The role of formal methods

dr. István Majzik
dr. Tamás Bartha
dr. András Pataricza

BME Department of Measurement and Information Systems

What are formal methods?

- Mathematical techniques,
  - Mainly discrete mathematics and
  - Mathematical logic

  to build and analyze hardware and software systems'
  - Specification,
  - Designs (models),
  - Implementation (behavior),
  - Documentation.

How can a computer engineer benefit from formal methods?

Typical example:
algorithm verification

An engineering task

- Multi-processor application
- Only one process can access a hardware resource at a time (mutual exclusion needed)
  - Example: Use of communication channel
  - "Critical section" to be protected in the program
  - The platform (OS, framework*) does not give support: no semaphore, no monitor, etc.
  - Only shared variables (reading or writing is atomic) can be used
- How to do it?
  - Classical solutions
  - Custom algorithm

Mutual exclusion algorithm (pseudocode)

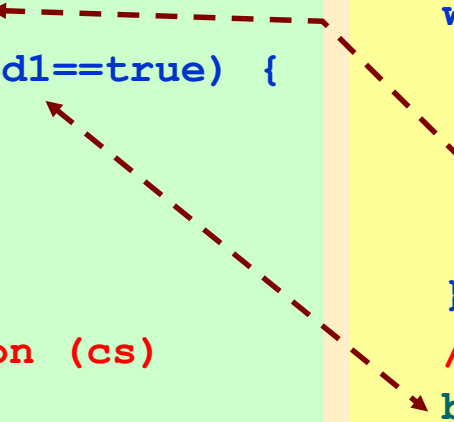
- 2 participants, 3 shared variables (H. Hyman, 1966)
  - **blocked0**: First participant (P0) wants to enter
  - **blocked1**: Second participant (P1) wants to enter
  - **turn**: Who is the next to enter? (P0 if turn=0, P1 if turn=1)

P0

```
while (true) {
   blocked0 = true;
   while (turn!=0) {
        while (blocked1==true) {
               skip;
        }
        turn=0;
   }
   // Critical section (cs)
   blocked0 = false;
   // Do other things
}
```

P1

```
while (true) {
   blocked1 = true;
   while (turn!=1) {
        while (blocked0==true) {
               skip;
        }
        turn=1;
   }
   // Critical section (cs)
   blocked1 = false;
   // Do other things
}
```



Is this algorithm correct?

When is this algorithm correct?

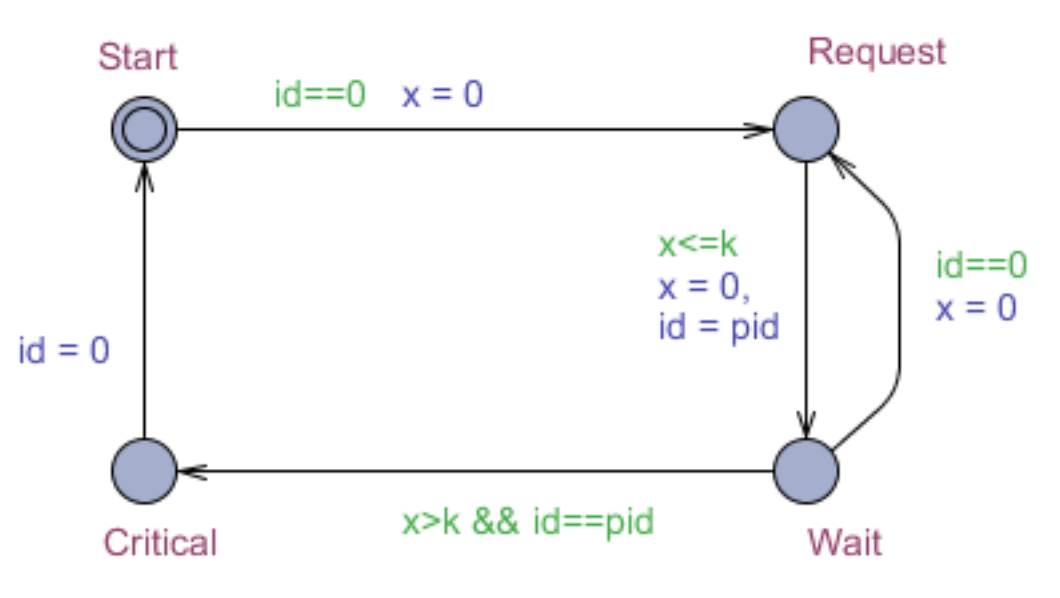
- Mutual exclusion is ensured:
  - Only one of the processes (P0 or P1) may be in the critical section
- The required behavior is possible:
  - P0 can enter the critical section
  - P1 can enter the critical section
- No starvation:
  - P0 will eventually enter the critical section
  - P1 will eventually enter the critical section
- Deadlock freedom:
  - There is no mutual waiting (blocking)

How can we check the requirements?

- By implementing and testing
  - Can we construct test cases covering all possible executions (i.e., possible overlapping instruction order)?
  - Dedicated checker is needed to analyze the critical cases
  - The faults are expensive to fix (found only after implementation)
- By modeling and simulation of the model
  - Can we simulate all possible executions?
  - Detection of problematic cases requires particular care
  - Removing the faults from the model is less expensive

Let's create a formal model!

- Automaton formalism:
  - States and state transitions
  - Variables, constants
  - Expressions evaluated on variables for the execution of transitions
  - Value assignment actions on execution of transitions
- Syntax (an example diagram):

