Digital Systems

Validation, verification

R. Pacalet

Telecom Paris
Institut Mines-Telecom

August 20, 2024

Outline

1 Introduction

2 Formal verification
- Simulation versus formal verification
- Combinational equivalence checking
- Sequential equivalence checking
- Model checking

3 Simulation
- Functional simulation
- Gate level simulation
- Electrical simulation

4 Hardware emulation

5 Conclusion

Outline

1 Introduction

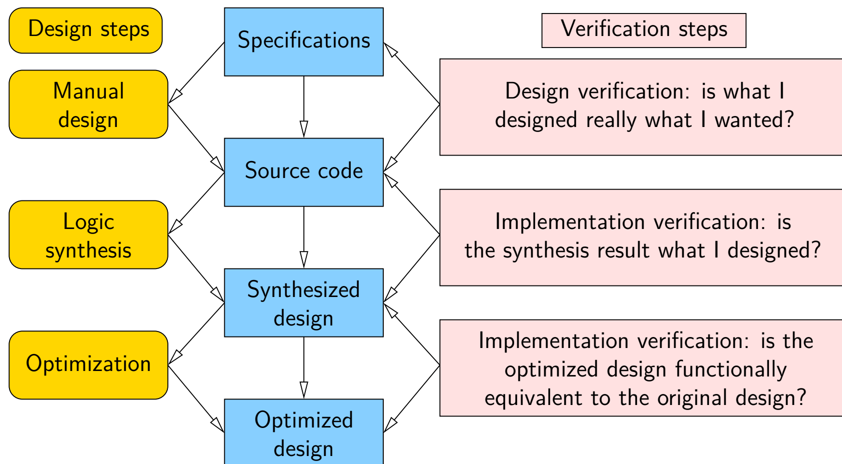
2 Formal verification
- Simulation versus formal verification
- Combinational equivalence checking
- Sequential equivalence checking
- Model checking

3 Simulation
- Functional simulation
- Gate level simulation
- Electrical simulation

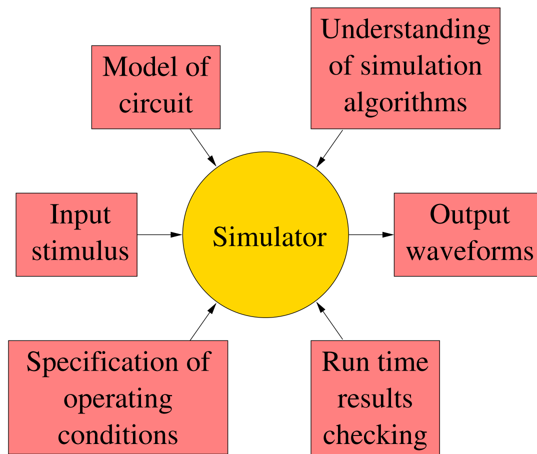
4 Hardware emulation

5 Conclusion

Validating



The simulator



Simulation

- Extra design tasks
  - Reference model
  - Simulation environment
- A simulation cannot be exhaustive
  - Can discover a bug
  - Cannot guarantee correctness
- Simulation before and after synthesis sometimes behave differently
- Usually CPU intensive

Formal verification

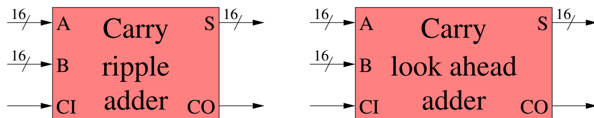
- Supported only for synthesizable models (up to now)
- Three main classes of tools
  - Structural equivalence checking
    - Same memory elements
    - Fast
    - Multi-million gates designs
  - Sequential equivalence checking
    - Functionally equivalent circuits
    - Different memory elements
    - Slow and memory consuming
    - Small designs (a few hundreds of DFFs)
  - Model checking
    - Check a design against temporal-logic properties
    - Slow and memory consuming
    - Small designs (a few hundreds of DFFs)

TELECOM Paris

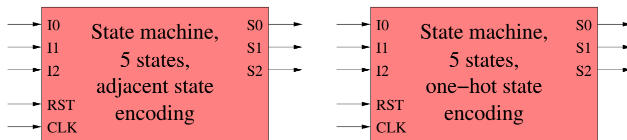
IP PARIS

Formal verification

- These two circuits are structurally equivalent



- These two are not, but they are sequentially equivalent



Formal verification

- Examples of temporal logic properties
  - The two traffic lights are never simultaneously green (safety)

$$AG(\neg((\text{tl1} == \text{green}) \land (\text{tl2} == \text{green})))$$

  - If a client requests the bus it will be finally granted (liveness)

$$AG(\text{request} \Rightarrow (AF(\text{granted})))$$

  - A client holds its request until it's granted the bus

$$AG(\text{request} \Rightarrow (AX(\text{request} \lor \text{granted})))$$

Outline

1. Introduction
2. Formal verification
   - Simulation versus formal verification
   - Combinational equivalence checking
   - Sequential equivalence checking
   - Model checking
3. Simulation
   - Functional simulation
   - Gate level simulation
   - Electrical simulation
4. Hardware emulation
5. Conclusion

Outline

1 Introduction

2 Formal verification
- Simulation versus formal verification
- Combinational equivalence checking
- Sequential equivalence checking
- Model checking

3 Simulation
- Functional simulation
- Gate level simulation
- Electrical simulation

4 Hardware emulation

5 Conclusion

Simulation

- Simulation-based verification
  - Generate input stimulus
    - Random
    - Driven by a given functionality
    - Generated from constraints (Vera, Verisity)
  - Generate expected results (reference model)
  - Simulate with input stimulus
  - Compare simulation output with expected results (offline or online)

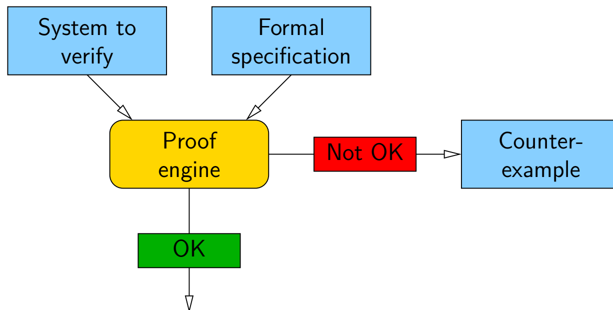
TELECOM Paris

IP PARIS

Drawbacks of simulation

- Good coverage ⇒ billions of cycles ⇒ slow
- Simulation is not exhaustive
  - Simulation gives no guarantee on non-simulated sequences
  - Limited number of use cases are covered
- A reference model is needed to produce the expected results
- Efficient input stimulus are difficult to generate
  - Must have a high coverage ratio
  - Corner cases
  - Bugs usually located where designers were less careful

What is formal verification?



