Verification of the Detailed Design

Istvan Majzik
majzik@mit.bme.hu

Budapest University of Technology and Economics
Dept. of Measurement and Information Systems

Overview

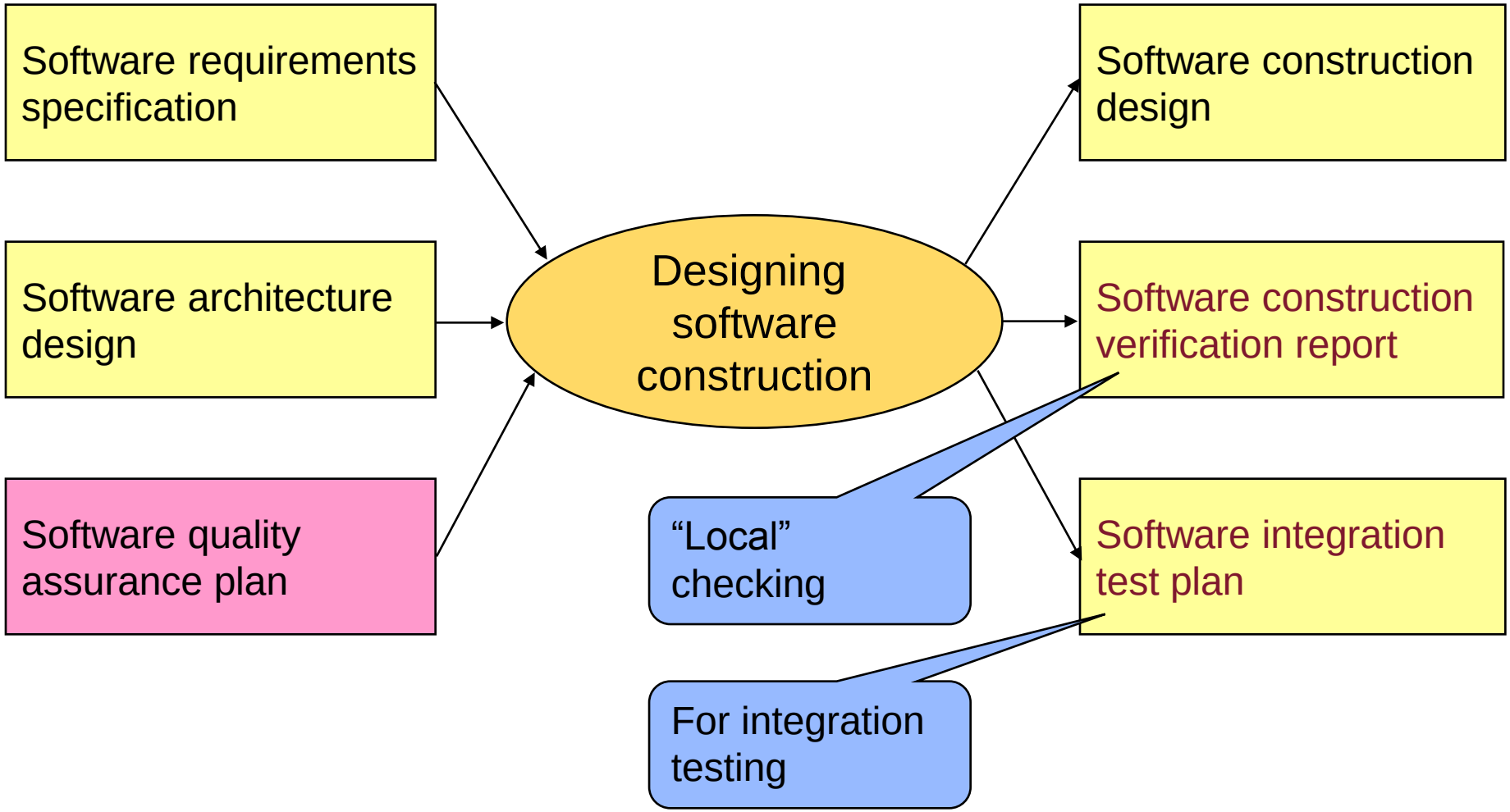
- Preparation of the detailed design
  - Software construction
  - Module (component) design
- Verification techniques
  - Verification criteria
  - Static and dynamic techniques
- Introduction to formal verification
  - Formal syntax and semantics
  - Categorization of techniques