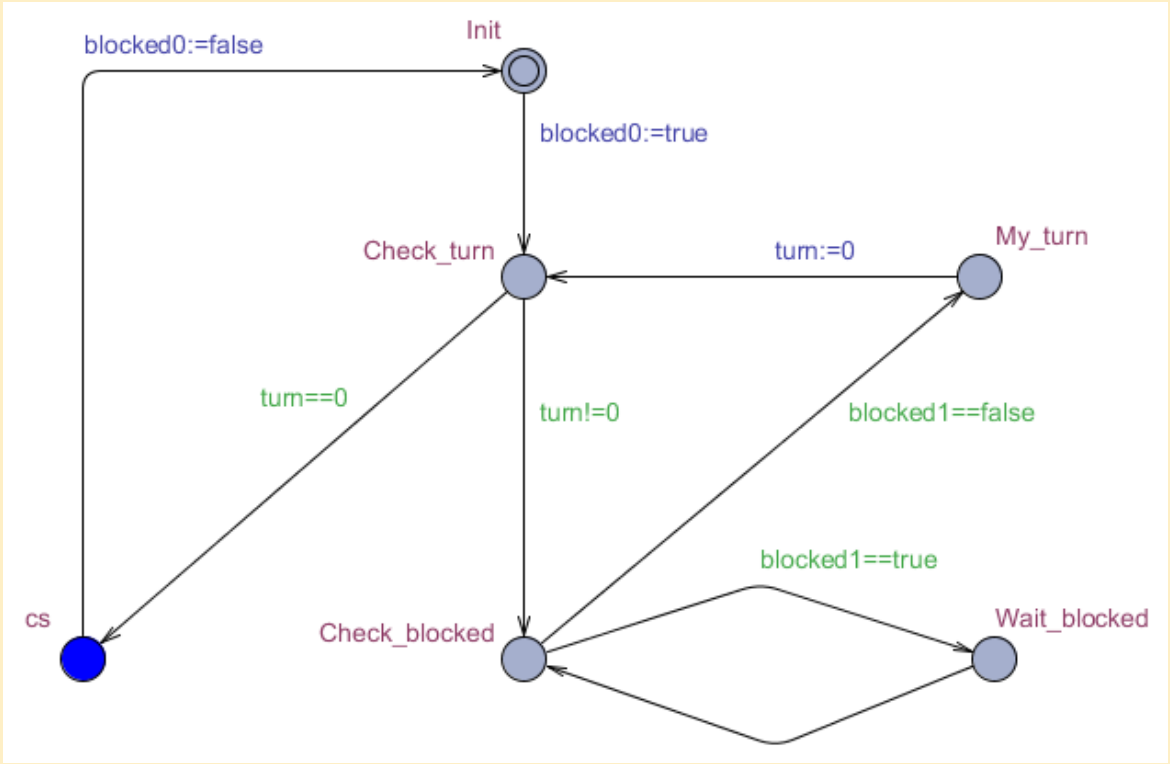


The formal model of the process P0

Declarations:

```
bool blocked0;
bool blocked1;
int[0,1] turn=0;
system P0, P1;
```

The automaton P0:



P0

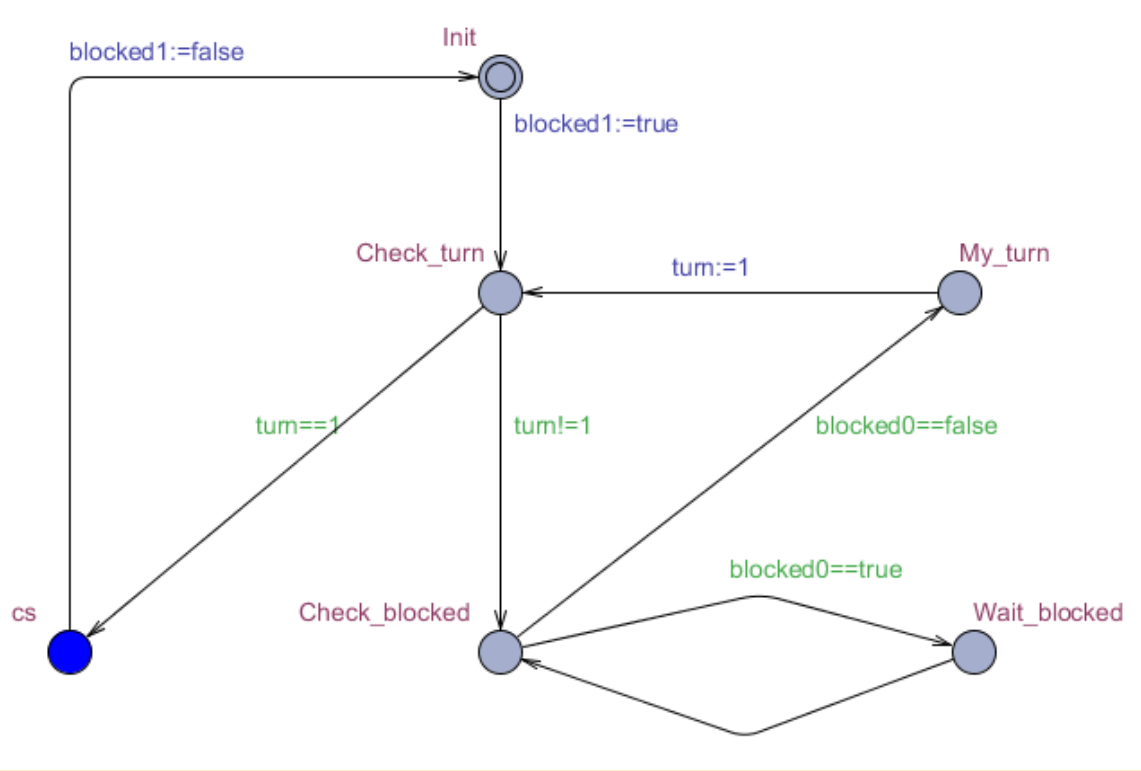
```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

The formal model of the process P1

Declarations:

```
bool blocked0;
bool blocked1;
int[0,1] turn=0;
system P0, P1;
```

The automaton P1:



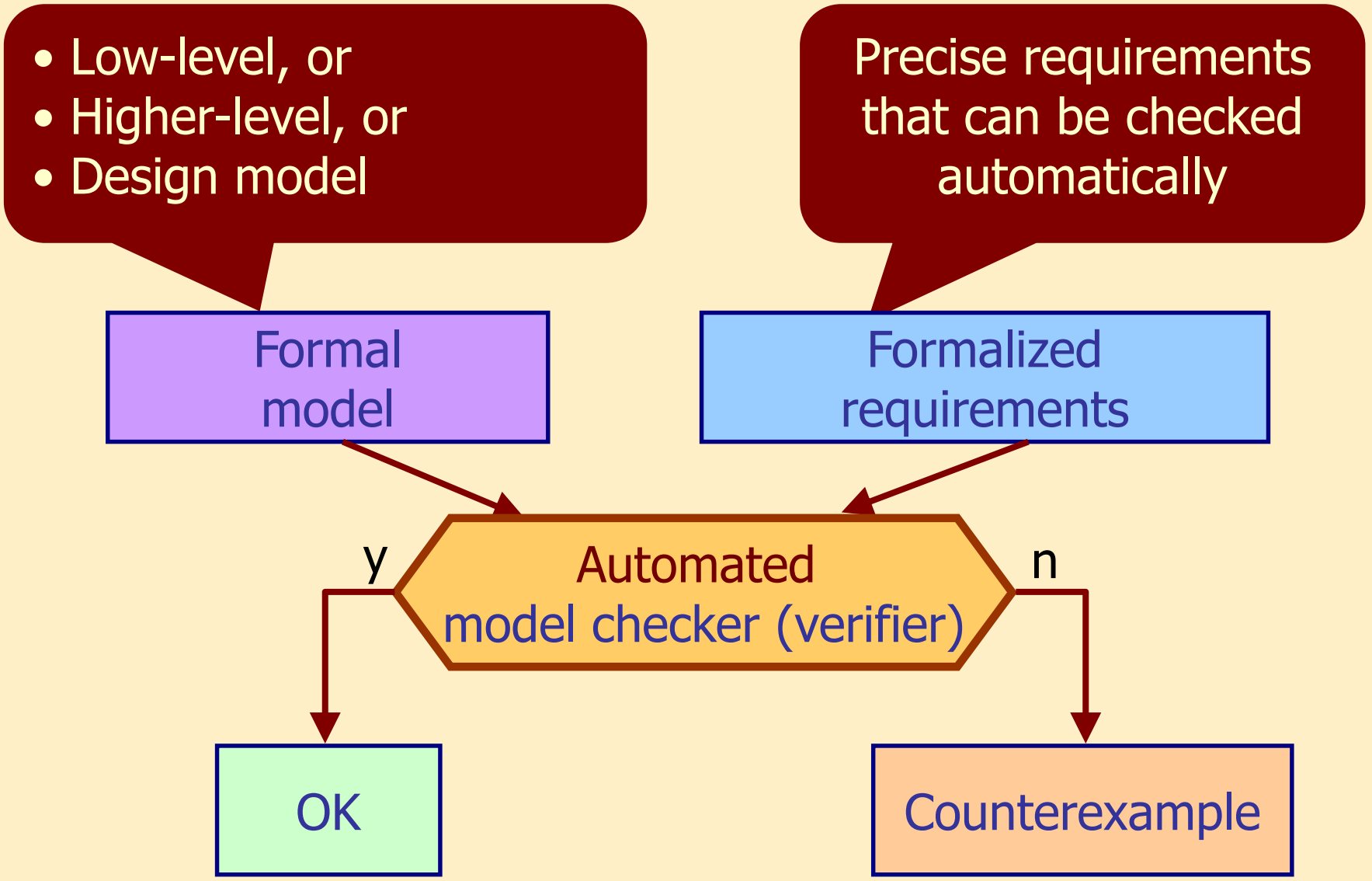
P1

