University of Twente

Software Testing and Risk Assessment

Summary of the Lectures

W. van den Brink
Based on the 2025 version of STAR

April 14, 2025

Contents

Part I

Risk

Lecture 1

Introduction to Risk

1.1 What is risk?

Risk is the (positive and negative) effect of uncertainty on objectives. Both positive and negative impacts are analyzed. We emphasize on goals. Goals help in keeping focus.

Risk has two main ingredients: (negative) **impact** and **uncertainty**. We find it in all domains, all systems and all phases.

The impact of a risk is usually part of the **big five**:

- Money
- Time
- Reputation
- Safety
- Quality

Impact has a qualitative aspect (what is it?) and a quantitative aspect (how bad is it?). An example scale of the latter is given below.

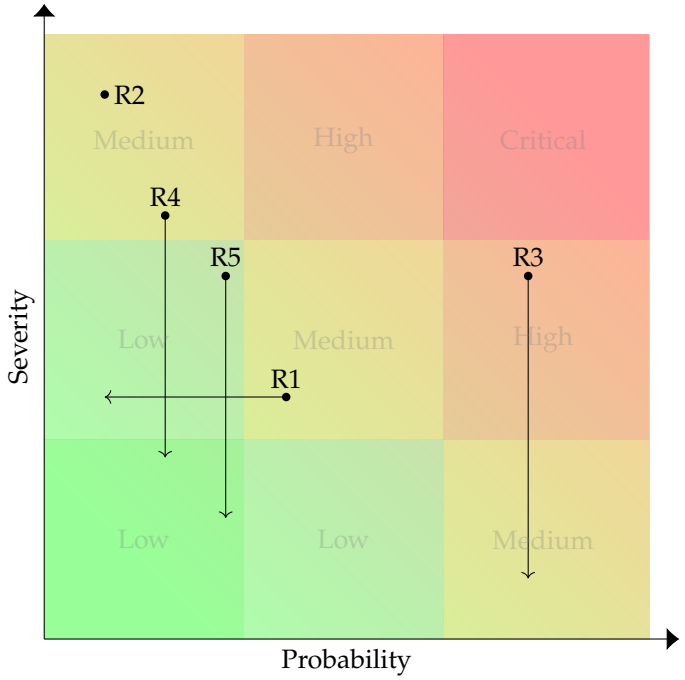
Scale		Patient safety
1	Minor	Discomfort
2	Moderate	Light injury
3	Critical	Permanent injury
4	Catastrophic	Death

The **uncertainty** of a risk indicates how rare it occurs. An example scale is given below.

Scale		Probability of occurrence
1	Very unlikely	Never occurred in our field
2	Possibly	Has occurred once in 10 years
3	Once in a while	Every few years
4	Often	Several times a year
5	Regularly	Several times a month

1.2 Risk assessment matrix

Risk is often displayed in a **risk matrix** (see below). In this heat map, the horizontal axis represents a risk's probability, and the vertical axis shows the severity, if the risk becomes reality. A **risk level** is more or less its probability times its severity. Consequently, the further a risk moves away from the origin in both directions, the higher its level.



Risk assessment can be performed using the Demming cycle, which consists of four repeating phases:

1. **Plan**
   (a) Determine goals
2. **Do**
   (b) Identify risks
   (c) Prioritize risks
   (d) Design measures
3. **Check**:
   (e) Evaluate effectiveness
4. **Act**
   (f) Implement
   (g) Document and communicate

Boehm's law states that fixing an issue is more costly as the moment of discovery is further along the development process of a product.

The recommendation is to always perform a Demming cycle in every part of the process.

There are four **strategies to risk management**:

- **Tolerate**: accept the risks. No effect on the risk's severity or probability.
- **Terminate**: stop, or do not start, the activity. Reduces the probability.
- **Treat**: either reduce the likelihood, or reduce the severity.
- **Transfer**: transfer the risk to someone else. Reduces severity.

1.3 Failure Mode & Effects Analysis (FMEA)

FMEA is a text/spreadsheet based design tool for assessing risks for different ways (modes) a system can fail. The goal is to take appropriate measures. Its main benefit is that risks can be sorted by their risk priorty number (RPN):

$$\text{RPN} = \text{probability} \times \text{severity} \times \text{detection}$$

In practice, an FMEA is a spreadsheet with the following columns:

Column	Description
Process step	What is the step?
Potential failure mode	In what ways can the step go wrong?
Potential failure effect	What is the impact on the customer if the failure mode is not prevented or corrected?
Severity (SEV)	How severe is the effect on the customer on a scale from 1 to 10?
Potential causes	What causes the step to go wrong?
Occurrence (OCC)	How frequently is the casuse likely to occur on a scale from 1 to 10?
Current process controls	What are the existing controls that either prevent the failure mode from occurring or detect it should occur?
Detection (DET)	How difficult is detection of the failure mode or its cause on a scale from 1 to 10? **N.B.** Perhaps counter-intuitively, a higher detection value in the FMEA means it is *more difficult* to detect.
RPN	Risk probability calculated as $\text{RPN} = \text{SEV} \times \text{OCC} \times \text{DET}$
Action recommended	What are the actions for reducing the occurrence of the cause or for improving its detection? Provide actions on all high RPNs and on severity ratings of 9 or 10.

Constructing an FMEA:

1. Determine design tree: decompose the system in its individual components.
2. Determine function tree: determine the functions of the system.

   First list all functions of the system. Then determine what every system component should be doing for those functions. Finally determine the function tree.

   The function tree is a tree with the design at its root, systems as inner nodes, and concrete components and their tasks or desired properties as leaves.
3. Determine failures for each function: annotate every node and leave, including the root node, with possible failures:
   - Designs (root node) have failure effects: the user observes that something went wrong. Failure effects have an effect on the user.
   - Systems (inner nodes) have failure modes: the subsystem does not do what it has to do, due to an error in the environment. Failure modes are actual failures.
   - Concrete components (leaves) have failure causes: something went wrong, so it does not perform its action or have its desired property. Failure causes contribute to a failure.
4. Quantify: start putting all failure effects and modes in a spreadsheet.
5. Find measures: note ways to migitate the risk in the spreadsheet.
6. Reassess: keep applying FMEA.

The systematic approach of an FMEA helps. The mathematics for the RPN are a bit wonky, but they *can* be appropriate.

Lecture 2

Fault Trees

2.1 Definitions and properties

Definition 1 (Fault tree)**.** A fault tree is a directed, acyclic graph. Vertices in the graph represent events:

- An event with no outgoing edges is a basic event, with optional known probability.
- An event with incoming and outgoing edges is an intermediate event. Its behavior is dictated by a gate:
  - An intermediate event with an AND gate fails if all of its children fail.
  - An intermediate event with an OR gate fails if one or more of its children fail.
  - An intermediate event with a voting gate fails if n or more of its children fail.
- An event with no incoming edges is the top level event.

Note. Important assumption: the basic events should be independent and have no common cause. Consequently, if there is a common cause for some subset of the basic events, it should be added to the fault tree.

Advantages:

- Graphical, so easy to understand
- Compositional, so easy to expand/develop basic events into intermediate events
- Detailed analysis makes for well-informed decisions

Disadvantages:

- Impact is binary, either the system fails or it does not. Not easy to model partial failure.
- Relation between events is binary, either it causes the event or does not. Suitable for hard engineering models, not so for soft things like a holiday.
- Probabilistic analysis requires accurate statistics.

2.2 Applications

- **Validation of the fault tree.** If all basic events in a cut set fail, will the system fail? If the fault tree is correct, it should.
- **Identification of weak points.** If a minimal cut set has few elements, few things have to go wrong for the system to fail. If a minimal cut set has one element, that element is a single point of failure. Do keep the probability of this event in mind, so ask the experts whether it is acceptable. The fault tree helps you identify questions for experts.
- **Common cause failures.** Do elements in a cut set have a common cause? Examples: spikes on the road for tire failures. If you have common causes, update the fault tree.
- **Risk prioritization.** Use cut set metrics.

- **Calculating failure probability.** Use probability theory.

2.3 Metrics

A fault tree can be used for both qualitative and quantitative analysis.

Definition 2 (Cut set)**.** A cut set is a set of events that makes the system fail.

Definition 3 (Minimal cut set)**.** A minimal cut set is a cut set of which no elements can be removed.

In other words: every event in the minimal cut set is required to make the system fail.

