

Introduction to Formal Verification

Gianpiero Cabodi
Politecnico di Torino

2007/2008

04_Formal Verification

Outline

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

2007/2008

04_Formal Verification

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

2007/2008

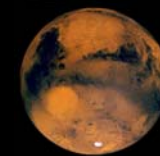
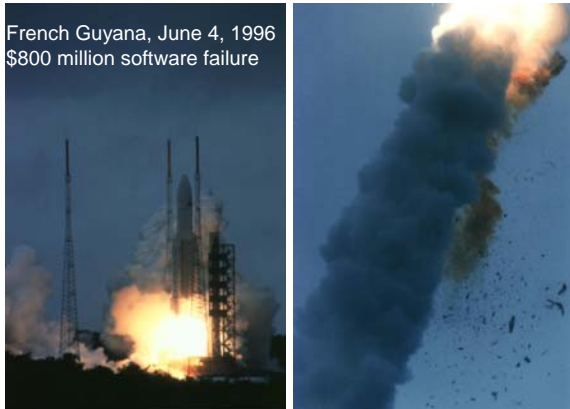
04_Formal Verification

- 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

2007/2008

04_Formal Verification

French Guyana, June 4, 1996
\$800 million software failure



Mars, December 3, 1999
Crashed due to uninitialized variable



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...

2007/2008 04_Formal Verification

The main point is NOT

- correctness proof of entire systems
- replacing test entirely

2007/2008 04_Formal Verification

BUT

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

2007/2008 04_Formal Verification

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

2007/2008 04_Formal Verification

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 the question "are we building the right system?"
- **Verification**: answers the question "are we building the system right?"

2007/2008 04_Formal Verification

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

2007/2008

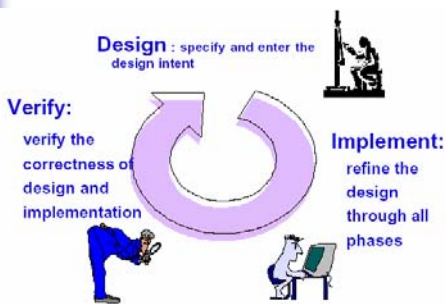
04_Formal Verification

- **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?

2007/2008

04_Formal Verification

Design Process



2007/2008

04_Formal Verification

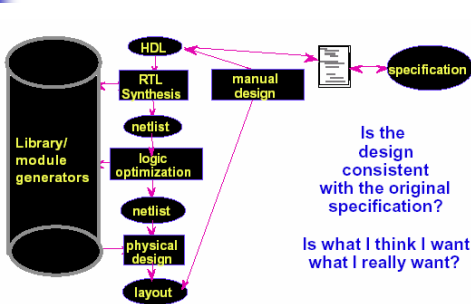
Verification

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

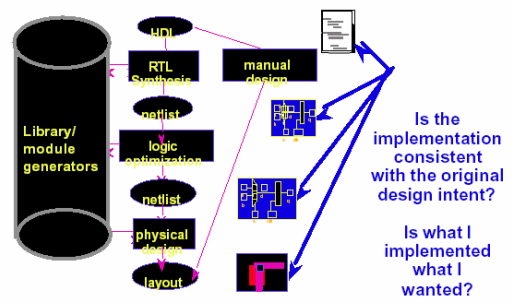
2007/2008

04_Formal Verification

Design Verification

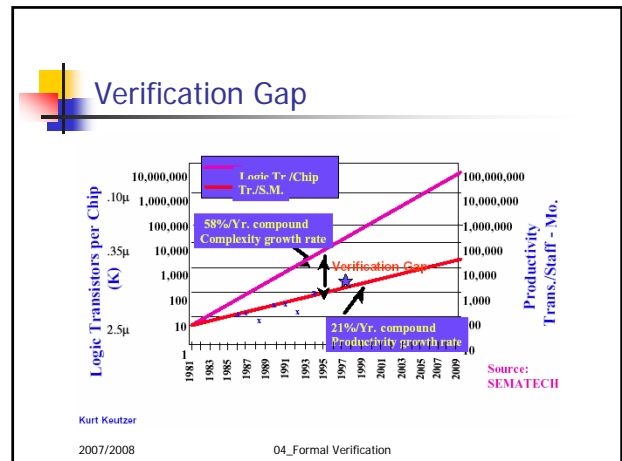
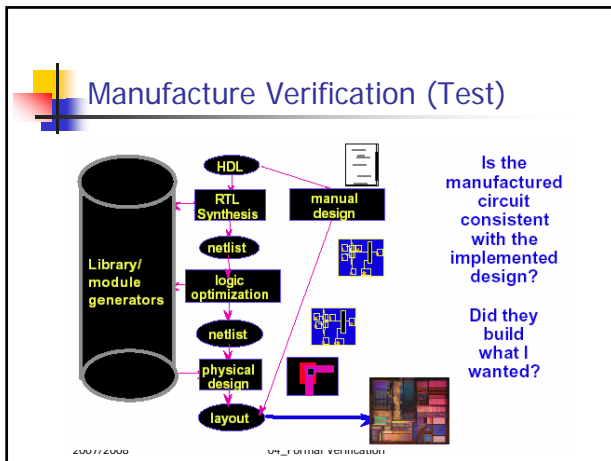


