Introduction to Formal Verification

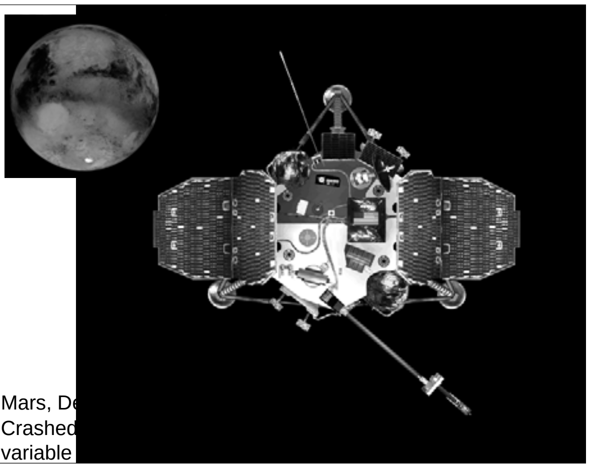
Gianpiero Cabodi

Outline

- Motivations
- Verification Approaches
- Formal Verification vs. Simulation
- Theorem Proving
- Model Checking
- Equivalence Checking

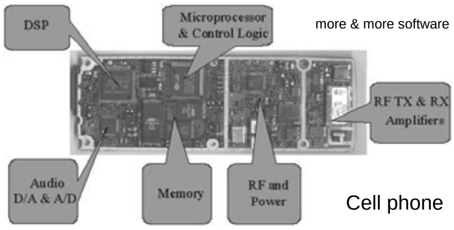
Motivations

- Digital systems continuously grow in scale and functionality, 10Mgates now, ...
  - Performance of integrated circuits (IC) doubling every year
  - Microprocessors containing 5M gates, doubling of frequency per generation, transistor scale by 30% per generation
  - Telecommunication chips are deep submicron application-specific integrated circuits (ASICs) with more than 1M gates
  - I/O pins limit observability and controllability, likelihood of design errors increasing
  - In 1994, problems with Intel Pentium and Pentium Pro microprocessors. Cost of correction about $250 M. In 1995, problem with TI 320C32 floating point digital signal processor
  - Failure of Ariane 6 due to bad specification of SW module for reuse



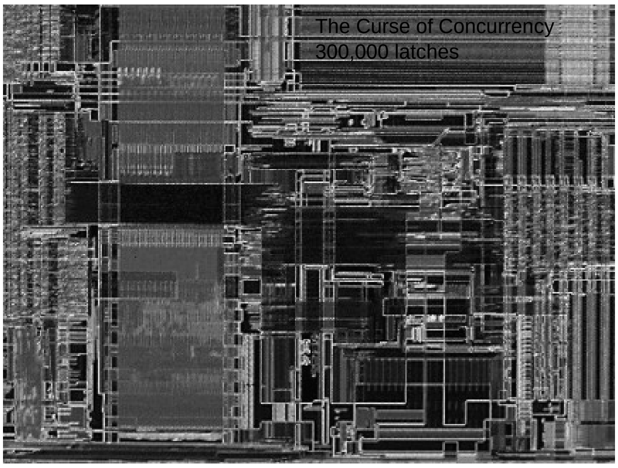
Mars, De
Crashed
variable

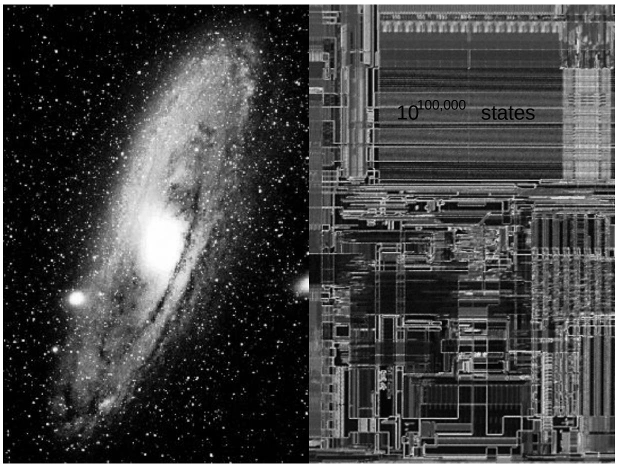
Embedded System



Concurrency: - component-based design
- system interacts with environment

Heterogeneity: - digital and analog components
- discrete-time and real-time interaction





Verification is an Industry-wide issue

Intel: **Processor project verification:**
"Billions of generated vectors"
"Our VHDL regression tests take 27 days to run."

