Universität Bremen

dfki Deutsches Forschungszentrum für Künstliche Intelligenz German Research Center for Artificial Intelligence

Systems of High Safety and Security

Lecture 3 from 29.10.2025: The Development Process

Winter term 2025/26

Christoph Lüth

Organisatorisches

- Die Vorlesung und Übung am 05.11.2025 **fällt aus.**

Software Development Models

Software Development Process

- A software development process is the **structure** imposed on the development of a software product.
- We classify processes according to **models** which specify
  - the artefacts of the development: the software product itself, specifications, test documents, reports, reviews, proofs, plans etc;
  - the different stages of the development;
  - and the artefacts associated to each stage.
- Different models have a different focus: correctness, development time, flexibility.
  - Note you cannot have all three
- What does **quality** mean in this context?
  - What is the **output** — just the software product, or more? (specifications, test runs, documents, proofs. . .)

Artefacts in the Development Process

Planning:

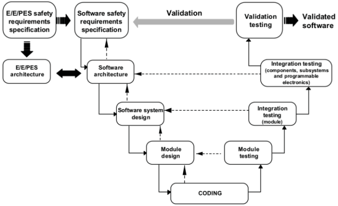
- Document plan
- V&V plan
- QM plan
- Test plan
- Project manual

Specifications:

- Requirements
- System specification
- Module specification
- User documents

Implementation:

- Source code
- Models
- Documentation



Verification & validation:

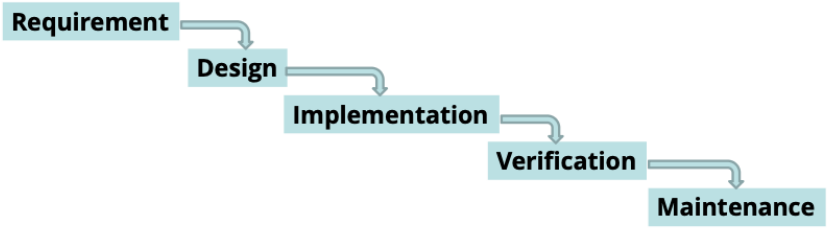
- Code review protocols
- Test cases, test results
- Proofs

Possible formats:

- Documents:
  - Word/LaTeX documents
  - Excel sheets
  - Wiki text
  - Database (Doors)
- Models:
  - UML/SysML diagrams
  - Formal languages: Z, HOL,B, etc.
  - Matlab/Simulink or similar diagrams
- Source code

Waterfall Model (Royce 1970)

- Classical top-down sequential workflow with strictly separated phases.

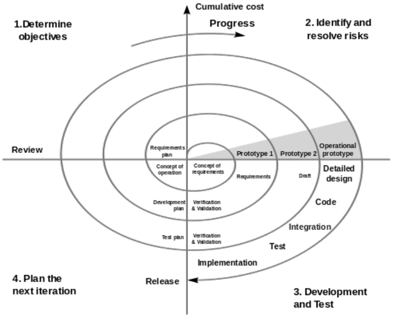


- Unpractical as an actual workflow (no feedback between phases), but even the original paper did **not** really suggest this.

Spiral Model (Böhm 1986)

- Incremental development guided by **risk factors**

- Four phases:
  - Determine objectives
  - Analyse risks
  - Development and test
  - Review, plan next iteration



- See e.g.
  - Rational Unified Process (RUP)