Note. A minimal cut set need not be the smallest cut set.

Definition 4 (Order of a cut set)**.** The order of a cut set is the number of elements in the cut set. It determines the importance of the cut set.

Generally, a cut set of lower order is less vulnerable than a cut set with higher order.

Definition 5 (Frequency of a basic event)**.** The frequency of a basic event is the amount of minimal cut sets that contain the event. It determines the importance of the basic event.

Generally, an event with high frequency has a large impact on safety.

Always ask the experts and think about probability. A very unlikely event might occur in many cut sets. It has a high frequency, but is not very important.

If you know the probability of each basic event, you can calculate the probability of the cut set as the product of the basic event probabilities.

2.4 Structure function

Definition 6 (Structure function)**.** The structure function $\Phi_F : \{0,1\}^{\#\text{BEs}} \to \{0,1\}$ describes the behavior of the fault tree.

Given values $e_1, e_2, \ldots, e_n$, with $e_i \in \{0,1\}$, $\Phi_F\left(\overrightarrow{e}\right)$ tells whether F fails.

Definition 7 (Extended structure function)**.** The structure function $\Phi_F : \{0,1\}^{\#\text{BEs}} \times \text{Elements} \to \{0,1\}$ describes the behavior of an element E in the the fault tree.

Given values $e_1, e_2, \ldots, e_n$, with $e_i \in \{0,1\}$, and element E, $\Phi_F\left(\overrightarrow{e}, E\right)$ tells whether fault tree element E fails.

The function is recursively defined:

- If E is an AND gate: $\Phi_F\left(\overrightarrow{e}, E\right) = 1$ iff $\Phi_F\left(\overrightarrow{e}, E'\right) = 1$ for **all** children E' of E;
- If E is an OR gate: $\Phi_F\left(\overrightarrow{e}, E\right) = 1$ iff $\Phi_F\left(\overrightarrow{e}, E'\right) = 1$ for **some** children E' of E;
- ...et cetera.

2.5 Determining unreliability

2.5.1 Bottom-up

Suppose $p_i = \mathbb{P}(e_i = 1)$ are known and are independent.
Then the unreliability $\mathcal{U}(F) = \mathbb{P}\left(\Phi_F\left(\overrightarrow{e}, \text{top}\right) = 1\right)$ is the probability that the system fails.

It is then a matter of combining the probabilities bottom-up using probability theory:

- Combining probabilities with an AND gate:

$$\mathbb{P}[A \wedge B] = \mathbb{P}[A]\mathbb{P}[B]$$

iff A and B are independent;

- Combining probabilities with an OR gate:

$$\begin{aligned}\mathbb{P}[A \vee B] &= \mathbb{P}[A] + \mathbb{P}[B] - \mathbb{P}[A \wedge B] \\ &= 1 - (1 - \mathbb{P}[A])(1 - \mathbb{P}[B])\end{aligned}$$

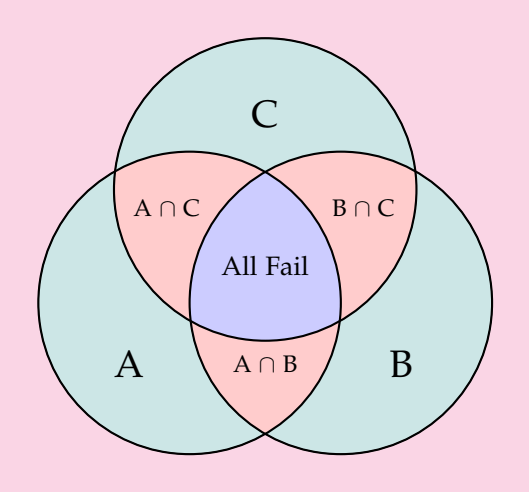
- Combining probabilities with a voting gate, with p_e the probability of the event occurring and n total events:

$$\begin{aligned}\mathbb{P}[\text{0 events occur}] &= (1 - p_e)^n \\ \mathbb{P}[\text{exactly } k \text{ events occur}] &= np_e(1 - p_e)^{n-k} \\ \mathbb{P}[\text{k or more events occur}] &= \sum_{i=k}^{n} \binom{n}{i} p^i (1 - p)^{n-i}\end{aligned}$$

However, the course seems to only use voting gates where $k = 2$, which simplifies the equation:

$$\begin{aligned}\mathbb{P}[\text{2 or more events occur}] &= \mathbb{P}[\text{exactly 2 events occur}] + \mathbb{P}[\text{all events occur}] \\ &= \binom{n}{2} p_e^2 (1 - p)^{n-2} + p^n\end{aligned}$$

Why does this work? Well, look at the following Venn diagram:



The system fails in the red regions and in the blue region.

While it intuitively feels wrong to only consider the cases where either 2 or all events occur, the Venn diagram shows that this approach indeed covers all cases. **But only when $k = 2$!**

We can get rid of the binomial coefficient by using the complement rule:

$$\begin{aligned}\mathbb{P}[\text{2 or more events occur}] &= 1 - \mathbb{P}[\text{exactly 0 events occur}] + \mathbb{P}[\text{exactly 1 event occurs}] \\ &= (1 - p)^n + np(1 - p)^{n-1}\end{aligned}$$

Example 1. 5 tires can fail, 2 or more are needed for the subsystem 'tires' to fail.

Calculate the probability of the subsystem failing as 1 minus the probability it does not fail, i.e. 0 or 1 tires fail:

$$\begin{aligned}\mathbb{P}[\text{0 tires fail}] &= (1-p_e)^5 \\ \mathbb{P}[\text{1 tire fails}] &= 5p_e(1-p_e)^4 \\ \mathbb{P}[\text{subsystem fails}] &= 1-(1-p_e)^5 - 5p_e(1-p_e)^4\end{aligned}$$

Note. The above computations **only** work if the children of the subsystems are independent. This fails for example when one basic event causes failure in subsystems that are (children of) the gate.

In this case, use conditional probabilities:

1. Calculate the failure probability $\mathbb{P}(F \mid e=1)$ if the event does occur;
2. Calculate the failure probability $\mathbb{P}(F \mid e=0)$ if the event does not occur;
3. The failure probability is now equal to the sum of these probabilities.

The computation gets quite large as the fault tree grows. Three ways to speed it up:

1. Use minimal cut sets
2. Disregard extremely small possibilities
3. Use BDDs

2.5.2 With minimal cut sets

1. List all minimal cut sets of the system.
2. Compute the probability of every minimal cut set.
3. Sum these probabilities to determine the system failure probability.

This gives an over-approximation. That does not have to be a bad thing. The general question in an FT is: how (un)reliable is the system? If we overestimate the unreliability, we are careful, which is good.

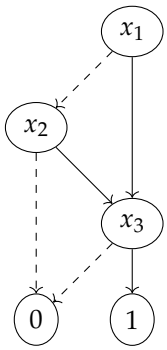
You can also disregard MCS that are not prominent, i.e. with a large order. If the probabilities of the events are low, those MCS will rarely occur and are not very important.

2.5.3 Binary decision diagrams

Definition 8 (Binary decision diagram)**.** A binary decision diagram is a compact graphical representation for Boolean functions $f(x_1, x_2, \ldots, x_n)$ with $x_i \in \{0,1\}$.

The diagram consists of vertices and leaves forming a decision tree. Every vertex represents a decision: choosing 0 or 1 for a variable. We follow the dashed line for value 0, and the solide line for value 1.

The following BDD models the binary function $f(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge x_3$:



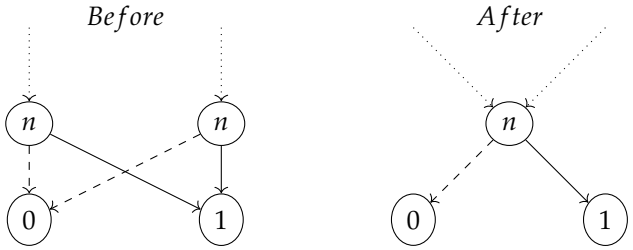
Lemma. A structure function of a fault tree is a binary function and can thus be compactly represented using a binary decision diagram.

Generally, we start with an exponentially sized BDD, and by reducing, the BDD gets smaller. Not always. Also depends on order of variables.

