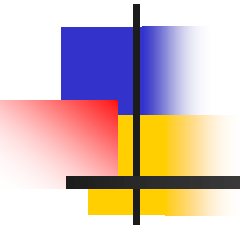


Formal methods
(in RT embedded systems design)

Prof. Jüri Vain

Spring 2015

ITI0130, ITI8530

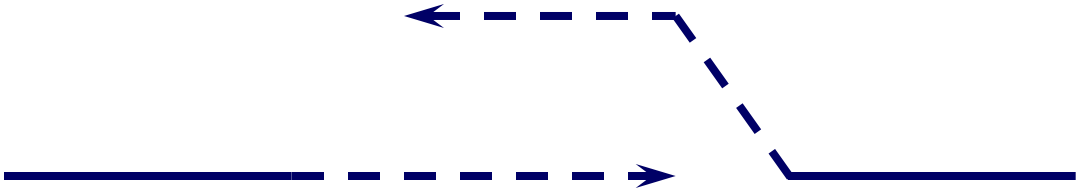
Formal methods – why? Example: auto-pilot

Problem:

Design a module in aircraft auto-pilot that avoids collision with other planes.

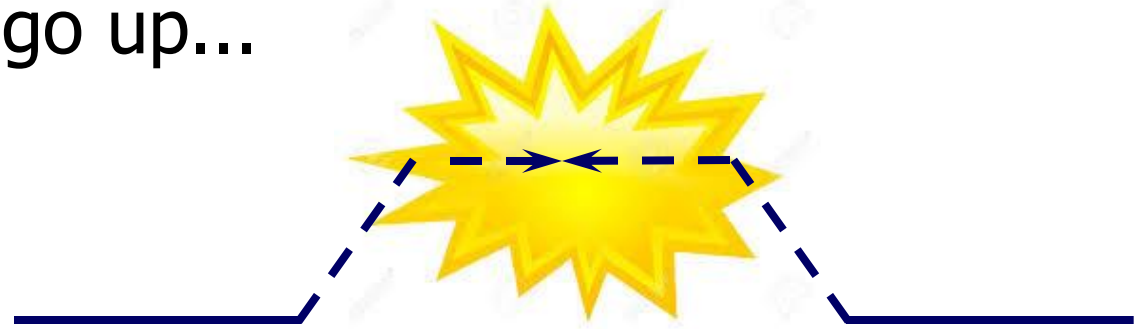
One possible design solution:

When distance is 1km, give warning to other plane and notify pilot. When distance is 300m, and no changes in the course of other plane, go up.



Problem with solution

- Both planes have the same software. Both go up...



This happens in real software!

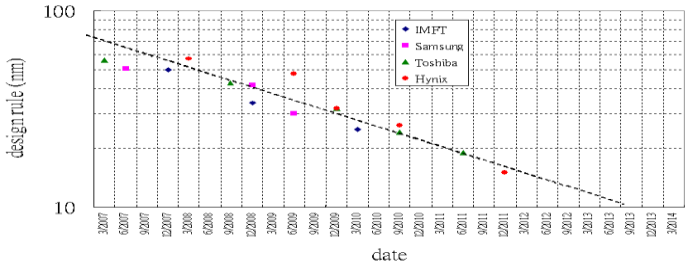
- Some famous bugs
  - several NASA space missions lost,
  - Intel floating point processor, etc.
- Hard to predict all behaviours!
  - US aircraft went to southern hemisphere and … flipped when crossing the equator
  - Software written for US F-16
    - accidents when reused in Israeli aircraft Dead Sea

(altitude < sea level)

Why FMs? (I)
Increasing complexity/costs of system development

Moore's Law:

The performance of microprocessors doubles every 18 months



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

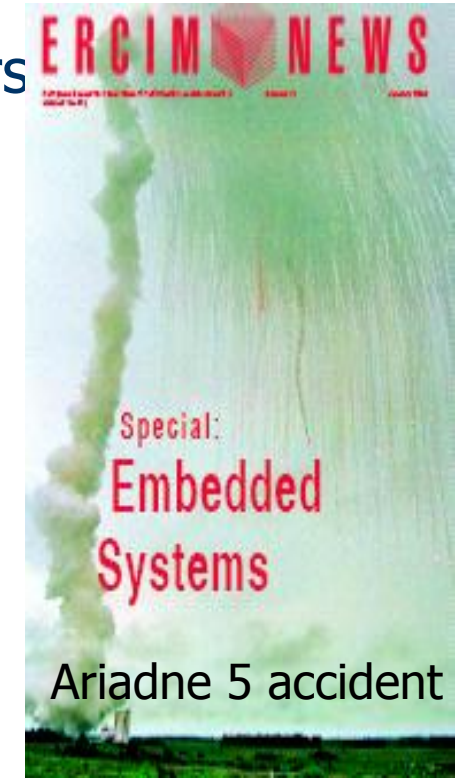
Gilder's Telecosom Law: 3x bandwidth/year for 25 more years

- Today:
  - 10 Gbps per channel
  - 4 channels per fiber: 40 Gbps
  - 32 fibers/bundle = 1.2 Tbps/bundle
- In lab 3 Tbps/fiber (400 x WDM)
- In theory 25 Tbps per fiber
- 1 Tbps = USA 1996 WAN bisection bandwidth

Design costs tend to grow faster than the size of the system

Why FMs? (II)
Increasing dependability of systems

- Everything important depends on computers
  - stir by wire aircrafts
  - banking
  - stock market
  - manufacturing workflow, …
- Quality concerns due to the increasing
  - functionality
  - security
  - mobility
  - new business processes, …



The launch failure brought the high risks associated with complex computing systems to the attention of the general public, politicians, and executives, resulting in increased support for research on ensuring the reliability of safety-critical systems. The subsequent automated analysis of the Ariane code was the first example of large-scale static code analysis by abstract interpretation

Implications of complexity & dependability growth

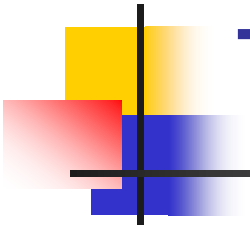
- Quality dilemma: drop quality for more features
- Testing and verification are the bottlenecks of sw processes
- Typically > 50% of development costs are spent on error detection/diagnosis/repairment

⇒ FM research challenge: find efficient methods for sw synthesis, test and verification

⇒ Trends: combine FM and testing in the sw process
FMs for isolated tasks → integrating FMs into full life cycle

⇒ Current practice: MDD (Model Driven Development)

Experts in FM are *increasingly needed* in high-tech industry, specially in *cyber-physical systems* (robotics, smart energy grids, smart houses, mobile applications etc).