```
while (true) {
    blocked1 = true;
    while (turn!=1) {
        while (blocked0==true) {
            skip;
        }
        turn=1;
    }
    // Critical section
    blocked1 = false;
    // Do other things
}
```

How can we check the requirements?

- By implementing and testing
  - Can we construct test cases covering all possible executions (i.e., possible overlapping instruction order)?
  - Dedicated checker is needed to analyze the critical cases
  - The faults are expensive to fix (found only after implementation)
- By modeling and simulation of the model
  - Can we simulate all possible executions?
  - Detection of problematic cases requires particular care
  - Removing the faults from the model is less expensive
- By modeling and checking the complete state space
  - Checks each possible execution: push-button state space exploration of the model with systematic algorithm
  - Automated checking of requirements: description language and efficient verification algorithm is needed
  - Faulty model: counterexample is given

Typical formal verification



The role of formal methods (overview)

What do we aim to achieve?

- Formal analysis of real problems:
  1. Problem formalization, creation of formal model ← assumptions, abstraction
  2. Analysis of formal model ← automated tools too
  3. Interpretation, usage of the results ← applicability

- Prerequisites
  1. Formal language to describe the problem
  2. Method to analyze the formal model
  3. Validity of results (cf. assumption, tools)

Step 1: Formal modeling language

- Goal of formalization: mathematically precise description
  - Designs: models, engineer's decisions (modeling language)
  - Requirements: expected properties (requirement description lang.)
- Structure of formal languages
  - Formal syntax
    - Notation: language elements and their connections
  - Formal semantics
    - Interpretation of the notation: what is the meaning?
- What do we want to describe with formal languages?
  - Functionality: behavior, conditions, …
  - Structure: components, interfaces, …
  - Extra-functional aspects: performance, reliability, …
- Advantages of using a formal language
  - Unambiguity, verifiability
  - Possibility for automated processing

Formal syntax (overview)

- Mathematical description:

$KS = (S, R, L)$ and AP, where

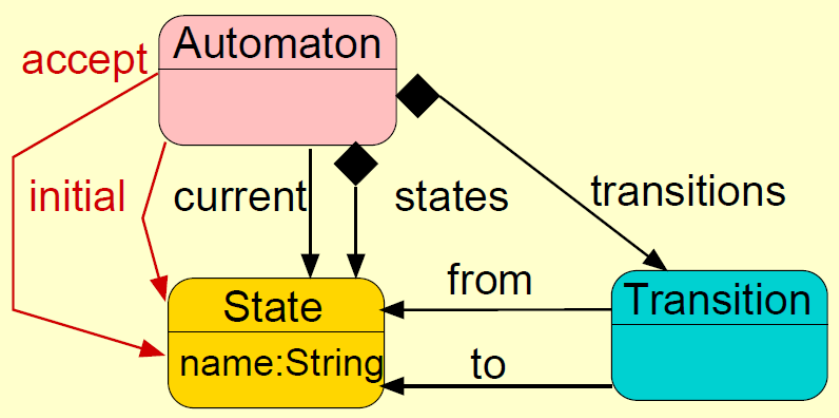
$\mathrm{AP} = \{P, Q, R, \ldots\}$

$S = \{s_1, s_2, s_3, \ldots s_n\}$

$R \subseteq S \times S$

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

- 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 execution (operation)
  - Builds on simple items: e.g., 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 systems
- 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 (concatenation, union, etc.)
    - Analysis of the model: analysis of the underlying mathematical domain
  - E.g., for code generation

Step 2: Use of formal model

Part of formal methods:

A (mathematical) method that gives information about the formal model

