Software Testing and Risk Assessment

1 Introduction

1.1 Testing Basics

A test consists of a setup, operations and actions to execute the program/system, and a check to see if the expected result has been obtained.

There are different levels of testing:
- User acceptance test (↑ cost ↓ execution time)
- System/end-to-end tests
- Integration tests
- Unit test (↓ cost ↑ execution time)

Focus on functional testing (what the system should do). Tests can be classified by level of automation.
- Manual testing
- Automated testing (no human running the tests)
  - Recorded tests
  - Scripted testing
  - Keyword driven testing
  - Testing as part of behavior driven development
- Model based testing: testing generation is also automated.

Methodology	Definition Effort	Execution Effort
Manual testing	High	High
Automated Testing	High	Low
Model Based Testing	Low	Low

Different techniques should be used for different projects. Manual testing is unavoidable in legacy systems.

1.2 Test Driven Development

Tests are written before the code. It's a design strategy focused on the what rather than the how. Issues are actively looked for, rather than confirming expectations. The quality of the project is increased trough small incremental changes. TDD is traditionally applied in unit testing, but it's also possible to use it in integration and system tests.

Rules:
1. No code before failing test is written
2. No more tests than sufficient to fail
3. No more code than sufficient to pass tests

TDD is a discipline, design details are needed. The expected result might be unknown.

1.3 Regression Testing

Testing after every change to the source code guarantees that nothing broke between changes. This makes the behavior of code immediately recognizable, and allows to monitor the functioning of your software system. Potential introduced errors are detected earlier (less expensive in the long term).

Regression testing should be done
- After implementing a new feature to be sure that previously implemented features still work
- After refactorings to make sure that the functionality is not affected.

Automated tests are re-executed after each change in the code through a continuous integration (CI) pipeline.

- All tests are executed automatically after every push to the repository
- Avoid merging code with failing tests on the main branch
- Prevent bugs from carrying on into other features/development cycles.

1.4 System Testing

Systems tests are used to test functionality/behavior/features at the system level, interactions between user and system. The entire software system is treated as a sealed container. They test what the system is supposed to do, not how it is built.

There are different ways to formulate a domain logic functionality/feature.
- Feature/requirement description
- User story

EX: User Story

As a <type of user>
I want <some goal>
so that <reason> (optional)

- Behavior-driven development scenario

Behavior-Driven Development (BDD)

In BDD, the expected behavior of a system is defined through textual descriptions called scenarios. BDD extends agile practices and facilitates collaboration and shared understanding among roles.

BDD scenarios are defined trough examples, focusing on user interaction. The **Given-When-Then** structured natural language (Gherkin) is used to specify and test essential use cases.

Given <precondition>
When <action(s)>
Then <post-condition or resulting action>

- Model
- Property

BDD Pipeline:
1. Start from agile approach: choose feature/user story from the current sprint
2. Discovery: the 3 amigos meet. They brainstorm and create concrete examples
3. Formulation: write scenarios in the given-then-when style
4. Automate: create tests from scenarios

Feature	TDD	BDD
Focus	Code/Implementation Correctness. (White Box)	Behavior/Requirement Correctness. (Black Box)
Level	Micro: Unit level (Implementation details).	Macro: System level (User behavior).
Language	Code (Java, Python, C#).	English/Natural Language (Given-When-Then).
Audience	Developers only.	Developers, Testers, and Product Owners.

Table 2: TDD vs BDD

1.5 Three Amigos

The Product Owner Defines the requirements and manages the project scope.

The Product Owner ensures the team builds the right thing by providing User Stories that are clear, valuable, and scope-limited. Instead of vague requests, they must provide specific Acceptance Criteria. This clarity prevents "scope creep" and ensures the developers and testers know exactly when a feature is considered "done."

The Developer Implements the product, handles deployment, and writes technical documentation.

The Developer ensures the requirements are technically feasible. To smooth the process, they must write code that is testable by design (e.g., adding unique IDs to UI elements so automation tools can find them). Furthermore, they bridge the gap between the English scenarios and the code by writing the "glue code" (step definitions) that allows the automated tests to actually drive the application.

The Tester / QA Evaluates the product quality and identifies potential issues.