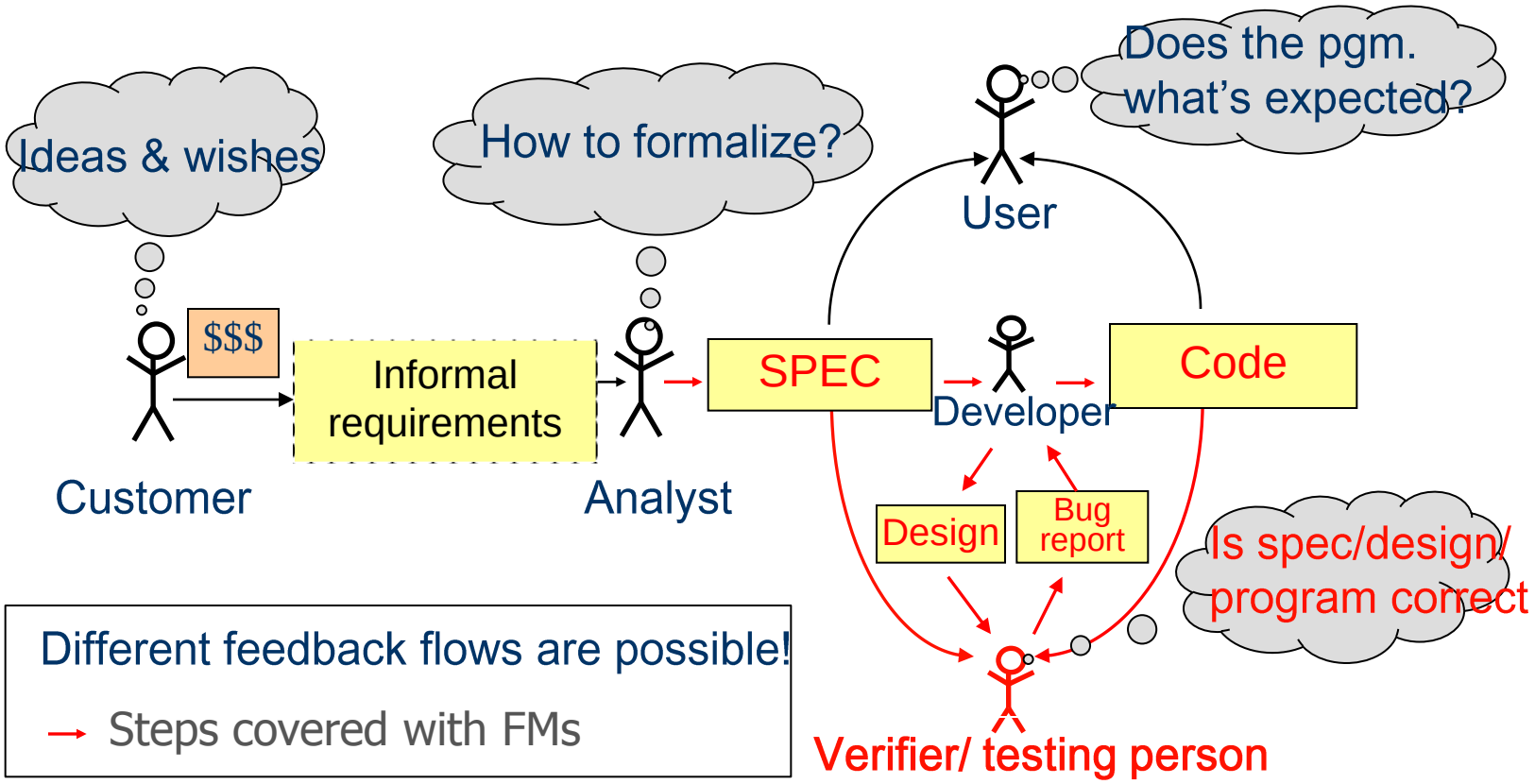
Test & Verification

- **Testing**
  - *Standard definition*: dynamic execution / simulation of a system
  - *Present view*: tests have to be integrated in development process
  - *Extreme view*: testing should "drive" the development process

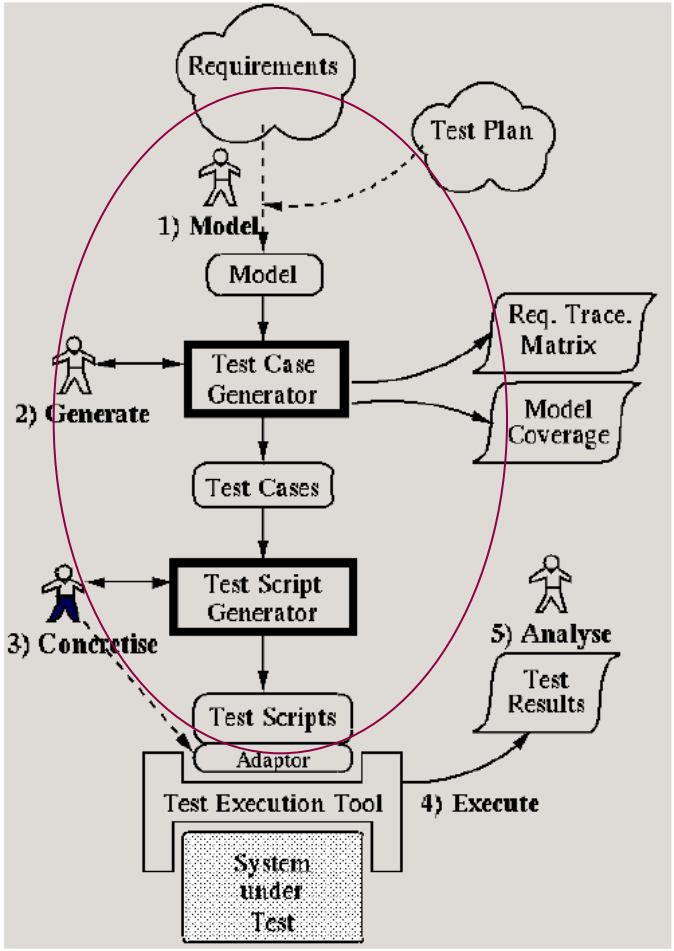
- **Verification**
  - *Standard definition*: static checking, symbolic execution.
  - In hw design community: verification means also testing