- Drawbacks:
  - Risk identification is the key, and can be quite difficult

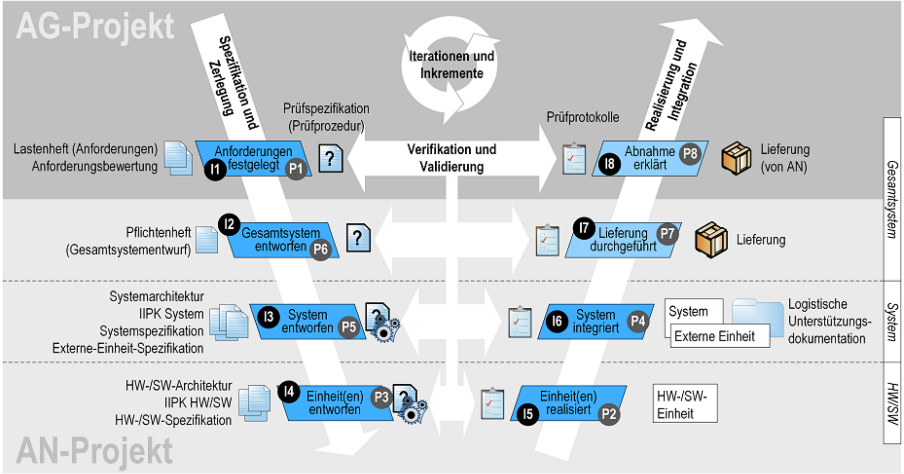
Agile Methods

- **Prototype-driven** development
  - e.g. Rapid Application Development
  - Development as a sequence of prototypes
  - Ever-changing safety and security requirements
- **Agile programming**
  - e.g. extreme Programming (XP), Scrum
  - Development guided by functional requirements
  - Process structured by rules of conduct for developers
  - Rules capture best practice
  - Less support for non-functional requirements
- **Test-driven development (TDD)**
  - Tests as **executable specifications:** write tests first
  - Often used together with the other two

V-Model

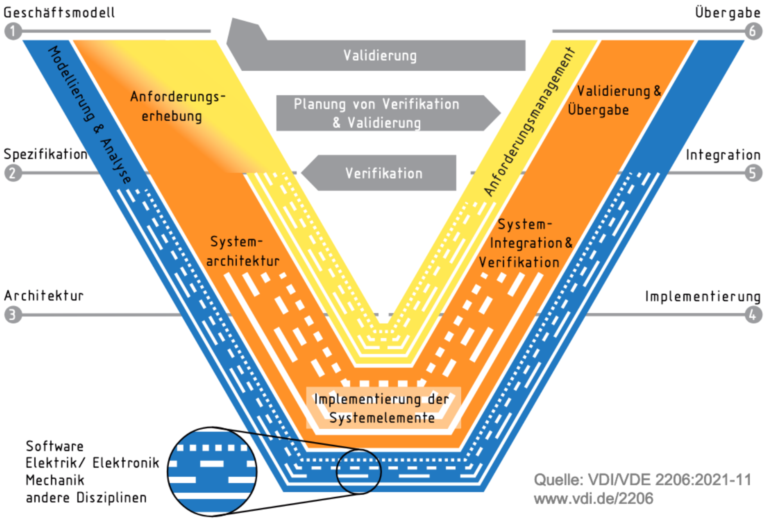
- Evolution of the waterfall model:
  - Each phase supported by corresponding verification & validation phase
  - Feedback between next and previous phase
- Standard model for public projects in Germany
  - ... but also a general term for models of this shape.
- Current: V-Modell XT (“extreme tailoring”)
  - Shape gives **dependencies**,
  - **not** necessarily **development timeline**.