Sun: **Sparc project verification:**
Test suite ~1500 tests > 1 billion random simulation cycles
"A server ranch ~1200 SPARC CPUs"

Bull: **Simulation including PwrPC 604**
"Our simulations run at between 1-20 CPS."
"We need 100-1000 cps."

Cyrix : **An x86 related project**
"We need 50x Chronologic performance today."
"170 CPUs running simulations continuously"

Kodak: "hundreds of 3-4 hour RTL functional simulations"
Xerox: "Simulation runtime occupies ~3 weeks of a design cycle"
Ross: 125 Million Vector Regression tests

Design Teams are *Desperate* for Faster Simulation

Kurt Keutzer

Goals of Formal Verification

- **Complement to simulation to improve design quality.**
- *Formal Methods*: mathematically-based languages, techniques, and tools for specifying and verifying systems
- Increase understanding of a system by revealing *inconsistencies*, *ambiguities*, and *incompleteness*

.... often even by just going through the process of rigorous specification...

The main point is NOT

- correctness proof of entire systems
- replacing test entirely

BUT

- one proof can replace many test cases
- formal methods can be used in automatic test case generation

Successful formal methods

- Integrated in the design flow
- Avoid new demands on the user
- Work at large scale
- Save time or money in getting a good quality product out

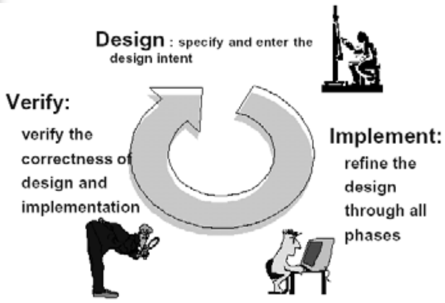
Terminology

- **Formal Methods** is the application of logic to the development of "correct" systems
- **Correctness** is classically viewed as two separate problems, **validation** and **verification**
- **Validation**: answers "are we building the right system?"
- **Verification**: answers "are we building the system right?"
- **Formal Validation**: Can we use logic to help ensuring that the specification is complete, consistent, and accurately captures the customer's requirements
- **Formal Verification**: Can we use logic to help ensuring that the system built faithfully implements its specification

Application of Formal Verification

- Formal methods are used today in many **applications** including:
  - - Microprocessor Design
  - - Cache Coherency Protocols
  - - Telecommunications Protocols
  - - Rail and Track Signaling
  - - Security Protocols
  - - Automotive Companies

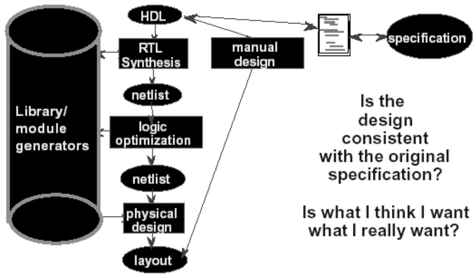
Design Process



Verification

- Design Verification
- Implementation Verification
- Manufacture Verification (Test)

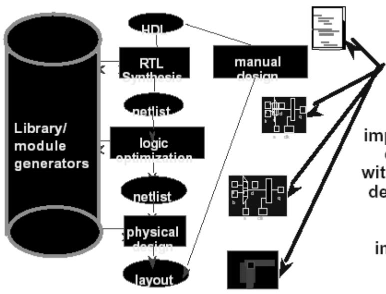
Design Verification



Is the design consistent with the original specification?

Is what I think I want what I really want?

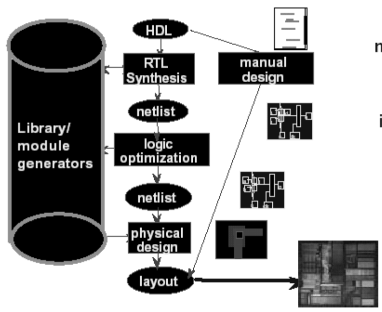
Implementation Verification



Is the implementation consistent with the original design intent?

Is what I implemented what I wanted?

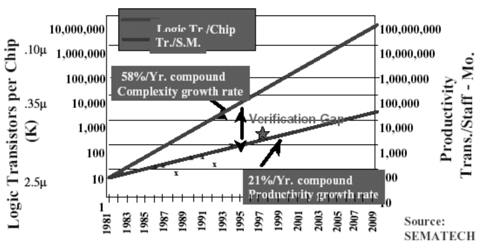
Manufacture Verification (Test)



Is the manufactured circuit consistent with the implemented design?

Did they build what I wanted?

