Software testing: Finite State Machine based test derivation strategies

Nina Yevtushenko
Institute for System Programming RAS, Russia
evtushenko@ispras.ru

*Some results have been presented in TAROT (Training And Research On Testing) summer schools

Types of testing

- **Conformance testing**
- **Security testing**
- **Performance testing**
- **...**

In this lecture, we focus on tests for checking functional requirements, i.e., on conformance testing

Conformance testing

```
int f(int *a, int size_a)
{
int i, m;
i = 0;
m = a[0];
while(i < size_a)
{
if(m < a[i]) m = a[i];
i++;
}
return m;
}
```

The function returns the maximal integer in the array *a* where *size_a* is the dimension of *a*

* A number of functional faults are not detected through static analysis

Code coverage

ITC'99 benchmarks (second release)

Mutant Coverage, MC

Statements/branches Coverage, HC

Benchmark	***MC (%)***	***HC (%)***
b01	75,35	100/100
b02	81,33	100/100
b03	68,92	73,21/76
b06	76,92	100/100
b07	1,8	93,93/94,73
b08	45,68	100/100
b09	2,29	100/100
b10	39,84	100/100

Conformance testing

```
int f(int *a, int size_a)
{
int i, m;
i = 0;
m = a[0];
while(i < size_a)
{
if(m < a[i]) m = a[i];
i++;
}
return m;
}
```

The function returns the maximal integer in the array *a* where *size_a* is the dimension of *a*

* A number of functional faults are not detected through static analysis

Solution: to check the behavior applying input sequences

Conformance testing

```
int f(int *a, int size_a)
{
int i, m;
i = 0;
m = a[0];
while(i < size_a)
{
if(m < a[i]) m = a[i];
i++;
}
return m;
}
```

The function returns the maximal integer in the array *a* where *size_a* is the dimension of *a*

How to check that the function is correctly implemented?

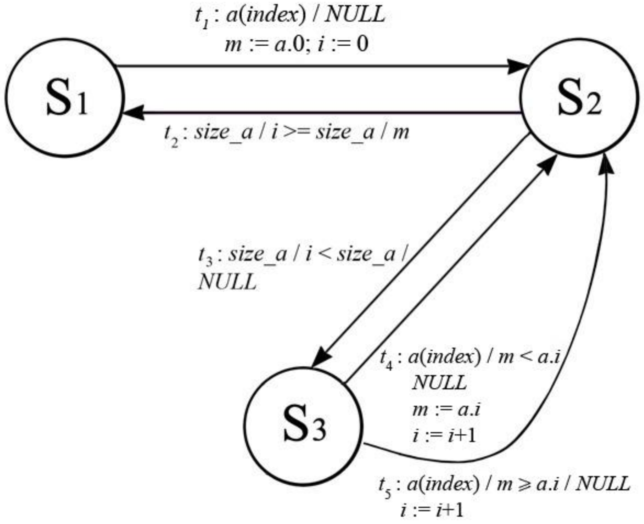
How many arrays should be checked?

Is it enough to check all the arrays of dimension 3?

Solution: to use formal models

```
int f(int *a, int size_a)
{
int i, m;
i = 0;
m = a[0];
while(i < size_a)
{
if(m < a[i]) m = a[i];
i++;
}
return m;
}
```

EFSM

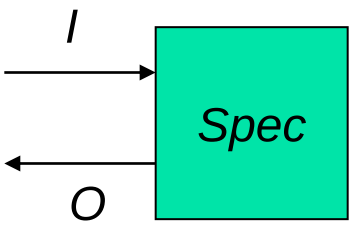


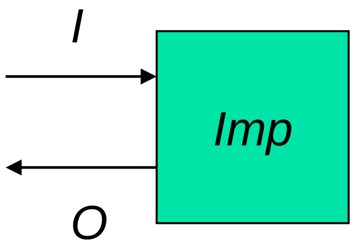
Model based testing

Extract

- A Formal Specification ***Spec*** (requirements) of the System
- Formally describe a set of faulty implementations

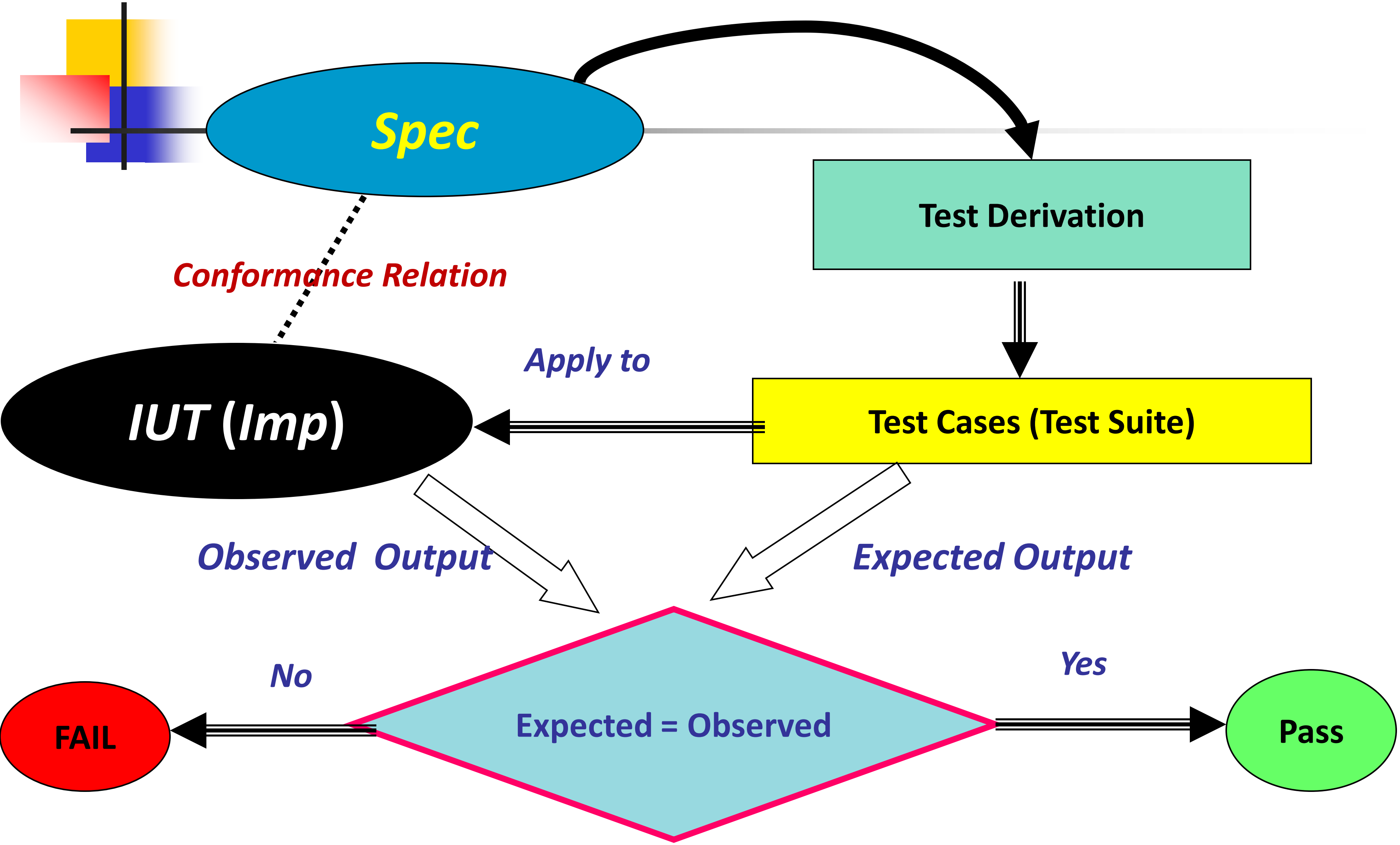
Derive a finite set of finite input sequences (*Test Suite*) such that after applying them to IUT ***Imp*** we can guarantee that ***Imp*** *conforms* to ***Spec***





- ***Conforms*** **has many definitions depending on the Formal Specification and should be formally defined**

Conformance Testing



FAULT MODEL in Conformance Testing

$< Spec, \Re, FD >$

Formal Specification

Conformance relation

Fault Domain

All Faulty Implementations (explicitly or implicitly described)

Questions

Two questions arise

1. How are the specification and an implementation formally described?
2. What does this mean «an implementation *conforms* to its specification»?

Spec and *Imp* are described using the same formal model (usually a system with finite number of states and/or transitions)

Testing with guaranteed Fault Coverage

< *Spec*, $\Re$, *FD* >

Formal Specification

Conformance relation

Fault Domain

All Faulty Implementations (explicitly or implicitly described)

Guaranteed Fault Coverage:

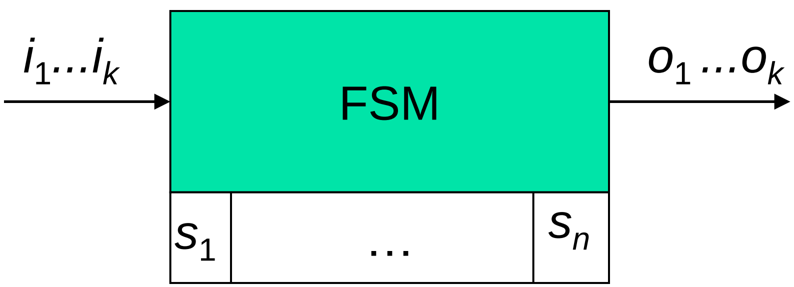
A *complete* test suite w.r.t. <***Spec***, $\Re$, ***FD***> has to detect each ***Imp*** $\in$ *FD* such that *Imp* does not conform (i.e., not equivalent, not reduction, etc.) to ***Spec***