Two optimization steps:

- If a vertex has the same destination for both 0 and 1, remove the vertex, and instead let its incoming node(s) point to the destination:

- If multiple vertices point to the same destination, remove all but one and let all parent nodes point to the remaining node:



To calculate failure probability with a binary decision diagram, model the fault tree as a BDD and reduce it. Label the edges with the probabilities, then sum all edges that lead to 1. This is all simple graph theory.

BDDs can also find MCSs, but that is not required for the exam.

One can also reduce a BDD from scratch. Here, we do not start with the full, exponential BDD, but we simplify the boolean equation every step of the way. This should in principle give a minimal BDD, but in practice we never apply every possible simplification step. Thus, one should always apply BDD reduction with the above steps after creating a reduced BDD from scratch.

2.6 Unreliability over time

Definition 9 (Unreliability)**.** The unreliability $U(t) = P[\text{failure before time } t]]$ of a system expresses the probability that a system fails given its runtime t.

Definition 10 (Reliability)**.** The reliability $R(t) = P[\text{no failure until time } t] = 1 - U(t)$ of a system expresses the probability that a system still works after its runtime t.

If every basic event has an unreliability function, you could calculate the unreliability function of the system as a whole. This is a continuous probability function.

There are some standard choices for the reliability function:

- Uniform distribution
- Gaussian distribution
- Exponential distribution, commonly used because:
  - Realistic model for degradation
  - Mathematically tractable: reasonably easy formulae
  - Approximation via composed exponentials
- Weibull distribution, also often used, but not discussed during this course

Described using the cumulative density function: $F(x) = P[X \leq x]$

With exponential failure, $p_i(t) = \mathbb{P}[BE_i \text{ fails before time } t] = 1 - e^{-\lambda i \cdot t}$, and now $p_i(t)$ varies over time. You can simply substitute the probabilities in the computation: simply replace p_i with $p_i(t)$ in your calculations. If you then graph the probability over time, you approximate the failure rate over time.

The difficulty lies in making the FT and BDD, not in executing the computation.

Lecture 3

Dynamic fault trees

3.1 System (un)reliability continued

Recall the following from the previous lectures:

- The **unreliability** of a system $\mathcal{U}(t) = \mathbb{P}[\text{failure before time } t]$.
- The **reliability** (or **survivor function**) of a system $\mathcal{R}(t) = \mathbb{P}[\text{no failure until time } t] = 1 - \mathcal{U}(t)$.
- Generally, $\mathcal{U}(t)$ tends to 1 as time goes on. Formally: $\lim_{t\to\infty} \mathcal{U}(t) = 1$.
- Furthermore, $\mathcal{U}(0) = 0$ and, consequently, $\mathcal{R}(0) = 0$. Intuitively: a system cannot fail if it has not been in use at all, i.e. it has *just* left the factory.

Unreliability is generally modeled as a cumulative density function using the exponential distribution, with:

$$\mathbb{P}[\text{fail before } t] = \mathbb{P}[X \leq t] = 1 - e^{-\lambda t}$$

Here, X is a random variable denoting failure time, $X \sim \exp(\lambda)$, with λ the **failure rate**: the expected number of fails per time unit, and $\mathbb{E}[X] = \frac{1}{\lambda}$.

Theorem 1 (Expected value of the exponential distribution)**.** The expected value of an exponential distribution with parameter λ is

$$X \sim \exp(\lambda) \implies \mathbb{E}[X] = \frac{1}{\lambda}$$

Proof. Well, we know $\mathbb{E}[X]$, with $P(x) = \mathbb{P}[X \leq x]$, is $\mathbb{E}[X] = \int_0^\infty xP'(x)\,dx$.

For $X \sim \exp(\lambda)$, $P(x) = \mathbb{P}[X \leq x] = 1 - e^{-\lambda x}$.

Then $P'(x) = \lambda e^{-\lambda x}$.

$$\begin{aligned}
\mathbb{E}[X] &= \int_0^\infty xP'(x)\,dx \\
&= \int_0^\infty x\lambda e^{-\lambda x}\,dx \\
&= \lambda \int_0^\infty xe^{-\lambda x}\,dx \\
&= \lambda \cdot \frac{1}{\lambda^2} \\
&= \frac{\lambda}{\lambda^2} = \frac{1}{\lambda}
\end{aligned}$$

This works, because $\int_0^\infty xe^{-\lambda x}\,dx$ is a standard integral with

$$\int_0^\infty x e^{-\lambda x}\, dx = \frac{1}{\lambda^2}, \text{ for } \lambda > 0$$

□

We can use the unreliability CDF to construct the relibability CDF:

$$\begin{aligned}
\mathbb{P}[\text{fail after } t] &= 1 - \mathbb{P}[\text{fail before } t] \\
&= 1 - \left(1 - e^{-\lambda t}\right) \\
&= e^{-\lambda t} \\
&= \left(e^{-\lambda}\right)^t \qquad (\text{write } e^{-\lambda} = \alpha) \\
&= \alpha^t
\end{aligned}$$

Intuitively, we see that the system "survives" each time unit, with probability $\alpha = e^{-\lambda}$. We can substitute any positive real number for t, but the computations can get quite difficult by hand.

Theorem 2 (Exponential distribution is memoryless)**.** An exponential distribution is memoryless. Informally: the current age of the system says nothing about its future reliability. If you have a system that has been running for 20 years, and one that has been running for 5 years, their failure probabilities are equal. Formally:

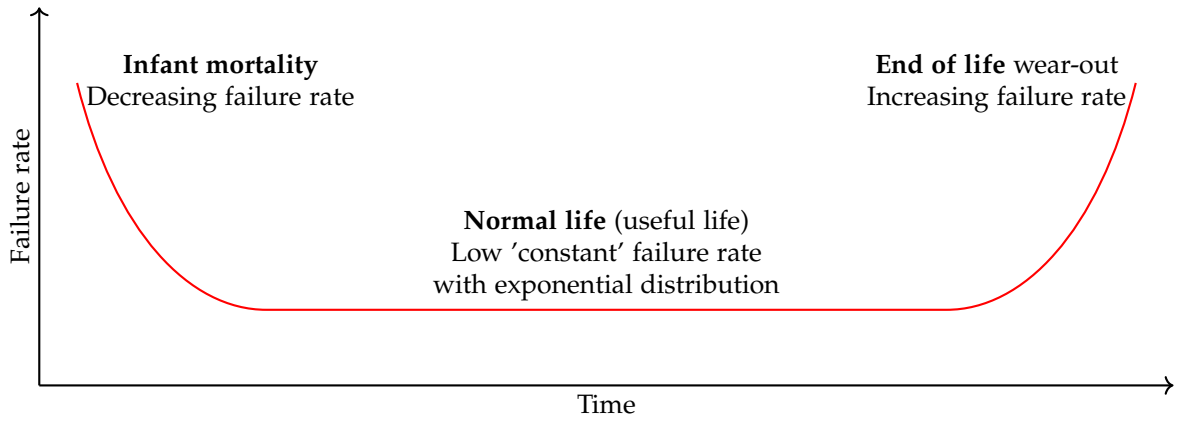
$$\underbrace{\mathbb{P}\left[X > t + s \mid X > s\right]}_{\substack{\text{Given that the system} \\ \text{was operational for } s \\ \text{time units, remain operational} \\ \text{for at least } t+s \text{ more units}}} = \underbrace{\mathbb{P}\left[X > t\right]}_{\substack{\text{Remain operational} \\ \text{for } t \text{ more time units,} \\ \text{independent of } s}}$$

Proof. Well, let $\alpha = e^{-\lambda}$. Then $\mathbb{P}\left[X > t\right] = \alpha^t$, and

$$\begin{aligned}
\mathbb{P}\left[X > t + t' \mid X > t'\right] &= \frac{\mathbb{P}\left[X > t + t' \wedge X > t'\right]}{\mathbb{P}\left[X > t'\right]} \\
&= \frac{\mathbb{P}\left[X > t + t'\right]}{\alpha^{t'}} \\
&= \frac{\alpha^{t+t'}}{\alpha^{t'}} \\
&= \alpha^t = \mathbb{P}\left[X > t\right]
\end{aligned}$$

Clearly, $\mathbb{P}\left[X > t + t' \mid X > t'\right] = \mathbb{P}\left[X > t\right]$, and the property holds. □

Naturally, not all systems have a "pure" exponential failure distribution. They tend to follow the **bathtub curve**:



The **bathtub curve** is a model for degradation behaviour of systems.