Formal verification

- Exhaustive (for a given property)
  - Result is mathematically guaranteed
- No need to generate expected results (property is a formal model of the expected results)
- Generates a counter-example if property fails
- A very powerful bug-catcher

Formal verification, history

- The theory dates back to the 70's
  - In 1980 systems with about 10^6 states could already be verified
- In 1990 Clarke et al. (CMU) dramatically improve model checking algorithms with SMV
  - 10^{20} states
- Hard critical bugs discovered in real world designs
  - Cache coherency protocol,...
- Today
  - Commercial tools exist
  - Industry is interested
  - Very large states space systems verified

TELECOM Paris

IP PARIS

Simulation versus formal verification

- Simulation
  - Non exhaustive
  - Reference model is needed
  - Corner cases are difficult to cover
- Formal verification
  - Exhaustive (for the given property)
  - No need for a reference model
  - Corner cases automatically covered

Simulation versus formal verification

- Simulation
  - Huge CPU runtime (billions of cycles)
  - Applicable on large designs
- Formal verification
  - Huge memory usage
  - Internal data structures (BDD)
  - Memory needs depends on size (number of states) of the system to verify

TELECOM Paris

IP PARIS

Simulation versus formal verification

- Simulation is a very efficient way to verify quickly, even in the early phases of design
- Formal verification increases confidence
- Both techniques are complementary

Outline

1 Introduction

2 Formal verification
- Simulation versus formal verification
- Combinational equivalence checking
- Sequential equivalence checking
- Model checking

3 Simulation
- Functional simulation
- Gate level simulation
- Electrical simulation

4 Hardware emulation

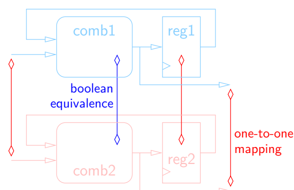
5 Conclusion

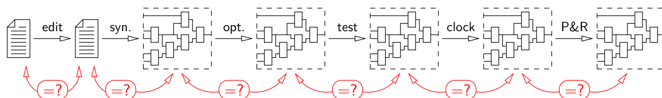
Combinational equivalence checking

- Comparison of 2 designs
  - Synthesizable or synthesized
  - (Almost) same memory elements (re-targeting)
- Fast (several runs/day, multi-million gates designs)
- Exhaustive
- Affordable (tools, learning curve)
- Reference ”golden” model needed

Combinational equivalence checking

- Purely functional (physical characteristics ignored)
- Two designs equivalent iff
  - One-to-one mapping of memory elements and IOs
  - Equivalent boolean functions
- Used to (quickly) validate
  - Minor modifications of synthesizable models
  - Netlist changes



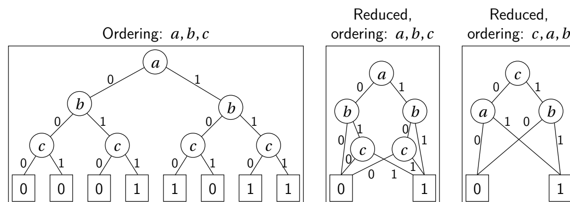


Combinational equivalence checking

- Boolean formula equivalence is co-NP-complete
  - *co-NP*-complete: counterexample verifiable in polynomial-time
  - co-NP-*complete*: Hardest problem in co-NP class
  - Real complexity is unknown ($P \stackrel{?}{=} NP \stackrel{?}{=} co\text{-}NP$)
  - A lot of heuristics are used
- Representation of boolean functions critical
  - Computation time
  - Memory usage

Combinational equivalence checking

- BDDs (Binary Decision Diagrams) are used to represent boolean functions (set of states, state transitions, etc.)

