Software Synthesis and Verification

Prof. Jüri Vain
Tallinn University of Technology

Why formal methods?

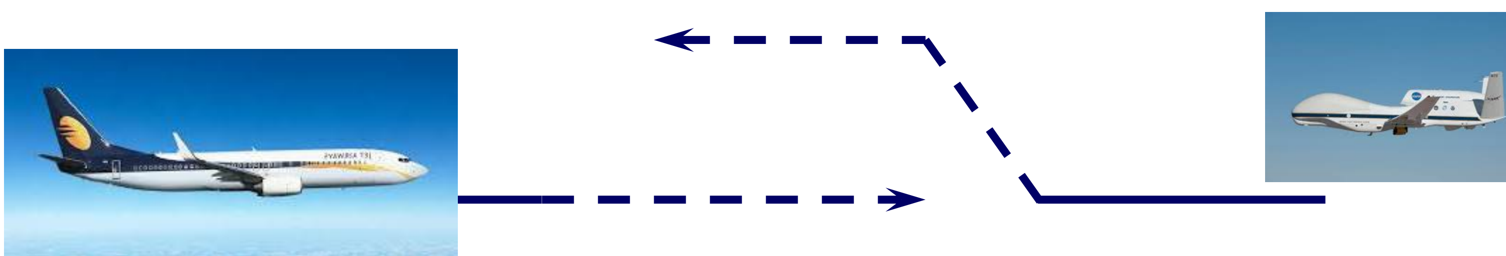
Auto-pilot example

Problem

Design a module for airplane auto-pilot that avoids collision with other airplanes.

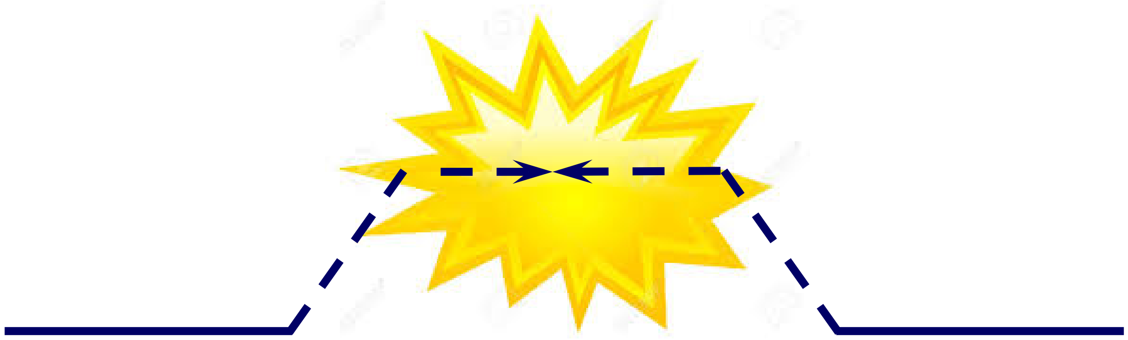
Possible design solution (conceptually):

When distance is 3km, give warning to approaching plane and notify own pilot. When distance is 1km, and no course change is taken, go up.



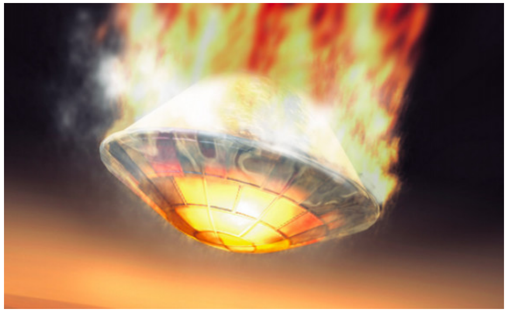
Problem with (bad) solutions

- Assume both planes have the same collision avoidance software. Then both go up and ...



This happens in real software!

- Some famous bugs
  - Several NASA space missions have been lost
  - Intel floating point processor bug
  - A military aircraft flipped when crossing the equator.
  - Bug in US F-16 software: aircraft control lost when flying low over Dead Sea (altitude < normal sea level)



What makes CPS design so hard?

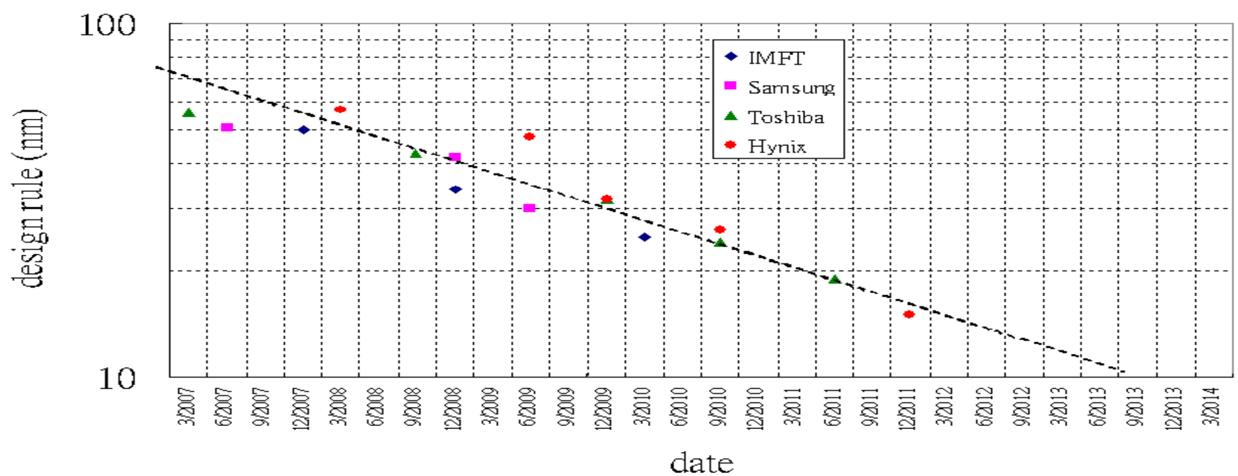
Common characteristic of CPS: Complexity & Heterogeneity