Preparation of the detailed design

Software construction

Module (component) design

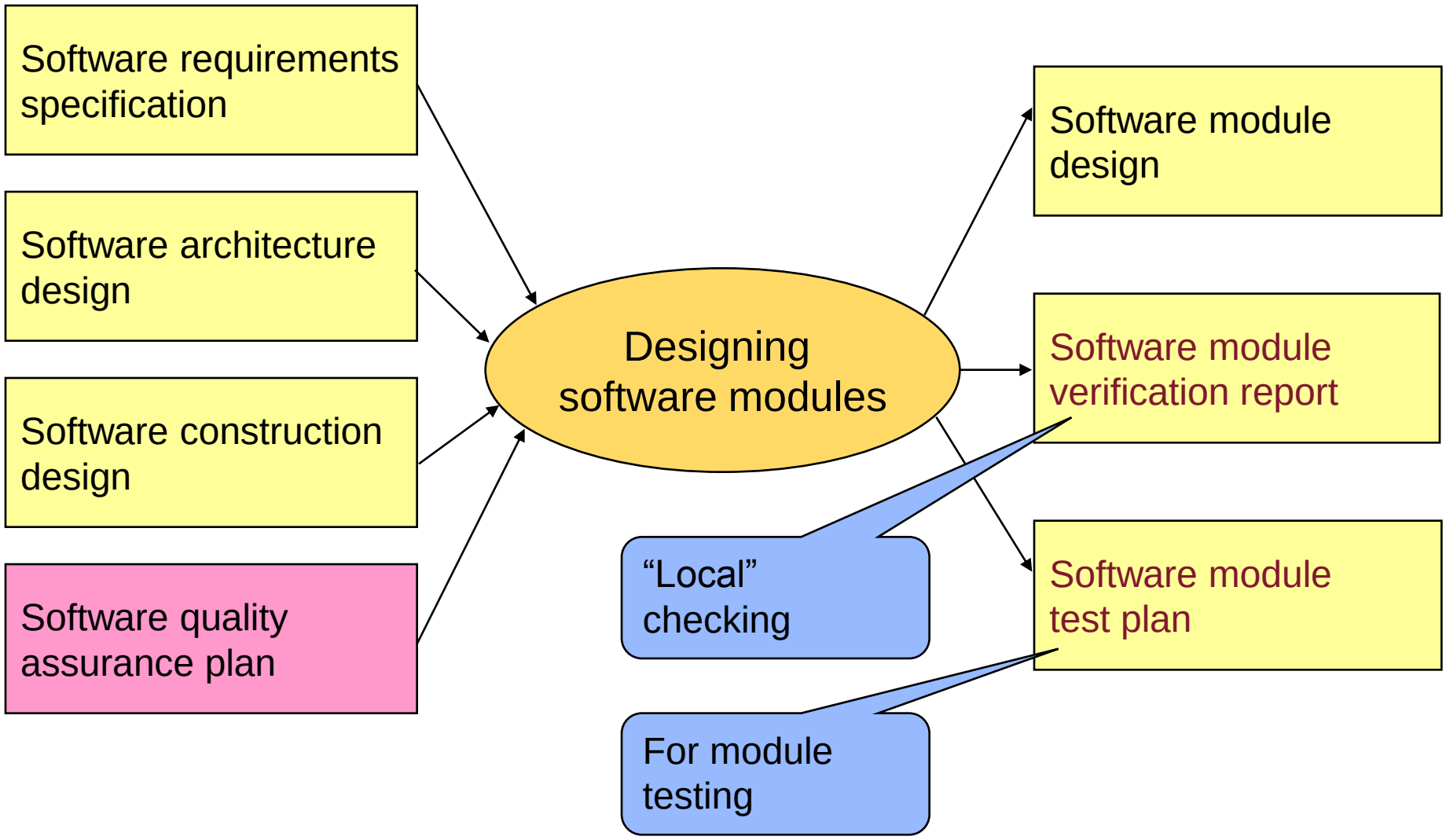
Software construction



Software construction

- To be designed:
  - System level algorithms for the interaction of modules
  - Global data structures
  - (Refinement of modules if needed)
- Design (description) language:
  - Capturing interactions and information exchange (ordering, timeliness)
  - Representing (abstract / concrete) data structures
  - Characterized by modularity, abstraction, precision
  - Formal, semi-formal, structured languages
- Specific characteristics (in critical systems):
  - Fully defined interfaces
  - Module and parameter size / complexity limits
  - Information hiding

Software module (component) design



Software module design

- Internal design of software modules
  - Algorithms
  - Data structures
- Design (description) language
  - Languages closer to implementation
    - Pseudo-codes can also be used
  - Formal, semi-formal, structured languages
    - Description of the behavior is important:
      control flow automata, state machines, statecharts

Verification techniques

Verification criteria

Static checking

Dynamic checking

Criteria for the verification of the detailed design

- Local characteristics of the design
  - Completeness, consistency, verifiability, feasibility
- Conformance (to the outputs of previous steps)
  - Behavioral properties specified earlier
    - Safety properties: “Something bad never happens”
    - Liveness properties: “Something good will eventually happen”
  - Conformance of abstract and refined behavior
    - Simulation, bisimulation, refinement relations