FSM-based conformance testing

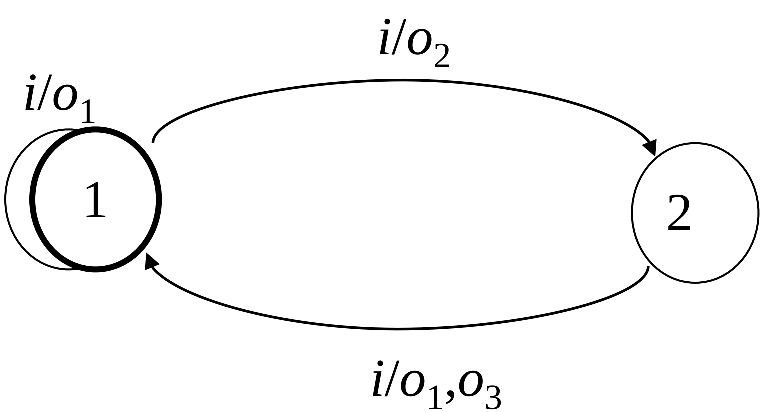
1. *Spec* and *Imp* are Finite State Machines (FSMs)
2. *Imp* *conforms* to *Spec* iff *Spec* and *Imp* have the same behavior

Finite State Machines (FSMs)

Initialized FSM



S is a finite set of *states* with the *initial* state s_1
I is a finite non-empty set of *inputs*
O is a finite non-empty set of outputs
h_S is a *transition* (behavior) relation



(s, i, o, s') is a *transition* from state s under input i to state s'

FSM traces are I/O sequences at the initial state

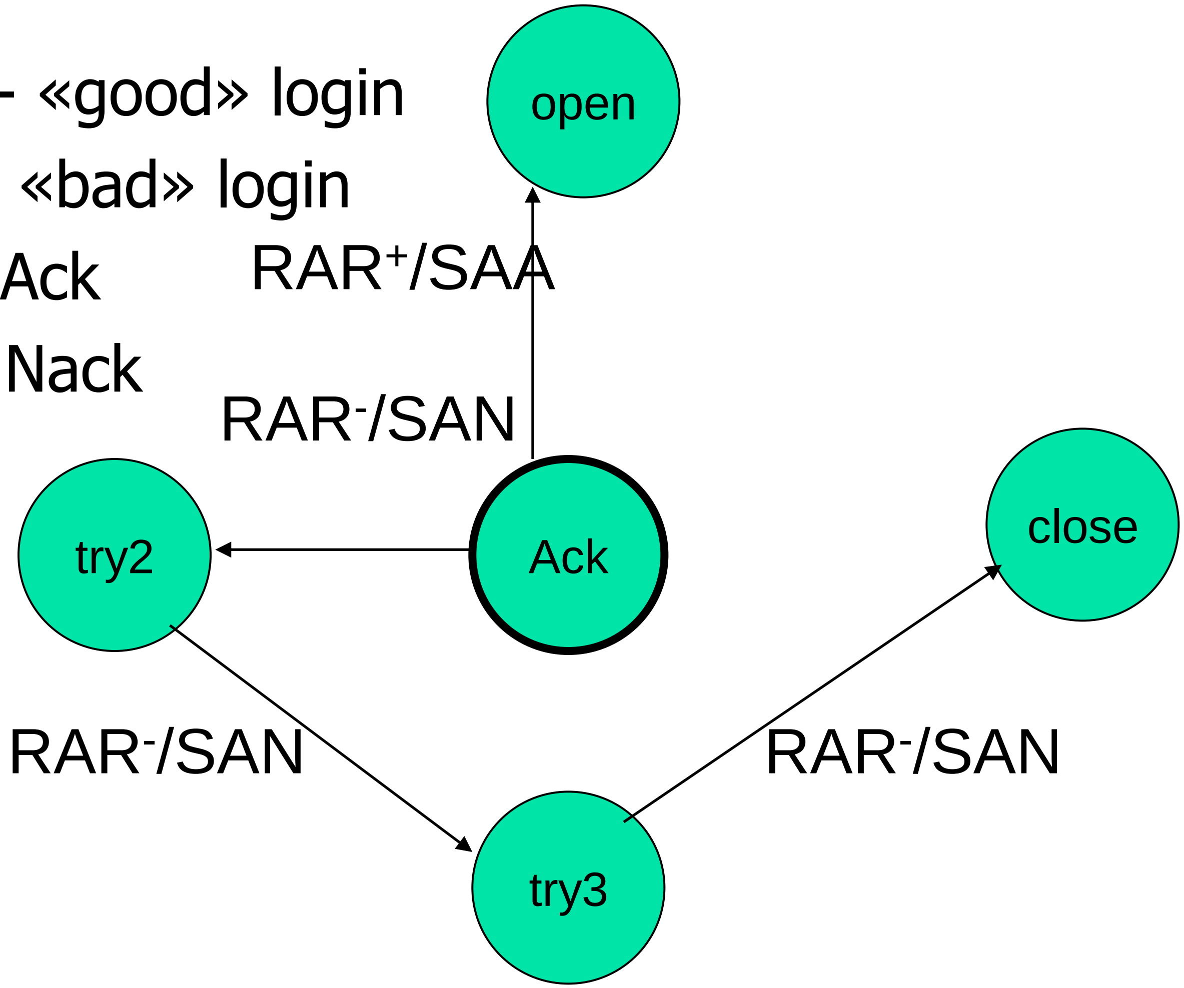
Password Authentification protocol (PAP)

RAR$^+$ - «good» login

RAR$^-$ - «bad» login

SAA - Ack

SAN - Nack



FSMs can be

- *Complete*

There is a transition for EACH input at EACH state

- *Partial*

There is no transition for SOME input at SOME state

- *Deterministic*

There is a single transition for an input at EACH state

- *Non-deterministic*

There are several transitions for for SOME input at SOME state

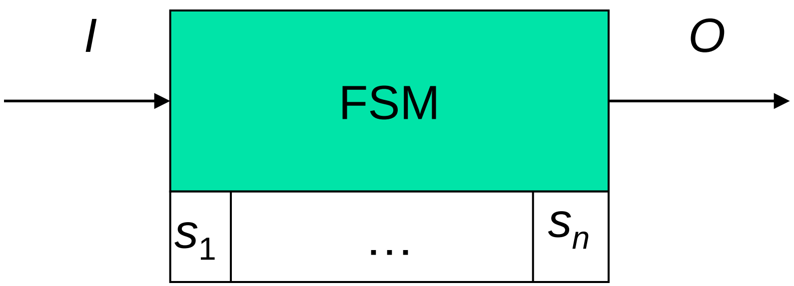
- *Initialized* FSMs have the designated initial state

Reliable reset is usually assumed

! We start with initialized deterministic complete FSMs

Complete deterministic FSMs

Initialized deterministic complete FSM is a 5-tuple $(S, I, O, \delta_S, \lambda_S, s_1)$



S is a finite set of states with the initial state s_1
I is a finite non-empty set of inputs
O is a finite non-empty set of outputs

transition function $\delta_S(s, i)$
*output f*unction $\lambda_S(s, i)$

(s, i, o, s') is a transition from state s under input i to state s'
with the output o if $\delta_A(s, i) = s'$ and $\lambda_A(s, i) = o$
s is the *initial* state of the transition
s' is the *final* state of the transition
o is the *output* of the transition

! At each state for each input sequence there is a single output sequence

Faults when deriving tests based on a complete deterministic FSM

In fact, a *fault* is an FSM s.t. its behavior is different from that *of Spec*

If faults do not increase the number of states then in *Spec* we can consider

- *Output* faults

If the *transition output* is different from that of *Spec*

- *Transition* faults

If the *transition destination state* is different from that of *Spec*

- *Mixed* faults

When both output and transition faults are possible

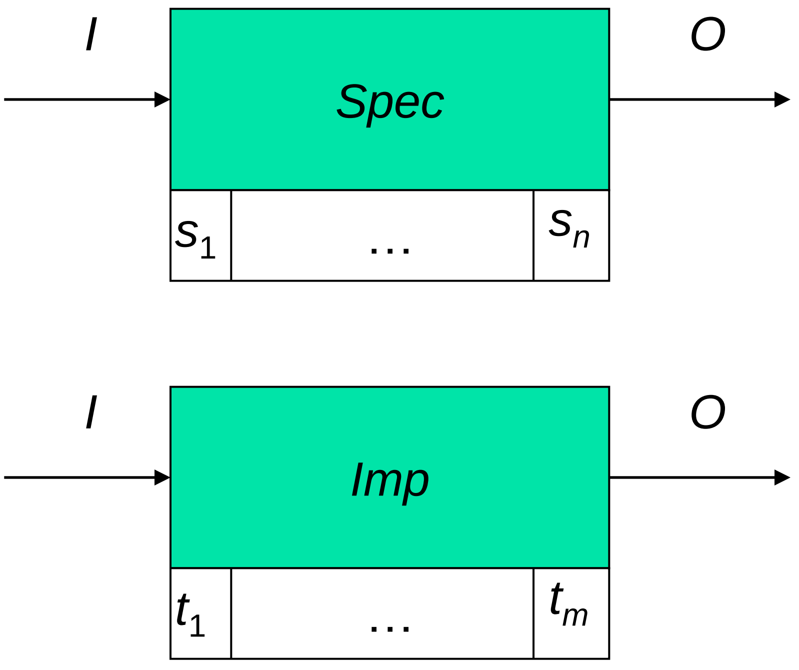
* For non-deterministic partial FSMs, there exist more fault types

Equivalence relation

FSMs *Imp* and *Spec* are equivalent if their output replies to each input sequence coincide

Caution: Number of input sequences is infinite, while we can apply only finite number of input sequences when testing the conformance