The Tester brings a critical, "destructive" mindset to the planning phase. They improve the process by identifying edge cases and "unhappy paths" (e.g., "What if the user enters a negative number?") that the Product Owner and Developer might miss. They act as the custodian of the "Living Documentation," ensuring that the test suite remains clean, relevant, and trustworthy over time.

1.6 Black-box Testing

Exploratory Testing Explore the system, observe behavior, adjust tests on the fly based on previous results. Attempt to identify critical parts:

- unclear requirements;
- developed by unexperienced programmer;
- complex code according to software metrics;
- apply risk analysis to identify high risk code

Equivalence partitioning Divide all possible inputs into a finite number of equivalence classes (partitions). An arbitrary representative is picked for each equivalence class. Choose classes that are assumed to:

- Function analogously for inputs in the same class;
- Have sufficient tests with one input per class
- All cause the same fault

Example

Test a program that:

$$\text{for any } N, \max \in \mathbb{Z} : \operatorname{sum}(N, \max) = \begin{cases} \sum_{k=0}^{|n|} \text{ if} \sum_{k=0}^{|n|} \leq \max \\ \texttt{error} \text{ otherwise} \end{cases}$$

Equivalence classes for the inputs[1]:

1. Provide values for both
2. Too few or too many values
3. Integers
4. A non-integer
5. Positive values
6. Negative values

Equivalence classes for the output[2]:

[1]These tests ensure the function handles the type and quantity of data correctly before it even tries to do the math.
[2]These tests ensure the logical branching inside the function works.

7. Values for each of the cases

Boundary Value Analysis Test inputs on, or directly above or below class boundaries. Also consider output boundaries.
Some edge cases for

$$\text{for any } N, \max \in \mathbb{Z} : \text{sum}(N, \max) = \begin{cases} \sum_{k=0}^{|n|} \text{ if } \sum_{k=0}^{|n|} \leq \max \\ \texttt{error} \text{ otherwise} \end{cases}$$

- Empty sum for $N = 0$
- Absolute value for $N = 0$, or $N = -1$
- A value for max such that result $= \max$, or result $= \max + 1$
- Integer overflow boundary

Random Testing Try many random values: use random generators, used to avoid bias. It's easy to implement for numbers, more challenging for: strings, lists, trees, complicated data structures, large data files, constrained data,...

1.7 White-box testing

This type of testing depends on the internal structure of the code and the language it is written in. The code **coverage** is measured trough test cases:

Metric	**Difficulty**	**What it guarantees**
Statement	Low	Every executable statement was executed at least once.
Branch	Medium	Each possible outcome of a decision has been taken at least once.
Modified Decision/Decision Coverage	High	The conditions within the decision independently affect the decision outcome
Path	Extreme	Each possible execution path was covered

Statement Coverage measures the number of source code statements that execute when the code runs. This type of coverage is used to determine whether every statement in the program has been invoked at least once.
The percentage of statement coverage is represented by $\left(\frac{\text{Number of executed statements}}{\text{Total number of statements}}\right) \times 100$.

This metric is satisfied if every single executable line of code is run at least once. It doesn't care how you got there, just that the line was touched.

Condition coverage analyzes statements that include conditions in source code. Conditions are boolean expressions that contain relation operators (`<`, `>`, `<=`, or `>=`), equation operators (`!=` or `==`), or logical negation operators (`!`), but that do not contain logical operators (`&&` or `||`). This type of coverage determines whether every condition has been evaluated to all possible outcomes at least once. Conditions that are inside branching constructs, such as `if`, `while`, and `do-while`, report decision coverage instead of condition coverage.
The percentage of condition coverage is represented by $\left(\frac{\text{Number of executed condition outcomes}}{\text{Total number of condition outcomes}}\right) \times 100$.

> This metric is satisfied only if every “Decision” (like an `if` or `while`) has evaluated to `TRUE` at least once and `FALSE` at least once. You must go down both paths at every fork in the road.

Decision coverage analyzes statements that represent decisions in source code. Decisions are boolean expressions composed of conditions and one or more of the logical operators `&&` or `||`. Conditions within branching constructs (`if/else`, `while`, and `do-while`) are decisions. Decision coverage determines the percentage of the total number of decision outcomes the code exercises during execution. This type of coverage is used to determine whether all decisions, including branches, in the code are tested.
The percentage of decision coverage is represented by $\left(\frac{\text{Number of executed decision outcomes}}{\text{Total number of decision outcomes}}\right) \times 100$.