Verification Gap



Kurt Keutzer

Why the gap

$$\frac{\text{logic\_transistors}}{\text{chip}} \times \frac{\text{lines\_in\_design}}{\text{logic\_transistors}} \times \frac{\text{bugs}}{\text{line\_of\_design}} = \frac{\text{bugs}}{\text{chip}}$$

Kurt Keutzer

Filling in reasonable numbers

$$\frac{\text{logic\_transistors}}{\text{chip}} \times \frac{\text{lines\_of\_design}}{\text{logic\_transistors}} \times \frac{\text{bugs}}{\text{lines\_of\_design}}$$

$$\frac{10{,}000{,}000\ \text{trs}}{\text{chip}} \times \frac{1}{10} \times \frac{1}{10{,}000}$$

$$= \frac{100\ \text{bugs}}{\text{chip}}$$

Kurt Keutzer

Raising the Level of Abstraction

$$\frac{\text{logic\_transistors}}{\text{chip}} \times \frac{\text{lines\_of\_design}}{\text{logic\_transistors}} \times \frac{\text{bugs}}{\text{lines\_of\_design}}$$

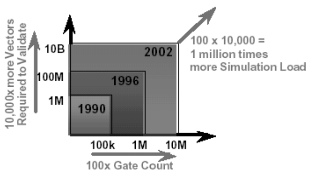
$$\frac{10{,}000{,}000\ \text{trs}}{\text{chip}} \times \frac{1}{100} \times \frac{1}{10{,}000}$$

$$= \frac{10\ \text{bugs}}{\text{chip}}$$ ***this year!!***

Kurt Keutzer

The Verification Bottleneck

Verification problem grows even faster due to the combination of increased gate count and increased vector count

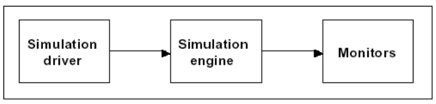


Kurt Keutzer

Approaches to Design Verification

- **Software Simulation**
  - **Application of simulation stimulus to model of circuit**
- **Hardware Accelerated Simulation**
  - **Use of special purpose hardware to accelerate simulation of circuit**
- **Emulation**
  - **Emulate actual circuit behavior - e.g. using FPGA's**
- **Rapid prototyping**
  - **Create a prototype of actual hardware**
- **Formal verification**
  - **Model checking - verify properties relative to model**
  - **Theorem proving - prove theorems regarding properties of a model**

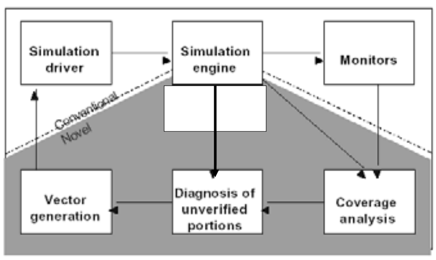
Simulation: The Current Picture



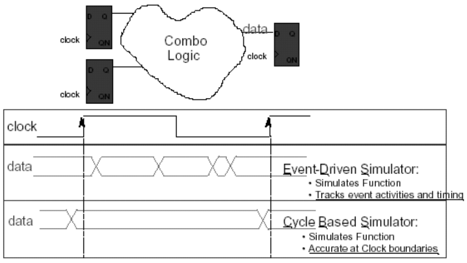
SHORTCOMINGS:

- **Hard to generate high quality input stimuli**
  - **A lot of user effort**
  - **No formal way to identify unexercised aspects**
- **No good measure of comprehensiveness of validation**
  - **Low bug detection rate is the main criterion**
    - **Only means that current method of stimulus generation is not achieving more.**

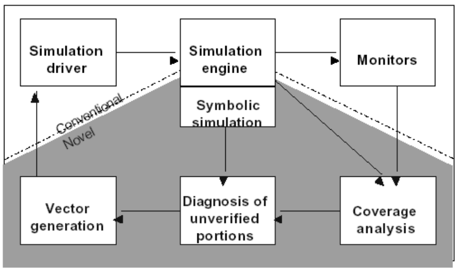
Simulation-based Verification



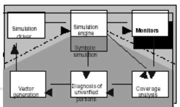
Event vs. Cycle-based Simulation



Symbolic Simulation



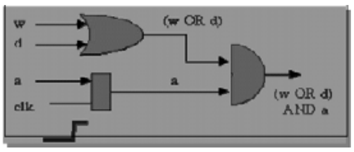
Symbolic Simulation



IDEA: One symbolic run covers many runs with concrete values.