- Multi-scale technology integration and mapping that:
  - data gathering from simple sensors … data warehouses;
  - data, computing, and services are distributed in the cloud;
  - access to data and services via various end-user utilities.
- The architecture is open and dynamic with
  - heterogeneous networking and
  - heterogeneous components.
- High level of concurrency with
  complex interactions where
  - the location of data&computation
  - and timing is critical

Increasing performance of core technologies

Moore's Law:

The performance of microprocessors doubles every 18 months



Proebsting's law: Compiler technology doubles the performance of programs every 18 years

Increasing performance of core technologies

- ***Gilder's Telecom Law***:

  3x bandwidth/year for 25 more years

- in 1996: whole US WAN bisection bandwidth was 1 Tbps
- in 2014:
  - TUE and U. of Central Florida have ashieved **255 terabits** per second per optical fiber, i.e. 5.1 terabits per carrier

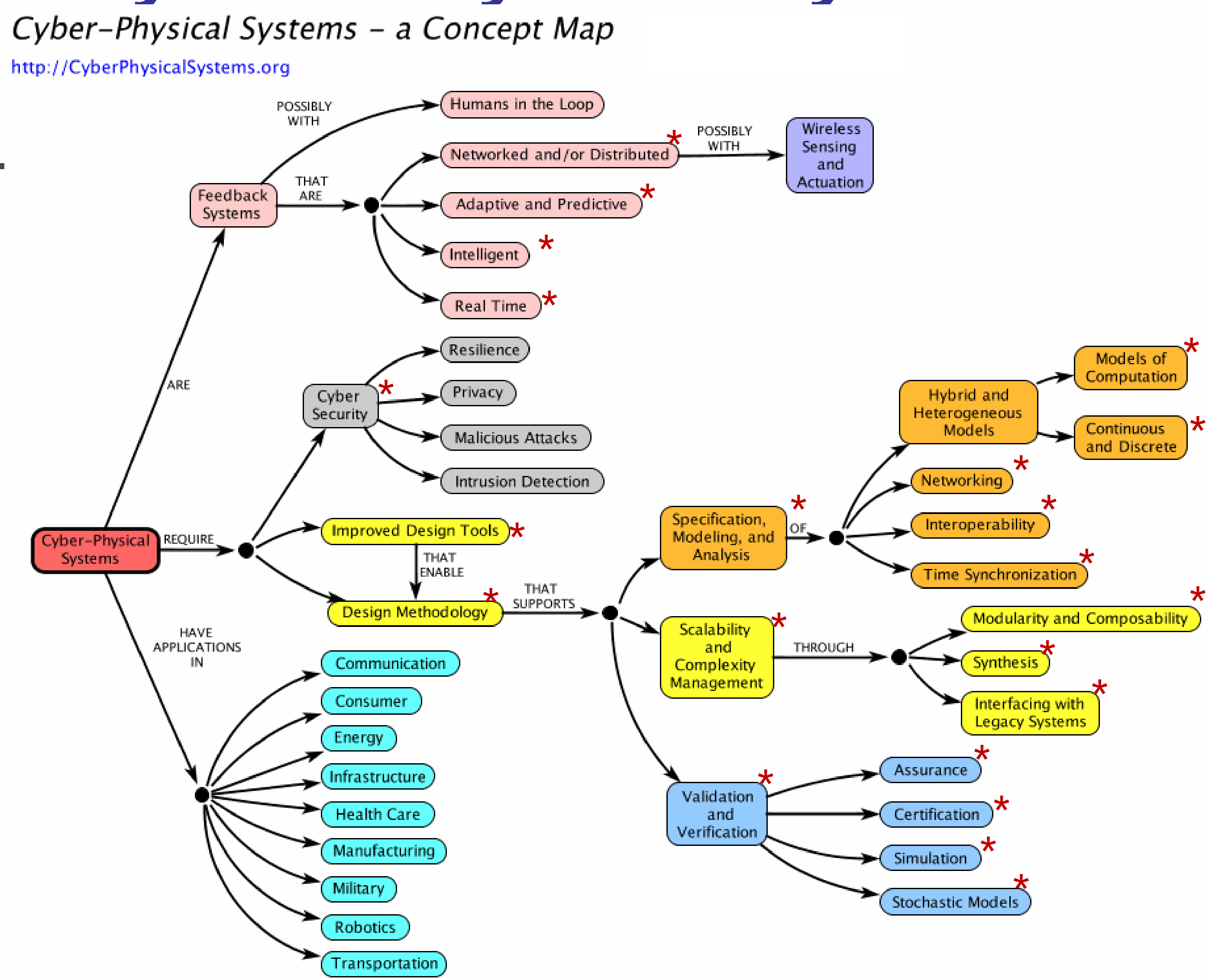
Increasing dependability requirements

- Dependability is systems' and services' integral measure that captures
  - availability
  - reliability
  - maintainability
  - durability
  - safety
  - security
  - ...