Implementation Verification



2007/2008

04_Formal Verification



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

2007/2008

04_Formal Verification

Filling in reasonable numbers

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

Kurt Keutzer

2007/2008

04_Formal Verification

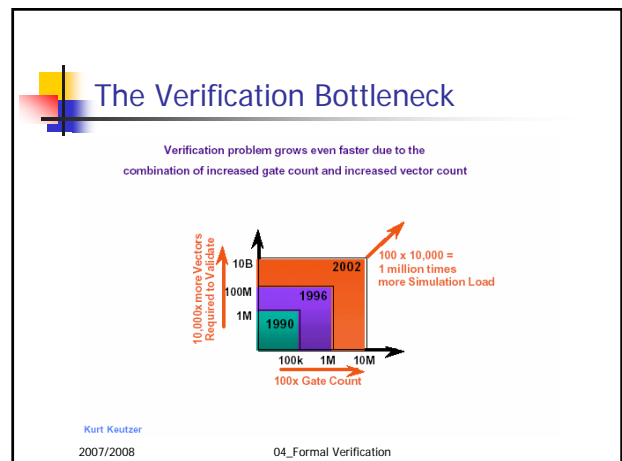
Raising the Level of Abstraction

$$\frac{10,000,000 \text{ trs}}{\text{chip}} \times \frac{1}{100} \times \frac{1}{10,000} = \frac{10 \text{ bugs}}{\text{chip}} \text{ this year!!}$$

Kurt Keutzer

2007/2008

04_Formal Verification



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

2007/2008

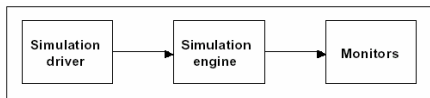
04_Formal Verification

- 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

2007/2008

04_Formal Verification

Simulation: The Current Picture



2007/2008

04_Formal Verification

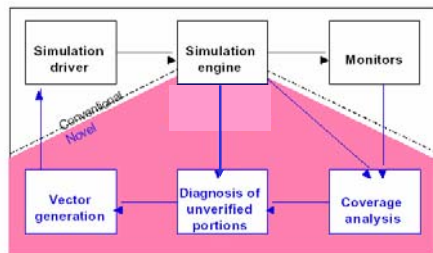
SHORTCOMINGS:

- 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.

2007/2008

04_Formal Verification

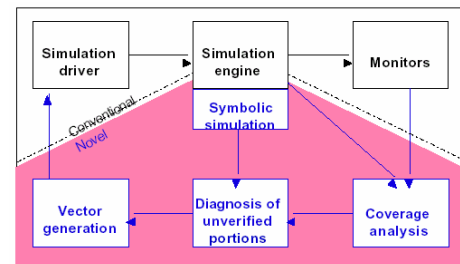
Simulation-based Verification



2007/2008

04_Formal Verification

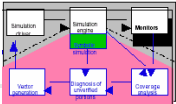
Symbolic Simulation



2007/2008

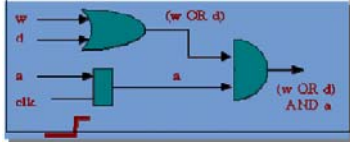
04_Formal Verification

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



2007/2008 04_Forma Verification

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*.

2007/2008 04_Forma Verification

- 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

2007/2008 04_Forma Verification

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.

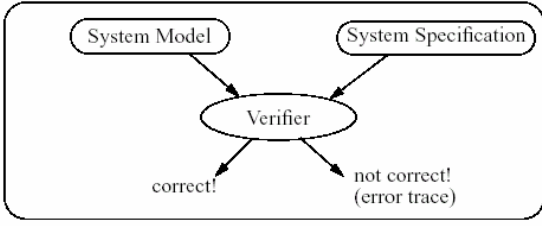
- Use of *mathematical reasoning* to prove that an implementation satisfies a specification
- Like a mathematical proof: correctness of a formally verified hardware design holds *regardless of input values*.

2007/2008 04_Forma Verification

- 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).

2007/2008 04_Forma Verification

Formal Verification



2007/2008 04_Forma 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

2007/2008 04_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
...

2007/2008 04_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.

2007/2008 04_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!

2007/2008 04_Formal Verification

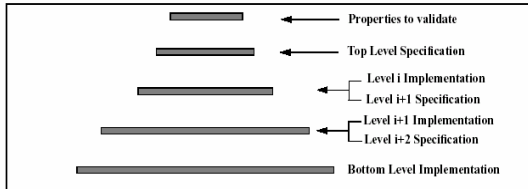
- Techniques are complementary — formal verification gives additional confidence, e.g.,
 - Apply formal verification of abstract model
 - Obtain error trace if bug found (may be false negative!)
 - Simulate error trace on the real model

2007/2008 04_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]

2007/2008 04_Formal Verification

Hierarchical Verification



2007/2008

04_Forma 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.