1.8 Integration Testing

Integration Testing is the phase of testing where individual software modules (which have already been unit tested) are combined and tested as a group. Component integration testing involves

Components a class or set of classes;
Component interfaces public methods used by other components;
Successful integration component interface is used as expected by other components.

Integration testing is performed by incrementally combining components, and then test each increment with tests.

If one of the components is not implemented or is external the system, test drivers, stubs and mocks are used in its place. The **Test Driver** calls the component under test, provides input data and collects and evaluates results. Test doubles for external components replace real, missing or external components. A stub is a simple actor that returns hard-coded responses, and has no decision-making logic. A mock is a more sophisticated actor that imitates the behavior of the intended component. It can make decisions, register and validate interactions.

There are different approaches to integration testing:

Top Down simulates the complete application structure early on. It uses Stubs to replace missing lower-level workers
Bottom Up Validates the Core Logic and Data Integrity first. It verifies that the complex calculations, database saves, and utility functions work perfectly before building a user interface around them.
Sandwich use a mix of stubs/mocks and drivers
Big Bang put all components together and test the whole system.

1.9 Software Testability

Software design influences it’s testability, during integration testing. A design pattern specialized for integration is **dependency injection**. Dependencies are injected in the using class. This aims to separate configuration from use, achieve separation of concerns, loose coupling, high cohesion and the flexibility needed for stubbing and mocking.

Difference Between Integration and System Test

Integration Testing operates under the hood. It tests subsets of components to ensure they communicate correctly. It verifies the technical “wiring” between modules.
System Testing operates at the highest level. It ignores the wiring and treats the software as a complete, sealed box. It verifies that the product as a whole delivers value to the user.

2 LTS, VLTS & STS

Model Based Testing: tests that can be executed automatically are generated from a model.

2.1 Labeled Transition Systems

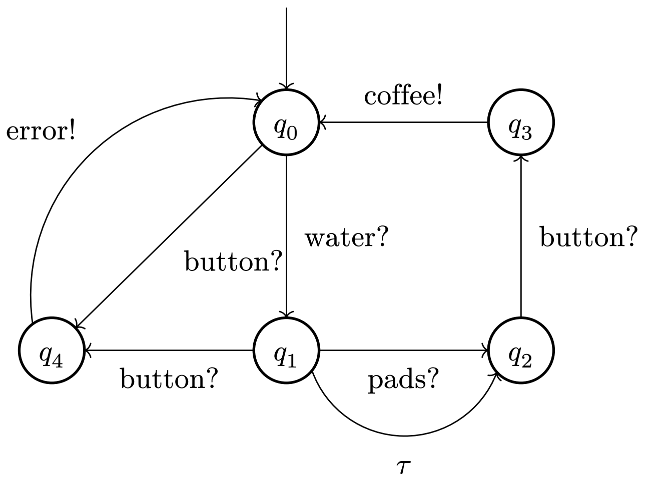


Figure 1: Coffee machine LTS

LTS can be deterministic or non-deterministic. If from the same label there are multiple of the same transitions, the LTS is non deterministic.
Actions marked with ? are inputs, and ones with ! are outputs.
τ is a non-deterministic internal action, that cannot be seen externally.

DEF: LTS

A Labeled Transition System with inputs, outputs is the tuple (Q,L_I,L_O,T,q_0), where
- Q is the set of states
- L_I is the set of input labels
- L_O is the set of output labels such that $L_I \cap L_0 = \emptyset$
- $T \subseteq Q \times (L_I \cup L_= \cup \{\tau\})$ is the transition relation
- q_0 is the initial state

Label sets are divided in inputs and outputs, and they cannot overlap ($L_I \cup L_O = L$).

