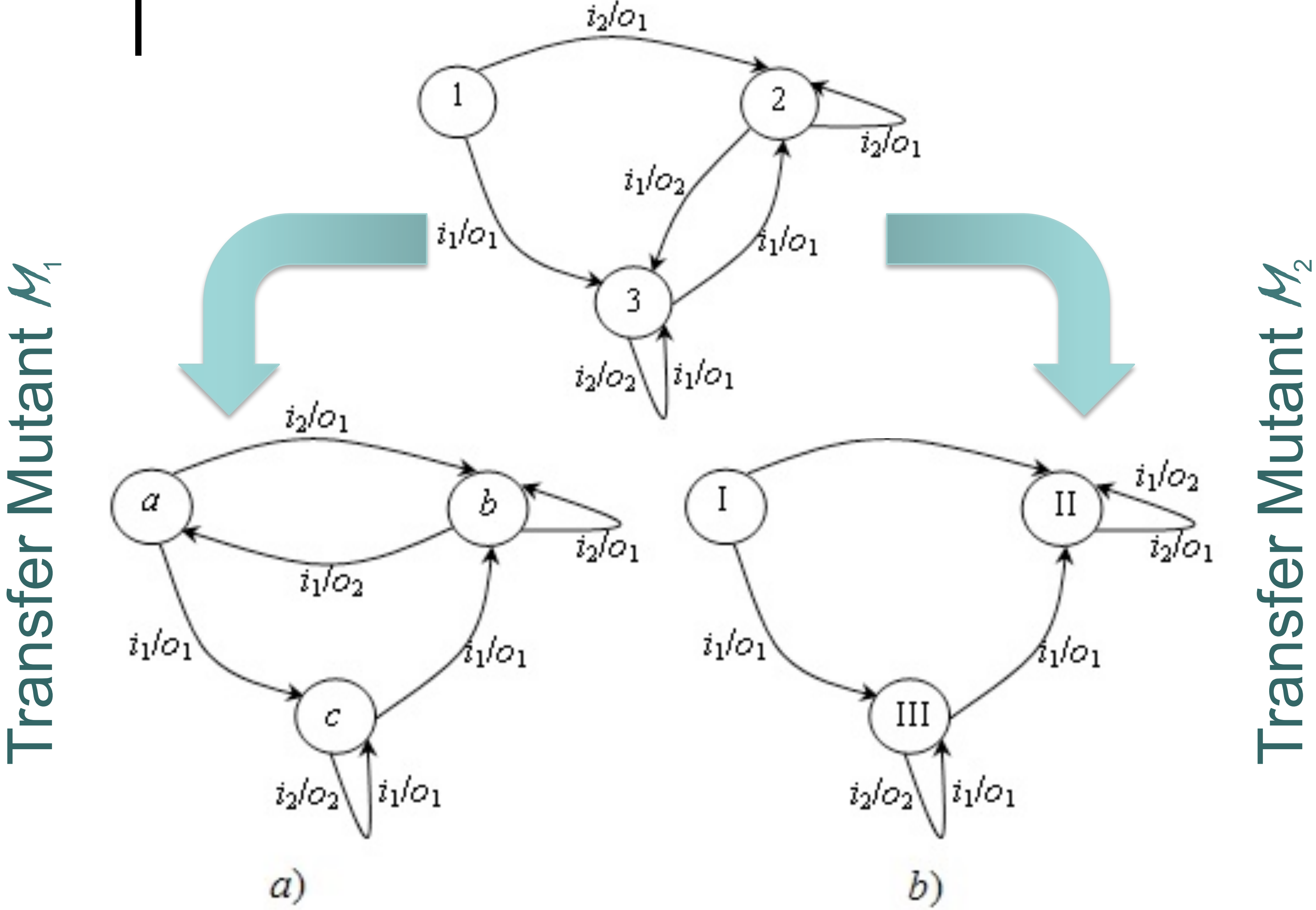
Return a corresponding message

Otherwise,

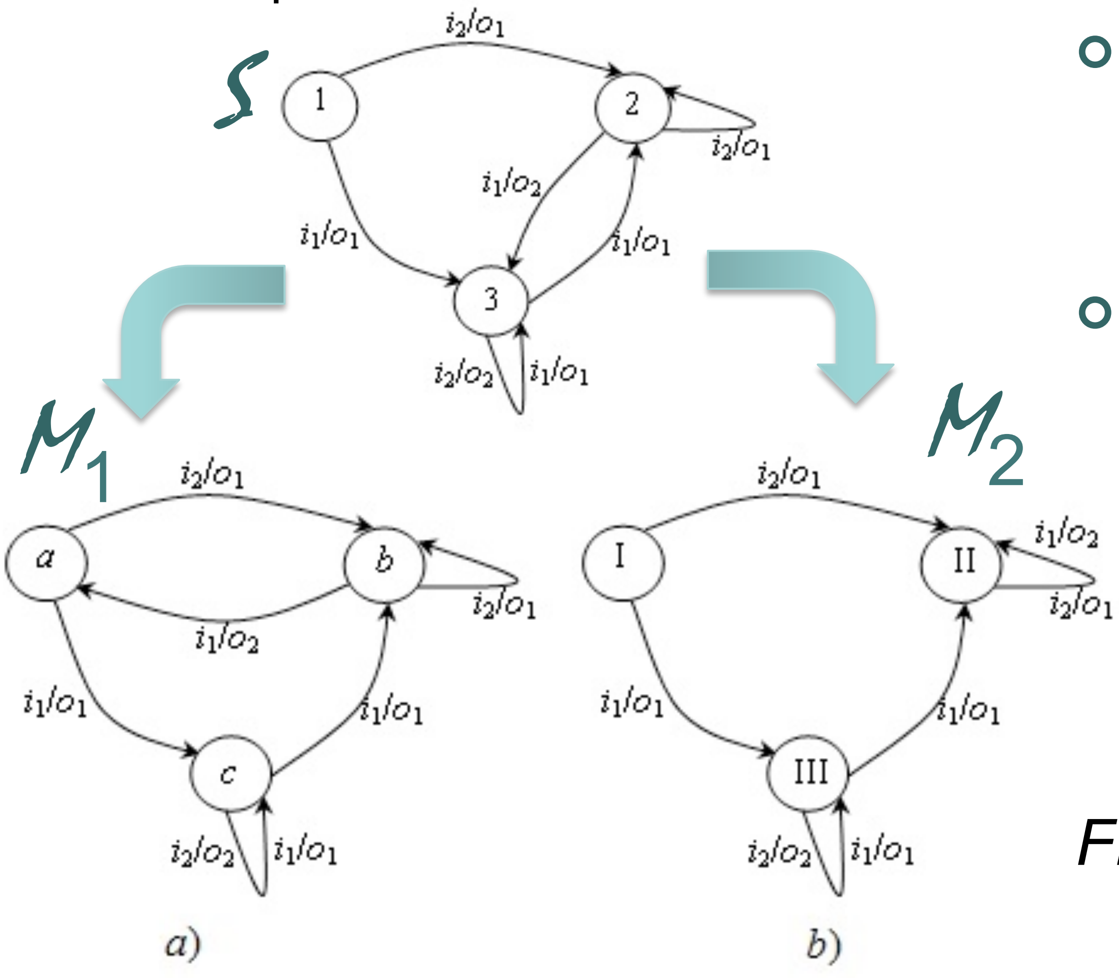
If $\alpha \notin TS$ then add α into the test suite *TS*

i++, and go to Step 2

For example…


Transfer Mutant $\mathcal{M}_1$
Transfer Mutant $\mathcal{M}_2$
i_2/o_1
1
2
3
i_1/o_2
i_1/o_1
i_2/o_2
a
b
c
I
II
III
a)
b)

For example (cont-d)…



- A distinguishing sequence for $S \oplus M_1$ can be

  $\alpha = i_2 i_1 i_2$
- Moreover, $\alpha = i_2 i_1 i_2$ is a distinguishing sequence for $S \oplus M_2$

The test suite *TS* for $FD = \{M_1, M_2\}$ is $TS = \{i_2 i_1 i_2\}$

How trees can be used for testing (in context)

- What if *S* has a specific structure?
- *S* can still be
  - Complete or partial
  - Deterministic or non-deterministic
  - Observable or non-observable

BUT: *S* diagram has a tree structure!