Variations of the V-Modell: CIO Bund

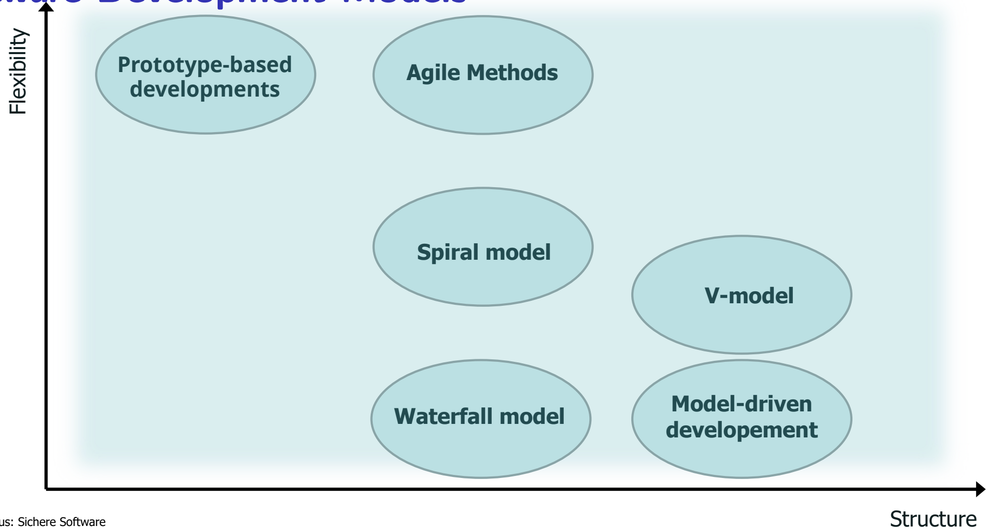


Quelle: `https://www.cio.bund.de/`, Beauftragter der Bundesregierung für IT

Variations of the V-Modell: VDI/VDE



Software Development Models



Quelle S. Paulus: Sichere Software

Development Models for Safety-Critical Systems

Development Models for Critical Systems

- Ensuring safety/security needs structure.
  - ...but **too much** structure makes developments bureaucratic, which is **in itself** a safety risk.
  - Cautionary tale: Ariane-5
- Standards put emphasis on **process**.
  - Everything needs to be planned and documented.
  - Key issues: **auditability**, **accountability**, **traceability**.
- Best suited development models are **variations of the V-model** or spiral model.
- A new trend? V-Model XT allows variations of original V-model, e.g.
  - V-Model for initial developments of a new product,
  - Agile models (e.g. Scrum) for maintenance and product extensions.

Auditability and Accountability

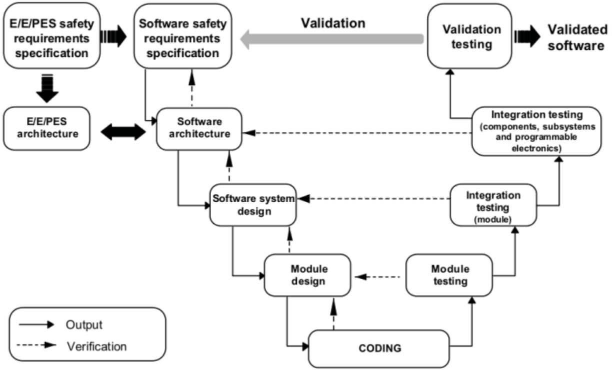
- **Version control and configuration management** is **mandatory** in safety-critical development (auditability).
- Keeping track of all artifacts contributing to a particular instance (**build**) of the system (**configuration, baseline**), and their **versions**.
- **Repository** keeps **all artifacts** in **all versions**.
  - Centralised (one repository) vs. distributed (every developer keeps own repository)
  - General model: check out – modify – commit – draw baseline
  - Baseline: identification of artefacts that are part of the same product release
  - Concurrency: enforced **lock**, or **merge** after commit.
- Well-known systems:
  - Commercial (all outdated): ClearCase, Perforce, Bitkeeper...
  - Open Source: **git** (outdated: svn, cvs, Mercurial)

Traceability

- The idea of being able to **follow requirements** (in particular, safety requirements) from requirement spec **to the code** (and possibly back).
- On the simplest level, an Excel sheet with (manual) links to the program.
- More sophisticated tools (e.g. DOORS):
  - Decompose requirements, hierarchical requirements
  - Two-way traceability: from code, test cases, test procedures, and test results back to requirements
  - e.g. RTCA DO-178C requires that all code is derived from requirements
- The SysML modelling language has traceability support:
  - Each model element can be traced to a requirement.
  - Special associations to express traceability relations.

Development Model in IEC 61508

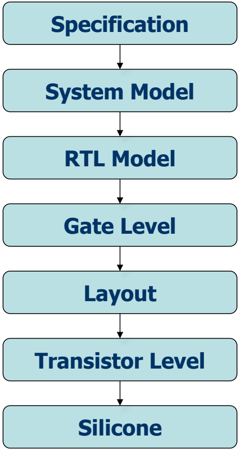
- IEC 61508 is agnostic with respect to the development model, but:
  - safety-directed activities are required for each phase of the life cycle (**safety life cycle**).
  - Development is one part of the life cycle.
- The only development model mentioned is a V-model:



Development Model in DO-178C

- DO-178C defines different **processes** in the SW life cycle:
  - Planning process
  - Development process, structured in turn into
    - Requirements process
    - Design process
    - Coding process
    - Integration process
  - Verification process
  - Quality assurance process
  - Configuration management process
  - Certification liaison process
- There is no conspicuous diagram, but the Development Process has sub-processes suggesting the phases found in the V-model as well.
  - Implicit recommendation of the V-model.

