Universität Bremen

Systeme hoher Sicherheit und Qualität

WS 2019/2020

Lecture 3: The Software Development Process

Christoph Lüth, Dieter Hutter, Jan Peleska

Organisatorisches

- Die Übung am Donnerstag, 31.10.2019, fällt aus (Reformationstag).

- Nächste Übung am Dienstag, 05.11.2019.

Where are we?

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, such as
    - 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.

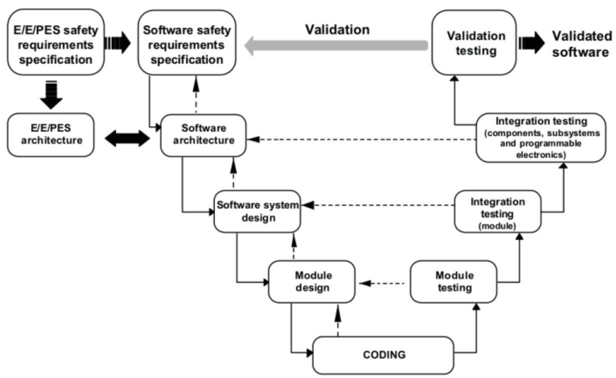
- What does quality mean in this context?
  - What is the **output**? Just the software product, or more? (specifications, test runs, documents, proofs…)

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

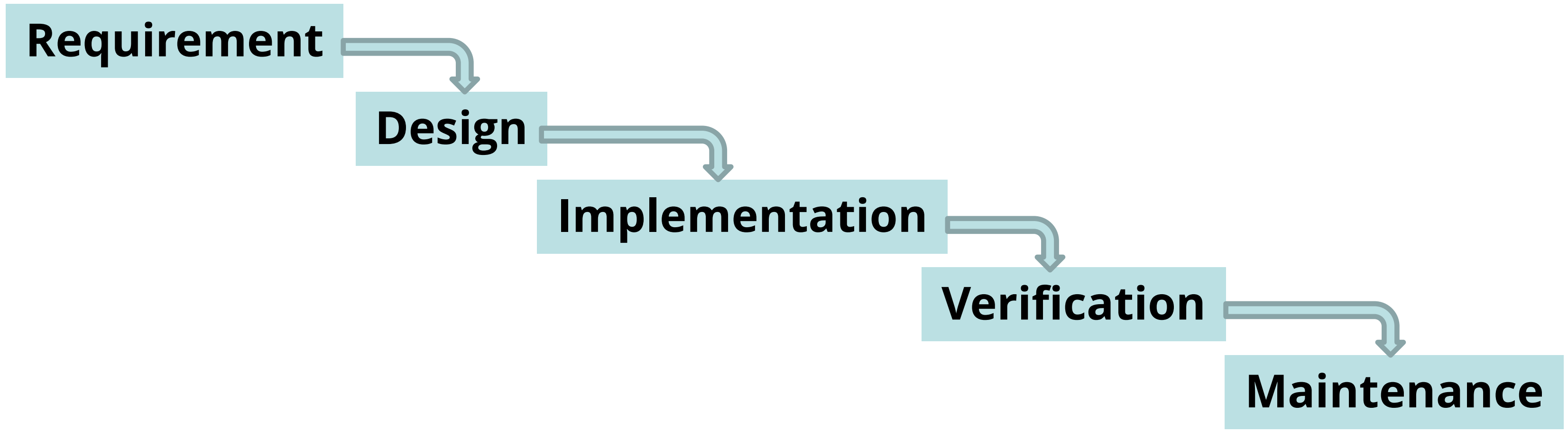


Verification & validation:
- Code review protocols
- Test cases, procedures, and test results
- Proofs

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

Waterfall Model (Royce 1970)

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

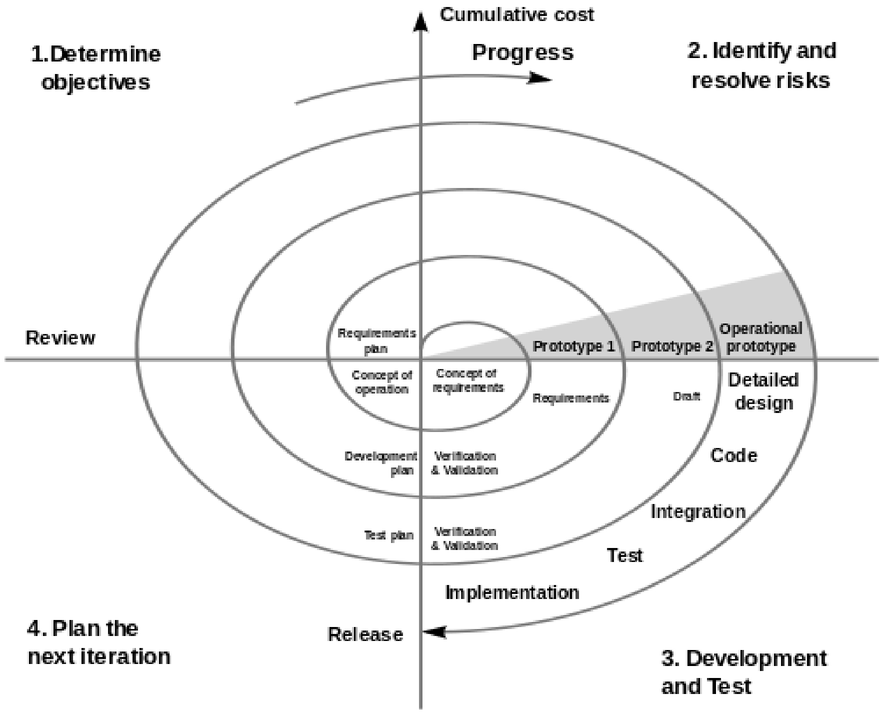


- Unpractical as an actual workflow (no feedback between phases), but even the original paper did **not** really suggest this.

Spiral Model (Böhm 1986)

- Incremental development guided by **risk factors**

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

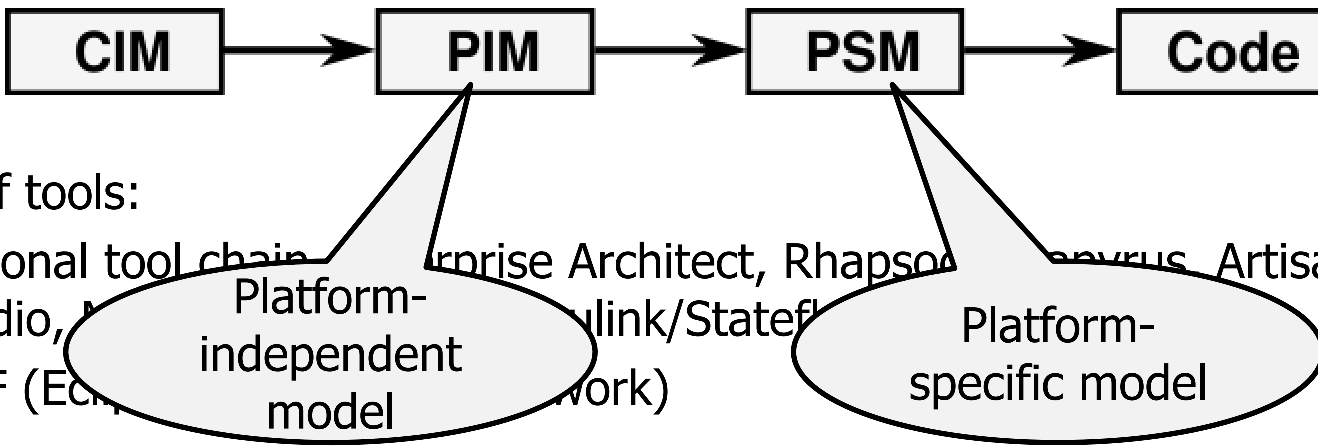


- See e.g.
  - Rational Unified Process (RUP)

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

Model-Driven Development (MDD, MDE)

