Entwurfsüberprüfung digitaler Schaltungen mittels formaler Verifikation — Ein Überblick

Bernd Straube
Jens Schönherr
Eva Fordran

Famous Failures

The »Pentium Bug«, 1994
• Error in floating-point division algorithm
• Bug rarely occurs after 4th decimal digit
• $200,000 to repair design
• Replacement of faulty processor chips costs Intel $480,000,000

Railway Accident in the Clayton Tunnel, 1861
• Only one train was allowed per track in the tunnel.
• Predefined messages between the signalmen did not consider more than one train in the tunnel
• Non-compatible software interfaces

4 June 1996: ARIANE 5 Disaster
• An if-then branch with a wait-statement used only for the software test was not removed.
• Due to a faulty signal coming from the network the program just came to that wait-statement.

Outline

Introduction
• Famous Failures, Design Errors
• The Basic Design Cycle
• Validation, Verification

Binary Decision Diagrams (BDD)

Equivalence Checking
• Combinational Circuits
• Sequential Circuits

Model Checking
• CTL

Theorem Prover

Design Verification Outlook
• Design for Verifiability
• Limitations of Formal Verification

Design Cycle, Verification, Testing

Phase 1
• Idea
  • formalization
    • specification
      • design steps
        • network, layout
          • prototyping
            • fabrication
              • TC
                • application
                  • field returns
                    • defect analysis
                      • undetected defects
                        • defects
                          • disturbances
                            • usage faults

Phase 2
• design errors

Phase 3
• Phase 4
• Phase 5
• Phase 6
**Design Cycle, Verification, Testing**

- **Phase 1**: Idea
- **Phase 2**: Formalization
- **Phase 3**: Specification
- **Phase 4**: Design steps
- **Phase 5**: Network, layout
- **Phase 6**: Prototyping, fabrication

**Design Errors**

- **Examples**
  - Misinterpreting the informal specification
  - Incorrect or incomplete or vague specifications
  - Misunderstandings between designers
  - Mistake in one's logic
  - Using false algorithms
  - Typing mistake
  - Resource conflicts

- Unlikely events are usually ignored.
  - The consequences of an error are often far more important than the probability of the error.
  - »Hypothetical« once-in-a-lifetime errors

- Applying a design tool with a bug

**Example**

- **Correct**
  - 
  - \[
  \begin{align*}
  W / R / J & \to 21 \text{ drew} \\
  J / W / J & \to 23 \text{ f} \\
  J / J / W & \to 22 \text{ f} \\
  J / W & \to 24 \text{ f}
  \end{align*}
  \]

- **Faulty transition**
  - 
  - \[
  \begin{align*}
  W / R / J & \to 21 \text{ drew} \\
  J / W & \to 23 \text{ f} \\
  J / J / W & \to 22 \text{ f} \\
  J / W & \to 24 \text{ f}
  \end{align*}
  \]

**Example**

- **Correct**
  - 
  - \[
  \begin{align*}
  W / R / J & \to 21 \text{ drew} \\
  J / W / J & \to 23 \text{ f} \\
  J / J / W & \to 22 \text{ f} \\
  J / W & \to 24 \text{ f}
  \end{align*}
  \]

- **Faulty network**
  - 
  - \[
  \begin{align*}
  W / R / J & \to 21 \text{ drew} \\
  J / W / J & \to 23 \text{ f} \\
  J / J / W & \to 22 \text{ f} \\
  J / W & \to 24 \text{ f}
  \end{align*}
  \]
The Basic Design Cycle

Validation and Verification

Concept 1: Verification Validation

The design is incompletely checked. By validation methods faults can be detected rather than the absence of faults can be shown.

Concept 2: Formal Verification Validation

Validation

Remark

Formal Hardware Verification: A description of hardware is verified rather than the hardware per se.

Validation of the Specification

Does the specification correspond to the idea?

Approaches

• Visual inspection
• Simulation
• Model checking
**Verification of the Implementation**

Does the implementation satisfy the specification?

- Correctness by construction
- Equivalence checking
- Model checking
- Exhaustive simulation

**Approaches**

**Validation of the Implementation**

Does the implementation comply with the idea?

- Visual inspection
- Simulation
- Emulation
- Model checking

**Approaches**

**Simulation**

Simulation of a commercial ALU

- Central part of a processor
- 2 * 32 data inputs, 13 control inputs
- 77 inputs → $2^{77} \approx 1.5 \times 10^{23}$ cases
- Simulation-time for 1.5 * $10^{23}$ cases:
  - $10^{-6}$ sec. per case → $1.5 \times 10^{17}$ sec. ~ 4.8 * $10^9$ years
- Exhaustive simulation impossible

- Who provides the simulation stimuli?
- Who interprets the simulation results?