2007/2008

04_Forma Verification

■ 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.

2007/2008

04_Forma Verification

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.

2007/2008

04_Forma Verification

■ *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

2007/2008

04_Forma Verification

■ 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+)

2007/2008

04_Forma Verification

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)

2007/2008

04_Formal Verification

Formalisms for representing specifications

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

2007/2008

04_Formal Verification

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

2007/2008

04_Formal Verification

- *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)

2007/2008

04_Formal Verification

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:
 $out = (in_1 + in_2) \bmod 2$

2007/2008

04_Formal Verification

- 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?

2007/2008

04_Formal Verification

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)

2007/2008

04_Formal Verification

- 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

2007/2008

04_Formal Verification

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

2007/2008

04_Formal Verification

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, Verisys, IBM,*

2007/2008

04_Formal Verification

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

2007/2008


04_Formal Verification

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.

2007/2008


04_Formal Verification



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.


2007/2008 04_Formal Verification



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


2007/2008 04_Formal Verification



Propositional logic	→	First-order logic	→	Higher-order logic
Less expressive (-)	-----→		-----→	Very expressive (+)
Decidable (+)	-----→		-----→	Undecidable (-)
Complete (+)	-----→		-----→	Incomplete (-)

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


2007/2008 04_Formal Verification



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

2007/2008 04_Formal Verification



- Purely syntactic (mechanical) activity; not concerned with the meaning of statements, but with the arrangement of these statements, and whether proofs can be constructed

2007/2008 04_Formal Verification



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

2007/2008 04_Formal Verification

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*

2007/2008

04_Formal Verification

■ 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.

2007/2008

04_Formal Verification

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 (*semantica*) truth of any formula (*theorems*)
⇒ A proof system is acceptable only if it is consistent (may not be complete nor decidable)

2007/2008

04_Formal Verification

Application of logic to verification

- *Specification* represented as a *formula*
- *Implementation* represented as a *formula* or as a *semantic model*
- **Formula** \models **Formula:**
 - Verification as theorem proving, i.e., relationship (implication or equivalence) between the specification and the implementation is a theorem to be proven.

2007/2008

04_Formal Verification

■ Model \models 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).

2007/2008

04_Formal Verification

Relation between Spec and Imp

- $\text{Imp} \equiv \text{Spec}$: the implementation is *equivalent* to the specification
- $\text{Imp} \rightarrow \text{Spec}$: the implementation *logically implies* the specification
- $\text{Imp} \models \text{Spec}$: the implementation is a *semantic model* in which the specification is true

2007/2008

04_Formal Verification

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

2007/2008

04_Formal Verification

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)

2007/2008

04_Formal Verification

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

2007/2008

04_Formal Verification

Can it handle:

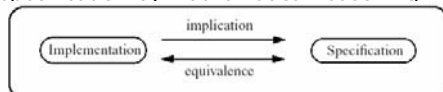
- **Compositional proofs**: constructed syntactically from proofs of component parts?
- **Hierarchical proofs**: for system organized hierarchically at various levels of abstraction?
- **Inductive proofs**: reason inductively about parameterized descriptions?

2007/2008

04_Formal Verification

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.

2007/2008

04_Formal Verification

A proof system:

- A set of axioms and inference rules (simplification, rewriting, induction, etc.)

2007/2008

04_Formal Verification

- 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

2007/2008 04_Formal Verification

- Limitations
 - Interactive (under user guidance)
 - Requires expertise for efficient use
 - Automated for narrow classes of designs

2007/2008 04_Formal Verification

FSM-based Methods

Finite State Machines (FSM)

- Well-developed theory for analyzing FSMs (e.g., reachable states, equivalence)
- An FSM $(I, O, S, \delta, \lambda, S_0)$
 - 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)
 - S_0 : set of initial states.

2007/2008 04_Formal Verification

- Deterministic machines: $\delta: S \times I \rightarrow S$ and $\lambda: S \times I \rightarrow O$ are functions, $S_0 = \{s_0\}$.

2007/2008 04_Formal Verification

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*

2007/2008 04_Formal Verification

Equivalence Checking

- Equivalence by reachability analysis of the Product Machine

2007/2008 04_Formal Verification

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*)

2007/2008

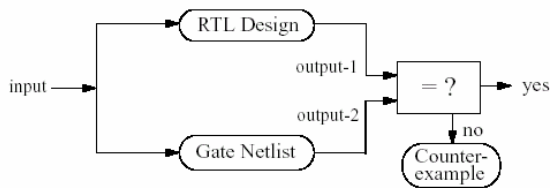
04_Formal Verification

- 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

2007/2008

04_Formal Verification

Equivalence Checking: Application example



2007/2008

04_Formal Verification

Types of Equivalence Checking

- **Combinational equivalence:**
 - possible if one-to-one state mapping to exit
 - relatively straightforward (equivalence of sets of functions (BDDs))
 - tools already part of verification flow

2007/2008

04_Formal Verification

- **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

2007/2008

04_Formal Verification

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).