- Describe problems on abstract level using *a modeling language* (often a *domain-specific language*), and derive implementation by model transformation or run-time interpretation.
- Often used with UML (or its DSLs, eg. SysML)



- Variety of tools:
  - Rational tool chain, … rprise Architect, Rhapso… …pyrus, Artisan Studio, … …ulink/Statefl…
  - EMF (Ec… …work)

- Strictly sequential development

- Drawbacks: high initial investment, limited, reverse engineering and change management (code changes to model changes) is complex

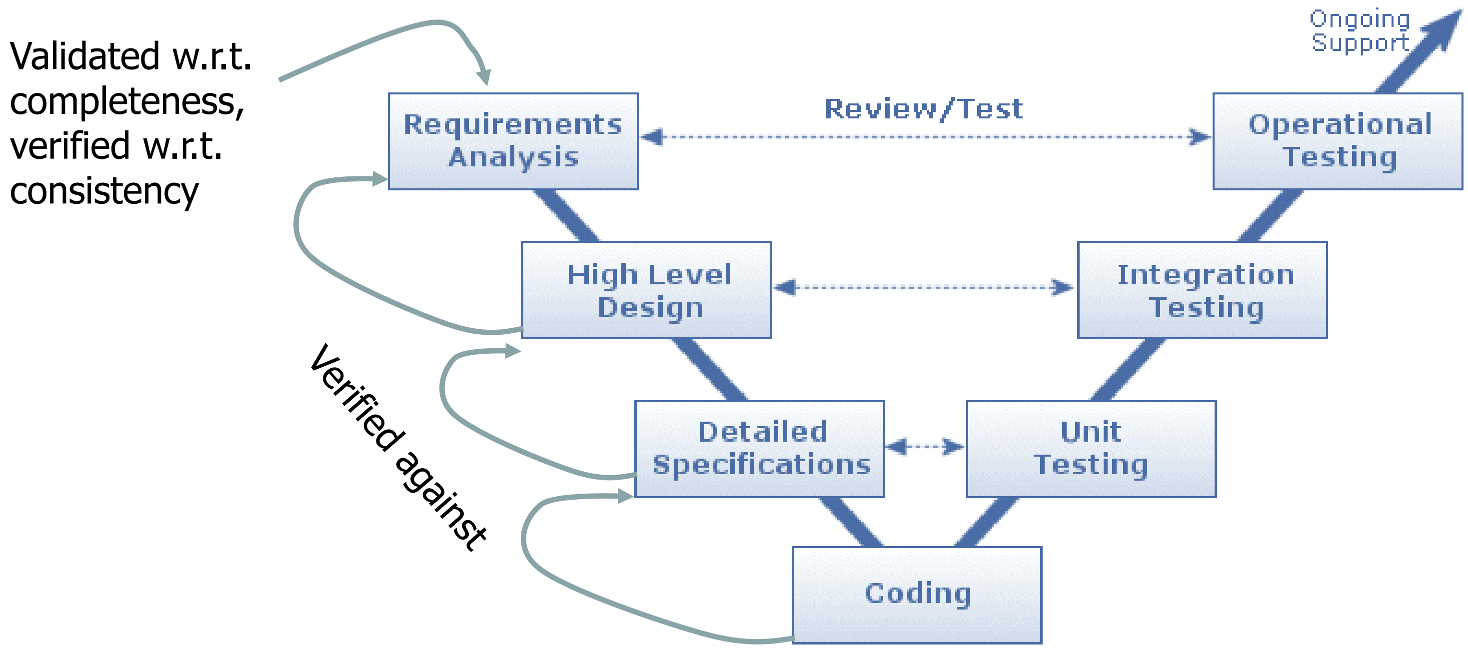
* Proprietary DSL – not related to UML

Agile Methods

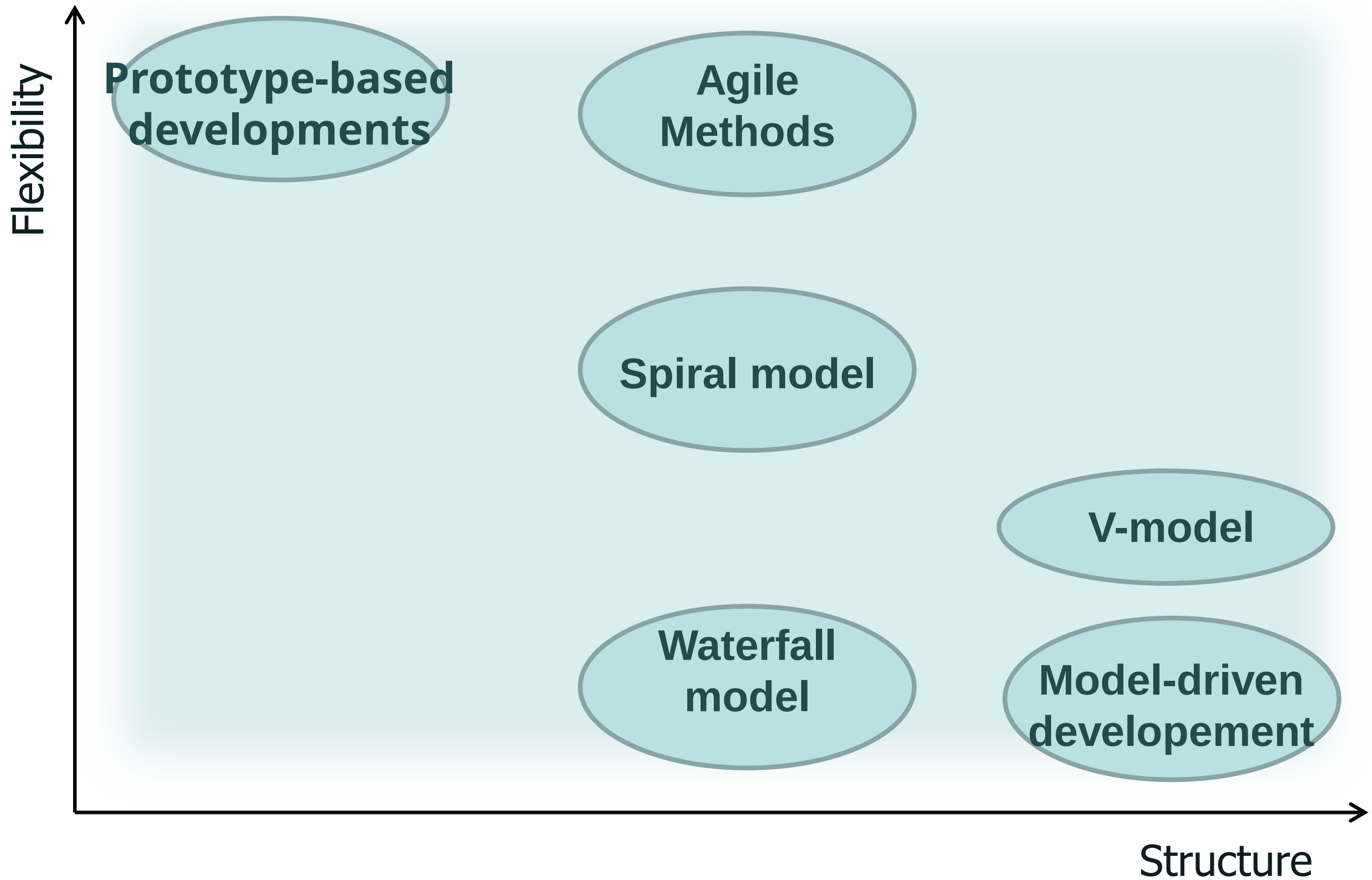
- Prototype-driven development
  - E.g. Rapid Application Development
  - Development as a sequence of prototypes
  - Ever-changing safety and security requirements
- Agile programming
  - E.g. Scrum, extreme programming
  - Development guided by functional requirements
  - Process structured by rules of conduct for developers
  - Rules capture best practice
  - Less support for non-functional requirements
- Test-driven development
  - 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 depencies, not development sequence



Software Development Models



from 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**), 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
  - Concurrency: enforced **lock**, or **merge** after commit.