Equivalent FSMs have the same set of traces, i.e., the same behavior



FSM-based conformance testing

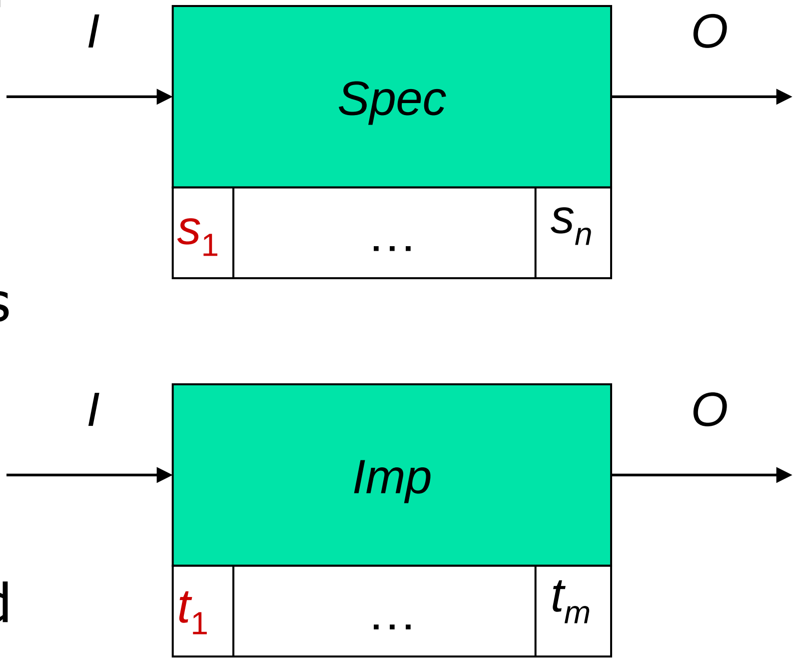
1. *Spec* and *Imp* are Finite State Machines
2. *Imp conforms* to *Spec* iff *Spec* and *Imp* are equivalent, i.e., have the same behavior

! The main problem: how to check the equality for infinite number of input sequences when applying finite number of sequences

Test Suite

- A *test case* is a finite input sequence of the specification FSM *Spec*
- A *test suite* is a finite set of test cases
- We assume that each implementation FSM *Imp* has a reliable reset *r* that takes the *Imp* from each state to the initial state
- Each test case in the test suite is headed by *r*, i.e., is applied to *Imp* at the initial state

Specification and implementation FSMs



Complete test suite

Fault model < *Spec*, $\cong$, *FD* > where *Spec* is a deterministic initialized complete FSM

Fault domain *FD* is the set of FSMs that describe all possible faults when implementing the specification

$$FD = \{Imp_1, \ldots, Imp_n, \ldots\}$$

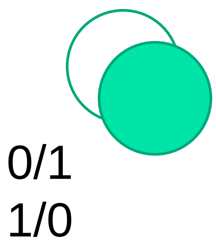
A test suite *TS* is *complete* w.r.t. *FM* if *TS* detects each FSM *Imp* $\in$ *FD* that is not equivalent to Spec

! If the fault domain contains each FSM over alphabets *I* and *O* and *Spec* is complete and deterministic then there is no complete test suite w.r.t. such fault domain

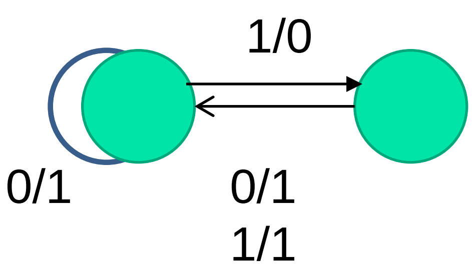
Example

Inverter

FSM *Spec* with a single state



FSM *Imp* with two states



Complete tests

- Complete test when *Imp* has a single state

{01} or {10}

- Complete test when *Imp* has at most two states

{01, 10, 00, 11}

! Nothing can be deleted

Conclusion: a complete test significantly depends on the number of states of *Imp*

Straightforward approach

Straightforward test derivation approach

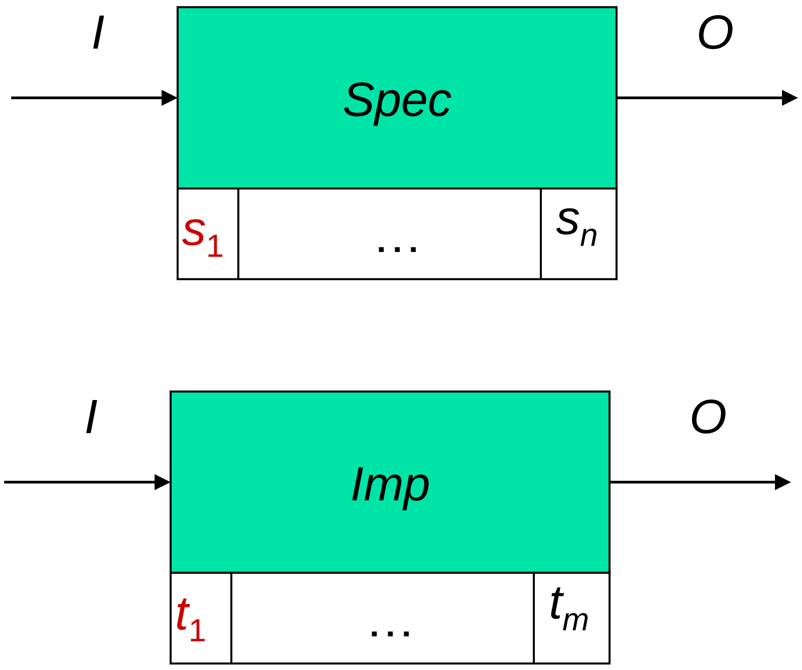
- Extract the specification *Spec*
- Insert a number of faults (get a finite set of *mutants*)
- Distinguish each *mutant from Spec* (if possible)
- Problems:
- To extract *Spec*
- Which faults to insert
- How to distinguish *Spec* and a *mutant*
- all the mutants have to be explicitly enumerated

For distinguishing two initialized FSMs a separating (distinguishing) sequence can be used

Separating sequences

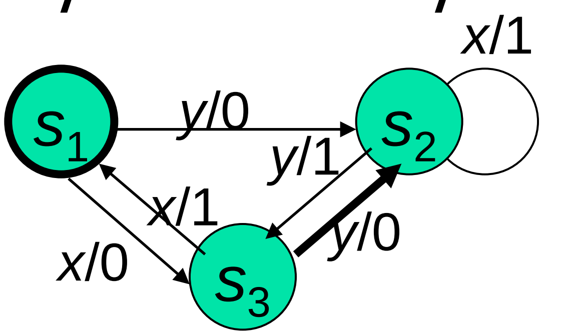
Spec and *Imp* are *separated* (*distinguished*) by input sequence α if *Spec* and *Imp* have different output responses to α

Spec and *Imp*

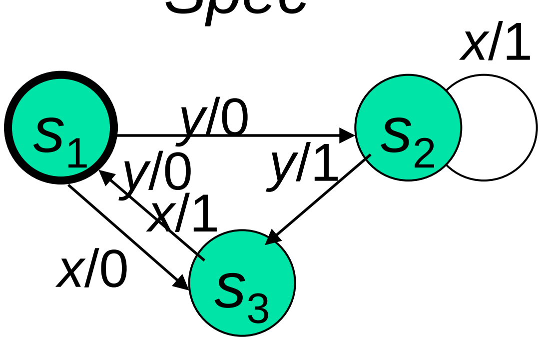


Separating sequences (2)

Spec and *Imp*



Spec



Imp

Transition from state s_3 under *y* is wrongly implemented

A separating sequence is *y y y y*:
y/0 *y*/1 *y*/0 *y*/?

For deriving a separating sequence the product of *Spec* and *Imp* can be used

If *Spec* has *n* states and *Imp* has *m* states then the product has at most *mn* states

* Can be also used for partial and nondeterministic FSMs

When using the explicit enumeration of mutants

Advantages

- Easy to implement
- Total length of the obtained test suite is close to optimal

Disadvantage

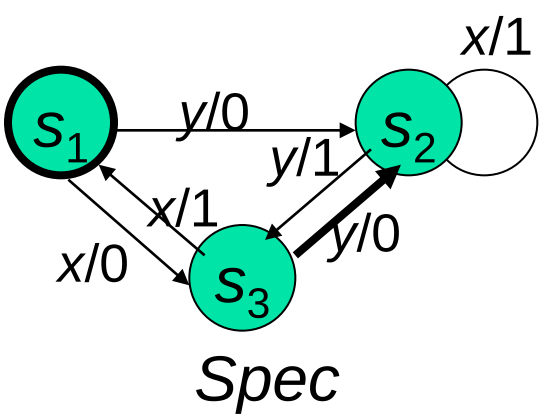
- Cannot explicitly enumerate all the FSMs with at most *n* states even for small *n*

! There exist ***fault models and test derivation methods*** which allow to guarantee the fault coverage without explicit mutant enumeration

Test suite derivation using only the specification FSM

Transition tour is a set of sequences which traverse each transition of *Spec*

Proposition. If only output faults can occur in *Imp* or states of *Imp* can be observed then a transition tour is a complete test suite



y x y x x y

Experimental results / fault coverage evaluation

ITC'99 benchmarks (second release)

Circuit \ Fault Domain	SSF coverage	SBF coverage	HDF coverage	Total coverage
b01	100%	97.62%	70.73%	92.40%
b02	95.83%	86.96%	82.61%	90.43%
b06	98.94%	97.78%	75%	92.90%