- **Our view**: Testing $\neq$ Verification
  - Testing is *partial* exploration method (not all executions are covered)
  - Verification is *complete* method but more costly than testing

Verification: process and actors



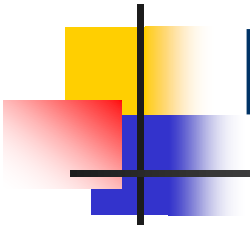
Closer look on Model-Based Testing (MBT) process



- **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 in **regression testing**
  - **- modeling overhead!**

Formal Methods in general

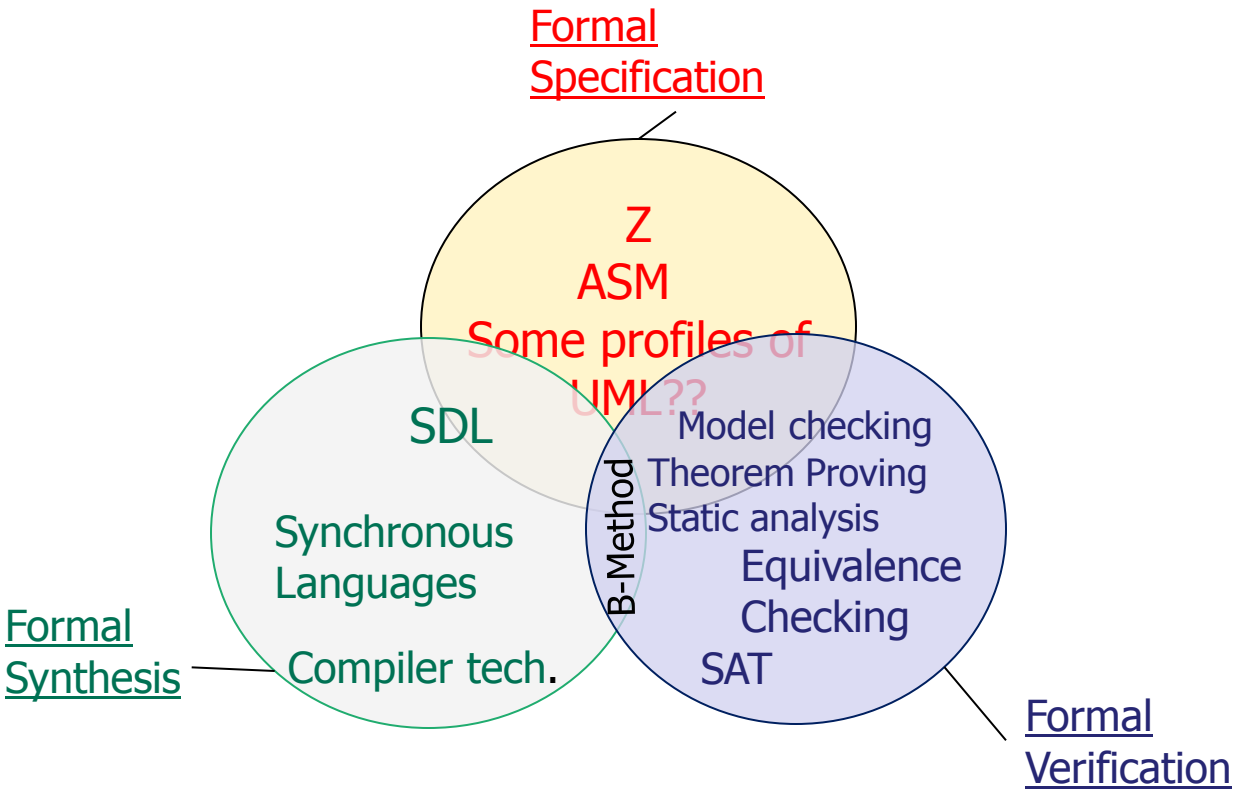
- FMs deal with *formal notations* – state, type, data refinement,...
- Formal notions have *rigorous semantics*
- Emphasizes *static / symbolic* reasoning about *abstractions*
  (standard definition of verification falls into this category)
- Too narrow view on FMs in digital design – covers only equivalence and model checking, but there is much more
- FMs are not esoteric, e.g. compilation in a broad sense is a FM
  (high-level description is translated into low-level description).

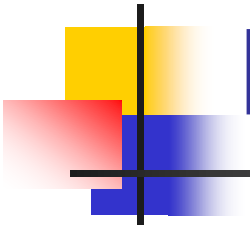


Focus of this course

- Tool-supported reasoning about the ***correctness*** of programs and systems.