- The failure rate is very high at the beginning due to production errors
- Then its quite low during normal usage. This is the important part for us in this course.
- It ends being quite high during end of life wear out

3.2 Dynamic fault trees

Definition 11 (Dynamic fault tree)**.** A dynamic fault tree is a static fault tree with more possible gates:

- The priority AND (PAND) gate fails only when all its children fail *in order*. If one of its children fail *before* a child to the left of it fails, the PAND gate cannot fail.

  Example. Consider a system with two power supplies. There are two situations where the system fails:

  - The primary and secondary power supply fail. We can model this with a simple AND gate.
  - The power supply switch fails, and then the primary power supply fails. The secondary power supply still works, but as the switch has failed, the system cannot switch to it, and it fails.

  The reverse order does not result in failure: if the primary power supply fails, the system switches to the secondary power supply. If the switch fails afterwards, there is no system failure, as it is already using the secondary power supply.

  Remark that the model relies on two assumptions: (1) the switch takes place instantly, and (2) nothing else happens afterwards.

- The SPARE gate indicates that there is some spare basic event that can take on the function of the primary basic event. Some remarks:

  - The failure rate of the inactive spare is usually different from the failure rate of the primary.
  - When the spare takes on the role of the primary, it also takes on its failure rate.

  Example. Consider a car with four tires and a spare tire. The failure rate of the spare tire is lower than that of the tires actively in use, as it does not wear down due to road contact. However, when one of the tires is replaced by the spare tire, its failure rate is now equal to the tire it replaced, as it experiences the same wear.

- The functional dependency (FDEP) gate models a dependency between events. If the dependency event fails, the dependent events immediately fail. This gate is syntactic sugar; it can always be replaced with one or more AND or OR gates.

  Example. Consider a complex system with many subsystems relying on utilities like gas, water and electricity. Instead of adding OR gates to every subsystem to model its reliance on one or more of the utilities, add one FDEP gate per utility and connect the subsystems relying on a utility as dependents of this FDEP gate.

3.2.1 Qualitative analysis

A **cut sequence** is an ordered set of failure events $(e_1, e_2, \ldots, e_n)$. When modelling failure in dynamic fault trees, we use cut sequences instead of events, to account for the ordering requirements in priority AND gates.

3.2.2 Quantitative analysis

Binary Decision Diagrams do not work on dynamic fault trees. We instead use (continuous) Markov chains:

1. Create a (continuous) Markov chain based on the dynamic fault tree. This is an iterative process:
   (a) Start with the state where no events have failed yet.
   (b) Create edges and vertices for every basic event in the system. Label the edges with the failure rate of the basic event.
   (c) Continue for every combination of failure. Recursion stops at nodes where failure or lack of failure is guaranteed.
2. Simplify the Markov chain.
   - Merge failed states, guaranteed no-fail (green) states, and equivalent states:

     $$u \xrightarrow{\alpha} \text{fail}_1, v \xrightarrow{\beta} \text{fail}_2 \Longrightarrow u \xrightarrow{\alpha} \text{fail}, v \xrightarrow{\beta} \text{fail}$$

   - Merge double edges, which happens when there are two distinct edges with identical source and target vectors. Merge them to a new edge with the probabilities summed:

     $$u \xrightarrow{a} v, u \xrightarrow{b} v \Longrightarrow u \xrightarrow{a+b} v$$

   - Eliminate self-loops. A self-loop occurs when a state transitions to itself. These transitions do not impact the failure time of a system and can be safely removed:

     $$u \xrightarrow{\alpha} u \Rightarrow \emptyset$$

3. Derive the **transition matrix**. The transition matrix is a $n \times n$ matrix, with n the number of states in the Markov chain. An element in row r and column c represents the transition label (which is a failure rate) from state r to state c in the Markov chain. How the states are labeled does not really matter, as long as you give the failure state the highest label.

   $$A = \begin{pmatrix} \lambda_{s_1 \to s_1} & \lambda_{s_1 \to s_2} & \lambda_{s_2 \to s_3} & \cdots & \lambda_{s_1 \to s_n} \\ \lambda_{s_2 \to s_1} & \lambda_{s_2 \to s_2} & \lambda_{s_2 \to s_3} & \cdots & \lambda_{s_2 \to s_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \lambda_{s_n \to s_1} & \lambda_{s_n \to s_2} & \lambda_{s_n \to s_3} & \cdots & \lambda_{s_n \to s_n} \end{pmatrix}$$

   If there is no transition from state s_i to state s_j, then we define $\lambda_{s_i \to s_j} = 0$.

   The elimination of self-loops would lead to the assumption that $\lambda_{s_k \to s_k} = 0$ in the transition matrix. However, for the sake of the following computations, we define $\lambda_{s_k \to s_k}$ as the inverse of sum of its row:

   $$\lambda_{s_r \to s_r} = -\sum_{k=1}^{n} A_{rk}$$

   This ensures that the following property holds:

   $$\sum_{k=1}^{n} A_{rk} = 0$$

4. Compute the **probability vector**. The probability vector is defined as follows:

$$
\begin{aligned}
&p(t) = \text{probability vector of system state at time } t \\
&p(0) = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \end{pmatrix} \\
&p(t) = p(0) \cdot e^{tA} \\
&p(t) = p(0) \cdot M^t \text{ with } M = e^A, \text{ iff } t \in \mathbb{Z}
\end{aligned}
$$

The size of the probability vector is equal to the amount of vertices in the Markov chain. One can find $\mathbb{U}(t)$ by looking at the element in the vector representing the failure state.

Computing tA is easy; multiply each value in the matrix by t. Computing e^{tA} is not so easy:

$$e^X = \sum_{k=0}^{\infty} \frac{1}{k!} X^k$$

N.B. The exact formulas are not important. The computation can be done with any mathematics workbench, e.g. Wolfram Alpha and SciPy. One can simply search for `matrix exponential [favorite tool]` with Google.

3.2.3 Deep compositionality

Composing a Markov chain from the entire fault tree is generally difficult. It gives huge Markov chains that are difficult to validate and modify. Instead, we make components and compose them using smart composition. This works using interactive Markov chains.

Part II

Testing

Lecture 4

Testing in software engineering

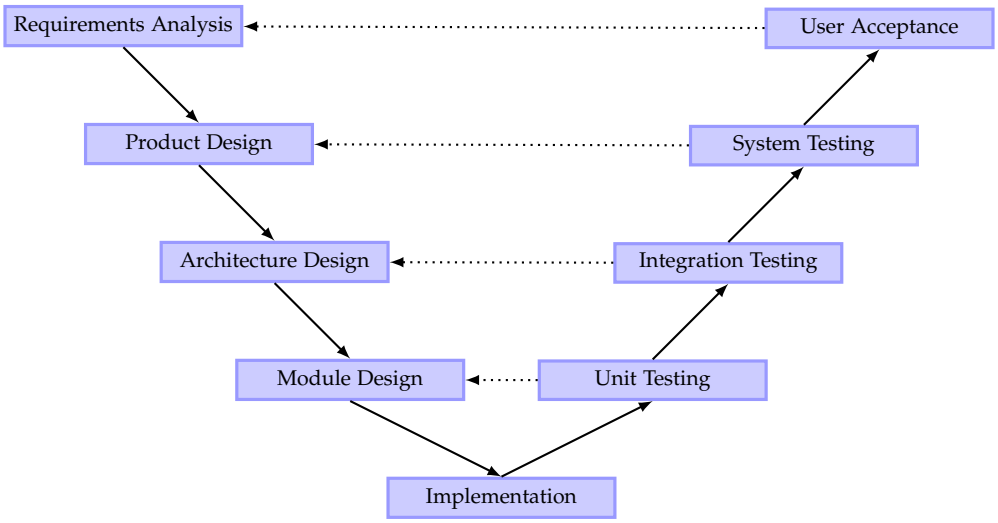
4.1 Testing basics

We test, because it is impossible to write code without bugs, and bugs cause problems for users.

A **test** consists of:

- Setup (if needed)
- Operations/actions to execute program/system
- A check whether expected result has been obtained — this gives a *pass* of *fail* in the test execution

The **V-model** graphically displays the relation between tests, the development cycle, and the design process:



The **Agile development method** relates to testing: there are no big upfront steps, we just write code and perform tests. There is a focus on quality, measured through testing. Agile prefers testing first (TDD). And you should specify your requirements through scenarios.