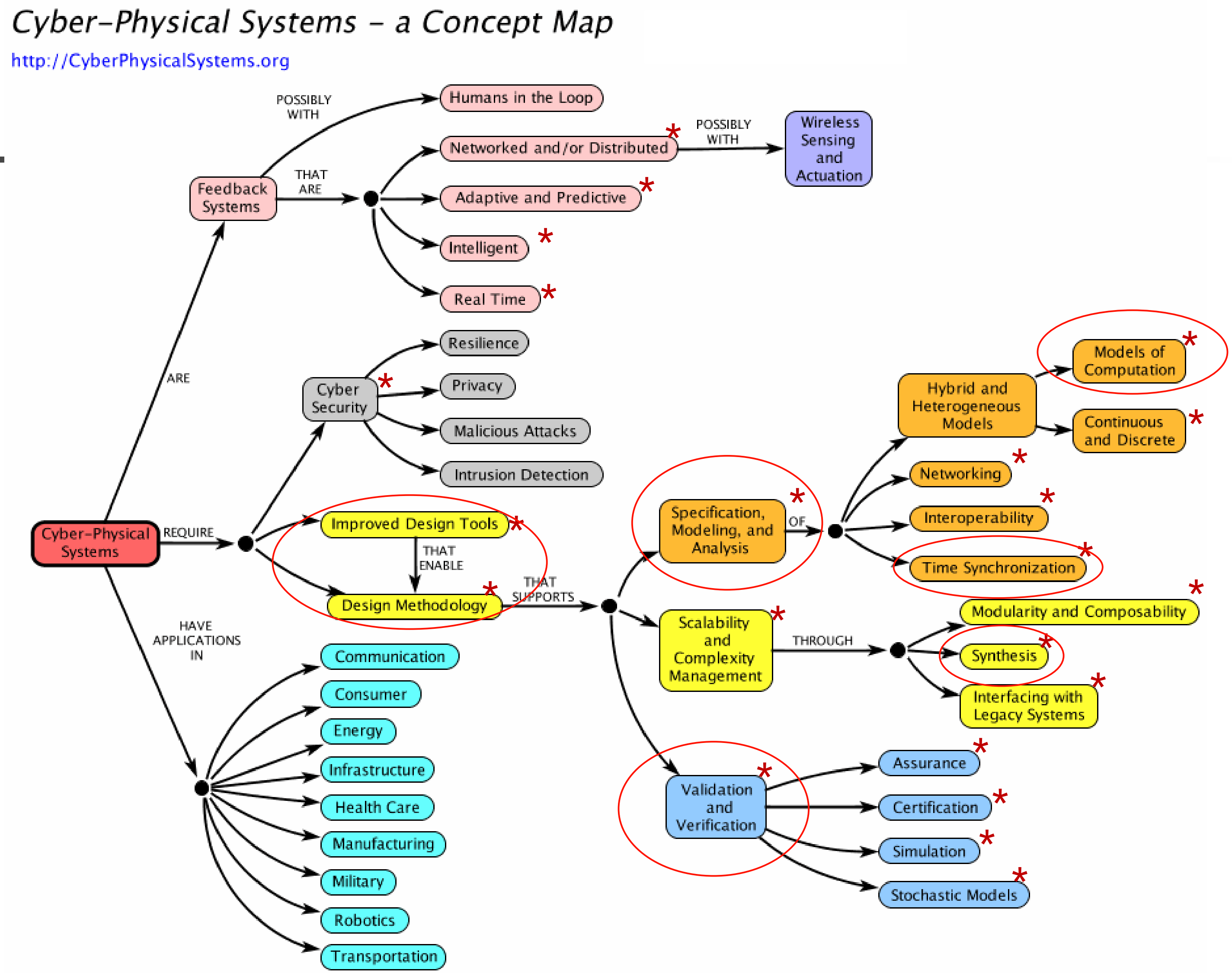
Ariane 5 accident

The launch failure brought the risks of complex systems to the attention of general public. The subsequent automated analysis of the Ariane code was the first example of large-scale static code analysis by abstract interpretation. This led to the discipline of dependability for * - critical systems.

Cyber-Physical Systems



Cyber-Physical Systems



Implications of complexity & dependability for CPS development process

- Quality dilemma: drop the quality for more features
- Test and verification are the bottlenecks in design processes
- Error detection/diagnosis/repairment constitute 50-70% of costs

Design task	Tasks delayed (%)			Tasks causing delay (%)		
	Auto-motive	Auto-mation	Medical	Auto-motive	Auto-mation	Medical
System integration T & V	63	56,5	66,7	42,3	19	37,5
System Archi-tecture D&S	29,6	26,1	33,3	38,5	42,9	31,3
SW application and/or middleware D & T	44,4	30,4	75	26,9	31	25
Project management & planning	37	28,3	16,7	53,8	38,1	37,5

Source: Inria research report n° 8147 november 2012

CPS design challenges (I): heterogeneity of requirements

- Technology integration and mapping,
- Reliability and resilience,
- Power and energy consumption,
- Security,
- Diagnostics,
- Run-time management,
- Real-time feedback,
- ....

CPS design challenges (II): Complexity

- Strong dependency between design aspects (functionality, safety, security, ...)
  → how to assure the coherence of aspects?
- Variety of control/communication/coordination scales from minuscule pace makers to national power-grids
  → how the time scales match?
- High level of concurrency with complex interactions
  → how to explore the full state space?
- Wide spectrum of timing requirements of interacting components.
  → is design feasible regarding timing constrstraints

How can formal methods/tools address CPS design challenges?