Formal Methods: Classification





Formal Specification

= stating structure, behavior, properties of some artifact in formal way

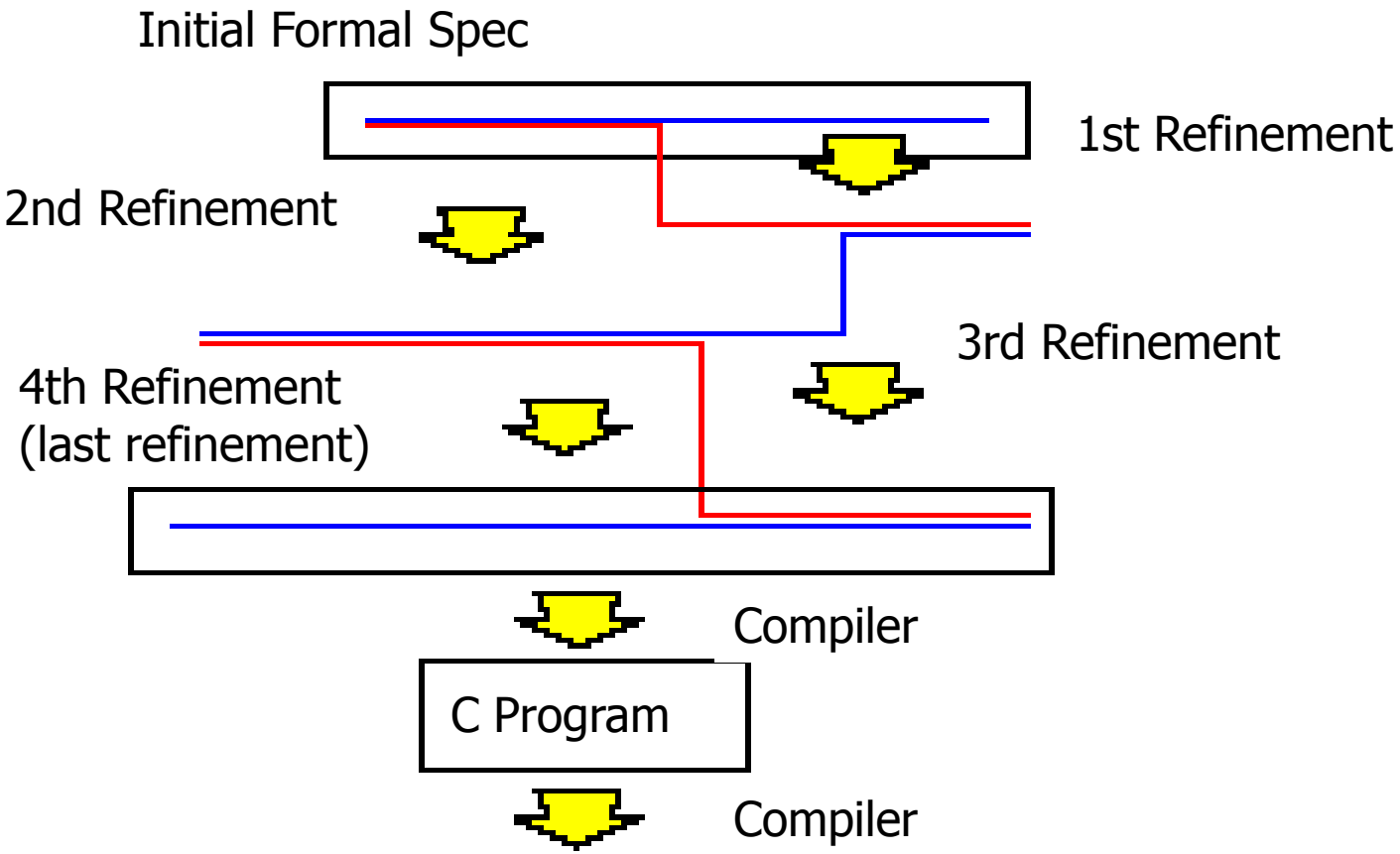
Formalization

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

Difficulties:

- Difficult to comprehend by engineers
- Few practical tools for refinement/ checking/ feature oriented specs
  - good example: ASM (Gurevich), B-method, Bogor,...

Formal Synthesis I

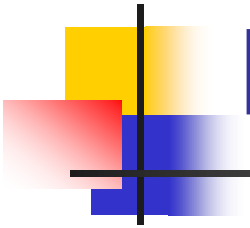


Formal Synthesis II

- integrates development process and verification
- incremental refinement steps guided by domain heuristics
- splits large verification tasks (*divide et empera*) …

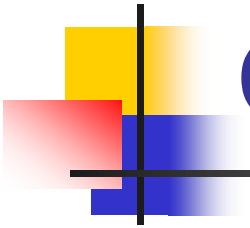
… but forces dramatic change in development process

- it works but it is *costly*
- each refinement step eiher
  - is correct by construction or
  - uses FMs for verification
- example: B-Method and Rodin tool



Formal Verification

- General assumption: requirements spec and system spec defined
- formal verification checks whether implementation representation satisfies requirements specification or not.
- full blown verification,e.g., "post mortem verification" is difficult.
- simplifications:
  - focus on simple partial specifications →
  - feature orientation:
    - type safety,
    - functional equivalence of systems,…
- methods (implemented in tools):
  - simple algorithms for deducing isolated properties directly
  - complex algorithms for hard or even generally undecidable problems



Classes of verification methods

- Boolean methods:

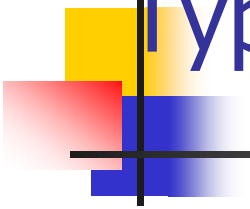
  SAT, BDDs, ATPG, combinational equivalence check
- Finite state methods:

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

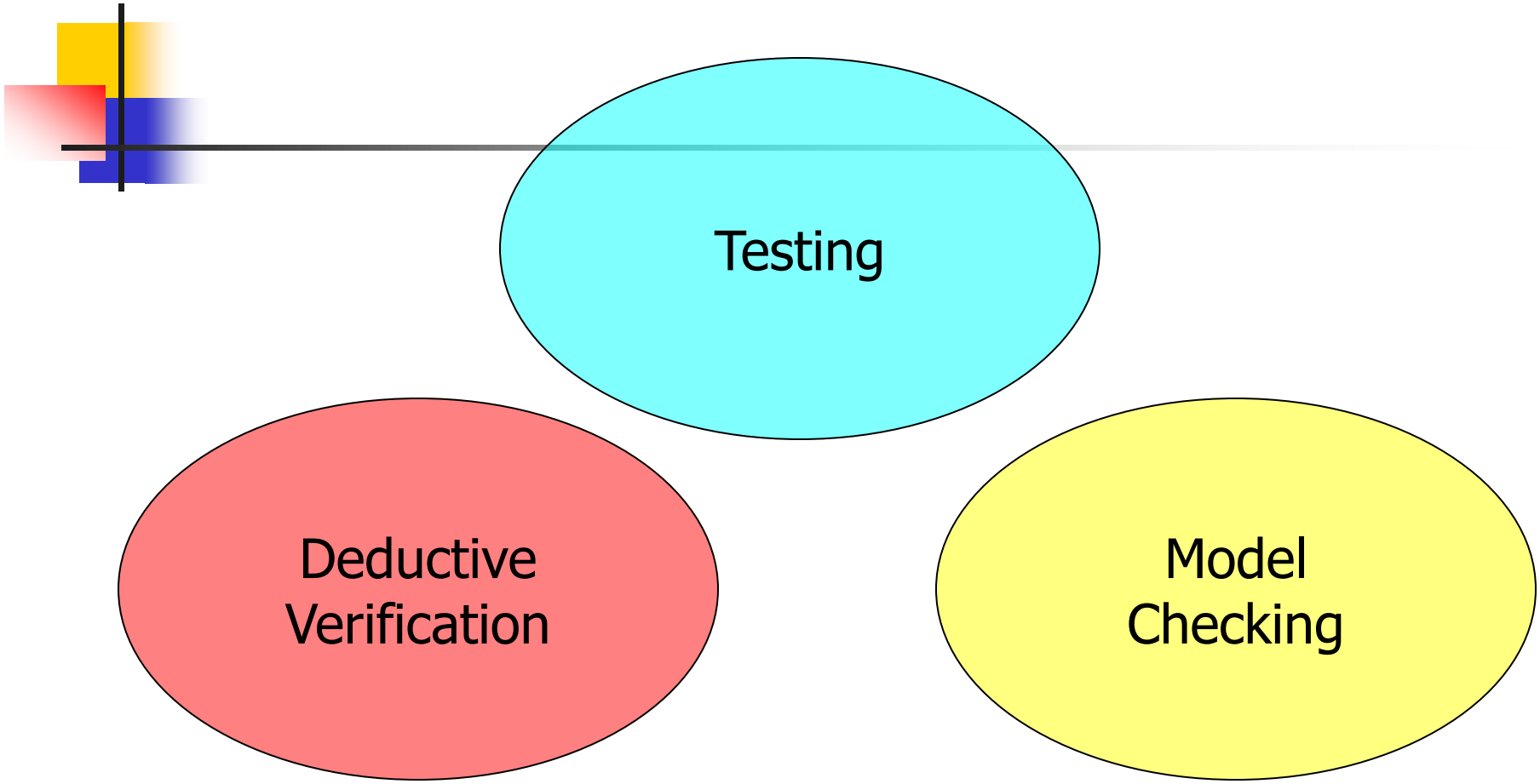
  term rewriting, resolution, tableaux, theorem proving
- Abstraction based methods

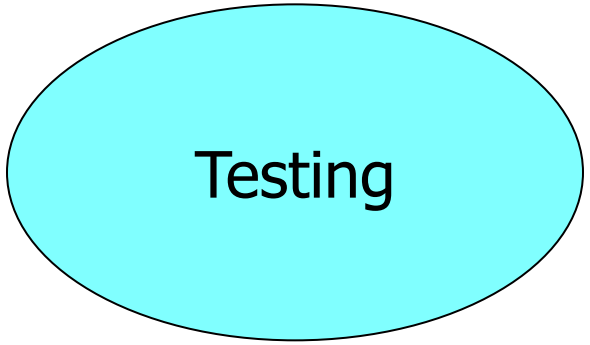
  BDDs, symbolic MC, theorem proving

Typical Formal Methods for Software



- Testing
- Deductive verification
- Model checking (automatic verification)
- Static analysis
- Combinations of the above

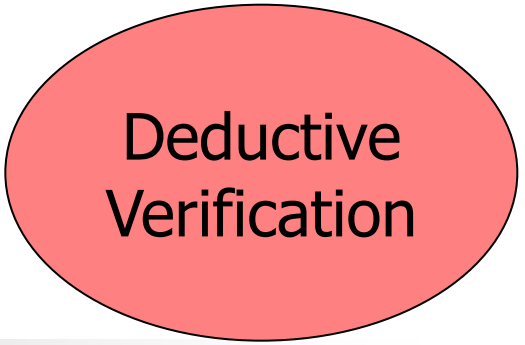

Testing
Deductive Verification
Model Checking



(Traditional) testing

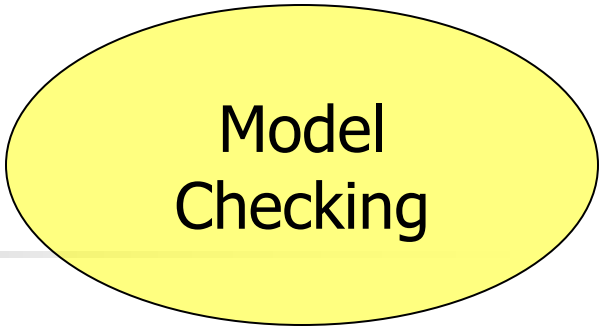
- Executing paths in the software in order to exercise (and discover) errors
- The traditional and still most common method in sw industry +
- Partially manual, some automation tools exist (for running tests and reporting) -
- Applied directly to software (some times small modifications necessary to support testing, e.g. resets) +
- Not comprehensive. Errors often survive -
- Based on intuition and experience of tester +/-
- Formal spec is not needed +/-

Deductive Verification



- Apply *theories* and *logic inference* to prove properties of a system **specification** formally
- Based on mathematical principles +
- Requires expertise in logic, math and tools usage -
- Highly time consuming -
- Susceptible to discrepancies between sw and model -
- Practical only with tool support -
- Applicable on small and medium size examples -
- Requires accurate specification -
- If doable provides full certainty of correctness +

