Testing Networked Systems: Theory and Practice

Mohammad Mousavi

A discipline of testing is:

extremely important, and
can be rigorous, too.

Part 1:

(Model-Based) Testing Fundamentals: Theory and Practice

Based on joint work with:

- **Hamid Reza Asaadi** (U Tehran, now at Stony Brook U),
- Rachid Kherrazi (Philips Healthcare, now at Promedico),
- **Ramtin Khosravi** (U Tehran),
- Mehmet Kovacioglu (Philips Healthcare, now at Credit Suisse),
- **Neda Noroozi** (TU Eindhoven, now at Nspyre),
- Mahsa Varshosaz (Halmstad U),
- Vivek Vishal (Philips Healthcare, now at ASML), and
- Tim Willemse (TU Eindhoven).

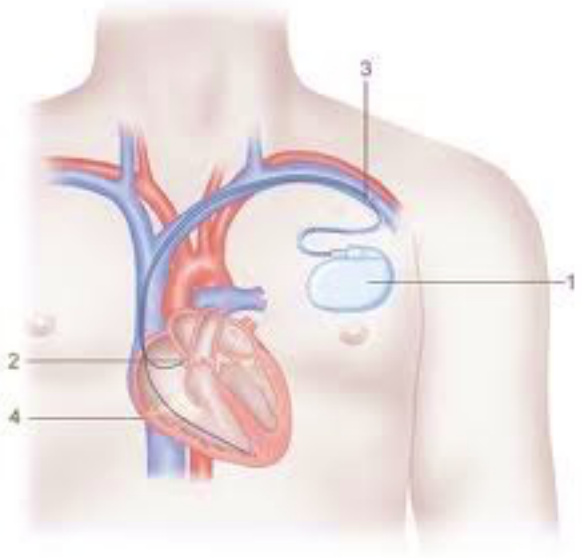
Testing:

Why? What? How?

Why?
Software at Your Heart

Software glitches in pacemakers

"Company said it has not received any reports of deaths or clinical complications resulting from the glitch, which appears in about 53 out of every 199,100 cases."



[Killed by Code, 2010]

Why?
Software at Critical Infrastructure

… a glitch caused more than 3,200 US prisoners to be released early. The software calculates a prisoner's sentence depending on good/bad behaviour and was introduced in 2002.

Photo by Thomas Hawk @ Flicker

[BBC News 2015]

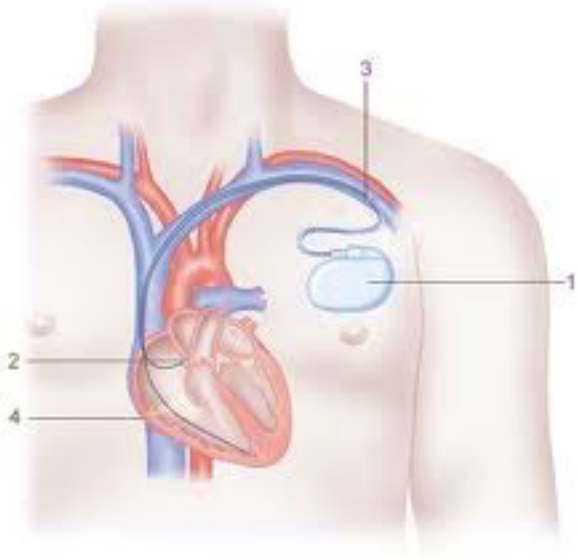
Why?
Software at Your Car

Over the past two years Nissan has been recalling airbags adding up to over 1 million cars … due to a glitch in the airbag's sensory detectors. There has been a reported two accidents due to this software failure.

Photo from Wikipedia

Why?
Software at Your Heart

At least 212 deaths from device failure in five different brands of implantable cardioverter-defibrillator (ICD) according to a study reported to the FDA



[Killed by Code, 2010]

Why?
Bugs (Faults): Facts of Life

"Coders introduce bugs at the rate of 4.2 defects per hour of programming. If you crack the whip and force people to move more quickly, things get even worse."

[Watts Humphreys]

Why?
Bugs (Faults): Facts of Life

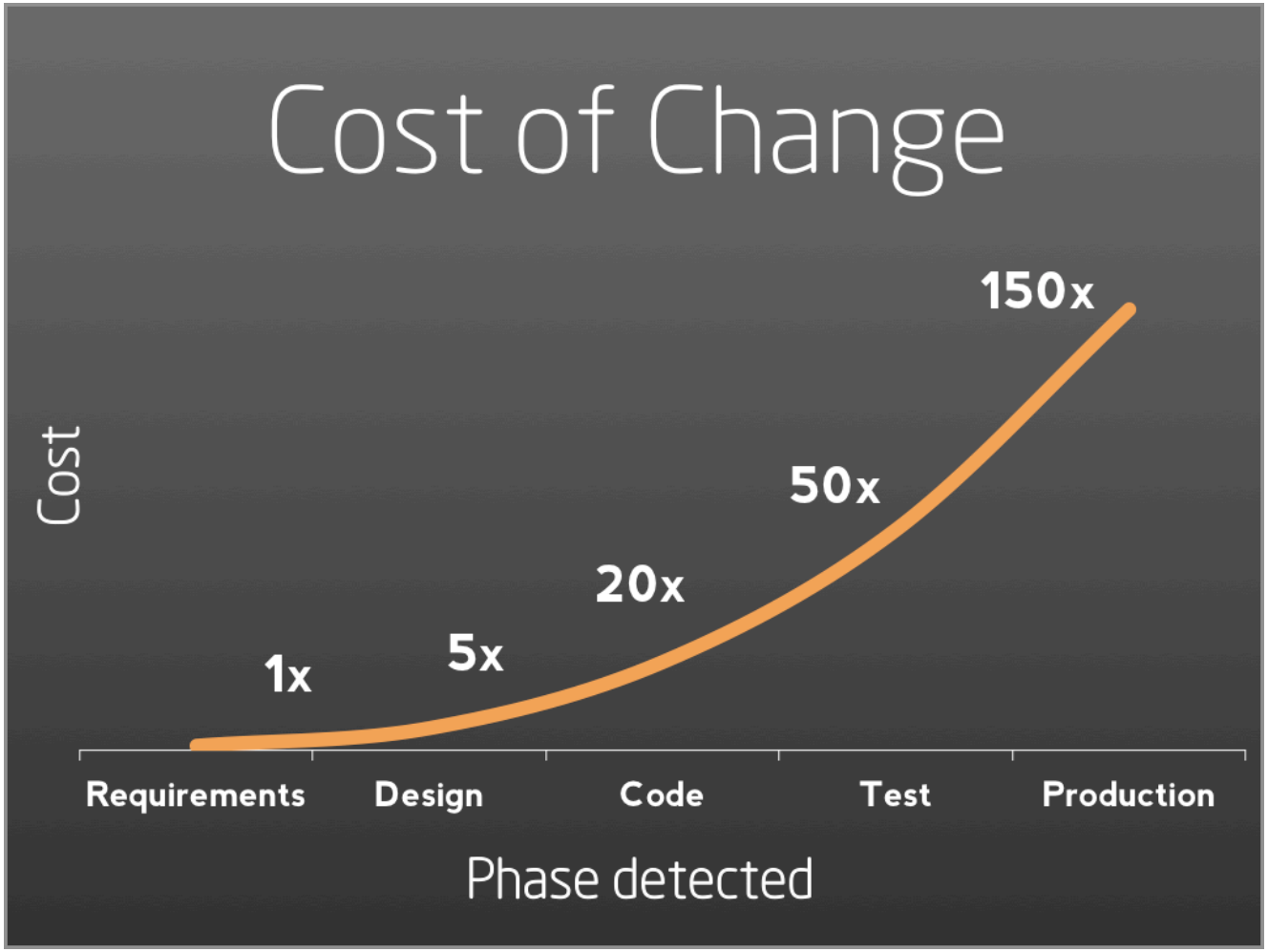
"Cost of software faults in 2016: 1100 Billion USD,

Number of people affected by software faults: 4.4 Billion people."

[Tricentis, Software Fail Watch, 2016]

Photo Copyright: Tricentis

Why?
Boehm's Curve



What?
Faults, Errors, Failures

- Fault: incorrect implementation:
  - commission: wrong implementation
  - omission: forgotten implementation (the more difficult one)
- Error: incorrect system state
- Failure (anomaly, incident) : visible error in the behavior

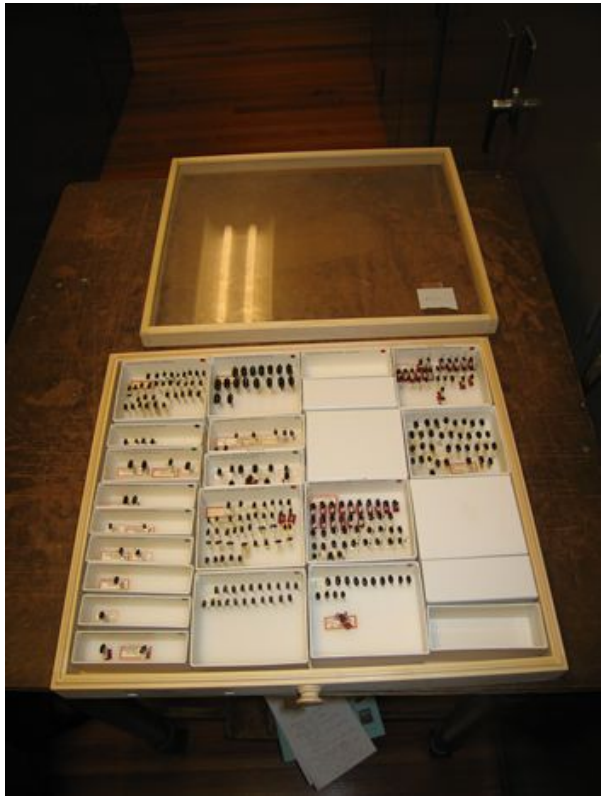


Photo from Wikipedia

Example

Spec: inputs an integer, and outputs $2*i^3$

Implementation:

```
#include <iostream>
#include  <math.h>

int main() {
    int i;
    cin >> i;
    i = 2 * i;
    i = pow(i, 3);
    cout << i;
   return 0;
}
```

Example

1. cin >> i;
2. i = 2 * i;
3. i = pow(i, 3);
4. cout << i;

- Conceptual mistake: confusing the binding power of operators
- Fault: Statements 2 and 3 are in the wrong order
- Error: State of the program after line 3 may have the wrong value for i.
- Failure:
  - Test-case: input 1, expected output 2.
  - Actual execution: input 1 ... output 8!

What?
Testing

Planned experiments to:

1. reveal bugs (turn faults into failures, test to fail), “Testing can show the presence of bugs, but not their absence.” [Dijkstra]

2. gain confidence in software quality (test to pass)

What?
Testing: Validation and Verification

- **Validation:** Have we made the right product; compliance with the intended usage (often: user-centered, manual process, on the end product)
- **Verification:** Have we made the product right; compliance between artifacts of different phases (often: artifact-driven, formalizable and mechanizable process among all phases)

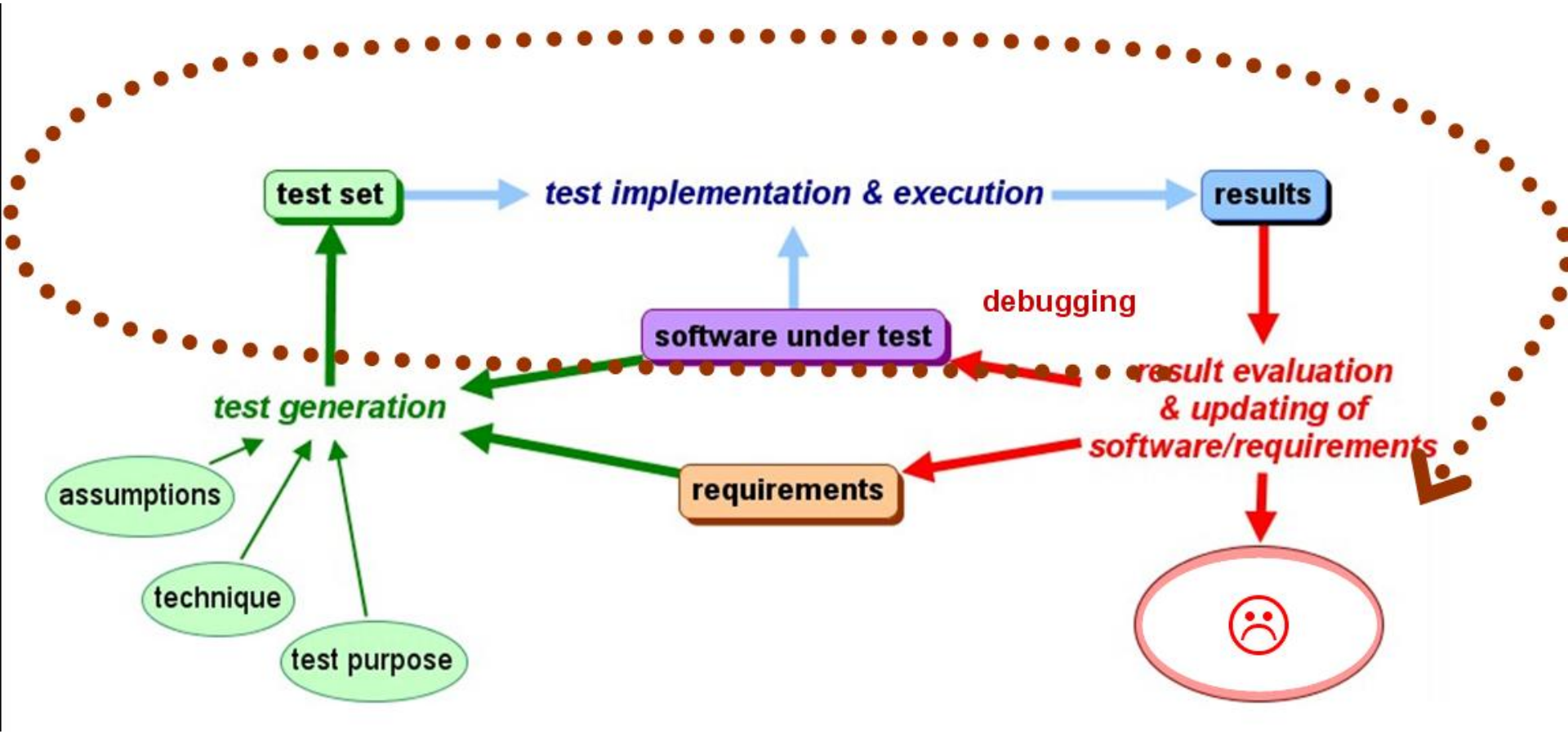
Our Focus

How?
Test-Case, Test-Suite

- **Test-Case:** a pair of
  - inputs (e.g., running environment, input values or pre-conditions, timing of events) and
  - expected outputs (e.g., concrete output values or symbolic properties input and output)

- **Test-Suite:** a set or list of test-cases

What?
Testing

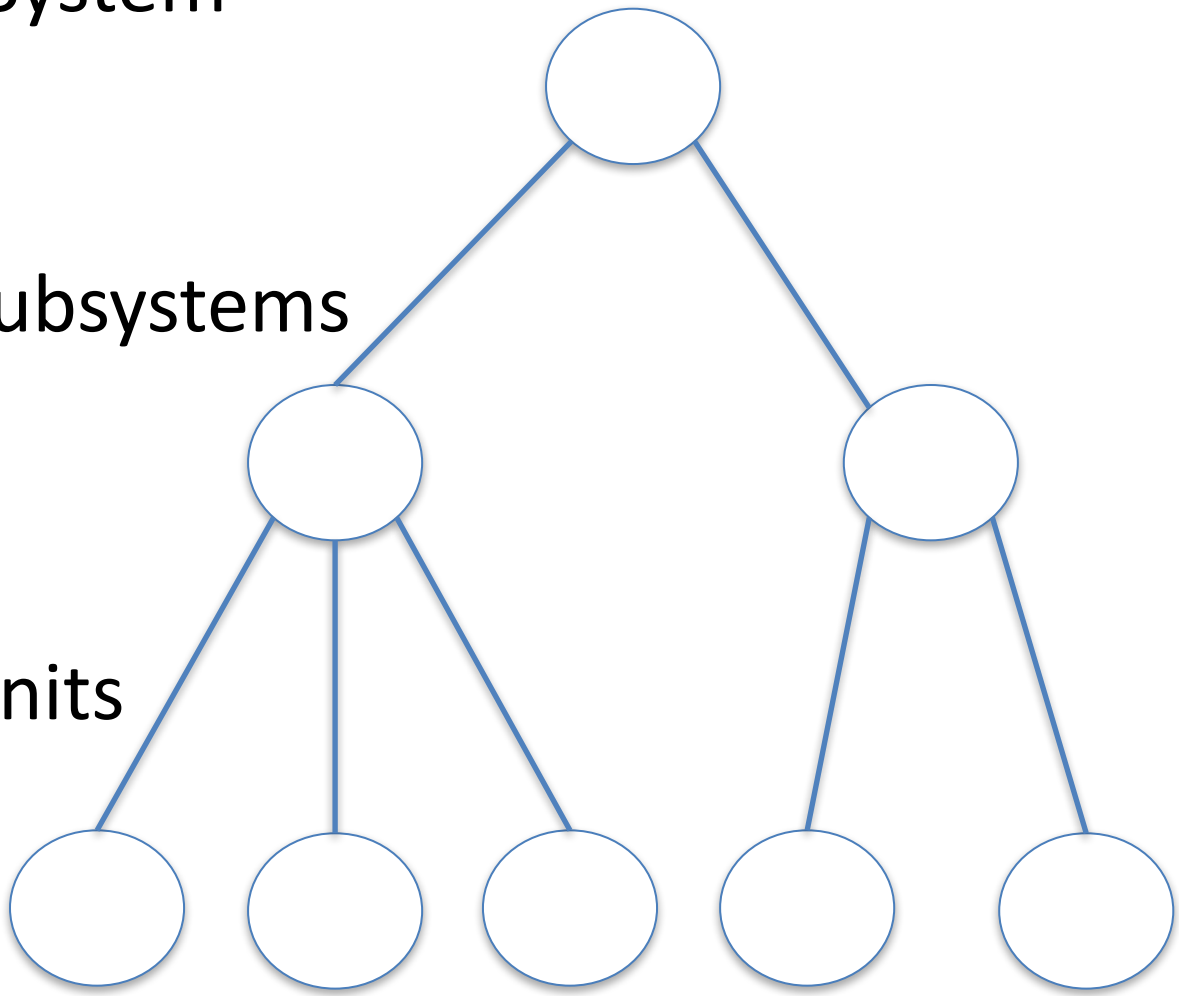


What?
Levels of Testing

System

Subsystems

Units



System Testing (Acceptance, User)
Selenium, EyeAutomate
GUI Model Testing,
Automated GUI Testing

Integration Testing
jUnit, Mockito
Dependency Injection
Mocking

Unit Testing
jUnit, QuickCheck,
EvoSuite
Test-Driven Dev.
Equivalence Partitioning,
Decision Tables,
Classification Trees

Example

Test-Driven Development of a SimpleStack Class (in Java)

Step 1:

Fix the signature of the class

```
package example.stack;
public class SimpleStack {
      public boolean isEmpty();
      public int pop();
      public void push(int item)
 }
```

Example

Test-Driven Development of a SimpleStack Class (in Java)

Write class invariants and a few properties for each method:

/*

Pre-Condition: True (can be called in all states, with all inputs)

Returns:

- true on an empty initialized SimpleStack

- false on a SimpleStack on which more successful “push”es are performed than “pop”s

State remains unchanged in both cases */

Example

Test-Driven Development of a SimpleStack Class (in Java)

Step 3:

Start with a test:

```
@Test
   public void testNewStackIsEmpty() {
      SimpleStack stack = new SimpleStack();
      Assert.assertTrue("New stack should be empty!",
                              true == stack.isEmpty());
   }
```

Example

Test-Driven Development of a SimpleStack Class (in Java)

Step 4:

Test and check if any test fails.

If so, write the minimal amount of code to pass the test(s):

```
public class SimpleStack {
    public boolean isEmpty() {
        return true;
    }
}
```

Example

Test-Driven Development of a SimpleStack Class (in Java)

Step 5:

Refactor the code if needed.

Repeat steps 3 to 5 until the requirements are covered.

Example

Test-Driven Development of a SimpleStack Class (in Java)

Step 3:

Start with a test:

```
 @Test
   public void testNewStackPush() {
      SimpleStack stack = new SimpleStack();
      int item = 1;
      stack.push(item);
      Assert.assertFalse("Stack shouldn’t be empty after a push!",
                                    stack.isEmpty());
   }
```

Example

Step 4:

Test and check if any test fails.

If so, write the minimal amount of code to pass the test(s):

```
public class SimpleStack {
    boolean empty = true;
    public void push(int item) {
        empty = false;
    }
    public boolean isEmpty() {
        return empty;
    }
}
```

Example

Test-Driven Development of a SimpleStack Class (in Java)

Step 5:

Refactor the code if needed.

Repeat steps 3 to 5 until the requirements are covered.

What We Do Not Cover: Test Management and Policy

What We Do Not Cover: Alternatives to Testing

- **Model Checking:** test the state-space (all executions) for formally specified properties

  \+ rigorous analysis, push-button technology

  \- not (yet) scalable to very large systems (state-space explosion)

What We Do Not Cover: Alternatives to Testing

- **Static Analysis:** test abstract properties without running the program, e.g., division by zero and empty/unspecified cases

  \+ automatic and scalable for generic and abstract properties;
  \+ existing powerful tools;

  \- involves approximation (true negatives and false positives);
  \- complicated (may involve theorem proving) for concrete and specific properties (proving the abstraction function to be “correct")

Theory:

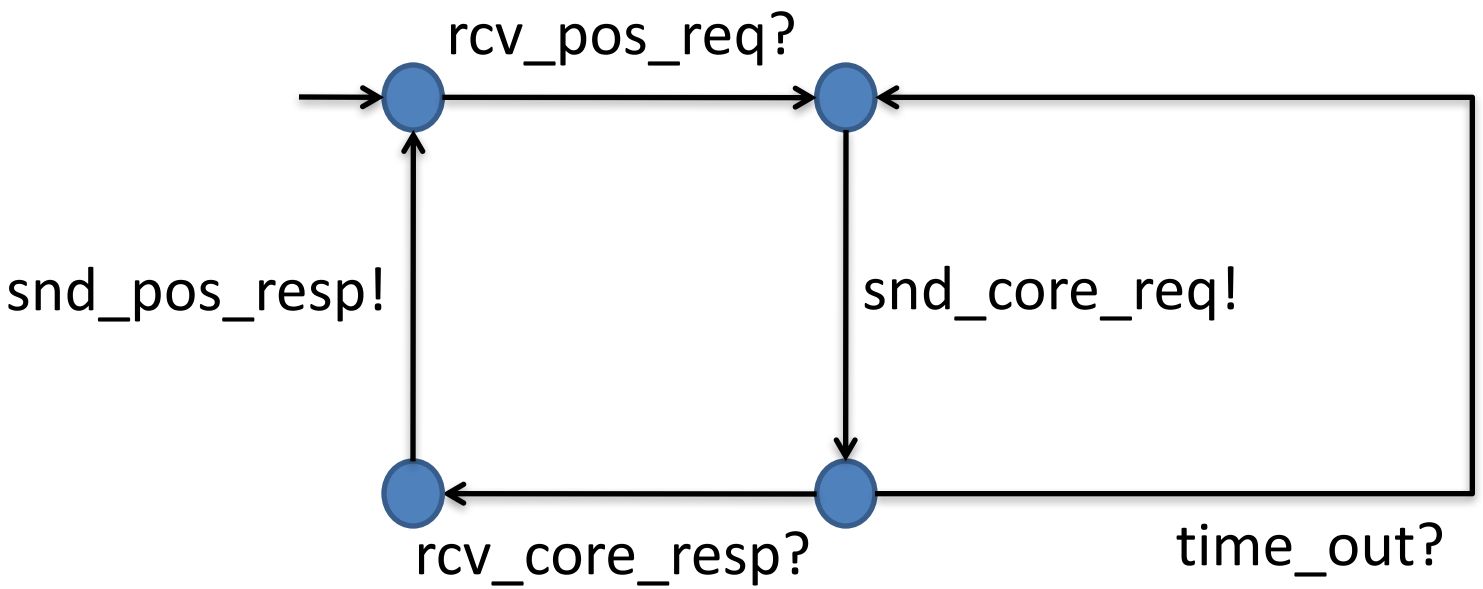
Introduction to Model-Based Testing

Model-Based Testing

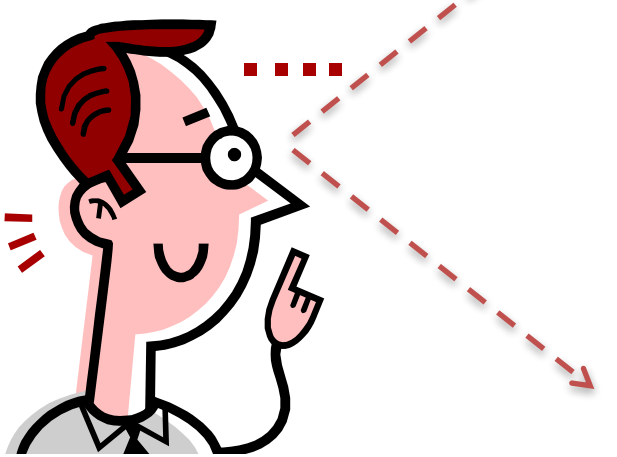
- **Abstractions** from reality
- Separating different **concerns**
- Approximating system behavior and / or its **environment**