- Execution of the formal model
  - Simulation
- Verification of the formal model: formal verification
  - Verification "on its own"
    - Consistency (free of contradictory statements)
    - Completeness, closure
  - Verification of "compliance"
    - Between models and required properties (design ↔ specification)
    - Between models (original design ↔ modified design)
- Synthesis based on the formal model:
  - Generation of software (source code, configuration)
  - Generation of hardware implementation

What is the target of formal methods?

What do we try to use FM for?

What are the challenges?

How can formal methods help?

An enlightening story…

- Vasa Swedish warship, 1628: Sank right after launching

- Problems:
  - Changing requirements (King Gustav II Adolf)
  - Missing precise specification (Shipbuilder Henrik Hybertsson)
  - Uncontrolled planning (Assistant Johan Isbrandsson)
  - Ignored warnings (Admiral Fleming)

- More information:
  - Linda Rising. The Vasa: A Disaster Story with Software Analogies. The Software Practitioner, January-February 2001.
  - Richard E. Fairley and Mary Jane Willshire. Why the Vasa Sank: 10 Problems and Some Antidotes for Software Projects. IEEE Software, March-April 2003.

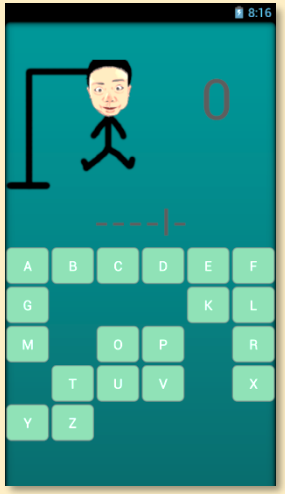
Design of complex systems

- Minimal design
- Implicit process
- Simple tools

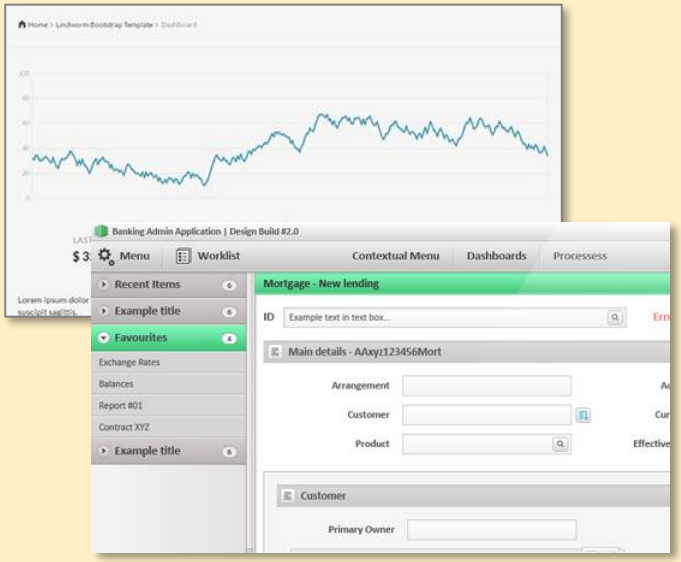
- Thorough design
- Defined process
- Efficient tools

- Verified design
- Defined process
- Automated tools

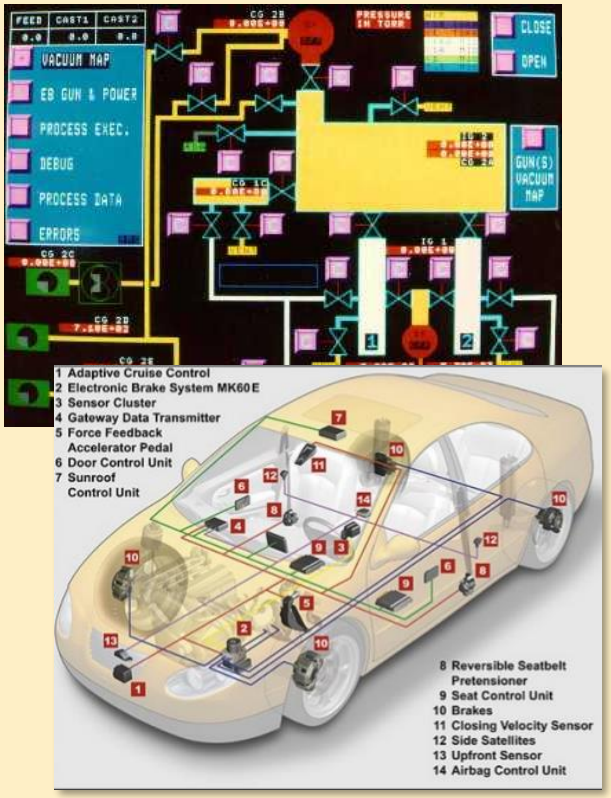
Design of complex systems



- Minimal design
- Implicit process
- Simple tools



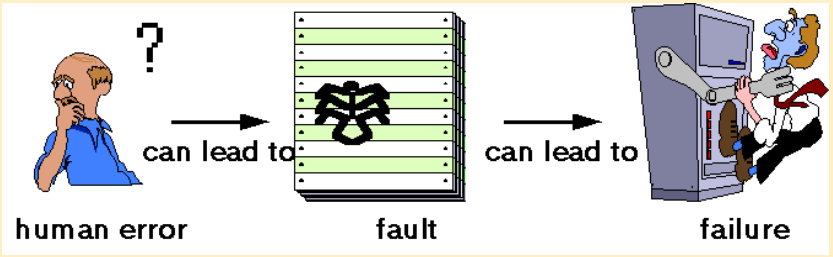
- Thorough design
- Defined process
- Efficient tools



- Verified design
- Defined process
- Automated tools

Software quality

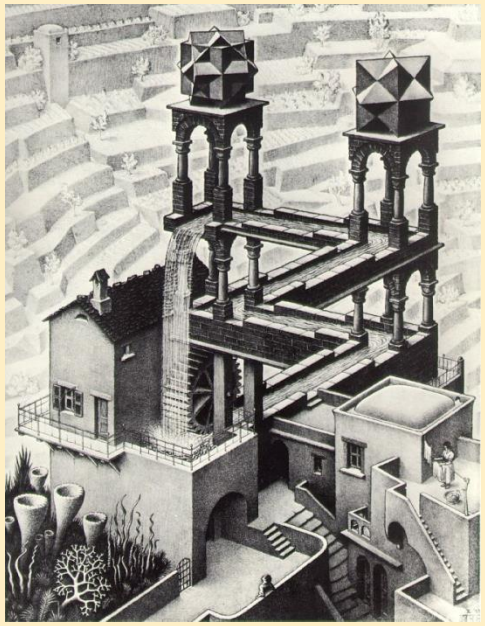
- Typical code size:
  - 10 kLOC … 1000 kLOC
- Development effort:
  - Large software: 0.1–0.5 man-year / kLOC
  - Critical software: 5–10 man-year / kLOC
- Fault removal (verification, testing, correction):
  - 45–75% effort
- Variation of fault density:
  - 10–200 faults / kLOC introduced during development



Checking, debugging, correcting

  - 0.01–10 faults / kLOC in the production version

Challenges of IT application development



- High-quality specification and design
  - Complete
  - Consistent, unambiguous
  - Verifiable
- Verification of design
  - Verification of design decisions
    - Design that is proven correct for the next steps
  - Avoidance or early detection of faults
    - Optimization of: Quality ↔ Cost ↔ Development time
- Use of tools with proven correctness
  - Synthesis of source code, configuration, test and monitor

Formal methods may be the basis.

Analyses: Verification and validation

Verification	Validation
"Are we building the system right?"	"Are we building the right system?"
Checks the conformance in and between the development phases	Checks the result of the development
