Model-based Testing

Gordon Fraser - Saarland University
Conformance Testing

Model

specifies

System

conforms to

Test Generation

Test Suite

Test Execution

Verdict
Soundness of Conformance Testing

Model

System

specifies

conforms to

Test Generation

Test Execution

Verdict

Test Generation specifies System, which conforms to Model. The system undergoes Test Execution, resulting in a Verdict.
A test system, which always says \textit{true} is sound.

A test system, which always says \textit{false} is complete.

We want test systems which are \textit{sound and complete}!
Soundness and Completeness of Conformance Testing

- Model
- System
- Test Generation
- Test Execution
- Verdict
Testing can never be sound and complete!

Edsger W. Dijkstra
Proving Soundness and Completeness

Model specifies System

System conforms to Test Generation

Model

Test Generation

Test Suite

Test Execution

Verdict
Proving Soundness and Completeness

Model

System

Test Generation

Test Suite

Test Execution

Verdict

Physical ingredients

specifies

conforms to
Proving Soundness and Completeness

- Model
  - specifies
  - conforms to
- System
- Physical ingredients
- Test Generation
- Test Suite
- Test Execution
- Verdict
- Observations
Executing a test case on the system yields a set of observations.

Every observation represents a part of the **implementation model** of the system, i.e. the model describing how the real system behaves.
The set of all observations made with all possible test cases represents the complete implementation model of the system!
Depending on the chosen class of implementation models, the observations might have to be transformed, first.
Assuming from now on the validity of the test hypothesis, we know that for every system there is a corresponding observational equivalent implementation model. This implementation model is unknown since in practice we cannot execute all possible test cases at the system. But since we know it exists, we can now define formally what conformance means!
Proving Soundness and Completeness
Proving Soundness and Completeness

Model

specifies

Implementation Model

Test Generation

imp

Test Suite

Formal Test Execution

Verdict

Observations
Now we can define:

System **conforms-to** the specification model

⇔

the implementation model is **imp-correct** to the specification model
Proving Soundness

Test Generation

Model

specifies

Implementation Model

formal

Test Execution

Verdict

Observations
The proof-obligation to show the soundness and completeness of a test generation algorithm w.r.t. an implementation relation \( \text{imp} \) is:

Show for all implementation models:

implementation model \( M \) is \textbf{imp-correct} to the specification model

\[ \iff \]

\( M \) passes all test cases which the algorithm can generate
Having done so, you have shown that:

System passes all test cases which the algorithm can generate

\[ \iff \]

System conforms-to the specification model
Summary

We want test generation algorithms to be sound and complete for the conforms-to relation.

Every system has an underlying implementation model consisting of all possible observations one can make with all possible test cases.

To restrict the class of systems, assumptions are made on the test execution.

Based on these assumptions, one has to prove that an implementation model exists which is observational equivalent to the system.
Now the implementation model can be substituted for the real system (aka the **test hypothesis**).

Between the implementation model and the specification model **implementation relations** can be defined.

**Conformity** of a system to a model is then defined by the **imp-correctness** of its underlying implementation model.

The main **proof obligation** is to show the soundness and completeness of the test generation algorithm w.r.t. the chosen implementation relation.
Finite State Machines

- Original domains:
  - sequential circuits
  - communication protocols

- Two types of **Finite State Machines (FSM)** matter for testing:
  - Mealy Machines
  - Moore Machines

- Commonly, FSM is identified with **Mealy Machine**.
Definition 1 (Mealy Machine). A Mealy Machine $M$ is a quintuple $M = (I, O, S, \delta, \lambda)$ where

- $I$ is a finite and nonempty set of inputs
- $O$ is a finite and nonempty set of outputs
- $S$ is a finite and nonempty set of states
- $\delta : S \times I \rightarrow S$ is the state transition function
- $\lambda : S \times I \rightarrow O$ is the output function
Alternating Bit Protocol

The Sender as a Mealy Machine:

User → Sender → Receiver

The Sender as a Mealy Machine:
Conformance

Specification models and implementation models are Mealy Machines.

What does conformance mean here?
We have $M_i \text{ imp } M_s \iff M_i$ is equivalent to $M_s$.

Two FSM are equivalent iff for every input sequence they produce the same output sequence.

\[ M_s(\text{bbb}) = 110 \neq M_{i1}(\text{bbb}) = 111 \]
A test case is an input sequence together with its output sequence, derived from the specification model.

Test case: input: bbb
output: 110
**Formal Test Execution**

- **Test case** is an input sequence together with its output sequence, derived from the specification model.
- **Formally executing** a test case means giving the input sequence to the implementation model.