Some inputs driven with symbols instead of concrete values

- $2^{(\# \text{ symbols})}$ **equivalent binary coverage**



Automated Synthesis: an Alternative to Simulation?

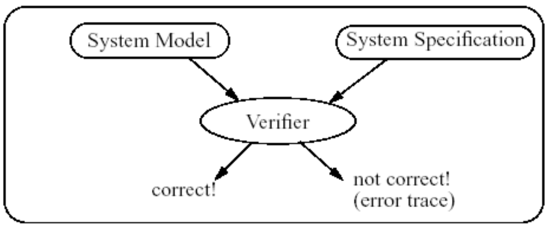
- An alternative to post-design verification is the use of *automated* synthesis techniques—*correct-by-construction*
- Logic synthesis techniques successful in automating low-level (gate-level) logic design
- Progress needed to automate the design process at *higher levels*.
- Until synthesis technology matures high-level design done manually
  - Requires post-design verification.
- Top-level specification/design must always be checked against properties of the "idea"
  - No golden reference at that level

Formal Verification: Another Alternative to Simulation!

Formal Verification is the process of constructing a proof that a target system will behave in accordance with its specification.

- Use of *mathematical reasoning* to prove that an implementation satisfies a specification
- Like a mathematical proof: correctness of a formally verified hardware design holds *regardless of input values*.
- Consideration of *all cases is implicit* in formal verification.
- Must establish:
  - A formal *specification* (properties or high-level behavior).
  - A formal description of the *implementation* (design at higher level of abstraction — *model* (observationally) equivalent to implementation or implied by implementation).

Formal Verification



Formal Verification

- Complete with respect to a given property (!)
- Correctness guaranteed mathematically, regardless the input values
- No need to generate expected output sequences
- Can generate an error trace if a property fails: better understand, confirm by simulation
- Formal verification useful to detect and locate errors in designs
- Consideration of *all cases is implicit* in formal verification

Simulation vs. Formal Verification

Example: $(x+1)^2 = x^2 + 2x + 1$

Simulation Values:

x	$(x+1)^2$	$x^2 + 2x + 1$
0	1	1
1	4	4
2	9	9
3	16	16
9	100	100
67	4624	4624
...	...	...

Simulation vs. Formal Verification

- Formal Proof

1.	$(x+1)^2 = x^2 + 2x + 1$	definition of square
2.	$(x+1)(x+1) = (x+1)x + (x+1)1$	distributivity
3.	$(x+1)^2 = (x+1)x + (x+1)1$	substitution of 2. in 1.
4.	$(x+1)1 = x+1$	neutral element 1
5.	$(x+1)x = xx + 1x$	distributivity
6.	$(x+1)^2 = xx + 1x + x + 1$	substitution of 4. and 5. in 3.
7.	$1x = x$	neutral element 1
8.	$(x+1)^2 = xx + x + x + 1$	substitution of 7. in 6.
9.	$xx = x^2$	definition of square
10.	$x + x = 2x$	definition of 2x
11.	$(x+1)^2 = x^2 + 2x + 1$	**substitution of 9. and 10. in 8.**

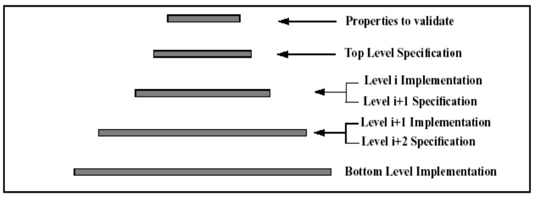
Simulation vs. Formal Verification

- Simulation: *complete* (real) model, *partial* verification
  Verification: *partial* (abstract) model, *complete* verification
- Simulation still needed to tune specifications; for large complete designs
- Verification can generate counter-examples (error traces); possibly false negatives!
- Techniques are complementary — formal verification gives additional confidence, e.g.,
  - Apply formal verification of abstract model
  - Obtain error trace if bug found (may be false negative!)
  - Simulate error trace on the real model

Simulation vs. Formal Verification

- Common difficulty in all verification methods:
  - lack of “golden” reference
  - what properties to verify .....?
- *“Simulation and formal verification have to play together.”* **[IEEE Spectrum, January 1996]**

Hierarchical Verification



Hierarchical Verification

- ***Specification*** *(Spec)*
  - *Properties*: enumeration of assumptions and requirements,
  - *Functions*: desired behavior or design descriptions,
  - *State machines*: desired behavior or design descriptions,
  - *Timing requirements*, etc.
- ***Implementation*** *(Imp)* refers to the design to be verified.
  - Corresponds to a description at any level of abstraction, not just the final physical level.
  - Can serve as a specification for the next lower level.