2007/2008

04_Formal Verification

- Problem: "State explosion"
- Partial Solution: Symbolic Model Checking
- Represent transition/output relations and sets of states symbolically using ROBDD

2007/2008 04_Formal Verification

Model Checking - Basic idea

```

    graph TD
      A[Gate or RT Design] -- or --> B[Behavioral Model]
      B --> C((FSM))
      D(Property) --> E(Model Checker)
      C --> E
      E --> F(True / Counterexamples)
  
```

2007/2008 04_Formal Verification

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

2007/2008 04_Formal Verification

Symbolic Model Checking - Basic idea

```

    graph TD
      A[Design] --> B[Finite State Machine]
      C[Specification] --> D[CTL Formula]
      B --> E((ROBDD))
      D --> E
      E --> F[Model Checker]
      F --> G[OK / Counter-example]
  
```

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

2007/2008 04_Formal Verification

Model Checking

property: $G(p \rightarrow F q)$

MC algorithm

finite-state model

counterexample

(Ken McMillan)

2007/2008 04_Formal Verification

Model Checking vs. Simulation

```

    graph TD
      subgraph Model_Checking
        A[Properties] --> C[Model Checker]
        B[Beh./RTL Description] --> C
        D[Environment Constraints] --> C
        C --> E[True/Counterexamples]
      end
      subgraph Simulation
        F[Beh./RTL Description] --> G[Simulator]
        H[Test Bench] --> G
        G --> I[Simulation output e.g. waveform]
      end
  
```

2007/2008 04_Formal Verification

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

2007/2008

04_Formal Verification

- X halves # simulation runs but loses info.
- a halves # runs but makes BDDs bigger
- Tradeoff

2007/2008

04_Formal Verification

Theorem Proving vs. Model Checking

Theorem Proving: useful for architectural design and verification

- Process:
 - Implementation description: Formal logic
 - Specification description: Formal logic
 - Correctness: $\vdash Imp \Rightarrow Spec$ (implication) or $\vdash Imp \Leftrightarrow Spec$ (equivalence)
- High abstraction level possible, expressive notation, powerful logic and reasoning

2007/2008

04_Formal Verification

- 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

2007/2008

04_Formal Verification

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: $Impl \Rightarrow Spec$ (property holds in the FSM model)

2007/2008

04_Formal Verification

- 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

2007/2008

04_Formal Verification

Formal Verification Tools				
Supplier	Tool Name	Class of Tool	HDL	Design Level
COMMERCIAL TOOLS				
Chrysalis	Design Verifier	Equiv. Checking	VHDL/Verilog	RTL/Gate
Synopsys	Formality	Equiv. Checking	VHDL/Verilog	RTL/Gate
Cadence	Alfama	Equiv. Checking	VHDL/Verilog	RTL/Gate
Compass	VFormal	Equiv. Checking	VHDL/Verilog	RTL/Gate
Versys	Tornado	Equiv. Checking	VHDL/Verilog	RTL/Gate
Abstract Hardware Ltd.	Checkoff-L	Equiv. Checking	VHDL/Verilog	RTL/Gate
IBM	BondEye	Equiv. Checking	VHDL/Verilog	RTL/Gate
Cadence	Formality-check	Model Checking	VHDL/Verilog	RTL
Abstract Hardware Ltd.	Checkoff-M	Model Checking	VHDL/Verilog	RTL/Gate
IBM	RouteBase	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 C1.1	ACL2	Theorem Proving	(LISP)	universal

2007/2008 04_Formal Verification

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

2007/2008 04_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

2007/2008 04_Formal Verification

Model Checking

2007/2008 04_Formal Verification

Outline

- Propositional Logic
- First-Order-Logic
- Temporal Logic (LTL)
- Temporal Logic (BTTL-CTL)
- Model Checking

2007/2008 04_Formal Verification

Propositional Logic (calculus)

Syntax

P, Q, R,... — propositional symbols (atomic propositions)
 t: true; f: false — constants

¬P: not P P ∧ Q: P and Q P ∨ Q: P or Q;
 P ⇒ Q: if P then Q (proposition equivalent to ¬P ∨ Q)
 P ⇔ Q: P if and only if Q, i.e., P equivalent to Q
 (proposition equivalent to (P ∧ Q) ∨ (¬P ∧ ¬Q))

2007/2008 04_Formal Verification

Semantics
Given through the Truth Table:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
t	t	f	t	t	t	t
t	f	f	f	t	f	f
f	t	t	f	t	t	f
f	f	t	f	f	t	t