Test case: input: `bbb`
output: `110`
A **test case** is an input sequence together with its output sequence, derived from the specification model. **Formally executing** a test case means giving the input sequence to the implementation model, and **observing** the corresponding output sequence.

**Test case:**
- Input: `bbb`
- Output: `110`

**Observation:**
- Input: `bbb`
- Output: `111`
A **test case** is an input sequence together with its output sequence, derived from the specification model.

Formally executing a test case means giving the input sequence to the implementation model, and **observing** the corresponding output sequence - leading to a **verdict**.

Test case: input: bbb
output: 110

Observation: input: bbb
output: 111

\[=\]

Mi1
But we don't know the implementation model a priori, we have just executed a single test case!

What has really happened, is this:

Test case: input: bbb
output: 110

Observation: input: bbb
output: 111
All we know is a little puzzle-piece from $M_{i1}$. 
But this is sufficient to observe non-conformity, since all possible completions of the Black Box are non-conforming!
A sound and **complete** test generation algorithm must generate **all possible test cases**.

### Test Cases

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>ab</td>
<td>01</td>
</tr>
<tr>
<td>aab</td>
<td>001</td>
</tr>
<tr>
<td>bbb</td>
<td>110</td>
</tr>
<tr>
<td>abababab</td>
<td>01110001</td>
</tr>
</tbody>
</table>
A sound and **complete** test generation algorithm must generate all possible test cases.

**Test Cases**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>ab</td>
<td>01</td>
</tr>
<tr>
<td>aab</td>
<td>001</td>
</tr>
<tr>
<td>bbb</td>
<td>110</td>
</tr>
<tr>
<td>abababab</td>
<td>01110001</td>
</tr>
</tbody>
</table>

*Testing can never be sound and complete!*
A sound and complete test generation algorithm must generate all possible test cases.
When should we stop testing?

Which test cases shall we select?

⇒ How to deal with the practical incompleteness of testing?

1) Accept it, and focus on **heuristics** like code coverage, model coverage, timing constraints, randomness, test purposes, etc.

2) Try to find further **assumptions**, which makes testing complete in practice, i.e., leading to a **finite** sound and complete test suite.
Dijkstra Revisited

**When** should we stop testing?

**Which** test cases shall we select?

⇒ How to deal with the practical incompleteness of testing?

1) Accept it, and focus on **heuristics** like code coverage, model coverage, timing constraints, randomness, test purposes, etc.

2) Try to find further **assumptions**, which makes testing complete in practice, i.e., leading to a **finite** sound and complete test suite.
Remember...

Dijkstra is right (of course).

He refers to the fact, that the number of test cases in a sound and complete test suite is usually infinite (or at least too big).

If that would not be the case, testing could prove the conformity of the system to the model (given some assumptions on the system).

2) Try to find further assumptions, which makes testing complete in practice, i.e., leading to a finite sound and complete test suite.
A checking sequence for $M_s$ is an input sequence that distinguishes the class of machines equivalent to $M_s$ from all other machines.

The length of this sequence can be used to compare the time complexity of the several algorithms.
Mandatory Assumptions

(1) $M_s$ is **minimized**, meaning that $M_s$ has no **equivalent states**. Equivalent states produce the same output sequence for every input sequence.
Mandatory Assumptions

(1) $M_s$ is **minimized**, meaning that $M_s$ has no equivalent states. Equivalent states produce the same output sequence for every input sequence.
(1) $M_s$ is **minimized**, meaning that $M_s$ has no equivalent states. Equivalent states produce the same output sequence for every input sequence.
(1) $M_s$ is minimized.

(2) $M_s$ is strongly connected, meaning every state can reach every other state.
Mandatory Assumptions

(1) \( M_s \) is minimized.
(2) \( M_s \) is strongly connected.
(3) \( M_i \) has the same inputs and outputs as \( M_s \), and does not change during runtime.
Mandatory Assumptions

1. $M_s$ is minimized.
2. $M_s$ is strongly connected.
3. $M_i$ has the same inputs and outputs as $M_s$, and does not change during runtime.
Mandatory and Additional Assumptions

(1) $M_s$ is minimized.
(2) $M_s$ is strongly connected.
(3) $M_i$ has the same inputs and outputs as $M_s$, and does not change during runtime.
(4) $M_s$ and $M_i$ have an initial state.
Mandatory and Additional Assumptions

(1) $M_s$ is **minimized**.

(2) $M_s$ is **strongly connected**.

(3) $M_i$ has the **same inputs and outputs** as $M_s$, and **does not change** during runtime.

(4) $M_s$ and $M_i$ have an **initial state**.