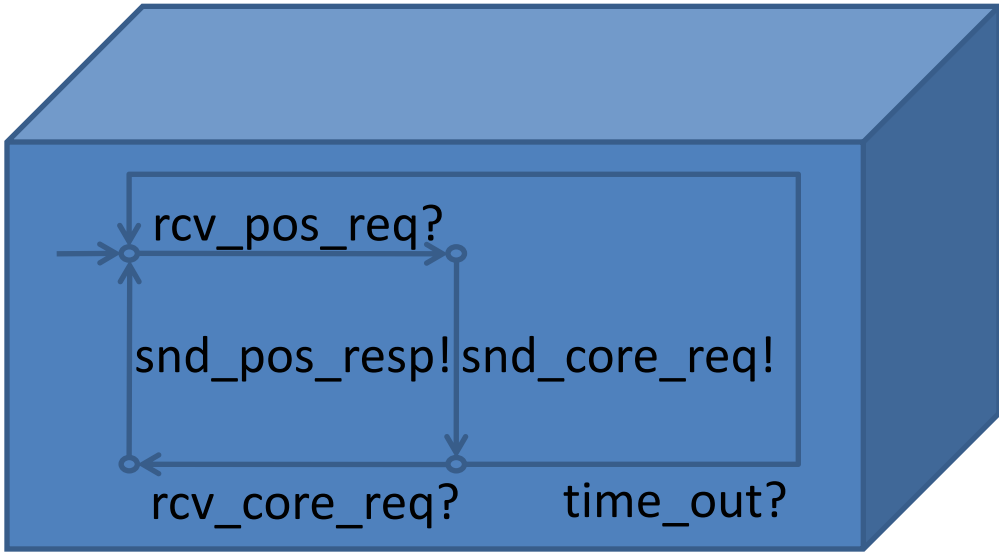
Model-Based Testing

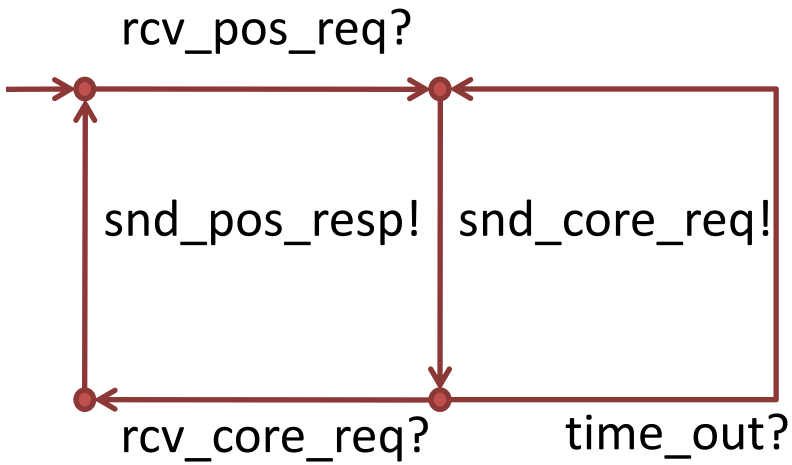
- Modeling the desired behavior (system) / possible interactions (environment)



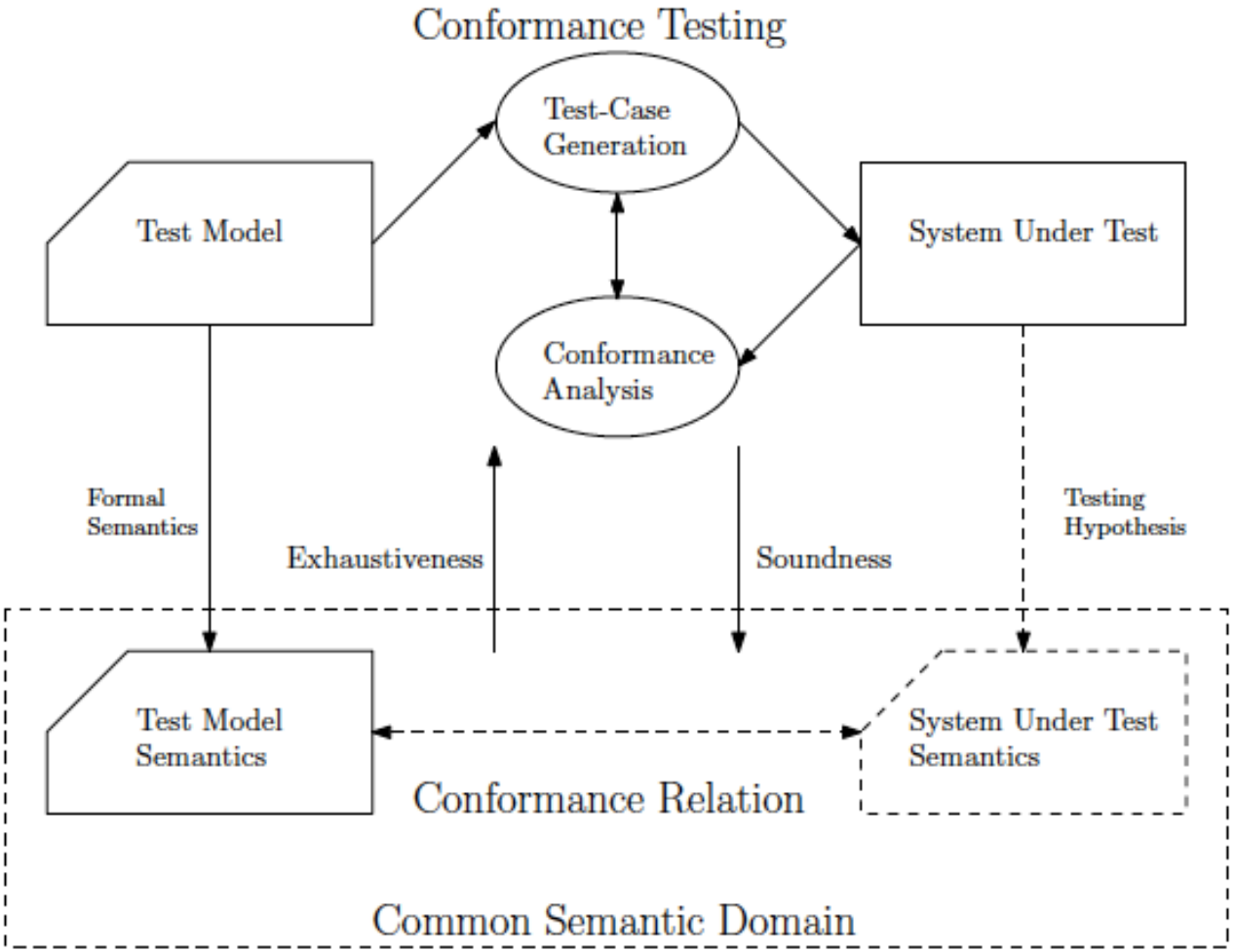
Model-Based Testing







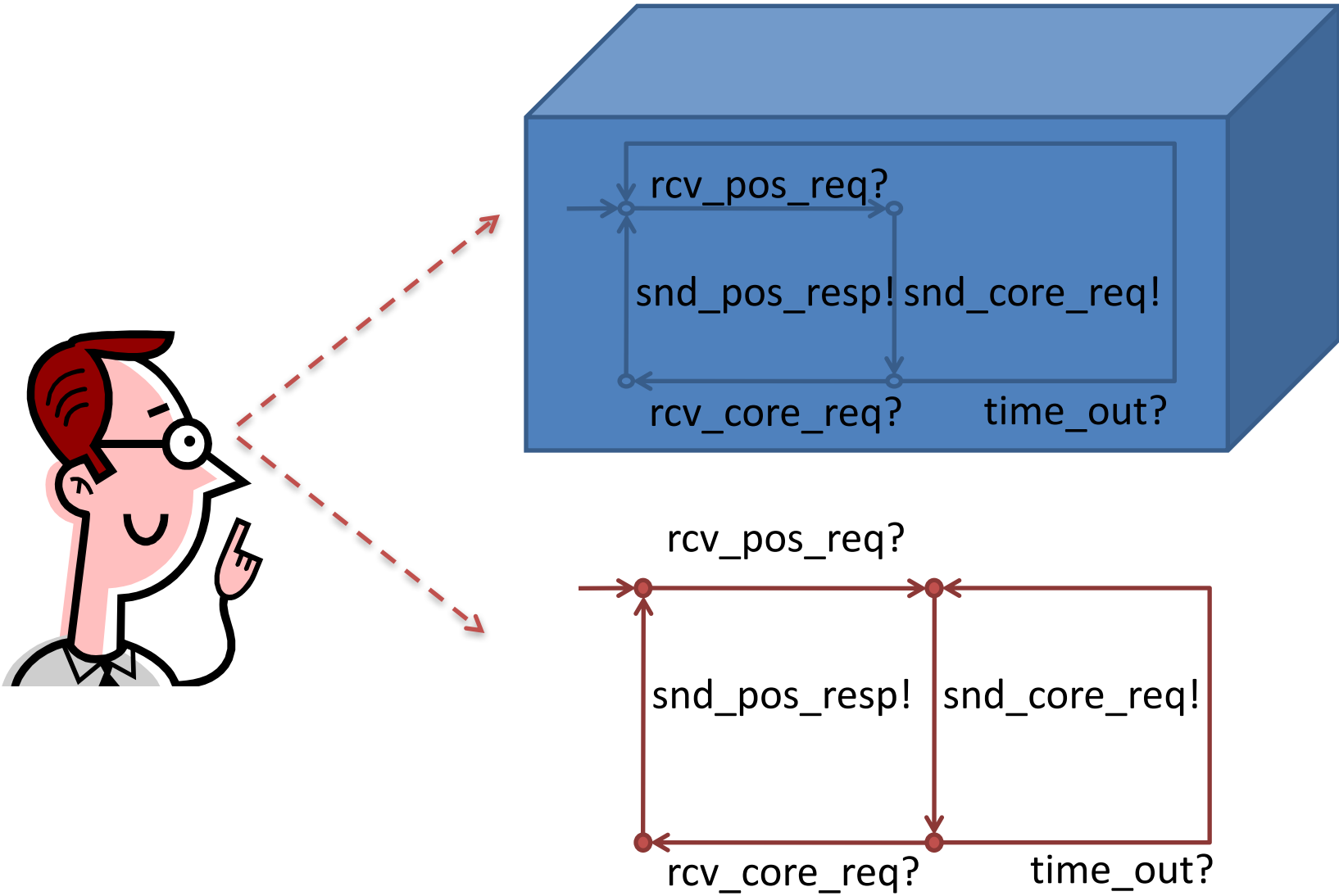
[Tretmans, 2008]

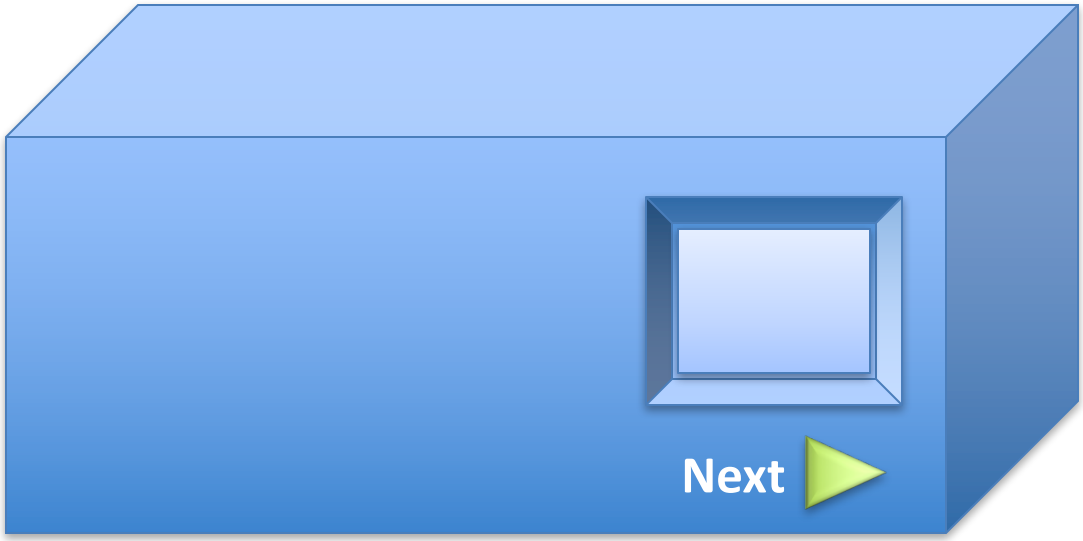

Conformance Testing
Test-Case Generation
Test Model
System Under Test
Conformance Analysis
Formal Semantics
Exhaustiveness
Soundness
Testing Hypothesis
Test Model Semantics
System Under Test Semantics
Conformance Relation
Common Semantic Domain

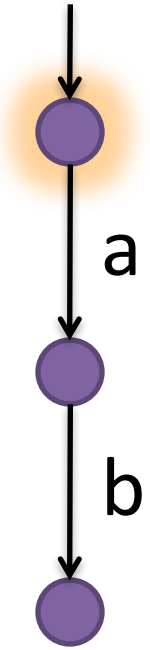
Theory:

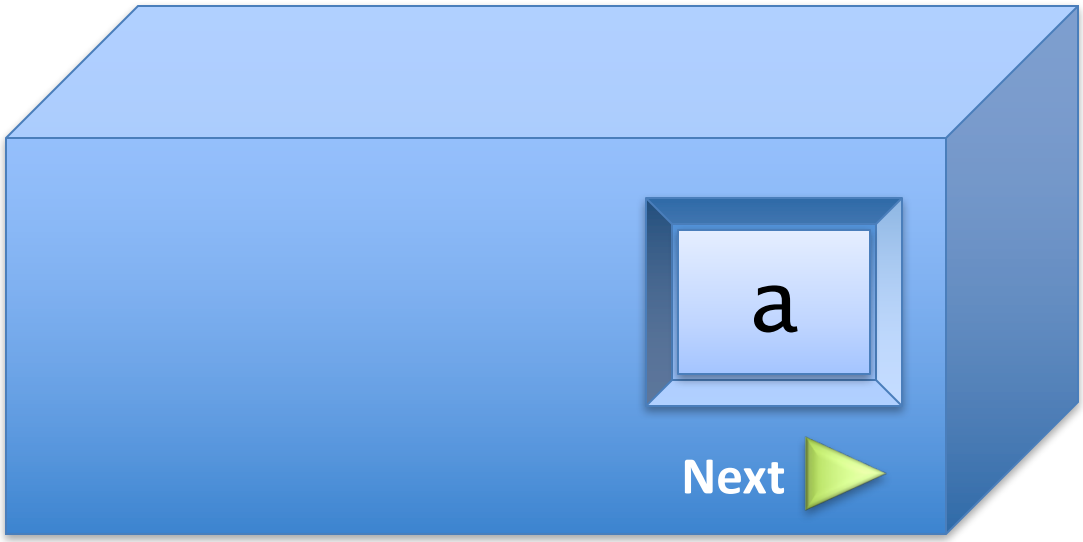
Testing from Labeled Transition Systems

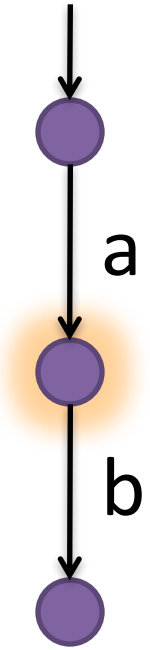
Equivalence By Observation

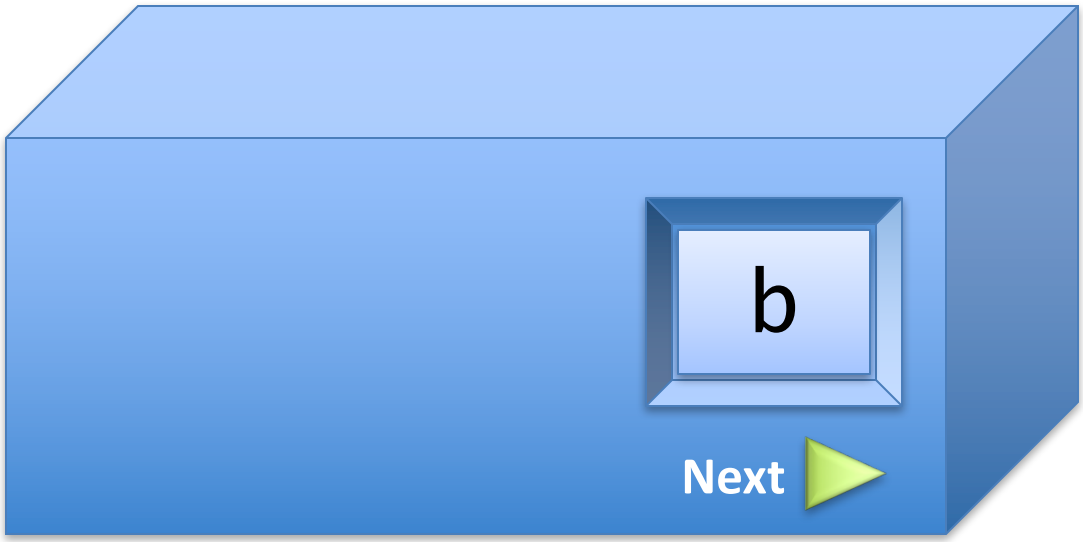


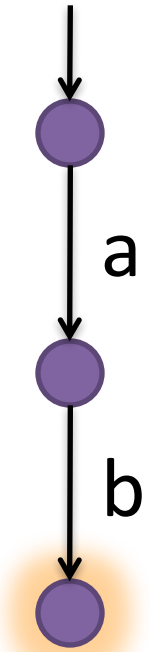

Next

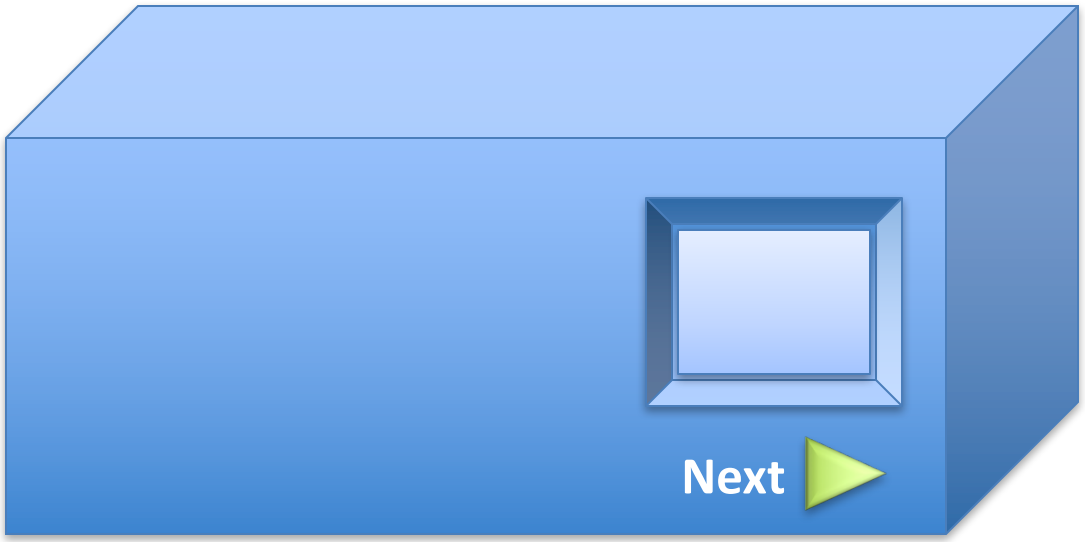

a
b

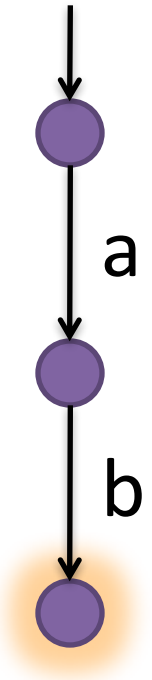

a
Next


a
b


b
Next


a
b


Next

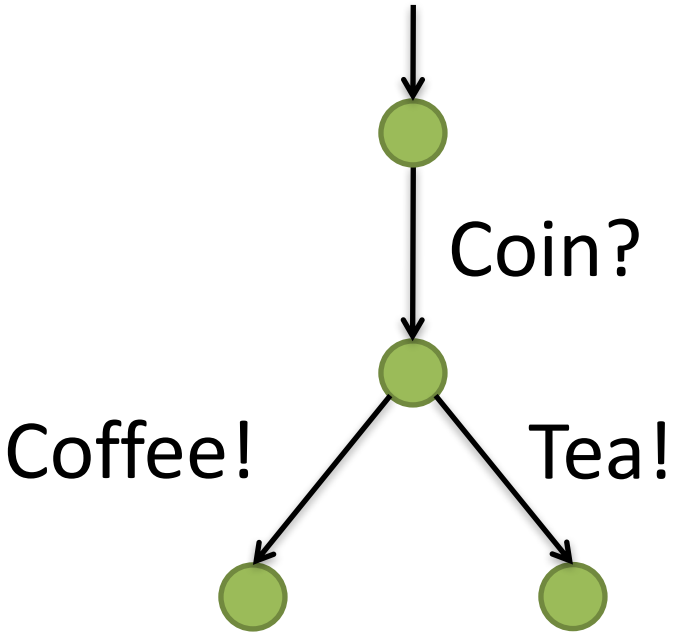

a
b

Completed Trace Equivalence

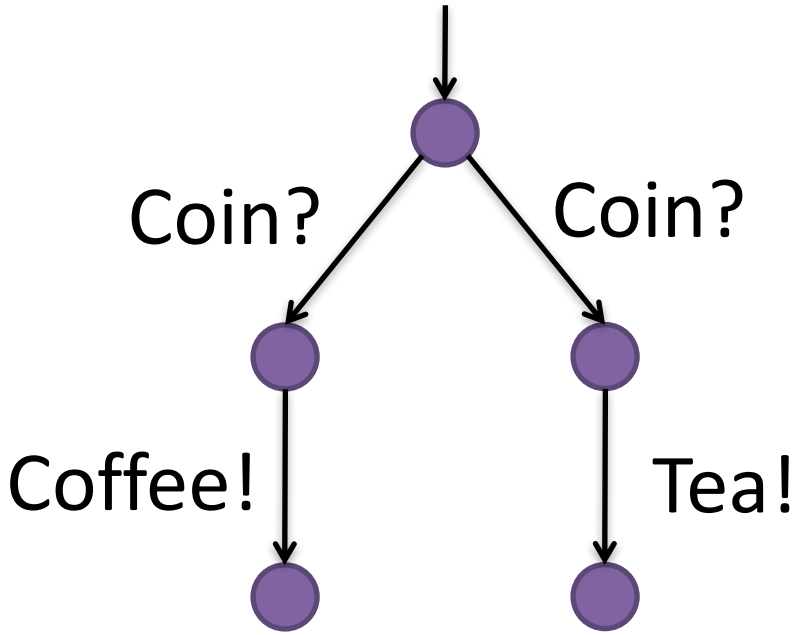
$$I \approx_{ctr} S$$

traces(I) = traces(S)

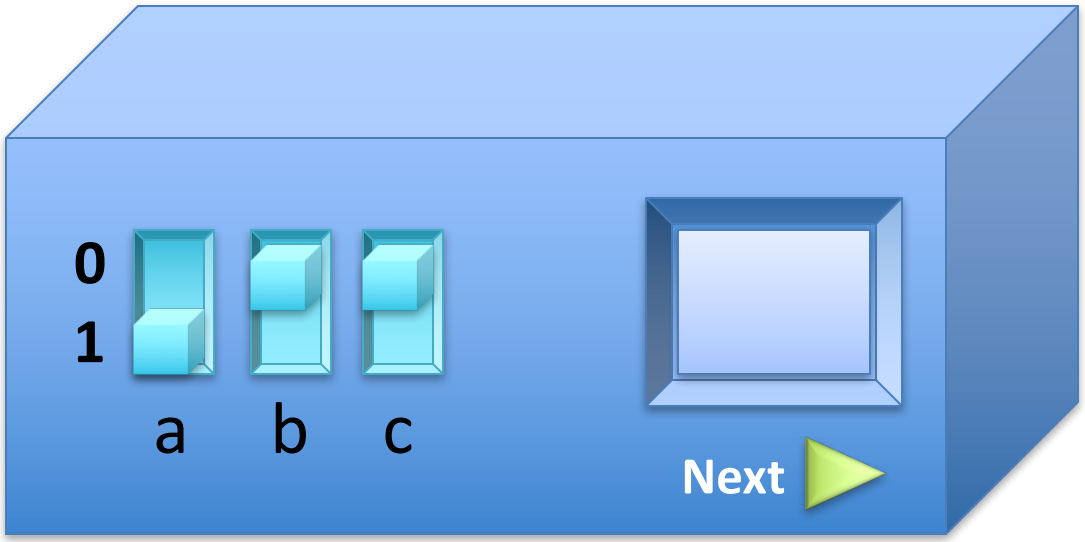
c_traces(I) = c_traces(S)