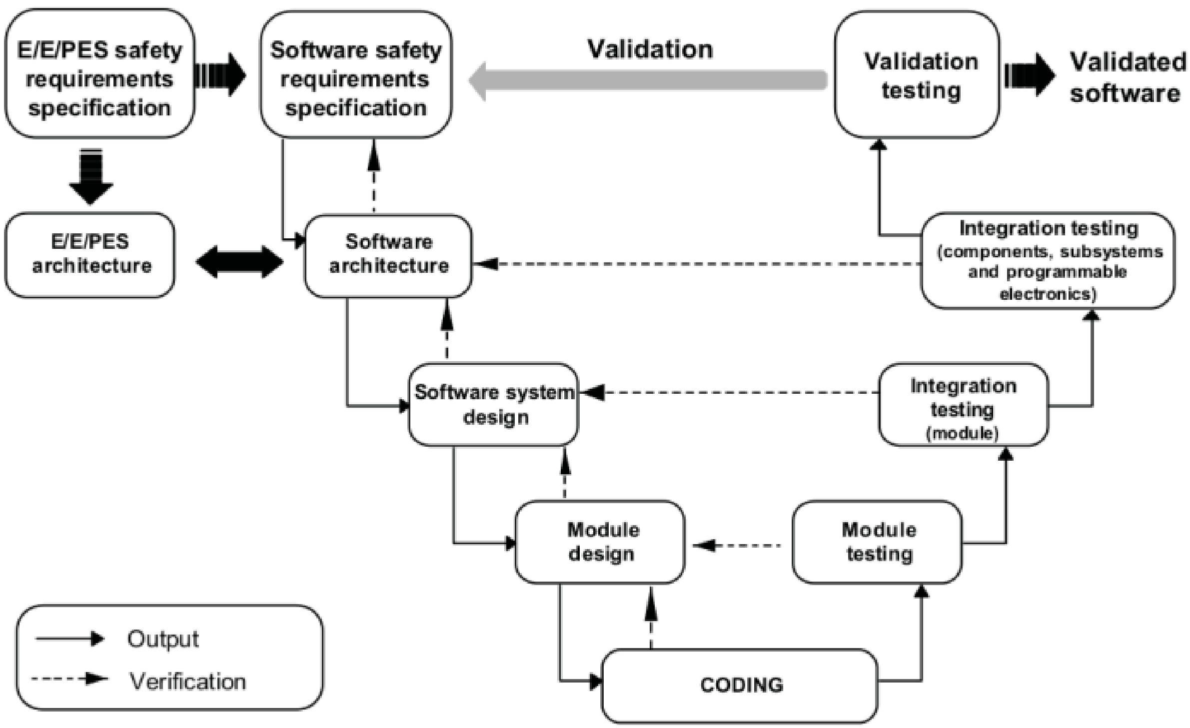
- Well-known systems:
  - Commercial: ClearCase, Perforce, Bitkeeper…
  - Open Source: Subversion (centralised); Git, Mercurial (distributed)

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 include DOORS:
  - Decompose requirements, hierarchical requirements
  - Two-way traceability: from code, test cases, test procedures, and test results back to requirements
  - E.g. DO-178B requires all code derives 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 in principle allows any development model, but:
  - It requires safety-directed activities in each phase of the life cycle (safety life cycle, cf. last lecture).
  - Development is one part of the life cycle.
- The only development model mentioned is a V-model:

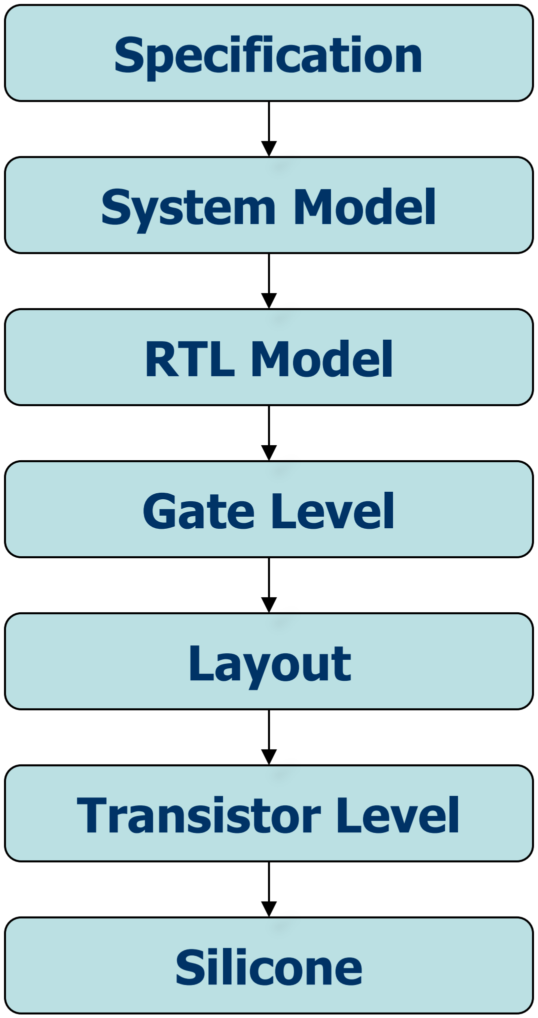


Development Model in DO-178B/C

- DO-178B/C 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



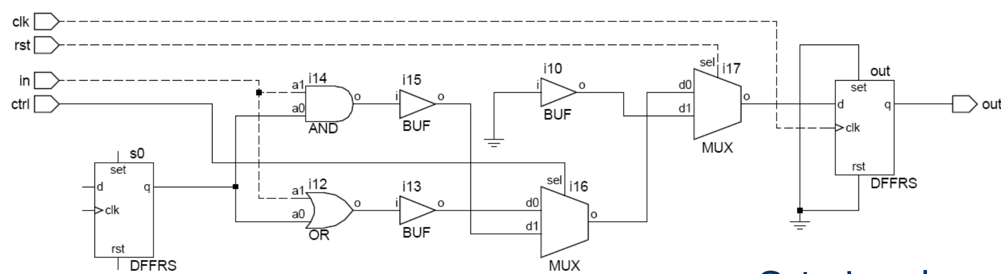

```
SC_MODULE(example) {
sc_in_clk clk;
sc_in<bool> rst, in, ctrl; sc_out<bool> out;
int o, s0;

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

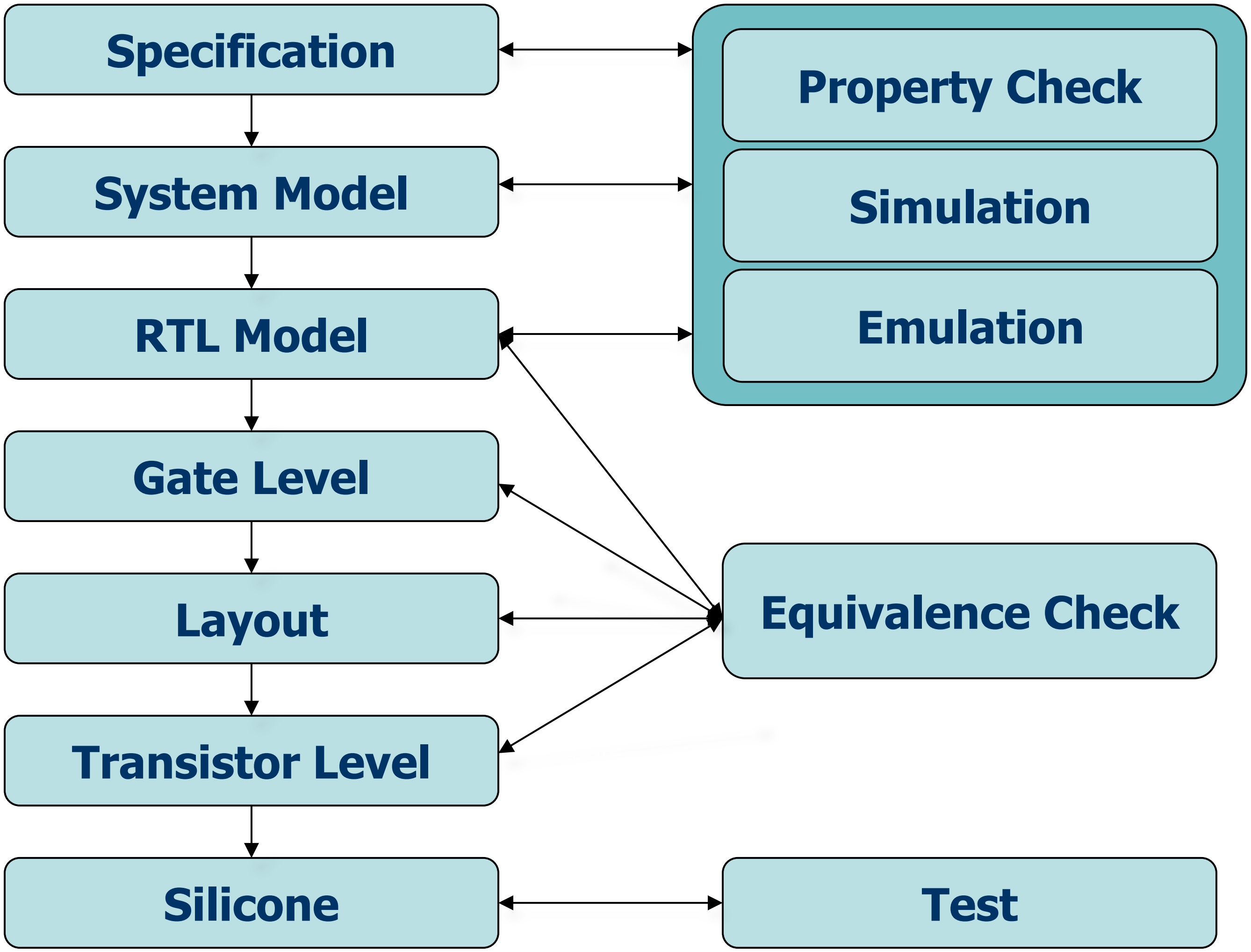
```
always @(posedge clk)
  if (rst) out <= 0;
  else
    if (! ctrl)  out <= s0 | in;
    else       out <= s0 & in;
```

Register-Transfer-Ebene: Verilog



Gate Level

Development Model for Hardware



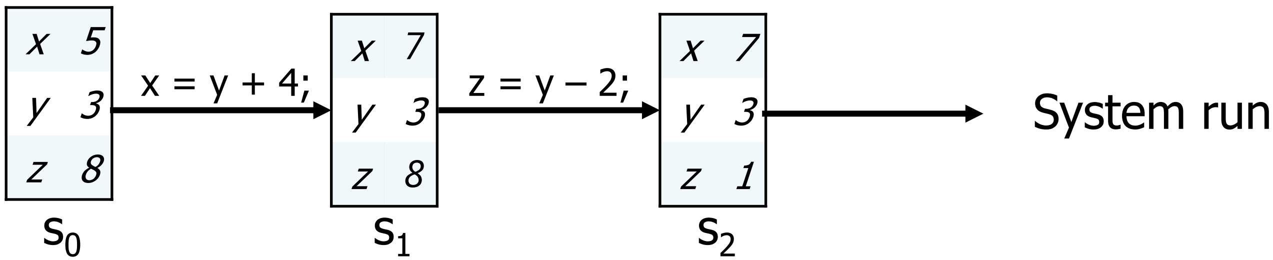
Basic Notions of Formal Software Development

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.
  - Recommend use of formal methods 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).

Formal Semantics

- **States** and transitions between them:



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

$$\frac{s \vdash e \to n}{s \vdash x = e \;\to\; s[x / n]} \qquad \text{hence:} \qquad \frac{s_0 \vdash y + 4 \to 7}{s_0 \vdash x = y + 4 \;\to s_1}$$

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

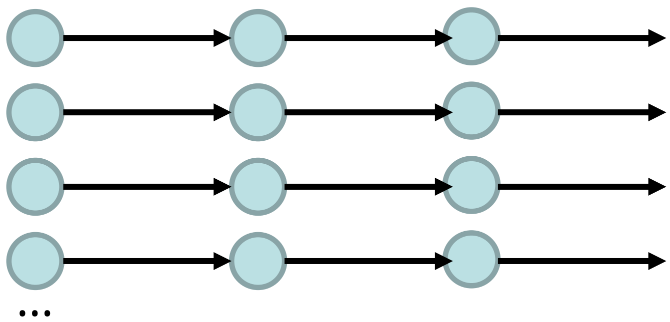
  x = y + 4; z = y – 2;

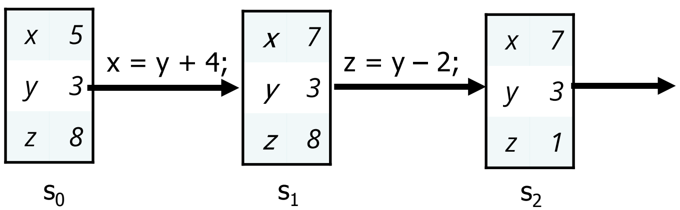
  yields the same final state as

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

Semantics of Programs and Requirements

- Set of all possible system runs





- **Requirements** related to safety and security:
  - Requirements on single states ?
  - Requirements on system runs ?
  - Requirements on sets of system runs ?

Alpern & Schneider
Clarkson & Schnei

Some Notions

- Let b, t be two traces then

  $b \leq t$ iff. $\exists t'. t = b \cdot t'$ i.e. b is a *finite* prefix of t

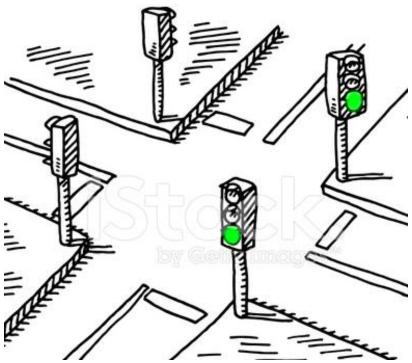
- A **property** is a set of infinite execution traces (like a program)
  - Trace t satisfies property P, written $t \vDash P$, iff $t \in P$

- A **hyperproperty** is a set of sets of infinite execution traces (like a set of programs)
  - A system (set of traces) S satisfies H iff S $\in$ H
  - An observation Obs is a finite set of finite traces
  - $Obs \leq S$ (Obs is a prefix of S) iff

    Obs is an observation and $\forall m \in Obs. \exists t \in S. m \leq t$

Requirements on States: Safety Properties

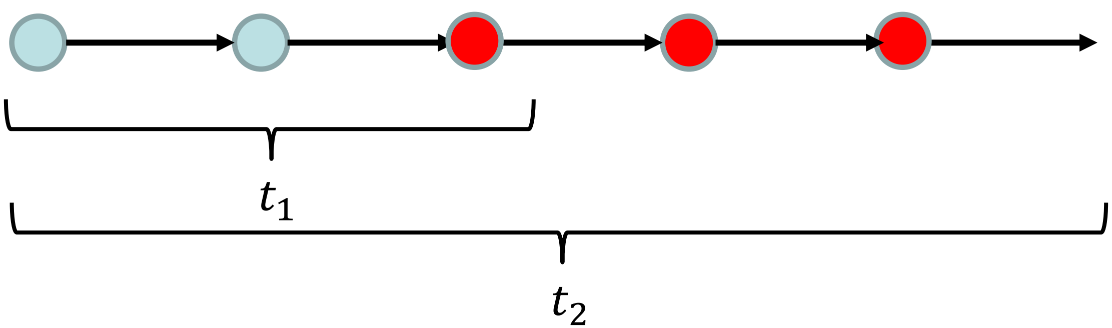
- Safety property *S*: „Nothing bad happens"
  - i.e. the system will never enter a *bad* state
  - E.g. "Lights of crossing streets do not go green at the same time"



- A bad state:
  - can be **immediately** recognized;
  - **cannot be sanitized** by following states.

- *S* is a safety property iff

$$\forall t.\ t \notin S \Longrightarrow (\exists t_1.\ t_1 \leq t \Longrightarrow \forall t_2.\ t_1 \leq t_2 \Longrightarrow\ t_2 \notin S),\quad t_1 \text{ finite}$$



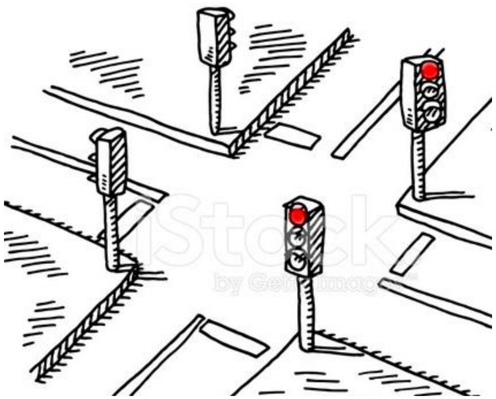
Proving Safety Properties

- In the previous specification, t_1 is **finite**. As a consequence,
  - a property is a safety property if and only if its violation can be detected on a finite trace.

- Safety properties are typically proven by induction
  - Base case: initial states are good (= not bad)
  - Step case: each transition transforms a good state again in 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 L:
  - „Good things will happen eventually"
  - E.g. "my traffic light will go green eventually * "