A test can **target** one of many things:

- functionality
- security
- performance
- usability
- reliability
- . . . and more

We discern different **levels of automation** in testing:

- Manual testing: manually create tests, and manually execute them.

- Automated testing: manually create tests, and automatically execute them.
  - With recorded tests,
  - With scripted testing,
  - With keyword-driven testing, or,
  - By testing as part of behavior-driven development.
- Model-based testing: manually create a specification, automatically derive tests with an algorithm, and automatically execute them. Mostly applied on the system level (black box). We will later see that there is a conformance relation (IOCO) between the SUT (system under test) and the model.

Generally, the development of hardware outgrows the development of software, which outgrows the development of tests.

Testing can only detect the presence of errors, not their absence. Thus it is not complete. We can increase our confidence, but we can never say: I have tested my software, every test passes, ergo my code does not contain bugs.

4.2 Regression testing

A **regression** is an unexpected behavior after a change in the code. **Regression testing** is testing after every code change. Why? Bugs that are detected earlier are easier to fix.

After implementing a new feature, regression testing verifies that other behavior of the system stays the same. After refactoring, regression testing verifies that nothing in the system behavior has changed.

The **advantages of regression testing** are as follows:

- Make behavior of code observable
- Monitor functioning of the software system
- Detect introduced errors early
- Manual testing takes more effort

Regression testing is just a form of automated testing; can be done with any testing framework. It can (should) be integrated in CI pipelines.

4.3 Test-driven development

The idea of **test-driven development** (TDD) is to write tests before code is written. The procedure is as follows:

1. Write test for new functionality
2. Run test
3. Check result: test should fail. Pass? Back to step 1.
4. Implement to obtain functionality
5. Run all tests: check if no regression
6. Check result:
   - Pass? Done!
   - Fail? Go back to step 4, or adapt the test and go to step 5.

The **laws of test-driven development** are as follows:

1. No code is written before the failing unit test is written.
2. No more tests are written than the amount that is sufficient to fail.
3. No more code is written than the amount that is sufficient to pass the tests.

The **challenges of test-driven development** are as follows:

- TDD is a discipline, which can be difficult to thoroughly follow.
- TDD needs some design details for the test interface.
- The expected result of a test might be unknown beforehand.

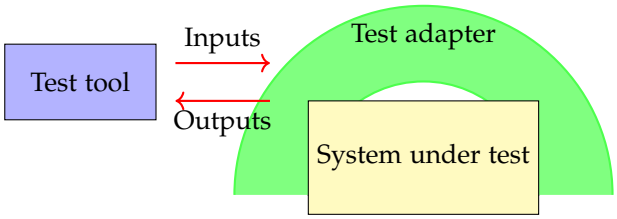
TDD is a code **design** strategy:

- We think of the what before the how.
- We actively look for issues, instead of confirming (happy flow) expectations.
- The quality is increased by small incremental changes.

TDD is traditionally applied for unit tests, but it is also possible for integration and unit tests.

4.4 System testing with behavior-driven-development

Generally, some connection is needed between the **system under test** (SUT) and the **test tool**. This is the **test adapter**. It performs the test using inputs from the test tool, and reports the outputs of the system back.



A **system test** on functionality — which this course is about — tests the functionality/behavior/feature at a system level. It tests the interaction between the user and the system. It has a black-box assumption; inputs and outputs are provided on a system level.

Functionality can be formulated in many ways:

- Feature/Requirement description
- User stories
- Behaviour-Driven Development scenario
- Model
- Property

Definition 12 (User story)**.** A user story describes system functionality as follows:

- As a type of user,
- I want some goal,
- So that reason (optional).

Example 2. As a student, I want to search for the contact details of a UT employee.

Behavior-driven-development (BDD) extends the Agile framework. It facilitates collaboration between different roles and a shared understanding of what's to be implemented.

Example 3. Scenario: login to see email.

- *Given* a user has an email account
- *When* the user logs in with valid credentials
- *Then* the system displays the 20 most recent emails of the user

There is still natural language, but there is some structure, and you can easily derive the test case.

Definition 13 (BDD Scenario)**.** A BDD scenario is an instance of required software behavior (specification by example). It specifies user interaction with the system. Quite some work, so prioritize essential use

cases for the system.

It is described as follows:

- **Given** precondition,
- **When** action(s),
- **Then** postcondition or resulting action.

A BDD scenario can also be **parameterized** for more flexibility:

- **Given** a player has loaded the level "move",
- **When** the player pressed key,
- **Then** the player moves direction.

The scenarios are generated by giving the parameters values, for example:

key	direction
w	forward
a	left
s	backwards
d	right

The **steps in behavior-driven development** are as follows:

1. Start from the agile approach: choose feature/user story/requirement from sprint.
2. Discovery: Brainstorm and create concrete examples. Ensures communication and shared understanding.
3. Formulation: Write scenarios in Given-Then-When-Style. Possibly multiple scenarios in a feature.
4. Automate: create tests from scenarios. Living documentation.

In BDD, there is a **common understanding** between product owner, developer and tester:

- Product owner:
  - Goal: achieve business goal,
  - Contribution: sees what the product should be for user.
- Developer:
  - Goal: achieve implementation of product,
  - Contribution: sees technical possibilities and obstacles.
- Tester:
  - Goal: achieve testability of product,
  - Contribution: sees edge cases.

To go from a BDD scenario to a test case, tools should be used to generate bindings for the scenario steps. Developers can then give meaning to the steps (given/then/when). The tool can then use the bindings to explicitly show the scenario steps.

The **challenges in behavior-driven development** are as follows:

- It is often difficult to bring the system to the 'given' step.
- Natural language is ambiguous. Implementing the bindings is thus difficult.
- It is often difficult to check the 'then' step.

There is some existing **research on behavior-driven development for model-based testing**. Guidelines have been set up for a more specific BDD scenario format. Those scenarios can be translated to smaller MBT models. These smaller models are then composed into a large model, and finally model-based test generation algorithms can be used to derive tests.

4.5 Classical black-box testing

In **black-box testing**, we do not know what the system looks/works like internally ("under the hood"). Consequently, we cannot know what errors it has. There are multiple **approaches on error guessing**:

- Just 'guess' where the errors are — not smart and not useful.
- Rely on the intuition and experience of the tester.
- Employ a strategy:
  - Make a list of possible errors and error-prone situations, e.g. edge cases.
  - Write test cases based on this list.
- Applying risk analysis.
- Identifying critical parts of the program:
  - Parts with unclear requirements,
  - Parts that are developed by un(der)experienced programmer(s), and,
  - Complex code according to software metrics.
- Writing more and more thorough tests for high-risk code.

Equivalence partitioning is a strategy where all possible inputs for a (sub)system are divided into a finite number of equivalence classes (partitions). For every equivalence class, an (arbitrary) input (a **representative**) is chosen for a test case.

Equivalence classes should be chosen, such that we can assume that:

- The function behaves analogously for inputs in the same class.
- One test with one input from each class is sufficient.
- If the representative causes a fault, then others in the same class would do so as well.

With **boundary value analysis** we also test inputs on, directly above or directly below equivalence class boundaries. We also consider output boundaries. These boundary values are combined with arbitrary representatives from equivalence partitioning.

Picking random values for **random testing** is easy for numbers, using random number generators. It gets considerably more difficult as valid inputs get more complex and/or get more (complex) constraints.

Example 4 (Equivalence partitioning on a `sum` program)**.** Consider a program that does the following:

- It computes the sum of the first `N` integers.
- If `N` is negative, it takes the absolute value $|\texttt{N}|$.
- If the sum is greater than than `max`, an error should be reported.

Formally: for any $\texttt{N}, \texttt{max} \in \mathbb{Z}$:

$$\texttt{sum(N, max)} = \begin{cases} \sum_{k=0}^{|\texttt{N}|} k & \text{if} \sum_{k=0}^{|\texttt{N}|} \leq \texttt{max} \\ \texttt{error} & \text{otherwise} \end{cases}$$

We can derive some equivalence classes for the input parameters:

- Providing values for both parameters
- Providing too few or too many values
- Integers
- Non-integers
- Positive values
- Negative values

- Zero
- Some combination resulting in a sum greater than `max`

The equivalence classes for the output parameters are values for each of the cases. From the formal definition, we derive that these classes are $\{x \in \mathbb{Z} \mid x \geq 0\}$ and `error`.