(H. Eveking: Tutorial »Formal Verification«, ED&TC'97)

**Design Process and Verification**

- Informal Description, Requirements
- Formalizing
- Formal Specification
- Specification (a priori correct, fault-free)
- Design Step
- Does the implementation fulfill the specification?

This implementation is used as specification.
**Design Process and Verification**

- Abstraction Level (i)
- Specification
- Design Step
- Implementation

**Principle of Verification**

- Specification → Transformation $T_s$ → Model-$S$
- Implementation → Transformation $T_i$ → Model-$I$

$T_s, T_i$: e.g. setting-up the automata functions

**Representations of Boolean Functions**

- Truth table:
  - $a \ b \ c$
  - $0 \ 0 \ 0$
  - $0 \ 1 \ 0$
  - $1 \ 0 \ 0$
  - $1 \ 1 \ 1$
- Expression: $c = ab$
- Logical network
- BDD (binary decision diagram):

**Binary Decision Diagrams (BDDs)**

- $a$
- $b$
- $0$
- $1$
- $0$
- $1$
- $0$
- $1$
Boole’s Expansion Theorem

Example

- Given \( f(a,b,c) \)
- cofactors of \( f \) w.r.t. \( a \)
  \[
  f_{\neg a} = f(0,b,c) \\
  f_a = f(1,b,c) \\
  f = \neg a \cdot f_{\neg a} + a \cdot f_a
  \]

\[
\begin{array}{c|cc}
  a & b & c \\
  \hline
  0 & 0 & 0 \\
  0 & 0 & 1 \\
  1 & 1 & 0 \\
  1 & 1 & 1 \\
\end{array}
\]

Reduced Ordered BDD

Reduced Ordered BDD (ROBDD)

variable ordering \( \rightarrow \) unique representation

OBDDs are canonical

rules for sharing identical subtrees

BDD-Size Reduction Techniques

- Sharing identical subtrees

\[
\begin{array}{c|c|c}
  a & b & c \\
  \hline
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 0 \\
  1 & 1 & 1 \\
\end{array}
\]

- Subtree-sharing

- Negation edges

Example 1

- Variable ordering given

Example:

\[
y = \overline{abc} + ac
\]

Ordering: \( a < b < c \)
Example 1 (Forts.)

\[ y = \overline{a}bc + ac, \quad a < b < c \]

Example 2

Unique Representation

- The Boolean functions of these different XOR networks are represented by the same BDD

Influence of Variable Ordering

BDDs for \( f = (x_0 \leftrightarrow x_2) \land (x_1 \oplus x_3) \)

Ordering: \( x_0 < x_1 < x_2 < x_3 \)

11 BDD-nodes

Ordering: \( x_0 < x_2 < x_1 < x_3 \)

8 BDD-nodes

Complexity

Space

- Worst case: exponential wrt. the number of variables
- Best case (AND, OR, XOR): linear wrt. the number of variables
- Size depends on variable ordering (linear \( \leftrightarrow \) exponential)
- For some Boolean functions size of all BDDs is exponential for every variable ordering

Time

- Binary Boolean operations are polynomial with the number of nodes
- Negation and comparison is constant time
- Setting-up a BDD for a circuit description is exponential
Representing Sets by means of Boolean Functions

Characteristic Function

\[ A \subseteq X \implies \chi_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \not\in A \end{cases} \]

- \( B = \{0, 1\} \)
- \( X = B^n = \{0, 1\}^n \)
- \( \chi_A : B^n \to B \) is a Boolean function

Example

- \( P_1 = \{(a, b)\} \)
- \( P_2 = \{(a, b)\} \)
- \( I = P_1 \cup P_2 \)
- \( i = p_1 + p_2 = a + b \)

Example: Boolean Relation

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ r = \overline{a} \overline{b} \overline{c} d + \overline{a} b \overline{c} \overline{d} + a \overline{b} \overline{c} \overline{d} + a b c d \]

Algebra of Boolean Relations

Union +, Intersection *, etc. and Composition

Composition of the functions \( r \) and \( i \):

\[ o(y) = \exists_x (r(x,y) i(x)) \]

\[ \bigcup_u g(u,v) = g(1,v) + g(0,v) \]

\[ \exists (a,b)(... \implies \exists a( \exists b( ... )) \)
Input-Output Equivalence

Definition

Two automata are said to be **input-output-equivalent** if they answer to any input sequence with the equal output sequence.