- A good thing is always possible and possibly infinite.
- L is a liveness property iff
  - $\forall\, t.\ \mathrm{finite}(t)\ \rightarrow \exists\, t_1.\ \ t \cdot t_1 \in L$
  - i.e. all finite traces t can be extended to a trace in L.

* Achtung: "eventually" bedeutet "irgendwann" oder "schlussendlich" aber *nicht* "eventuell" !

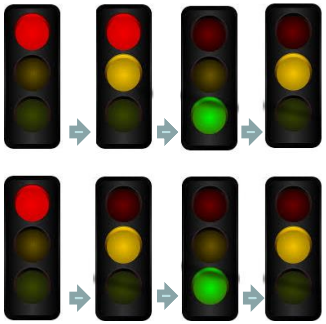
Satisfying Liveness Properties

- Liveness properties cannot (!) be enforced by run-time monitors.

- Liveness properties are typically proven by the help of well-founded orderings
  - Measure function m on states s
  - Each transition decreases m
  - $t \in L$ if we reach a state with minimal m

- E.g. measure denotes the number of transitions for the light to go green

Requirements on Sets of Runs: Safety Hyperproperties



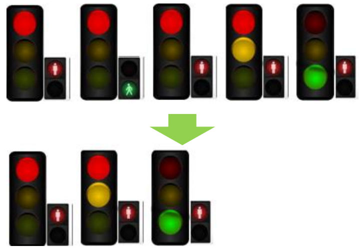
- Safety hyperproperty: „System never behaves bad“
  - No bad thing happens in a finite set of finite traces
  - (the prefixes of) different system runs do not exclude each other
  - E.g. "the traffic light cycle is always the same"

- A bad system can be recognized by a bad observation (set of finite runs)
  - A bad observation cannot be sanitized regards less how we continue it or add additional system runs
  - E.g. two system runs having different traffic light cycles

- S is a safety hyperproperty iff (see safety property):

$$\forall T. T \notin S \implies (\exists Obs.\ Obs \leq T \implies \forall T'. Obs \leq T' \implies T' \notin S)$$

Requirements on Sets of Runs: Liveness Hyperproperties



- Liveness hyperproperty S:
  „The system will eventually develop to a good system"
  - Considering any finite part of a system behavior, the system eventually develops into a "good" system (by continuing appropriately the system runs or adding new system runs)
  - E.g. "Green light for pedestrians can always be omitted"
- L is liveness hyperproperty iff

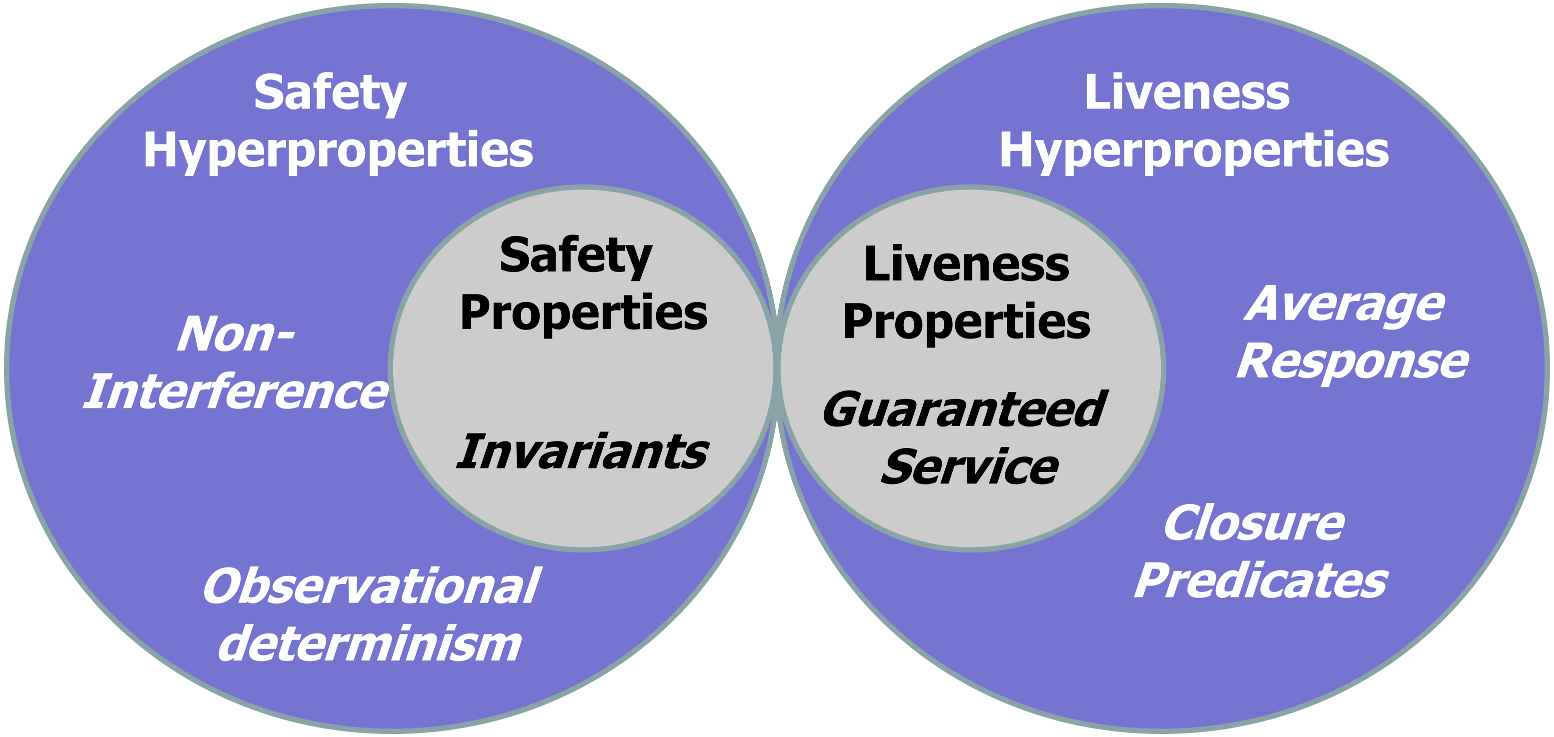
$$\forall T.\ \exists G.\ T \leq G \wedge G \in L$$

  - T is a finite set of finite traces (observation)
  - Each observation can be explained by a system G satisfying L

- Examples:
  - Average response time
  - Closure operations in information flow control
  - Fair scheduling

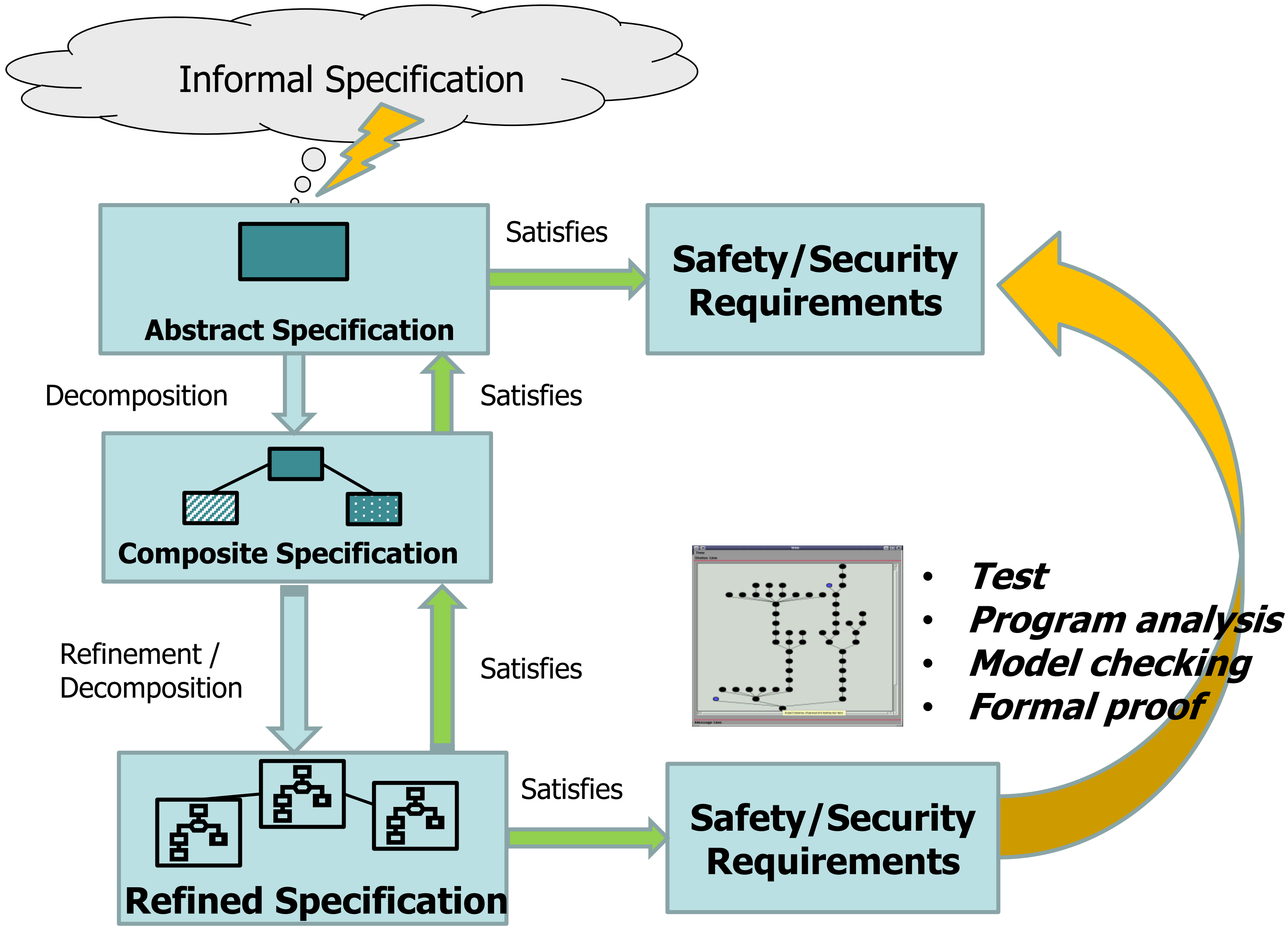
Landscape of (Hyper)Properties

- Each (hyper-) property can be represented as a combination of safety and liveness (hyper-) properties.



Structuring the Formal Development

The Global Picture

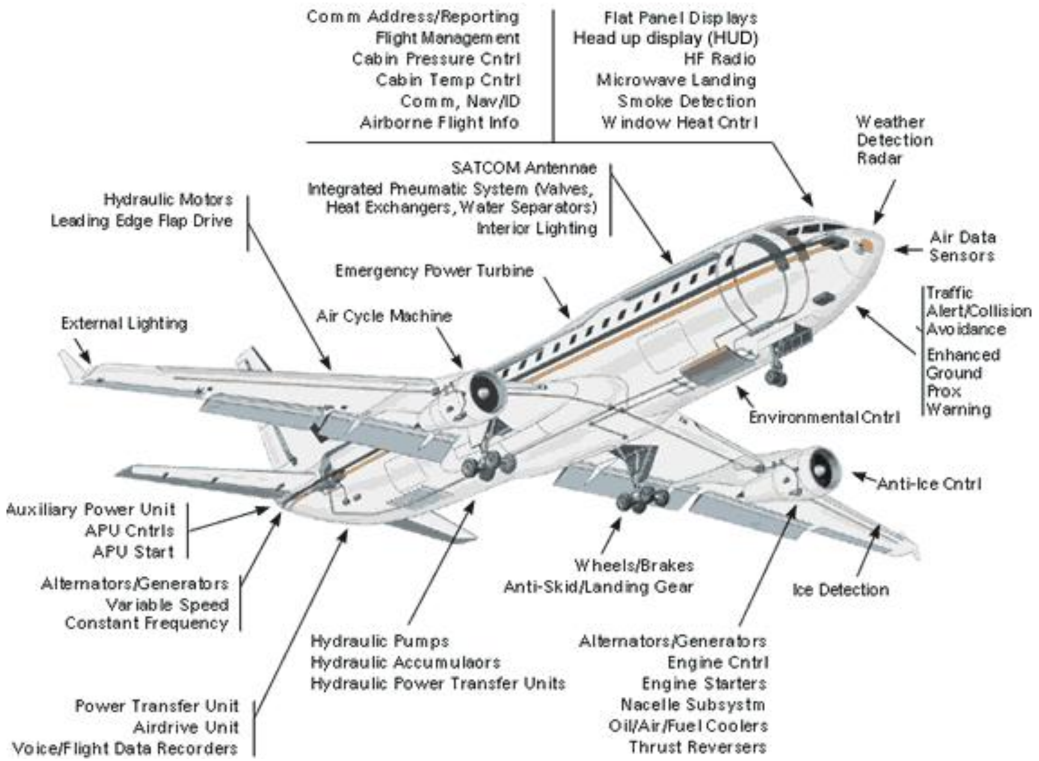


Structuring the Development

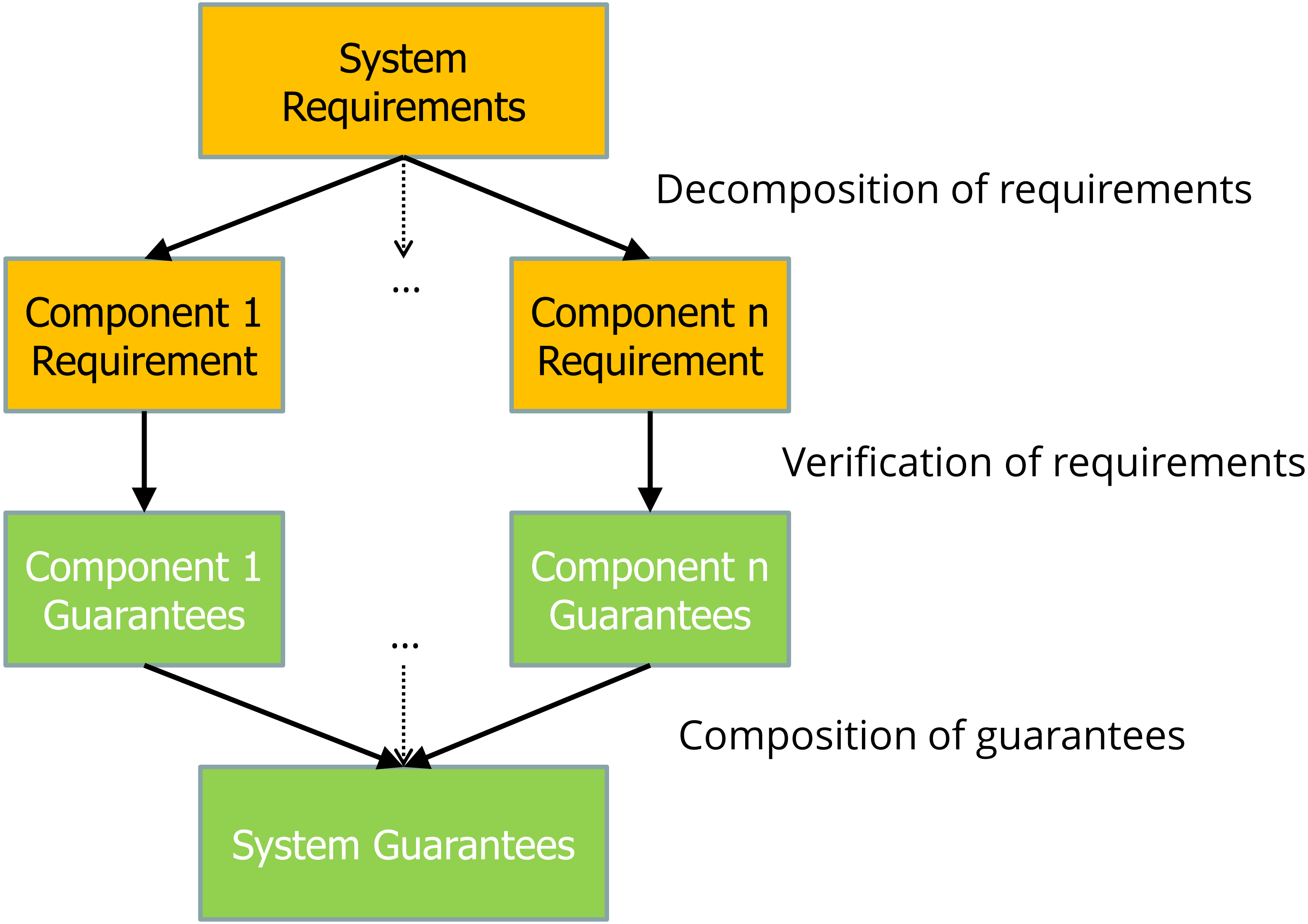
- Horizontal structuring:
  - Modularization into components
  - Composition and Decomposition
  - Aggregation
- Vertical structuring:
  - Abstraction and refinement
    from design specification to implementation
  - Declarative vs. imparative specification
  - Inheritance of properties
- Views:
  - Addresses multiple aspects of a system
  - Behavioral model, performance model, structural model, analysis model(e.g. UML, SysML)

Horizontal Structuring (informal)

- Composition of components
  - Dependent on the individual layer of abstraction
  - E.g. modules, procedures, functions,…
- Example:

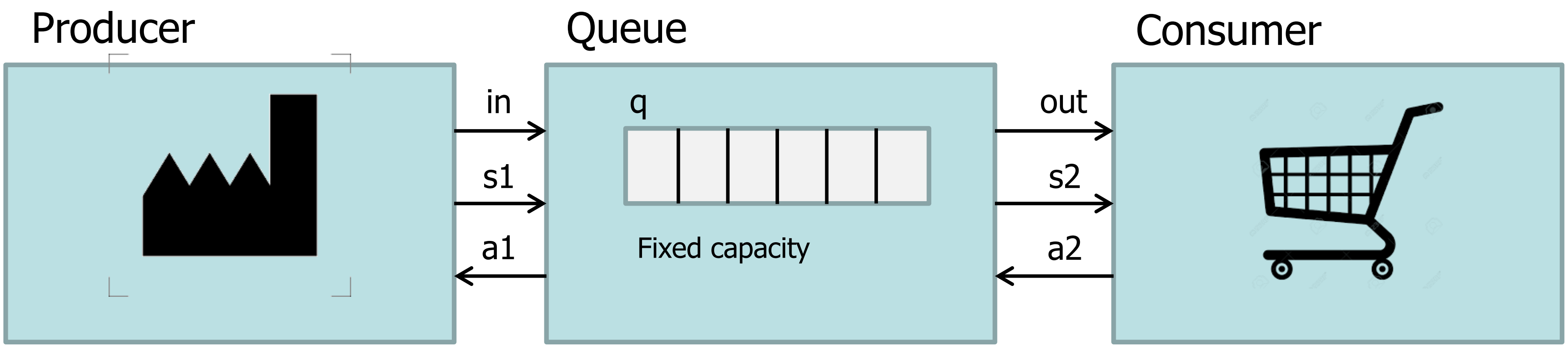


Modular Structuring of Requirements



Mutual Dependencies: Assume/Guarantee

- Safety requirement: Queue does not loose any items.