(5) $M_s$ and $M_i$ have a **reset message** that from any state of the machine causes a transition which ends in the initial state, and produces no output.
(1) $M_s$ is **minimized**.

(2) $M_s$ is **strongly connected**.

(3) $M_i$ has the **same inputs and outputs** as $M_s$, and **does not change** during runtime.

(4) $M_s$ and $M_i$ have an **initial state**.

(5) $M_s$ and $M_i$ have a **reset message**.

Having a reset message, $M_s$ is strongly connected!
Mandatory and Additional Assumptions

(1) $M_s$ is minimized.
(2) $M_s$ is strongly connected.
(3) $M_i$ has the same inputs and outputs as $M_s$, and does not change during runtime.
(4) $M_s$ and $M_i$ have an initial state.
(5) $M_s$ and $M_i$ have a reset message.
(6) $M_i$ has the same number of states than $M_s$. 
Mandatory and Additional Assumptions

(1) $M_s$ is minimized.
(2) $M_s$ is strongly connected.
(3) $M_i$ has the same inputs and outputs as $M_s$, and does not change during runtime.
(4) $M_s$ and $M_i$ have an initial state.
(5) $M_s$ and $M_i$ have a reset message.
(6) $M_i$ has the same number of states than $M_s$. Under this assumption, two types of faults can be present in $M_i$:

Output faults: a transition produces a wrong output

Transfer faults: a transition goes to a wrong state
Output- and Transfer Faults

Output fault

Transfer and Output faults

$s_1$, $s_2$, $s_3$, and $i_1$, $i_2$, $i_3$ are nodes in the diagram. The arrows represent transitions between states and inputs, with labels indicating the input/output values.

$M_s$, $M_i$, $M_{i1}$, and $M_{i2}$ are labeled in the diagram, indicating different states or inputs.

The diagram illustrates the flow of signals through the system, highlighting the output fault and transfer output faults.
Mandatory and Additional Assumptions

(1) $M_s$ is minimized.
(2) $M_s$ is strongly connected.
(3) $M_i$ has the same inputs and outputs as $M_s$, and does not change during runtime.
(4) $M_s$ and $M_i$ have an initial state.
(5) $M_s$ and $M_i$ have a reset message.
(6) $M_i$ has the same number of states than $M_s$. 
(1) $M_s$ is minimized.
(2) $M_s$ is strongly connected.
(3) $M_i$ has the same inputs and outputs as $M_s$, and does not change during runtime.
(4) $M_s$ and $M_i$ have an initial state.
(5) $M_s$ and $M_i$ have a reset message.
(6) $M_i$ has the same number of states than $M_s$.
(7) $M_s$ and $M_i$ have a status message. Giving a particular input “status”, the output uniquely defines the current state.
Mandatory and Additional Assumptions

(1) $M_S$ is minimized.
(2) $M_S$ is strongly connected.
(3) $M_i$ has the same inputs and outputs as $M_S$, and does not change during runtime.
(4) $M_S$ and $M_i$ have an initial state.
(5) $M_S$ and $M_i$ have a reset message.
(6) $M_i$ has the same number of states than $M_S$.
(7) $M_S$ and $M_i$ have a status message.
(8) $M_S$ and $M_i$ have a set message. From the initial state the system can be transferred to every other state $s$ by giving the input set(s). No output is produced while doing so.
Mandatory and Additional Assumptions

1. $M_s$ is minimized.
2. $M_s$ is strongly connected.
3. $M_i$ has the same inputs and outputs as $M_s$, and does not change during runtime.
4. $M_s$ and $M_i$ have an initial state.
5. $M_s$ and $M_i$ have a reset message.
6. $M_i$ has the same number of states than $M_s$.
7. $M_s$ and $M_i$ have a status message.
8. $M_s$ and $M_i$ have a set message.

Additional
For all states $s$ and all inputs $a$ do:
1. Apply the reset message to bring $M_i$ to the initial state.
2. Apply a set message to transfer $M_i$ to state $s$.
3. Apply the input $a$.
4. Verify that the output conforms to the specification $M_s$.
5. Apply the status message and verify that the final state conforms to the specification $M_s$.

This algorithm is sound and complete given that all assumptions (1) – (8) hold.

The length of the checking sequence is $4 \times |I| \times |S|$
To get rid of the *set message*, and possibly shorten the test suite, we can build a sequence that visits every state and every transition at least once – a *transition tour*.

The shortest transition tour visits each transition exactly once, and is called an *Euler tour*. It only exists for *symmetric* FSM (every state is the start state and end state of the same number of transitions).

An Euler Tour can be computed in linear time w.r.t. the number of transitions.