The edge cases in this example include (but are not limited to):

- Empty sum, i.e. $\texttt{N} = 0$
- Absolute values when $\texttt{N} = 0$ or $\texttt{N} = 1$
- A value for `max`, such that $\texttt{result} = \texttt{max}$, or $\texttt{result} = \texttt{max} + 1$
- Integer overflow boundaries
- ...

4.6 Classical white-box testing

In **white-box testing**, we know what the system looks/works like internally ("under the hood"). Testing is done based on internals: the structure of the code. It is heavily dependent on the programming languages. Code coverage is measured by test cases.

Coverage can be examined using a **flow diagram**. The flow diagram is a visual representation of the code. Statements are represented by vertices of various shapes. Directed edges indicate that the code *might* go from one statement to another statement.

- The **statement coverage** is the amount of vertices that are visited by the entire test suite.
- The **branch coverage** or **decision coverage** is the amount of edges that are visited by the test suite.
- The **path coverage** is the amount of paths that are visited by the test suite.

A test suite can be considered complete when it has 100% path coverage. A directed graph may contain infinite paths, which proves that not all test suites can be complete.

4.7 Integration testing

With **integration testing**, we test the integration of components. The exact definition of a component depends on the system and the programming language. In Java, a component is a class or a set of classes, and a component interface is the set of public methods used by other components. Integration is considered successful when it is used as expected by other components.

Generally, integration testing is performed as follows:

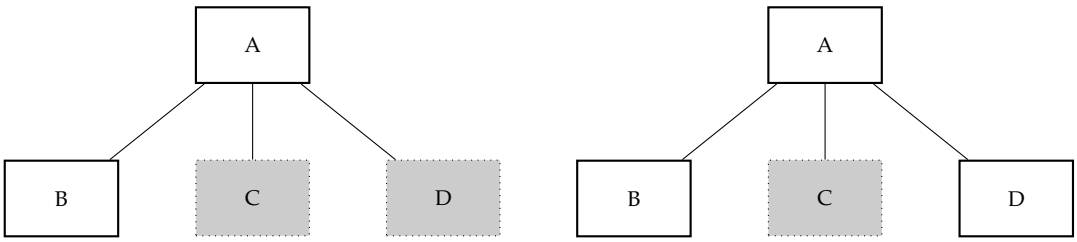
1. Components are combined incrementally.
2. Each increment is tested with integration tests.
3. Bugs of failing integration tests are fixed.
4. The combinations are incremented until all components have been combined.

A **test driver** is a test program that calls the components to be tested. It provides input data, and returns test results.

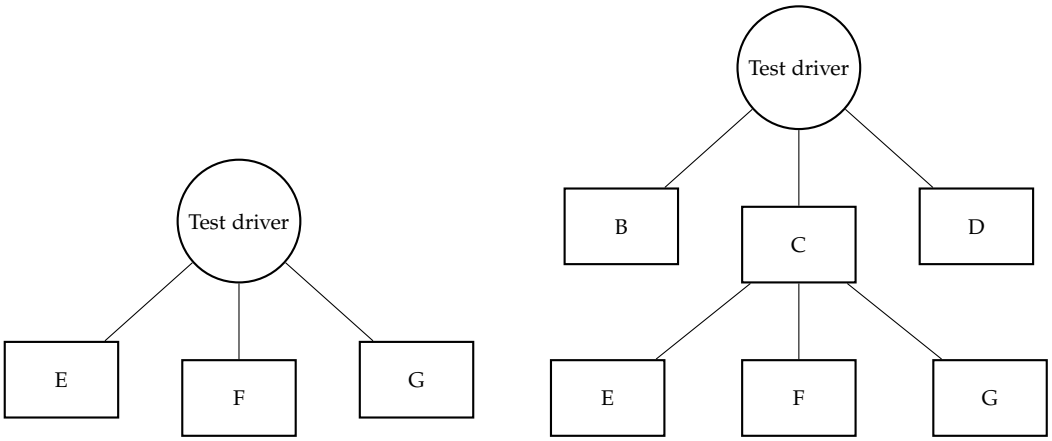
A **test stub** is a dummy program that receives calls from components. It emulates a non-implemented component.

There are multiple **approaches to perform integration testing**:

- **Top-down**: stubs are used for components not yet under testing.



- **Bottom-up**: the test driver replaces a subcomponent and tests its descendant components.



- **Sandwich**: use a mix of stubs and drivers.
- **Big-bang**: put all components together and test the whole system. Essentially skips integration testing; thus not recommended.

4.8 Testability of software

The design of software influences its testability. This is especially true for integration testing.

Dependency injection is a method to improve the design of software for integration testing. Dependencies of a class are injected, with the purpose of separating configuration from use. This achieves a separation of concerns, loose coupling, and the flexibility that is required for stubbing.

Lecture 5

Model-based testing with labeled transition systems

5.1 Model-based testing

Recap: specifcy the system with a model, algorithmically derive tests, execute them on the system under test.

What kind of testing with MBT?

- Black-box testing; on a system level
- Functional behavior and interaction of the system

What kind of systems?

- Systems with a software interface
- Discrete event-driven systems
- Reactive, dynamic systems
- Data-intensive systems
- Real-world example systems: ASML wafer machine, Canon industrial printer

What kind of model? Well, in this course:

- Labeled transition systems (today and next lecture)
- Symbolic transition system (lecture 6)
- AML model (Axini) (lecture 7 — not tested on the exam)

Models for testing are not the same as models for constructing software. Testing models provide abstraction. Their smaller size makes them easier to maintain. They may also be partial — we only model the relevant/interesting part.

5.2 Labeled transition systems

Definition 14 (Labeled transition system with inputs, outputs and quiescence)**.** A **labeled transition system** is a 5-tuple (Q, L_I, L_U, T, q_0):

- Q: set of states
- L_I: set of input labels
- L_U: set of output labels
- Special label for quiescence: $\delta \notin L_I \cup L_U$
- $T \subseteq Q \times \{L_i \cup L_U \cup \{\delta\}\} \times Q$: transition relation
- q_0: initial state

5.3 Quiescence

Quiescence is the absence of output. The SUT will not produce any output until the next input arrives, no matter how long the tester waits. Informally: any state that only has outgoing input transitions is quiescent. Formally:

$$q \in Q \wedge \forall_{x \in L_U} \forall_{q' \in Q} : (q, x, q') \notin T \Rightarrow Q \text{ is quiescent}$$

5.4 Nondeterministic LTS

Two ways to make an LTS nondeterministic:

- FDA is deterministic, but you have no control over output
- FDA is nondeterministic; multiple identical output transitions, no way to know which one

5.5 Formal definitions

- Let $S = (Q, L_I, L_U, T, q_0)$ be an LTS.
- Let $L = L_I \cup L_U \cup \{\delta\}$.
- Let $q \in Q$ be a state, $\mu \in L$ a label, and $\rho \in L^*$ a sequence of labels.
- A state q is **quiescent**, denoted $\delta(q)$, if:

  $$\forall_{x \in L_U}, \forall_{q' \in Q} : (q, x, q') \notin T$$

- We define after : $Q \times L^* \to \mathbb{P}(Q)$, with:
  - $q \text{ after } \epsilon = \{q\}$
  - $q \text{ after } \mu\rho = Q' \text{ after } \rho$, where

    $$Q' = \{q' \in Q \mid (q, \mu, q') \in T\} \cup \{q \mid \delta(q) \wedge \mu = \delta\}$$

  - Extend to after : $\mathbb{P}(Q) \times L^* \to \mathbb{P}(Q)$:
    - Let $Q' \subseteq Q$.
    - $Q' \text{ after } \rho = \bigcup_{q \in Q'} q \text{ after } \rho$

    Informally: q after ρ is the set of states the LTS can be 'in' after observing/supplying the sequence of stimuli and responses ρ.
- We define Straces : $Q \to \mathbb{P}(L^*)$, with:

  $$\text{Straces}(q) = \{\rho' \in L^* \mid q \text{ after } \rho' \neq \emptyset\}$$

  Informally: Straces(q) is the set of all possible sequences of observable interactions that the LTS can legally perform, starting from state q.
- in defines all ingoing transitions to (sets of) states:
  - $\text{in}(q) = \{a \in L_I \mid q \text{ after } a \neq \emptyset\}$
  - $\text{in}(Q') = \bigcup_{q \in Q'} \text{in}(q)$
- out defined all outgoing transitions from (sets of) states:
  - $\text{in}(q) = \{x \in L_U \cup \{\delta\} \mid q \text{ after } x \neq \emptyset\}$
  - $\text{in}(Q') = \bigcup_{q \in Q'} \text{in}(q)$
- An LTS S is deterministic if

  $$\forall_{q \in Q}, \forall_{\rho \in \text{Straces}(q)} : |q \text{ after } \rho| \leq 1$$

  Informally: the LTS is determinstic if it is always in one state after any sequence, or, if every state only has up to 1 transition for every possible behavior.