```
Loop:
  if (s1 == a1) {
   send(x, in);  s1 = not s1 }
```

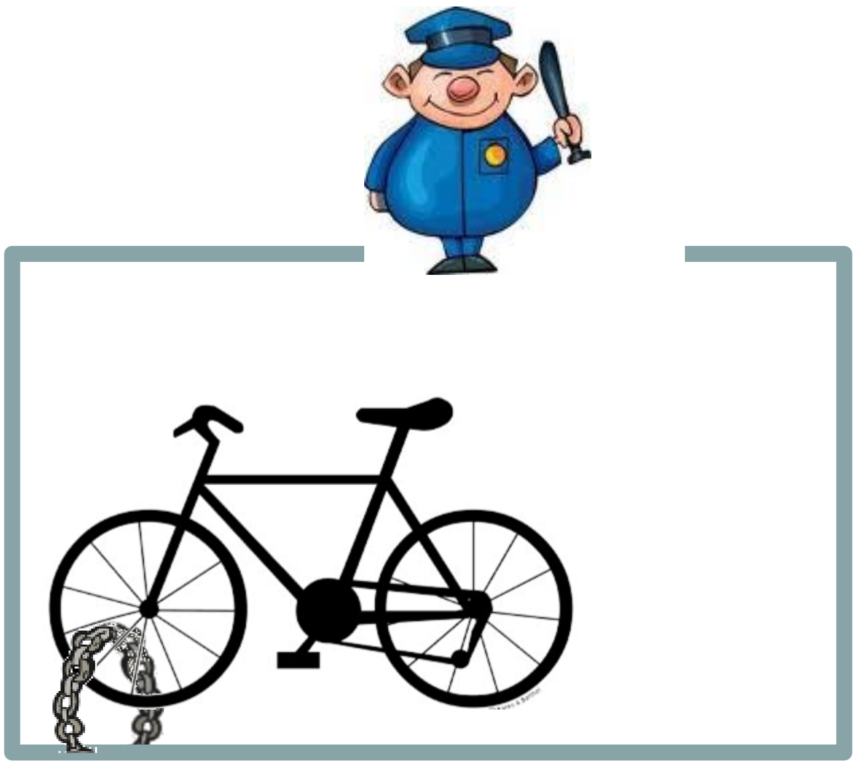
```
Loop:
  if (s1 != a1 && |q| < max)  {
      enq(q, in);
      a1 = not a1;
      }
  if (s2 == a2 && |q| > 0)  {
      deq(q, out);
      s2 != not s2
      }
```