An **interpretation** is a function from the propositional symbols to $\{t, f\}$

2007/2008 04_Formal Verification

- Formula F is **satisfiable** (consistent) iff it is **true** under **at least one** interpretation
- Formula F is **unsatisfiable** (inconsistent) iff it is **false** under **all** interpretations
- Formula F is **valid** iff it is **true** (consistent) under **all** interpretations
- Interpretation I satisfies a formula F (I is a **model** of F) iff F is true under I .
Notation: $I \models F$
- Theorem:** A formula F is valid (a **tautology**) iff $\neg F$ is unsatisfiable. Notation: $\models F$

2007/2008 04_Formal Verification

The relationship between F to $\neg F$ can be visualized by "mirror principle":

All formulas in propositional logic

Valid formulas	Satisfiable, but non-valid formulas	Unsatisfiable formulas
G	F \leftrightarrow $\neg F$	$\neg G$

To determine if F is satisfiable or valid, test finite number (2^n) of interpretations of the n atomic propositions occurring in F ... but it is an exponential method... **satisfiability is an NP-complete problem**

2007/2008 04_Formal Verification

First-Order Logic (Predicate Calculus)

- First-Order Logic speaks about objects, which are the domain of discourse or the universe.
- First-Order Logic is also concerned about Properties of these objects (called Predicates), and the Names of these objects.
- Also we have Functions of objects and Relations over objects. (For example, Socrates' father is a function of Socrates, while Socrates' son(s) is a relation about the object Socrates). (Properties would be then mapped to relations on objects).

2007/2008 04_Formal Verification

Syntax of First-Order Logic


- Using functions and relations, and using the notation x_i to denote a variable (of type Object) to name individuals (so that we have a set of Vars = $\{x_1, x_2, \dots, x_n\}$) we could define the syntax of formulas in First-Order Logic.

2007/2008 04_Formal Verification

Terms & Formulas


- Terms of First-Order Logic formulas are defined recursively as follows:
 - Vars \subseteq Terms
 - If $t_1, t_2, \dots, t_k \in$ Terms and f is a k -ary function name, then $f(t_1, t_2, \dots, t_k) \in$ Terms

2007/2008 04_Formal Verification



- Formulas of First-Order Logic could be defined as:
 - If $t_1, t_2, \dots, t_k \in \text{Terms}$ and P is a k -ary relation name
 - then $P(t_1, t_2, \dots, t_k)$ is an atomic formula
 - If θ, ψ are formulas then $(\neg\theta)$, $(\theta \wedge \psi)$ are formulas.
 - If θ is a formula and $x \in \text{Vars}$ then $(\exists x)\theta$ and $(\forall x)\theta$ are formulas.

2007/2008 04_FormaI Verification




An Example

$$((\forall x(H(x) \rightarrow M(x)) \wedge (\exists x)(G(x) \wedge H(x))) \rightarrow (\exists x)(G(x) \wedge M(x)))$$

If all humans are mortal and some Greeks are human then some Greeks are mortal.


2007/2008 04_FormaI Verification



Hierarchy of Logic

- First-Order logic is concerned about objects
 - logic quantifiers (\forall, \exists) quantify over elements (objects).
- Second-order logic: elementary elements are functions and relations (i.e., sets of objects)
- Third-order logic: main objects are sets of sets of objects.
 - logic quantifiers (\forall, \exists) quantify over relations and functions.


2007/2008 04_FormaI Verification



Higher Order Logic


- Example 1:** (mathematical induction)
 - $\forall P. [P(0) \wedge (\forall n. P(n) \rightarrow P(n+1))] \rightarrow \forall n. P(n)$
(Impossible to express it in FOL)
- Example 2:** Function Rise defined as
 - Rise $(c, t) = \neg c(t) \wedge c(t+1)$
 - Rise expresses the notion that a signal c rises at time t .
 - Signal is modeled by a function $c: \mathbb{N} \rightarrow \{F, T\}$, passed as argument to Rise.
 - Result of applying Rise to c is a function: $\mathbb{N} \rightarrow \{F, T\}$.

2007/2008 04_FormaI Verification



- Advantage:** high expressive power!
- Disadvantages:**
 - Incompleteness of a sound proof system for most higher-order logics
 - Theorem** (Gödel, 1931) *There is no complete deduction system for the second-order logic.*

2007/2008 04_FormaI Verification



- Reasoning more difficult than in FOL, need ingenious inference rules and heuristics.
- Inconsistencies can arise in higher-order systems if semantics not carefully defined
 - "Russell's Paradox":** Let P be defined by $P(Q) = \neg Q(Q)$. By substituting P for Q , leads to $P(P) = \neg P(P)$,
($P: \text{bool} \rightarrow \text{bool}, Q: \text{bool} \rightarrow \text{bool}$) contradiction!