Checks the correspondence between designs (models) and their specification over the design phases	Compares the final system and the user requirements
Objective process; can be formalised and automated	Some requirements may be subjective; acceptance checking
Sensitive to design and implementation faults	Sensitive to the problems of requirements too (e.g., missing req.)
Not needed, if there is an automated mapping between the requirement and the implementation	Not needed, if the specification is perfect (simple enough)

Classic method: Cleanroom Software Engineering

- Origin:
  - IBM proposal (1980s),
  - US military developments (1990s)
- Goal:
  - Fault avoidance instead of removal
  - Verification based on formal models
- Principles:
  - Use and verification of formal models
  - Incremental development with quality control (step-by-step increase of complexity)
  - Statistical testing based on formal models
    - Selecting the representative trajectories
    - Manual validation of modeling

Development of safety-critical software

- IEC 61508 standard: development guidelines
  - Functional safety in electrical / electronic / programmable electronic safety-related systems
  - Base of domain-specific standards
- Guidelines of requirement-specifications:

Table A.1 – Software safety requirements specification (see 7.2)

	Technique/Measure*	Ref.	SIL1	SIL2	SIL3	SIL4
1	Computer-aided specification tools	B.2.4	R	R	HR	HR
2a	Semi-formal methods	Table B.7	R	R	HR	HR
2b	Formal methods including for example, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z	C.2.4	---	R	R	HR

NOTE 1 – The software safety requirements specification will always require a description of the problem in natural language and any necessary mathematical notation that reflects the application.

NOTE 2 – The table reflects additional requirements for specifying the software safety requirements clearly and precisely.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. Only one of the alternate or equivalent techniques/measures has to be satisfied.

Development of safety-critical software

- IEC 61508: Guidelines of software design and development

Table A.2 – Software design and development: software architecture design (see 7.4.3)

	Technique/Measure*	Ref	SIL1	SIL2	SIL3	SIL4
1	Fault detection and diagnosis	C.3.1	---	R	HR	HR
2	Error detecting and correcting codes	C.3.2	R	R	R	HR
3a	Failure assertion programming	C.3.3	R	R	R	HR
3b	Safety bag techniques	C.3.4	---	R	R	R
3c	Diverse programming	C.3.5	R	R	R	HR
3d	Recovery block	C.3.6	R	R	R	R
3e	Backward recovery	C.3.7	R	R	R	R
3f	Forward recovery	C.3.8	R	R	R	R
3g	Re-try fault recovery mechanisms	C.3.9	R	R	R	HR
3h	Memorising executed cases	C.3.10	---	R	R	HR
4	Graceful degradation	C.3.11	R	R	HR	HR
5	Artificial intelligence - fault correction	C.3.12	---	NR	NR	NR
6	Dynamic reconfiguration	C.3.13	---	NR	NR	NR
7a	Structured methods including for example, JSD, MASCOT, SADT and Yourdon.	C.2.1	HR	HR	HR	HR
7b	Semi-formal methods	Table B.7	R	R	HR	HR