Model Checking



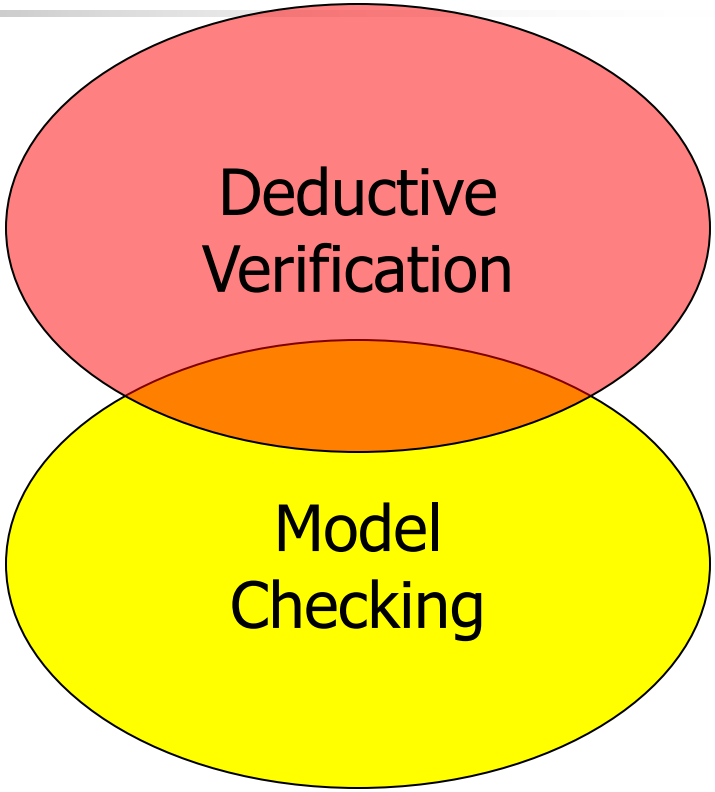
- Uses *graph theory* and *automata theory* to verify properties of programs *automatically*
- Requires modelling and specification
- State space explosion: often bad modeling causes insufficient memory and exponential time growth
- Algorithmic state space exploration makes it limited to finite state systems
- Many heuristics to reduce time/space