5.6 Determinisation

1. Make quiescence explicit
2. Transform NFA to DFA

5.7 Test cases

A test case specifies:

- Inputs to provide to the SUT
- Outputs to expect from the SUT
- A verdict (pass or fail)
- A finite execution of the SUT. When the test case stops, it returns its verdict.

Definition 15 (Test case)**.** A test case for an LTS S is an LTS $t = \left(Q^t, L_I, L_U, T^t, q_0^t\right)$, such that:

- There are two special states $Pass, Fail \in Q^t$
- States *Pass* and *Fail* have *implicit* self-loops for outputs, including δ:

$$\forall_{x \in L_U \cup \{\delta\}} : Pass \text{ after } x = \{Pass\}$$
$$\forall_{x \in L_U \cup \{\delta\}} : Fail \text{ after } x = \{Fail\}$$

- States *Pass* and *Fail* have no transition for inputs:

$$\forall_{a \in L_I} : Pass \text{ after } a = Fail \text{ after } a = \emptyset$$

- t has no cycles, except those in *Pass* and *Fail*.
- t is deterministic.
- Every state enables all outputs L_U, and either one input or δ:

$$\forall_{q \in Q} : (|\text{in}(q)| = 0 \wedge \text{out}(q) = L_U \cup \{\delta\}) \vee (\text{out}(q) = L_U \wedge |\text{in}(q)| = 1)$$

5.8 Distinguishing and homing test cases

A **distinguishing test case** distinguishes two start states.

A **homing test case** is a test case that transitions to a particular state from any starting state. It helps reset the system to a known state.

Lecture 6

Test generation & ioco

6.1 Batch test generation

Definition 16 (Algorithm *batchGen*)**.** The algorithm *batchGen* works as follows.

Input: An LTS $S = (Q, L_I, L_U, T, q_0)$, and current states $Q' \subseteq Q$.

Output: A test case for S.

Defined recursively. To execute $batchGen(S, Q')$, take either of the following steps:

- End the test case: return *Pass*.
- Observe output: create branches for every possible response, including quiescence (δ).
  - For forbidden outputs, simply end the branch with *Fail*.
  - For allowed outputs, supply the branch with the result of $batchGen(S, Q' \text{ after } x!)$.
- Supply some input a?, where $Q' \text{ after } a? \neq \emptyset$. Create branches for every possible response (the SUT can still respond!), including quiescence, *and* the supplied input.
  - For forbidden outputs, simply end the branch with *Fail*.
  - For the supplied input, supply the branch with the result of $batchGen(S, Q' \text{ after } a?)$.
  - For allowed outputs, supply the branch with the result of $batchGen(S, Q' \text{ after } x!)$.

Remark. See slide 7 of lecture 5 for a graphical representation.

The *batchGen* algorithm is non-deterministic. The normal usage of *batchGen* is to store the produces test cases for execution.

6.2 On-the-fly test generation

Definition 17 (Algorithm *onTheFlyGen*)**.** The algorithm *onTheFlyGen* works as follows.

Input: An LTS $S = (Q, L_I, L_U, T, q_0)$, and current states $Q' \subseteq Q$.

Output: A verdict *Pass* or *Fail*.

Defined recursively. To execute $onTheFlyGen(S, Q')$, take either of the following steps:

- End the test case: return the verdict *Pass*.
- Observe output:

$$onTheFlyGen(S, Q') = \begin{cases} onTheFlyGen(S, Q' \text{ after } x!) & \text{if } x! \in \text{out}(Q') \\ Fail & \text{otherwise} \end{cases}$$

- Choose some input a?, where $Q' \text{ after } a? \neq \emptyset$.

If the SUT provides an output x! already, then handle its observation as described above. Otherwise, supply a? and continue with $onTheFlyGen(S, Q' \text{ after } a?)$.

On-the-fly test generation is very intuitive; it works just like a human tester would explore/test a SUT with an LTS.

6.3 Test assumptions on SUT for ioco

The **test hypothesis** is that an SUT can be modeled as an LTS (or, later, an STS). We assume that the SUT accepts any input (unless it is providing an output). This is modeled with an **input-enabled LTS (IELTS)**.

Formally: an LTS $S = (Q, L_I, L_U, T, q_0)$ is input-enabled if $\forall_{q \in Q} : \text{in}(q) = L_I$, i.e. every state accepts any input.

Optionally, we can assume fairness: if the SUT *can* provide an output, it *will* eventually.

6.4 ioco

Let $S = (Q, L_I, L_U, T, q_0)$ be an LTS, and let $\rho \in L^*$ be a sequence of observable behavior. Then:

- $\text{Straces}(S) = \text{Straces}(q_0)$
- $S \text{ after } \rho = q_0 \text{ after } \rho$
- $\text{out}S = \text{out}(q_0)$
- $\text{in}S = \text{in}(q_0)$

Definition 18 (ioco)**.** IOCO means Input Output COnformance. It is a relation between an IELTS and an LTS: **ioco** : IELTS $\times$ LTS $\to$ Bool.

We write: $\mathcal{I}$ **ioco** $\mathcal{S}$. This means:

- Implementation $\mathcal{I}$ allows the same or more inputs than $\mathcal{S}$, and,
- Implementation $\mathcal{I}$ provides the same or fewer outputs than $\mathcal{S}$.

More informally: the implementation has implemented *at least* what was requested by the specification, but it may have more features.

Formally:

- Let $\mathcal{S} = \left(Q^{\mathcal{S}}, L_I, L_U, T^{\mathcal{S}}, q_0^{\mathcal{S}}\right)$ be an LTS.
- Let $\mathcal{I} = \left(Q^{\mathcal{I}}, L_I, L_U, T^{\mathcal{I}}, q_0^{\mathcal{I}}\right)$ be an IELTS. Remark how these LTSs share the same label sets.
- Then **ioco** : IELTS $\times$ LTS $\to$ Bool is defined as:

$$\mathcal{I} \textbf{ ioco } \mathcal{S} = \forall_{\rho \in \text{Straces}(\mathcal{S})} : \text{out}(\mathcal{I} \text{ after } \rho) \subseteq \text{out}(\mathcal{S} \text{ after } \rho)$$

If $\mathcal{I}$ can produce an output x! after ρ, then $\mathcal{S}$ can produce the output x! after ρ as well.

6.5 Soundness and exhaustiveness

A test case t for an LTS S is **sound** if, for any IELTS I, I **ioco** S implies that execution of t on I yields *Pass*.

Alternatively, a test case t for an LTS S is **sound** if, for any IELTS I, execution of t on I yielding *Fail* implies that I ~~ioco~~ S.

A test suite TS is **sound** if all its test cases $t \in TS$ are sound. Informally: the test suite will pass on any conformant implementation.

A test suite TS for an LTS S is **exhaustive** if, for any IELTS I, I ~~ioco~~ S implies that there is a $t \in TS$, such that execution of t on I yields *Fail*. Informally: the test suite detects any non-conformant implementation by having a test fail on any non-conformant implementation.

A test suite TS is **complete** for an LTS S if it is sound and exhaustive for S. Realistically, this is not achievable for most specifications.

6.6 Explicit underspecifcation

Let $S = (Q, L_I, L_U, T, q_0)$ be an LTS. Then a? $\in L_I$ is **underspecified** in $q \in Q$ if a? $\notin$ in$Q$. Informally: an input is underspecified if not every state in the LTS defines what should happen for that input.

There are two approaches to 'fixing' this issue:

- **Demonic completion**: add a **chaos state** χ, where anything is allowed. Add a transition to χ for all underspefified inputs.
- **Angelic completion**: add a self-loop for all underspecified inputs. This approach does not preserve ioco.

Lecture 7

Symbolic Transition Systems

7.1 Symbolic Transition Systems

Symbolic transition systems (STS) allow for modelling real systems, with control flow (states and actions) and data (variables, parameters, conditions, et cetera).