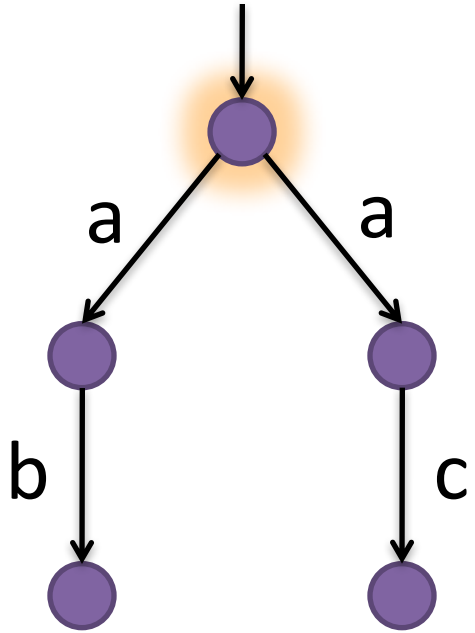


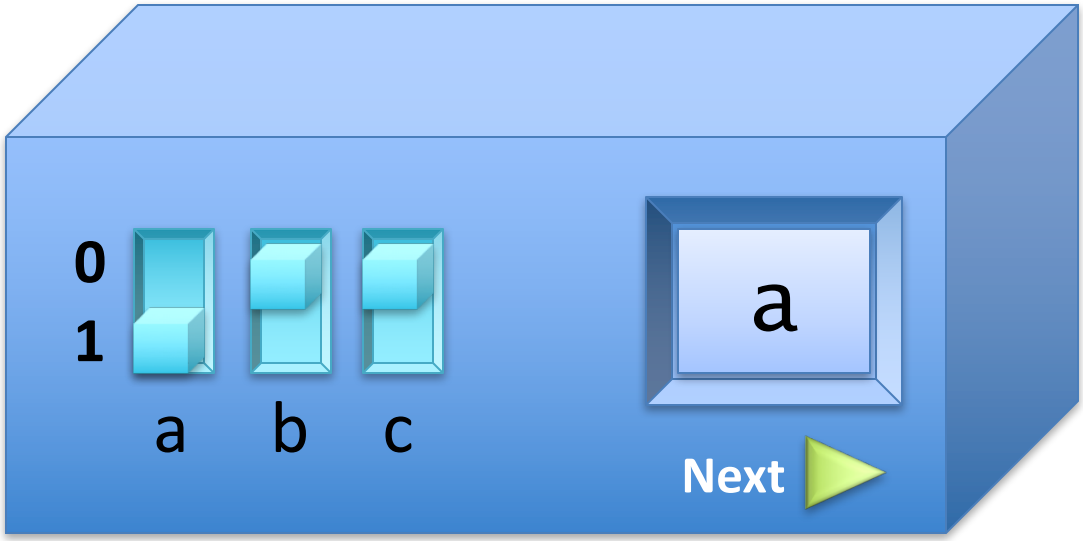
$\overset{?}{\approx}_{ctr}$

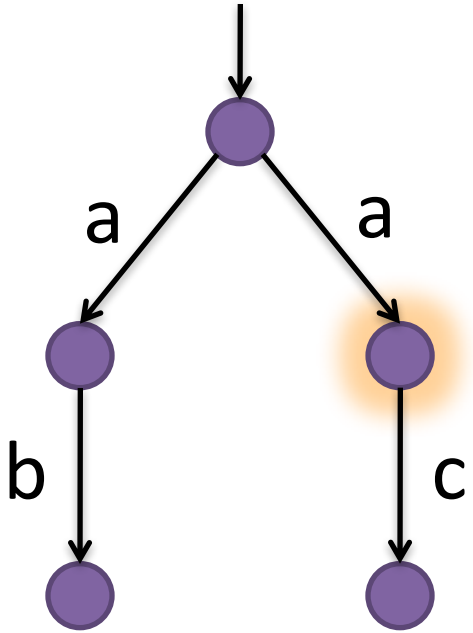


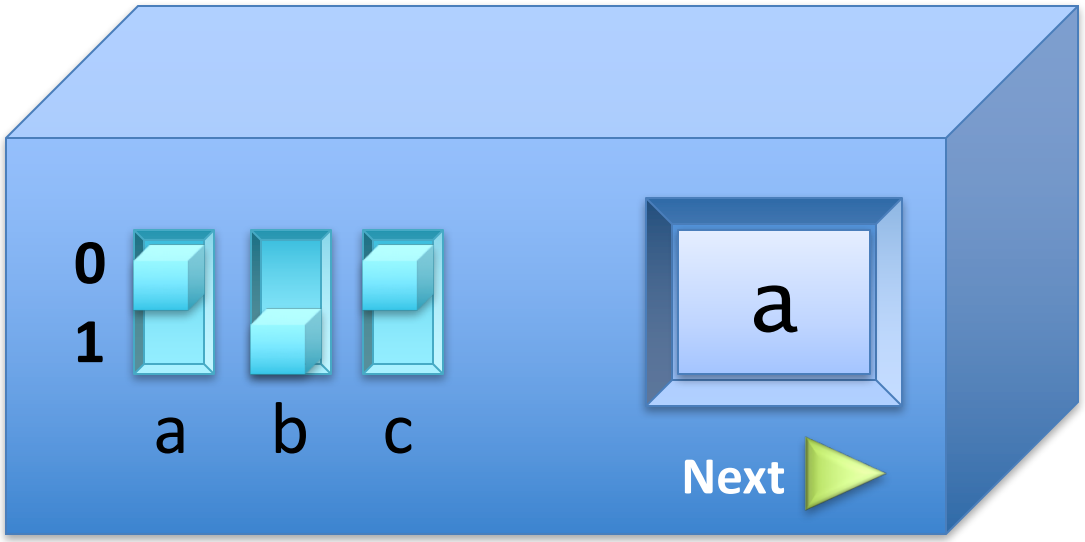
Moral of the story:
trace equivalence is too coarse
for conformance testing open systems

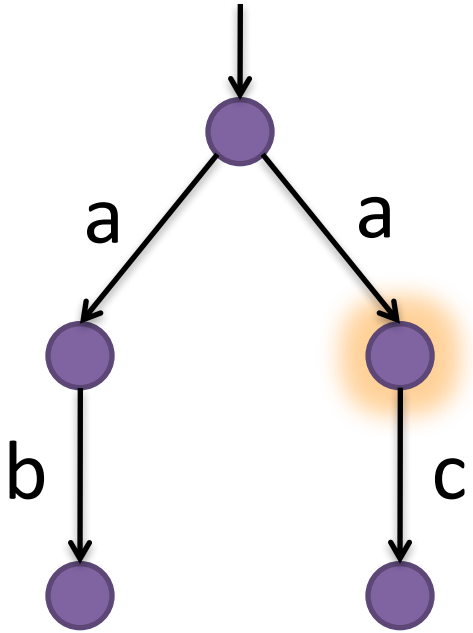

0
1
a
b
c
Next

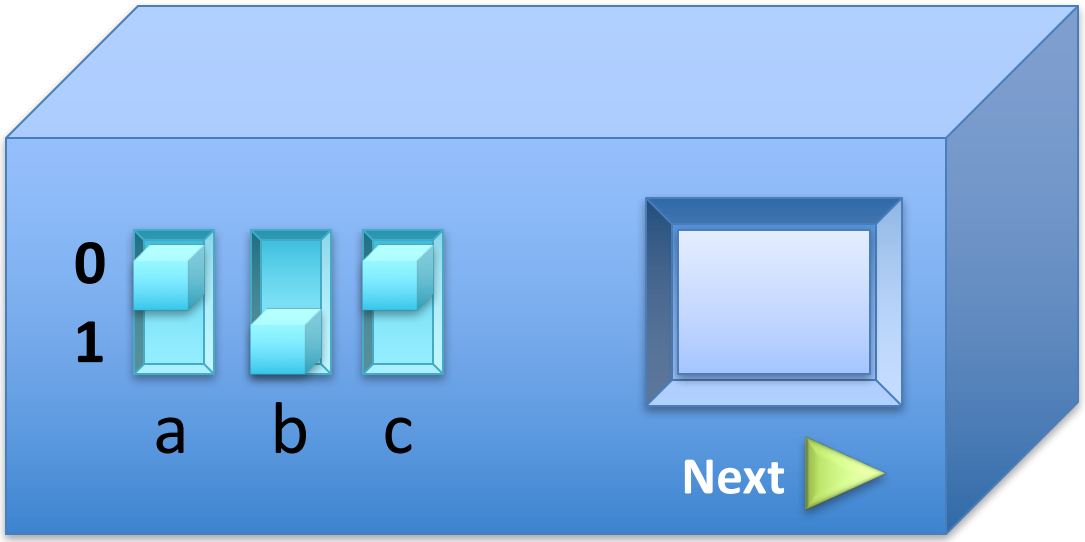

a
a
b
c

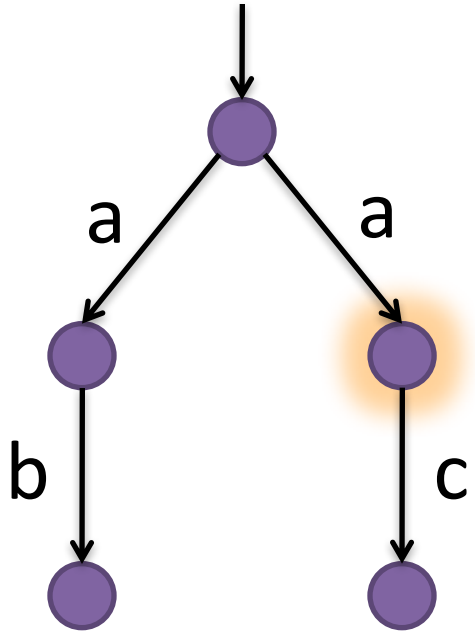

0
1
a
b
c
a
Next

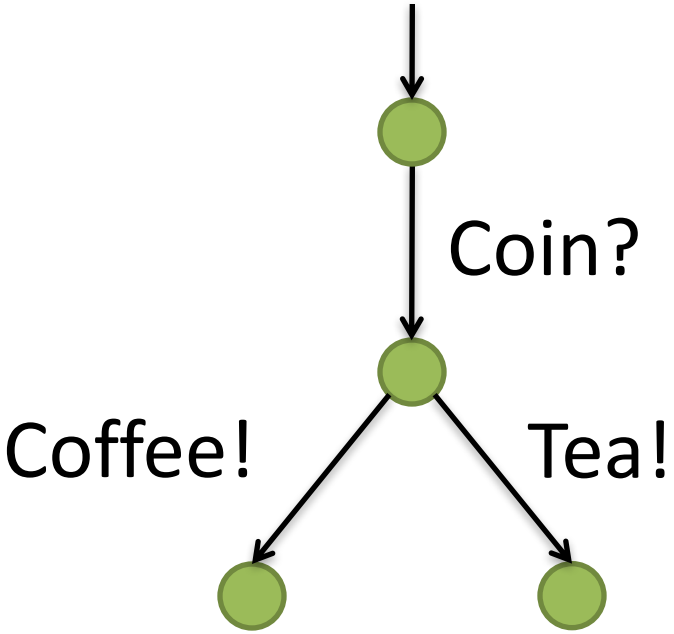

a
a
b
c

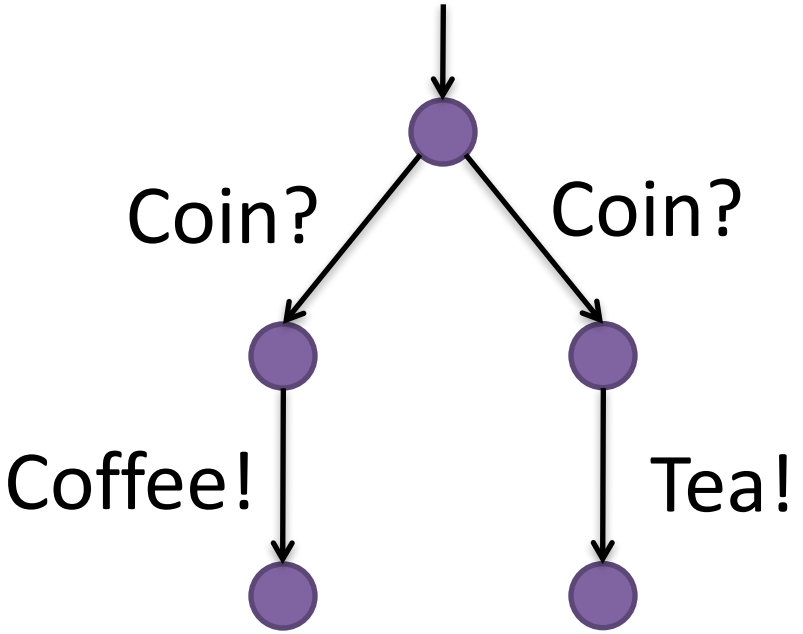

0
1
a
b
c
a
Next


a
a
b
c

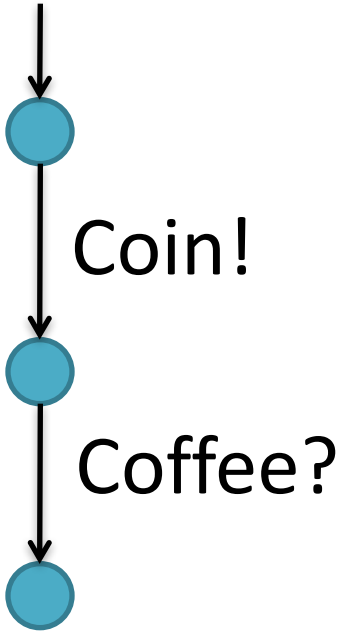

0
1
a
b
c
Next


a
a
b
c

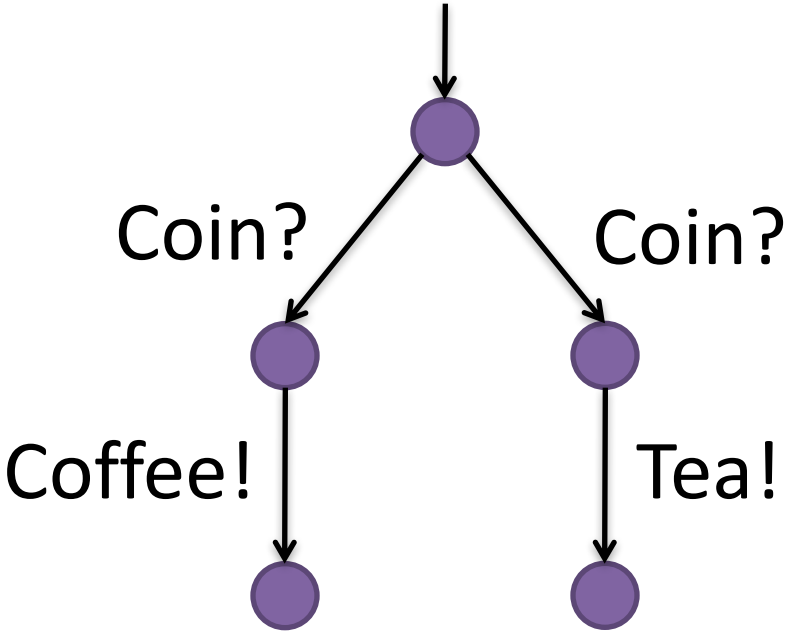

Coin?
Coffee!
Tea!


Coin?
Coin?
Coffee!
Tea!

What we did using switches



Environment



System

Testing Equivalence

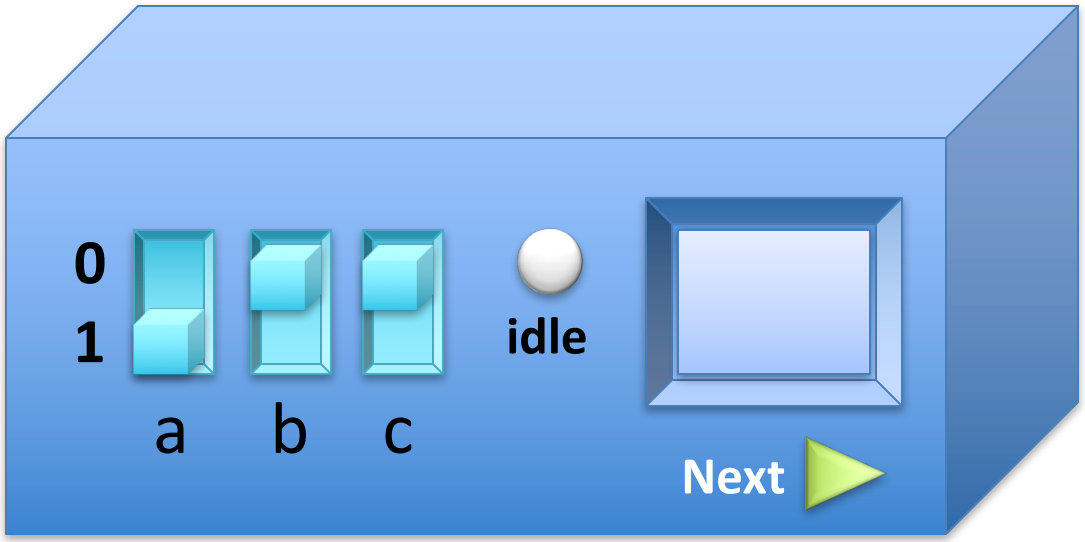
$$I \approx_{te} S$$

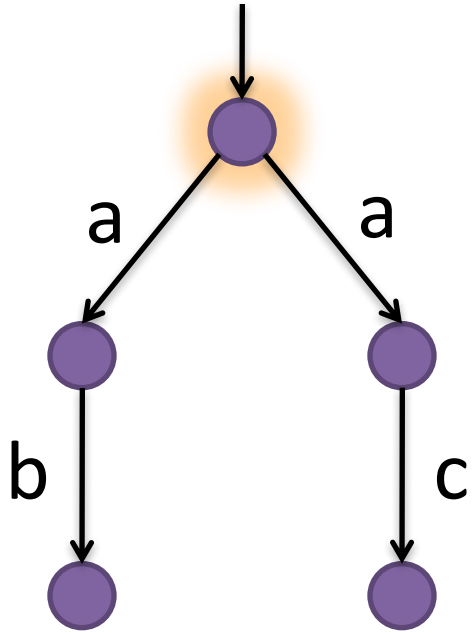
[De Nicola and Hennessy'84]
[Brookes,Hoare, and Roscoe'84]
[Darondeau'82][Kennaway'81]

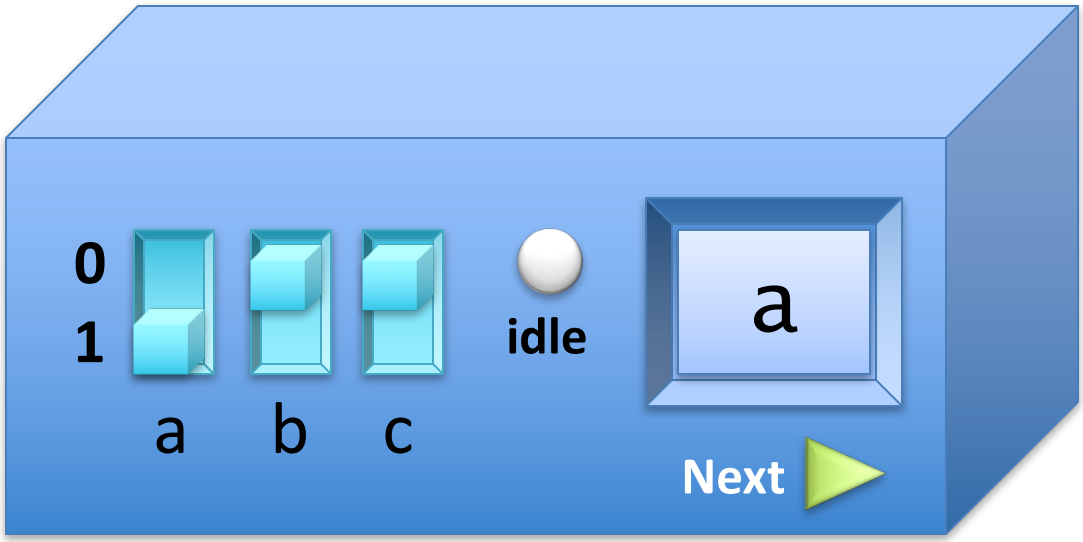
for every environment E:

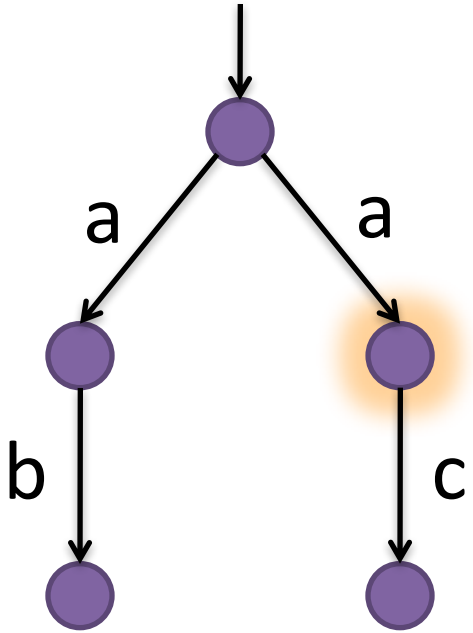
$$traces(E \parallel I) = traces(E \parallel S)$$

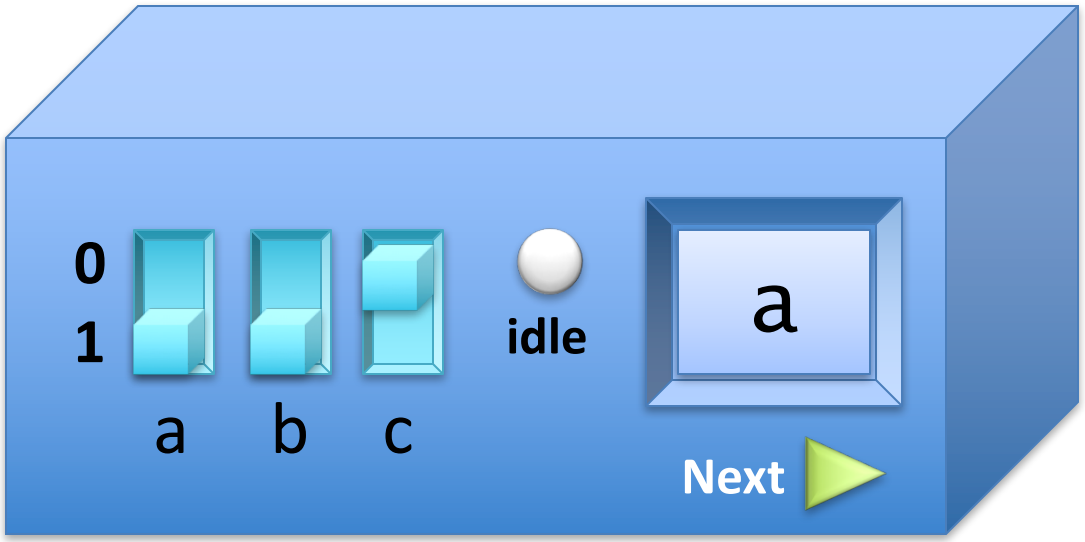
$$c\_traces(E \parallel I) = c\_traces(E \parallel S)$$