In non-symmetric FSM finding the shortest tour is referred to as the *Chinese Postman Problem*. It can be solved in polynomial time.
**Problem:**
Covering all transitions of $M_s$, and checking whether $M_i$ produces the same output, is not complete!
Problem:
Covering all transitions of $M_s$, and checking whether $M_i$ produces the same output, is not complete!

$M_s$

$ababab$ is an Euler tour
**Problem:**

Covering all transitions of $M_s$, and checking whether $M_i$ produces the same output, is not complete!

The Euler tour is sufficient to spot the output fault of $M_{i1}$: $011100 \neq 011101$

$M_s$

ababab is an Euler tour

$M_{i1}$

$M_{i2}$
Problem:
Covering all transitions of $M_s$, and checking whether $M_i$ produces the same output, is not complete!

The Euler tour is sufficient to spot the output fault of $M_{i_1}$: $011100 \neq 011101$

The Euler tour is not sufficient to spot the faults of $M_{i_2}$: $011100 = 011100$

$ababab$ is an Euler tour
State Identification and Verification

Solution 1:
Use the **status message** to verify the states while doing the transition tour.

Solution 2:
If the status message does not exists, use **separating sequences** instead. Examples are:
- Characterizing set (W Method)
- Identification set (Wp Method)
- UIO sequence (UIO Methods)
- Distinguishing sequence (Distinguishing Sequence Method)
- ...

Not all separating sequences are guaranteed to exists.
Not all of these methods are complete.
When even a reset message is not available, more can be done...

...transfer sequences...

...using distinguishing sequences without reset...

...using identifying sequences instead of distinguishing sequences...

...adaptive distinguishing sequences...

...homing sequence...
The General Procedure

Every method follows the same scheme:

For all states \( s \) and all inputs \( a \) do:
1. Bring \( M_i \) to the state \( s \).
2. Apply the input \( a \).
4. Verify that the output conforms to the specification \( M_s \).
5. Verify that the final state conforms to the specification \( M_s \).
The test hypothesis for FSM-based testing makes some general assumptions regarding the system to be tested:

- The system has finite state.
- The system is deterministic.
- The system communicates in a synchronous manner (input / output).

FSM-based testing focused on testing for equivalence.

Based on a given set of further mandatory and additional assumptions, the FSM algorithms can give a finite sound and complete test suite.

In other words, these algorithm can prove the equivalence.

Most of the theoretical problems have been solved.
Summary

- FSM-based testing can be the underlying testing model of several other formalisms, like UML state machines, Abstract State Machines, RPC-like systems, etc.

- Tools related to FSM-based testing are for instance:
  - Conformance Kit, PHACT, TVEDA, Autofocus, AsmL Test Tool,...

- Results regarding other types of state machines have shown that there is no hope that feasible algorithms can yield a finite sound and complete test suite, for instance:
  - Nondeterministic machines
  - Probabilistic machines
  - Symbolic machines
  - Real-Time machines
  - Hybrid machines
Original domains:
- sequential and concurrent programs
- hardware circuits

Several formalisms have an underlying Labeled Transition System (LTS) semantics, for instance:
- Statecharts
- Process Algebras
Models: Labelled Transition Systems

Labelled Transition System: \(<S, L_I, L_U, T, s_0>\)

- **states**
- **input actions**
- **output actions**
- **initial state**
- **transitions**

? = input
! = output
Observable Behaviour

“Some systems are more equal than others”
Relating two LTS can be done in a variety of manners, e.g.:

**Equivalence relations:**
- Isomorphism, Bisimulation, Trace Equivalence, Testing Equivalence, Refusal Equivalence, ...

**Preorder relations:**
- Observation Preorder, Trace Preorder, Testing Preorder, Refusal Preorder, ...

**Input-Output relations:**
- Input-Output Testing
- Input-Output Refusal
- `ioconf`
- `ioco`
- ...

...
An implementation relation is called **stronger** than another, if the classes of related LTS are more differentiated.

Implementation relations may also be incomparable.

We want an implementation relation to

- relate systems we **naturally consider** as being conforming
- be **applicable** in practice, i.e., having a feasible testing scenario
- be **as strong as possible**
Isomorphism

Two LTS are **isomorph** (or: equivalent) iff they are exactly the same modulo state names.

Isomorphism is the strongest notion of conformance.

Isomorphism is *not suited for testing* since we cannot observe the unobservable \( \tau \) action!
Two LTS are (weak) **bisimular** iff they simulate each other and go to states from where they can simulate each other again.

![Diagram of bisimulation](attachment:image.png)

**Bisimulation** is **not suited for testing** since its testing scenario comprises means which are infeasible in practice.
A trace is an observable sequence of actions.