Formal Specification

- A *specification* is a description of a system and its desired properties
- Useful as a communication device:
  - between customer and designer,
  - between designer and implementor, and
  - between implementors and tester
- Companion document to the system's source code, but at a higher level of abstraction
- Properties relate to function, interfaces, timing, performance, power, layout, etc.

Formal Specification

- *Formal specification*. Use of formal methods (a language with mathematically-defined syntax and semantics) to describe the intended behavior of the system:
  - The language of logic provides an unambiguous method of recording the specification
  - We can reason about a formal specification to check that the system specified will possess other desired properties
- The process of writing a formal specification helps uncover ambiguity and incompleteness
- Formal specifications most successful for functional behavior, also interface & timing
- Trend to integrate different specification languages, each for a different aspect (e.g. VERA, SystemC, VHDL+)

Specification Validation

- Whether the specification means what it is intended to mean
- Whether it expresses the required properties
- Whether it completely characterizes correct operation, etc.

(Validation methods: simulation or formal techniques)

Formalisms for representing specifications

- Logic: propositional, first-order predicate, higher-order, modal (temporal), etc.
- Automata/language theory: finite state, omega automata, etc.

Types of properties

- Functional correctness properties;
- Safety (invariant) and Liveness properties
  E.g.: in a mutual exclusion system with two processes A and B
  - *Safety property* (**nothing bad will ever happen**): e.g. simultaneous access will never be granted to both A and B. If false, can be detected by finite sequences
  - *Liveness property* (**something good will eventually happen**): e.g. if A wants to enter its critical section, it will eventually do so. Can only be proved false by infinite sequences (any finite sequence can be extended to satisfy the eventuality condition)

Limitations of Formal Verification

Just because we have proved something correct does not mean it will work! There are gaps where formal verification connects with the real world.

- Does the specification actually capture the designer's intentions?
  - Specification must be simple and abstract
  - Example of a good specification for a half-adder: $out = (in_1 + in_2) \bmod 2$
- Does the implementation in the real world behave like the model?
  - Can in_1 drive three inputs
  - What happens if the wires are fabricated too close together?
  - Do we need to model quantum effects on the silicon surface?

State of the Art

- In the 1960-70's, high expectations for "software verification", but hopes gradually fizzled out by the late 1970's
- Theorem proving approaches have "cultural roots" in software verification in 1970's (Hoare, Owicki, Gries)
- The use of formal methods did not seem practical
  - notations too obscure
  - techniques did not scale with problem size
  - tool support inadequate or too hard to use
  - Only a few non-trivial case studies available
  - Few people had the necessary training

State of the Art

- Why formal methods might work well for "hardware verification"?
  - Hardware is often regular and hierarchical
  - Re-use of design is common practice
  - Hardware specification is more common, e.g., VHDL models
  - Primitives are simpler, e.g., behavior of an NAND-Gate easier to describe than the
  - semantics of a while-loop
  - Cost of design error can mean a 6 months delay and a costly set of lithography masks

State of the Art

- Recently more promising picture
  - Software specification: industry trying out notations like SDL or Z to document system's properties
  - Protocol verification successful
  - Hardware verification: industry adopting model checking and some theorem proving to complement simulation
  - Industrial case studies increasing confidence in using formal methods
  - Verification groups: *IBM, Intel, Motorola, HP, Nortel, NEC, Fujitsu, SUN, Cadence, Siemens, Synopsys, Lucent Technologies,* ....
  - Commercial tools from: *Chrysalis, Cadence, Synopsys, Verysys, IBM,* ....

Focus

- In this course, we focus on formal verification methods of digital hardware
- ... but model checking is making inroads into software verification of real-time reactive systems and protocols

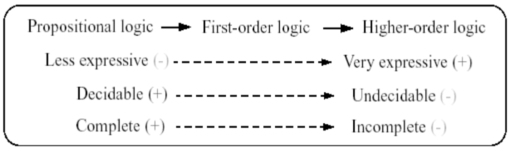
Formal Logic

- A method is **formal** if its rules for manipulation are based on form (*syntax*) and not on content (*semantics*)
- Majority of existing formal techniques are based on some flavor of formal (symbolic) logic: Propositional logic, Predicate logic, other logics.
- **Formal logic**
  - Every logic comprises a formal language for making statements about objects and reasoning about properties of these objects.
  - Statements in a logic language are constructed according to a predefined set of formation rules (depending on the language) called *syntax rules*.
  - A logic language can be used in different ways.