- Completeness of test plans

Static checking of the detailed design

- Review: Checklist, error guessing
- Structure based analysis
  - Control flow: complexity (e.g., McCabe metrics), structure (e.g., unreachable states), …
  - Data flow: initialization and use of variables, ordering of access, …
  - Boundary values: switching to different behavior
- Analysis of unwanted behavior
  - Potential unwanted influences through reserving resources (CPU, memory), …
- Symbolic execution
  - Checking inputs that cause parts of a program to execute

Dynamic checking of the detailed design

- Prototype implementation and animation
  - Detection of problematic cases requires particular care
- Simulation
  - Can we simulate all possible executions?
- Formal verification
  - For proving properties (“exhaustive” checking)
  - Formal methods
    - Formal languages: formal syntax and semantics
    - Formal techniques for verification

Introduction to formal verification

Formal syntax and semantics

Categorization of techniques

Formal verification

- Use of precise, mathematical techniques (esp. from discrete mathematics, mathematical logic) for verification
  - Formal language: Formal syntax and semantics
    - Design description (structure, behavior)
    - Property description (property specification)
  - Mathematical algorithm for verification
    - Checking design properties (e.g., ambiguity)
    - Checking changes (e.g., refinement)
    - Conformance of behavior and property descriptions
- Crucial aspect: Formalization of the real problem
  - Not automatized
  - Simplification, abstraction is needed (it has to be validated)

Formal syntax

- Mathematical description:

$$KS = (S, R, L) \text{ and AP, where}$$

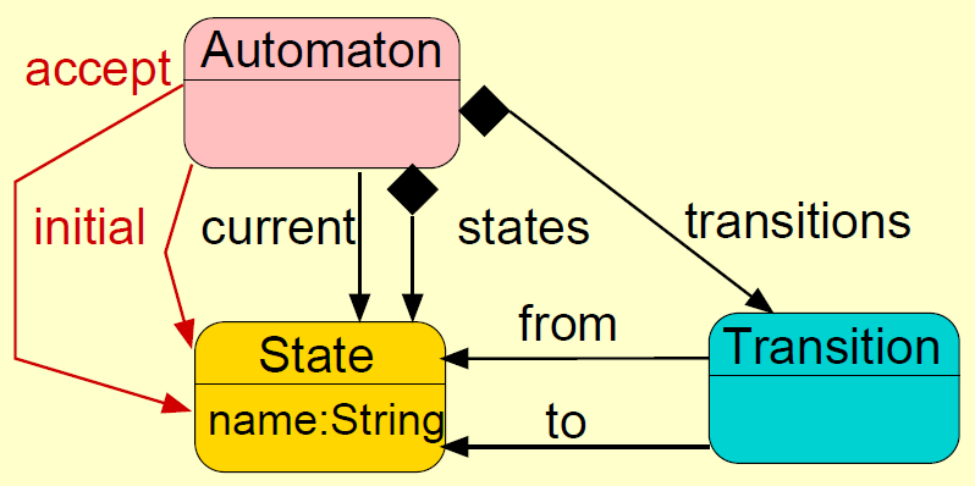
$$\text{AP} = \{\text{P,Q,R,...}\}$$

$$\text{S} = \{s_1, s_2, s_3, ...s_n\}$$

$$\text{R} \subseteq \text{S} \times \text{S}$$

$$\text{L: S} \rightarrow 2^{AP}$$

- BNF: BL ::= true | false | p∧q | p∨q
- Metamodel:
  - Abstract syntax: grammar rules
  - Concrete syntax: representation



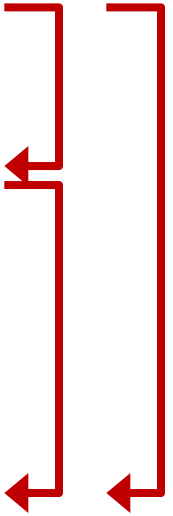
Formal semantics (overview)

The meaning of the model following the syntax:

- Operational semantics: "for programmers"
  - Defines what happens during operation (computation)
  - Builds on simple notions of execution: states, events, actions, …
  - E.g., to describe the state space for verification
- Axiomatic semantics: "for correctness proofs"
  - Predicate language + set of axioms + inference rules
  - E.g., for automated theorem prover tools
- Denotational semantics: "for compilers"
  - Mapping to a known domain, driven by the syntax
    - Known mathematical domain, e.g., computation sequence, control-flow graph, state set, … and their operations (union, concatenation, etc.)
    - Analysis of the model: analysis of the underlying domain
  - E.g., for synthesis tasks

Models for formal verification

- Design models (with operational semantics)
  - Engineering (design) models:
    - E.g., DSL, UML with (semi-)formal semantics
  - Higher-level formal models:
    - Control oriented: automata, Petri nets, …
    - Data processing oriented: dataflow networks, …
    - Communication oriented: process algebra, …
  - Basic mathematical models:
    - KS, KTS, LTS, finite state automata, Büchi automata
- Property descriptions
  - Higher level:
    - Time diagram, message sequence chart (MSC) variants
  - Lower level:
    - First order logic, temporal logic, reference automaton