2007/2008 04_FormaI Verification

Temporal Logic

- Temporal logic is a type of modal logic that was originally developed by philosophers to study different *modes* of "truth"
- Temporal logic provides a formal system for qualitatively describing and reasoning about how the truth values of assertions change *over time*
- It is appropriate for describing the time-varying behavior of systems (or programs)

2007/2008 04_Formal Verification

Classification of Temporal Logic

- The underlying nature of time:
 - Linear:** at any time there is only one possible future moment, linear behavioral trace
 - Branching:** at any time, there are different possible futures, tree-like trace structure
- Other considerations:
 - Propositional vs. first-order
 - Point vs. intervals
 - Discrete time vs. continuous time
 - Past vs. future

2007/2008 04_Formal Verification

Linear Temporal Logic

- Time lines
 - Underlying structure of time is a totally ordered set $(S, <)$, isomorphic to $(\mathbb{N}, <)$: Discrete, an initial moment without predecessors, infinite into the future.
- Let AP be set of atomic propositions, a *linear time structure* $M = (S, x, L)$
 - S : a set of states
 - $x: \mathbb{N} \rightarrow S$ an infinite sequence of states, $(x = s_0, s_1, \dots)$
 - $L: S \rightarrow 2^{AP}$ labeling each state with the set of atomic propositions in AP true at the state.

2007/2008 04_Formal Verification

Example

X: $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$

$p \quad p \ q \quad r \quad u \ v$

- AP = {p, q, r, u, v}
- $L(s_0) = \{p\}$, $L(s_1) = \{p, q\}$, $L(s_2) = \{r\}$, $L(s_3) = \{u, v\}, \dots$

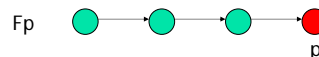
2007/2008 04_Formal Verification

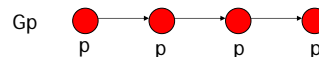
Propositional Linear Temporal Logic (PLTL)

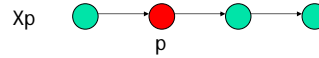
- Classical propositional logic + temporal operators
- Basic temporal operators**
 - Fp ("eventually p ", "sometime p ")
 - Gp ("always p ", "henceforth p ")
 - Xp ("next time p ")
 - pUq (" p until q ")
- Other common notation:
 - $G = \square$
 - $F = \diamond$
 - $X = O$

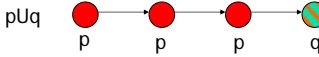
2007/2008 04_Formal Verification

Examples

Fp 

Gp 

Xp 

pUq 

2007/2008 04_Formal Verification

PLTL Syntax

- The set of formulas of PLTL is the least set of formulas generated by the following rules:
 - Atomic propositions are formulas,
 - p and q formulas: $p \wedge q$, $\neg p$, $p \cup q$, and Xp are formulas.
- The other formulas can be introduced as abbreviations:
 - $p \vee q$ abbreviates $\neg(\neg p \wedge \neg q)$

2007/2008

04_FormaI Verification

- $p \Rightarrow q$ abbreviates $\neg p \vee q$
- $p \equiv q$ abbreviates $(p \Rightarrow q) \wedge (q \Rightarrow p)$,
- *true* abbreviates $p \vee \neg p$,
- *false* abbreviates $\neg \text{true}$,
- Fp abbreviates $(\text{true} \cup p)$,
- Gp abbreviates $\neg F\neg p$.

2007/2008

04_FormaI Verification

Examples

- $p \Rightarrow Fq$: "if p is true now then at some future moment q will be true."
- $G(p \Rightarrow Fq)$: "whenever p is true, q will be true at some subsequent moment."

2007/2008

04_FormaI Verification

PLTL semantics

Semantics of a formula p of PLTL with respect to a linear-time structure $M=(S, x, L)$

- $(M, x) \models p$ means that "in structure M , formula p is true of timeline x ."
- x^i : suffix of x starting at s_i , $x^i = s_i, s_{i+1}, \dots$

2007/2008

04_FormaI Verification

Semantics

- $(M, x) \models p$ iff $p \in L(s_0)$, for atomic proposition p
- $(M, x) \models p \wedge q$ iff $(M, x) \models p$ and $(M, x) \models q$
- $(M, x) \models \neg p$ iff it is not the case that $(M, x) \models p$
- $(M, x) \models Xp$ iff $x^1 \models p$
- $(M, x) \models Fp$ iff $\exists j. (x^j \models p)$
- $(M, x) \models Gp$ iff $\forall j. (x^j \models p)$
- $(M, x) \models p \cup q$ iff $\exists j. (x^j \models q$ and $\forall k, 0 \leq k < j (x^k \models p)$)

2007/2008

04_FormaI Verification

Duality between linear temporal operators

- $\models G\neg p \equiv \neg Fp$
- $\models F\neg p \equiv \neg Gp$
- $\models X\neg p \equiv \neg Xp$
- PLTL formula p is *satisfiable* iff there exists $M=(S, x, L)$ such that $(M, x) \models p$ (any such structure defines a **model** of p).