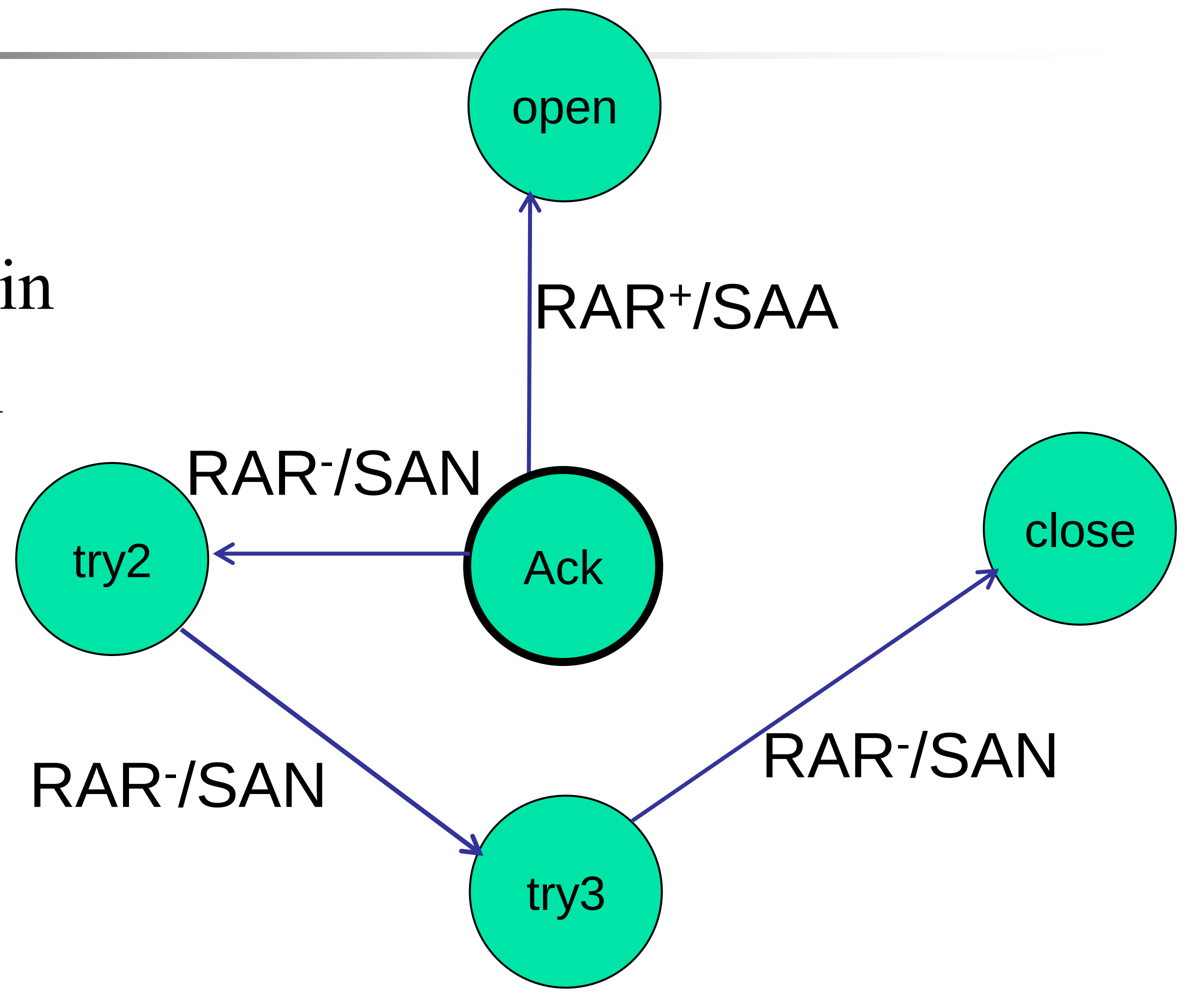
One of FSMs for PAP

RAR$^+$ - «good» login

RAR$^-$ - «bad» login

SAA - Ack

SAN – Nack



Deriving tests

Under assumption…

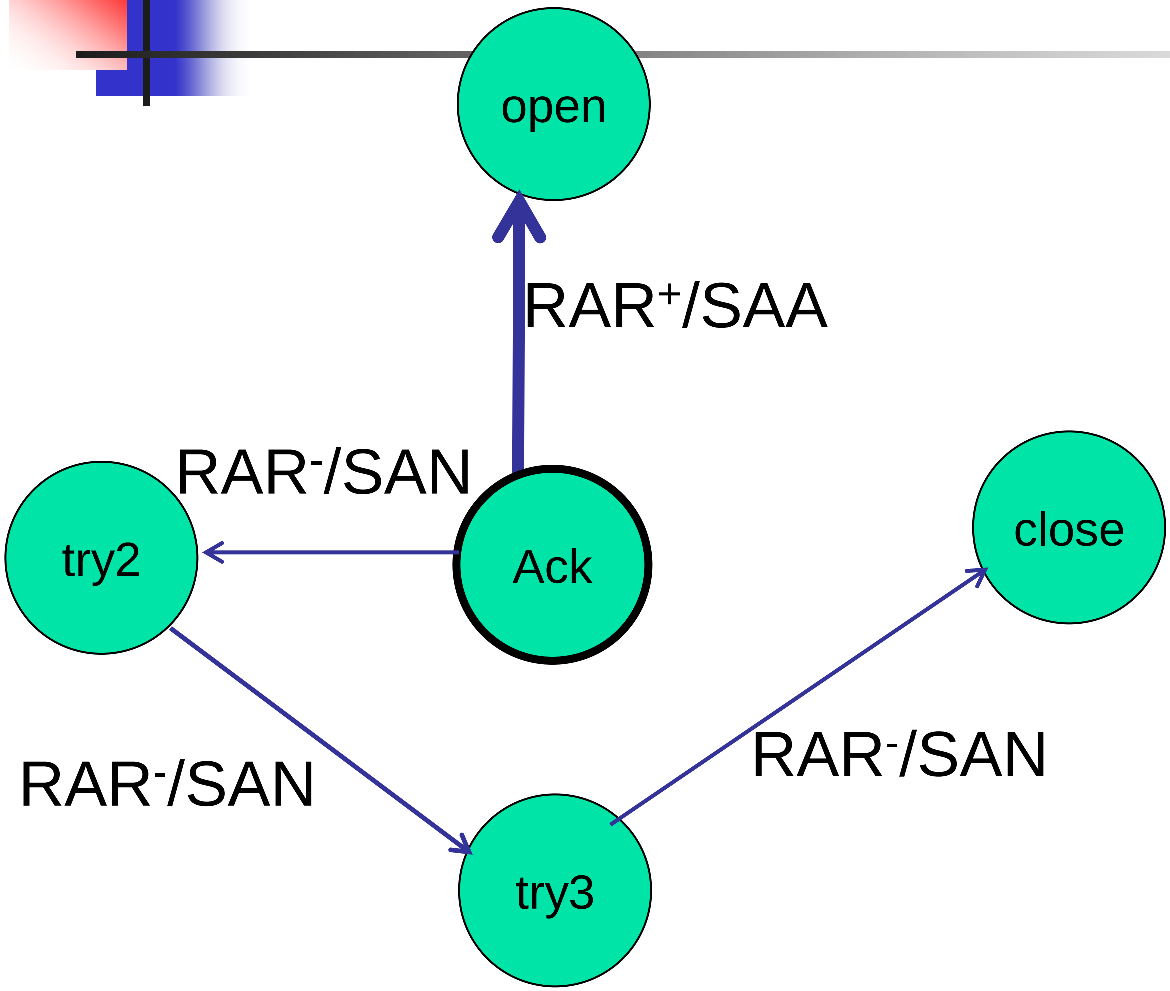
- We can 'build' an FSM that simulates a faulty implementation
- There can be faults of two types:

- 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 PAP



Test suite:

RAR$^+$

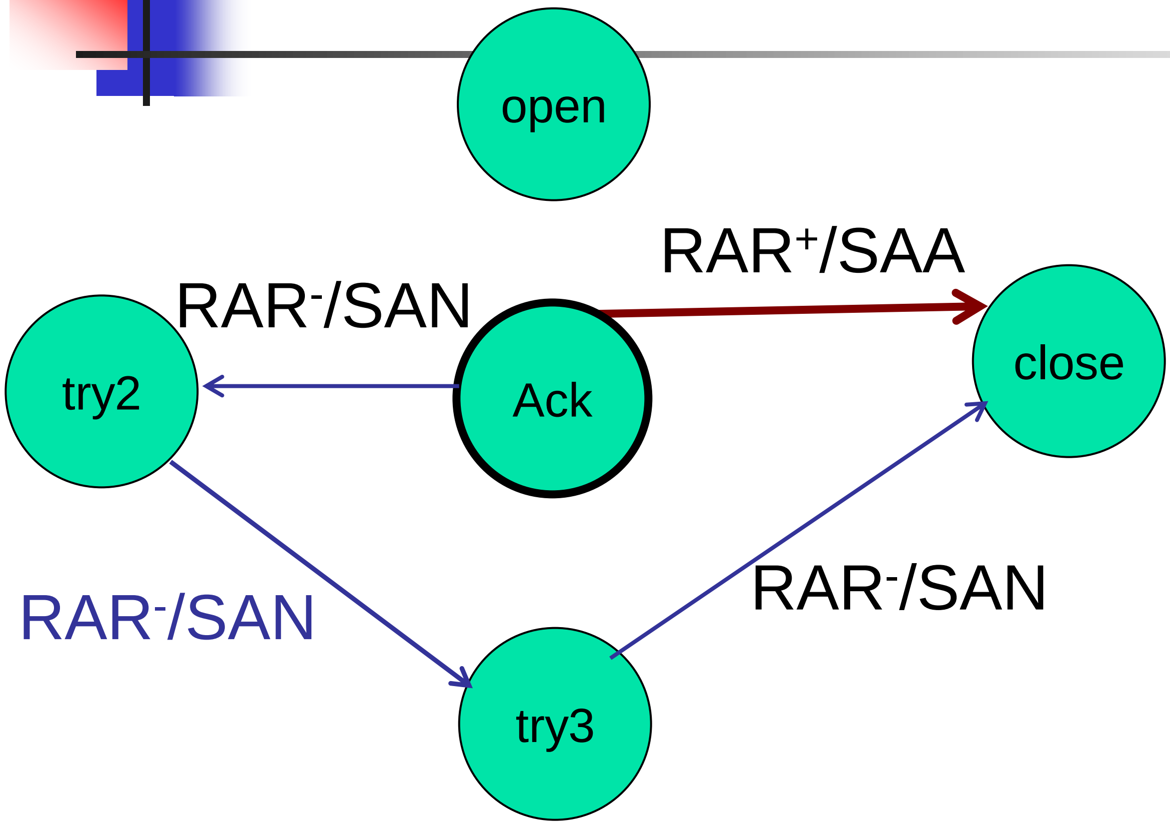
RAR$^-$RAR$^-$RAR$^-$

Expected output responses:

SAA

SAN SAN 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!!!