7c	Formal methods including for example, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z	C.2.4	---	R	R	HR
8	Computer-aided specification tools	B.2.4	R	R	HR	HR

NOTE – The measures in this table concerning fault tolerance (control of failures) should be considered with the requirements for architecture and control of failures for the hardware of the programmable electronics in IEC 61508-2.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. Only one of the alternate or equivalent techniques/measures has to be satisfied.

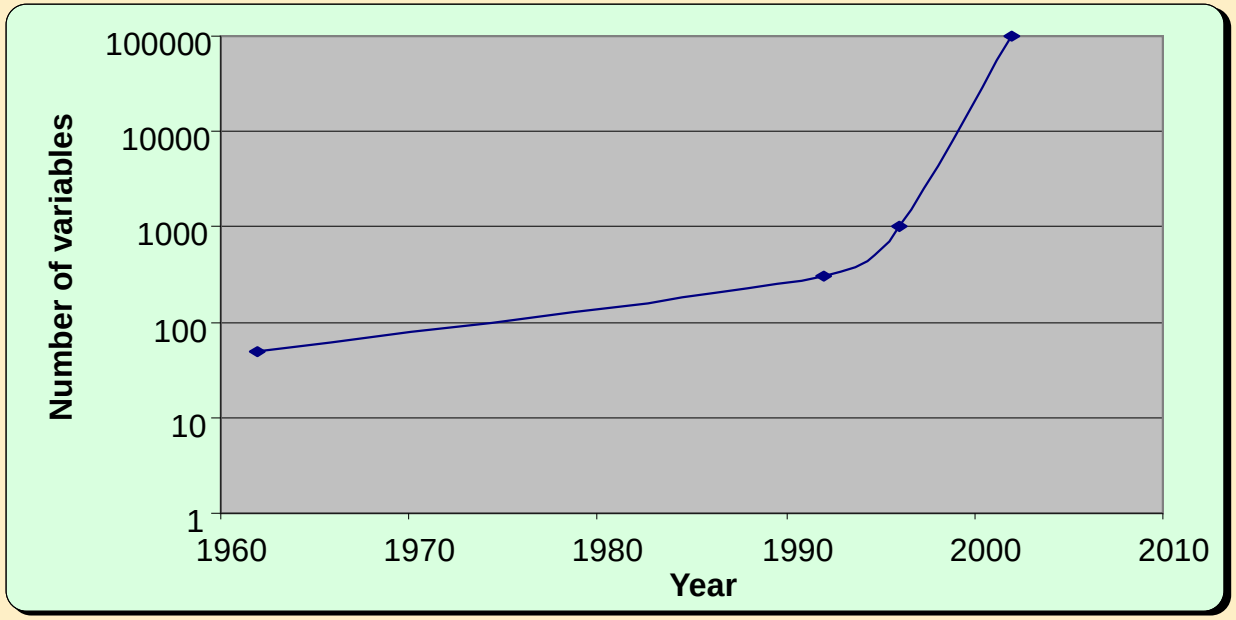
Evaluation of formal models

Challenges of formal analysis

- Realistic modeling
  - Lack of knowledge, assumptions (e.g., about environment)
  - But: this problem is independent from use of formal methods
- Needs special knowledge from the user
  - Mathematical models and notations
  - But: design modeling languages may hide them
- Complex verification and synthesis methods
  - We need to know the limitations of algorithms and techniques
  - Manual intervention may be needed (e.g., theorem prover systems)
  - But: "push-button" tools are more and more available
- Only applicable to the "small scale" problems
  - May big models/state spaces be handled using the available resources?
  - But: the performance of tools is steadily increasing

Evolution of formal verification

- Example: SAT tools (Boolean satisfiability)



- Capabilities of model checker tools:
  - $10^{20} \approx 2^{66}$ states initially (ROBDD, 1990)
  - $10^{100} \approx 2^{328}$ states may be feasible (concrete example)
  - 10^{62900} states have been analyzed too :-O

Current situation

- Constraints
  - Discrete state
  - Discrete time
  - Discrete event

  } systems
- Challenges, research directions
  - Efficiency of mathematical algorithms
  - Limitations of model classes (e.g., timing)
  - Difficulty of model construction (e.g., abstraction)
  - Expressivity of languages (e.g., requirements)
  - Precise definition of design languages (e.g., UML)

Successful approach

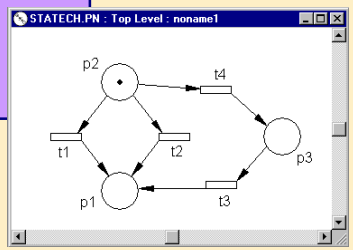
System design

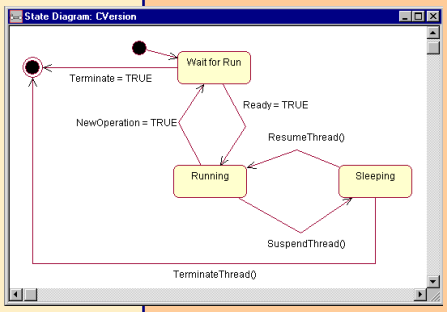
Formal verification

Design model (e.g., DSL)

Automated model generation

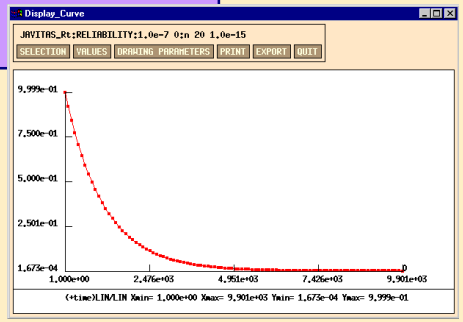
Formal model





Back-annotation of results

Analysis



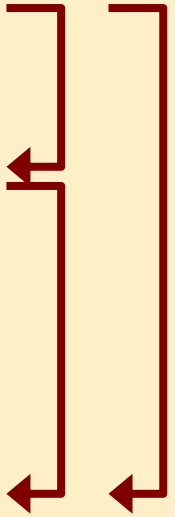
Implementation

Implementation

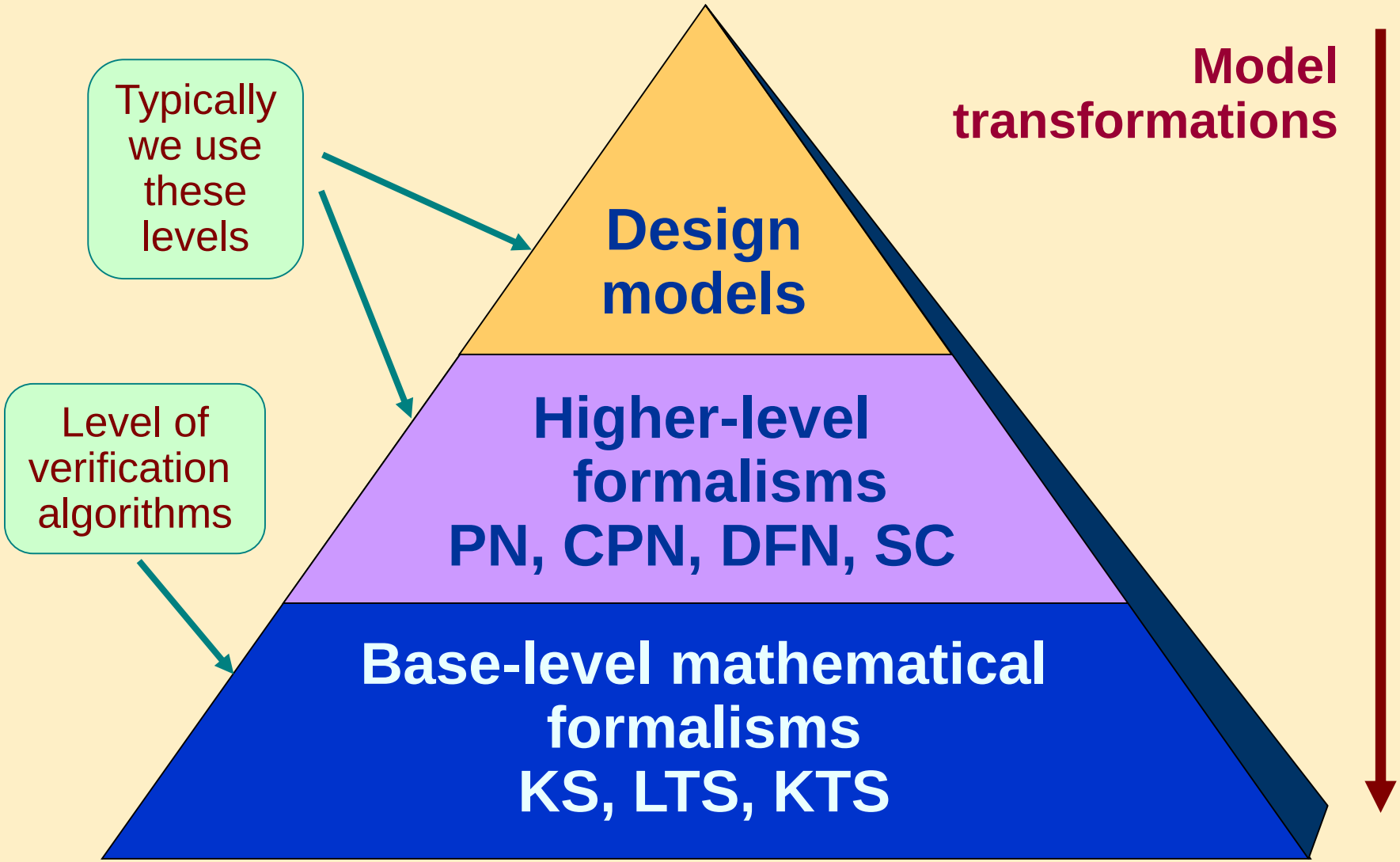
```
int main() {
  while (i<z){
  a=x+x*b[i++];
  }
 }
```

Models for formal verification

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



The structure of the course



Success stories

Classical applications

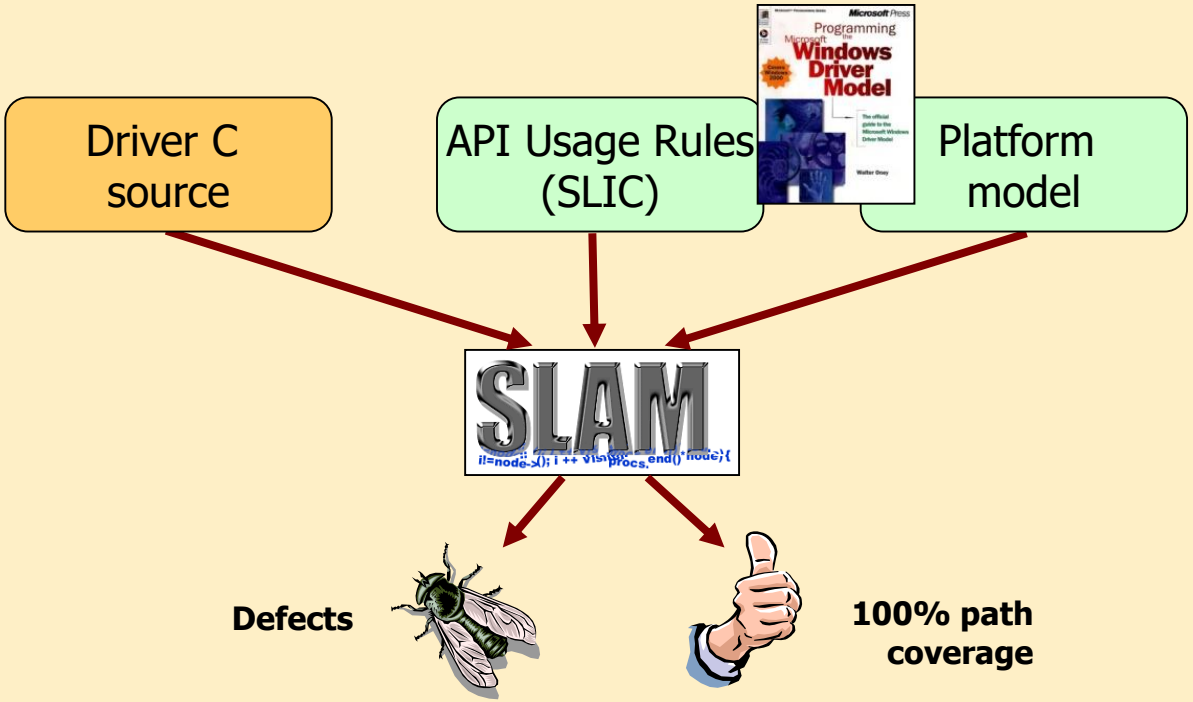
- USA TCAS-II traffic control system (collision avoidance for airplanes)
  - Specified in RSML language; analysis of completeness and consistency
- Philips Audio Protocol
  - 1994: manual verification, then 1996: automated verification (HyTech)
- Lockheed C130J flight software
  - Software development with correctness proof (CORE language + Ada)