2007/2008

04_FormaI Verification

Example

A simple interface protocol, pulses one clock period wide

Env. (USER) sends Ready, Accepted, Validated to System.

2007/2008 04_Formal Verification

Safety property — nothing bad will ever happen:

- $\forall t. (\text{Validated}(t) \Rightarrow \neg \text{Validated}(t + 1))$
- $\square (\text{Validated} \Rightarrow \text{O } \neg \text{Validated})$
- $G (\text{Validated} \Rightarrow X \neg \text{Validated})$

Liveness property — something good will eventually happen:

- $\forall t. \text{Ready}(t) \Rightarrow \exists t' \geq t. \text{Accepted}(t')$
- $\square (\text{Ready} \Rightarrow \diamond \text{Accepted})$
- $G (\text{Ready} \Rightarrow F \text{Accepted})$

2007/2008 04_Formal Verification

Constraints

- **Fairness constraint:**
 - $G(\text{Accepted} \Rightarrow F \text{Ready})$
models a live environment for System
- Behavior of environment (constraint):
 - $G (\text{Ready} \Rightarrow X(\neg \text{Ready} \cup \text{Accepted}))$
- What about other properties of *Accepted* (initial state, periodic behavior), etc.?
 - Prove the system property under the assumption of valid environment constraints

2007/2008 04_Formal Verification

Branching Time Temporal Logic (BTTL)

- **Structure of time: an infinite tree**, each instant may have many successor instants. Along each path in the tree, the corresponding timeline is isomorphic to \mathbf{N}
- **State quantifiers:** Xp, Fp, Gp, pUq (like in linear temporal logic)

2007/2008 04_Formal Verification

- **Path quantifiers:** *for All paths (A) and there Exists a path (E) from a given state*
- Other frequent notation:
 - $G = \square$
 - $F = \diamond$
 - $X = \text{O}$
 - $A = \forall$
 - $E = \exists$

2007/2008 04_Formal Verification

BTTL and LTL

- In linear time logic, temporal operators describe events along a single future, however, when a linear formula is used for specification, there is usually an *implicit universal quantification* over all possible futures (linear traces)

2007/2008 04_Formal Verification

- In contrast, in branching time logic the operators usually reflect the branching nature of time by allowing *explicit quantification* over possible futures in any state
 - One supporting argument for branching time logic is that it offers the ability to reason about *existential* properties in addition to *universal* properties
 - But, it requires some knowledge of internal state for branching, closer to implementation than LTL that describes properties of observable traces and has simpler fairness assumptions

2007/2008 04_Formal Verification

CTL: a BTTL

- CTL = Computation Tree Logic
- Example of Computation Tree:
 - Paths in the tree = possible computations or behaviors of the system

2007/2008 04_Formal Verification

CTL syntax

- Every atomic proposition is a CTL formula
- If f and g are CTL formulas, then so are $\neg f$, $f \wedge g$, AXf , EXf , $A(fUg)$, $E(fUg)$
- Other operators:
 - $AFg = A(\text{true } U \ g)$ $EFg = E(\text{true } U \ g)$
 - $AGf = \neg E(\text{true } U \ \neg f)$ $EGf = \neg A(\text{true } U \ \neg f)$
- EX , $E(\dots U \dots)$, EG are sufficient to characterize the entire logic:
 - $EFp = E(\text{true } U \ p)$
 - $AXp = \neg EX\neg p$ $AGp = \neg EF\neg p$
 - $A(qUp) = \neg(E(\neg p \ U \ q) \wedge \neg p) \vee EG\neg p$

2007/2008 04_Formal Verification

Intuitive Semantics of Temporal Operators

2007/2008 04_Formal Verification

CTL semantics

- A *Kripke structure*: triple $M = \langle S, R, L \rangle$
 - S : set of states $R \subseteq S \times S$: transition relation
 - L : $S \rightarrow 2^{AP}$: (Truth valuation) set of atomic propositions true in each state
- R is *total*:
 - $\forall s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$

2007/2008 04_Formal Verification

- Path* in M :
 - infinite sequence of states, $x = s_0, s_1, \dots, i \geq 0, (s_i, s_{i+1}) \in R$
 - x^i : suffix of x starting at s_i , $x^i = s_i, s_{i+1}, \dots$
 - x_i denotes the suffix of x starting at s_i : $x_i = s_i, s_{i+1}, \dots$

2007/2008 04_Formal Verification

- Truth of a CTL formula is defined inductively:
 - $(M, s_0) \models p$ iff $p \in L(s_0)$, for atomic proposition p
 - $(M, s_0) \models \neg p$ iff $(M, s_0) \not\models p$
 - $(M, s_0) \models p \wedge q$ iff $(M, s_0) \models p$ and $(M, s_0) \models q$
 - $(M, s_0) \models AX\rho$ iff \forall states $t, (s_0, t) \in R, (M, t) \models \rho$
 - $(M, s_0) \models EX\rho$ iff \exists states $t, (s_0, t) \in R, (M, t) \models \rho$
 - $(M, s_0) \models A(p \cup q)$ iff $\forall x = s_0, s_1, s_2, \dots, \exists j \geq 0, (M, s_j) \models q$ and $\forall k, 0 \leq k < j, (M, s_k) \models p$
 - $(M, s_0) \models E(p \cup q)$ iff $\exists x = s_0, s_1, s_2, \dots, \exists j \geq 0, (M, s_j) \models q$ and $\forall k, 0 \leq k < j, (M, s_k) \models p$