The coffee machine in the LTS is defined as the touple:

$$\begin{aligned}&(\{q_0, q_1, q_2, q_3, q_4\},\\ &\ \{\text{water?}, \text{pads?}, \text{button?}, \text{coffee!}, \text{error!}\},\\ &\ \{(q_0, \text{water?}, q_1), (q_0, \text{button?}, q_4), (q_4, \text{error!}, q_0), \ldots\},\\ &\ q_0)\end{aligned}$$

And the transition functions are

$$\begin{aligned}&T(q_0) = \{(\text{water?}, q_1), (\text{button?}, q_4)\}\\ &\ldots\\ &T(q_4) = \ldots\end{aligned}$$

DEF: After

Let ε be the empty sequence, $a \in L$ an action, and $\omega \in L^*$ a sequence of actions. Then states after these sequence, starting in $q \in Q$ are:

$$q \textbf{ after } \varepsilon = \{q\}$$
$$q \textbf{ after } a = \left\{q' | q \overset{a}{\Rightarrow} q'\right\}$$
$$q \textbf{ after } a\sigma = \bigcup\{q' \textbf{ after } \sigma \mid q' \in q \textbf{ after } a\}$$

2.2 Quiescence

Quiescence is the absence of output. A state is quiescent if it has no output transitions. The SUT will not produce any output until the next input arrives. A different state may cause a different output.

DEF: Quiescence

A state $q \in Q$ is quiescent iff $\forall o! \in L_0 : q \overset{a!}{\not\Rightarrow}$.
We write $L^\delta = L \cup (\delta)$, and $L_O^\delta = L_O \cup \{\delta\}$

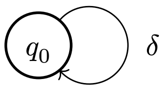


Figure 2: Explicit quiescent state

Quiescent states are denoted with $\delta(q)$. δ can be made explicit by adding them to the LTS. If a state has no outputs (denoted by !, a quiescent transition must be made explicit by adding a δ transition from the node to itself).

> There's always either a quiescent transition or an output transition outgoing from a state.

A non-quiescent state after δ is always $\emptyset$.

An LTS is deterministic if $\forall q \in Q, \forall \rho \in \text{traces}(q) : |q \textbf{ after } \rho| \leq 1$. A non-deterministic state refers to a situation where the future behavior of the system cannot be uniquely determined by its current state and the input label alone.

A state q is considered non-deterministic if one of the following conditions is met:

Multiple Transitions for the same label From state q, there are two or more outgoing transitions with the exact same label leading to different destination states.

EX: Non deterministic state

If the system is in State A and receives input 'x', it can go to State B OR it can go to State C. You cannot predict which one it will choose.

Hidden Transitions (τ-transitions) The system can transition from state q to another state without consuming any input label (denoted as τ). This means the system can change states "silently" or spontaneously.

2.3 Paths and Traces

DEF: Path and Trace

A path π is an infinite sequence $q_0 a_0 q_1 a_1 \ldots$ such that $\forall i \in \mathbb{N} : (q_i, a_i, q_{i+1}) \in T$. We define $\text{traces}(q) = \left\{\sigma \in L^* | \; q \overset{\sigma}{\Rightarrow}\right\}$.
A trace is a projection of paths to its labels excluding τ.

- $\text{trace}(\varepsilon) = \varepsilon$
- $\text{trace}(qaq') = \begin{cases} \varepsilon \text{ if } a = \tau \\ a \text{ otherwise} \end{cases}$
- $\text{trace}(qaq'\pi) = \text{trace}(qaq')\text{trace}(\pi)$

We write $\text{traces}(S) = \text{traces}(q_0)$.

Equivalent definition of traces and **after**:

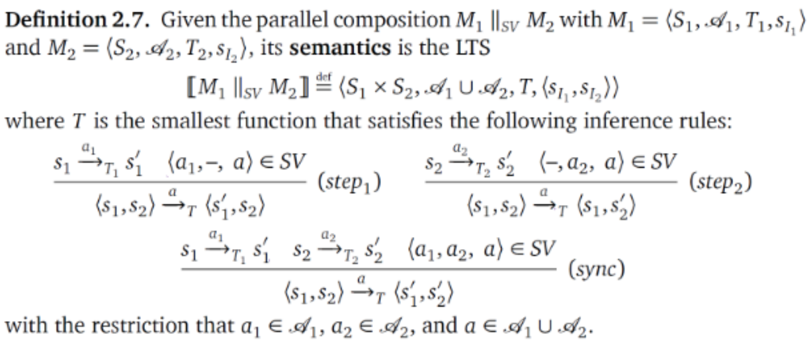
- $\text{traces}(q) = \{\text{trace}(\pi) | \text{first}(\pi) = q \wedge \pi \text{ is a finite path}\}$
- q **after** $\sigma = \text{last}(\{\pi \in \text{trace}^{-1}(\sigma) | \text{first}(\pi) = q\})$