Development Model for Hardware



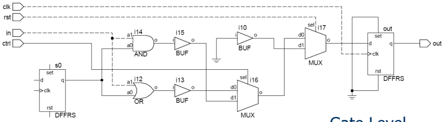

```
SC_MODULE(example) {
sc_in_clk clk;
sc_in<bool> rst, in, ctrl; sc_out<bool> out;
int o, s0;

void tick() {
  if (rst.read) o= 0;
  else if (!ctrl.read) o= s0 | in.read;
         else o= s0 & in.read;
  out.write(o); s0= o;
}
...
}
```

System-Model: SystemC

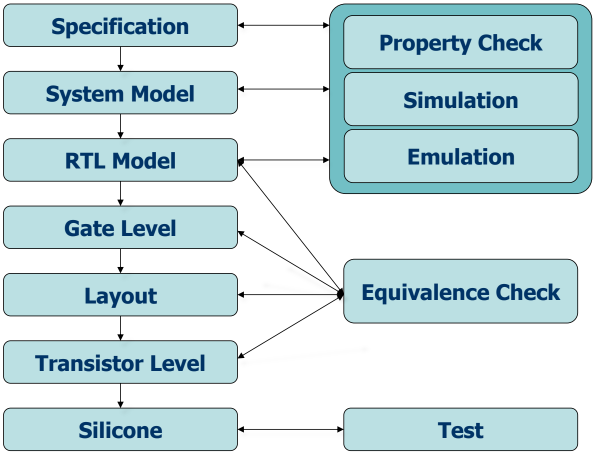
```
always @(posedge clk)
  if (rst) out <= 0;
  else
    if (! ctrl)  out <= s0 | in;
    else       out <= s0 & in;
```

Register-Transfer-Ebene: Verilog



Gate Level

Development Model for Hardware

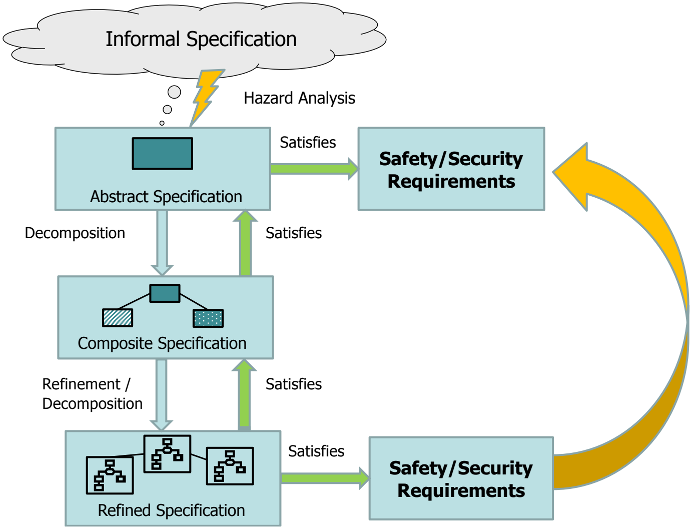


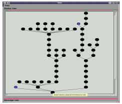
Basic Notions of Formal Software Development

Formal Software Development

- In a formal development, properties are stated in a rigorous way with a **precise mathematical semantics**.
- Formal specification requirements can be **proven**.
- **Advantages** :
  - Errors can be found early in the development process.
  - High degree of confidence into the system.
  - Recommended for high SILs/EALs.
- **Drawbacks** :
  - Requires a lot of effort and is thus expensive.
  - Requires qualified personnel (that would be **you**).
- There are tools which can help us by
  - finding (simple) proofs for us (model checkers), or
  - checking our (more complicated) proofs (theorem provers).