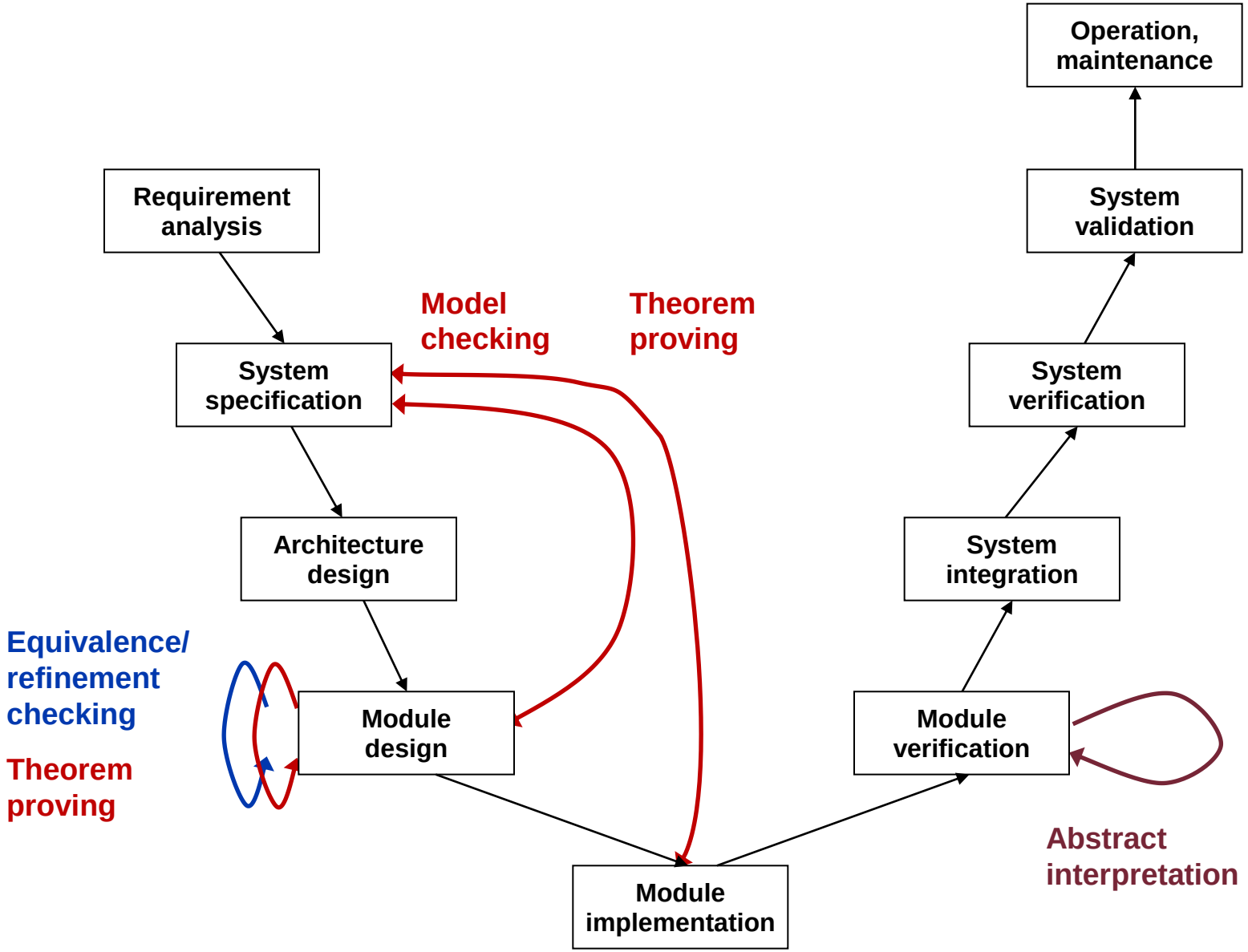
Typical formal verification techniques

Models / techniques	Behavior description (basic model)	Property description (basic property)
Model checking	Kripke structure (KS), Kripke transition system (KTS)	Temporal logics, first order logics
Equivalence / refinement checking	Labeled transition system (LTS), finite automata	LTS, automata (as reference behavior)
Theorem proving	Deduction system	Theorem to be proved (first order logic)
Static analysis (abstract interpretation)	Kripke transition system, Control Flow Automaton (extracted from the program)	Assertion (first order logic)