DEF: Angelic completion

The angelic completion of an LTS $S = (Q, L_I, L_O, T, q_0)$ is the LTS $S^\delta = (Q, L_I, L_O, T', q_0)$ where $T' = T \cup \{(q, \delta, q) |\ q \in Q \wedge \delta(q)\}$. We define the **suspension traces** of S as $\text{Straces}(S) = \text{traces}(S^\delta)$.

2.4 Networks of LTS

Labeled Transition Systems (LTS) are foundational and simple models for describing the behavior of concurrent systems. They consist of states and labeled transitions, but using them explicitly to represent large systems can be unwieldy. To address this, large LTS are often expressed in compact forms by constructing them from smaller, modular components. This is achieved through parallel composition (denoted as $P_1 \parallel P_2$), which allows independent processes to execute simultaneously while interacting through specific points. These interactions are defined by synchronization vectors, which list the actions that must occur together across different components. Internal computations or unobservable steps within the system are abstracted away using the silent action (τ), ensuring that the model focuses only on externally visible behavior.



Definition 2.7. Given the parallel composition $M_1 \parallel_{SV} M_2$ with $M_1 = \langle S_1, \mathcal{A}_1, T_1, s_{I_1} \rangle$ and $M_2 = \langle S_2, \mathcal{A}_2, T_2, s_{I_2} \rangle$, its **semantics** is the LTS

$$[\![M_1 \parallel_{SV} M_2]\!] \stackrel{\text{def}}{=} \langle S_1 \times S_2, \mathcal{A}_1 \cup \mathcal{A}_2, T, \langle s_{I_1}, s_{I_2} \rangle \rangle$$

where T is the smallest function that satisfies the following inference rules:

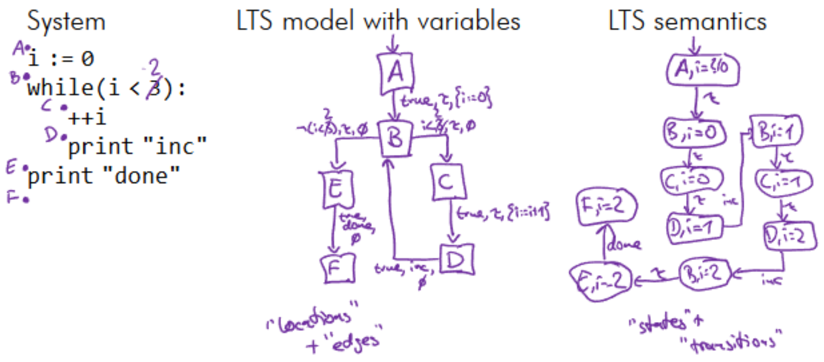
$$\frac{s_1 \xrightarrow{a_1}_{T_1} s_1' \quad \langle a_1, -, a \rangle \in SV}{\langle s_1, s_2 \rangle \xrightarrow{a}_T \langle s_1', s_2 \rangle} \ (step_1) \qquad \frac{s_2 \xrightarrow{a_2}_{T_2} s_2' \quad \langle -, a_2, a \rangle \in SV}{\langle s_1, s_2 \rangle \xrightarrow{a}_T \langle s_1, s_2' \rangle} \ (step_2)$$

$$\frac{s_1 \xrightarrow{a_1}_{T_1} s_1' \quad s_2 \xrightarrow{a_2}_{T_2} s_2' \quad \langle a_1, a_2, a \rangle \in SV}{\langle s_1, s_2 \rangle \xrightarrow{a}_T \langle s_1', s_2' \rangle} \ (sync)$$

with the restriction that $a_1 \in \mathcal{A}_1$, $a_2 \in \mathcal{A}_2$, and $a \in \mathcal{A}_1 \cup \mathcal{A}_2$.

2.5 LTS with variables

Real systems work with data (VLTS)



```
A•i := 0
B•while(i < 3):
  C•  ++i
  D•  print "inc"
E•print "done"
F•
```

Parallel composition of VLTS can also be denoted

2.6 Symbolic Transition Systems (STS)

Actions now have parameters. Variables are no longer just in the states. States are locations with variables.