FM must support

- Model-based development with rigorous formal semantic basis.
- Scalability and extendability of methods and tools.
- Complience with state-of-the-art programming technologies and standards.
- Simplicity of use in industrial engineering.

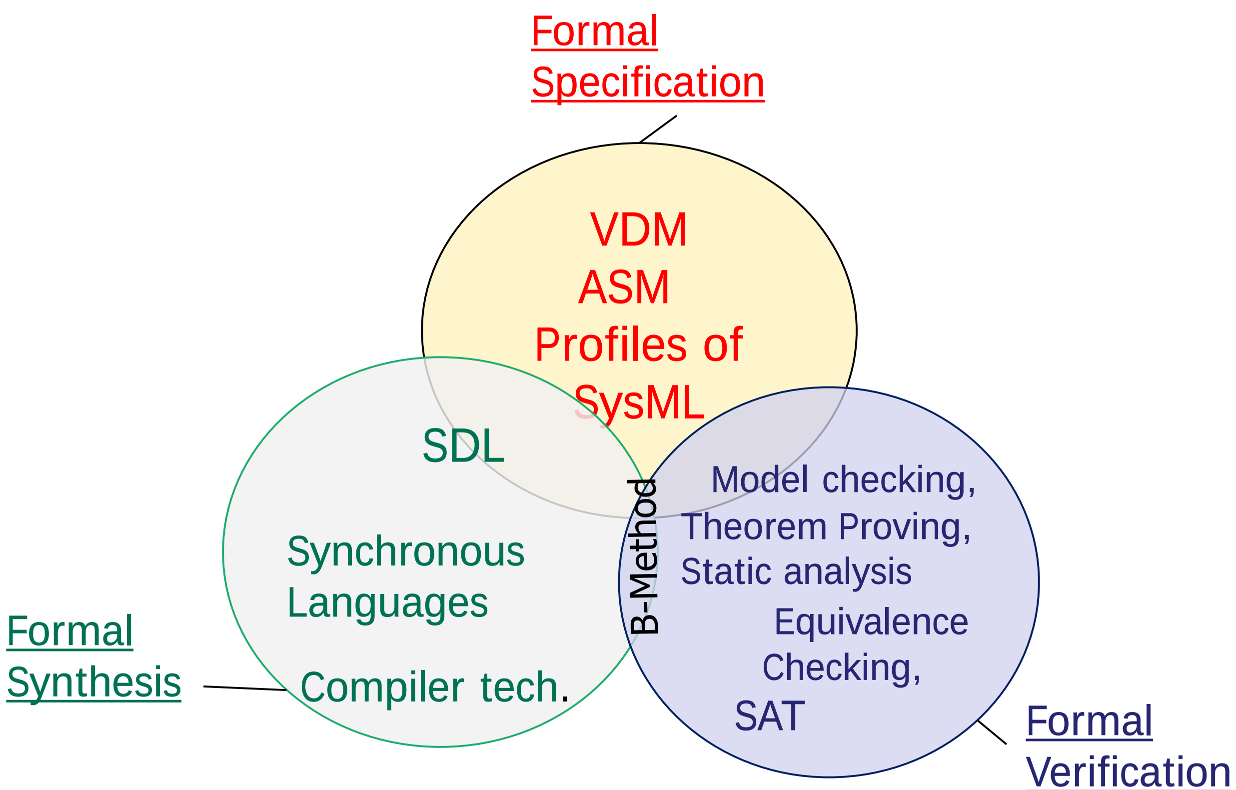
Formal methods in a nutshell

- FMs deal with *formal notation* – *state*, *data type*, *refinement*,…
- Formal notation has *rigorous semantics*
- FMs emphasize
  - *symbolic reasoning*
  - *transformations*
  - *analysis*

  of *abstract formal notations*.
- FMs is not esoteric science,

  e.g. compilation in a broad sense is a FM: high-level notation is transformed to low-level executable notation.

Taxonomy of formal methods



Formal Specification (1)

- Given:
  - *possibly unstructured, fragmented, incomplete, … descriptions of the system or its requirements expressed in different informal forms of representation.*
- Goal:
  - *express this knowledge about the structure, behavior, properties in some formal language.*

Formal Specification (2)

- *abstracts* from unnecessary implementation details
- provides *rigorous mathematical* semantics
- abstraction allows high-level reasoning while implementation details are not clear yet
- allows to avoid ambiguous or inconsistent specifications.