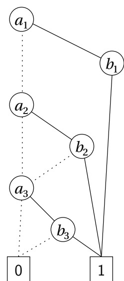


- Note: other techniques (SAT, SMT...) used in recent tools

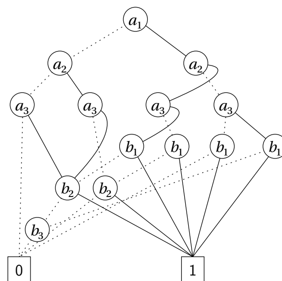
Combinational equivalence checking

- The size of a BDD highly depends on the variable ordering

Function: $a_1 b_1 + a_2 b_2 + a_3 b_3$



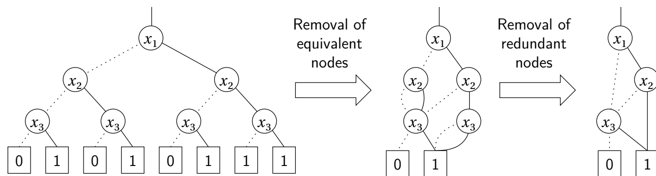
$a_1 < b_1 < a_2 < b_2 < a_3 < b_3$



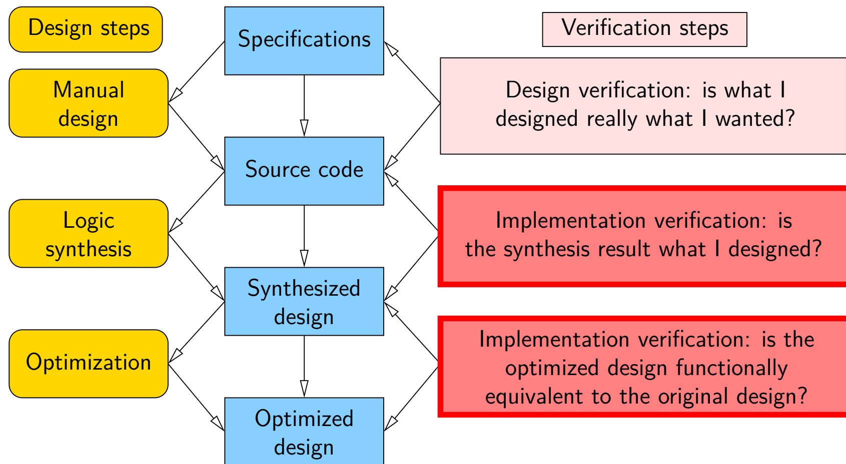
$a_1 < a_2 < a_3 < b_1 < b_2 < b_3$

Combinational equivalence checking

- BDD reduction



Combinational equivalence checking



Combinational equivalence checking

- Commercial tools
  - Synopsys (Formality)
  - Cadence (Conformal)
  - Mentor (FormalPro)
- Warning
  - Synthesizable models are first synthesized (symbolic synthesis)
  - Using same engine for synthesis and equivalence checking unsafe
  - Equivalence of synthesizable model and synthesis result?
  - Check this with your CAD tools vendors

Outline

1 Introduction

2 Formal verification
- Simulation versus formal verification
- Combinational equivalence checking
- Sequential equivalence checking
- Model checking

3 Simulation
- Functional simulation
- Gate level simulation
- Electrical simulation

4 Hardware emulation

5 Conclusion

TELECOM Paris

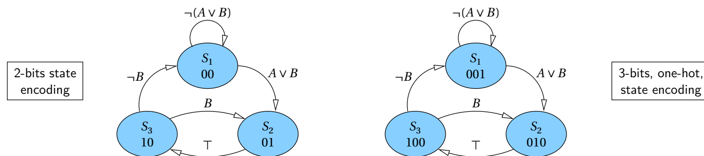
IP PARIS

Sequential equivalence checking

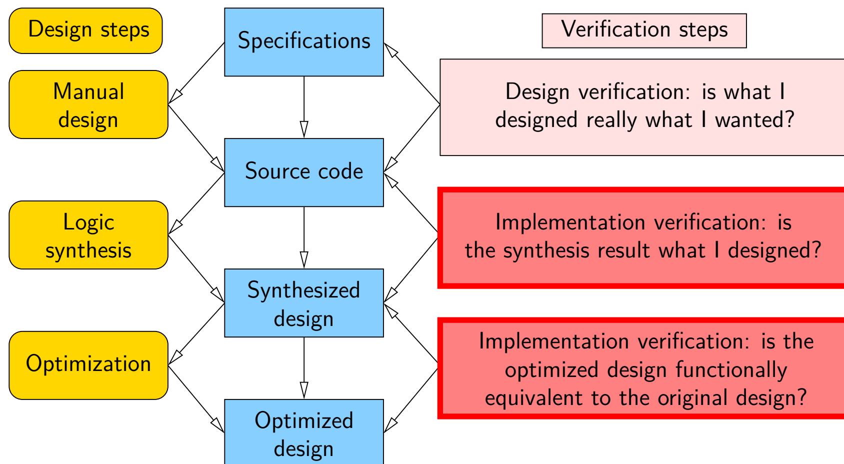
- Comparison of 2 designs
  - Synthesizable or synthesized
  - Without constraints on memory elements
- Slow and memory-hungry (limited to a few hundreds of DFF)
- Exhaustive
- Affordable (tools and learning curve)
- Reference ”golden” model is needed

Sequential equivalence checking

- Purely functional (physical characteristics ignored)
- Two designs equivalent iff
  - Same black-box behaviour (cycle accurate - bit accurate equivalence at interfaces)
- Used to validate major modifications of synthesizable source code
- No (few?) commercial tools
- Example: these 2 designs are sequential-equivalent but not combinational-equivalent



Sequential equivalence checking



Outline

1 Introduction

2 Formal verification
- Simulation versus formal verification
- Combinational equivalence checking
- Sequential equivalence checking
- Model checking

3 Simulation
- Functional simulation
- Gate level simulation
- Electrical simulation

4 Hardware emulation

5 Conclusion

TELECOM Paris

IP PARIS

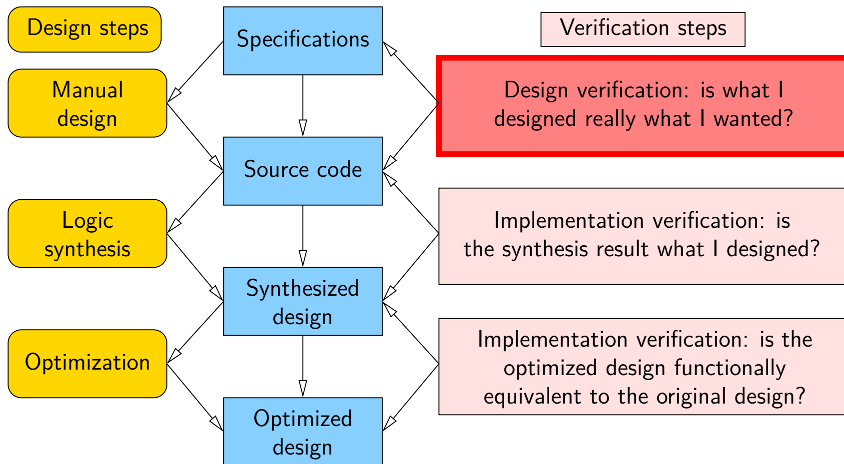
Model checking

- Comparison of one design and temporal-logic properties
  - Synthesizable or synthesized models
  - Without constraints on memory elements
- Exhaustive
- Expensive (especially learning curve)
- Well adapted to verify control parts of a design (”small” state machines) but not to computation parts (state space explosion)

Model checking

- Purely functional (physical characteristics ignored)
- A design passes property iff
  - No counter-example
- Used to validate
  - Functionality of synthesizable model
  - Modifications