Types of Logic

- Propositional logic: traditional *Boolean* algebra, variables $\in\{0,1\}$
- First-order logic (Predicate logic): quantifies *for all* ($\forall$) and *there exists* ($\exists$) over variables
- Higher-order logic: adds reasoning about (quantifying over) sets and functions (predicates)
- Modal/temporal logics: reason about what *must* or *may* happen

Types of Logic



- Propositional logic: decidable and complete
- First-order logic: decidable but not complete
- Higher-order logic: not decidable nor complete

Proof Theory

- A formal logic system consists of:
  - a notation (syntax)
  - a set of axioms (facts)
  - a set of inference (deduction) rules
- A formal proof is a sequence of statements where every statement follows from a preceding one by a rule of inference
- Purely syntactic (mechanical) activity; not concerned with the meaning of statements, but with the arrangement of these statements, and whether proofs can be constructed

Model Theory

- The second use of a logic language is for expressing statements that receive a meaning when given an interpretation
- The language of logic is used here to formalize properties of structures, to determine when a statement is true on a structure
- This use of a logic language is called *model theory*
- Forces a *precise* and *rigorous* definition of the concept of *truth* on a structure

Logic = Syntax + Semantics

- Syntax and semantics of logic are not independent
- A logic language has a syntax, and the meaning of statements by an interpretation on a structure
- The interaction between model theory and proof theory *makes logic an interesting and effective tool*
- **Proof System**
  - Given a logic (syntax and semantics), there can be one or more proof systems, e.g. HOL and PVS are two proof systems based on higher-order logic.

Issues of proof systems

- **Consistency (Soundness)**: all provable formulas (*theorems*) are logically (*semantically*) true
- **Completeness**: all valid formulas (*semantically true*) are provable (*theorems*)
- **Decidability**: there is an algorithm for deciding the (*semantical*) truth of any formula (theorems)

  $\Rightarrow$A proof system is acceptable only if it is consistent (may not be complete nor decidable)

Application of logic to verification

- *Specification* represented as a *formula*
- *Implementation* represented as a *formula* or as a *semantic* model
- **Formula |- Formula:**
  - Verification as theorem proving, i.e., relationship (implication or equivalence) between the specification and the implementation is a theorem to be proven.
- **Model |= Formula:**
  - Both theorem proving and model checking can be used
  - *Model checking* deals with the semantic relationship: shows that the implementation is a model for the specification formula (property).

Relation between Spec and Imp

- Imp ≡ Spec: the implementation is *equivalent* to the specification
- Imp→ Spec: the implementation *logically implies* the specification
- Imp |= Spec: the implementation is a *semantic model* in which the specification is true

Formal Verification Methods

FV methods can be categorized in following main groups:

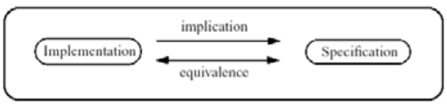
- ***Interactive (deductive) Methods***
  - Theorem Proving: relationship between a specification and an implementation is a theorem in a logic, to be proven within the context of a proof calculus
- ***Automated Methods***
  - Combinational Equivalence Checking: proof of structural equivalence of logic designs
  - Sequential Equivalence Checking: proof of behavioral equivalence of FSMs
  - Model Checking: proof of (temporal) logic property (safety & liveness) against a semantic model of the design
  - Invariant Checking (safety property)
  - Language Containment (model checking of w-automata)

Issues in Verification methods

- **Soundness**: every statement that is provable is actually true.
- **Completeness**: every statement that is actually true is provable.
- **Automation**: proof generation process automatic, semi-automatic or user driven
- Can it handle:
  - ***Compositional proofs***: constructed syntactically from proofs of component parts
  - ***Hierarchical proofs***: for system organized hierarchically at various levels of abstraction
  - ***Inductive proofs***: reason inductively about parameterized descriptions

Theorem Proving

- Prove that an implementation satisfies a specification by mathematical reasoning.



  - Implementation and specification expressed as *formulas* in *a formal logic*.
  - Relationship (logical equivalence/logical implication) described as *a theorem* to be proven.
- **A proof system**:
  - A set of axioms and inference rules (simplification, rewriting, induction, etc.)