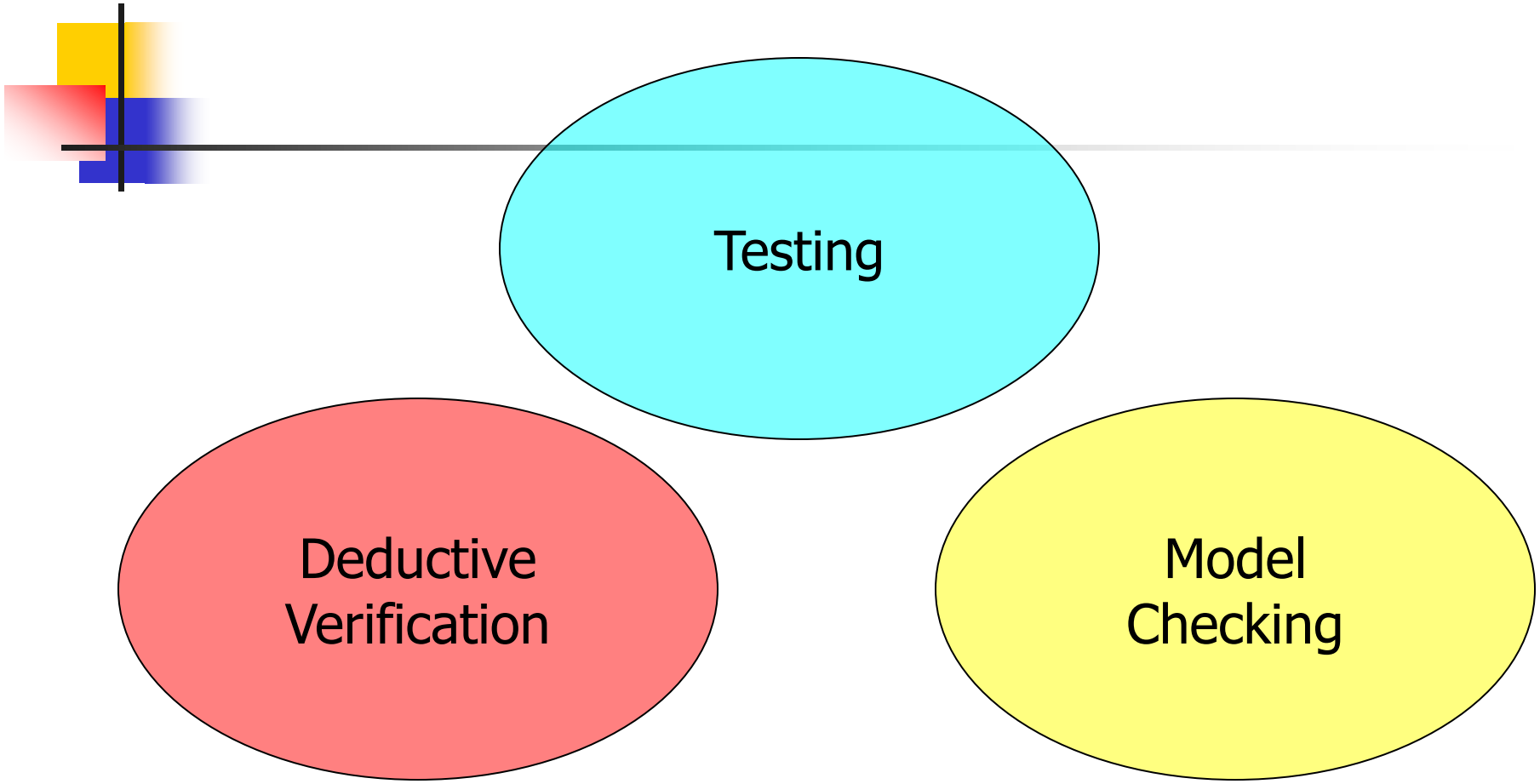
Comparing verification methods

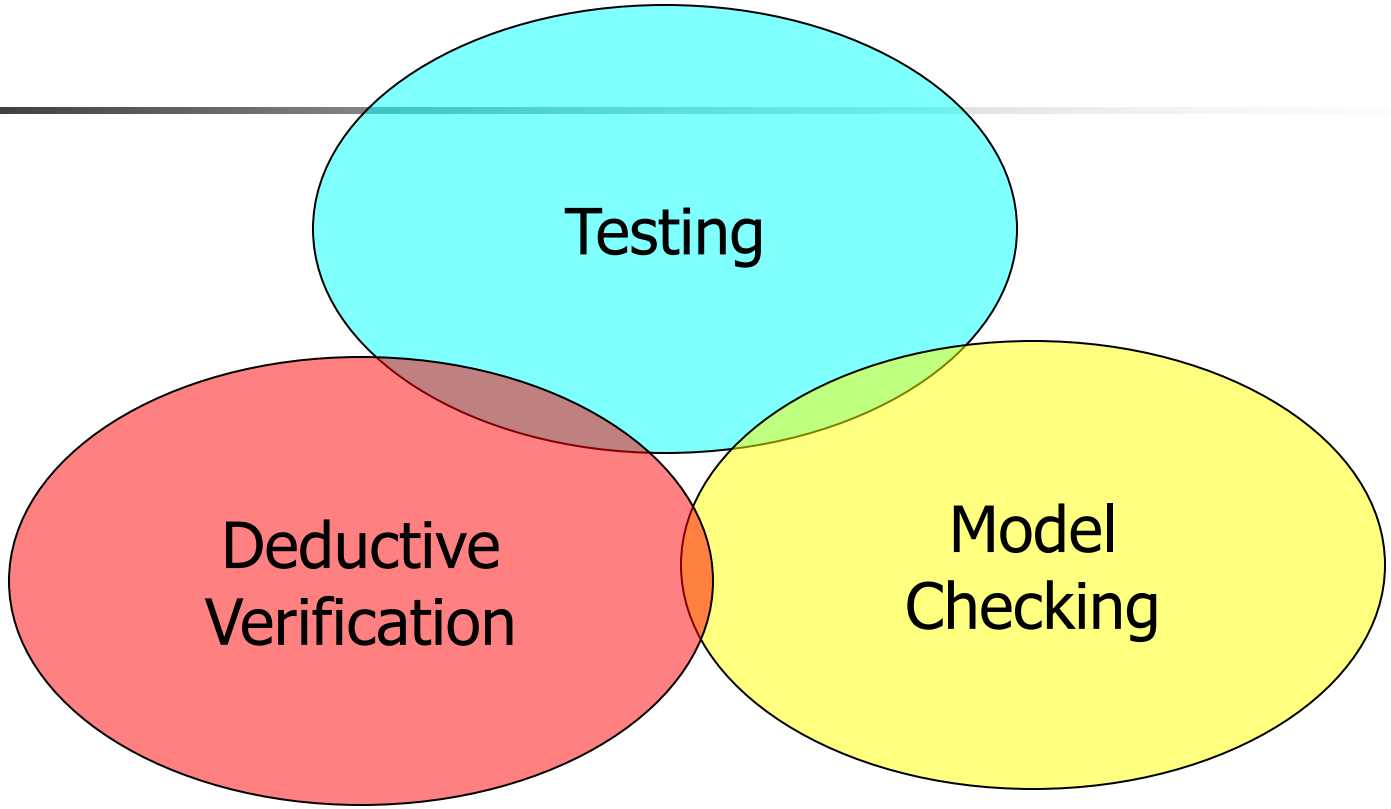
Method / Criterion	Testing	Deductive Verification	Model Checking
Size of system	Small-Very large	Limited examples	100s-1000s lines
Time	Minutes-Hours	Days-Weeks	Minutes-Hours
Expertise	Test engineers/ programmers	Mathematicians, Comp-Sci., Logic.	Comp.-Scientists/ sw engineers
Popularity	SW/HW industry	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

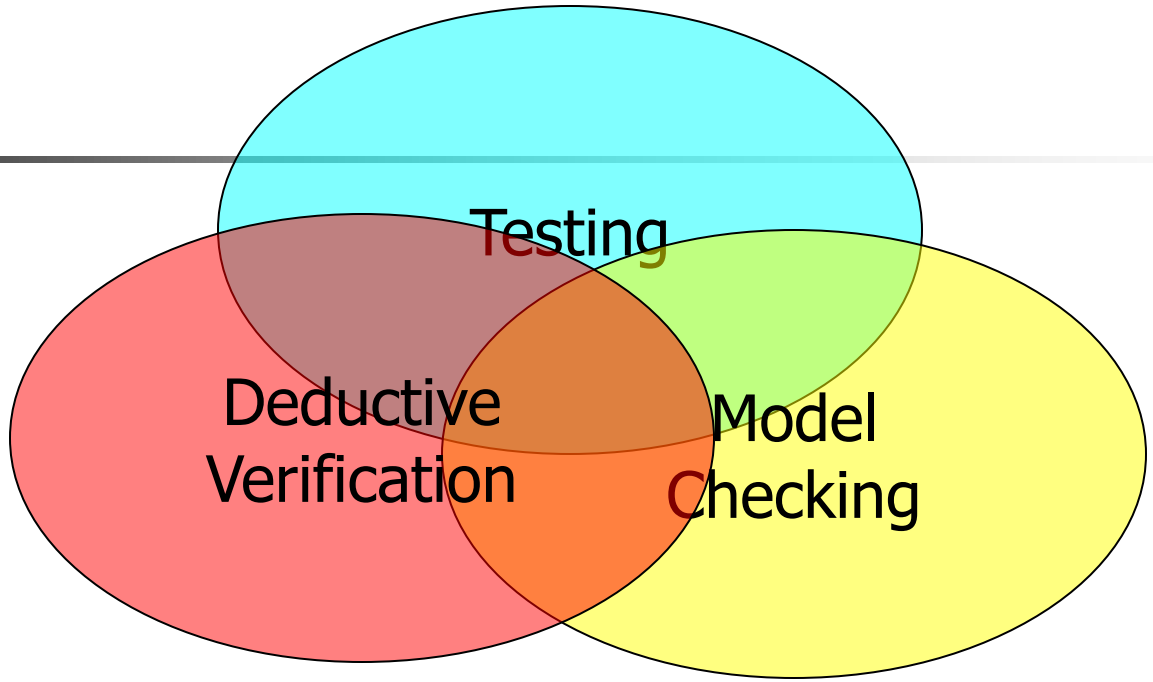
Verification of Abstraction

- General startegy
  - Do abstractions to reduce the system state space (e.g., to finite states, if possible).
  - Then verify correctness properties of that abstraction.