![Schematic diagram of input-output equivalence](image1)

Verifying Combinational Networks

**Boolean Comparison**

- Correspondences between the network pins
- Interconnection of the corresponding input pins
- For each output:
  - Calculating the Boolean functions (BDDs)
  - Comparing the corresponding output functions of the specification and implementation

![Schematic diagram of Boolean comparison](image2)

**SAT-Approaches**

- For each output:
  - Calculating the Boolean functions (BDDs)
  - Comparating the corresponding output functions of the specification and implementation

![Schematic diagram of SAT-Approaches](image3)
Reducing the Boolean Complexity

Methods

- Blackboxing a subnetwork
- Introducing breakpoints (equivalent signal pairs)
- Finding isomorphic subnetworks (netlists)

Blackboxing

- Removing a module out of the network to be verified
- Both parts are verified separately

Blackboxing - False Negatives

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

 pseudo inputs

 pseudo outputs

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>u</th>
<th>v</th>
<th>x</th>
<th>y</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>u</th>
<th>v</th>
<th>x</th>
<th>y</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Blackboxing - False Negatives

Reducing the Boolean Complexity

Introducing breakpoints

- Finding internal nodes in both specification and implementation such that the corresponding cones (transitive fanin) are equivalent.
  - during the calculation of the BDDs
  - selecting candidates by excluding nodes by means of random pattern simulation
- These cones are removed and this node becomes a pseudo input.

False negatives may also arise.
Reducing the Boolean Complexity

Introducing breakpoints

- False negatives may also arise.

Verifying Synchronous Sequential Networks

Boolean Comparison

- Correspondences between the network pins
- Let the correspondences between the state variables be given.
  - Combinational network verification problem
- For each output and next-state variable:
  - Calculating the Boolean function (BDDs)
  - Comparing the corresponding output and next-state functions of the specification and the implementation

Huffman Model

input \( x^t \)

present state \( s^t \)

\( y^t = \lambda(s^t, x^t) \) output

\( s^{t+1} = \delta(s^t, x^t) \) next-state

combinational network

F/F

Clock
Finding Correspondence between State Variables

Possibilities

- Correspondences are given by designer.
- Information delivered by the synthesis tool
- Unique (sub) strings in the names of the state variables
- Exclusions by random-pattern simulation
- Automated technique for setting-up a correspondence relation on the state variables
  - A correspondence relation is an equivalence relation.
  - Forming an equivalence relation by iterative exclusion of non-corresponding variables

Verifying Synchronous Sequential Networks

Problem

- There is a correspondence between some state variables only.
  (e.g., after the optimization of the state encoding)

Input / Output Equivalence

Two automata are said to be **input-output-equivalent** if their product automaton answers to any input sequence with the constant 0 output sequence.

Forming the Product Automaton

specification

implementation

A_s = (I, O, S_s, δ_s, λ_s, S_0s)

A_i = (I, O, S_i, δ_i, λ_i, S_0i)

A_p = (I, (0,1), S_p, δ_p, λ_p, S_0p)
Example Product Automaton

Specification

Implementation (faulty)

Example Product Automaton

Specification \times Implementation (faulty)

Example Product Automaton

Specification \times Implementation (correct)

Input / Output Equivalence

Techniques

- Symbolic state space traversals
  - forward traversal
  - backward traversal
  - approximate forward / exact backward traversal
Forward Traversal

Start
- Initial states $S_0$

Fixpoint iteration
- Recursive calculation of the next-state set

Result
- Set of reachable states $R$

Check
- Are there any non-equivalent states in $R$?

Calculation of the fixpoint

$R = \mu X. (S_0 \cup \delta(I \times X))$ by iteration:

- $R_0 = S_0$
- $R_i = R_{i-1} \cup \exists I . \delta(I \times R_{i-1})$
- Termination if $R_i = R_{i-1}$ with $R = R_i$

Forward Traversal: Example

Specification $\times$ Implementation (faulty)

$R_{14} = R_{15} = R$
Forward Traversal: Example

Specification x Implementation (correct)

State Space Traversals

Experiences

- The efficiency of the different techniques and their variants depends on the kind of automata.
- Forward traversal should be applied at first.
- Backward traversal shows very different efficiencies.
- A forward / backward traversal does not result in handling larger automata.

There are several heuristic state space traversal approaches with very different efficiencies.