Structuring the Formal Development





- ***Test***
- ***Program analysis***
- ***Model checking***
- ***Formal proof***

Finite State Machines

Finite State Machine (FSM)

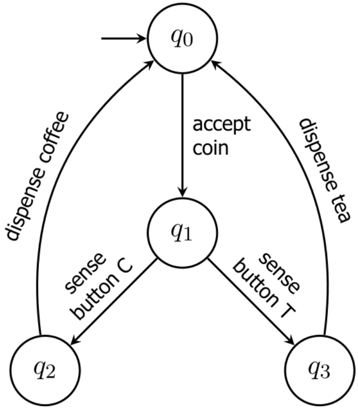
A FSM is given by $\mathcal{M} = \langle \Sigma, \Sigma_0, \rightarrow \rangle$ where

- Σ is a finite set of **states**,
- $\Sigma_0 \subseteq \Sigma$ is a set of **initial states**, and
- $\rightarrow \subseteq \Sigma \times \Sigma$ is a **transition relation**, such that $\rightarrow$ is left-total:

$$\forall s \in \Sigma. \exists s' \in \Sigma. s \rightarrow s'$$

- The most basic notion of a system.
- Many variations of this definition exists, e.g. no initial states, state variables or labelled transitions.
- Note there is no input or output, and no **final** state (key difference to automata).
- If $\rightarrow$ is a function, the FSM is **deterministic**, otherwise it is **non-deterministic**.

Example: Vending Machine



Transitions:

1. Insert/accept coin
2. Press/sense button: tea or coffee
3. Dispense tea or coffee, return to (1)

$$\mathcal{M} = \langle Q, Q_0, \rightarrow \rangle$$

$$\begin{aligned} Q &= \{q_0, q_1, q_2, q_3\} \\ Q_0 &= \{q_0\} \\ \rightarrow &= \{(q_0, q_1), (q_1, q_2), (q_1, q_3), (q_2, q_0), (q_3, q_0)\} \end{aligned}$$

Traces

Trace

Given a set Σ of states, a (finite) **trace** is a (finite) sequence $t = \langle t_0, t_1, t_2, \ldots, t_n \rangle$ with $t_i \in \Sigma$.

A trace is **admissible** for a FSM $\mathcal{M} = \langle \Sigma, \Sigma_0, \rightarrow \rangle$ iff

(i) $t_0 \in \Sigma_0$, and

(ii) $\forall i.\, i < n \Longrightarrow t_i \rightarrow t_{i+1}$.

- The empty sequence $\varepsilon = \langle\rangle$ is the empty trace. It is admissible for all FSMs.
- For a (finite) trace $t = \langle t_i \rangle_{i=0,\ldots,n}$, we write $t[i]$ for t_i.
- The set of all **finite** traces for Σ is written Σ^*; the set of **infinite** traces is written Σ^ω, and the set of **all** traces is written $\mathrm{Tr}(\Sigma) = \Sigma^* \cup \Sigma^\omega$.

Trace Algebra

- For a (finite) trace $t = \langle t_i \rangle_{i=0,\ldots,n}$, we define the **length** of t as $|t| \stackrel{def}{=} n + 1$, with $|\varepsilon| = 0$.

Concatenation

Given a (finite) trace s, and a (finite) trace t, their **concatenation** $s \cdot t$ is defined as

$$(s \cdot t)[j] \stackrel{def}{=} \begin{cases} s[j] & j < |s| \\ t[j - |s|] & j \geq |s| \end{cases}$$

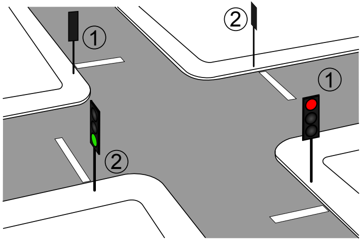
- Concatenation can be generalised to **sets** of traces, with $S \cdot T \stackrel{def}{=} \{s \cdot t \mid s \in S, t \in T\}$.

Prefix Ordering

A trace t is a **prefix** of a trace s, written $t \leq s$, iff $\exists t'.\, t \cdot t' = s$.

- The prefix ordering generalises to **sets** of traces by $T \leq S$ iff $\forall t.\, t \in T \implies \exists s.\, s \in S \wedge t \leq s$.