How to test destination state without observing

When states can be directly observed (white box testing) a transition tour is sufficient

Just to execute EACH transition at EACH state

Question: what to do when a final state of a transition cannot be observed?

Solution: to implicitly distinguish *Imp* states based on I/O sequences

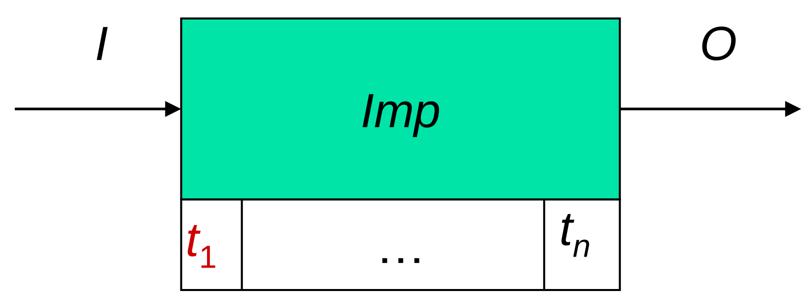
Separating (distinguishing) sequences

As we do not directly observe states of *Imp*, we use separating sequences to draw some conclusions

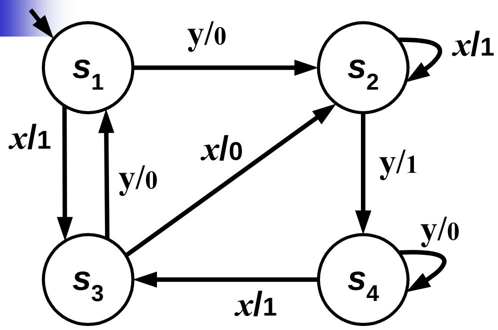
States s_j and s_k of *Spec* are *separated* by input sequence α if *Spec* has different output responses at s_j and s_k to α

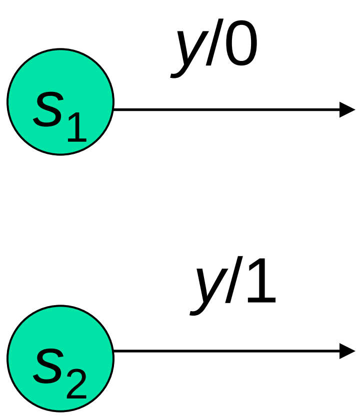
If *Imp* produces different outputs to α then *Imp* is at two different states t_j and t_k

… $tj\ \alpha/\beta_1$ … … $tk\ \alpha/\beta_2$ …



How to detect transition faults if states cannot be observed





y separates s_1 and s_2

	x	y	yy
s_1	1	0	01
s_2	1	1	10
s_3	0	0	00
s_4	1	0	00

Isomorphic FSMs

Two FSMs *Spec* and *Imp* are isomorphic iff

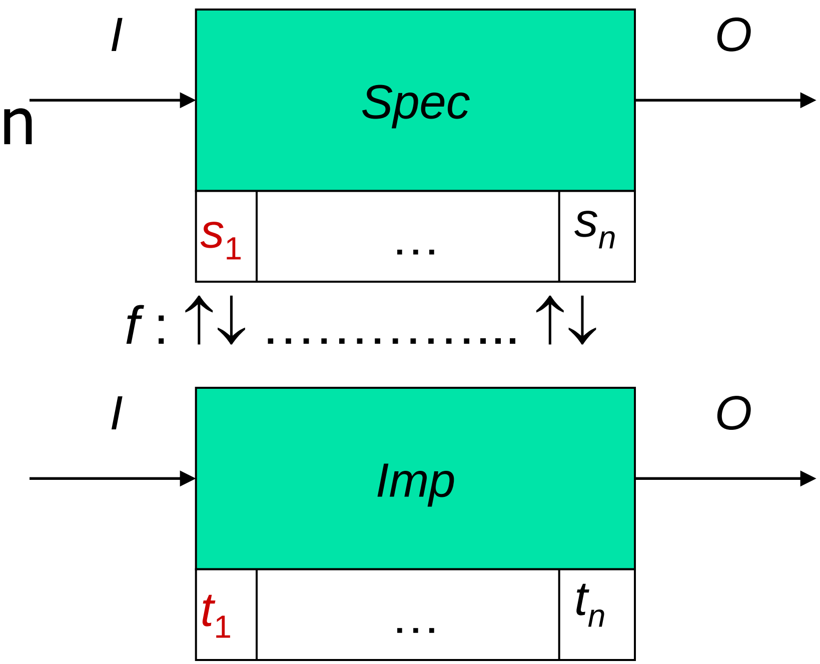
1. There exists one-to-one $f: T \rightarrow S$ between states, $f(t_1) = s_1$
2. The same f is kept between transitions

$\lambda_{Imp}(t, i) = \lambda_{Spec}(f(t), i)$

and

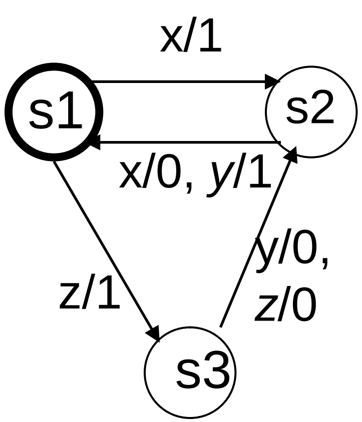
$f(\delta_{Imp}(t, i)) = \delta_{Spec}(f(t), i)$

Spec and *Imp* have the same number of states



Reduced FSM

An FSM is *reduced* if each two states can be distinguished with some input sequence (separating sequence)



Proposition If FSM *Spec* is reduced and *Imp* has the same number of states, then FSM *Imp* is equivalent to *Spec* iff *Imp* is isomorphic to *Spec*

For each deterministic complete FSM there exists a reduced FSM with the same Input/Output behavior

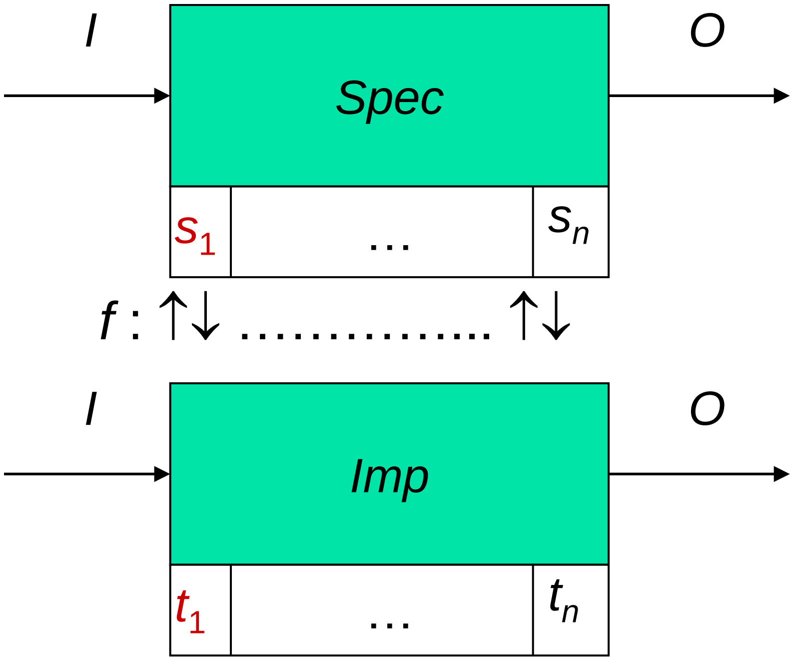
All *Specs* are reduced FSMs

How to check if an implementation is isomorphic to *Spec*

1. To assure that a given implementation *Imp* has *n* states

2. To assure that for each transition of *Spec* there exists a corresponding transition in the FSM *Imp*

Checking states and transitions of *Imp*



W-method

1. For each two states s_j and s_k of the specification FSM *Spec* derive a separating 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: $TS_1 = V.W$

Proposition If an implementation FSM *Imp* passes TS_1 then

- *Imp* has exactly n states
- V is a state cover set of the implelmentation
- there exists one-to-one mapping $f: T \rightarrow S$

  s. t. $f(t) = s \Leftrightarrow t \cong_W s$

W-method (3)