This fact simplifies the test derivation

When S has a tree structure

Input: FSM S with a tree diagram

Output: A test *TS* for *S* or a corresponding message

Step 1 $i = 0$

Step 2 Derive a mutant M_i in the lexicographical order for the FSM *S*

Derive a separating sequence α for an FSM $M_i \oplus S$

by covering the faulty transition and going down the branch until M_i and S output reactions do not coincide

If there is no separating sequence for the FSM $M_i \oplus S$, then

Return a corresponding message

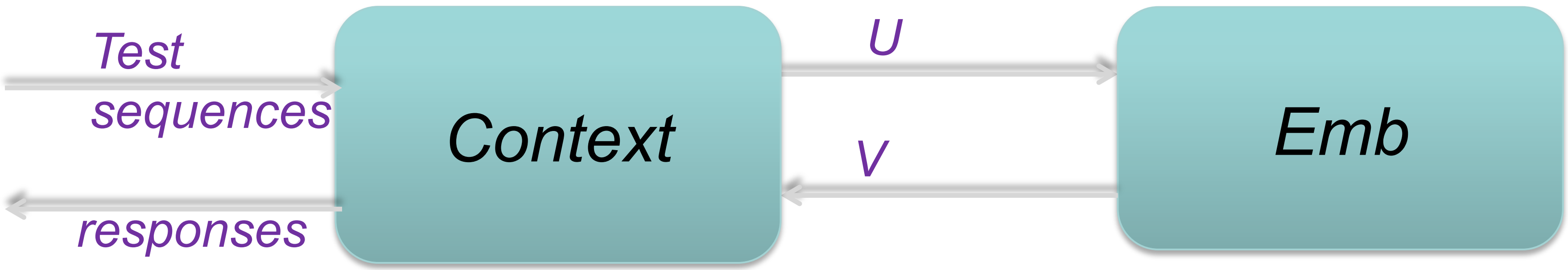
Otherwise,

If $\alpha \notin TS$ then add α into the test suite *TS*

i++, and go to Step 2

Any other restrictions on the Context

What if the initial state of the Context is unknown?



Channels *U* and *V* are not observable

$\Downarrow$

FSM specification for Context is nondeterministic

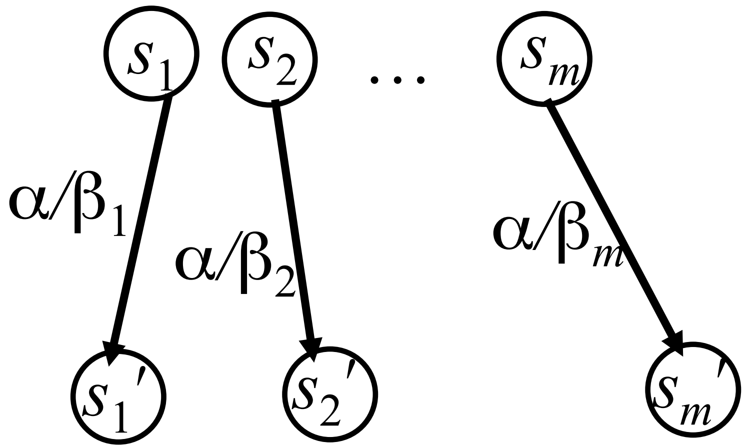
$\Downarrow$

One needs to first set the Context into a known state and then apply the test sequences

Homing sequences for nondeterministic FSMs

- The sequence α allows to detect the final state of the machine under experiment after α application
- After applying α at any state s_i and observing an output response β_i the final state s_i' becomes known

Homing sequence α

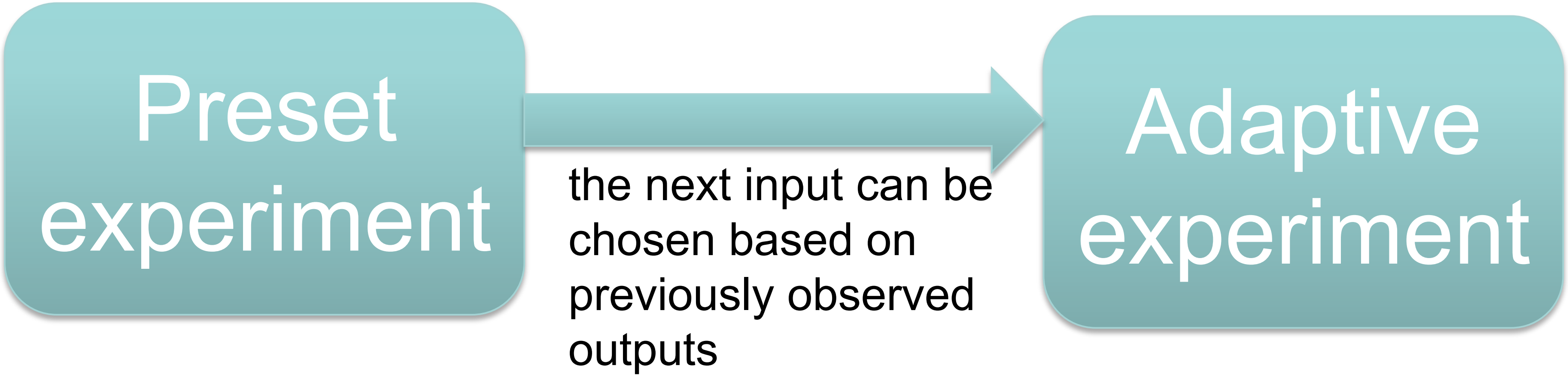


apply α + observe β_i + draw a conclusion about s_i'