Two LTS are trace equivalent iff they have the same traces.

Trace equivalence is the weakest notion of conformance.
For testing purposes it is usually considered too weak.

isomorphism \(\approx\) bisimilarity \(\approx\) trace equivalence
A completed trace is a trace leading to a state refusing all actions – a final state.

Two LTS are completed trace equivalent iff they are trace equivalent, and also share the same completed traces.

Here we need to be able to observe the absence of all actions, i.e., deadlocks.
Testing equivalence is stronger than completed trace equivalence, and demands a test scenario which can observe the refusal of actions.

conf is a modification of testing equivalence restricting the observations to only those traces contained in the specification (conf is not transitive).

Refusal equivalence is stronger than testing equivalence, and demands a test scenario which can continue the test after observing the refusal of actions.

Tool: Cooper for the Co-Op method for conf
Preorder Relations

\( \text{i imp s} \) means that implementation model \( i \) implements specification model \( s \).

Do we want \( \text{imp} \) to be

- reflexive \( s \text{ imp s} \)
  - ✔
- symmetric \( i \text{ imp s} \iff s \text{ imp i} \)
  - ⬤
- transitive \( i \text{ imp s} \land s \text{ imp t} \iff i \text{ imp t} \)
  - ✔
- anti-symmetric \( i \text{ imp s} \land s \text{ imp i} \iff i = s \)
  - ⬤
- total \( i \text{ imp s} \lor s \text{ imp i} \)
  - ⬤
- congruent \( i \text{ imp s} \lor f(i) \text{ imp f(s)} \)
  - ✔

An equivalence is reflexive, symmetric and transitive.

A preorder is just reflexive and transitive.
The motivation for preorder relations is to simplify the testing scenario. For almost every equivalence $\approx$ a corresponding preorder $\leq$ can be defined such that

$$p \approx q \iff p \leq q \land q \leq p$$

**Trace preorder:**

$$i \leq_{tr} s \iff \text{traces}(i) \subseteq \text{traces}(s)$$

In the same way **testing preorder** $\leq_{te}$ and **refusal preorder** $\leq_{rf}$ can be defined.
In Input-Output relations, the set of action labels is partitioned into *input actions* and *output actions*, leading to an *Input-Output Labeled Transition System (IOLTS)*.

Compared to FSM, IOLTS differ in
- having *asynchronous* transitions (either input or output)
- having potentially an *infinite* number of states
- being potentially *nondeterministic*
- being *not* necessarily *completely specified* for all inputs
- being *compositional*

An IOLTS, which is completely specified for all inputs is called an *input enabled IOLTS*. 
Specifications models are IOLTS
Implementation models are input-enabled IOLTS
What does ioco-conformance mean?

IOLTS

input enabled IOLTS
**Definition 3 (Quiescence).** A state $s \in S$ in $L$ is quiescent, denoted $\delta(s)$, iff $\forall \mu \in \Sigma_U \cup \{\tau\} : s \not\xrightarrow{\mu}$. Let $\delta$ be a constant not part of any action label set; $\Sigma_\delta$ abbreviates $\Sigma_I \cup \Sigma_U \cup \{\delta\}$.
Definition 4 (after). Let $s \in S$, $C \subseteq S$ and $\sigma \in \Sigma^*_\delta$. We define

$$s \text{ after } \sigma = _{\text{def}} \{ s' \in S \mid s \xrightarrow{\sigma}_{\delta} s' \}$$

- $s1$ after $\delta = \{s1\}$
- $s1$ after $?a = \{s2\}$
- $s2$ after $!x?b = \{s4,s5,s6\}$
- $s2$ after $!x\delta?b = \{s4,s5,s6\}$
- $s1$ after $?a!x\delta\delta?b!z\delta = \{s1\}$
Definition 5 (out). Let $s \in S$ and $C \subseteq S$. We define

- $out(s) = \begin{cases} \{\delta\} & \text{if } \delta(s) \\ \{a \in \Sigma_U \mid s \xrightarrow{a}\} & \text{otherwise} \end{cases}$
- $out(C) = \bigcup_{s \in C} out(s)$