Theorem Proving

- **Some known theorem proving systems**
  - Boyer-Moore/ACL2 (first-order logic)
  - HOL (higher-order logic)
  - PVS (higher-order logic)
  - Lambda (higher-order logic)
- **Advantages**
  - High abstraction and powerful logic expressiveness
  - Unrestricted applications
  - Useful for verifying parameterized datapath-dominated circuits
- **Limitations**
  - Interactive (under user guidance)
  - Requires expertise for efficient use
  - Automated for narrow classes of designs

FSM-based Methods

Finite State Machines (FSM)

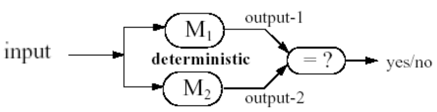
- Well-developed theory for analyzing FSMs (e.g., reachable states, equivalence)
- An FSM (I, O, S, δ, λ, S0)
  - I : input alphabet,
  - O: output alphabet,
  - S: set of states,
  - δ: next-state relation, $\delta \subseteq S \times I \times S$,
  - λ: output relation, $\lambda \subseteq S \times I \times O$ (Mealy), $\lambda \subseteq S \times O$ (Moore)
  - S0: set of initial states.
- Deterministic machines: δ: S×I -> S and λ: S×I –> O are functions, S0 = {s0}.

FSM Equivalence Verification

- Basic method:
  - If same state variables — *Combinational Equivalence* of δ and λ
  - If state space different - *State Enumeration* by *Reachability Analysis*
    Two FSMs are equivalent if they produce the same output for every possible input sequence — *Sequential Equivalence Checking*

Equivalence Checking

- **Equivalence by reachability analysis of the Product Machine**



Reachability Analysis

Start from initial state

repeat

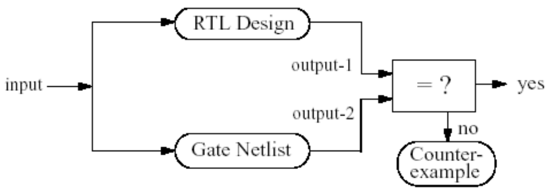
Apply transition relation to determine next state

In each reached state, check equivalence of corresponding outputs of M1, M2

until all reachable states visited

- Involves building a *state transition graph (Finite Kripke structure)*
- Problem: “State explosion”e.g., 32-bit register → 2^{32} states
- Partial solution: Implicit State Enumeration with
  - Reduced Ordered Binary Decision Diagrams (ROBDD)
- Represent transition/output relations and sets of states symbolically using ROBDD

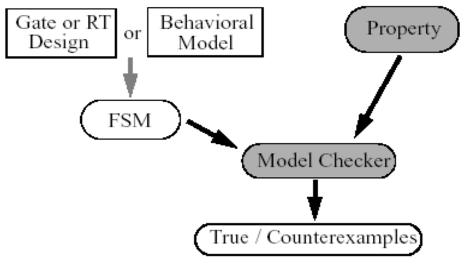
Equivalence Checking: Application example



Equivalence Checking

- **Combinational equivalence**:
  - possible if one-to-one state mapping do exit
  - relatively straightforward (equivalence of sets of functions (BDDs))
  - tools already part of verification flow
- **Sequential equivalence**:
  - no state mapping required (building of product machine)
  - hard to handle large circuits (also must consider all initial states)
  - no tools for industrial use

Model Checking - Basic idea



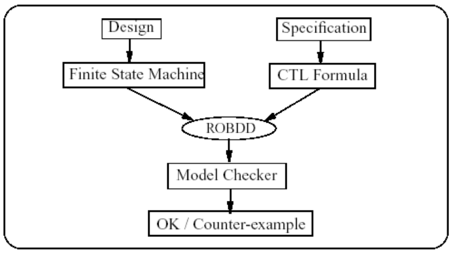
Model Checking

- Property described by temporal logic formula.
- System modeled by Labeled Transition Graph (LTG, LTS, *Finite Kripke structure).*
- *Exhaustive* search through the state space of the system (*Reachability Analysis*) to determine if the property holds (provides counterexamples for identifying design errors).
- Problem: “State explosion”
- Partial Solution: Symbolic Model Checking
- Represent transition/output relations and sets of states symbolically using ROBDD

Binary Decision Diagrams

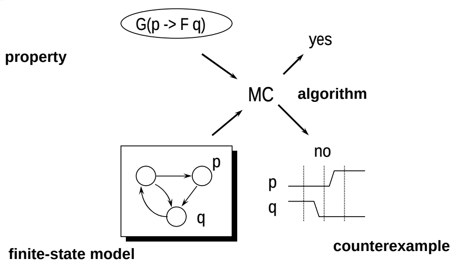
- Idea from 70s (maybe earlier)
- Adapted by Bryant ’86
- Take a formula
- Make decision tree for fixed variable order
- Reduction rules
  - merge duplicate nodes
  - both children point to same node -- remove redundant node