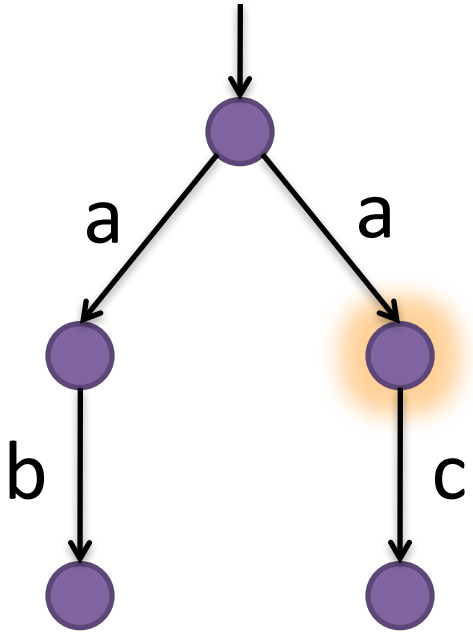

0
1
a
b
c
idle
Next

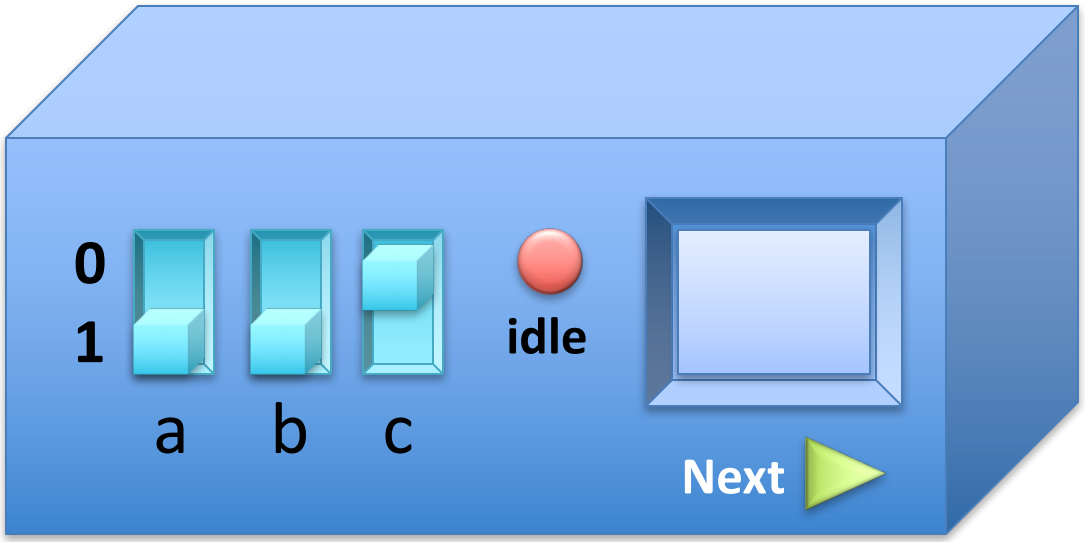

a
a
b
c

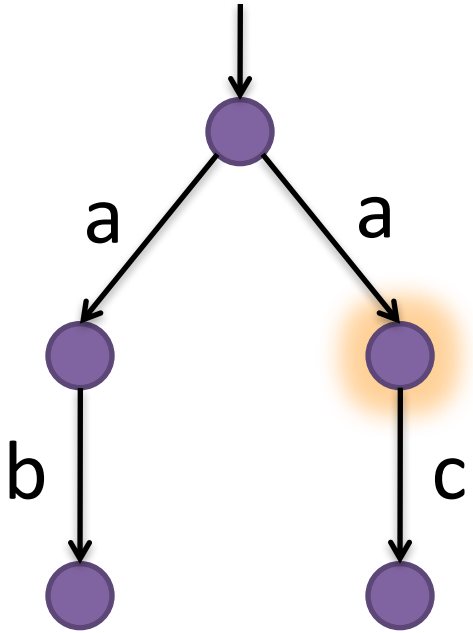

0
1
a
b
c
idle
a
Next


a
a
b
c

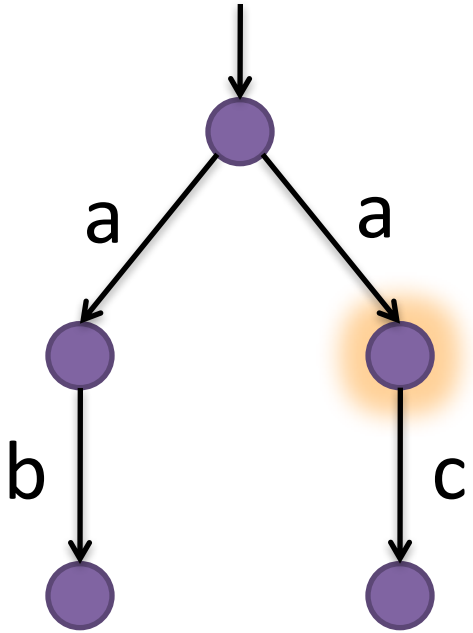

0
1
a
b
c
idle
a
Next

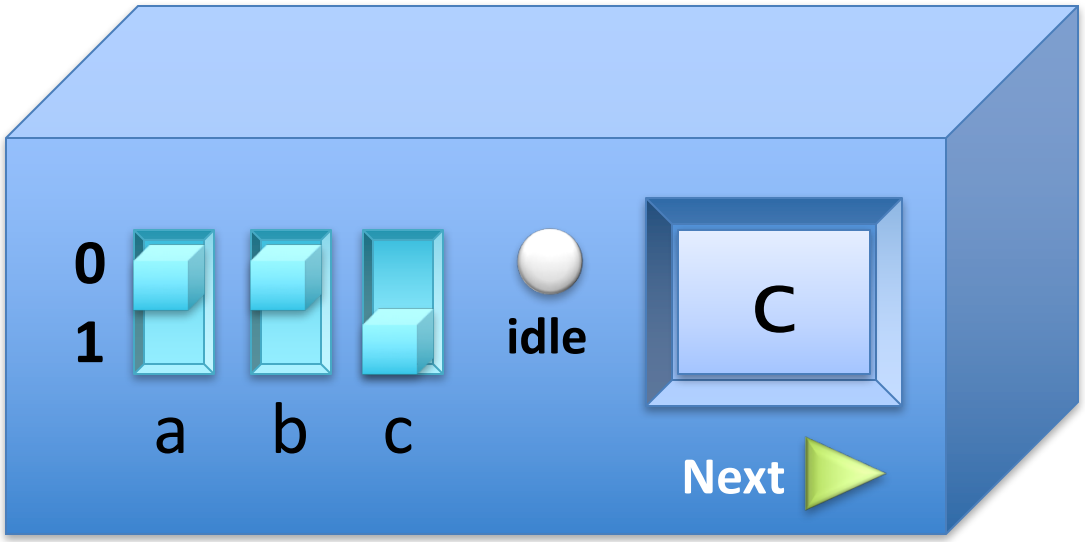

a
a
b
c

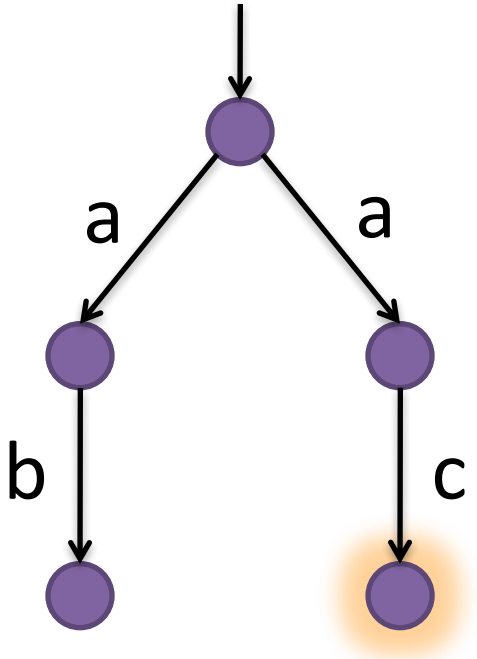

0
1
a
b
c
idle
Next


a
a
b
c

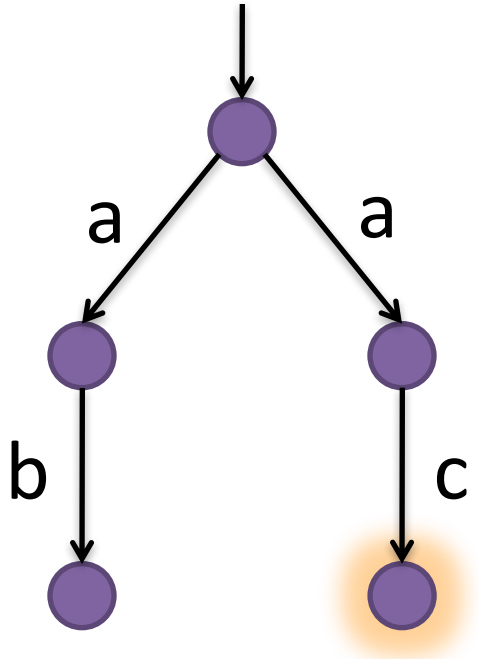

0
1
a
b
c
idle
Next

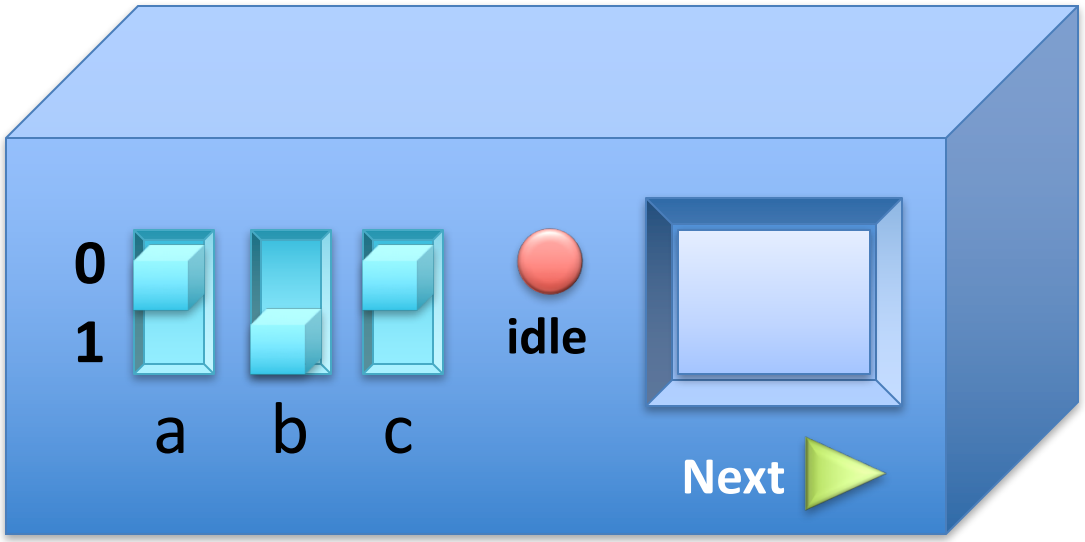

a
a
b
c

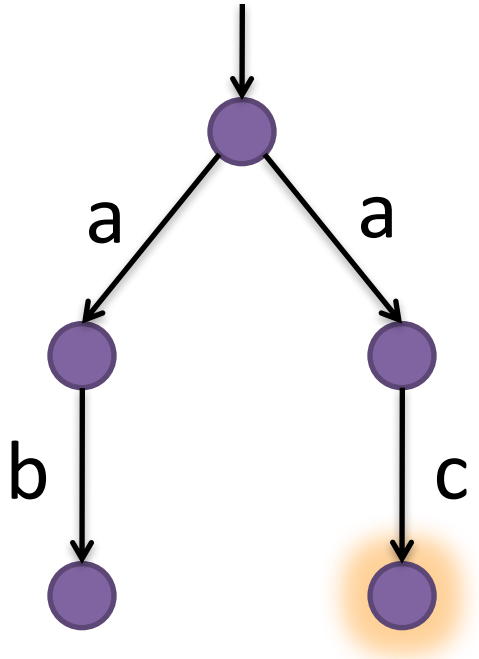

0
1
a
b
c
idle
C
Next

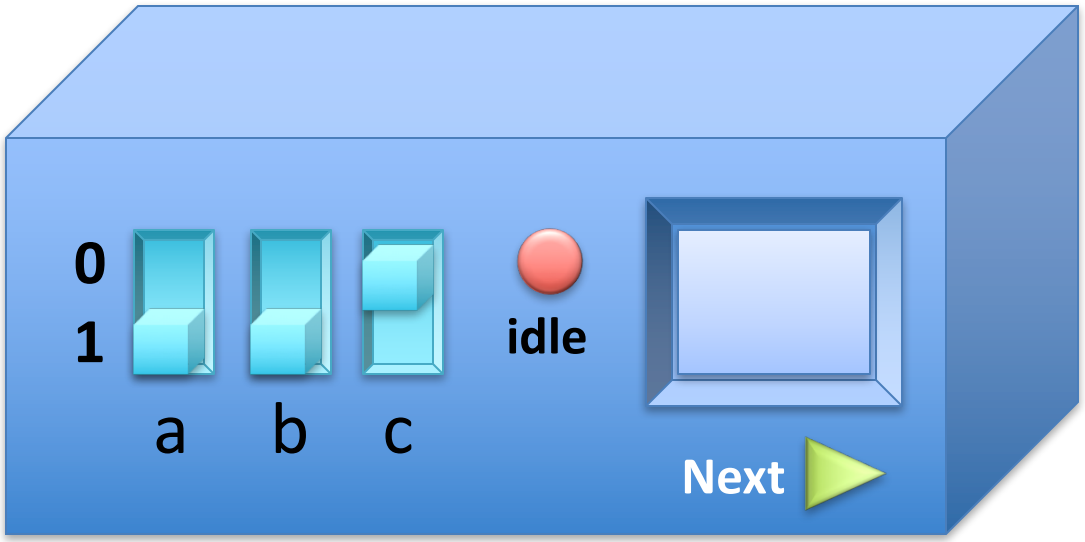

a
a
b
c

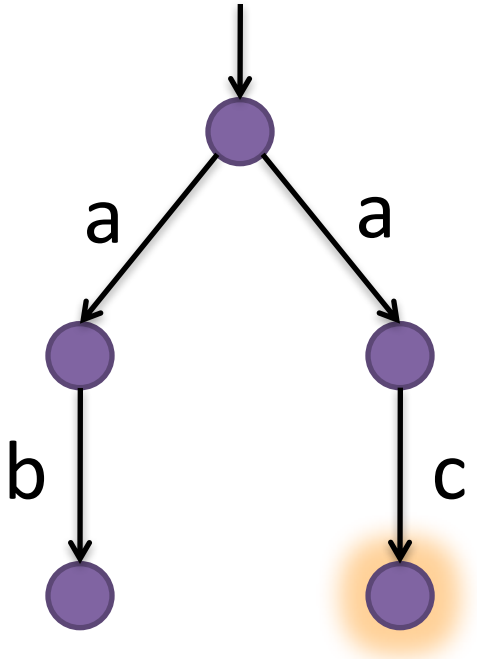

0
1
a
b
c
idle
Next

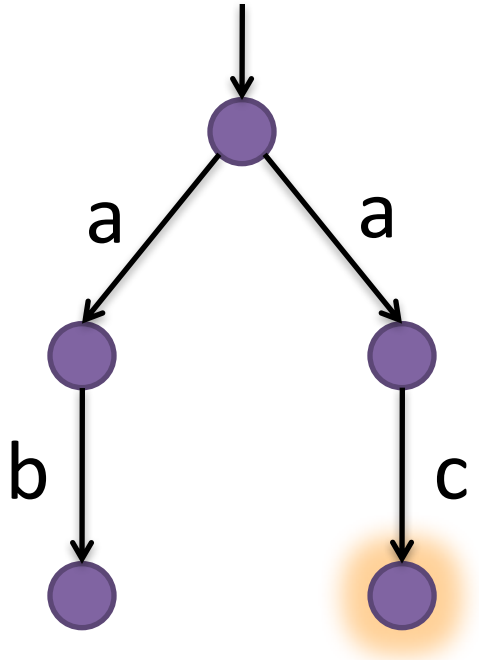

a
a
b
c


0
1
a
b
c
idle
Next

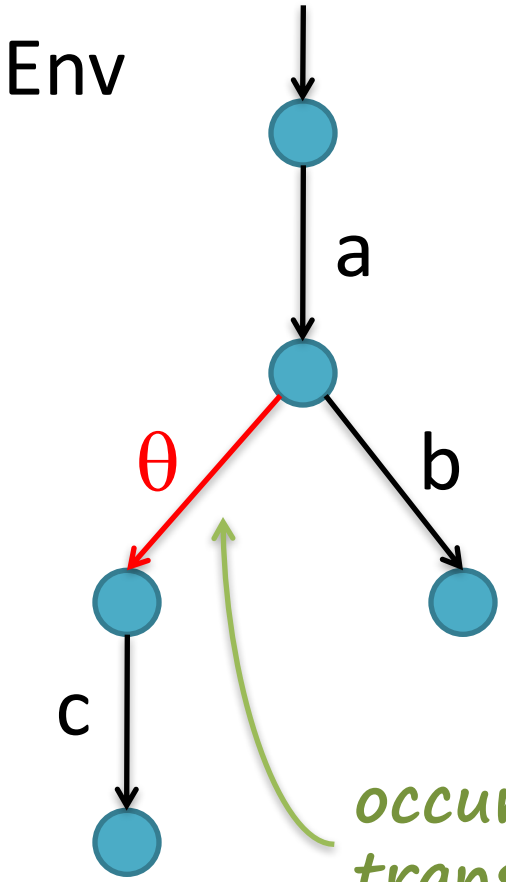

a
a
b
c


0
1
a
b
c
idle
Next

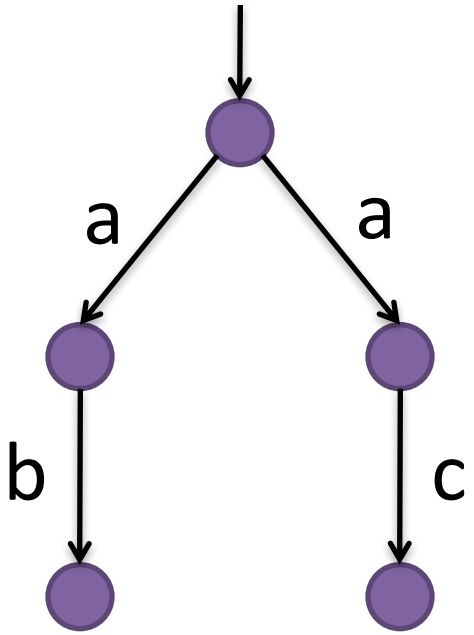

a
a
b
c

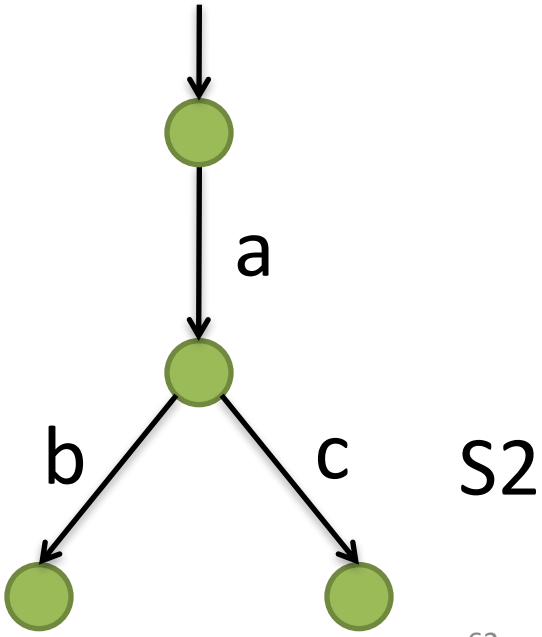


Now, we know that we have completed a trace.

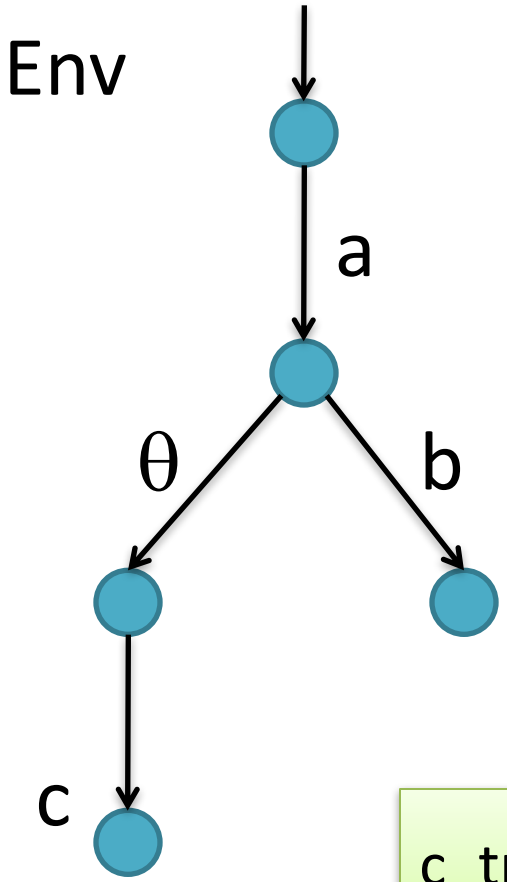


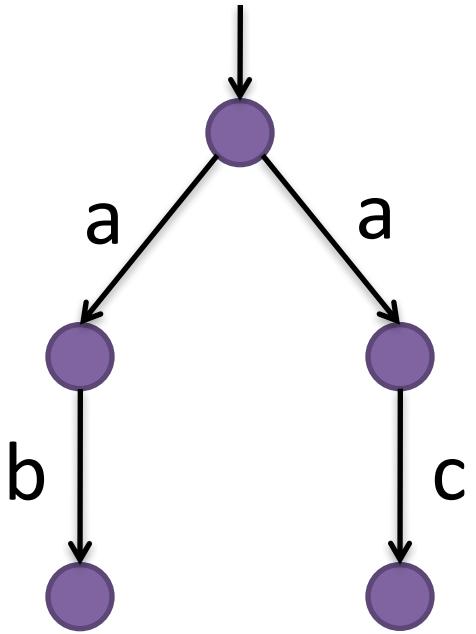
occurs if no other transition can

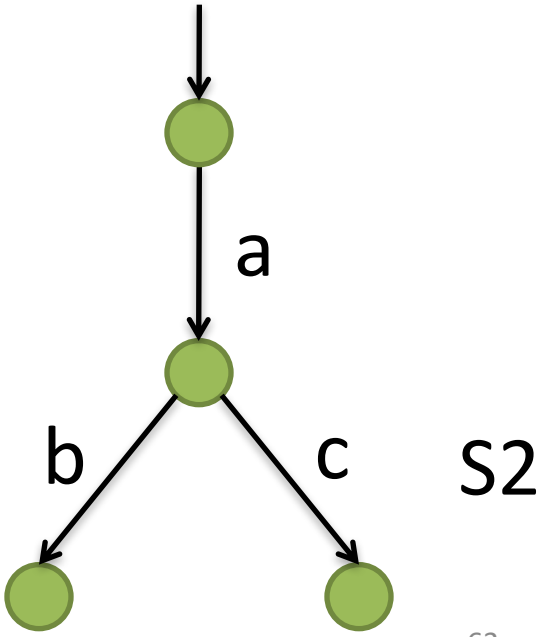




$c\_traces(Env \lceil\!| S_1) = \{ab,\ a\theta c\}$







$c\_traces(Env \lceil\!| S_2) = \{ab\}$

Refusal Equivalence

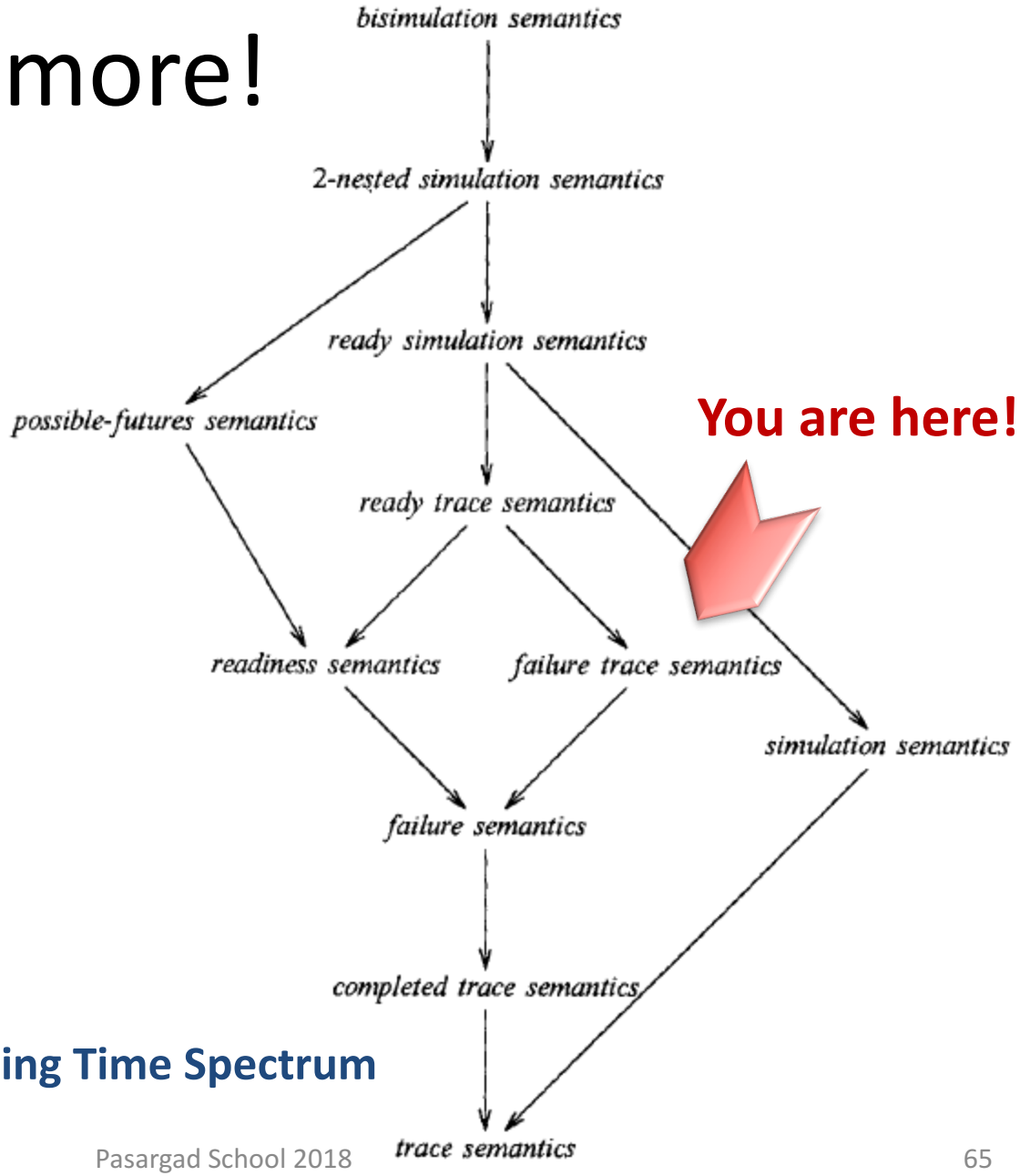
$$I \approx_{rf} S \quad \text{[Philips'87]}$$

for every environment E:

$$traces(E \parallel I) = traces(E \parallel S)$$

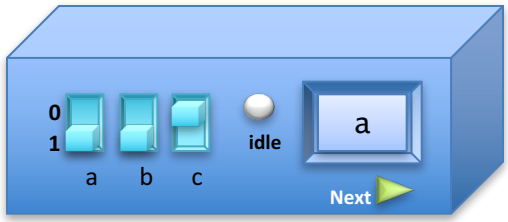
$$c\_traces(E \parallel I) = c\_traces(E \parallel S)$$

And there is more!