<table>
<thead>
<tr>
<th>State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>${\delta}$</td>
</tr>
<tr>
<td>s2</td>
<td>${!x}$</td>
</tr>
<tr>
<td>s3</td>
<td>${\delta}$</td>
</tr>
<tr>
<td>s4</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>s5</td>
<td>${!y}$</td>
</tr>
<tr>
<td>${s1, s2}$</td>
<td>${\delta, !x}$</td>
</tr>
<tr>
<td>out(s1 after $\delta ?a$)</td>
<td>${!x}$</td>
</tr>
<tr>
<td>out(s1 after $?a !x ?b$)</td>
<td>${!y, !z}$</td>
</tr>
</tbody>
</table>
**Definition 6.** Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$ is $\text{ioco}_\mathcal{F}$-conform to $S$, denoted by $\mathcal{P} \text{ ioco}_\mathcal{F} S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$
Definition 6. Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $F \subseteq \Sigma^*_\delta$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_\mathcal{P} \rangle$ is $ioco_F$-conform to $S$, denoted by $\mathcal{P} \ ioco_F \ S$, iff

$$\forall \sigma \in F : \text{out}(p_1 \ \text{after} \ \sigma) \subseteq \text{out}(s_1 \ \text{after} \ \sigma)$$

Definition 7 (Suspension Traces). The set of suspension traces is defined as

$$\text{Straces}(s) \stackrel{\text{def}}{=} \{ \sigma \in \Sigma^*_\delta \mid s \overrightarrow{\sigma} \}$$

Just writing $ioco$ abbreviates $ioco_F$ with $F = \text{Straces}(s_0)$. 
Definition 6. Let \( S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle \) be an IOLTS, and let \( \mathcal{F} \subseteq \Sigma_\delta^* \). An input-enabled IOLTS \( \mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_\mathcal{P} \rangle \) is io\text{co}_\mathcal{F}-conform to \( S \), denoted by \( \mathcal{P} \text{ io\text{co}}_\mathcal{F} S \), iff

\[
\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)
\]

**Intuition:**

i io\text{co} s, iff

- if \( i \) produces output \( x \) after trace \( \sigma \), then \( s \) can produce \( x \) after trace \( \sigma \).

- if \( i \) cannot produce any output after trace \( \sigma \), then \( s \) cannot produce any output after trace \( \sigma \) (quiescence).
**ioco**

**Definition 6.** Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma^*_\delta$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$ is $\text{ioco}_\mathcal{F}$-conform to $S$, denoted by $\mathcal{P} \text{ ioco}_\mathcal{F} S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$
**Definition 6.** Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_\delta^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$ is $\text{ioco}_\mathcal{F}$-conform to $S$, denoted by $\mathcal{P} \text{ioco}_\mathcal{F} S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$
**Definition 6.** Let \( S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle \) be an IOLTS, and let \( \mathcal{F} \subseteq \Sigma^*_\delta \). An input-enabled IOLTS \( \mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_P \rangle \) is \textit{ioco}_\mathcal{F}-conform to \( S \), denoted by \( \mathcal{P} \textit{ioco}_\mathcal{F} S \), iff

\[
\forall \sigma \in \mathcal{F} : \text{out}( p_1 \text{ after } \sigma ) \subseteq \text{out}( s_1 \text{ after } \sigma )
\]
Definition 6. Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_\delta^\ast$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$ is $\text{ioco}_\mathcal{F}$-conform to $S$, denoted by $\mathcal{P} \text{ioco}_\mathcal{F} S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$
Definition 6. Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma^*_\delta$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$ is $\text{ioco}_\mathcal{F}$-conform to $S$, denoted by $\mathcal{P} \text{ioco}_\mathcal{F} S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$
**ioco**

**Definition 6.** Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma^*_\delta$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$ is **ioco$_F$**-conform to $S$, denoted by $\mathcal{P} \textbf{ioco}_F S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$

---

**Diagram:**

- **S:**
  - $s_1 \xrightarrow{?1\$} s_2$
  - $s_2$ has transitions to $s_3$ and $s_4$:
    - $s_3$ has transitions to $!\text{coffee}$ and $!\text{tea}$
    - $s_4$ has transitions to $!\text{tea}$ and $!\text{coffee}$

- **P:**
  - $p_1 \xrightarrow{?1\$} p_2$
  - $p_2$ has transitions to $p_3$:
    - $p_3$ has transition to $!\text{tea}$
**Definition 6.** Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_{\delta}^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_{\mathcal{P}} \rangle$ is $ioco_{\mathcal{F}}$-conform to $S$, denoted by $\mathcal{P} \ ioco_{\mathcal{F}} \ S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \ \text{after} \ \sigma) \subseteq \text{out}(s_1 \ \text{after} \ \sigma)$$
**Definition 6.** Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_\delta^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$ is $\text{ioco}_\mathcal{F}$-conform to $S$, denoted by $\mathcal{P} \text{ioco}_\mathcal{F} S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$

---

Diagram:

- **S**
  - $s_1$ with transition $?1$ to $s_2$.
  - $s_2$ with transitions $!\text{coffee}$ to $s_3$ and $!\text{tea}$ to $s_4$.