BUT! We saw the very (!!!) huge complexity,

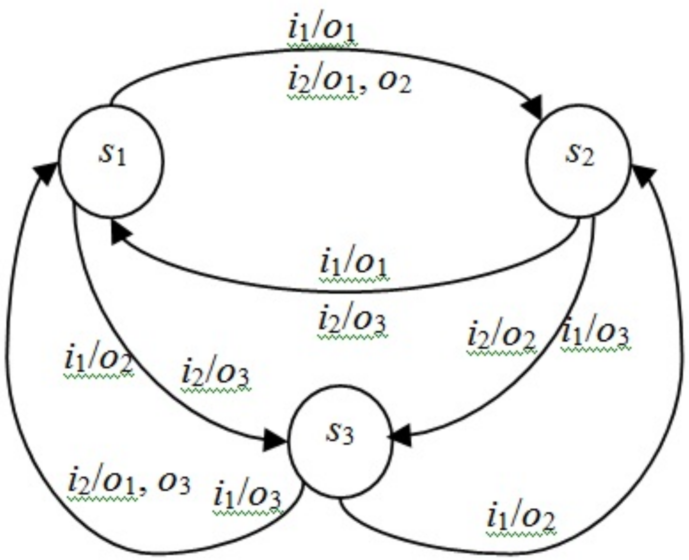
so… let's just decrease it

How to decrease the complexity through adaptive experiments



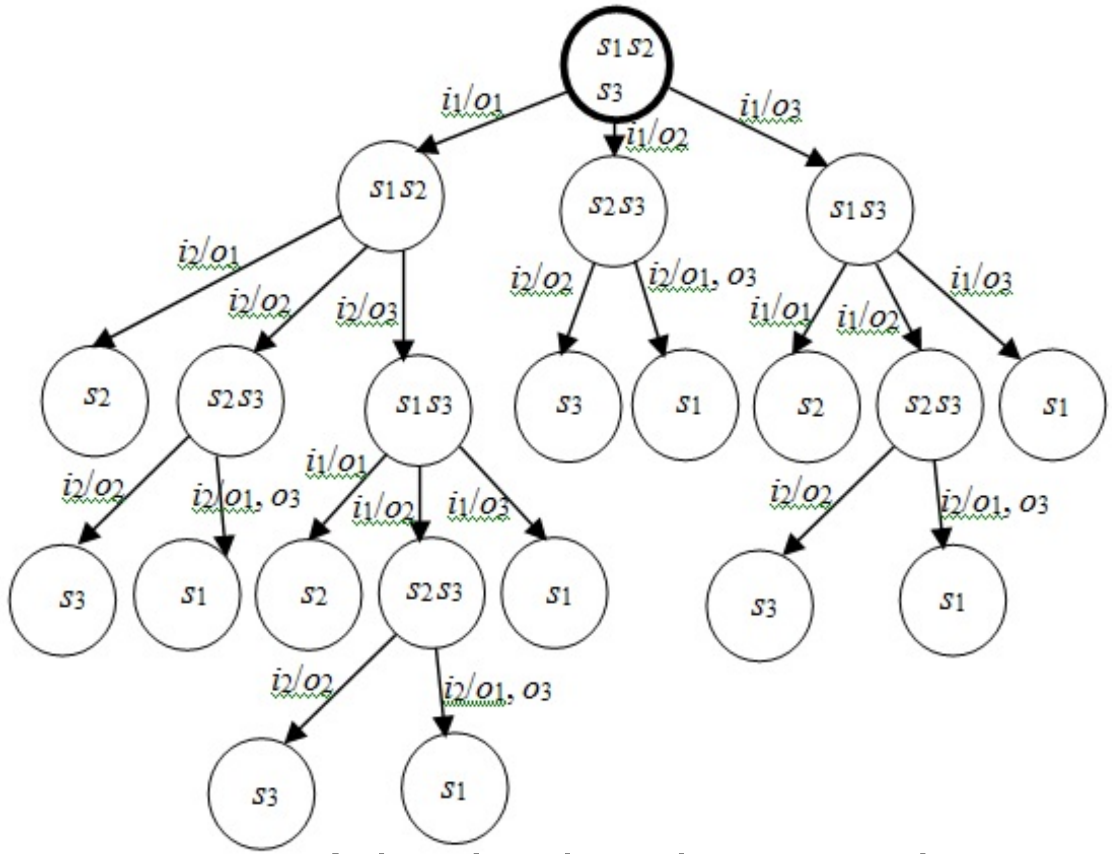
- The experiment is now represented by a ***Test Case***
- A ***Test Case*** is a connected single-input output-complete observable initialized FSM with the acyclic transition graph