A gate is the label or the interaction mechanism that is used during a transition, and is divided between input and output gates. Switches are the actual movement from one state (called location in an STS) to another, defined by a switch relation (R).

DEF: Switch

A switch is a tuple containing:
- l_1: the starting location
- λ (gate): the "label" used
- $p_0 \ldots p_k$: the data parameters
- ϕ (guard): the condition that must be true to take this path. If implicit resolves to true
- ψ (update): changes made to variables
- l_2: the target location

The gate is just the name of the event, the switch is the entire event.

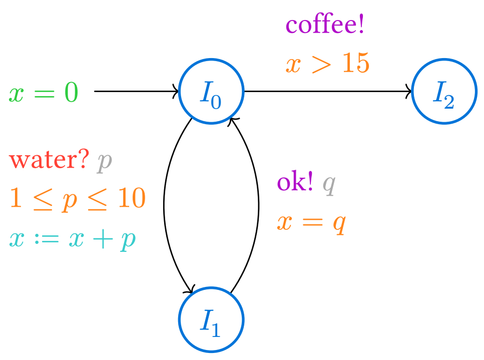


Figure 5: Coffee machine STS

- Locations
- Initial assignment (assigns values to location variables)
- Input gates
- Output gates
- Gate parameters
- Guards (boolean expressions)
- Assignments to location variables
- Switches

DEF: Symbolic Transition System

A symbolic transition system (STS) with inputs and outputs is a touple $(\mathcal{L}, \mathcal{V}_l, \mathcal{V}_p, \Gamma_I, \Gamma_O, \mathcal{R}, l_0, \text{ini})$ where:
- $\mathcal{L}$ is a finite set of locations ($\{I_0, I_1, I_2\}$),
- $\mathcal{V}_l$ is a finite set of location variables ($\{x\}$),
- $\mathcal{V}_p$ is a finite set of gate parameters such that $\mathcal{V}_p \cap \mathcal{V}_l = \emptyset$ ($\{p, q\}$),
- Γ_I is a finite set of input gates ($\{\text{water?}\}$),
- Γ_O is a finite set of output gates such that $\Gamma_I \cap \Gamma_O = \emptyset$ ($\{\text{ok!}, \text{coffee!}\}$),
- $\mathcal{R} \subseteq \mathcal{L} \times (\Gamma_I \cup \Gamma_O) \times V_p^* \times \mathcal{T}_{\text{bool}}(\mathcal{V}_l \cup V_p) \times \mathcal{T}_{\text{bool}}(\mathcal{V}_l \cup V_p)^{\mathcal{V}_l} \times \mathcal{L}$ is the switch relation with a finite number of elements,
- $l_0 \in \mathcal{L}$ is the initial location ($I_0 \in \mathcal{L}$),
- $\text{ini} \in \mathcal{T}^{\mathcal{V}_l}$ is the initialization ($\text{ini} \in \mathcal{T}(\emptyset)^{\mathcal{V}_l} = (x := 0)$)

Each gate λ is only used with a fixed sequence of parameters $p_0, ..., p_k$. The pair $(\lambda, p_0, ..., p_k) \in \Gamma \times \mathcal{V}_p^*$ is called an interaction.
For any switch $(l_1, \lambda, p_0...p_k, \phi, \psi, l_2) \in \mathcal{R}$ we require that:

- $p_0...p_k$ is a sequence of distinct variables
- $\phi \in \mathcal{T}_{\text{bool}}(\mathcal{V}_l \cup \{p_0, ..., p_k\})$
- $\psi \in \mathcal{T}(\mathcal{V}_l \cup \{p_0, ..., p_k\})^{\mathcal{V}_l}$
- For assignment $\psi \mapsto := e$ we have $\text{type}(x) = \text{type}(e)$

Assignments on switches happen simultaneously ($x := y, y := x$).
If a location variabile is not assigned by a switch, its value stays the same ($c := c$).

Scope

Location variables have global scope (whole STS). Gate parameters are local to the guards and switch assignments.