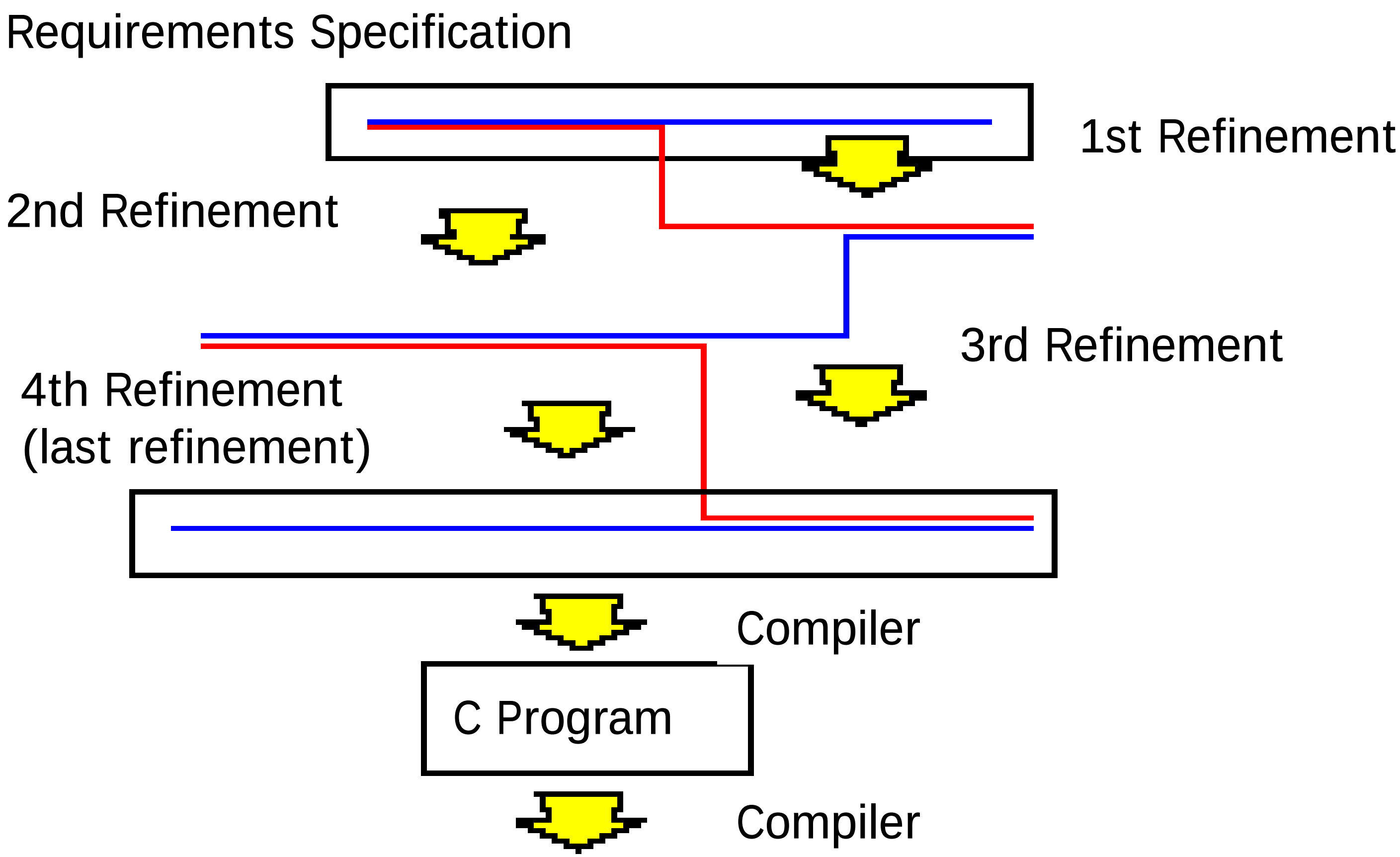
Challenges:

- Specification refinement/ consistency checks/ aspect extraction are difficult to comprehend by engineers without theoretical training.
- Elaboration of domain oriented languages and their mapping to standard formalisms is required

Formal Synthesis (I)

- Given:
  - Reguirements to the artefact to be synthesised
  - (Possibly) design templates, patterns..., components
  - Design constraints (structure, behavior, properties,...)
  - Other constraints (cost, ethical, aesthetic, ...)
- Goal:
  - Construct architecture, control structure, data, code

Refinement based Synthesis (I)



Refinement based synthesis (II)

- Integrates the refinement steps with verification.
- Incremental refinement steps are guided by domain heuristics.
- Refinement correctness verification is based on components'. specifications and their composition rules i.e. compositionality.
- Proofs can be automatized but are computationally expensive.
- Refinement steps may be eiher
  - correct by construction or
  - *'invent-and-verify'*.
- Example: B-Method and Rodin tool (Event-B.org)

Formal Verification

Given: system requirements specification and implementation

Prove: that implementation *satisfies* the requirements specification

- Full blown "post mortem verification" is too complex
- Simplifications applied are:
  - partial specifications (slicing, aspect orientation, contracts):
    - type safety,
    - functional equivalence of systems,...
  - compositionality (deduce the correctness of whole from the correctness of components);
  - property preserving abstractions, and reduction techniques.

Classes of verification methods

- Boolean methods:

  SAT, BDDs, ATPG, combinational equivalence check
- Finite state methods:

  bisimulation and equivalence checking of automata, model checking (MC)
- Term based methods:

  term rewriting, resolution, tableaux, theorem proving
- Abstraction based methods