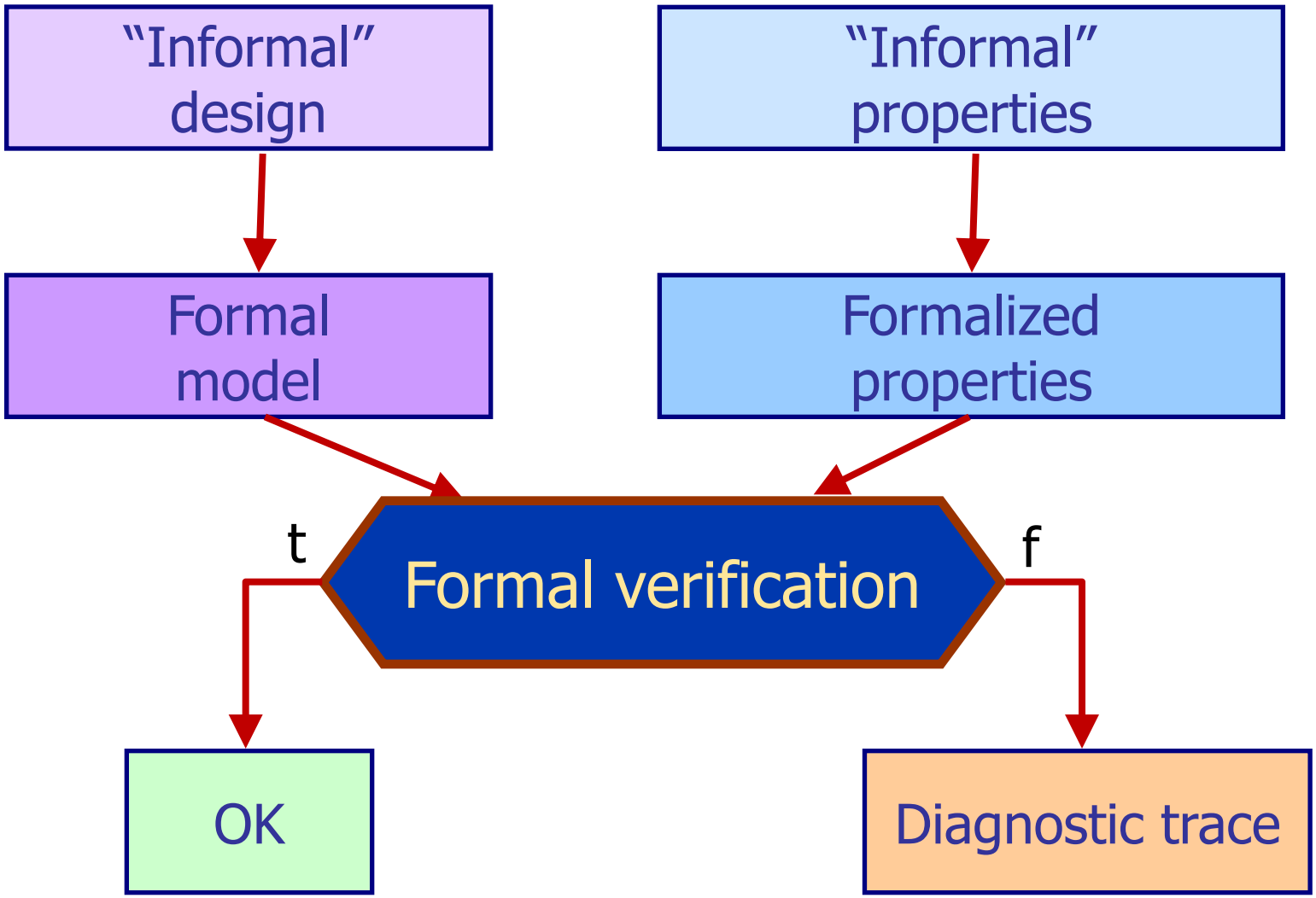
Advantages and limitations of the techniques

- Model checking, equivalence/refinement checking
  - ☺ Fully automated, exhaustive checking
  - ☺ Construction of diagnostic trace (for debugging)
  - ☹ State space exploration (handled partially)
- Theorem proving
  - ☺ Scalable for complex systems (e.g., by induction)
  - ☺ High expressive power
  - ☹ Interactive (need hints, e.g., to find a proof strategy)
  - ☹ There is no diagnostic trace (counter-example)
- Static analysis (abstract interpretation)
  - ☺ Handling state space explosion by abstraction
  - ☹ Abstraction is hard to automate

The role of formal verification techniques



Our goal


"Informal" design
"Informal" properties
Formal model
Formalized properties
t
Formal verification
f
OK
Diagnostic trace