Example for homing experiment



Nondeterministic FSM

! No homing sequence



Adaptive homing experiment for nondeterministic FSM

Problem of existence of a homing test case for a complete nondeterministic FSM

ADAPTIVE HOMING problem

Input: complete observable nondeterministic FSM
$\mathcal{S}$ = (S, I, O, h), |S| = n
Question: does there exist a homing test case for the FSM $\mathcal{S}$?

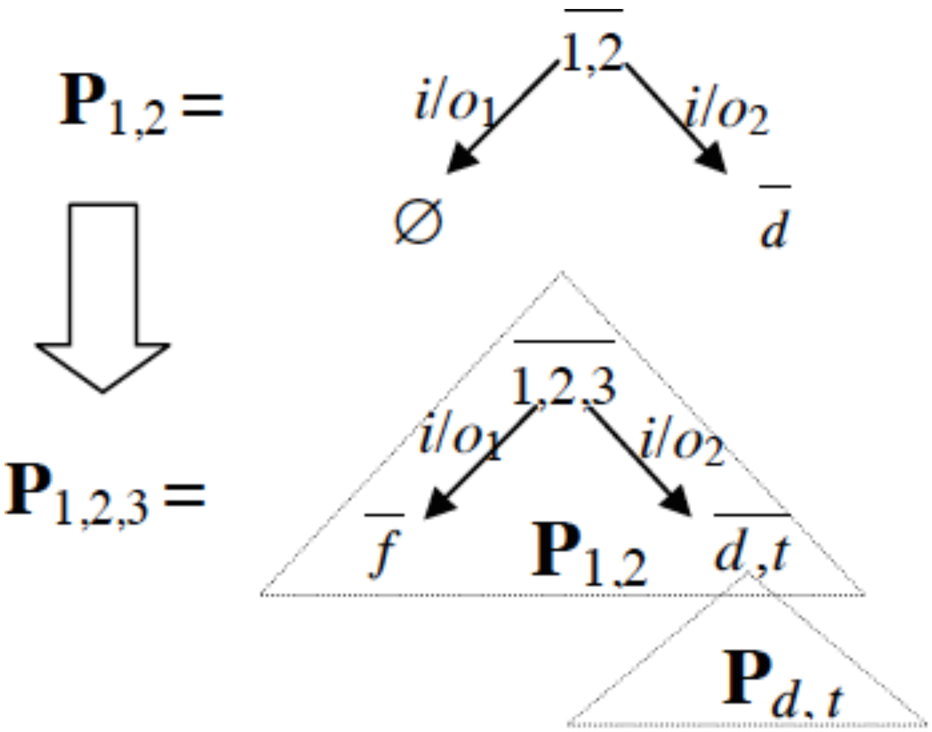
The problem can be reduced to that of checking if there exists a homing test case for each pair of FSM states

Theorem. Adaptive Homing Problem for a complete observable FSM S = (S, I, O, h), |S| = n, is in P

Deriving homing test cases for nondeterministic FSMs

Given an FSM S = (S, I, O, h), $|S| = n$, such that each state pair (i, j) is adaptively homing

- We build the test case iteratively starting from the pair (1, 2) of states
- We add other states one by one, to the set of initial states (the root of the tree)
- Test cases of a type $\mathbf{P}_{i,j}$ are used at each step



The height of the homing test case for S does not exceed $O(n^3)$

Conclusions

- Theoretically: almost all the problems in software testing and, moreover, testing in context that provide the guaranteed fault coverage have terrible (exponential or more!!!) complexity
- Practically: methods and tools for decreasing the complexity seem to be promising

$\Downarrow$

New models (or new heuristics) need to appear and new methods and tools need to be provided to decrease the complexity

$\Downarrow$

We do have something for the future work ☺

Never alone…

Original results presented here were obtained in collaboration with research groups lead by

Prof. Ana Cavalli,

Prof. Khaled El-Fakih,

Profs Alexandre Petrenko & Alexandr K. Petrenko (both, ☺),

PhD Stas Torgaev

…

Thank you!