Model checking

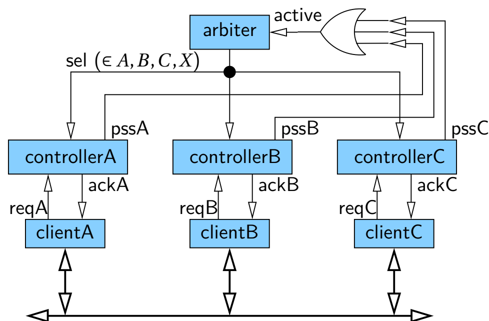


Model checking

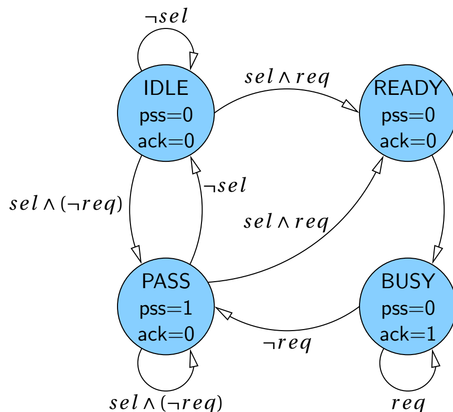
- Commercial tools
  - IBM (RuleBase)
  - Mentor Graphics (0in)
  - Synopsys (Magellan)
  - Cadence (Incisive)
- Academic tools
  - Cadence SMV
  - COSPAN
  - NuSMV
  - UPPAAL
  - Spin
  - VIS (the one we will use in lab)

Model checking

- Our example: bus controller



The controller



Model checking of controller

- Needed
  - HDL model of controller
  - Specification of controller as collection of temporal-logic properties
    - Never this...
    - Always that...
    - If this, then that...
  - Description of environment of controller (arbiter and clients)
    - Client never does this...
    - Arbiter always does that...
  - Model checker
  - Memory, time and coffee

HDL model of controller

```
1  entity controller is
2    port(clk, rstn, req, sel: in boolean;
3         pss, ack: out boolean);
4  end ctrl;
5
6  architecture beh of controller is
7
8    type state_type is (idle, ready, busy, pass);
9    signal state: state_type;
10
11 begin
12
13   process(clk, rstn)
14   begin
15     if not rstn then
16       state <= idle;
17     elsif rising_edge(clk) then
18       case state is
19         when idle =>
20           if sel and req then
21             state <= ready;
22           elsif sel and not req then
23             state <= pass;
24           end if;
25         when ready =>
26           state <= busy;
27         when busy =>
28           if not req then
29             state <= pass;
30           end if;
31         when pass =>
32           if sel and req then
33             state <= ready;
34           elsif not sel then
35             state <= idle;
36           end if;
37       end case;
38     end if;
39   end process;
40
41   ack <= state = busy;
42   pss <= state = pass;
43
44 end beh;
```

Specification of controller

- Controller specified with temporal logic
  - CTL (Computation Tree Logic)
  - LTL (Linear Temporal Logic)
  - CTL* (superset of CTL and LTL)
- Choice of temporal logic determines expressiveness
  - Temporal logics not equivalent

Temporal logics

- Classical boolean logic
  - $\text{true}, \text{false}, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
- Atomic formula
  - $p = ((ack = 1) \wedge (req = 0))$
- Temporal operators characterize execution (path in state tree)
  - X (neXt): $Xp =$ "next state satisfies p"
    - $(STATE = PASS) \Rightarrow X(STATE = IDLE)$
  - F (Futur): $Fp =$ "a future state satisfies p"
    - $(req = 1) \Rightarrow F(ack = 1)$

Temporal logics

- Temporal operators
  - G (Global) : $Gp =$ "all future states satisfy p"
    - $G\neg((ackA = 1) \wedge (ackB = 1))$
  - G and F dual: $Gp \Leftrightarrow \neg F \neg p$
    - $\neg F((ackA = 1) \wedge (ackB = 1))$
  - U (Until): pUq
    - q will be satisfied in future
    - Meanwhile, p remains satisfied
    - $(req = 1) \Rightarrow X((req = 1)U(ack = 1))$
  - W (Weak until): $pWq \Leftrightarrow (pUq) \vee Gp$
    - p remains satisfied until q satisfied
    - Or p satisfied forever (and q may be never satisfied)
    - $(req = 1) \Rightarrow X((req = 1)W(ack = 1))$

Temporal logics

- Temporal operators may be combined
  - $GF = F^{\infty} : GFp =$ "during execution, p satisfied an infinite number of times"
  - $FG = G^{\infty} : FGp =$ "during execution, p satisfied forever starting from given point (or p false finite number of times)"
- Path operators
  - Specify branching behaviour (several potential futures from given situation)
  - Ap: every execution from current state satisfies p (universal path quantifier)

Temporal logics

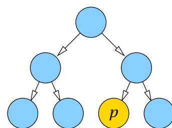
- Path operators
  - Ep: starting from current state at least one execution exists that satisfies p (existential path quantifier)
  - Warning, $A \neq G$ and $E \neq F$
    - A means "every execution"
    - G means "every state" along considered execution
    - E means "there is at least one execution"
    - F means "a future state" along considered execution
    - A and E operate on paths
    - G and F operate on states along given path
- Path and temporal operators usually go by pairs
  - EFp: there exist one execution satisfying p (in future)
  - AFp: for every execution p satisfied (in future)

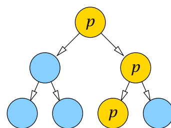
Temporal logics

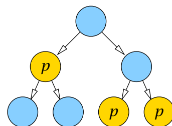
- Path - time pairs
- AGp: p always satisfied, for every execution, for every state (safety)
  - $AG\neg((ackA = 1) \wedge (ackB = 1))$
- EGp: there exist one execution for which p always satisfied
  - $EG\neg(reqA = 1)$
- AXp: starting from current state all next states satisfy p
- EXp: starting from current state there exist one next state satisfying p

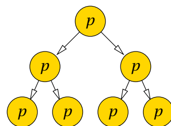
Temporal logics

- A, E, F and G









Temporal logics

- Path operators difficult to understand
  - $AGFp$: along every execution (A), from every state (G), a future (F) state satisfying p is reached. In other words, p will be satisfied an infinite number of times along every execution: $AGFp \Leftrightarrow AF^{\infty}p$
  - $AGEFp$: along every execution (A), from every state (G), it is possible (E) to reach a future (F) state satisfying p. In other words, p is always satisfiable along every execution. $AGEFp$ may be true even if, in one given execution, p is never satisfied
  - Can you write down a formula describing this last property (there exists one execution along which p is never satisfied)?