```
Loop:
  if (s2 != a2) then {
     read(y, out);
     a2 = not a2;
     consume(y)
     }
```

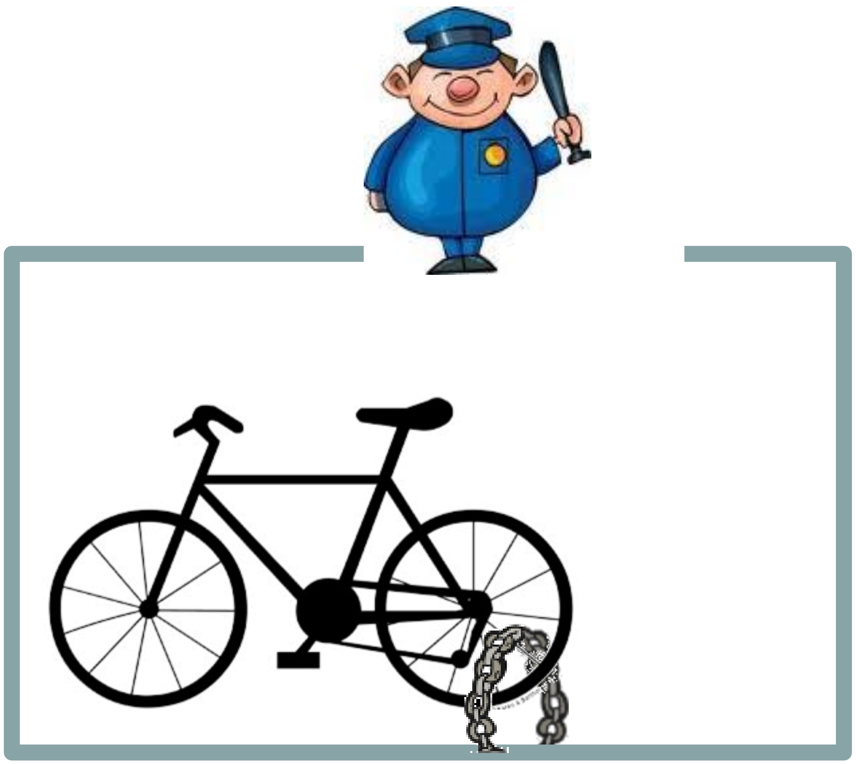
- Components depend on each other!
- Initialization ?

Composition of Security Guarantees

Only complete bicycles are allowed to pass the gate.



Secure !

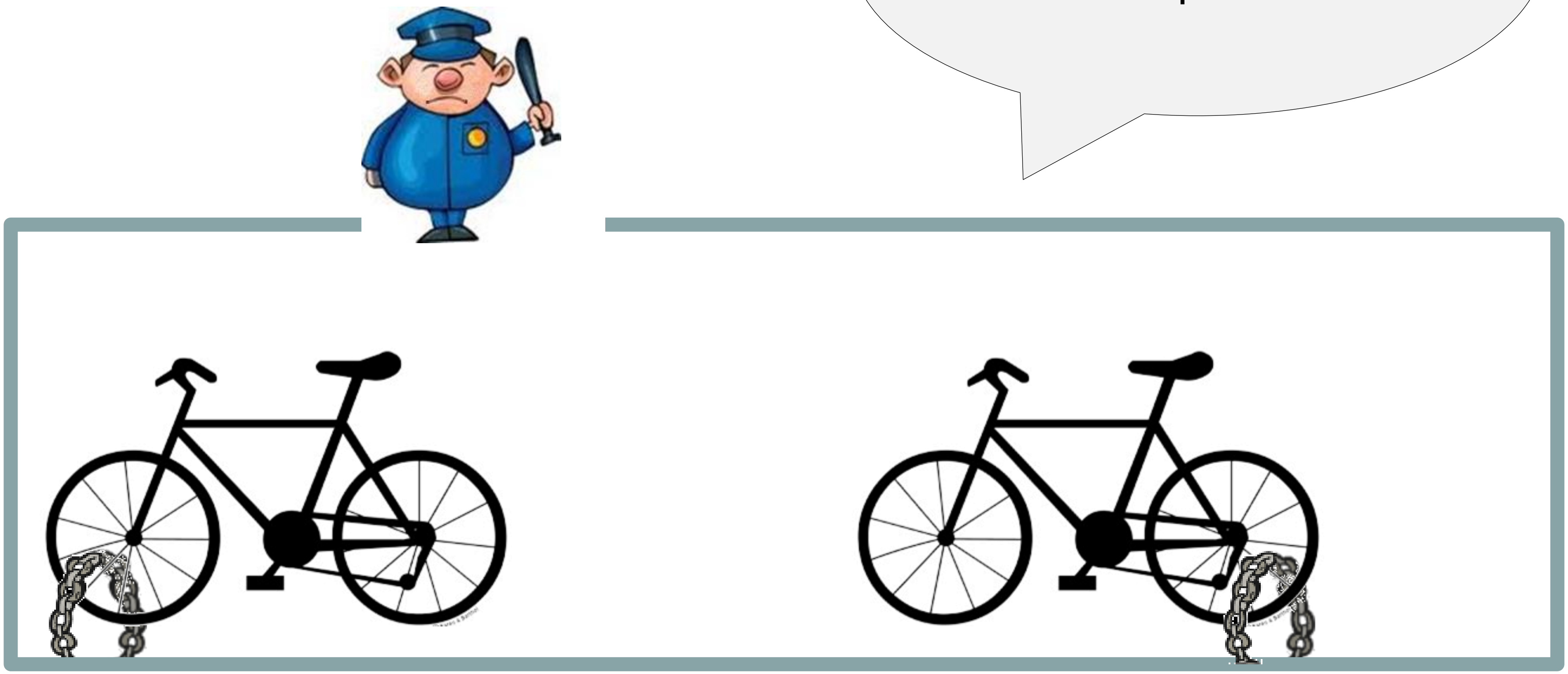


Secure !

Composition of Security Guarantees

Only complete bicycles are a

Security properties are non-compositional !



Insecure !

Concurrent shared variable programs are non-compositional