Symbolic Model Checking - Basic idea



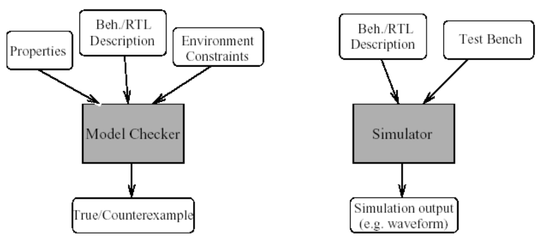
- Problem: again “State explosion” (max ~ 400 Boolean variables), low abstraction level.

Model Checking



(Ken McMillan)

Model Checking vs. Simulation



Symbolic simulation

- Constants 1 0
- unknown X
- symbolic values a,b,c…

- Adapt logic simulation to represent values on wires
- BDDs represent functions of symbolic values

Symbolic simulation

- X halves # simulation runs but loses info.

- a halves # runs but makes BDDs bigger

- Tradeoff

Theorem Proving vs. Model Checking

Theorem Proving: useful for architectural design and verification

- Process:
  - Implementation description: Formal logic
  - Specification description: Formal logic
  - Correctness: |- *Imp* ⇒ *Spec* (implication) or |- *Imp*⇔*Spec* (equivalence)
- High abstraction level possible, expressive notation, powerful logic and reasoning
- Interactive and deep understanding of design and higher-order logic required
- Need to develop rules (lemmas) and tactics for class of designs
- Need a refinement method to synthesizable VHDL / Verilog

Theorem Proving vs. Model Checking

Model Checking: at RT-level (or below) with at most ~400 Boolean state variables

- Process:
  - Implementation description: Model as FSM
  - Specification description: Properties in temporal logic
  - Correctness: *Impl Spec* (property holds in the FSM model)
- Easy to learn and apply (completely automatic), properties must be carefully prepared
- Integrated with design process, refinement from skeletal model
- State space explosion problem (not scalable to large circuits)
- Increase confidence, better verification coverage

Formal Verification Tools

Supplier	Tool Name	Class of Tool	HDL	Design Level
COMMERCIAL TOOLS				
Chrystalis	Design Verifier	Equiv. Checking	VHDL/Verilog	RTL/Gate
Synopsys	Formality	Equiv. Checking	VHDL/Verilog	RTL/Gate
Cadence	Affirma	Equiv. Checking	VHDL/Verilog	RTL/Gate
Compass	VFormal	Equiv. Checking	VHDL/Verilog	RTL/Gate
Verysys	Tornado	Equiv. Checking	VHDL/Verilog	RTL/Gate
Abstract Hardware Ltd.	Checkoff-E	Equiv. Checking	VHDL/Verilog	RTL/Gate
IBM	BooleEye	Equiv. Checking	VHDL/Verilog	RTL/Gate
Cadence	FormalCheck	Model Checking	VHDL/Verilog	RTL
Abstract Hardware Ltd.	Checkoff-M	Model Checking	VHDL/Verilog	RTL/Gate
IBM	RuleBase	Model Checking	VHDL	RTL
Abstract Hardware Ltd.	Lambda	Theorem Proving	VHDL/Verilog	RTL/Gate
PUBLIC DOMAIN TOOLS				
CMU	SMV	Model Checking	own language	RTL
Cadence	Cadence SMV	Model Checking	Verilog	RTL
UC Berkeley	VIS	Model/Equ. Check.	Verilog	RTL/Gate
Stanford U.	Murphy	Model Checking	own language	RTL
Cambridge U.	HOL	Theorem Proving	(SML)	universal
SRI	PVS	Theorem Proving	(LISP)	universal
UT Austin/CLI	ACL2	Theorem Proving	(LISP)	universal

Design Flow and Formal Verification

RT level

- ⇒ Simulation of RTL
  - (+) efficient for less interacting concurrent components
  - (−) incomplete for complicated control parts and difficult error trace
- ⇒ Model checking of RTL
  - (+) efficient for complicated interacting concurrent components
  - (+) counter-examples can trace design errors

Design Flow and Formal Verification

Netlist (Gate level)

- ⇒ Equivalence checking of netlist vs. RTL
  - (+) check the equivalence of submodules to ensure the correctness of synthesis
  - (+) trace synthesis errors using counter-examples
- ⇒ Model checking of netlist
  - (+) correctness of the entire gate-level implementation
  - (–) unpractical: state space explosion