Temporal logics

- CTL*: no constraint about the way path and temporal operators are mixed
- LTL: CTL* without path operators
  - LTL concerns executions, not the way they are organized along state tree
  - LTL formulas are path formulas (linear time logic)
  - LTL cannot express potentialities: $AGEFp$ ("p always potentially satisfiable") cannot be written in LTL
  - LTL is what designers usually need
  - LTL formulas implicitly preceded by universal path quantifier (A): they characterize all possible executions from initial state

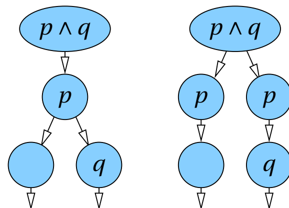
Temporal logics

- CTL: CTL* with the constraint that every temporal operator (X, F, G, U) is preceded by a path operator (A, E)
  - CTL formulas are state formulas
  - CTL formulas depend only on the current state, not on the current execution (path)
  - CTL cannot write F^∞
- FCTL: extension of CTL (F for Fair) allowing to express F^∞
- The model checking on CTL or FCTL is more efficient than on LTL...
- ... But designers prefer LTL

Exercise #1: temporal logics

- LTL cannot distinguish these two situations
- CTL can

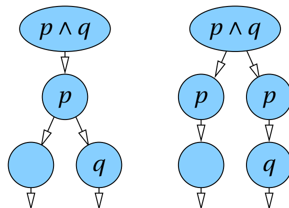
? Try to write a CTL formula stating that the choice between $\neg q \wedge \neg p$ and $q \wedge \neg p$ remains open a bit longer in one case than in the other



Exercise #1: temporal logics

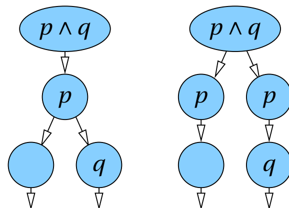
- LTL cannot distinguish these two situations
- CTL can

? Try to write a CTL formula stating that the choice between $\neg q \wedge \neg p$ and $q \wedge \neg p$ remains open a bit longer in one case than in the other



Exercise #1: temporal logics

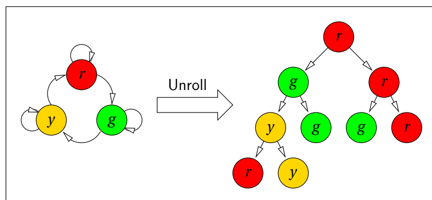
- LTL cannot distinguish these two situations
- CTL can
- ? Try to write a CTL formula stating that the choice between $\neg q \wedge \neg p$ and $q \wedge \neg p$ remains open a bit longer in one case than in the other

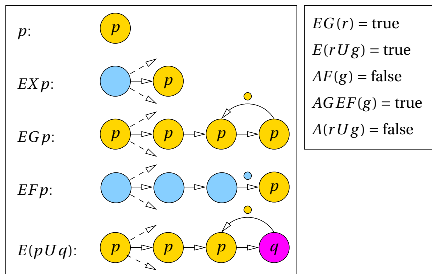


CTL logic

- Most famous example in the world: traffic light (unless you prefer coffee machine?)
- Farm road crossing highway with traffic lights and presence detector on farm road
- Example properties
  - Traffic lights can never be green both on highway and farm road (safety)
  - If car waiting for green light on farm road, it will eventually have green light (liveness)

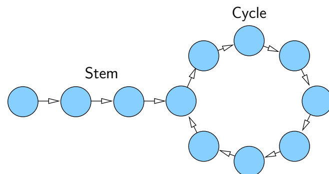
Tree representation of time





Counter-examples

- Counter-example may be finite...
  - Safety
  - $AG(r) : r, g$
- ... Or infinite
  - Liveness
  - $AF(g) : r, r, r, r, ...$
  - Infinite counter-example made of stem and cycle



CTL specification of controller

- Exercise #2: "If client releases bus, controller passes token on next cycle"
- Exercise #3: "If controller receives token but client does not request bus, controller passes token on next cycle"
- Exercise #4: "If client requests bus, when token sent to controller, controller asserts ack on next cycle"
- Exercise #5: "Controller does not pass token until client releases bus"

CTL specification of controller

- Exercise #2: "If client releases bus, controller passes token on next cycle"
- Exercise #3: "If controller receives token but client does not request bus, controller passes token on next cycle"
- Exercise #4: "If client requests bus, when token sent to controller, controller asserts ack on next cycle"
- Exercise #5: "Controller does not pass token until client releases bus"

CTL specification of controller

- Exercise #2: "If client releases bus, controller passes token on next cycle"
- Exercise #3: "If controller receives token but client does not request bus, controller passes token on next cycle"
- Exercise #4: "If client requests bus, when token sent to controller, controller asserts ack on next cycle"
- Exercise #5: "Controller does not pass token until client releases bus"

CTL specification of controller

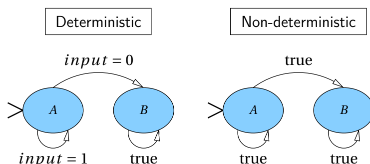
- Exercise #2: "If client releases bus, controller passes token on next cycle"
- Exercise #3: "If controller receives token but client does not request bus, controller passes token on next cycle"
- Exercise #4: "If client requests bus, when token sent to controller, controller asserts ack on next cycle"
- Exercise #5: "Controller does not pass token until client releases bus"

Model of controller environment

- Used to reduce complexity by limiting to realistic behaviors only
- Prevent false alarms (useless counter-examples)
- May use temporal logic...
  - $(req = 1) \Rightarrow AX((req = 1) \vee (ack = 1))$
- ... Or specific language (EDL)...
- ... Or HDL enhanced to handle non-determinism

Non-determinism

- Deterministic systems
  - For every {input, state}, one and only one {next state, output}
  - Implemented digital systems always deterministic
- Non-deterministic systems
  - There exist {input, state} for which {next state, output} not unique
  - Model set of behaviors



Why using non-determinism?