DEF: Interpretation of S

Let $S = (\mathcal{L}, \mathcal{V}_l, \mathcal{V}_p, \Gamma_I, \Gamma_O, \mathcal{R}, l_0, \text{ini})$ be an STS. The interpretation of S is defined as the LTS

$$[\![S]\!] = (\mathcal{L} \times \mathcal{U}^{\mathcal{V}_l}, \Gamma_I \times \mathcal{U}^*, \Gamma_O \times \mathcal{U}^*, \rightarrow, (l_0, \text{ini}))$$

where $\rightarrow = \{(q, i, q') |\ q \in \mathcal{L} \times \mathcal{U}^{\mathcal{V}_l}, i \in \Gamma \times U^*, q' \in q \textbf{ after } i\}$

3 Formal Testing

Focus on black box system level testing. Rely only on inputs and outputs without looking at the code.

DEF: Test Case

A test case is formally defined as a specification that dictates:

- Which inputs to provide to the SUT.
- Which outputs are expected from the SUT.
- A final verdict (**PASS** or **FAIL**) returned upon the completion of a finite execution.

A test case is a cycle of providing input, waiting for output, or stopping with a verdict.

Test cases can be modeled as Labeled Transition Systems with specific rules:
Angelic Completion Test cases include a special label δ representing quiescence or timeouts.
Pass/Fail States There are two distinct states, Pass and Fail, which have self-loops for all outputs (to handle any subsequent behavior harmlessly) but no transitions for inputs.
Structure A test case must be deterministic and acyclic (except for the recursion in the Pass/Fail states) to ensure a finite execution.

Test Case formal definition

A test case for an LTS S^δ is an LTS $t = (Q^t, L_I, L_O \cup \{\delta\}, T^t, q_0^t)$ such that:

1. t uses the same labels as S^δ
2. There are two special states: **PASS**, **FAIL** $\in Q^t$
3. States **PASS** and **FAIL** have self-loops for all outputs, including δ
4. States **PASS** and **FAIL** have no transitions for inputs
5. t has no cycles except those in **PASS** and **FAIL**
6. t is deterministic

7. Every state enables all outputs L_O, and either one input or

   $\delta : \forall q \in Q^t : (|\text{in}(q)| = 0 \land \text{out}(q) = L_O \cup \{\delta\}) \lor (|\text{in}(q)| = 1 \land \text{out}(q) = L_O)$

3.1 IOCO

ioco stands for **Input Output COnformance**. It is a mathematical relation that indicates if an Implementation (I) conforms to, or obeys, its Specification (S).

It relies on two rules regarding behavior:

Inputs The implementation must accept the same or more inputs than the specification. (It shouldn't crash just because we pressed a button we usually don't press).

Outputs The implementation must produce the same or fewer outputs than the specification. (It shouldn't invent behaviors we didn't authorize).

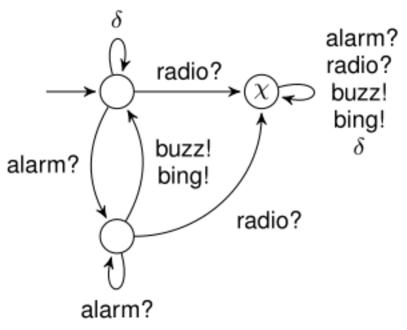
Test Assumptions

We assume the System Under Test (SUT) can be modeled as an Input-Enabled LTS (IELTS).

This means that in any state, the implementation is ready to accept any input from the logical set (L_I). Even if the button shouldn't work right now, the system physically accepts the press (even if it just ignores it): $\forall q \in Q : \text{in}(q) = L_I$

Alongside quiescence, we have to deal with **underspecification**: What if our specification doesn't mention what happens when I press a button? In ioco, this is treated as underspecification. We assume "anything is allowed" for missing input transitions. This is modelled by adding a "Chaos state" (χ). If the specific doesn't forbid it, the implementation is free to do it.

Underspeicification is made explicit with demonic completion by adding the chaos state χ.