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

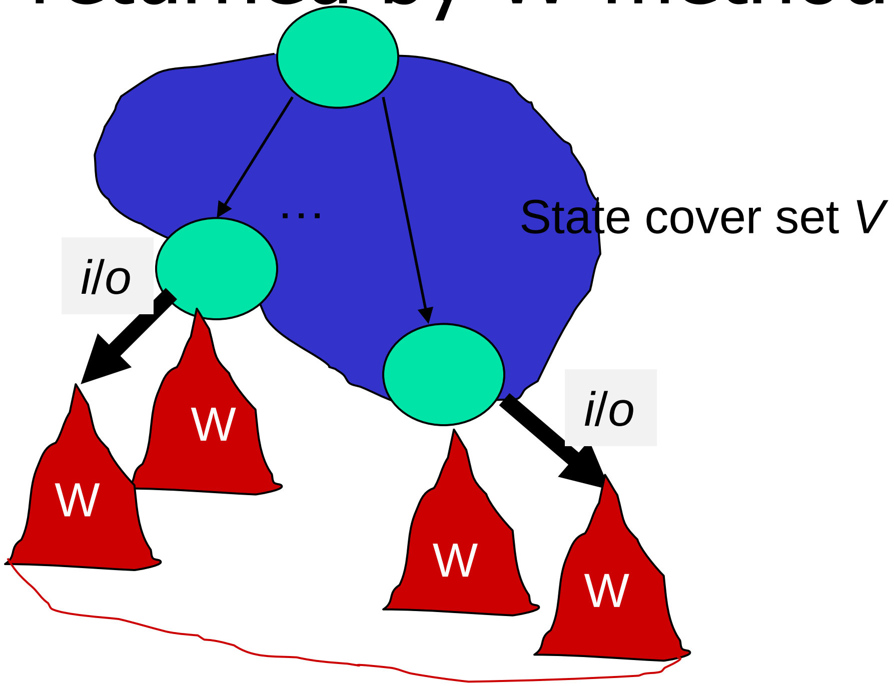
Proposition If an implementation FSM *Imp* that passed TS_1, passes also TS_2 then one-to-one mapping *f* satisfies the property:

$\lambda_{Imp}(t, i) = \lambda_{Spec}(f(t), i)$ & $f(\delta_{Imp}(t, i)) = \delta_{Spec}(f(t), i)$

i.e. FSM *Imp* is isomorphic, and thus, is equivalent to *Spec*

W-method (4)

Test suite returned by W-method



All the sequences that are prefixes of other sequences can be deleted from a complete test suite without loss of its completeness

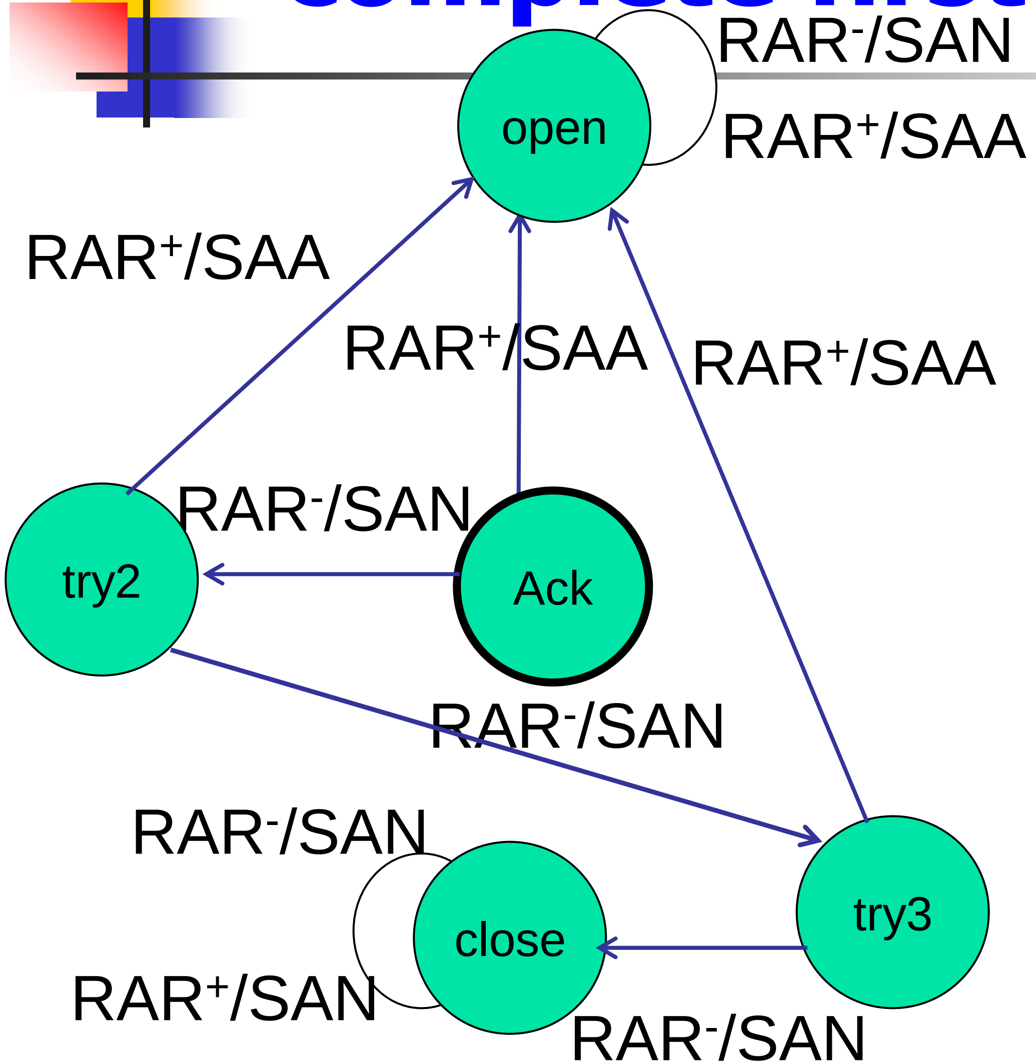
W-method (5)

When a state cover V is prefix closed, while the distinguishability set W is suffix closed the set

$$V.I.W$$

is a *complete test suite* for the case when faults do not increase number of states of the specification

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) : $RAR^- \ RAR^- \ RAR^- \ RAR^+$

(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^{+}

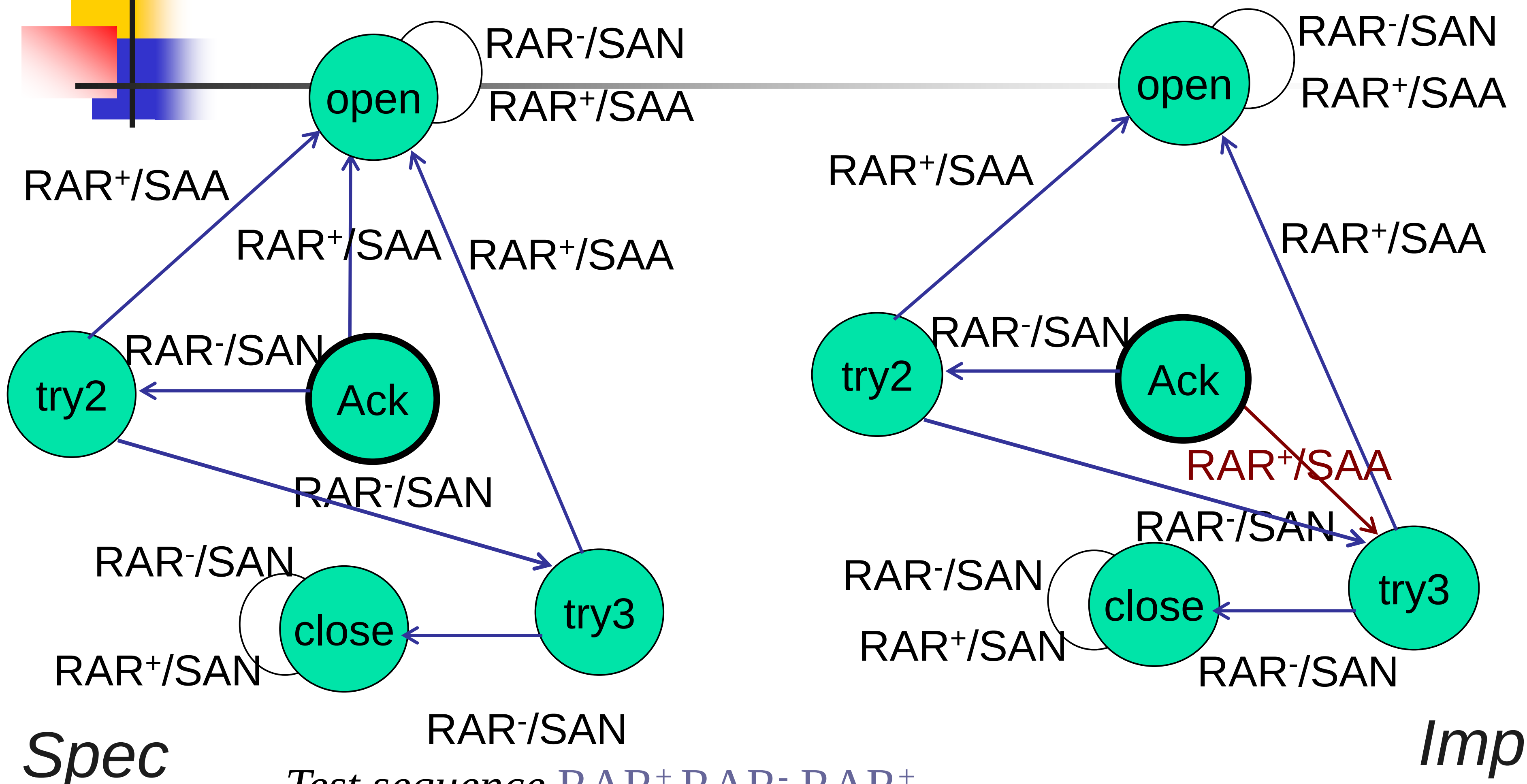
state try2: RAR^{+} RAR^{-} RAR^{-} RAR^{-} RAR^{-} RAR^{+}

RAR^{+} RAR^{-} RAR^{-} RAR^{+}

RAR^{+} RAR^{-} RAR^{+}

RAR^{+} RAR^{+} …

Detecting a transfer fault



Test sequence RAR^+ RAR^- RAR^+

Spec response : SAA SAN SAA

Imp response : SAA SAN SAN

Experimental results

State num.	Input num.	Output num.	Trans. num.	Average length
30	6	6	180	2545
30	10	10	300	3393
50	6	6	300	5203
50	10	10	500	6773
100	10	10	1000	17204