- **P**
  - $p_1$ with transitions $?1$ to $p_2$, $?2$ to $p_3$, $p_2$ to $p_4$, and $p_3$ to $p_5$.
  - The transitions are labeled with $!\text{coffee}$ and $!\text{cappuccino}$.
**Definition 6.** Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_\delta^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_\mathcal{P} \rangle$ is $\mathbf{ioco}_\mathcal{F}$-conform to $S$, denoted by $\mathcal{P} \mathbf{ioco}_\mathcal{F} S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$

![Diagram of IOLTS S and P with ioco relationship](image)
**Definition 6.** Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_\delta^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$ is $\text{ioco}_\mathcal{F}$-conform to $S$, denoted by $\mathcal{P} \text{ ioco}_\mathcal{F} S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$

\[
\text{out}(p_1 \text{ after } ?1€) = \{!\text{coffee}, !\text{cappuccino}\} \\
\oplus \\
\text{out}(s_1 \text{ after } ?1€) = \{!\text{coffee}, !\text{tea}\}
\]
**Definition 6.** Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_\delta^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$ is \textbf{ioco}_\mathcal{F}-conform to $S$, denoted by $\mathcal{P} \textbf{ioco}_\mathcal{F} S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$

![Diagram](https://via.placeholder.com/150)
**Definition 6.** Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma_\delta^*$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_\mathcal{P} \rangle$ is $\text{ioco}_\mathcal{F}$-conform to $S$, denoted by $\mathcal{P} \text{ioco}_\mathcal{F} S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$

![Diagram](image)

$\text{out}(p_1 \text{ after } ?1\epsilon) = \{!\text{coffee}, \delta\}$

$\text{out}(s_1 \text{ after } ?1\epsilon) = \{!\text{coffee}, !\text{tea}\}$
Definition 6. Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $F \subseteq \Sigma^*_\delta$. An input-enabled IOLTS $P = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$ is $\text{ioco}_F$-conform to $S$, denoted by $P \text{ ioco}_F S$, iff

$$\forall \sigma \in F : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$
Definition 6. Let $S = \langle S, s_1, \Sigma_I, \Sigma_U, \rightarrow_S \rangle$ be an IOLTS, and let $\mathcal{F} \subseteq \Sigma^*_\delta$. An input-enabled IOLTS $\mathcal{P} = \langle P, p_1, \Sigma_I, \Sigma_U, \rightarrow_P \rangle$ is \text{ioco}_\mathcal{F}$-conform to $S$, denoted by $\mathcal{P} \text{ioco}_\mathcal{F} S$, iff

$$\forall \sigma \in \mathcal{F} : \text{out}(p_1 \text{ after } \sigma) \subseteq \text{out}(s_1 \text{ after } \sigma)$$
A test case is an IOLTS

- having a quiescence label (modeling the observation of quiescence)
- having inputs and outputs swapped
- being tree-structured
- being finite and deterministic
- having final states pass and fail
- where from each state ≠ pass and fail:
  - either a single output and all inputs
  - or all inputs and θ
Formally executing a test case means putting it in parallel with the implementation model, leading to a verdict.
Formally executing a test case means putting it in parallel with the implementation model, leading to a verdict.
Formally executing a test case means putting it in parallel with the implementation model, leading to a verdict.
Formally executing a test case means putting it in parallel with the implementation model, leading to a **verdict**.
Formally executing a test case means putting it in parallel with the implementation model, leading to a verdict.

A test run has been completed: !€?tea with verdict pass

Formally executing a test case means putting it in parallel with the implementation model, leading to a verdict.
Formally executing a test case means putting it in parallel with the implementation model, leading to a **verdict**.
Formally executing a test case means putting it in parallel with the implementation model, leading to a **verdict**.
Formally executing a test case means putting it in parallel with the implementation model, leading to a verdict.
Formally executing a test case means putting it in parallel with the implementation model, leading to a **verdict**.
Formally executing a test case means putting it in parallel with the implementation model, leading to a **verdict**.
Formally executing a test case means putting it in parallel with the implementation model, leading to a verdict.

A test run has been completed: !€ !€ ?tea with verdict pass
The test runs represent the observations.

In the previous example, two observations have been made:
- !€?tea
- !€ !€?tea

Note that the set of all test runs for a given test case comprises all possible observations for all nondeterministic cases.

One more observation could have been made in the previous example:
- !€ !€?coffee (with verdict pass)
When should we stop testing?
Which test cases shall we select?
⇒ How to deal with the practical incompleteness of testing?

1) Accept it, and focus on **heuristics** like code coverage, model coverage, timing constraints, randomness, test purposes, etc.