Ioco defines whether a system implementation (I) correctly follows it specification (S). The implementation is not allowed to have new behaviors or inputs.

$$\text{ioco} : \text{IELTS} \times \text{LTS} \rightarrow \text{bool}$$

Mathematically, the relationship is defined between an IELTS I and a specification S. They are considered compliant (I ioco S) if and only if

$$\forall \sigma \in \text{traces}(S_\delta) : \text{out}(I_\delta \text{ after } \sigma) \subseteq \text{out } (S_\delta \text{ after } \sigma)$$

- $\forall \sigma \in \text{traces}(S_\delta)$: we only check for traces (sequences of actions) that are actually present in the specification.
- $\text{out}(I_\delta \text{ after } \sigma)$: this is a set of all outputs (including quiescence) the implementation produces after the sequence σ.

- out $(S_\delta$ after $\sigma)$: this is the set of all allowed outputs (including silence) that the specification permits after a specific history of events (σ).

> The outputs produced by the implementation must be a subset of the outputs allowed by the specification.

3.2 Test Generation

Test generation is used to bridge the gap between the specification and the actual testing.

3.2.1 Batch Test Generation

`batchGen` involves creating the entire test case as a complete diagram or data structure before interacting with the real system. This algorithm works recursively to build a tree-like test case from the specification.

FUNCTION: `batchGen`

The algorithm looks at the current states of the specification and makes a nondeterministic choice to do one of the following:

1. Stop: end the test case at the current point;
2. Input: choose a valid input (e.g. `button?`) and add it to the test case
3. Observe: create branches for all possible outputs (including δ) that the specification allows at that point

The output is a complete test case (LTS graph) that typically ends in **PASS** or **FAIL** states.

EXAMPLE: `batchGen`

1. The generator starts at the initial state q_0;
2. it adds a `button?` input and then creates two branches to handle the possible responses `coffee!` and `error!`;
3. It continues adding steps like `water?` until it decides to stop, resulting in a complete tree structure

3.2.2 On-the-fly Test Generation

`onTheFlyGen` combines generation and execution. Instead of building a full map beforehand, the algorithm generates a single step, executes it against the SUT immediately, and then decides what to do based on the result.

FUNCTION: `onTheFlyGen`

The algorithm sits in a loop with the real system. At each step, it checks the current state Q':

1. Pass: it can decide to stop and return a **PASS** verdict;
2. Observe input: it waits to see if the system produces an output $x!$
   - If the output occurs, it checks if $x!$ is allowed by the specification $(x! \in \text{out}(Q'))$
   - If the output is not allowed, it immediately returns **FAIL**
   - If the output is allowed, it updates its current state $(Q'$ after $x!)$ and continues
3. Supply input: it can choose to send an input (`button?`) to the system, provided the system hasn't already produced an output.

This approach doesn't produce a reusable graph. Instead, it produces a single verdict (**PASS** or **FAIL**) for that specific run.

EXAMPLE: `onTheFlyGen`

1. The tester sends `button?`
2. The SUT returns `coffee!`
3. The SUT returns `error!`
4. Since `error!` is not allowed at that specific moment in the specification, the test stops immediately with **FAIL**.

Summary:

Batch You build the whole test case first, then send the execution through it later. You can save these test cases and reuse them.

On-the-fly You build the path one step at a time while walking it. You generate a test step and execute it immediately.

3.3 Special Test cases

3.3.1 Test Cases for Non-deterministic Systems

When a Specification allows for multiple different outputs after the same input (nondeterminism), a standard linear test might not work. Specific test cases designed to "observe" specific transitions are needed.

3.3.2 Distinguishing Test Cases

A distinguishing test case is a test case designed to distinguish between two different start states (is the system currently in state A or state B?). In order to function, you send an input that produces a different output depending on which state the system is currently in. By observing the output, you can identify the initial state.

3.3.3 Homing Test Cases

A homing test case is defined as a test case that transitions the system to a particular "home" state like q_0 from **any** starting state. The goal is to synchronize the system to force it back to a known state.

4 Model Based Testing with Axini

Boehm's law: the cost of bugs grows exponentially with time. Most of the testing is done in the integration/acceptance test.

Model based testing is a type of black box testing. It works with expectations of what the SUT should do without having access to the source code. It has higher coverage and more automation.

The model is a reflection of the informal requirements. There is a correctness criterion between the model and the implementation, in this case it's IOCO.

Models are written in the Axini modelling language (AML). **The models are always from the perspective of the system**.

The SmartDoor code has to be memorized somewhere. Write to when lock, check when unlock. SECLOC-07 -> quiescence (slides p.12)