Structural Traversal of Sequential Circuits

Iterative Array Model

- derived from Huffman-Model
- is a combinational representation of a sequential circuit

Iterative Array Model

- derived from Huffman-Model
- is a combinational representation of a sequential circuit
Structural Traversal of Sequential Circuits

Iterative Array
Model of the Product Automaton

Generating a Counterexample by Sequential ATPG

Non-Equivalence
There exists an input series causing an output series $c \neq \text{const 0}$ ("counterexample")

Using (modified) Sequential Automatic Test Pattern Generation
- Injecting a stuck-at-0 fault on the output node of the product network (product automaton)
- If this fault is not redundant, i.e. in case of non-equivalence, there exists a test series for this fault
- A fault controllability series has to be generated only, i.e. backward justification process
- Assumption: Initial state known, e.g. by reset (0, 0)

Generating a Test Series
- Combinational iterative model of synchronous sequential network
- A fault controllability series has to be generated only.
- Initial state by reset (0, 0)
**Basic Task**

**Does the specification correspond to the idea?**

**Example**

**How to find out that state $z_3$ cannot be left?**

**By model checking**
Model Checking

Temporal properties are verified against a model of the system.

- Safety properties are "something bad that should never happen"
  - It is impossible that all green lights are on.
  - If signal A changed then signal B turns to high four clocks later unless signal C has changed.
- Liveness properties are "something desirable that should eventually happen"
  - Finally, there will be green light for me.
- There are many languages to express temporal properties.
  e.g. LTL, CTL, CTL*, μ-calculus, STE, ...

Examples of Temporal Properties

Definition of Properties

- The properties to be checked have to defined by the designer.

Model Checker

- The properties are completely verified or counterexamples are provided.

Computational Tree

Transition System

Computational Tree

(E. Clarke et al: Verification tools for finite-state concurrent systems. LNCS 803, 1993)
Computational Tree Logic CTL

- State graph is unwound to obtain an infinite tree, the computational tree.
- Temporal properties are properties of this tree.
- Propositional logic plus temporal operators
  - $A$, $E$ path quantifiers
    - $X$ ~ next
    - $F$ ~ finally
    - $G$ ~ globally
    - $U$ ~ until

Temporal properties:
- $AX(p)$ characterizes all states where $p$ is certain in the next step.

Fixed-point calculation:
$$K_0 := p$$
$$K_{i+1} := K_i \cup AX(K_i)$$
$$p \subseteq AF(p)$$

- $EX(p)$ characterizes all states where $p$ is possible in the next state.

Fixed-point calculation:
$$K_0 := p$$
$$K_{i+1} := K_i \cap AX(K_i)$$
$$AG(p) \subseteq p$$

AF(p): $p$ is unavoidable (now or later)
Fixed-point calculation:
$$K_0 := p$$
$$K_{i+1} := K_i \cup AX(K_i)$$
$$p \subseteq AF(p)$$

AG(p): $p$ holds always
Fixed-point calculation:
$$K_0 := p$$
$$K_{i+1} := K_i \cap AX(K_i)$$
$$AG(p) \subseteq p$$

(H. Eveking: Tutorial »Formal Verification«, ED&TC'97)
Model Checking - Example

How to find out that state $z_3$ cannot be left?

A CTL Formula: $AG AF\{z_1\}$
- If $AG AF\{z_1\}$ does not hold then there is a deadlock.
- $AG AF\{z_1\}$ says: It is unavoidable that the state $z_1$ is reached from each state with a finite number of clocks.

Calculating $AG AF\{z_1\}$ in two steps:
- step 1: Calculating $AF\{z_1\}$
  $K_0 := \{z_1\}$
  $K_1 := \{z_1\} \cup AX\{z_1\} = \{z_1\} \cup \{z_4\}$
  $K_2 := \{z_1, z_4\} \cup AX\{z_1, z_4\} = \{z_1, z_4\} \cup \{z_4\}$
  $K_{fp1} := \{z_1, z_4\}$
- step 2: Calculating $AG\{z_1, z_4\}$
  $K_0 := K_{fp1}$
  $K_{fp2} := K_1 \cap AX\{K_1\}$ until fixpoint $K_{fp2}$ is reached

Calculating $AG AF\{z_1\}$

(Faulty Network)

• step 1: Calculating $AF\{z_1\}$
  $K_0 := \{z_1\}$
  $K_1 := \{z_1\} \cup AX\{z_1\} = \{z_1\} \cup \{z_4\}$
  $K_2 := \{z_1, z_4\} \cup AX\{z_1, z_4\} = \{z_1, z_4\} \cup \{z_4\}$
  $K_{fp1} := \{z_1, z_4\}$

• step 2: Calculating $AG\{z_1, z_4\}$
  $K_0 := \{z_1, z_2, z_3, z_4\}$
  $K_1 := \{z_1, z_2, z_3, z_4\} \cap AX\{z_1, z_2, z_3, z_4\} = \{z_1, z_2, z_3, z_4\}$
  $K_{fp1} := \{z_1, z_2, z_3, z_4\}$