- To model environments
- Example
  - System to verify = arbiter + controller
  - Environment of system = clients
    - Requests asserted in non-deterministic way
    - Non-determinism allows modeling of any behavior of clients
- In the following $ND will be our non-deterministic statement

```
1 wire rand;
2 assign rand = $ND(0,1);
3 if (rand) ...
```

```
1 signal rand: boolean;
2 rand <= $ND(false,true);
3 if (rand) then
4 ...
```

Non-determinism

```
 1  entity client is
 2    port(clk, rstn, ack: in boolean;
 3         req: out boolean);
 4  end ctrl;
 5
 6  architecture beh of client is
 7
 8    type state_type is (idle, request, busy);
 9    signal state: state_type;
10    signal rand: boolean;
11
12  begin
13
14    rand <= $ND(false, true);
15
16    process(clk, rstn)
17    begin
18      if (not rstn) then
19        state <= idle;
20      elsif rising_edge(clk) then
21        case state is
22          when idle =>
23            if (rand) then
24              state <= request;
25            end if;
26          when request =>
27            if (ack and rand) then
28              if (rand) then
29                state <= busy;
30              else
31                state <= idle;
32              end if;
33            end if;
34          when busy =>
35            if (rand) then
36              state <= idle;
37            end if;
38        end case;
39      end if;
40    end process;
41
42    req <= state = request or state = busy;
43
44  end beh;
```

Fairness constraints

- Extend CTL to FCTL
- Express that state or set of states must be reached infinite number of times
  - Example for client: $F^{\infty}IDLE$ ($IDLE$ state must be reached infinite number of times; client cannot keep bus indefinitely)
- Very convenient to limit impact of non-determinism

Algorithms of CTL model checking

- Invented by Queille, Sifakis, Clarke, Emerson and Sistla (1982, 1986)
- Since improved in many ways
- Linear in automata and formula size
- Based on state marking
  - Let A be an automata and ϕ a CTL formula, for every sub-formula ψ of ϕ and for each state q of A, q marked if ψ true in state q
  - Eventually, for every state and every sub-formula
    - $q.\psi = \text{true}$ if q satisfies ψ,
    - else $q.\psi = \text{false}$
- Memory usage critical because marking of $q.\phi$ uses markings of $q'.\psi$ where ψ sub-formula of ϕ and q' state reachable from q
- After marking of ϕ done, $q_0.\phi$ (where q_0 initial state of A) true iff A satisfies ϕ

Algorithms of CTL model checking

- Notations
  - Let Q be the set of states of A
  - Let $L(q)$ be the set of atomic properties p satisfied in state q
  - Let T be the set of transitions (q, q') from state q of A to another state q' of A
  - Let $degree(q)$ be the number of successors of q in state diagram of A
- Complexity of model checking of CTL formula ϕ is $\mathcal{O}(|A| \times |\phi|)$

Algorithms of CTL model checking

- Case #1: $\phi = p$

procedure MARKING(ϕ) $\triangleright\ \mathcal{O}(|Q|)$
    for every $q \in Q$ **do**
        if $p \in L(q)$ **then**
            $q.\phi \leftarrow \text{true};$
        else
            $q.\phi \leftarrow \text{false};$
        end if
    end for
end procedure

Algorithms of CTL model checking

- Case #2: $\phi = \neg\psi$

procedure MARKING(ϕ) $\triangleright$ $\mathcal{O}(|Q|)$
  MARKING(ψ);
  for every $q \in Q$ **do**
    $q.\phi \leftarrow \neg q.\psi$;
  end for
end procedure

Algorithms of CTL model checking

- Case #3: $\phi = \psi_1 \wedge \psi_2$

```
procedure MARKING(φ)                                    ▷ O(|Q|)
    MARKING(ψ1);
    MARKING(ψ2);
    for every q ∈ Q do
        q.φ ← q.ψ1 ∧ q.ψ2;
    end for
end procedure
```

Algorithms of CTL model checking

- Case #4: $\phi = EX\psi$

procedure MARKING(ϕ) ▷ $\mathcal{O}(|Q| + |T|)$
    MARKING(ψ);
    for every $q \in Q$ **do** ▷ Initialization
        $q.\phi \leftarrow \text{false}$;
        for every $(q, q') \in T$ **do**
            if $q'.\psi$ **then**
                $q.\phi \leftarrow \text{true}$;
            end if
        end for
    end for
end procedure

Algorithms of CTL model checking

- Case #5: $\phi = E\psi_1 U \psi_2$

```
procedure INIT(φ)
    MARKING(ψ1);
    MARKING(ψ2);
    for every q ∈ Q do
        q.φ ← false;
        q.seen ← false;
    end for
    for every q ∈ Q do
        if q.ψ2 then
            L ← L ∪ q;
        end if
    end for
end procedure
```

```
procedure MARKING(φ)                          ▷ O(|Q| + |T|)
    INIT(φ);
    while L ≠ ∅ do
        take q ∈ L;                           ▷ q must be marked
        q.φ ← true;
        L ← L − q;
        for every (q', q) ∈ T do              ▷ q' predecessor of q
            if ¬q'.seen then
                q'.seen ← true;
                if q'.ψ1 then
                    L ← L ∪ q';
                end if
            end if
        end for
    end while
end procedure
```