  - No total cost increase due to the reduced testing costs
- IEEE Futurebus+ standard
  - Carnegie Mellon SMV: revealed a problem of the cache coherence protocol
- Hardware projects: ACL2 automated theorem prover
  - Motorola DSP Complex Arithmetic Processor core (250 registers): verification of DSP algorithms
  - AMD 5K86 processor: verification of floating-point division algorithm
- Intel Core i7 processor
  - "*For the recent Intel Core™ i7 design we used formal verification as the primary validation vehicle for the core execution cluster*"
  - Symbolic simulation for the complete analysis of data paths (2700 microinstruction, 20 manyears of work) – application of Binary Decision Diagrams
- Tools related to model-based software development
  - IBM (Telelogic), Ansys (Esterel), Prover, Mentor, Verum, Conformiq, …

Formal verification of source code

- Java
  - Bandera, PathFinder: model abstraction
  - Formalization of Java VM: Abstract State Machine
- Ada
  - SPARK Ada verification condition generator: for theorem prover
- C
  - BLAST: software model checker for C programs (abstraction)
  - CBMC: C-based bounded model checker
- C#, Visual Basic .NET
  - Zing (for MS Visual Studio): analysis of concurrent OO software
- Spec# (C# superset)
  - MS Research Boogie 2: Language extensions for specification
  - Checking correctness criteria: with program abstraction and theorem prover (Z3)
- Microsoft Windows Driver Kit (WDK)
  - Static Driver Verifier Research Platform, SLAM 2 tool
  - Static analysis of the usage conditions of Windows API

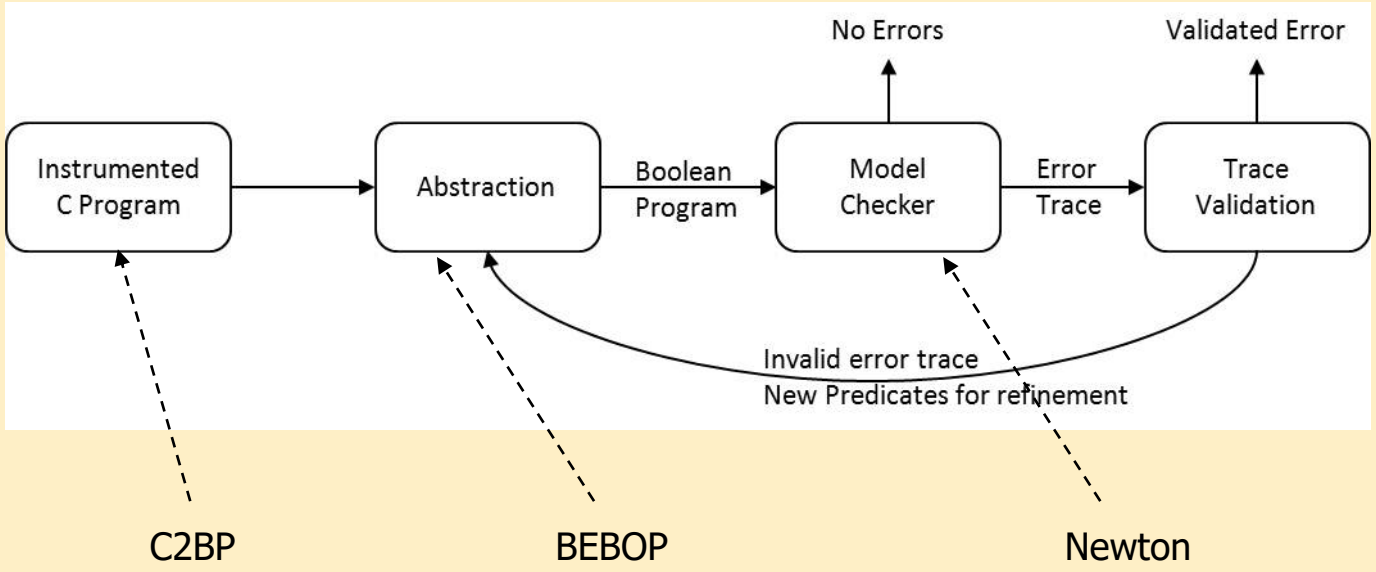
Example: SLAM

- Motivation: faulty drivers cause problems to the OS
  - E.g., incorrect locks on resources (missing lock, missing unlock, …)
- Solution:
  - Correct usage can be described with rules (e.g., state machine of locking)
  - SLAM checks the satisfaction of these rules on the source code
  - Uses source code abstraction: checking the states of Boolean programs
  - Faulty traces (not matching the rules) have to be cross-checked



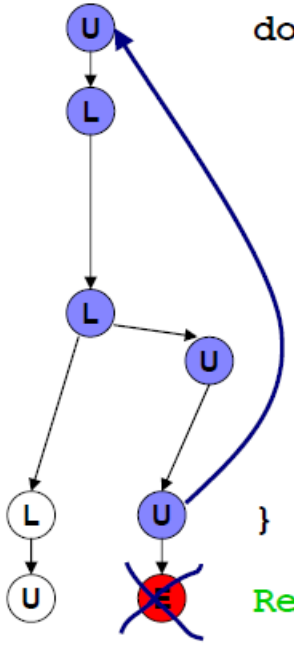
Example: SLAM

- Motivation: faulty drivers cause problems to the OS
  - E.g., incorrect locks on resources (missing lock, missing unlock, …)
- Solution:
  - Correct usage can be described with rules (e.g., state machine of locking)
  - SLAM checks the satisfaction of these rules on the source code
  - Uses source code abstraction: checking the states of Boolean programs
  - Faulty traces (not matching the rules) have to be cross-checked



Example: SLAM

- Motivation: faulty drivers cause problems to the OS
  - E.g., incorrect locks on resources (missing lock, missing unlock, …)
- Solution:
  - Correct usage can be described with rules (e.g., state machine of locking)
  - SLAM checks the satisfaction of these rules on the source code
  - Uses source code abstraction: checking the states of Boolean programs
  - Faulty traces (not matching the rules) have to be cross-checked