Example: Street Crossing with Traffic Lights



Source: Wikipedia

- States and Transitions:

$$
\begin{aligned}
\mathcal{M} &= \langle Q, Q_0, \rightarrow \rangle \\
L &= \{r, ry, y, g\} \\
Q &= L \times L \\
Q_0 &= \{\langle r, g \rangle\}
\end{aligned}
$$

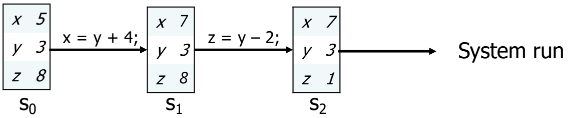
- Do traffic lights switch concurrently or interleaved?
- Each traffic light switches $r \rightarrow ry \rightarrow g \rightarrow y \rightarrow r$
- But not all states should be reachable
- Some states are "bad" (e.g. $\langle g, g \rangle$)

Semantics at Different Levels of Abstraction

- On an abstract level, the semantics of a **system** (programs running on computers) can be modelled as a FSM.
- On the **hardware level**, a single computer can be modelled as a FSM:
  - **State**: Registers, Memory
  - **Transitions**: read instruction from current PC, execute instruction.
- On the **software level**, the operational semantics of a program can be modelled as FSM:
  - **State**: Memory (Variables)
  - **Transitions**: Effects of each program statement
  - Different levels of abstraction, depending on programming language

Operational Semantics of Programs

- **States** and transitions between them:



- **Operational semantics** describes relation between states and transitions:

$$\frac{\langle s, e\rangle \to n}{\langle s, x = e\rangle \to s[^{n}_{x}]} \qquad \text{hence:} \qquad \frac{\langle s_0, y + 4\rangle \to 7}{\langle s_0, x = y + 4\rangle \to s_1}$$

- **Formal proofs**; e.g. proving

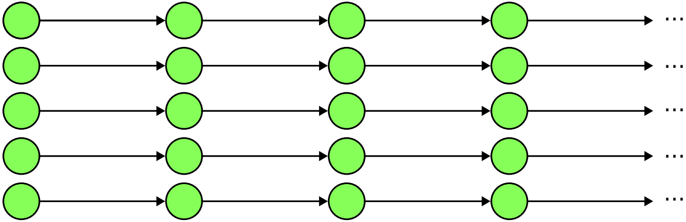
```
x = y + 4;
z = y - 2;
```

yields the same final state as

```
z = y - 2;
x = y + 4;
```

Semantics of Programs and Requirements

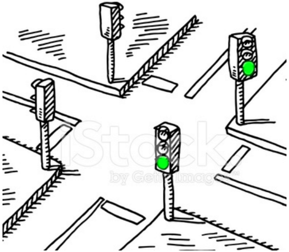
- Operational semantics gives us the set of all possible system runs:



- We can now consider safety/security-related **requirements**:
  - Requirements on **single states**,
  - Requirements on **system runs**,
  - Requirements on **sets of system runs**.
- Each gives rise to different **proof methods**.

Requirements on States: Safety Properties

- Safety property S: *Nothing bad happens.*
  - i.e. system will never enter a **bad** state.
  - e.g. lights are never switched to green at the same time.
- A **bad state**
  - can be **immediately** recognized;
  - can **not** be sanitized (by following states).



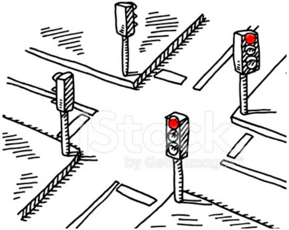
- $S \in \mathcal{P}(\Sigma^\omega)$ is a **safety property** iff

$$\forall t.\, t \notin S \longrightarrow (\exists t_t.\, t_1 \in \Sigma^* \wedge t_1 \leq t \longrightarrow \forall t_2.\, t_1 \leq t_2 \longrightarrow t_2 \notin S)$$

Proving Safety Properties

- In the previous specification, t_1 is **finite**, Hence:
  - a property is a safety property iff its violation can be detected on a finite trace.
  - Thus, the **violation** of safety properties can be detected by **testing** (and model-checking).
- Safety properties are typically proven by **induction**:
  - Base case: initial states are good.
  - Step case: each possible transition from a good state leads to a good state.
- Safety properties can be enforced by **run-time monitors**:
  - Monitor checks following state in advance, and allows execution only if it is a good state.