Algorithms of CTL model checking

- Case #6: $\phi = A\psi_1 U \psi_2$

```
procedure INIT(φ)
    MARKING(ψ1);
    MARKING(ψ2);
    for every q ∈ Q do
        q.φ ← false;

q.nb ← degree(q);
    end for
    for every q ∈ Q do
        if q.ψ2 then
            L ← L ∪ q;
        end if
    end for
end procedure
```

```
procedure MARKING(φ)                                  ▷ O(|Q| + |T|)
    INIT(φ);
    while L ≠ ∅ do
        take q ∈ L;                                   ▷ q must be marked
        q.φ ← true;
        L ← L − q;
        for every (q′, q) ∈ T do                      ▷ q′ predecessor of q
            q′.nb ← q′.nb − 1
            if ¬q′.nb = 0 ∧ q′.ψ1 ∧ ¬q′.φ then
                L ← L ∪ q′;
            end if
        end for
    end while
end procedure
```

Complexity of model checking

- $\mathcal{O}(|A| \times |\phi|)$: state space explosion critical issue of model checking (adding one DFF to system doubles number of states)
- Complexity reduction is active research field (abstraction)
- Preliminary reduction of problem (design size) mandatory
  - Exploit environment constraints
  - Remove every irrelevant aspect (for considered property)
  - Model checking not that exhaustive, after all
    - Every situation covered but not on whole design
    - Simulation operate on whole design but not on every situation
    - Simulation often find bugs that were not looked at

Specification of whole system

- Exercise #6: ”If client requests the bus, it will not release its request until it is granted the bus”
- Exercise #7: ”If client A holds the bus and client B requests it, it is impossible that A releases the bus, then requests and is granted the bus again before client B is granted the bus”

Specification of whole system

- Exercise #6: "If client requests the bus, it will not release its request until it is granted the bus"
- Exercise #7: "If client A holds the bus and client B requests it, it is impossible that A releases the bus, then requests and is granted the bus again before client B is granted the bus"

Outline

1 Introduction

2 Formal verification
- Simulation versus formal verification
- Combinational equivalence checking
- Sequential equivalence checking
- Model checking

3 Simulation
- Functional simulation
- Gate level simulation
- Electrical simulation

4 Hardware emulation

5 Conclusion

Outline

1 Introduction

2 Formal verification
- Simulation versus formal verification
- Combinational equivalence checking
- Sequential equivalence checking
- Model checking

3 Simulation
- Functional simulation
- Gate level simulation
- Electrical simulation

4 Hardware emulation

5 Conclusion

Functional simulation

- Validate function
  - Of whole model (exhaustive)
  - On given set of stimulus (non exhaustive)
- Need simulation environment
  - Input stimulus
  - Expected results (reference model)

Functional simulation

- Very efficient at finding quickly lot of ”simple” bugs
- Not adapted to ”complex” errors (corner cases)
- May be cycle accurate (CA) and/or bit accurate (BA)...
  - Physical characteristics ignored
- ... Or not
  - Algorithmic model
- Only one that can simulate whole design. Models must be optimized for simulation speed

Functional simulation

- A fast model must
  - Use right language
  - Save events, that is, signals
  - Save memory, that is, signals
  - Often be very far from actual implementation, even if CA/BA
  - Use right compilation / simulation options (4x with Modelsim for Verilog simulation with and without ”-fast -nocoverage”)
  - Performance measured in number of simulated cycles per second on given CPU with given simulator
- Example of CA/BA IDCT model

Language	**Clock cycles / s**
SystemC	1,000,000
Verilog	1,200,000
VHDL	640,000

Functional simulation

- Warning: simulation speed and debugging capabilities frequently antinomic
  - Dynamic verifications (VHDL)
- Warning: simulation speed and expressiveness frequently antinomic
  - Parallelism
  - Sequential scheduling
  - Variables versus signals

Generation of simulation vectors

- Some tools/languages target generation of simulation environments
  - Cadence (Specman e)
  - Synopsys (OpenVera)
- Theoretically reach higher coverage (code and functional) with much less effort
- Based on dedicated environment description languages
- Use constraint solvers

Types of CA/BA functional models

- Emulator (don't mix up with hardware emulators)
  - Empty shell reading/writing data files
  - Simulation speed = 32x
- Language-optimized non-synthesizable model
  - SystemC or VHDL or Verilog linked with C/C++ (through language/simulator API)
  - Simulation speed = 16x
- Non-synthesizable pure VHDL or Verilog model
  - Simulation speed = 8x
- Synthesizable model (RTL)
  - Simulation speed = 4x
- Synthesized model (netlist)
  - Simulation speed = 1x

Cycle-based simulation

- One single signal: clock
- Specific design effort (replace every signal by a variable: manual scheduling)
- May be automated (relates to synthesis)
- Every process is executed at most once per clock cycle
  - In RTL designs the same combinational process can be resumed several times per clock cycle
- Increase simulation speed (10x to 100x)

Outline

1 Introduction

2 Formal verification
- Simulation versus formal verification
- Combinational equivalence checking
- Sequential equivalence checking
- Model checking

3 Simulation
- Functional simulation
- Gate level simulation
- Electrical simulation

4 Hardware emulation

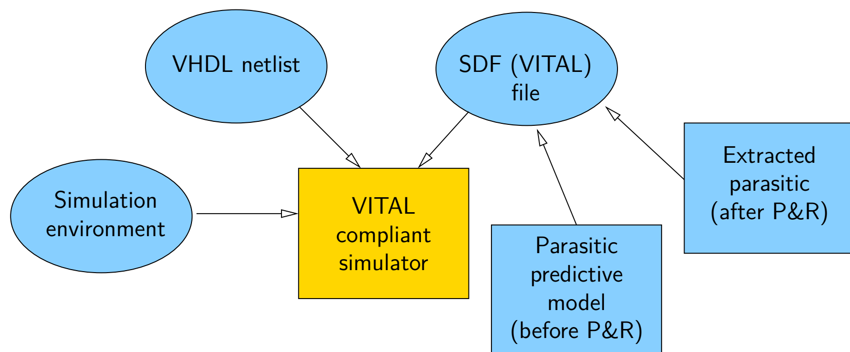
5 Conclusion

Gate level simulation

- Netlist (network of interconnected gates) can also be simulated
- Gate level simulation with or without timing information (back-annotation, VITAL/SDF)
- Different temporal models, with different accuracy vs. performance ratios
  - "Prop Ramp Delay"
  - "Input Slope Model"
  - "Wave tabular"
  - ...