```
do {

   AcquireLock(&devices>writeListLock);

   deviceNoOld = deviceNo;
   more = devices->Next;

   if (more){
        ReleaseLock(&devices->writeListLock);
        ...
        deviceNo++;
   }
} while (deviceNoOld != deviceNo);

ReleaseLock(&devices->writeListLock);
```

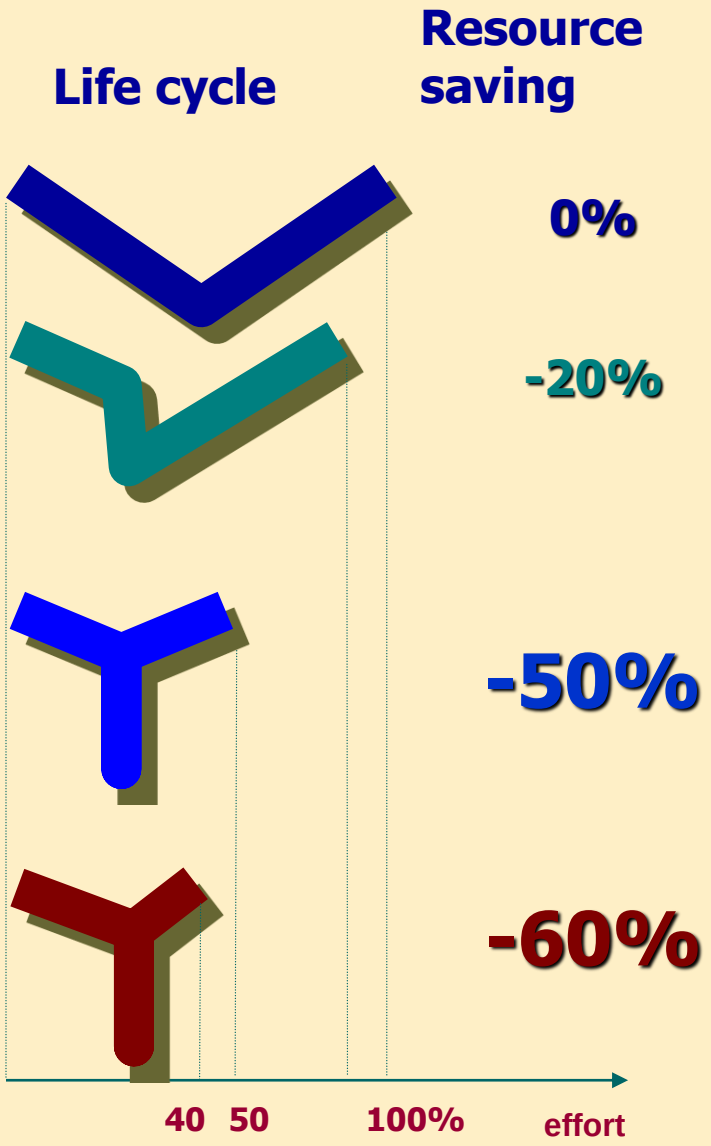
From the V to the Y development model

Manual coding

Use of "regular" automated code generator

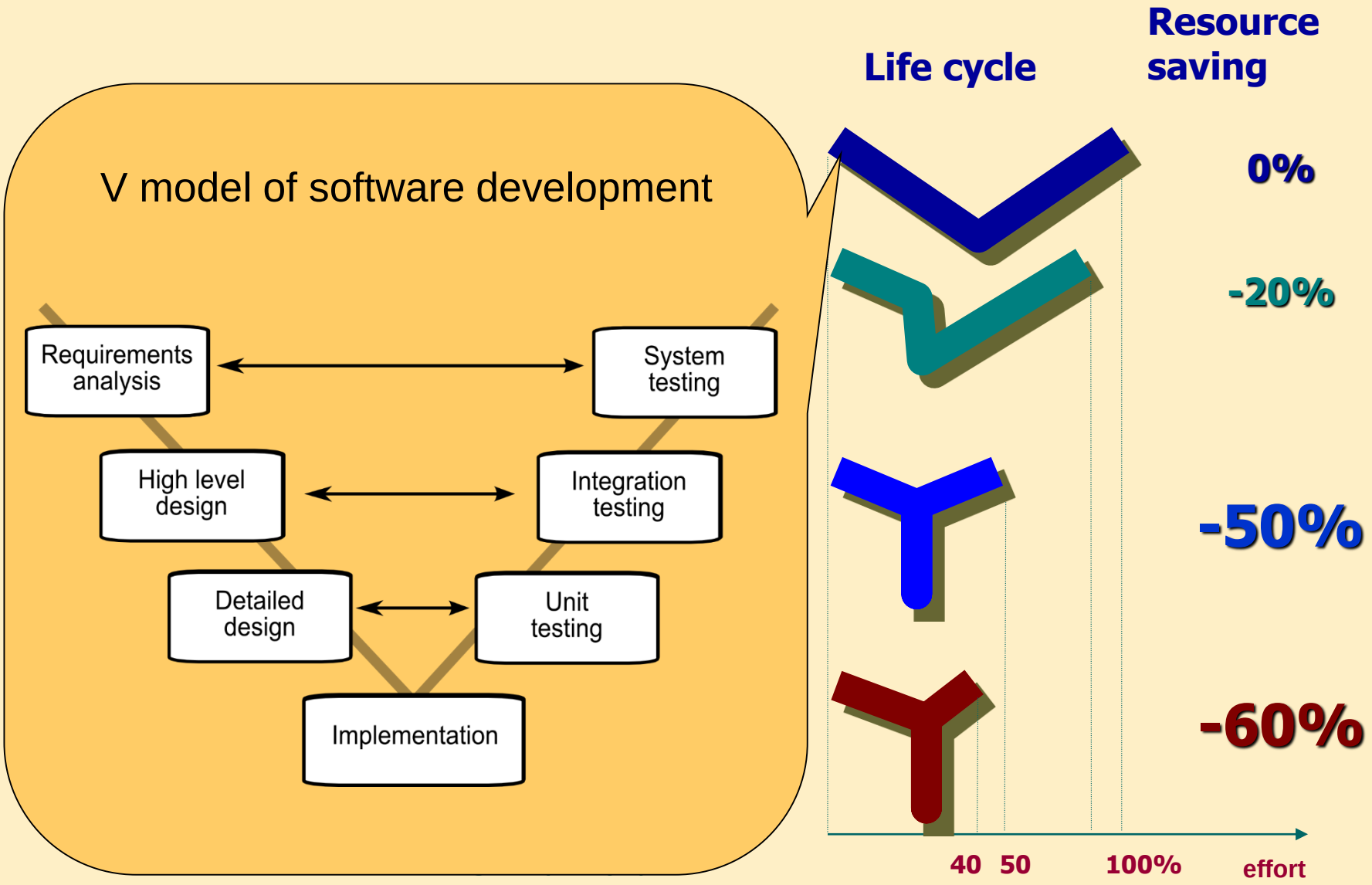
Use of certified automated code generator

Design with formal verification



* Source: Esterel Technologies (Ansys)

From the V to the Y development model



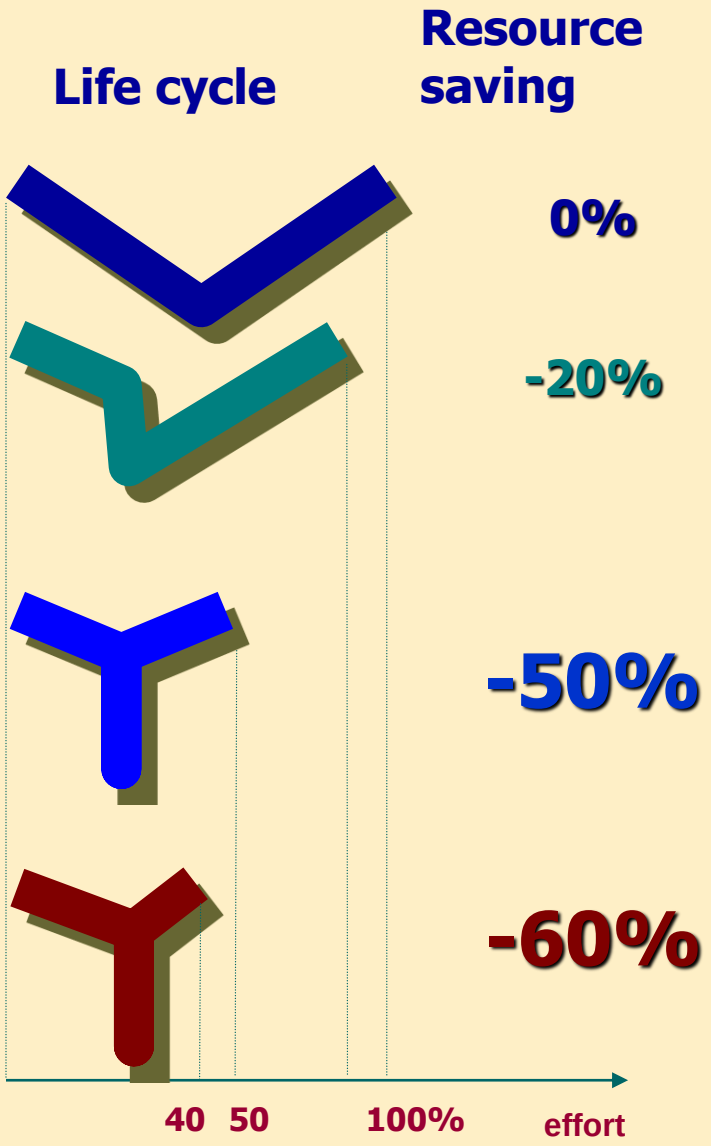
From the V to the Y development model

Manual coding

Use of "regular" automated code generator

Use of certified automated code generator

Design with formal verification



* Source: Esterel Technologies (Ansys)

Summary

- What are formal methods?
  - Formalism, formal language
  - Formal methods and tools:
    Simulation, formal verification, synthesis
- What are they good for?
  - Motivation: software quality challenges
  - Potential of formal methods
- What can we expect?
  - Limitations
  - Success stories