Requirements on Runs: Liveness Properties



- Liveness property S: *Good things will happen eventually**.
  - I.e. System will at some point enter a **good** state.
  - E.g. Traffic lights will eventually go green.
- A **good state**
  - is always possible, but
  - potentially infinite (i.e. no upper bound on when it will occur).
- $L \in \mathcal{P}(\Sigma^\omega)$ is a **liveness property** iff

$$\forall t.\, t \in \Sigma^* \Longrightarrow \exists t_1.\, t \cdot t_1 \in L$$

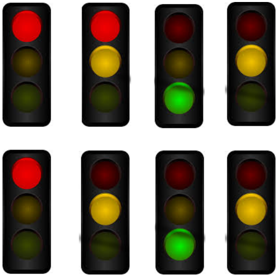
- i.e. all finite traces can be extended to a trace in L.

* NB:*eventually* bedeutet *irgendwann* oder *schlussendlich* aber **nicht** *eventuell*.

Proving Liveness Properties

- Liveness properties **cannot** be **enforced** by run-time monitors.
- Liveness properties **cannot** be checked by **testing**.
- Liveness properties are typically proven by the help of well-founded orderings:
  - Define measure function μ on states S: $\mu : S \to X$ with $(X, \preceq)$ well founded.
  - Show each transition decreases μ: if $s_1 \to s_2$ then $\mu(s_2) \preceq \mu(s_1)$
  - If $\mu(t)$ minimal in $\preceq$ then $t \in S$
- Example:
  - Ordering $(X, \preceq) = (\mathbb{N}, \leq)$
  - Measure denotes the number of transitions until light goes green.

Requirements on Sets of Runs: Safety Hyperproperties



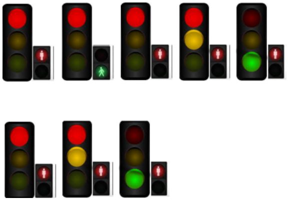
- Safety hyperproperty S: **System never behaves bad.**
  - No bad thing happens in a finite set of traces;
  - (prefixes of) different system runs do not exclude each other;
  - E.g. Traffic light cycle is always the same.
- A **bad system** can be recognized by a bad observation (set of finite runs)
  - A bad observation cannot be sanitized by adding additional runs.
  - E.g. two system runes having different traffic light cycles.
- $S \in \mathcal{P}(\mathcal{P}(\Sigma^\omega))$ is a **safety hyperproperty** iff

$$\forall T.\, T \notin S \longrightarrow (\exists Obs.\, Obs \in \mathcal{P}_{fin}(\Sigma^*) \wedge Obs \leq T \longrightarrow \forall T'.\, Obs \leq T' \longrightarrow T' \notin S)$$

(Same as safety property but we talk about sets of traces here!)

- Examples: Non-interference

Requirements on Sets of Runs: Liveness Hyperproperties



- Liveness hyperproperty S: *The system will eventually evolve to a good system.*
  - Considering any finite part of system behaviour, the system eventually develops into a good systems (by extending runs, or adding new runs)
  - E.g. Green lights for pedestrians can always be omitted.
- $L \in \mathcal{P}(\mathcal{P}(\Sigma^\omega))$ is a **liveness hyperproperty** iff

$$\forall T.\, T \in \mathcal{P}(\Sigma^*) \Longrightarrow \exists G. \in \mathcal{P}(\Sigma^\omega) \wedge T \leq G \wedge G \in L$$

  - T is a finite set of traces
  - Each observation can be completed to a system G satisfying L
- Examples: average response time, SLAs, fair scheduling

Facts about (Hyper)Properties

- Every property is an **intersection** of a **safety** and a **liveness** property.

- Every hyperproperty is an **intersection** of a **safety** and a **liveness** hyperproperty.

Conclusion & Summary

- Software development models: structure vs. flexibility
- Safety standards such as IEC 61508, DO-178C suggest V-model.
  - Specification and implementation linked by verification and validation.
  - Variety of artefacts produced at each stage, which have to be subject to external review and audits.
- Finite state machines are the most basic semantic notion.
- Requirements are formulated on basis of traces
- Safety and Security Requirements
  - Properties: sets of traces, requirements on single states or runs
  - Safety and Liveness properties
  - Hyperproperties: sets of properties, requirements on many runs
  - Safety and Liveness hyperproperties