Gate level simulation

- VITAL (IEEE 1076.4-2000) standard defines a way to back-annotate design after synthesis



Prop ramp delay model

- Used only for rough approximations...
- ... Or old processes ($>$ 500 nm)

$$tp_{lh}(A \rightarrow Z) = tp_{lh_0}(A \rightarrow Z) + \Delta tp_{lh_0}(A \rightarrow Z) + \Delta tp_{lh}(A \rightarrow Z) \times C_{ld}$$

$tp_{lh}(A \rightarrow Z)$: Propagation delay from input A to output Z for rising transition of output Z

$tp_{lh_0}(A \rightarrow Z)$: Intrinsic propagation delay (without output capacitance)

$\Delta tp_{lh_0}(A \rightarrow Z)$: Propagation delay due to output capacitance ($= \Delta tp_{lh}(A \rightarrow Z) \times C_s$)

$\Delta tp_{lh}(A \rightarrow Z)$: Propagation delay (per capacitance unit) due to load capacitance

C_{ld} : Load capacitance

TELECOM Paris

IP PARIS

Other models

- "Input Slope Model"
  - Input ramp taken into account
  - Close to "Prop Ramp Delay" for fast ramps
  - More complex -> slower to simulate
  - Not implemented in HDL
  - Reserved to switch-based simulation or static timing analysis
- "Wave tabular"
  - Input waveform taken into account
  - Segmented linear approximation
  - More complex
  - Not implemented in HDL
  - Reserved to switch-based simulation or static timing analysis

Outline

1 Introduction

2 Formal verification
- Simulation versus formal verification
- Combinational equivalence checking
- Sequential equivalence checking
- Model checking

3 Simulation
- Functional simulation
- Gate level simulation
- Electrical simulation

4 Hardware emulation

5 Conclusion

TELECOM Paris

IP PARIS

Electrical simulation

- Transistor-based electrical simulation (SPICE, ELDO) provides more accurate results
- Used for standard cells characterization or simulation of ”full custom” designs
- Very slow, very accurate
- Unavoidable in some specific cases
- Limited to few hundreds of transistors

TELECOM Paris
IP PARIS

Outline

1 Introduction

2 Formal verification
- Simulation versus formal verification
- Combinational equivalence checking
- Sequential equivalence checking
- Model checking

3 Simulation
- Functional simulation
- Gate level simulation
- Electrical simulation

4 Hardware emulation

5 Conclusion

What does it look like?

Figure: Palladium (Cadence) and Veloce (Mentor Graphics) series

Figure: ZeBu series by Synopsys

From single board to huge servers

- Based on
  - FPGAs
  - Dedicated ICs (interconnection)
  - Memories
  - Commercial off-the-shelf components (CPUs, fast I/Os, ...)
- Can emulate thousands to multi-millon gates ICs
- From thousands to multi-millon $ equipments
- At clock frequencies in the MHz range
  - RTL $\times 1000$
  - Gate-level $\times 1,000,000$
- Can be used to emulate the not-yet-available IC in full system (GPU)

Important features

- Operating frequency
- Synthesis / configuration time
- Limitations of design style
  - Synchronous
  - Multiple clock domains
  - Gated clocks
- Size of "emulatable" systems
- Emulation in host system
- Interactive debugging
- Co-simulation
- ...

What is it used for?

- Speed up simulation of synthesizable code
  - Warning: not always best choice
  - Warning: what you simulate not always what you think
- Speed up fault simulation
- Emulate chip in system environment
- Ease hardware-software co-design
- Increase designer's confidence
- Heat building

What is it NOT used for?

- Validate non synthesizable models
- Simulate chip ”as it will actually be”
  - In order to fit in emulator design must be adapted
    - 3 states → multiplexers
    - Memories → available memories
    - ...
- Validate physical characteristics
- Validate netlist modifications (why?)

Need to emulate synthesizable code?

- System level verification?
  - It is always better to have a more abstract model
  - Compare performance
- Validation of refinement for synthesis?
  - Compare with other methods
    - Model checking
    - Equivalence checking
    - Multi-level simulation

Outline

1 Introduction

2 Formal verification
- Simulation versus formal verification
- Combinational equivalence checking
- Sequential equivalence checking
- Model checking

3 Simulation
- Functional simulation
- Gate level simulation
- Electrical simulation

4 Hardware emulation

5 Conclusion

Pentium IV bugs

- 42 millions of transistors
- More than one million lines of RTL code
- 100 high level logic bugs found by formal verification
- Source: EE times 2001

Bug types	%
Goof (typos, cut-and-paste)	12.70
Miscommunication	11.40
Micro-architecture definition	9.30
Logic/microcode changes without care about side effects	9.30
Corner cases (implementation failures)	8.00
Powerdown (clock gating)	5.70
Documentation	4.40
Micro-architecture complexity	3.90
Initialization (reset)	3.40
Late definition	2.80
Incorrect RTL assertions (wrong or broken by design changes)	2.80
Design mistakes	2.60
Total	76.30

Verification effort

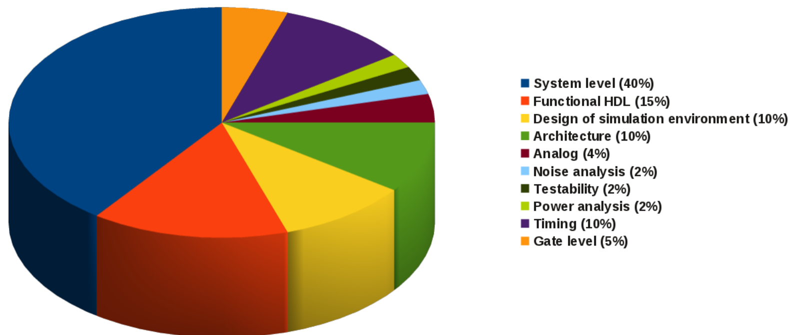


Figure: Breakdown of verification effort (ITRS 1999)

Summary

- Formal verification. Questions?
- Simulation. Questions?
- Other. Questions?