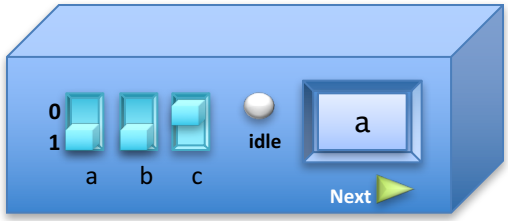


The RvG Linear Time – Branching Time Spectrum

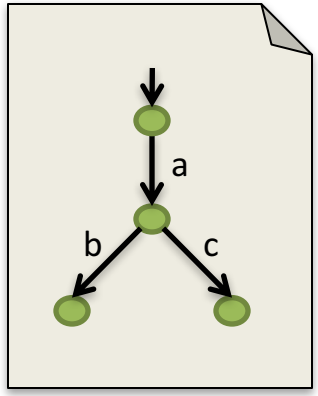
Defining specifications at a higher-level



Impl. 1



Impl. 2



Specification

Testing Pre-order

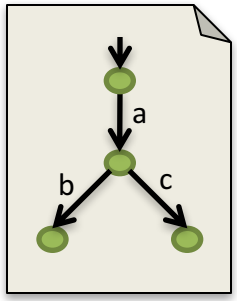
$$I \sqsubseteq_{te} S$$ [De Nicola and Hennessy'84]

for every environment E:

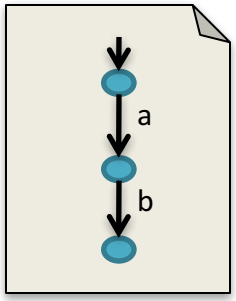
$$traces(E \parallel I) \subseteq traces(E \parallel S)$$

$$c\_traces(E \parallel I) \subseteq c\_traces(E \parallel S)$$

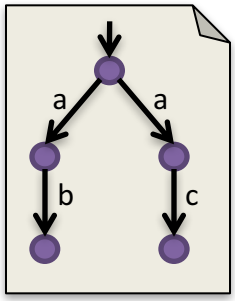
Restriction to Specification



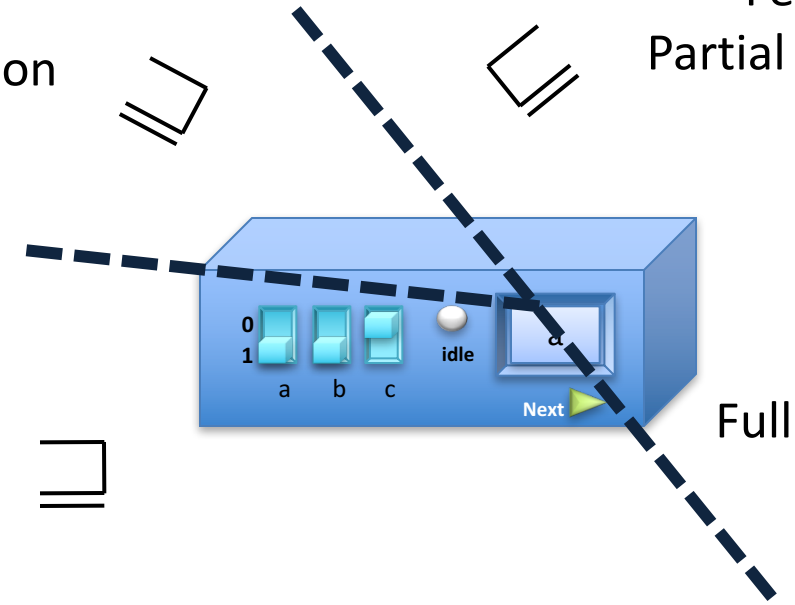
Feature #1
Partial Specification



Feature #2
Partial Specification



Feature #3
Partial Specification



Full Implementation

Restriction to Specification

I **conf** S [Brinksma87]

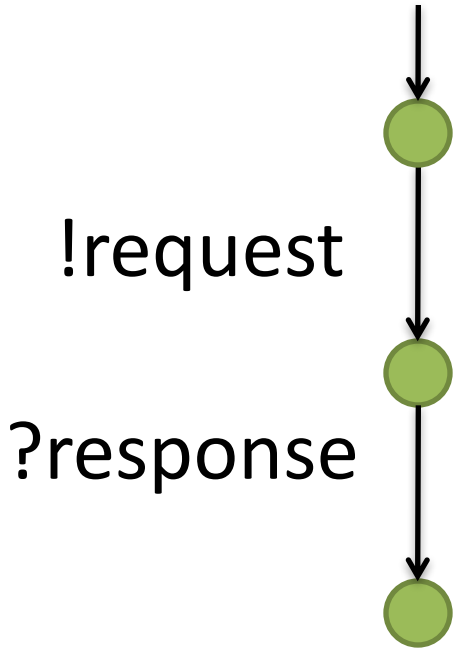
for every environment E:

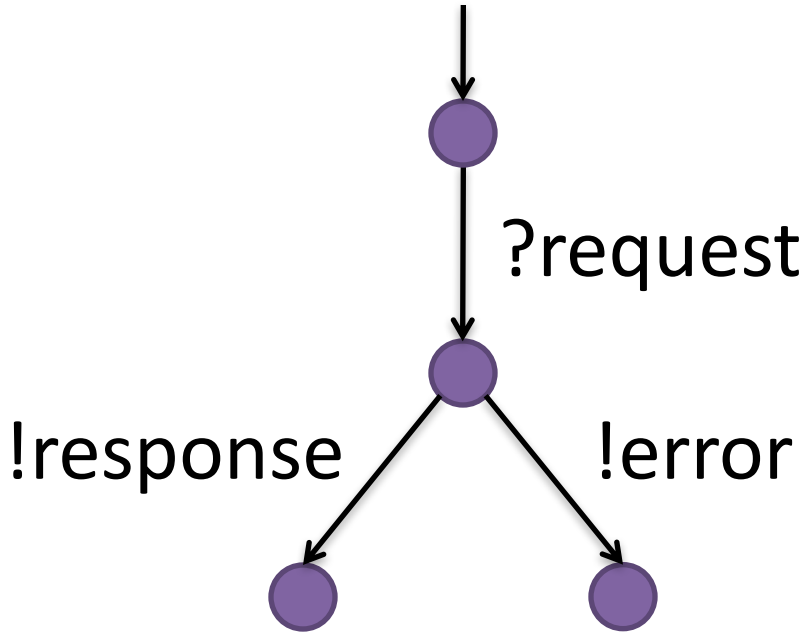
traces(E || I) ∩ traces(S) ⊆ traces(E || S)

c_traces(E || I) ∩ traces(S) ⊆ c_traces(E || S)

I/O Transition Systems

Distinguishing between input and output actions





Pre-orders on I/O transition systems

- The same notions apply here
  - I/O test pre-order
  - I/O refusal pre-order

$$\mathtt{I} \sqsubseteq_{\text{ior}} \mathtt{S}$$

for every environment E:

$$\text{traces}(\mathtt{E} \parallel \mathtt{I}) \subseteq \text{traces}(\mathtt{E} \parallel \mathtt{S})$$

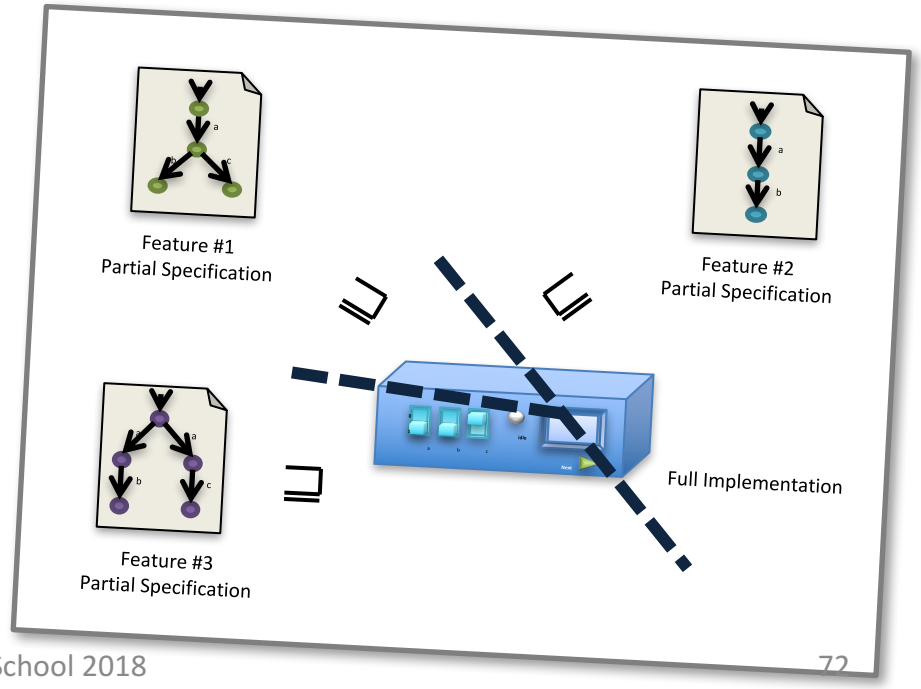
$$\text{c\_traces}(\mathtt{E} \parallel \mathtt{I}) \subseteq \text{c\_traces}(\mathtt{E} \parallel \mathtt{S})$$

I/O Conformance

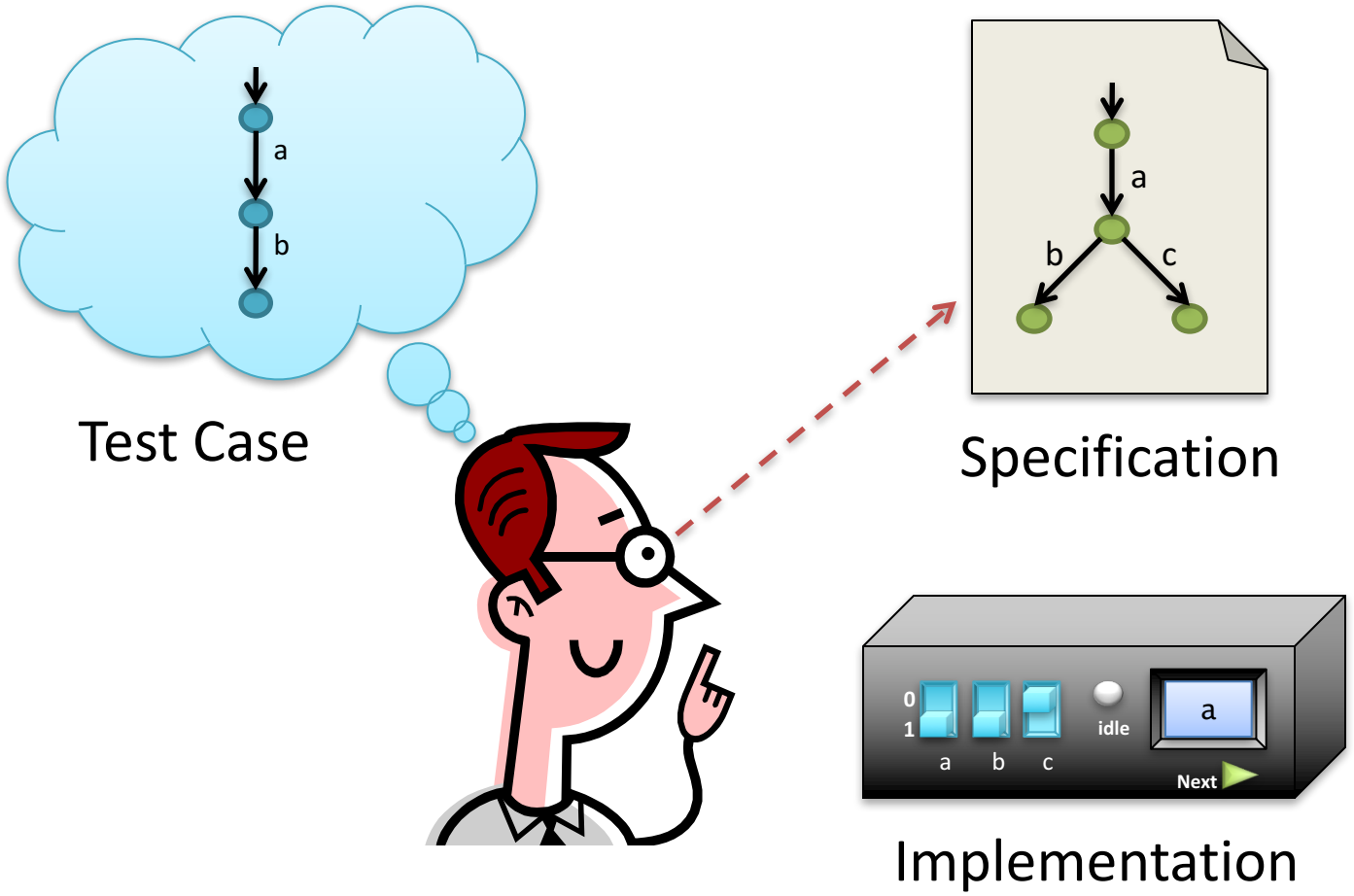
Informally,

I/O Conformance = I/O Refusal restricted to specification traces

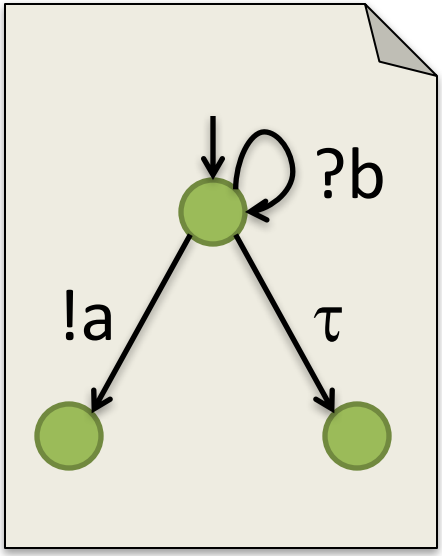
I ioco S [Tretmans'95]

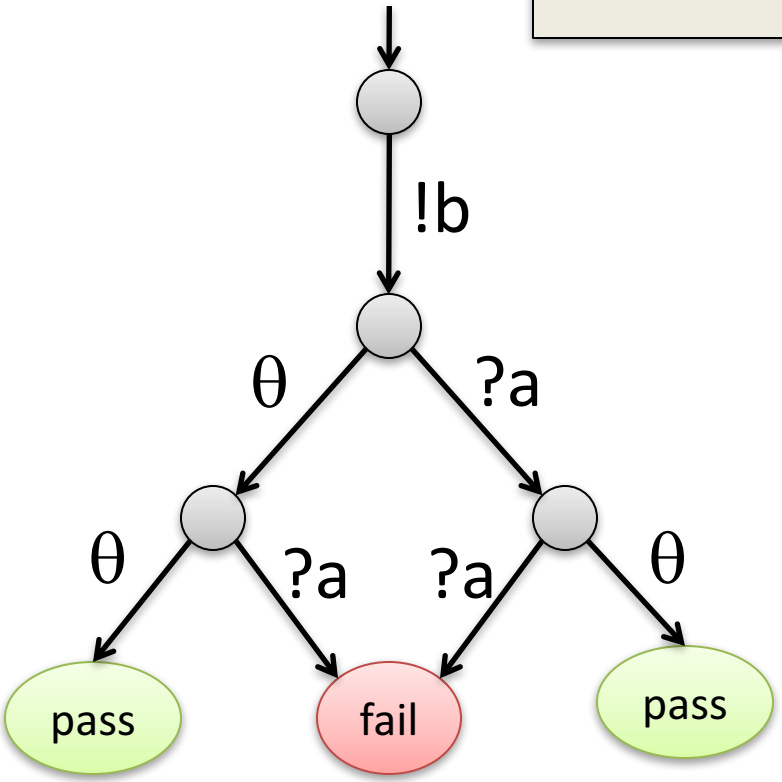


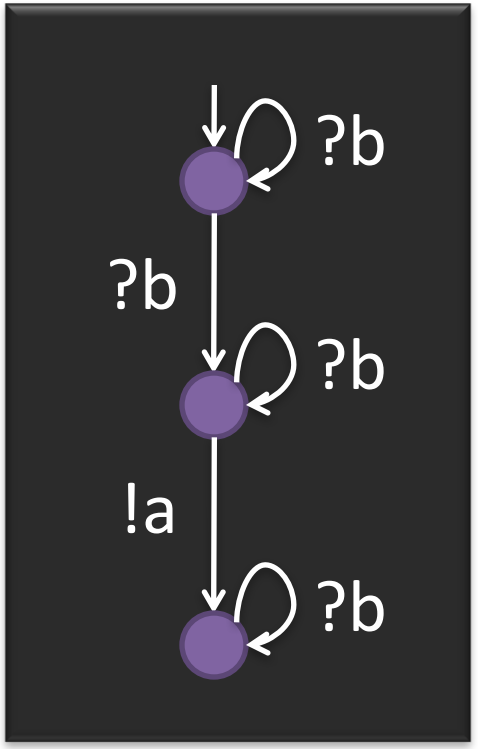
Black-box testing for ioco



Example

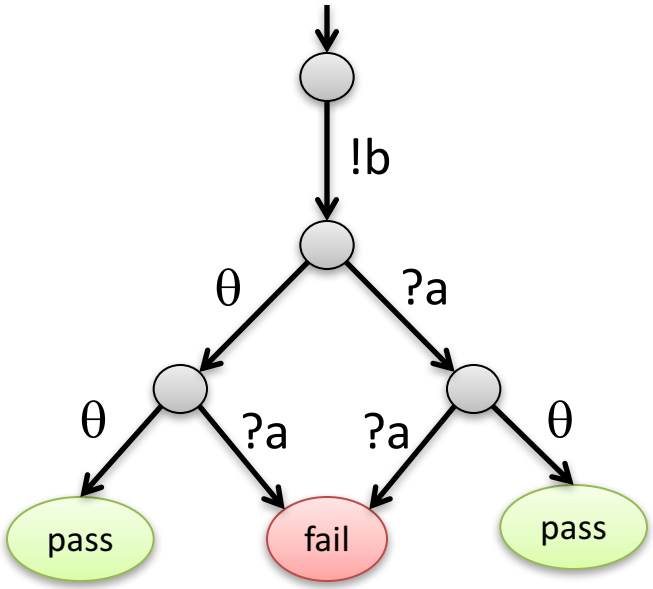






ioco Test Cases

- I/O transition systems
- The only terminal states: pass and fail
- Reversed I/O actions
- Special action θ
- Finite and deterministic



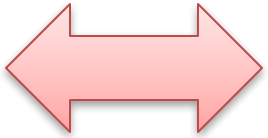
Automatic Test Case Generation

- Init:
  - Generate an initial state
- Recursion:
  - At each point in the recursion choose non-deterministically between:
    1. Stopping the recursion
    2. Supplying an input
    3. Observing an output (one transition per output action)

Practice:

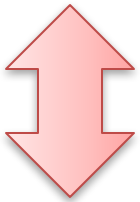
EFT and X-Ray Machine Cases

Electronic Funds Transfer (EFT)



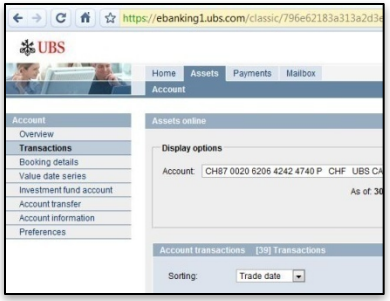
EFT Switch

Core Banking System



Interbank Network

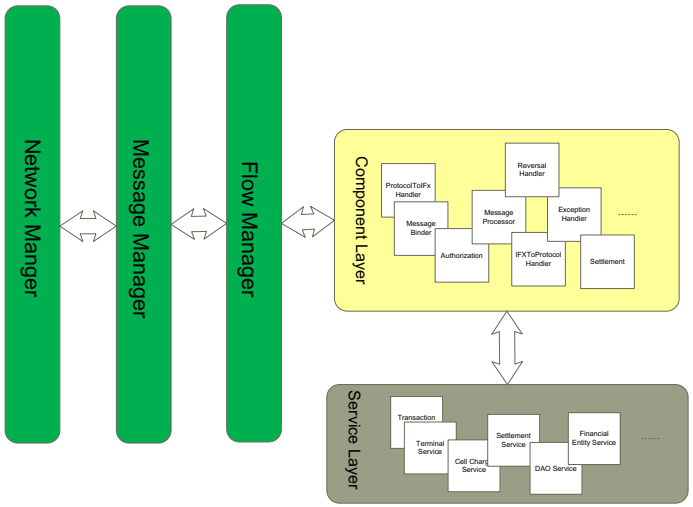
EFT Switch



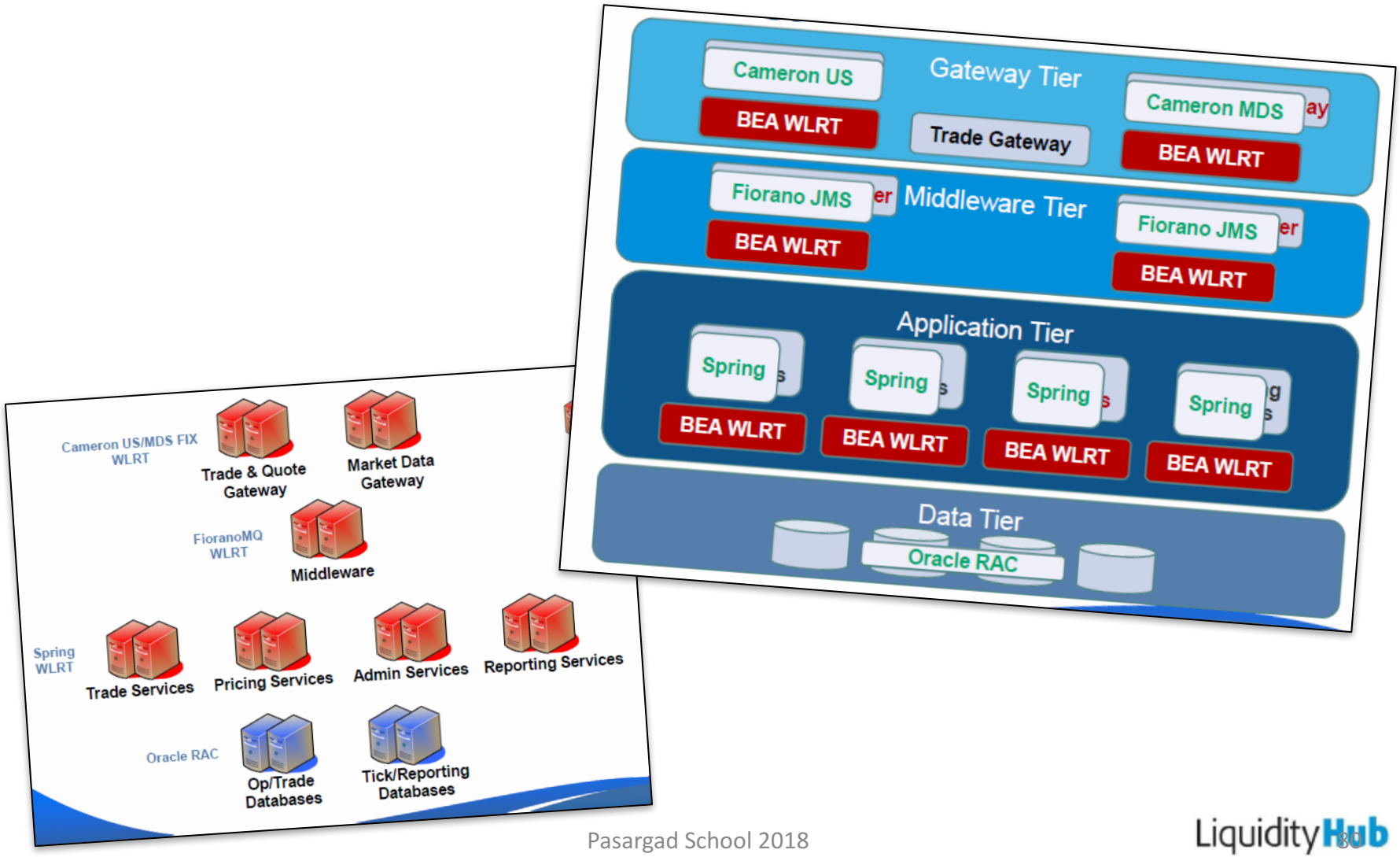
The System Under Test

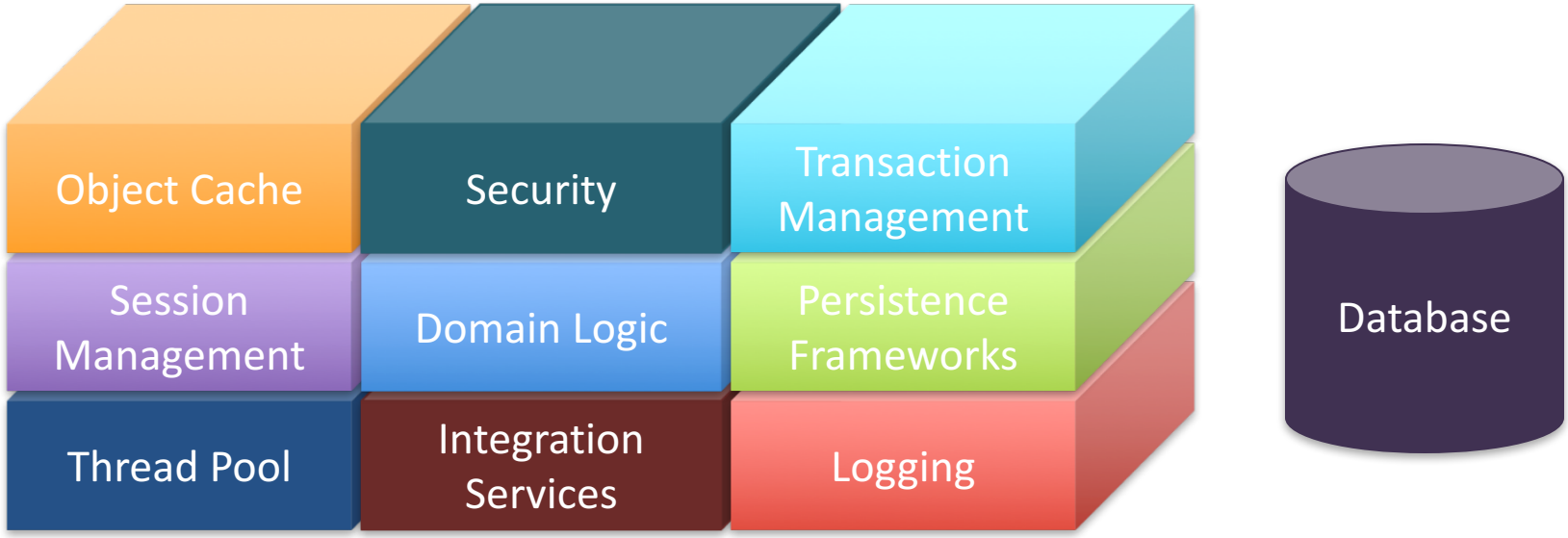
- An Operational EFT Switch, developed at Fanap
- Java Application (~100 KLOC)
- Extensive use of Java frameworks