  BDDs, symbolic MC, theorem proving, SMT constraint solving

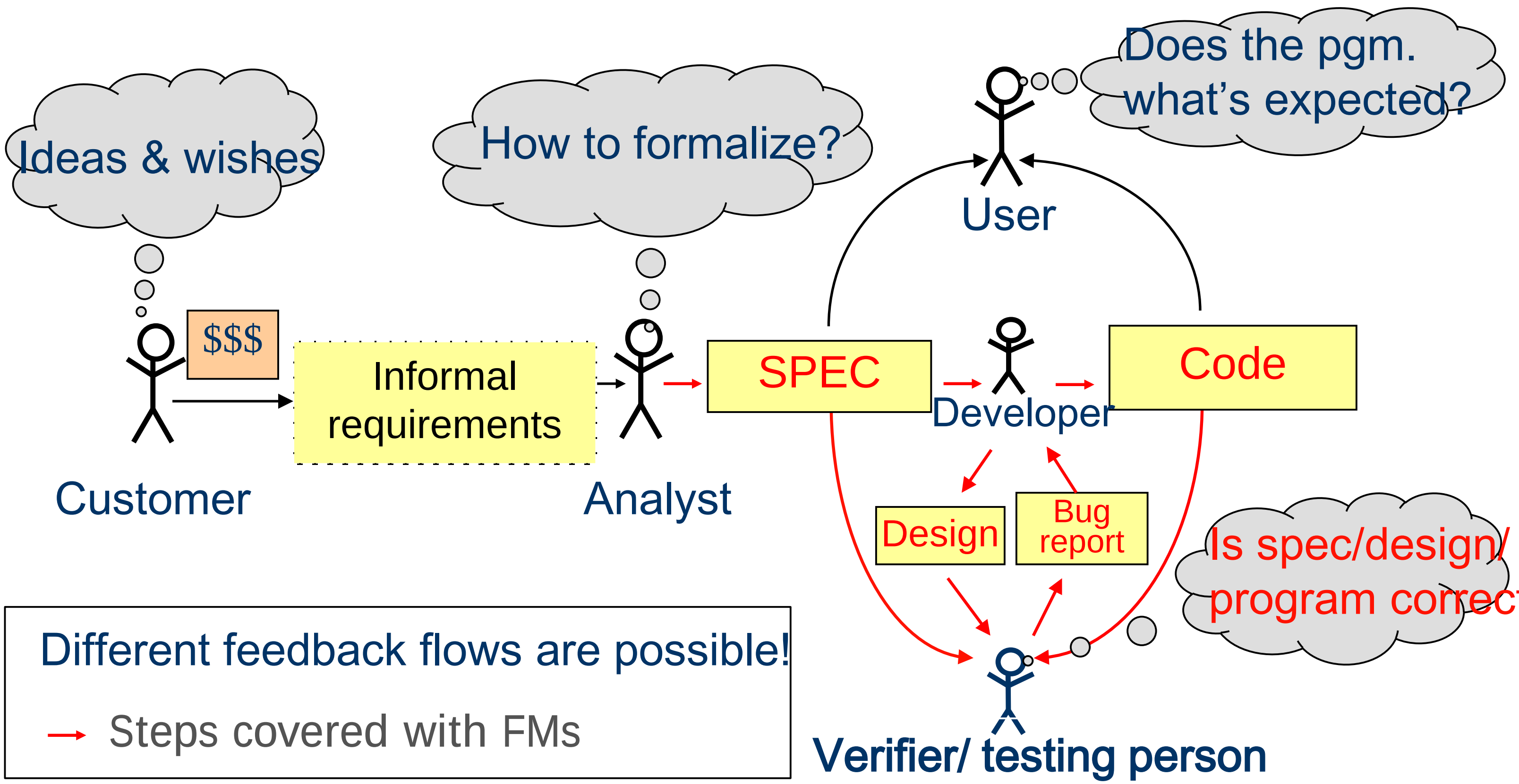
Software Oriented Formal Methods

- Model-based testing (MBT)
- Deductive verification
- Model checking (automatic verification)
- Static analysis
- SMT- constraint solving
- Combinations of the above

Test & Verification

- **Testing**
  - <u>dynamic</u> execution / <u>simulation</u> of system runs
  - *Present view*: tests have to be integrated in the development process
  - *Extreme view*: testing should “drive” the development process
- **Verification**
  - *Means*: <u>static</u> checking, <u>symbolic</u> execution.
  - In HW design community: verification means also testing
- **In FM community Testing ≠ Verification**
  - Testing is <u>*partial*</u> exploration method (not all executions are covered)
  - Verification is <u>*complete*</u> method but more costly than testing

Verification: process and parties



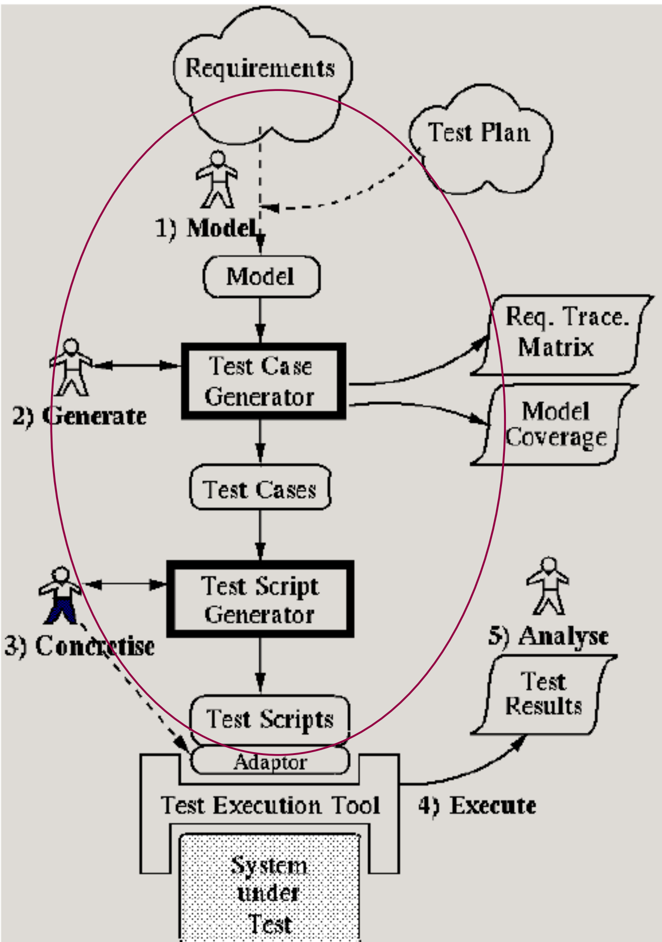
Different feedback flows are possible!