```
long long x;

Thread1() {
        x = 1;
}
// @post: x == 1


Thread2() {
        x = (1 << 64);
}
// @post: x == (1 << 64)



(Thread1() || Thread2());
// @post: x == 1 or x == (1<<64)
```

Global variable

Post conditions hold in absence of concurrent threads

Does composition hold?

Concurrent shared variable programs are non-compositional

```
long long x;

(Thread1() || Thread2());

// @post: x == 1 or x == (1<<64) or x == (1<<64) + 1
```

- This post-condition cannot be derived from any logical composition of the original post-conditions of `Thread1()` and `Thread2()`

- For writing a 128bit integer to memory, two writes on the memory bus are required. As a consequence, the final value of `x` may also be `(1<<64) + 1`

Vertical Structuring - Refinement

- **Idea**: start at an abstract description and add details step by step

From abstract specification to an implementation

- What do we want to refine?
  - Algorithm: algebraic refinement
  - Data: data refinement
  - Process: process refinement
  - Events: action refinement

Algebraic Refinement

Stack

empty: stack;
pop(stack):stack;
push(int, stack):stack

pop(empty) = empty
pop(push(x, y)) = y

Refinement

Satisfies

To prove:

safetail([]) == []
safetail(y:xs) == y

Implementing stacks by lists

empty $\mapsto$ []
push $\mapsto$ (:)
pop $\mapsto$ safetail

List

```
[]     :: [a]
head :: [a]-> a
(:)    :: a-> [a]-> [a]
tailSafe :: [a]-> [a]

tailSafe xs = if null xs then [] else tail xs
```

Refinement preserves properties of stack by transitivity of the logic !

Even More Refinements

- Data refinement
  - Abstract datatype is „implemented" in terms of the more concrete datatype
  - Simple example: define stack with lists
- Process refinement
  - Process is refined by excluding certain runs
  - Refinement as a reduction of underspecification by eliminating possible behaviours
- Action refinement
  - Action is refined by a sequence of actions
  - E.g. a stub for a procedure is refined to an executable procedure

Conclusion & Summary

- Software development models: structure vs. flexibility
- Safety standards such as IEC 61508, DO-178B suggest development according to V-model.
  - Specification and implementation linked by verification and validation.
  - Variety of artefacts produced at each stage, which have to be subjected to external review.
- Safety / Security Requirements
  - Properties: sets of traces
  - Hyperproperties: sets of properties
- Structuring of the development:
  - Horizontal – e.g. composition
  - Vertical – refinement (e.g. algebraic, data, process…)