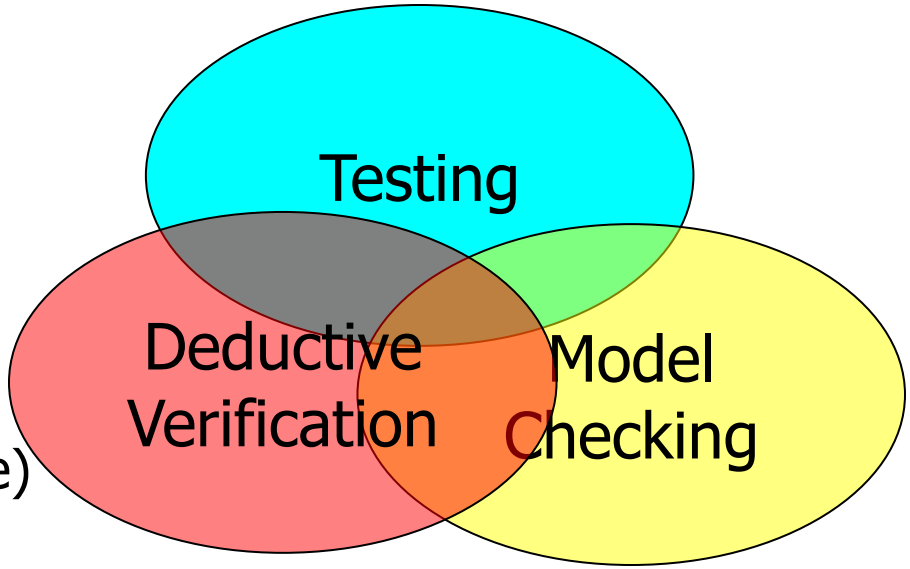

Testing
Deductive Verification
Model Checking

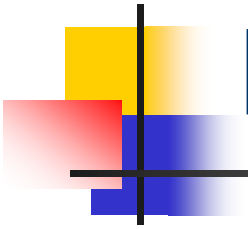

Testing
Deductive
Verification
Model
Checking


Testing
Deductive
Verification
Model
Checking

Symbolic Verification / Testing

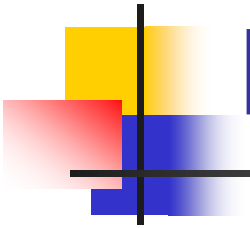
- Use symbolic verification to generate abstract test path conditions.
- Derive the explicit (executable) paths by model checker using abstract paths and temporal formulas describing test goals.





Foci of the course (refined)

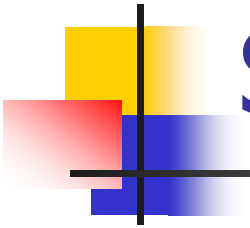
- Techniques: MC, deductive verification, refinement
- Tools (based on different theoretical backgrounds):
  - Theoretical background
    - Semantics / Algorithms / Datastructures
  - How does it work?
    - Arhitecture/ Capacity and restrictions
  - Tool in work: hands-on experience with Uppaal, ...
- Labs:
  - Read-write over unreliable channel
  - Self-stabilizing systems
  - Scheduling



Labs

- We will use model checker *UppAal* to check the properties of specifications.
- We use theorem proving assistant *Prover9* to prove formala of propositional 1-st order calculi and
- MB test generators

Server for experiments



- Tools that run under Linux will be available in server Dijkstra
- If you have X server (you run Linux, FreeBSD, MacOSX, ...) then just:
  - > ssh -X dijkstra.cs.ttu.ee

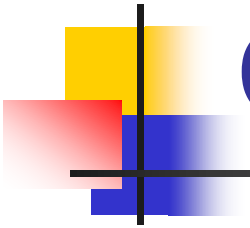
- Under Windows you need additional software, e.g. XWin32 (commercial), to run programs with GUI from dikstra.cs.ttu.ee. Use shell account by using e.g. Putty as the client.

Course organization I

- Lecture: *Prof. Jüri Vain*
  - Wed 14.00 – 15.30
  - Room ICT-A1
- Labs

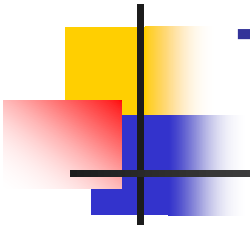
Instructors: *Evelin Halling*

  - Wed 16.00 – 17.30
  - Room ICT- 401
- Exercises: *Jüri Vain*
  - Cancelled since Feb. 2015



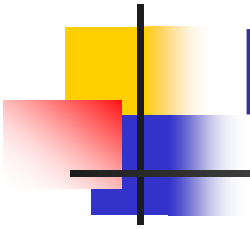
Course organization II

- 13 lectures, 8 labs
- 3 (small) lab projects
- 3 tests (> 50% means pass)
- Exam (written)



Topics to be covered

- Foundations: logics, models & specifications
- Algorithmic verification using model checking
- Deductive verification of (sequential and parallel) programs using Hoare logics
- Verification of RT- systems
- Verifying fault-tolerance
- Refinement based development
- Intro to model based testing



Home reading

- Formal methods homepage http://vl.fmnet.info/
- Formal Methods Europe: www.fmeurope.org
- Mike Gordon: Specification and Verification I. www.cl.cam.ac.uk/users/mjcg/Teaching/SpecVer1/SpecVer1.html
- Z- method http://www.cs.uiowa.edu/~fleck/181.html
- ASM http://www.di.unipi.it/~boerger/LC01.html
- SDL http://www.sdl-forum.org/sdl2000present/sld001.htm
- http://www.sdl-forum.org/sdl2000present/tsld001.htm
- Model checker: Uppaal: www.docs.uu.se and www.uppaal.com

- The reading list will be updated dynamically during the course