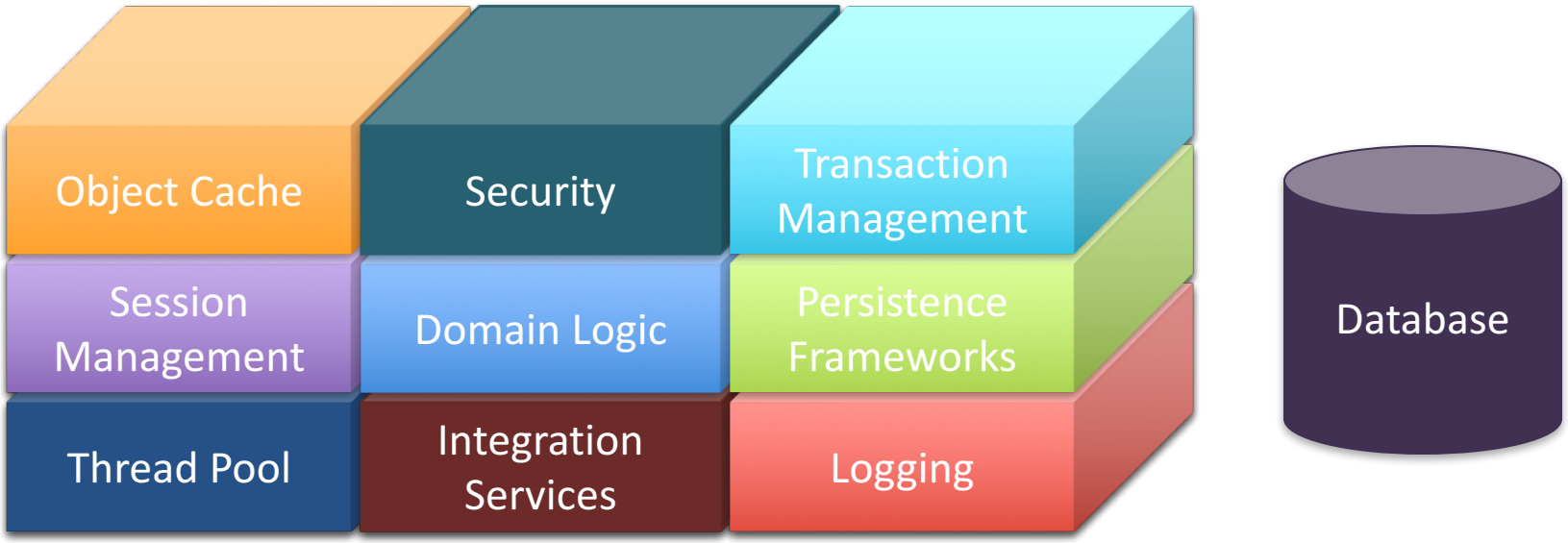
- *Already tested, but not with a disciplined view on concurrency*



Complex Architectures




Object Cache
Security
Transaction Management
Session Management
Domain Logic
Persistence Frameworks
Thread Pool
Integration Services
Logging
Database


Object Cache
Security
Transaction Management
Session Management
Domain Logic
Persistence Frameworks
Thread Pool
Integration Services
Logging
Database

Example Scenarios

POS Terminal

EFT Switch

Core Banking System

purchase request

purchase request

purchase response

purchase response

Example Scenarios

POS Terminal

EFT Switch

Core Banking System

purchase request

purchase request

purchase response

reversal req.

purchase resp.

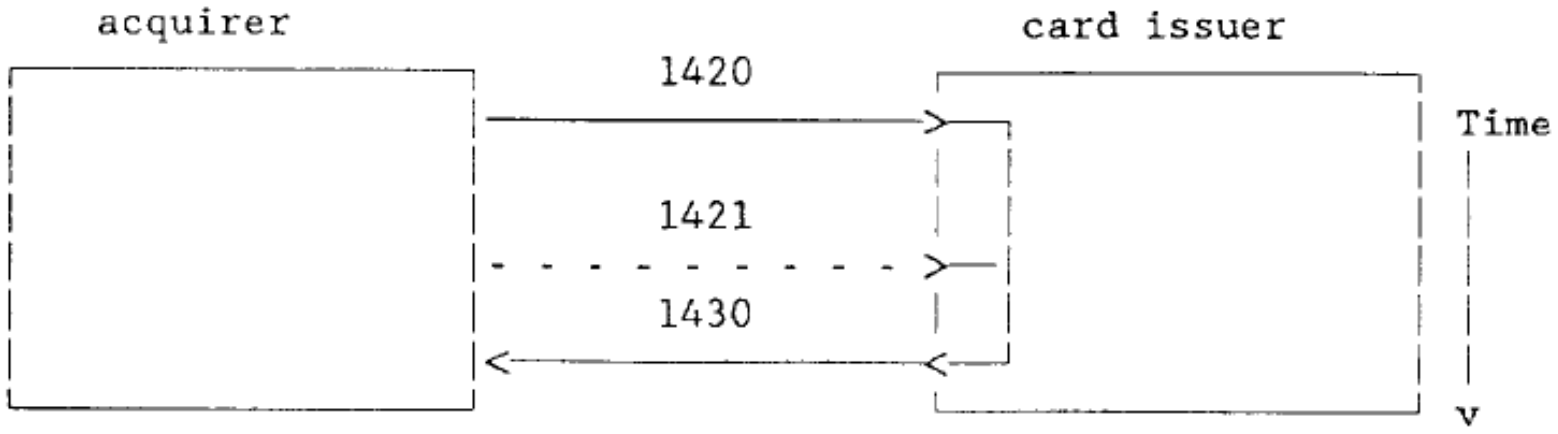
reversal response

reversal request

ISO 8583 Standard

Message Flows

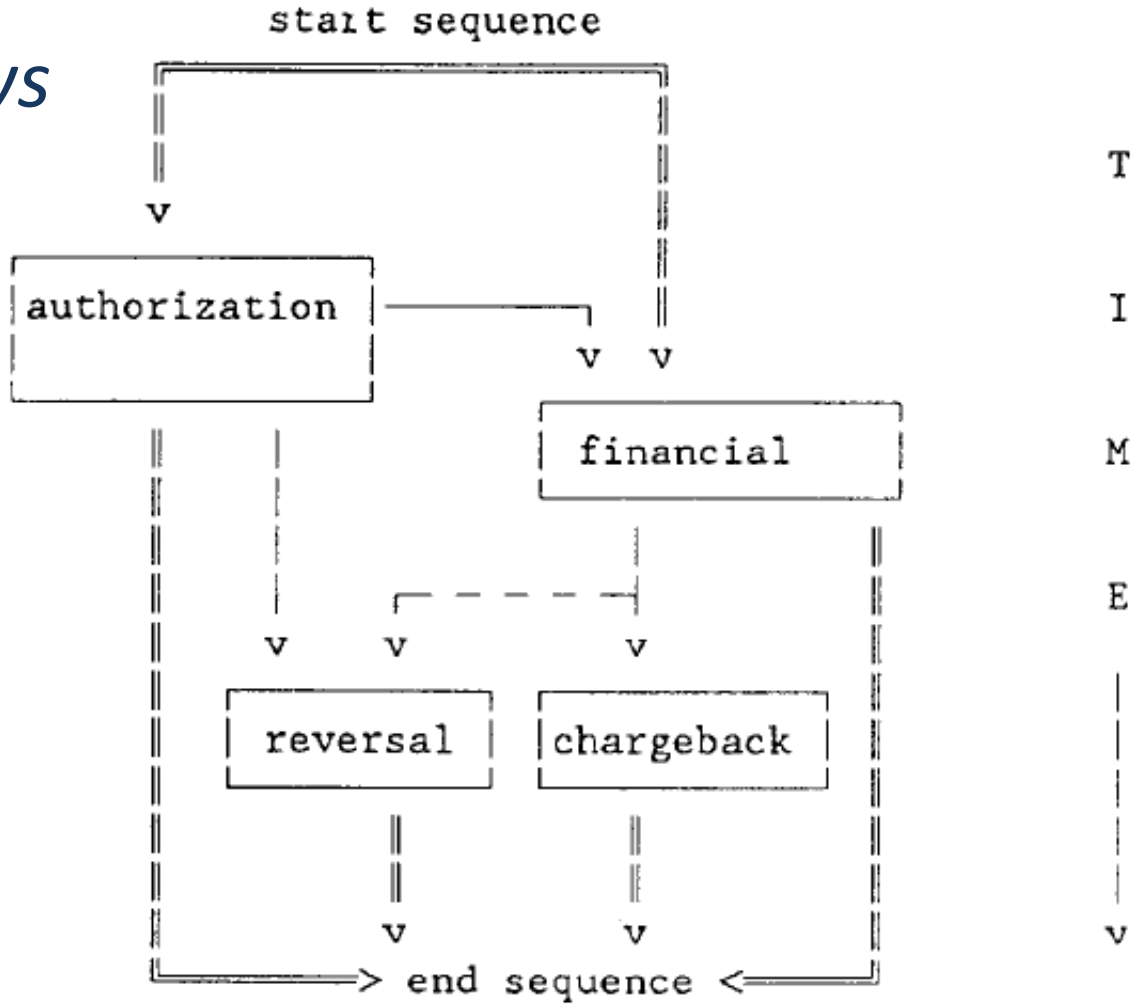
5.2.4 Reversal transactions



1420/1421 reversal advice/reversal advice repeat
1430 reversal advice response

ISO 8583 Standard

Transaction Flows



ISO 8583 Standard

Business Rules

b) The amount, transaction data element in a reversal advice or notification shall contain the amount to be reversed and shall be less than or equal to the original amount as shown in table 3.

c) Whenever the acquirer times out waiting for a response to an authorization or financial transaction request or advice, a reversal advice or notification of the transaction shall be sent (see 5.2.12).

Concurrent Transactions

POS #1

Purchase Transaction #1

Reversal Transaction #2

POS #2

Purchase Transaction #2

Our Method

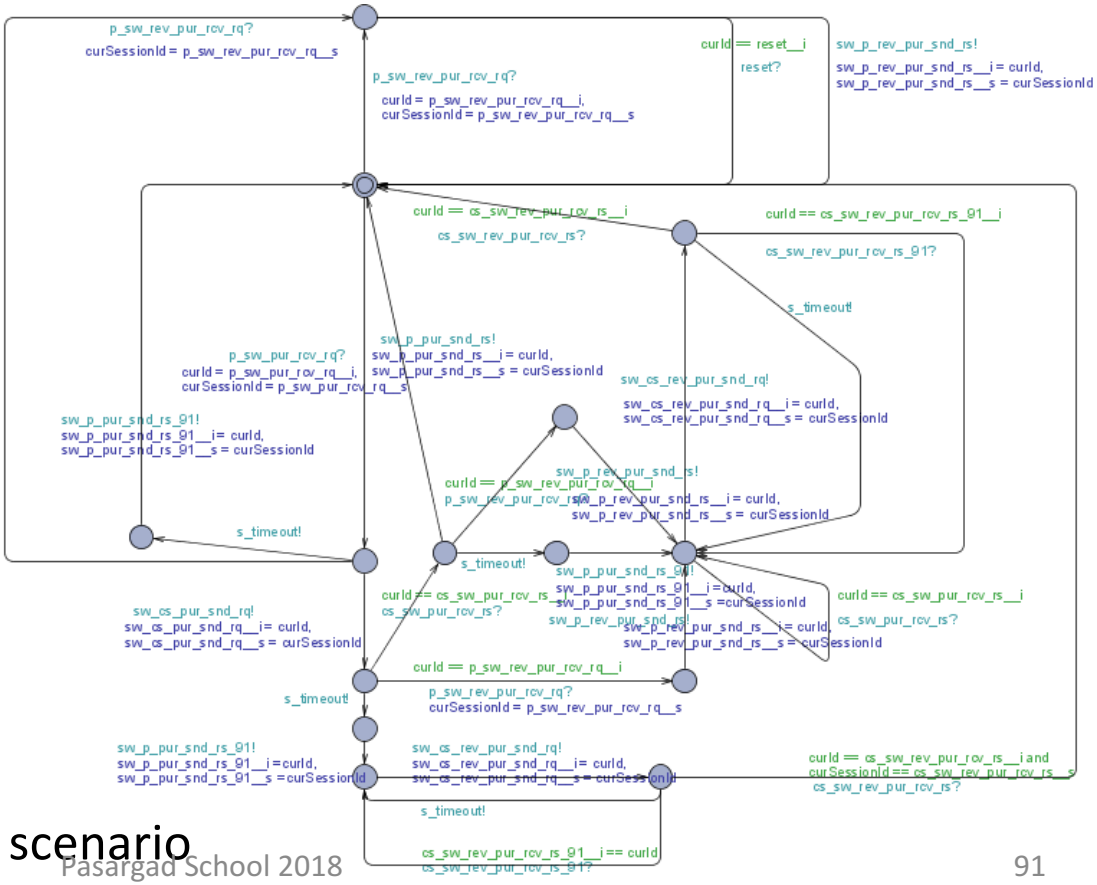
- Using Model-Based Testing

- Using a set of integrated tools
  - Generating and running test cases
  - Logging and prioritizing test cases
  - Measuring test coverage
  - Testing business rules

Back to the real world!

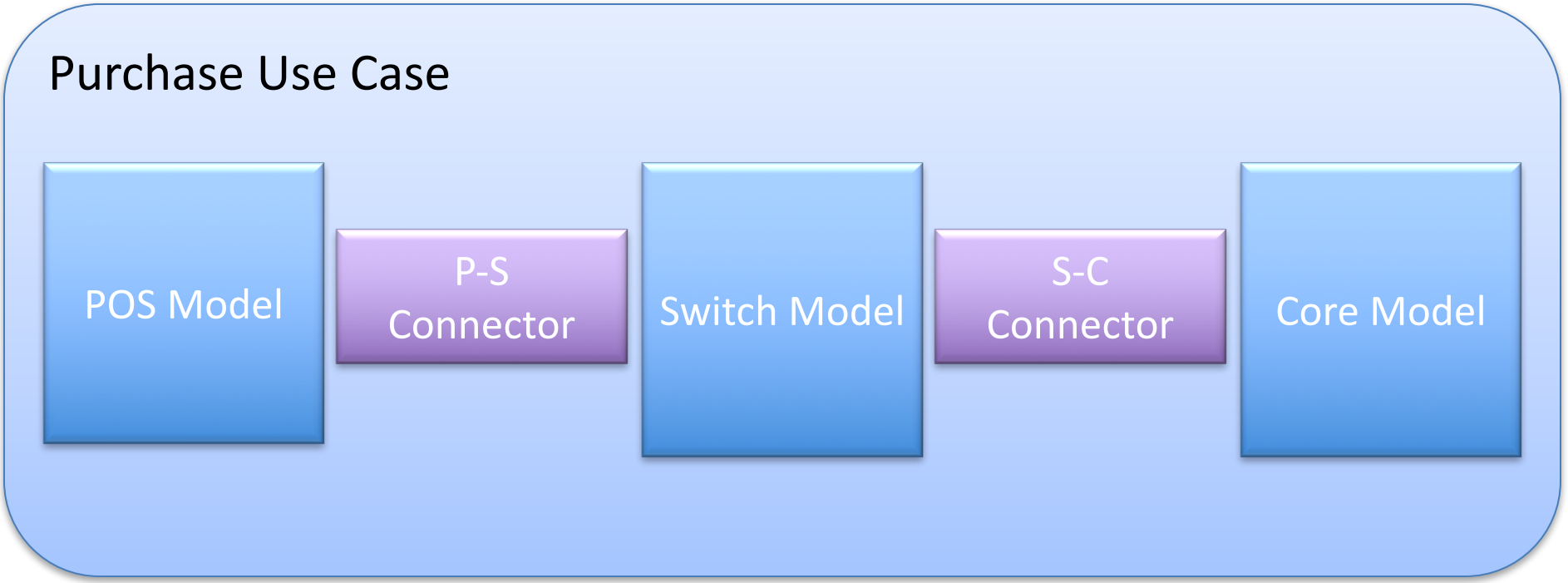
Switch Specification

- Modeling in UPPAAL
- ≈ 25 automata



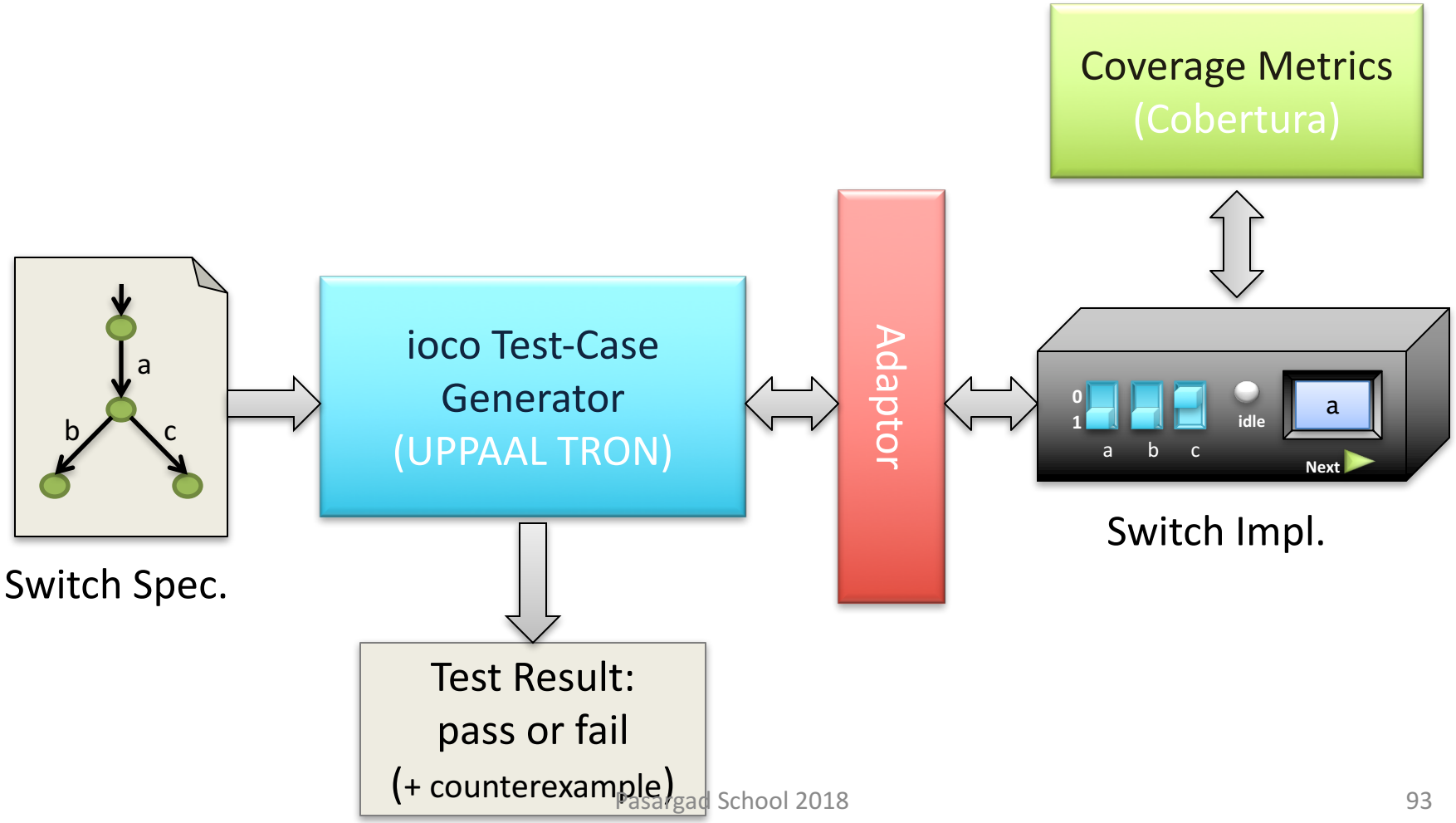
Switch behavior in purchase scenario

Specification Structure

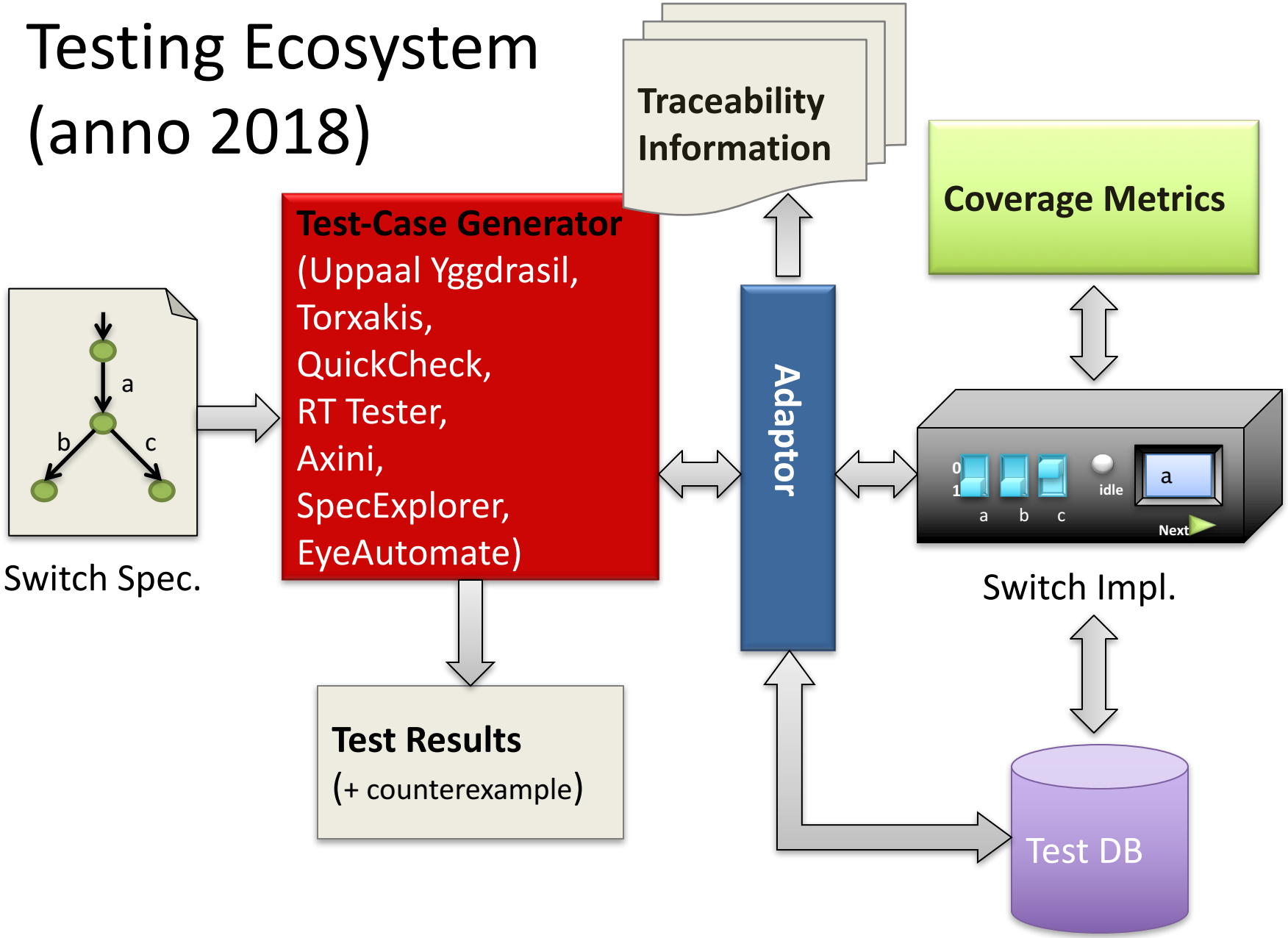


Four uses cases till now

Testing Ecosystem (anno 2008)