2) Try to find further **assumptions**, which makes testing complete in practice, i.e., leading to a **finite** sound and complete test suite.
Dijkstra Revisited

- **When** should we stop testing?
- **Which** test cases shall we select?

⇒ How to deal with the practical incompleteness of testing?

The possibly infinite state space, and the nondeterministic character, make computing a **finite** sound and complete test suite an infeasible task!

2) Try to find further **assumptions** which makes testing complete in practice leading to a **finite** sound and complete test suite.
When should we stop testing?
Which test cases shall we select?
⇒ How to deal with the practical incompleteness of testing?

1) Accept it, and focus on **heuristics** like code coverage, model coverage, timing constraints, randomness, test purposes, etc.

2) Try to find further **assumptions**, which makes testing complete in practice, i.e., leading to a **finite** sound and complete test suite.
Dijkstra Revisited

When should we stop testing?

Which test cases shall we select?

⇒ How to deal with the practical incompleteness of testing?

1) Accept it, and focus on **heuristics** like code coverage, model coverage, timing constraints, randomness, test purposes, etc.

2) Try to find further **assumptions**, which makes testing complete in practice, i.e., leading to a **finite** sound and complete test suite.
A Sound and Complete Test Generation Algorithm

Given a specification LTS with initial state $s_0$

Initially compute the set of states $K = s_0$

Do a finite number of recursive applications of the following three nondeterministic choices:

a) Stop the test case with the verdict pass

pass
b) Let the test case produce an output \( !a \) with \( K \) after \( a \leq \bigtriangledown \). Also accept all inputs at the same time.

\[ ta \]

\[ t_{x1} \]

\[ t_{xj} \]

\[ x_i \in \text{out}(S) \]

\[ y_k \notin \text{out}(S) \]

\[ \text{fail} \]

\[ \text{fail} \]

\[ t_a \] is obtained by applying the algorithm with \( K = K \) after \( !a \).

\[ t_{x_i} \] are obtained by applying the algorithm with \( K = K \) after \( x_i \).
c) Let the test case accept all inputs – and quiescence.

\[ t \] is obtained by applying the algorithm with \( K = K \) after
\[ x_i \in \text{out}(S) \]
\[ y_k \notin \text{out}(S) \]

\[ t_{x_i} \] are obtained by applying the algorithm with \( K = K \) after \( x_i \)
We generate a test case out of Q.

Initially, $K = \{q_1\}$
b) Let the test case produce an output !a with $K$ after $a \leq \emptyset$
Also accept all inputs at the same time.
c) Let the test case accept all inputs – and quiescence.
a) Stop the test case with the verdict **pass**
b) Let the test case produce an output \( !a \) with \( K \) after \( a \leq \Delta \).

Also accept all inputs at the same time.
c) Let the test case accept all inputs – and quiescence.
a) Stop the test case with the verdict \textbf{pass}
On-The-Fly Testing

- In every state a test case has to be defined for all possible inputs, i.e., outputs from the system.
- This can easily let the state space explode.
- Some tools do not firstly generate a test suite, and then apply it on the system.
- They combine the test case generation and execution process.
- By so doing, outputs observed from the system guide the “test case” generation.
- So doing avoids this state space explosion problem.
- This kind of testing is called on-the-fly testing.
On-The-Fly Testing

We test on-the-fly with $Q$.

Initially, $K = \{ q1 \}$
b) We choose some **output** !a with **K after a** \( \leq \nabla \)
We also accept all inputs at the same time.

We choose to give !€ to the system.
c) We accept all inputs – and quiescence.

We observe quiescence and hence only need to continue with state $t_5$ and $K = \{q_3\}$
LTS are a common formalism to model **reactive systems**.
LTS are the underlying semantics of several other formalisms like statecharts or process algebras.
Relating two LTS can be done in a **variety of manners**.
Not all relations are suited for testing purposes.
Partitioning the action labels into *inputs* and *outputs* leads to an **IOLTS**.
A common implementation relation for IOLTS is **ioco**.
*ioco* assumes implementation models to be **input enabled**.
*ioco* allows specifications to be not input enabled – allowing for **partial specifications**.
A test case is a tree-structured IOLTS with \textbf{pass} and \textbf{fail} leaves.

Test cases must be \textbf{output-complete} for all possible outputs of the system.

To avoid a state space explosion in test cases, the generation and execution of test cases can be combined – called \textbf{on-the-fly testing}.

A simple sound and complete test case generation for \textit{ioco} exists.

This algorithm is implemented in an on-the-fly manner in the \textbf{TorX tool}.

The \textbf{TGV tool} combines ioco testing with \textbf{test purposes}. 