(Correct Network)

• step 1: Calculating $AF\{z_1\}$
  $K_0 := \{z_1\}$
  $K_1 := \{z_1\} \cup AX\{z_1\} = \{z_1\} \cup \{z_2, z_3, z_4\}$
  $K_2 := \{z_1, z_2, z_3, z_4\} \cup AX\{z_1, z_2, z_3, z_4\} = \{z_1, z_2, z_3, z_4\}$
  $K_{fp1} := \{z_1, z_2, z_3, z_4\}$

• step 2: Calculating $AG\{z_1, z_2, z_3, z_4\}$
  $K_0 := \{z_1, z_2, z_3, z_4\}$
  $K_1 := \{z_1, z_2, z_3, z_4\} \cap AX\{z_1, z_2, z_3, z_4\} = \{z_1, z_2, z_3, z_4\}$
  $K_{fp2} := \{z_1, z_2, z_3, z_4\}$, i.e. property holds
**Bounded Model Checking**

**Basic Idea**
- Considering counterexamples of a particular length $k$
- Generating a propositional formula that is satisfiable iff such a counterexample exists.
- Unrolling the transition relation $k$-times
- Deriving a propositional formula
- Using a SAT-technique

**Advantages**
- Counterexamples are found very fast.
- The counterexamples are of minimal length.
- The designer can understand a counterexample more easily.
- Unlike BDD-based techniques, bounded model checking
  - uses much less space
  - does not needed variable ordering

**Theorem Prover**

**HOL**
- Interactive theorem proving based on Higher Order Logic (HOL)
- Very few number of axioms — prover seems to be correct
- Low level of automation

**PVS**
- Interactive theorem proving based on HOL
- Includes procedure for automatic reasoning about quantifier-free formulas of First Order Logic (FOL)
- Includes CTL model checker
- Is considered as to be the most easy-to-use theorem prover

**NQTHM (ACL2)**
- Quantifier-free FOL and recursive functions
- Automated nested induction proofs
- Interaction needed if proof fails — lemma(s) must be given.
- It is difficult to find out which lemmas are needed.
Example 1: Sensor System

**Task**

Qualitative recognition of a rotary motion

---

**Examples**

---

**Example 1: Sensor System**

**VHDL - Model**

![VHDL Diagram]

**Verification-Oriented Modelling**

- Environment of the sensor
- Clock and reset
- Default values and initial values
- Abstraction
  - Decreasing the number of the states of the timer

---

**Diagram of Sensor System**

- Sensor1
  - 45 degrees
  - Sensor2
- FE0
  - MAX
- FE1
  - fast LEFT
- LX
  - fast LEFT
- L
  - LEFT
- H
  - HALT
- R
  - RIGHT
- RX
  - fast RIGHT
Example 1: Sensor System

Model Checking Results
- Checking mission modes
  - Is there any input series that causes (R & L) = 1

Example 1: Sensor System

Model Checking Results
- Checking mission modes
  - Is there any input series that causes (R & L) = 1

- Aspects of Dependability
  - Behaviour in case of internal defects?
  - Behaviour in case of external failures?

Example 2: Clayton Tunnel

A Model
- Composition of non-deterministic automata

Design Verification Outlook
Verification Flow

- Specification → Specifying properties → Properties → Model Checking
- RTL Design → Synthesis → Equivalence Checking → Logic Design

Design for Verifiability

- Equivalence checking
  - Easy-to-find
    - correspondences between state variables
    - correspondences between internal variables
  - Model checking
    - Description of the specification should be
      - well-structured
      - unique
      - complete
      - The Problem
        - Has the designer defined a complete set of properties?
        - A property that, however, is missing would have detected a bug.

Fault Avoidance - Fault Detection

- Design Fault Avoidance
  - Correctness by construction
    - Synthesis based on proven transformations
  - Problem:
    - Verified algorithms implemented by unverified software

- Design Fault Detection
  - Validation techniques
  - Formal verification techniques

- Fault Tolerance
  - Fault masking by redundant system components

Limitations of Formal Verification

- The usage of formal verification cannot guarantee a correctly working hardware system
  - Properties are
    - missing
    - incomplete
    - contradictory
  - The verification tool has a bug.
  - The verification-oriented model is
    - inconsistent
    - too coarse
    - even wrong
    - undetected defects
    - incorrect specification
    - usage, disturbances
    - application
Design Verification Outlook

- System
  - verification method A
  - validation
  - verification method C
- verification method B
- model checking
- validation