W-test length evaluation

Theoretically

Length is $O(kn^2)$ where

k – number of transitions

n - number of states

Experiments show

Tests are much shorter but

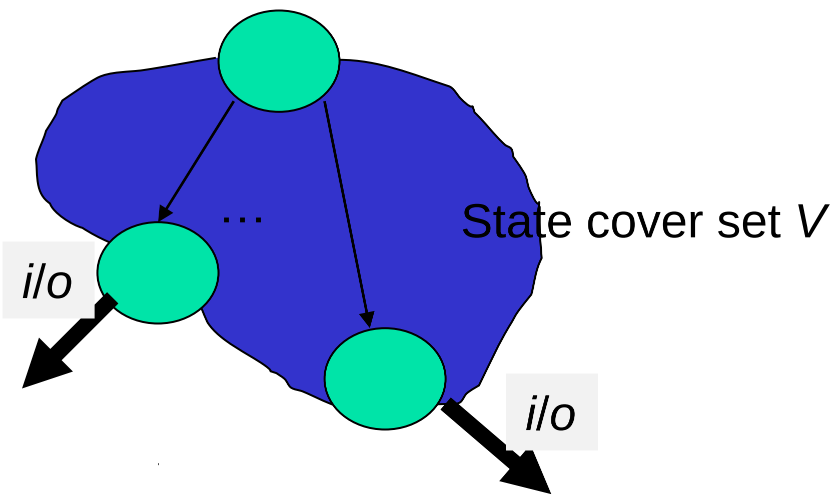
STILL LONG ENOUGH

Studying W-method

Conclusions:

1. The set *V.I* is presented in each complete test suite
2. The length of a complete test suite significantly depends how states are identified, i.e., on the choice of state identifiers

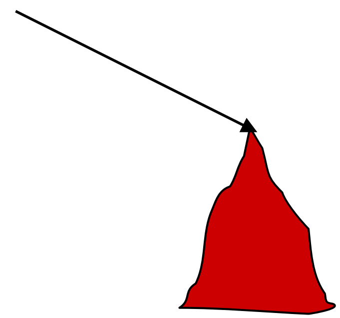
Core set



Modifications of W-method

1. DS-method (not always exists)
2. UIO-method (a test suite is not complete)
3. HSI-method
4. H-method
5. HSY-method
6. ...

Depending how a set of separating sequences is defined



! SPY method allows to check transitions after different sequences of a state cover set

DS-method (experiments)

State num.	Input num.	Output num.	Trans. num.	Average length
30	6	6	180	934 (2545)
30	10	10	300	1493 (3393)
50	6	6	300	1777 (5203)
50	10	10	500	2710 (6773)
100	10	10	1000	6602 (17204)

When DS exists (experimental results)

State num.	Input num.	Output num.	Trans. num.	% of exist.
50	4	4	200	0
80	6	6	480	0
80	8	8	640	1%
80	10	10	800	5%

HSI-method

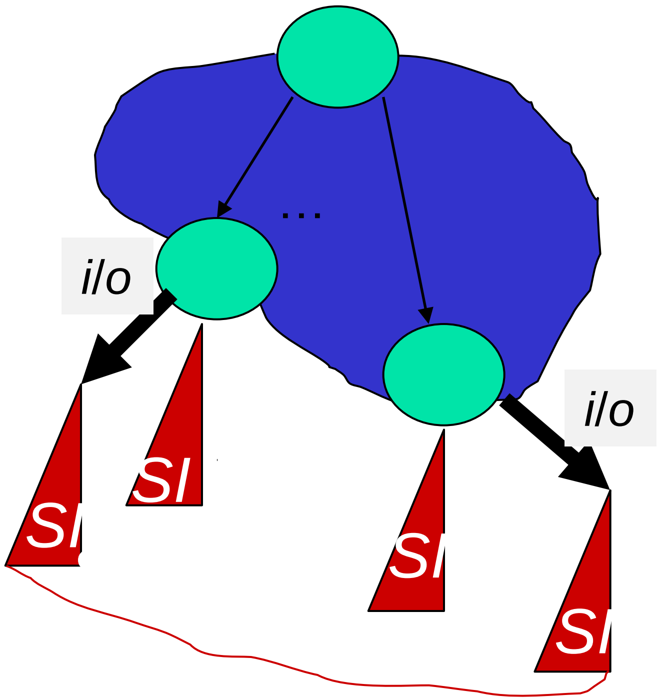
State Identifiers (SI)

Given state s_j and any other state s_k of the specification FSM *Spec*, derive a separating sequence γ_{jk}

Gather all the sequences into a set *SI* that is called a *state identifier* of state s_j

! But *SI* have to be harmonized

HSI-method



Experimental results

State num.	Input num.	Output num.	Trans. num.	HSI	W
30	6	6	180	1649	2545
30	10	10	300	2243	3393
50	6	6	300	3261	5203
50	10	10	500	4375	6773
100	10	10	1000	10503	17204

H-method

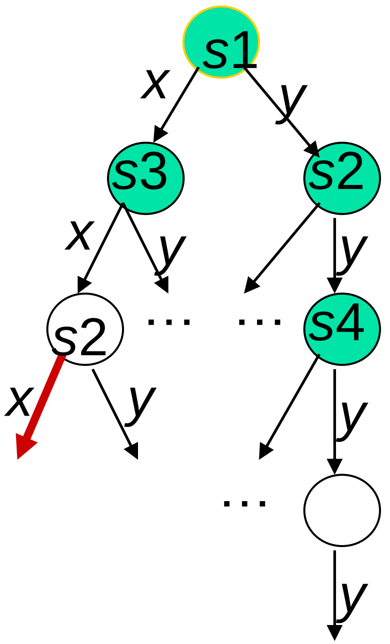
Solution:

- to use different *SI* for the same destination state
- If some *SI* are not harmonized then add necessary separating sequences

Conclusion: State identifiers can be derived on the fly

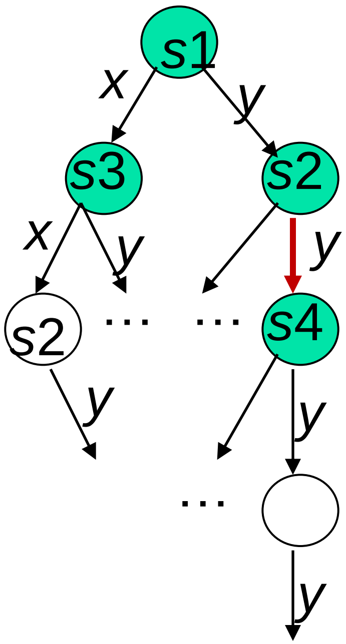
H-method (illustration)

HSI-method



$L = 41$

H-method



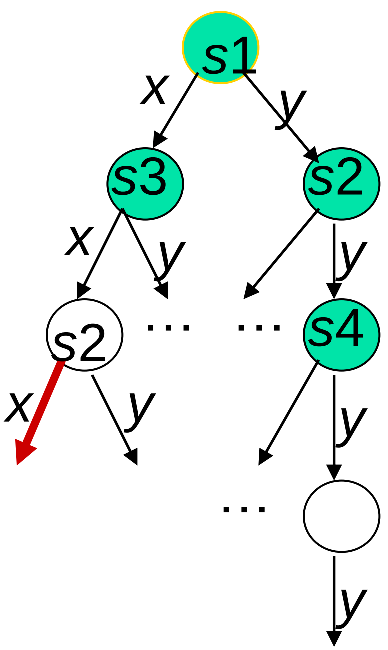
$L = 25$

Experimental results

State num.	Input num.	Output num.	Trans. num.	DS	H
30	6	6	180	934	1105
30	10	10	300	1493	1568
50	6	6	300	1777	2142
50	10	10	500	2710	2852
100	10	10	1000	6602	6880

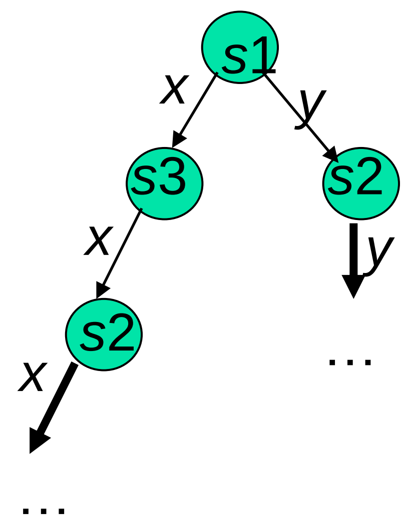
SPY-method (illustration)

HSI-method



L = 41

SPY-method



L = 25

To use different input sequences to reach the same state when checking
Different transition at this state

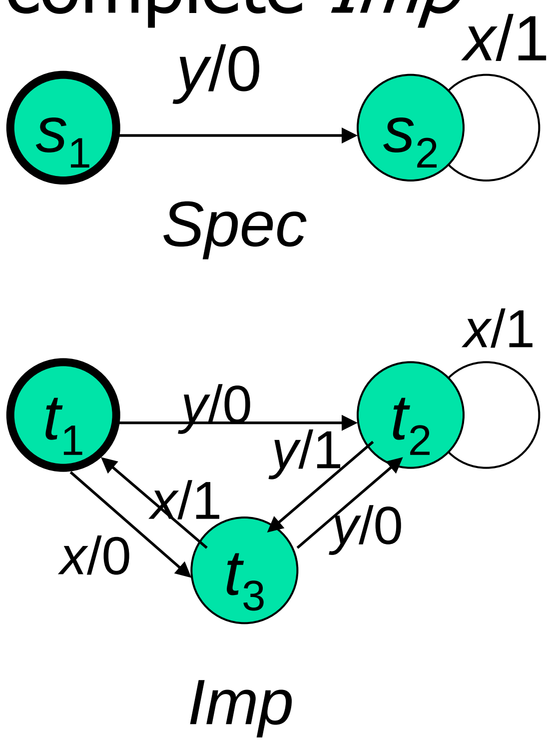
FSM-based conformance testing for partial FSMs