An STS consists of:

- **Locations.** These are equivalent to states in LTS.
- **Initial assignment.** Assigns values to location variables.
- **Input gates.** Require a certain input to be present before advancing to another location.
- **Output gates.** Require a certain output to be observed before advancing to another location.
- **Gate parameters.** Values that are derived from inputs and outputs, in essence making the behavior parameterized.
- **Guards.** Boolean expressions. A guard must be true in order for a transition to be valid.
- **Assignments to location variables.** These assignments take place after a transition has been taken.
- **Switches.** In essence, transitions.

Formally:

Definition 19 (Symbolic Transition System)**.** A Symbolic Transition System (STS) with inputs and outputs is a tuple $(\mathcal{L}, l_0, \mathcal{V}_l, \text{ini}, \mathcal{V}_p, \Gamma_I, \Gamma_O, \mathcal{R})$ where:

- $\mathcal{L}$ is a finite set of **locations**,
- $l_0 \in \mathcal{L}$ is the **initial location**,
- $\mathcal{V}_l$ is a finite set of **location variables**,
- $\text{ini} \in \mathcal{T}(\emptyset)^{\mathcal{V}_l}$ is the **initialization**,
- $\mathcal{V}_p$ is a finite set of **gate parameters**, such that $\mathcal{V}_p \cap \mathcal{V}_l = \emptyset$,
- Γ_I is a finite set of **input gates**,
- Γ_O is a finite set of **output gates**, such that $\Gamma_I \cap \Gamma_O = \emptyset$, and,
- $\mathcal{R} \subseteq \mathcal{L} \times (\Gamma_I \cup \Gamma_O) \times \mathcal{V}_p^* \times \mathcal{T}_{\text{Bool}}(\mathcal{V}_I \cup \mathcal{V}_p) \times \mathcal{T}(\mathcal{V}_I \cup \mathcal{V}_p)^{\mathcal{V}_i} \times \mathcal{L}$ is the **switch relation** with a finite number of elements.

  Every element in this set is a tuple $(l_1, \lambda, p_0 \dots p_k, \phi, \psi, l_2)$, with:

  - l_1 the **source location**,
  - λ the gate, e.g. p? or q! (just one!),
  - $p_0 \dots p_k$ the list of **parameters**,

- ϕ the **guard**,
- ψ the **assignment**, and,
- l_2 the target location.

The switch can only be traversed if we are at location l_1, the behavior λ is observed, and the guard ϕ is satisfied. By traversing the switch, the assignment ψ is executed, and the STS is now at location l_1. Remark that every switch models the behavior of just one interaction, at the granularity of a single stimulus or response.

Informally, an STS is an LTS with variables, allowing for more complex behavior to be modeled. We keep track of global variables (i.e. location variables) and local variables (i.e. guard parameters). In an STS, switches (transitions) are annotated not only by the required (sequence of) input or output, but also by some boolean condition that is to be satisfied, and some assignment to the location variables.

A switch in an STS can only be taken when the required behavior is observed *and* the boolean condition is satisfied. The boolean condition can impose requirements on location variables, but also on data supplied by the behavior, in the form of gate parameters. Remark that these requirements are bidirectional: we can require something for the input of the system, but also require (or perhaps rather guarantee) something for its output.

Finally, some details on the definition:

- Assignments on switches happen simultaneously:

$$x := a, y := b \equiv y := b, x := a$$

- If a location variable is not assigned by a switch, its value stays the same, i.e. $c := c$.
- Location variables are global to the whole STS. Gate parameters are local to guards and assignments of a switch and cannot be referenced elsewhere. Consequently, if the value of a gate parameter is needed in another switch, it must be stored in a location parameter.
- For any switch $(l_1, \lambda, p_0 \dots p_k, \phi, \psi, l_2) \in \mathcal{R}$, we require that:
  - The list of parameters $p_0 \dots p_k$ is a sequence of distinct variables,
  - $\text{type}_g(\lambda) = \text{type}_t(p_0 \dots p_k)$, i.e. there are no type conflicts between variables,
  - $\phi \in \mathcal{T}_{\text{Bool}}(\mathcal{V}_I \cup \{p_0, \dots, p_k\})$, i.e. the guard ϕ is a boolean expression, and it can only refer to the location variables and the gate parameters.
  - $\psi \in \mathcal{T}(\mathcal{V}_I \cup \{p_0, \dots, p_k\})^{\mathcal{V}_I}$, i.e. the assignment ψ is a function that assigns a new value to each location variable using expressions built from the current location variables, and the gate parameters.

  These requirements are very formal and for semantics; they are quite intuitive.

7.2 Relation between STS and LTS

An STS is a generalisation of an LTS. Any STS can be **interpreted** into an LTS, by replacing symbolic elements with concrete values. This expands into potentially infinite LTSs due to the potentially infinite domain of the variables in the STS.

Formally:

Let S be an STS. Then its **interpretation** is the LTS $S_{\text{intrp}} = (Q, L_I, L_U, T, q_0)$, where:

- $Q \subseteq \mathcal{L} \times \mathcal{U}^{\mathcal{V}_I}$ is a set of states,
- $L_I, L_U \subseteq \Gamma \times \mathcal{U}^*$ are set of labels called **gate values** such that $\text{type}_g(\lambda) = \text{type}_v(\overline{w})$

Consequently, the interpretation S_{intrp} has states $(l, \alpha) \in Q$, where:

- $l \in \mathcal{L}$ is a location,
- $\alpha \in \mathcal{U}^{\mathcal{V}_I}$ is an assignment of values to location variables.

It also has labels $(\lambda, \overline{w}) \in L$, where:

- $\lambda \in \Gamma$ is a gate,
- $\overline{w} \in \mathcal{U}^*$ is a sequence of values.

7.3 On-the-fly test generation and execution for STS

Definition 20 (Algorithm *onTheFlyGen*)**.** The algorithm *onTheFlyGen for an STS* works as follows.

Input: An STS $S = (\mathcal{L}, l_0, \mathcal{V}_l, \text{ini}, \mathcal{V}_p, \Gamma_I, \Gamma_O, \mathcal{R})$, and current states $Q' = \{(l_0, \text{ini})\}$.

Output: A verdict *Pass* or *Fail*.

Defined recursively. To execute $onTheFlyGen(S, Q')$, take either of the following steps:

- End the test case: return the verdict *Pass*.
- Observe output:

$$onTheFlyGen(S, Q') = \begin{cases} onTheFlyGen\,(S, Q' \text{ after}(\text{x!}, \overline{w})) & \text{if } (\text{x!}, \overline{w}) \in \text{out}(Q') \\ Fail & \text{otherwise} \end{cases}$$

- Choose some input $(\text{a?}, \overline{w})$, where $Q' \text{ after}(\text{a?}, \overline{w}) \neq \emptyset$.

  If the SUT provides an output $(\text{x!}, \overline{w}_2)$ already, then handle its observation as described above. Otherwise, supply $(\text{a?}, \overline{w})$ and continue with $onTheFlyGen(S, Q' \text{ after}(\text{a?}, \overline{w}))$.

This algorithm is essentially the same as for an LTS, but with values for parameters added.

A **test trace** is some sequence of 2-tuples $(\lambda, \overline{w})$ with:

- λ observed behavior, e.g. an input or output,
- $\overline{w}$ the values of the parameters.

Test traces can be used instead of test cases. To execute a test trace, apply it to the SUT. If a deviation is observed, the result is either failure or inconclusive (and the trace must be tried again). If the test trace is observable in the SUT, the trace passes.

7.4 Switch coverage test generation and execution

A **symbolic execution graph** is a tree that models a possible execution of an STS in semi-concrete form: the variables are symbolic (not assigned fixed values yet), and constrained by guards and assignments.

Each vertex is a tuple (l, ψ, ϕ), where:

- l the current location in the STS,
- ψ the mapping of location variables *so far*,
- ϕ a Boolean path condition — the conjunction of all guards and gate constraints seen on the path.

The edges of the symbolic execution graph are the switches in the STS. Before creating a symbolic execution graph, it makes sense to label the switches (e.g. $r_0, r_1, r_2, \ldots$) to easily refer to them in the graph.

A symbolic execution graph has 100% **switch coverage** if every switch is visited by at least one path in the graph.

SMT solvers can be used to find concrete values satisfying all conditions in the symbolic execution graph, thus deriving a concrete test case for the STS.