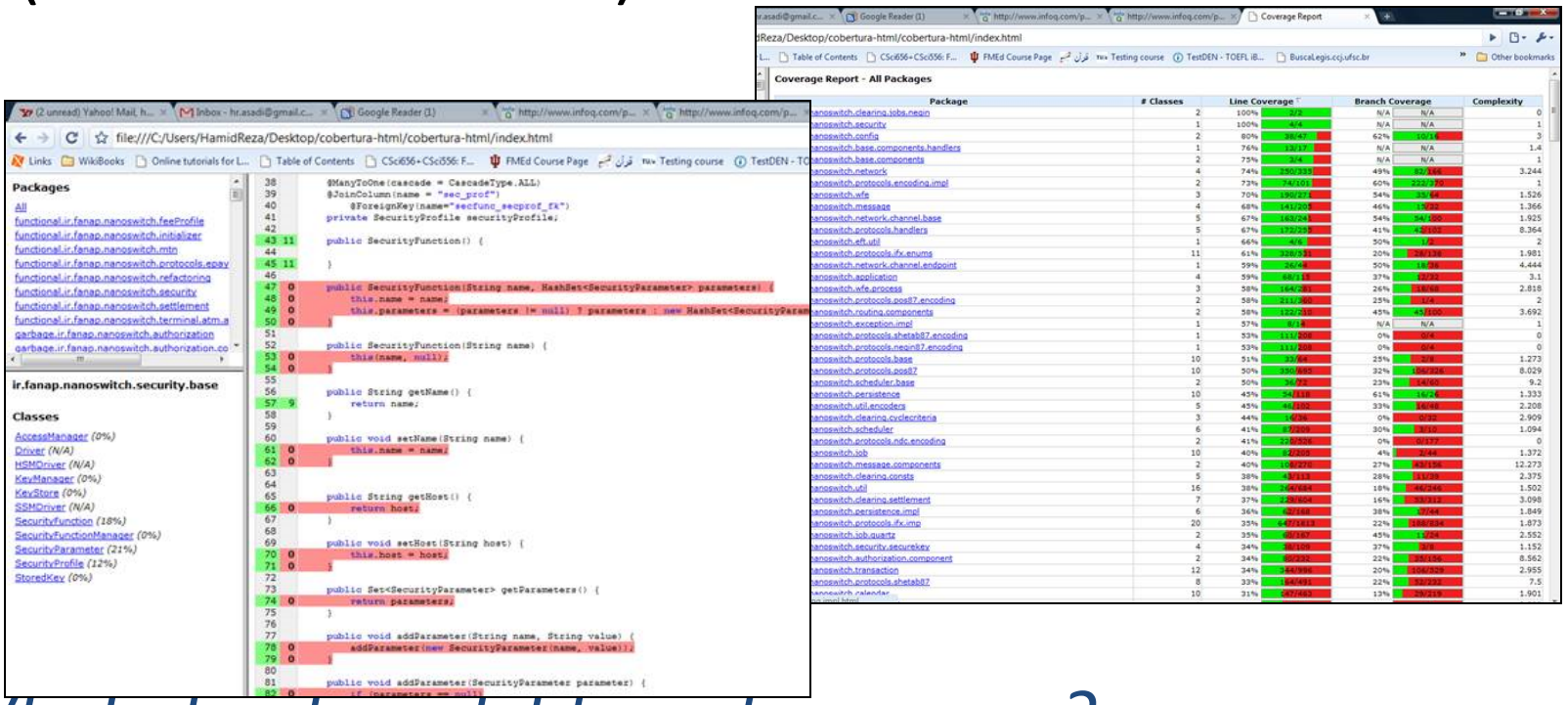


Testing Ecosystem (anno 2018)



Measuring Coverage

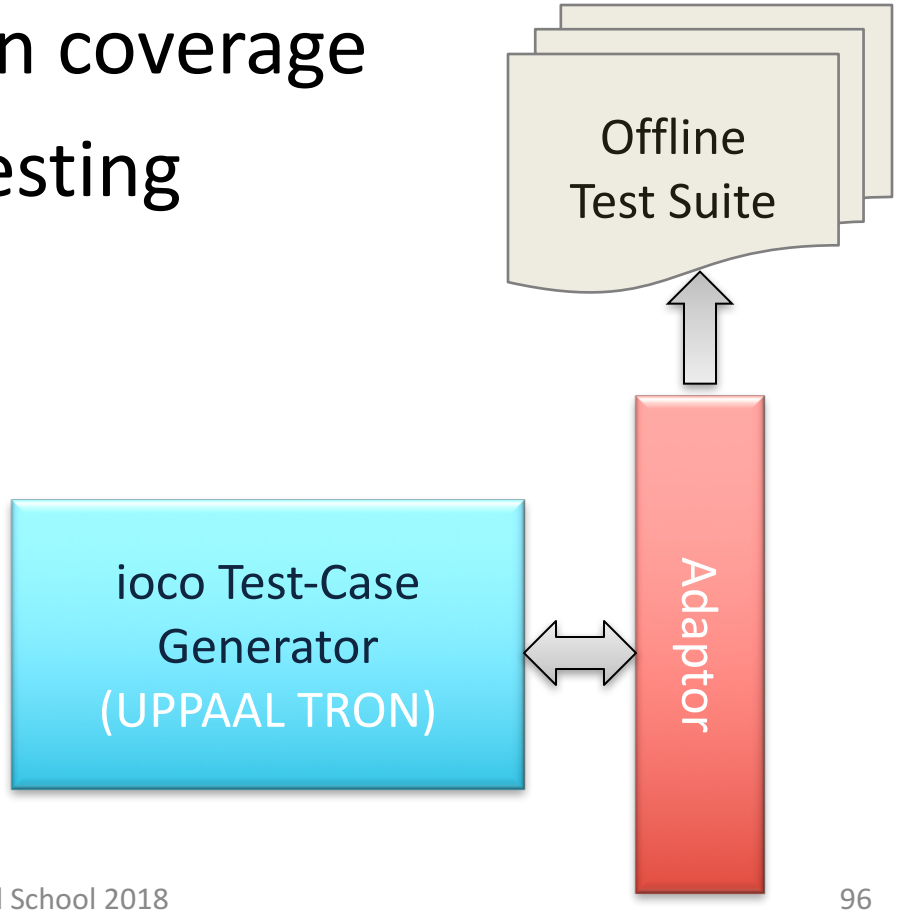
Using Cobertura code coverage tool
(now use EclEmma)



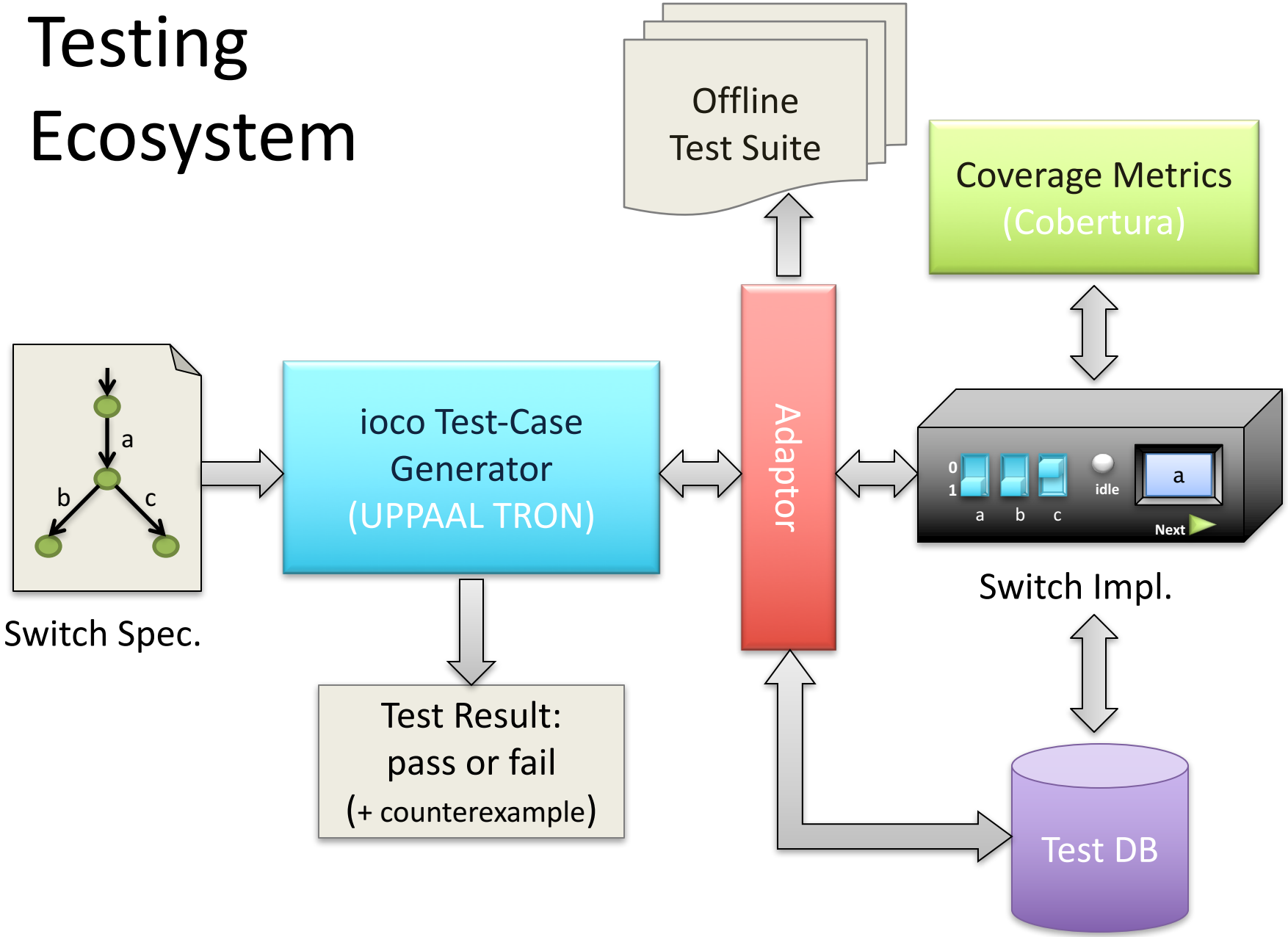
What about model-based coverage?

Logging Offline Test Suite

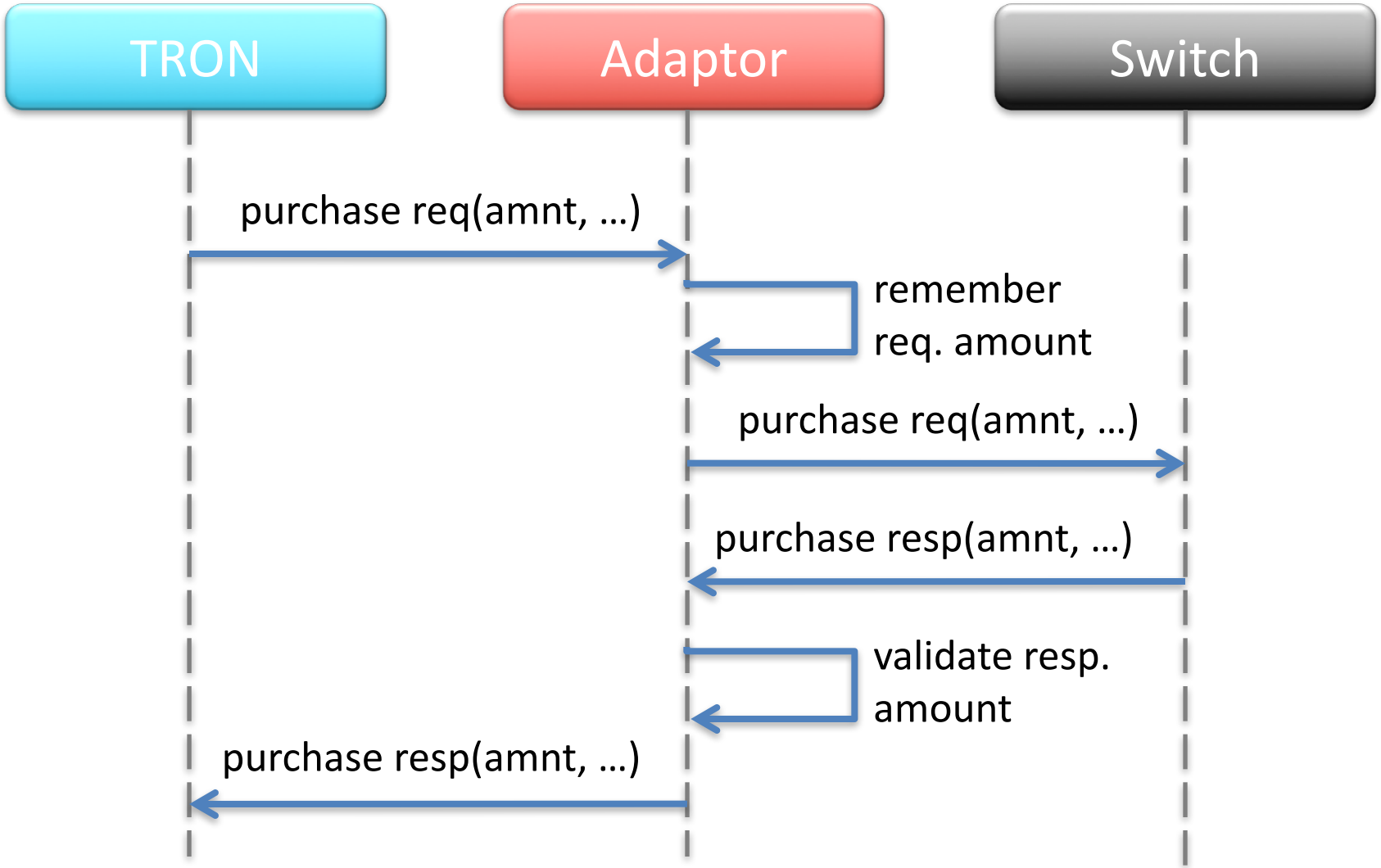
- The adaptor logs the messages from TRON
- Prioritization based on coverage
- Used for regression testing



Testing Ecosystem



Testing Business Rules



Results

- Have found a number of bugs in the operational system
  - One traced back to null-pointer dereferencing
  - Poor exception handling
- Code coverage of about 40%

Observations

- Modeling language and tool limitations
  - Component (de-compositional) testing
  - Asynchrony
  - Data specification and selection
  - Variability
- Scalability Issues
  - Running concurrent MBT instances
  - Reducing buffer lengths, pool sizes, etc.

Further Reading

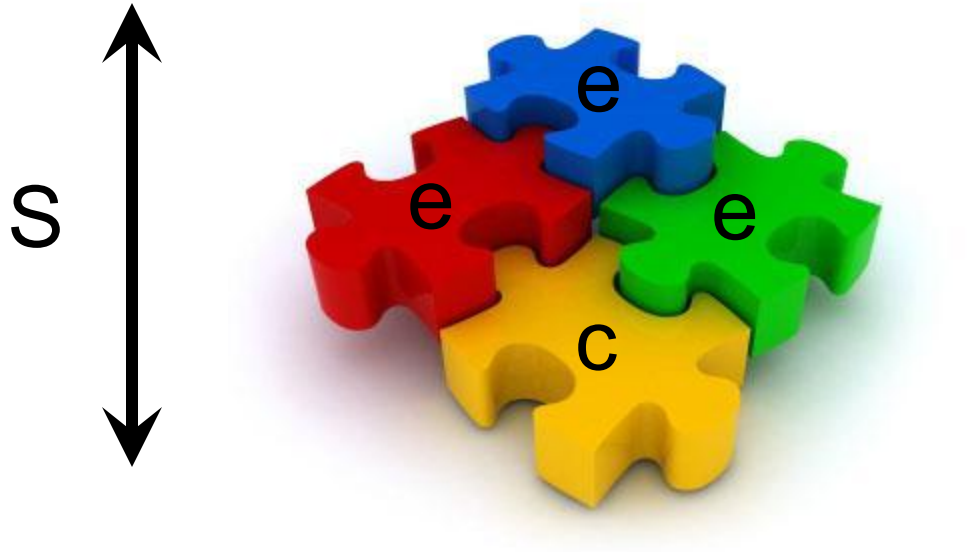
- Asaadi, Khosravi, MRM, and Noroozi. **Towards Model-Based Testing of Electronic Funds Transfer Systems**. Proc. of FSEN 2011. Models publicly available on Assembla.

- Vishal, Kovacioglu, Kherazi, and MRM. **Integrating Model-Based and Constraint-Based Testing Using SpecExplorer**. Proc. of MoTiP 2012.

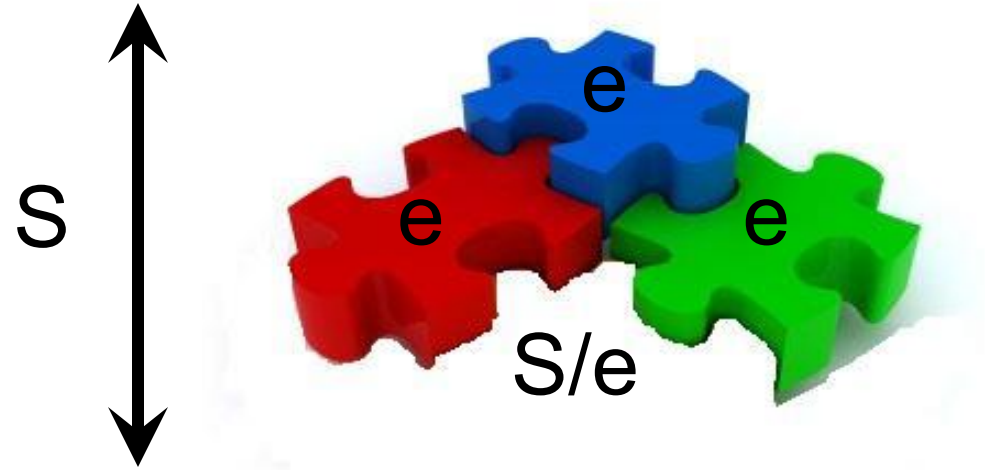
Back to Theory:

Component Testing

Decompositional Testing



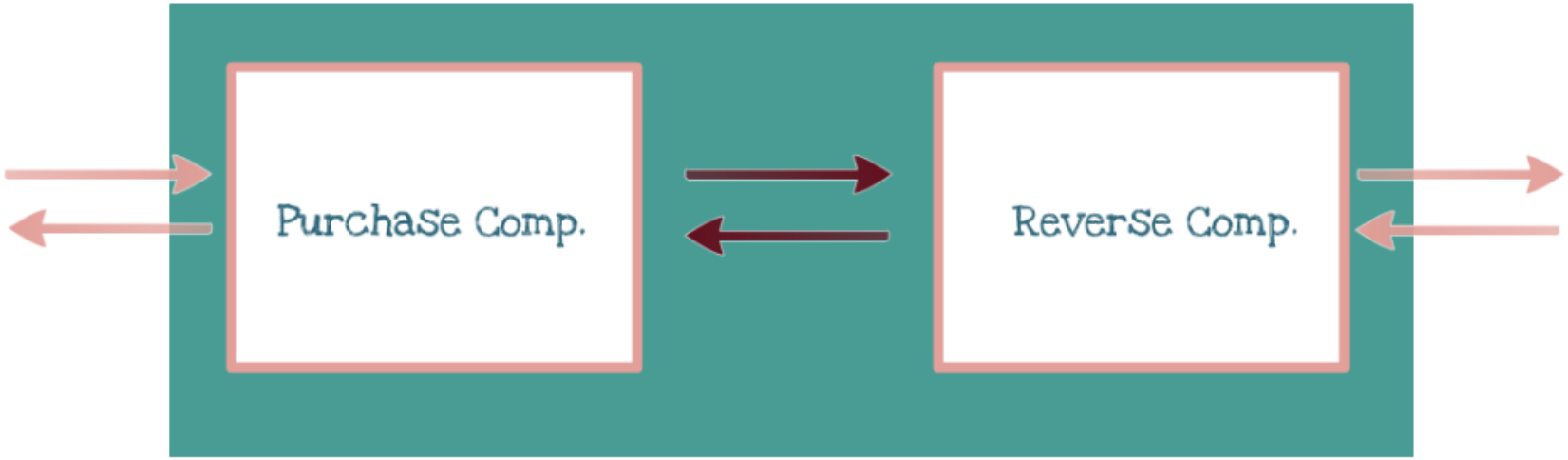
Decompositional Testing



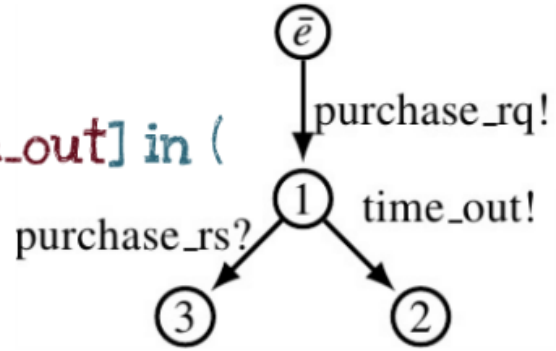
for all c, (c || e) **ioco** s iff c **ioco** S/e

Compositional Testing with ioco

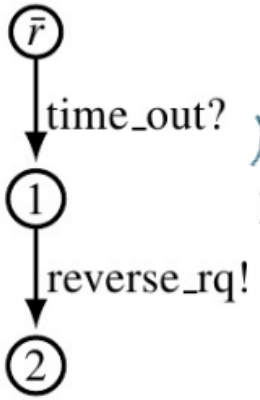
[M. van der Bijl, A. Rensink & J. Tretmans -2003]



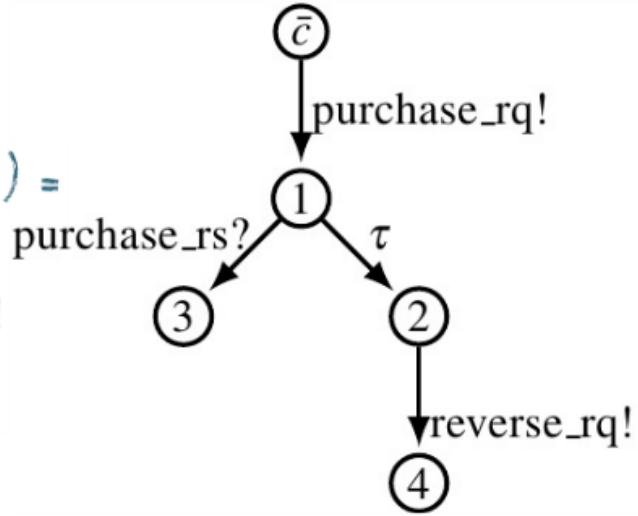
hide[time_out] in (



||



) =



Decompositional properties

1. Decomposability

$$\exists \bar{s}', \forall \bar{c}, \bar{e}.\ \ \bar{c}\ \mathbf{ioco}\ \bar{s}' \quad \Rightarrow \quad \bar{c} \| \bar{e}\ \mathbf{ioco}\ \bar{s}$$

2. Strong decomposability

$$\exists \bar{s}', \forall \bar{c}, \bar{e}.\ \ \bar{c}\ \mathbf{ioco}\ \bar{s}' \quad \Leftrightarrow \quad \bar{c} \| \bar{e}\ \mathbf{ioco}\ \bar{s}$$

Decompositional Testing

for all c, (c || e) **ioco** s iff c **ioco** S/e

Construction of S/e:

- Check:
  Can S be the composition of e with some c?
- Filter out the behavior of e in S

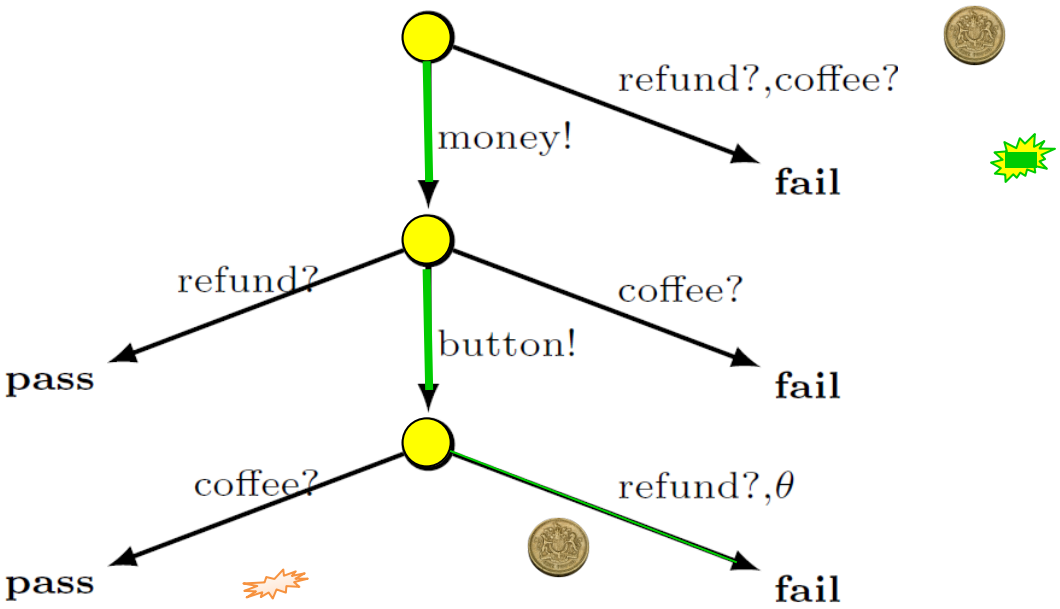
Further Reading

Noroozi, MRM, and Willemse. Decomposability in Input Output Conformance Testing. Proc. of MBT 2013.