→ Steps covered with FMs

(Traditional) testing

- Executing the software in order to exercise and discover errors
- Still most handy and common method in sw industry **+**
- Partially manual, some automation tools exist (for running tests, organizing test data and reporting) **-**
- Applicable directly on executable software **+**
- Not exhaustive, errors often survive **-**
- Depends on tester's intuition and experience **+**/**-**
- Formal spec is not needed **+**/**-**

Model-Based Testing (MBT)



- **Goal:** Check if real system conforms with requirements specification.
- **Advantages/disadvantages**

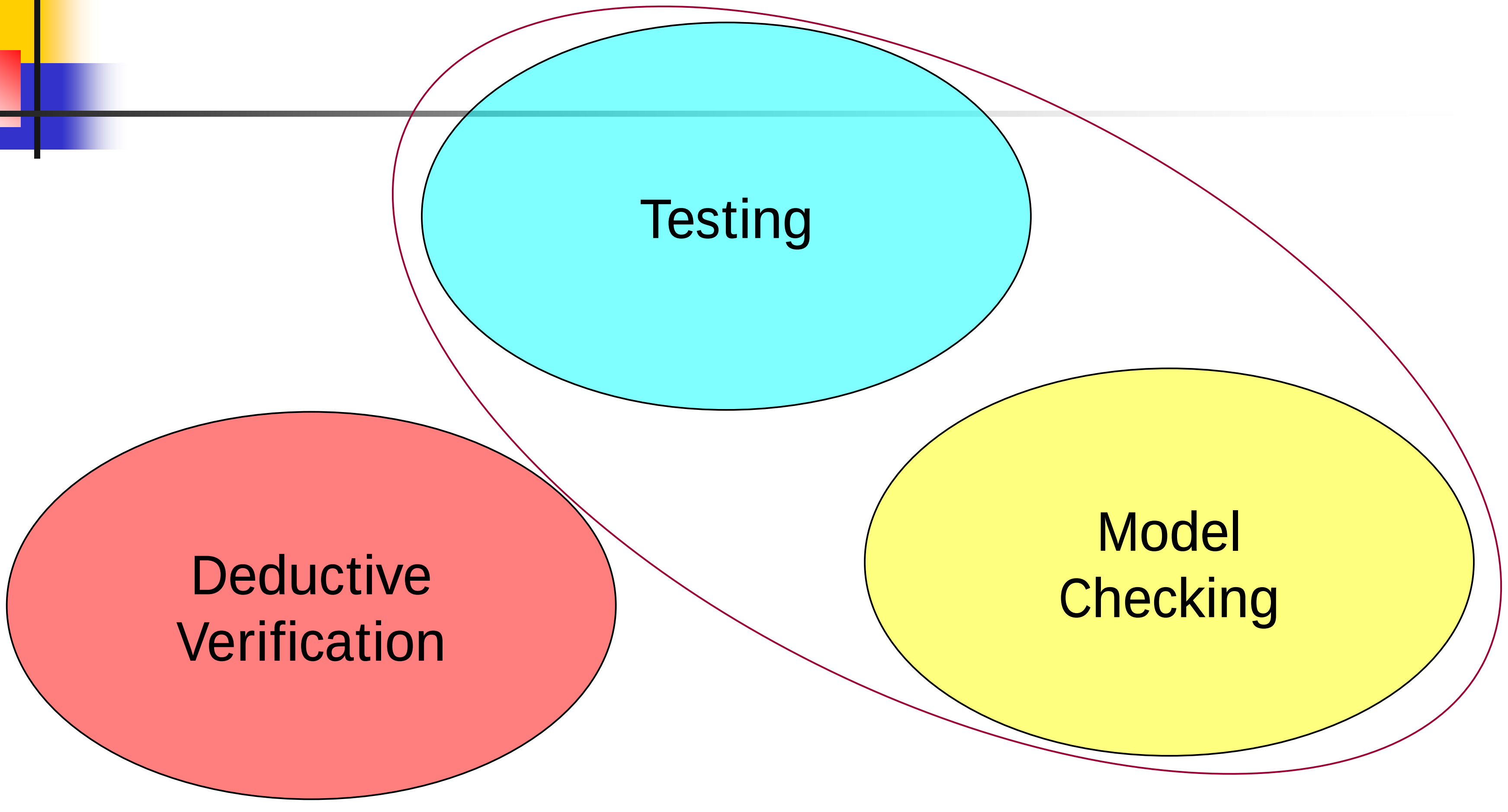
  \+ model hides irrelevant details of implementation;

  \+ automatic generation and execution of tests;

  \+ systematic coverage of requirements

  \+ relevant for **regression testing**

  \- modeling overhead!


