Overview on Formal Verification

P.Camurati G.Cabodi S.Nocco S.Quer

Formal Methods Group
Department of Computer Engineering
Politecnico di Torino
Torino, Italy

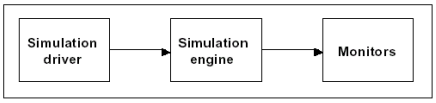
Outline

- State of the art in Formal Verification
- Ingredients in Formal Verification
- Classification

Outline

- State of the art in Formal Verification
- Ingredients in Formal Verification
- Classification

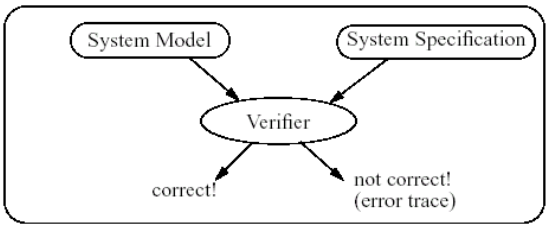
Simulation: The Current Picture



- 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.

Formal Verification: An 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: Pros

- 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

Formal Verification: Cons

- Just because we have proved something correct does not mean it will work!
- Common to other techniques
  - Does the specification actually capture the designer's intentions?
  - Does the implementation in the real world behave like the model?
- Scalabilty (with the size of the design to verify)

Simulation vs. Formal Verification (1/4)

- Example
  - $(x+1)^2 = x2 + 2x + 1$
- Simulation
  - Check Equation for all 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 (2/4)

- 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 (3/4)

- 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 (4/4)

- 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]

Outline

- State of the art in Formal Verification
- Ingredients in Formal Verification
- Classification

State of the Art (1/3)

- 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 (2/3)

- 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 (3/3)

- 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, ...*

Formal Verification Methods

- 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 automata)

Formal Specification (1/2)

- 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 (2/2)

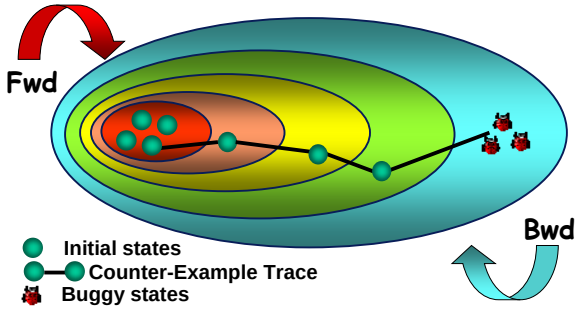
- *Formal specification*. Use of formal methods (a language with mathematically-defined syntax and semantics) to describe the intended behavior of the system:
  - The language of logic provides an unambiguous method of recording the specification
  - We can reason about a formal specification to check 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+)

Types of properties (2/2)

- 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)

Types of properties (2/2)

- Invariant property: Gp, Globally Property p holds, i.e., not(p) is never reached, not (E(not(p)))



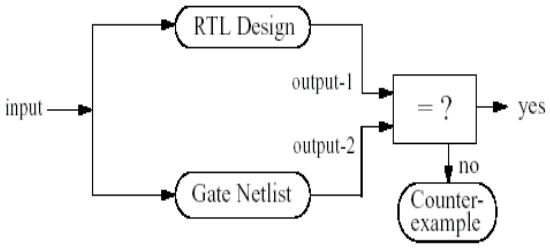
Outline

- State of the art in Formal Verification
- Ingredients in Formal Verification
- Classification

Equivalence Checking (1/3)

- If same state variables
  - *Combinational Equivalence* of δ and λ
  - *Difficulty strongly dependent from the Boolean function representation (canonical vs. non canonical)*
- 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 (2/3)



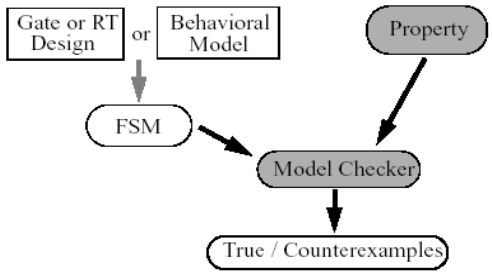
Equivalence Checking (3/3)

- **Combinational equivalence**
  - Possible if one-to-one state mapping do exit
  - Relatively straightforward (equivalence of sets of functions (BDDs))
  - Tools part of verification flow
- **Sequential equivalence**
  - Needs some sort of Reachability Analysis
  - No state mapping required (building of product machine)
  - Hard to handle large circuits (also must consider all initial states) because of the state explosion problem

Model Checking (1/3)

- **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
- Alternative methods based on Satisfiability Solvers

Model Checking (2/3)



Model Checking (3/3)