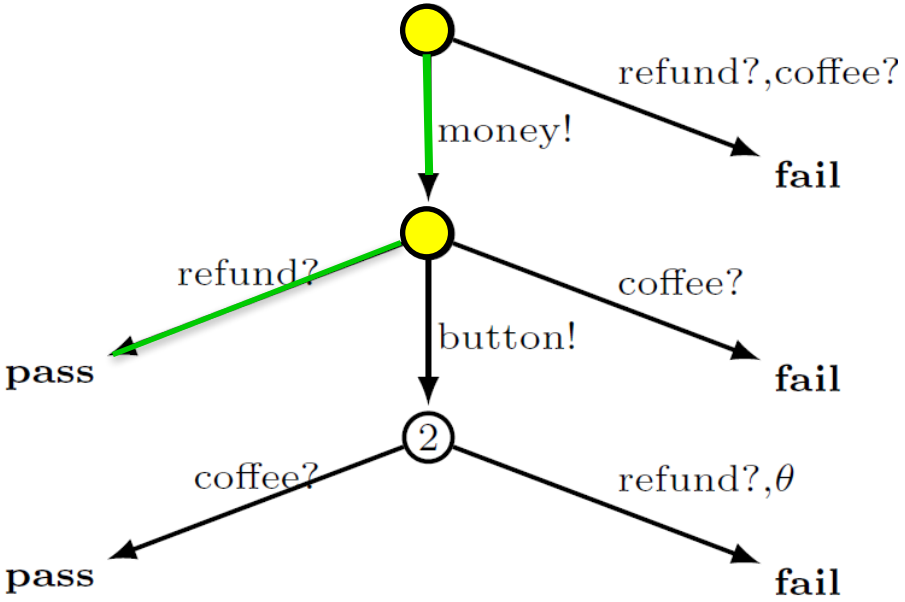
Back to Theory:

Asynchrony

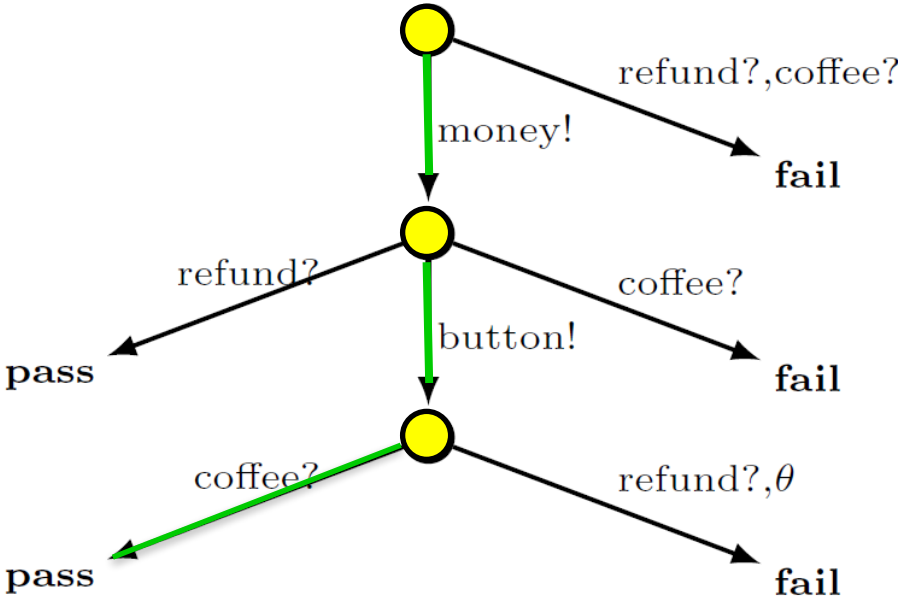
Asynchronous Communication



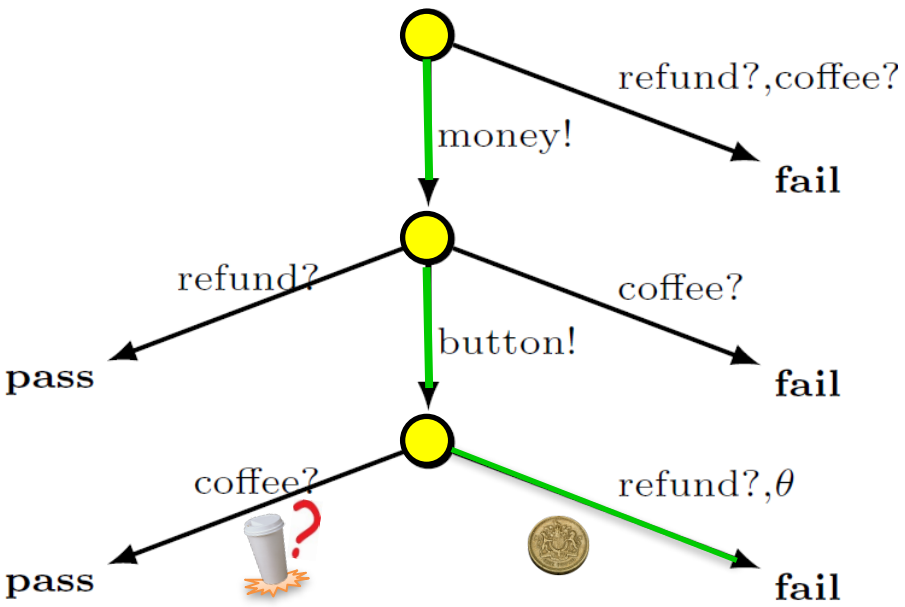
Synchronous Communication



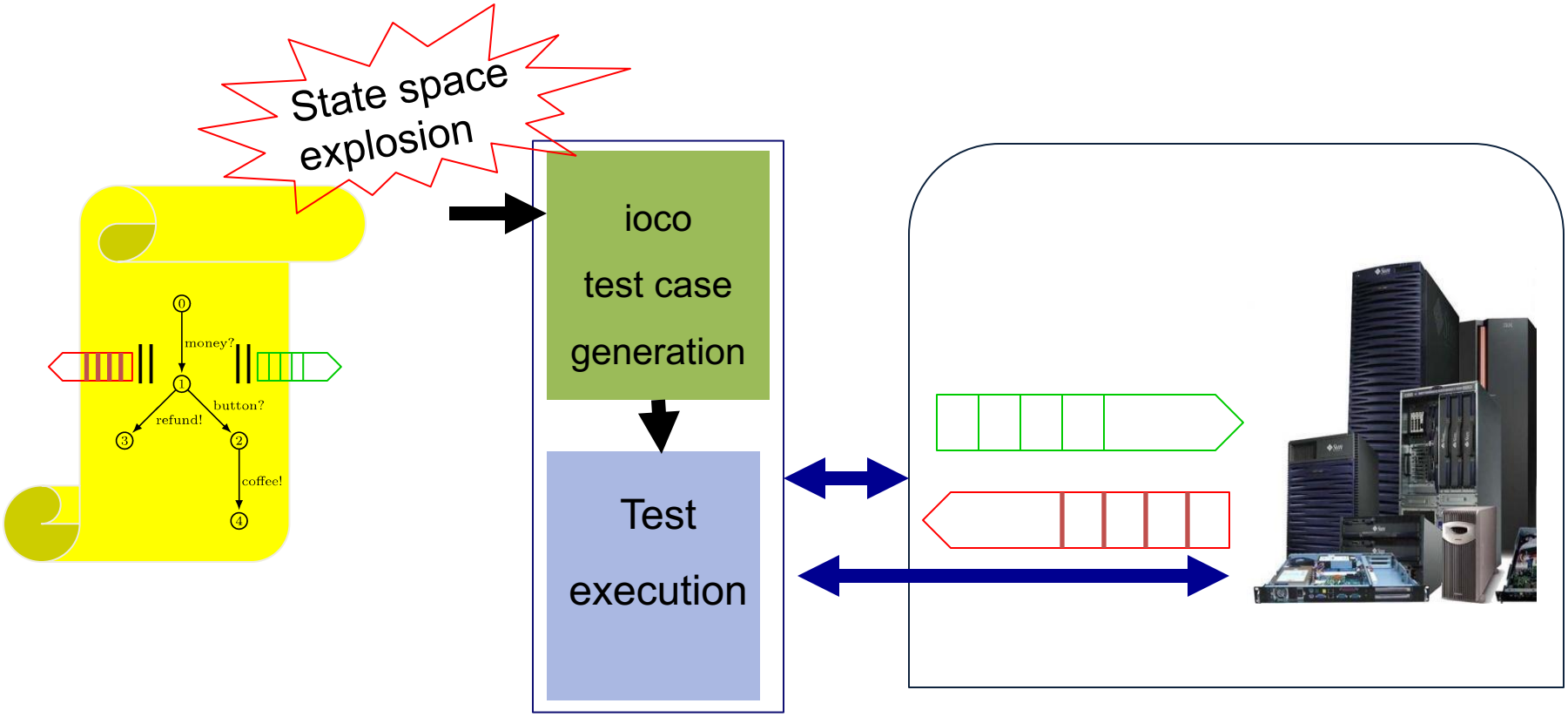
Synchronous Communication



Asynchronous Communication

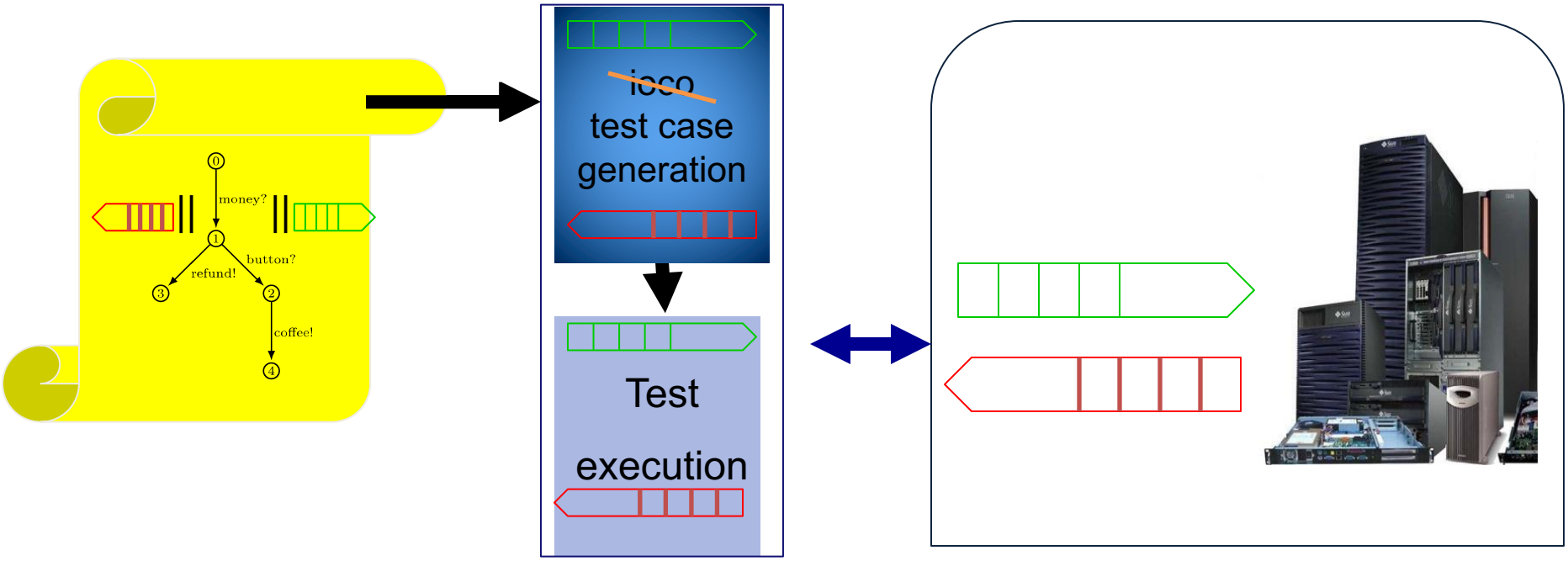


Prior Art [Tretmans&Verhaard'92]

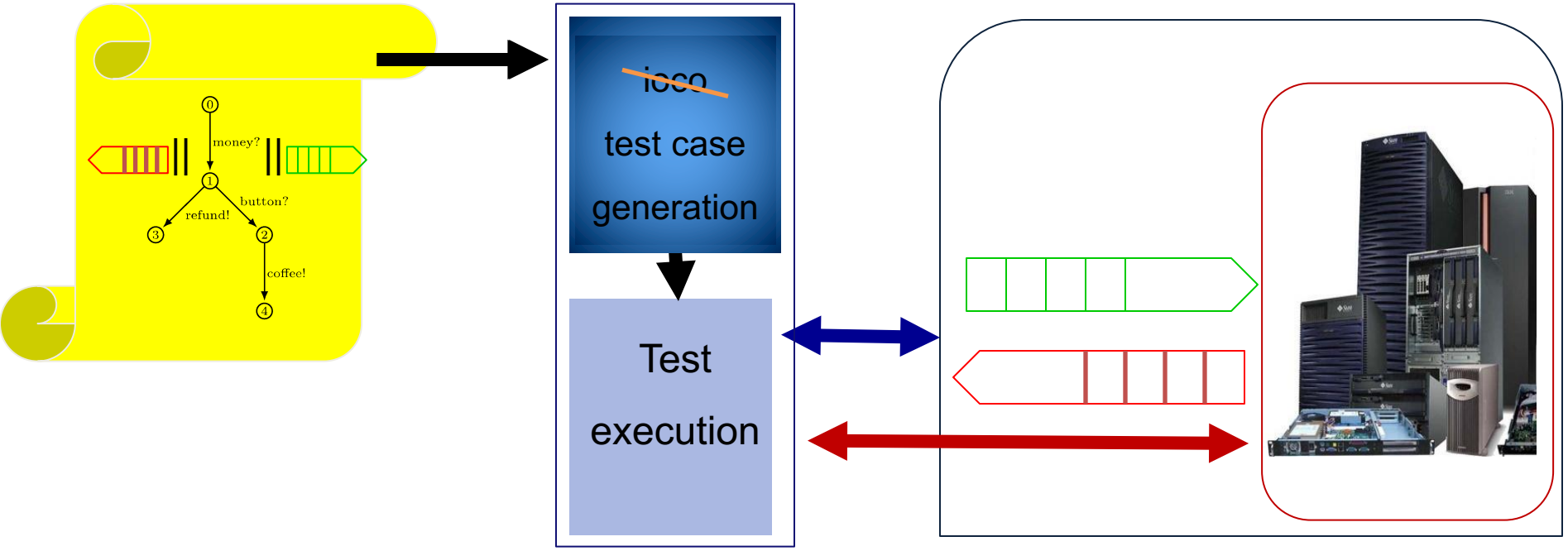


State of the Art

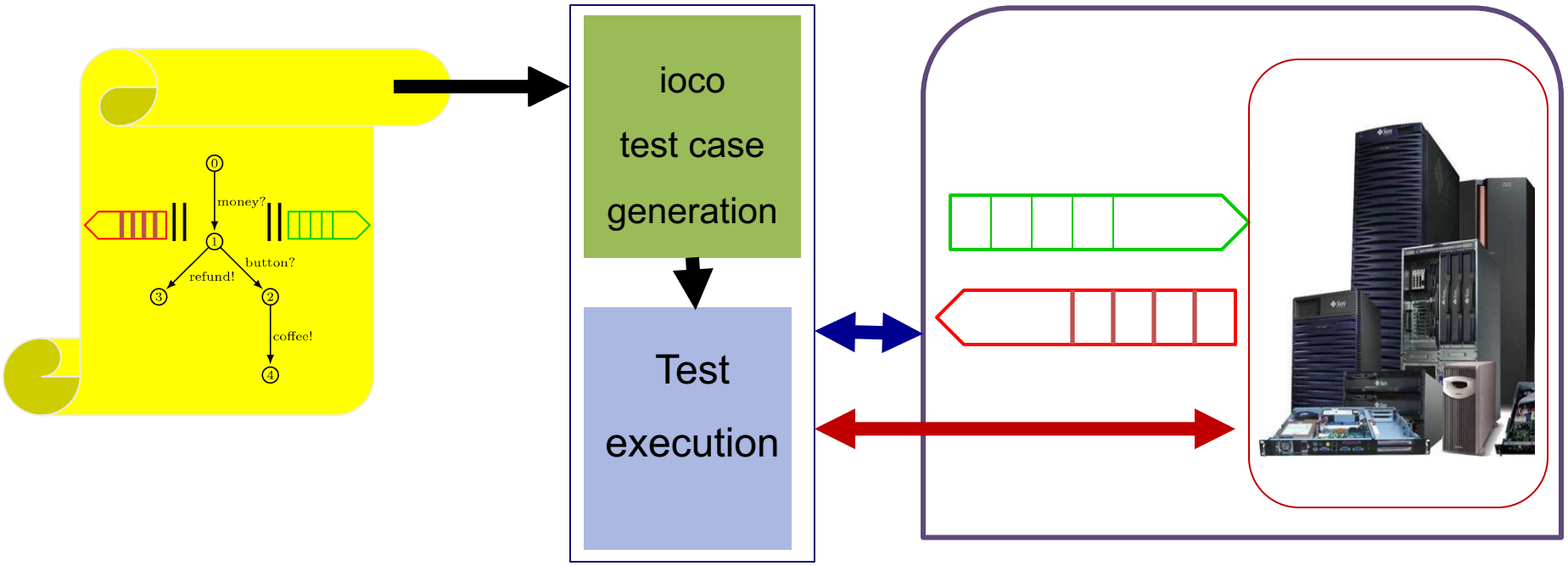
[Petrenko&Yevtushenko'02,03][Simao&Petrenko'10]



State of the Art [Weiglhofer&Wotawa'09]



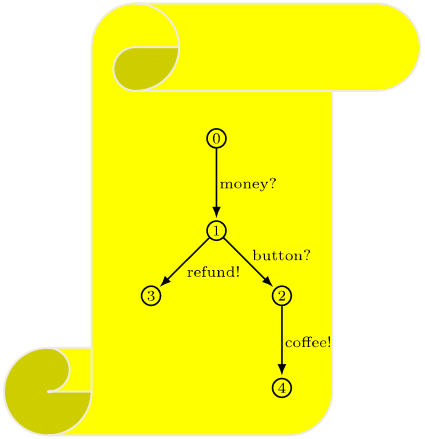
What we are after

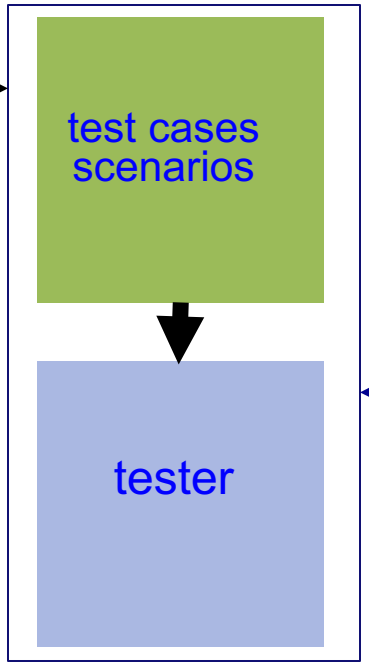


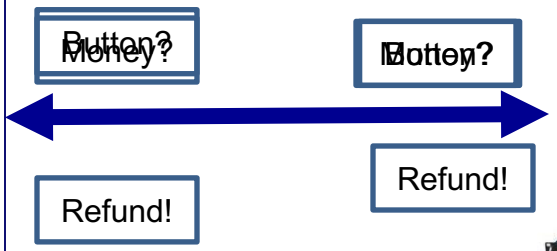
Delayed Traces

Trace:

money?	refund!	button?

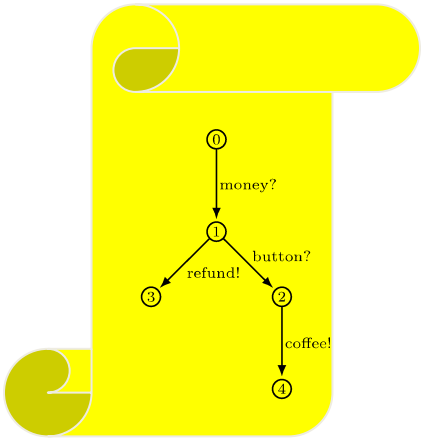


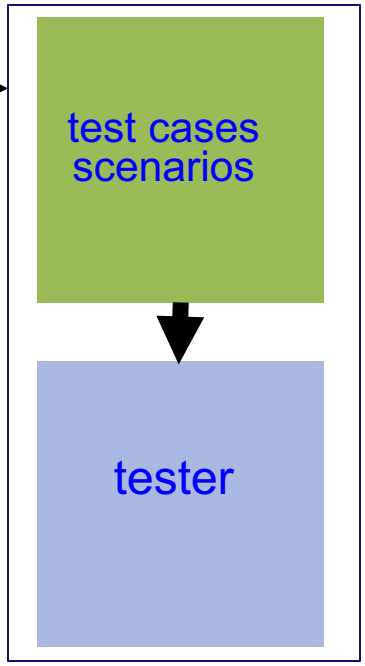


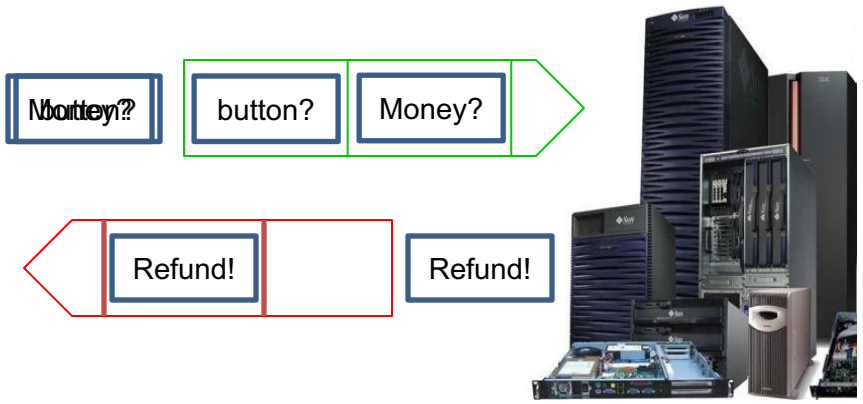


Delayed Traces

Trace: | money? | button? | refund! |

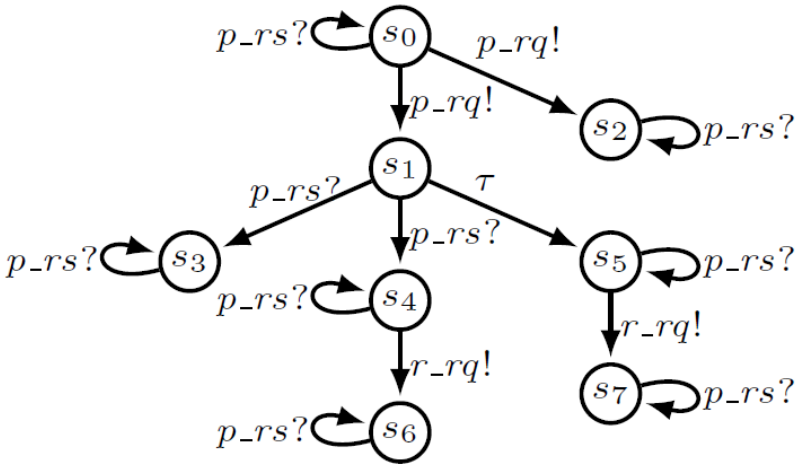






Delayed right-closed IOTS

Delay right-closed IOTS, *S* is an IOTS such that
$\forall \sigma \in$ Straces (S) then delayed trace of $\sigma \in$ Straces (S)



σ.x.a ∈ Straces (S) then σ.a.x ∈ Straces (S)

Theorems

Theorem

If implementation **i** is delay right-closed, then

i *ioco* **Spec** if and only if **Q(i)** *ioco* **Spec**

Theorem

If $\forall t \in$ TestCases (S), **i** *passes* **t** if and only if **Q(i)** *passes* **t**

then

implementation **i** is delay right-closed IOTS

Further Reading

Noroozi, Khosravi, MRM, and Willemse. Synchrony and Asynchrony in Conformance Testing. SoSym J., 2015.


B.A. APOLOGISE FOR ABNORMAL DELAYS - NORMAL DELAYS WILL BE RESUMED AS SOON AS POSSIBLE
CARTOONSTOCK .com
Search ID: gth0151

Thank You Very Much!

mm789@le.ac.uk