2007/2008 04_Formal Verification

Example Structure $M \langle S, R, L \rangle$

- $S = \{1, 2, 3, 4, 5\}, AP = \{a, b, c\}$
- $R = \{(1, 2), (2, 3), (5, 3), (5, 5), (5, 1), (2, 4), (4, 2), (1, 4), (3, 4)\}$
- $L(1) = \{b\}, L(2) = \{a\}, L(3) = \{a, b, c\}, L(4) = \{b, c\}, L(5) = \{c\}$

2007/2008 04_Formal Verification

Examples: CTL formulas

- $EF(started \wedge \neg ready)$: possible to get to a state where *started* holds but *ready* does not
- $AG(req \Rightarrow AF ack)$: if a *request* occurs, then there is eventually an *acknowledgment* (does not ensure that the number of *req* is the same as that of *ack* !)
- $AG(AF enabled)$: *enabled* holds infinitely often on every computation path
- $AG(EF restart)$: from any state it is possible to get to the *restart* state

2007/2008 04_Formal Verification

Model Checking Problem for Temporal Logic

- Given an FSM M (equivalent Kripke structure) and a temporal logic formula p , does M define a model of p ?
 - Determine the truth of a formula with respect to a given (initial) state in M
 - Find all states s of M such that $(M, s) \models p$
- For any **propositional** temporal logic, the model checking problem is **decidable**: exhaustive search of all paths through the finite input structure

2007/2008 04_Formal Verification

Structure of Model Checker

2007/2008 04_Formal Verification

- Specification Language: CTL
- Model of Computation: Finite-state systems modeled by labeled state-transition graphs
 - (*Finite Kripke Structures*)
- If a state is designated as the *initial state*, the structure can be unfolded into an infinite tree with that state as the root: *Computation Tree*

2007/2008 04_Formal Verification

Model Checking Algorithms

- Original algorithm described in terms of *labeling* the CTL structure (Clark83)
 - Required explicit representation of the whole state space
- Better algorithm based on *fixed point* calculations
- Algorithm amenable to *symbolic* formulation
 - Symbolic evaluation allows implicit enumeration of states
 - Significant improvement in maximum size of systems that can be verified

2007/2008

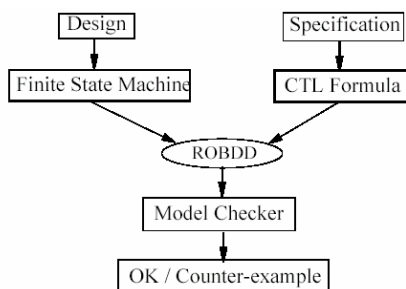
04_Formal Verification

Symbolic Model Checking

- Explicit State Representation. State Explosion Problem (about 10^8 states maximum)
- Breakthrough: Implicit State Representation using ROBDD (about 10^{20} states).
- Use Boolean characteristic functions represented by ROBDDs to encode sets of states and transition relations.

2007/2008

04_Formal Verification



2007/2008

04_Formal Verification

Model Checking Tools

- **SMV (Symbolic Model Verifier)**
 - A tool for checking finite state systems against specifications in the temporal logic CTL.
 - Developed at Carnegie Mellon University by E. Clarke, K. McMillan et. al.
 - Supports a simple input language: SMV
 - For more information: <http://www.cs.cmu.edu/~modelcheck/smv.html>

2007/2008

04_Formal Verification

■ Cadence SMV

- Updated version of SMV by K. McMillan at Berkeley Cadence Labs
- Input languages: extended SMV and synchronous Verilog
- Supports temporal logics CTL and LTL, finite automata, embedded assertions, and refinement specifications.
- Features compositional reasoning, link with a simple theorem prover, an easy-to-use graphical user interface and source level debugging capabilities

2007/2008


04_Formal Verification

■ NuSMV

- Updated version of SMV by Cimatti and Roveri (IRST Trento)
- Input language: extended SMV
- Supports temporal logics CTL and LTL.

2007/2008

04_Formal Verification



- **VIS (Verification Interacting with Synthesis)**
 - A system for formal verification, synthesis, and simulation of finite state systems.
 - Developed jointly at the University of California at Berkeley and the University of Colorado at Boulder.
 - Features:
 - Fast simulation of logic circuits
 - Formal "implementation" verification (equivalence checking) of combinational and sequential circuits
 - Formal "design" verification using fair CTL model checking and language emptiness

2007/2008 04_Formal Verification