Testing
Deductive Verification
Model Checking

Model checking

Model Checking

Given a model *M* and a property *P*, check if *M satisfies P*

- Exhaustive state space exploration method.
- Uses graph theory and automata theory to decide on properties of programs algorithmically
- State space explosion: complexity of the problem or bad modeling causes exponential memory and time growth
- Due to algorithmic state space exploration the method is limited, suites for finite state systems
- But there are many heuristics and techniques to reduce time and memory space

Deductive Verification

Deductive Verification

Aplies *theories* and *logic inference* to prove properties of system **specification** formally

- Is based on proof theory and techniques +
- When doable provides *full certainty* of correctness +

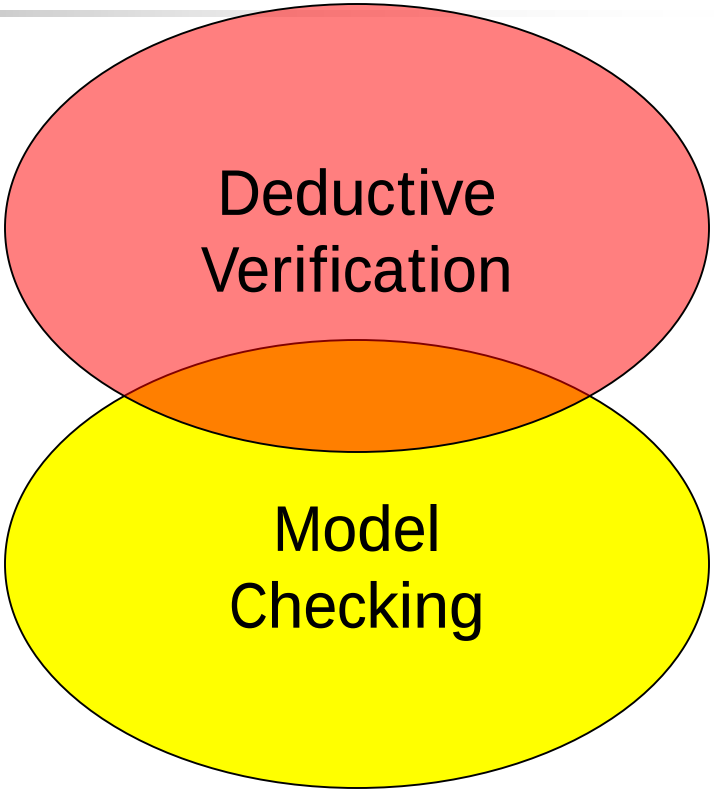
- Requires expertise in logic, math and tools usage -
- Highly time consuming (inereactive) -
- Susceptible to discrepancies between sw and model -
- Practical only with tool support -
- Applicable on small and medium size examples -
- Requires accurate specification -

Comparing verification methods

Method / Criterion	Testing	Deductive Verification	Model Checking
Size of system	Small-Very large	Limited examples	100s-1000s lines
Time	Minutes-Hours	Days-Weeks	Minutes-Hours
Expertise needed	Test engineers/ programmers	Mathematicians, Comp-Sci., Logic.	Comp.-Scientists/ sw engineers
Popularity	SW/HW industry	Mostly research	Reserch/industry
Specification	Informal requirement docs	Logic or automata based	Logic or automata based
Modelling / corrections	Not needed / code correction	Must /via formal representation	Must/via formal representation

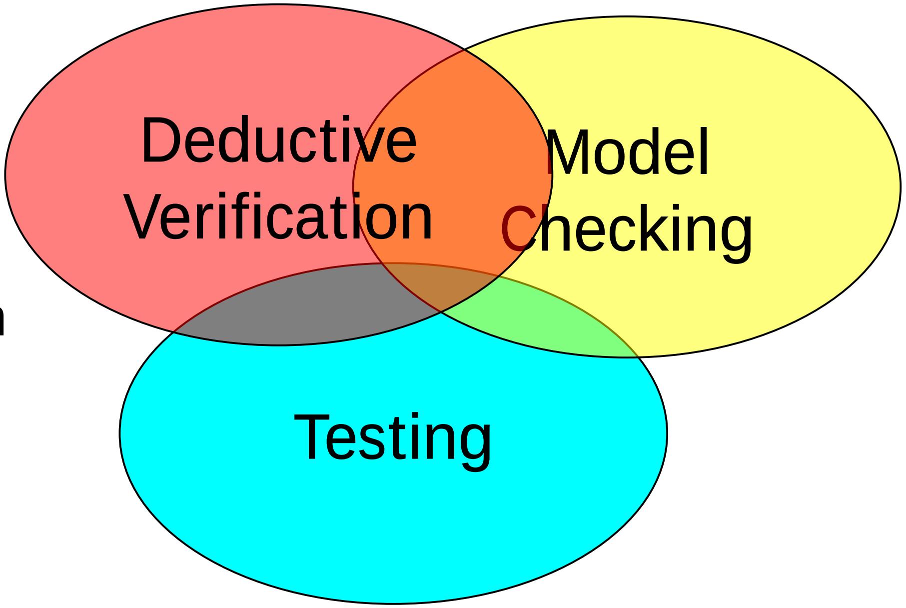
Integrated FM (I): Symbolic model checking

- General startegy
  - Find symbolic states and transitions by proving equivalences of explicit states.
  - Then apply model checking on this finite abstractions.



Integrating formal methods (II): Symbolic Verification / Testing

- Apply abstraction techniques to generate symbolic states and transitions of the system model.
- Apply symbolic model checking to generate abstract witness traces for temporal formulas that describe the test goals.
- Apply these witness traces as test sequences instantiated with concrete test data.



Instead of summary: Current trends of FM

- Trying to solve special cases of generally undecidable or highly complex problems.
- Improving usability
- Integrating mutually complementing FMs
- Building industry strength aka scalable tools
- Applying parallelization, distributed and high-performance computing
- Combing FM with AI and soft computing techniques.

Questions?