1. *Spec* can be partially specified;
Imp is a complete FSM

2. Imp conforms to *Spec* iff *Imp* is quasi-equivalent to *Spec*

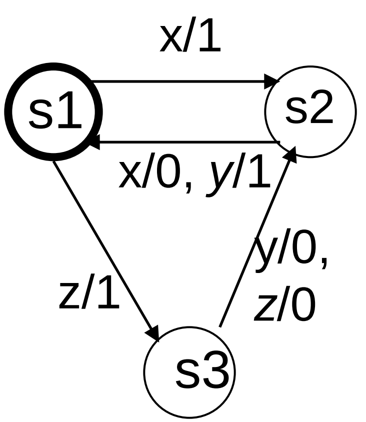
Quasi-equivalence relation

- A complete FSM *Imp* is *quasi-equivalent* to *Spec* if their output responses coincide for each input sequence that is defined in the *Spec*
- A partial *Spec* and a complete *Imp*



W- (Wp-, UIOv-) methods cannot be used

W- (Wp-, UIOv-) methods cannot be generally used as not each partial FSM has the distinguishability set W

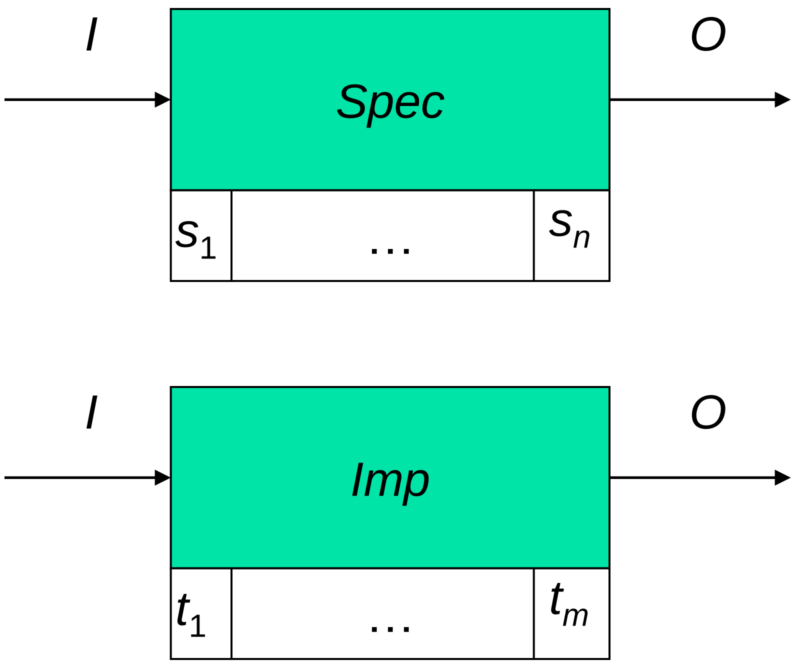


Distinguishability set not necessarily exists

Quasi-equivalence relation (2)

It can happen that a test suite is complete when a FD contains each FSM with limited number of states

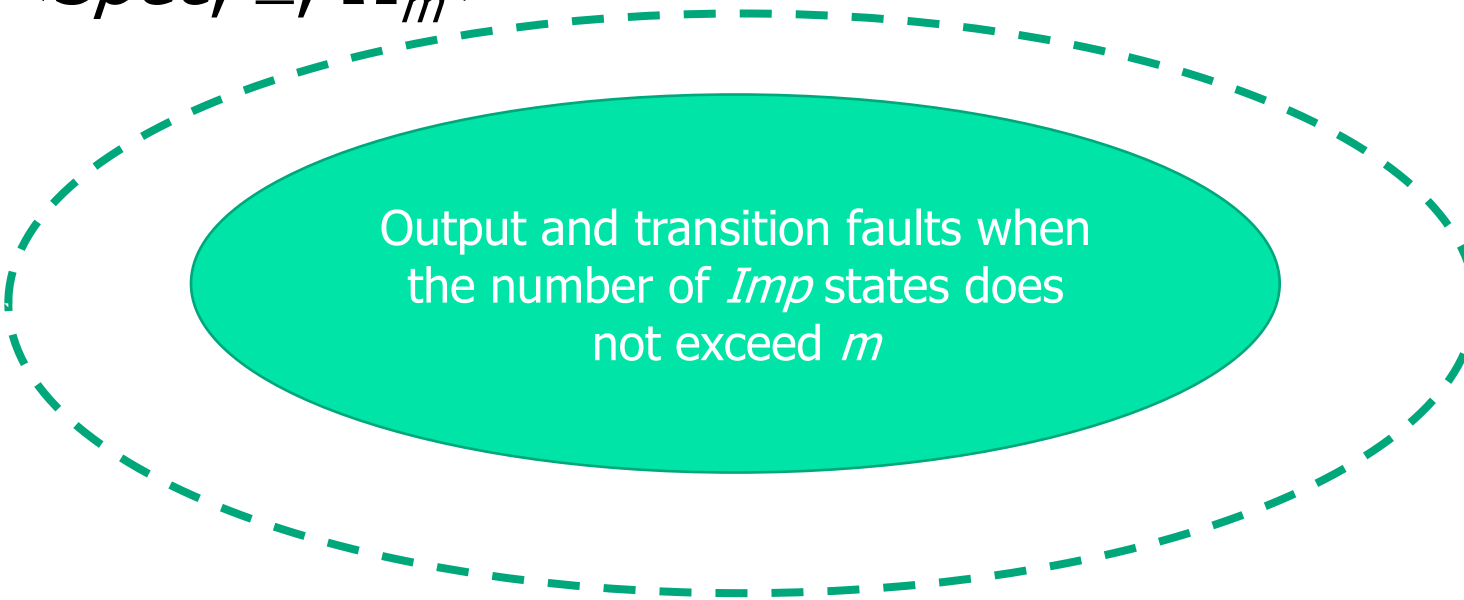
The set of traces of *Spec* has to be a subset of that of *Imp*



W-based test suite exhaustiveness

W-based tests are complete w.r.t. FM

$< \mathit{Spec}, \cong, \Omega_m >$



The same test suite detects much more faults but there is no guarantee

Conclusions about W-method

1. DS-method returns shortest test suites

But: less than 10% of specifications possess a DS

2. H- and SPY- methods return tests that are comparable with those returned by DS-method

and can be applied to any reduced (partial or complete) specification

3. All the methods can deal with the case when *Imp* has more states than *Spec*

Test suites returned by all above methods are too long ⇒

User defined faults can be considered

How to reduce the length of a test suite

Solution: To check only some transitions of the specification

↓

Incremental testing or

testing user-driven faults

Experimental results are very promising especially for the case when faults can increase the number of states of the specification

Experimental results

s	i	HSI length	0-5% suspi	5-10% suspi	10-15% suspi	15-20% suspi
20	10	2992	93	337	490	785
20	20	5818	148	477	999	1513
30	10	5333	135	518	957	1450
35	10	6588	148	539	1013	1537
40	5	3737	89	345	636	887

Protocol implementations were tested

- SCP
- Pop-3
- IRC
- TCP (also in context)
- FTP
- TFTP
- …

Not considered

- Nondeterministic FSMs and corresponding Fault Models
- EFSMs and corresponding FSM-like slices
- Timed FSMs and corresponding FSM slices
- Test derivation for FSM composition, testing in context
- …

Publications (deterministic FSMs)

1. Chow, T.S. Test design modeled by finite-state machines. IEEE Transactions on Software Engineering, 4(3), pp. 178-187 (1978)
2. Lee D. and Yannakakis, M. Principles and methods of testing finite state machines-a survey. Proceedings of the IEEE, 84(8), pp. 1090—1123 (1996)
3. Lai, R. A survey of communication protocol testing. The Journal of Systems and Software. 62. pp. 21-46 (2002)
4. M.Dorofeeva, K.El-Fakih, S.Maag, A.Cavalli, N.Yevtushenko. FSM-based conformance testing methods: A survey annotated with experimental evaluation. Information and Software Technology, 52, (12), pp. 1286-1297 (2010)
5. A. Simao, A. Petrenko, N. Yevtushenko. On reducing test length for FSMs with extra states. Softw. Test., Verif., Reliab., 22 (6), pp. 434-454 (2012)

Some publications (nondeterministic FSMs)

1. Hierons, R. M.: Adaptive testing of a deterministic implementation against a nondeterministic finite state machine. The Computer Journal, 41(5), pp. 349–355 (1998)
2. Petrenko, A., Yevtushenko, N.: Conformance Tests as Checking Experiments for Partial Nondeterministic FSM. In Proceedings of the 5th International Workshop on Formal Approaches to Testing of Software, LNCS vol. 3997, pp. 118—133 (2005)
3. Shabaldina, N., El-Fakih, K., Yevtushenko, N:. Testing Nondeterministic Finite State Machines with respect to the Separability Relation. Lecture Notes in Computer Science vol. 4581, pp. 305-318 (2007)
4. Adilson L. Bonifacio, Arnaldo V. Moura, Adenilso S. Simao. Experimental comparison of approaches for checking completeness of test suites from finite state machines. Information and Software Technology, 92, pp. 95-104 (2017)

Some publications (Timed FSMs)

1. Alur, R, and Dill. D. L.: A Theory of Timed automata. Theoretical Computer Science, 126(2),183----235 (1994)
2. M. G. Merayo, M. Nunez, I. Rodriguez. Extending EFSMs to Specify and Test Timed Systems with Action Durations and Time-outs. IEEE Transactions on Computers, 57(6), 2008, pp. 835—844.
3. Springintveld, J., Vaandrager, F., D'Argenio, P.: Testing Timed Automata. Theoretical Computer Science, 254(1-2), 225–257 (2001)
4. Khaled El-Fakih, Nina Yevtushenko, and Adenislo Simao: A practical approach for testing timed deterministic finite state machines with single clock. Science of Computer Programming, 80 (1), pp. 343-355 (2014)

Thanks for your attention