Formal verification in hardware design

Henrik Zant

January 10, 2024

Introduction

Testing vs verification [1]

Traditional testing

- Test if system behaves according to expectations
- "Correct" behaviour often described ambiguously
- Tests may not detect some bugs
  $\rightarrow$ only probabilistic assurances

Formal verification

- Prove a system behaves according to its specification using mathematically sound techniques
- "Correct" behaviour entirely based on unambiguous specification
- Works on abstractions of the physical device

Verification pitfalls [1]

- Specification may not capture client's intention
- Time/Space effort may be huge for bigger designs
- May not capture manufacturing faults/time/environment of physical device
  $\rightarrow$ assumptions need to be made

Verification Approaches [1], [2]

- Specifying desired properties
- High level model as specification
- Combinations of both

Specifying Desired Properties

Model-Checking Introduction [1], [3]

1. stipulate specification as properties that must hold
2. abstract system into a model
    - different modelling techniques available
    - modelling possible at different stages (VHDL/Netlist)
3. verify algorithmically that model fits specification
   or produce counter example if model doesn't fit specification

Model-Checking Approaches [1], [3]

- Different Model-Checking approaches
  - CTL*/CTL/LTL Model-Checking
    - symbolic state representation
    - explicit state representation
  - μ-calculus
  - Trajectory Formulas
  - ...
- Different approaches also differ in **expressiveness** and **runtime**
  $\rightarrow$ rule of thumb: Consider the properties you need and choose approach accordingly

Temporal logic and explicit Model-Checking - Overview [1], [3]

- Hardware model is explicit (e.g. State Transition Graph/Kripke Structure)
- Specification provided in temporal logic
    - Rough idea: allows specifying what propositions need to hold with respect to time
    - Based on propositional logic extended with temporal operators and path quantifiers
    - Many different temporal logics exit: CTL*/**CTL**/LTL)
    - Verification effort depends on which temporal logic is used
    - Expressiveness depends on which temporal logic is used

Modelling hardware using state transition graphs

```
module hardware_design(clk,s,bad);
  input clk;
  output reg [1:0] s;
  output reg bad;
  initial s = 2'b00;
  initial bad = 0;
  always @(posedge clk) begin
    case (s)
      2'b00: s <= 2'b10;
      2'b01: s <= 2'b01;
      2'b10: s <= 2'b01;
      2'b11: begin
        s <= 2'b10; bad <= 1;
      end
    endcase
  end
endmodule
```

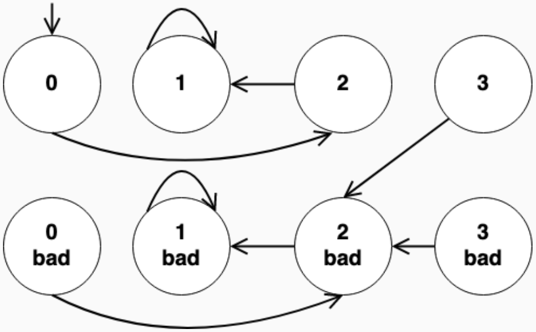

Figure 1: Hardware as state transition graph M_1

State transition graph - Example [1]

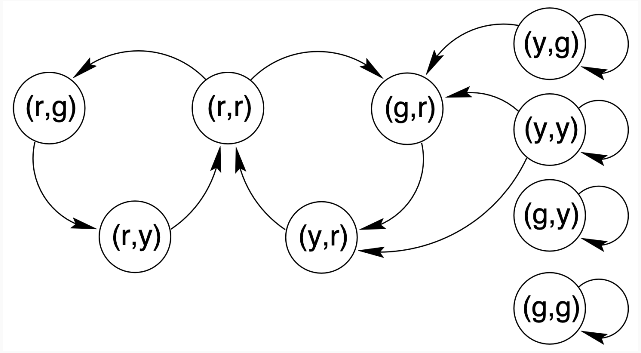

Figure 2: Traffic light state transition graph [1]

- System consisting of two traffic lights
- Atomic Propositions ([*ns*], [*ew*])
    - **n**orth-**s**outh
    - **e**ast-**w**est
    - **r**ed, **y**ellow, **g**reen colors

Computation path/tree example [1], [3]

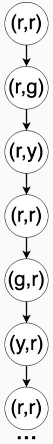

Figure 3: Possible Computation Path

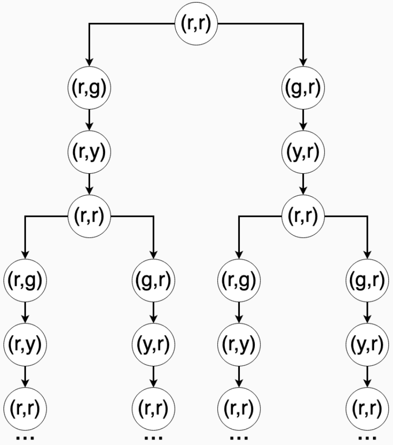

Figure 4: Full Computation Tree

CTL*/CTL/LTL [1], [3]

- Discrete model of time
- Set of shared temporal operators
- Set of path quantifiers
- CTL*/CTL/LTL differ in what temporal operator and path quantifier combinations are allowed

Temporal Operators:

- X: Next
- U: Until
- G: Globally
- F: Eventually

Path Quantifiers:

- A: for all paths
- E: there exists a path

We will focus on CTL for in this presentation

Temporal operators visualized

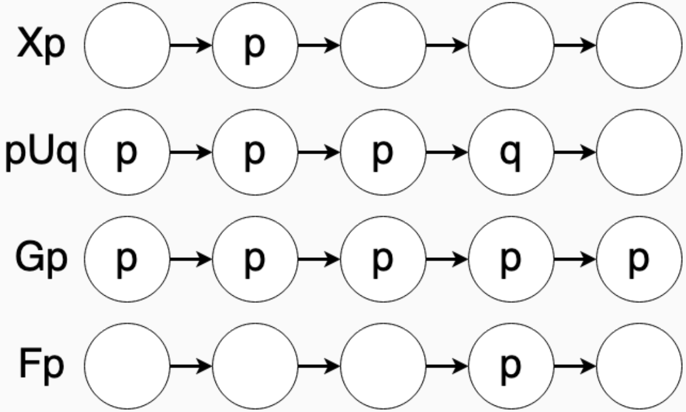

Figure 5: Temporal operators visualized

CTL Definition [1]

- $\forall p \in AP$ is a CTL formula
- If f_1 and f_2 are CTL formulas, then $\neg f_1$, $f_1 \wedge f_2$, AXf_1, EXf_1, $A(f_1\ U\ f_2)$ and $E(f_1\ U\ f_2)$ are also CTL formulas
    - AXf holds for s_0 if along all paths f holds in all direct successor states
    - EXf holds for s_0 if there exists a path on which f holds for the direct succesor state
    - $A(f_1 U f_2)$ holds for s_0 if along all paths f_1 always holds until f_2 holds
    - $E(f_1 U f_2)$ holds for s_0 if there exists a path on which f_1 always holds until f_2 holds
    - even more "composite" operators available:
        - $EFf = E(true\ U\ f)$ holds if there exists a path on which f eventually holds
        - $AGf = \neg EF \neg f$ holds if along all paths f holds for every state
        - ...

CTL verification using explicit Model-Checking [1]

- Based on labeling states of State Transition Graph
- Every operator pair (eg. EXp, AFp) comes with a graph-search based algorithm
  Example: For the pair EXp: a state is labeled EXp if some of its successors have the label p in $O(|S| + |R|)$
- For a given CTL formula f and an initial state s_0:
  1. Break up f into all of its subformulas (inside-out)
  2. Apply labels to the Graph using these subformulas
  3. If s_0 is labeled with f then $M, s_0 \models f$
- Complexity: $O(|f|(|S| + |R|))$

CTL Model-Checking - Example

- $f = AF(EXq)$, where $q = ([ew] = g)$
- $s_0 = (r, r)$

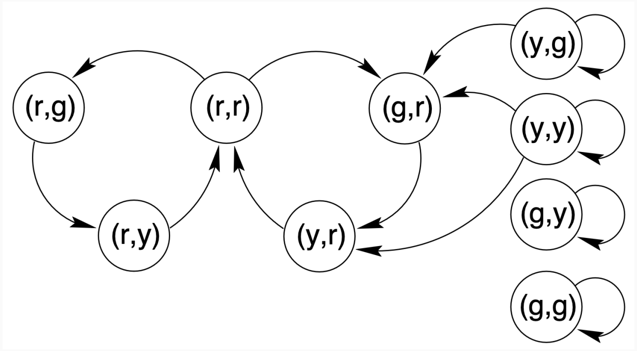

Figure 6: Traffic light state transition graph M [1]

CTL Model-Checking - Example

- $f = AF(EX\boldsymbol{q})$, where $q = ([ew] = g)$
- $s_0 = (r, r)$

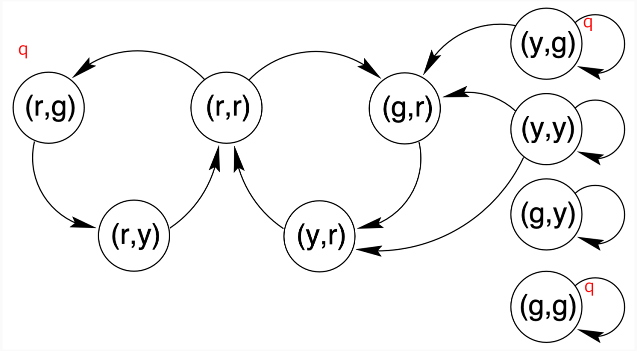

Figure 7: Traffic light state transition graph M [1]

CTL Model-Checking - Example

- $f = AF(\boldsymbol{EXq})$, where $q = ([ew] = g)$
- $s_0 = (r, r)$

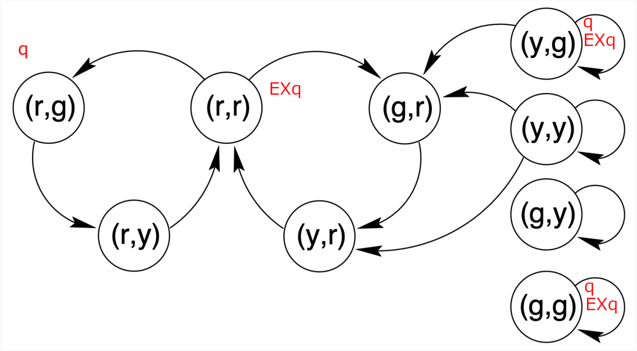

Figure 8: Traffic light state transition graph M [1]

CTL Model-Checking - Example

- $f = \boldsymbol{AF(EXq)}$, where $q = ([ew] = g)$
- $s_0 = (r, r)$
- $M, s_0 \models f$

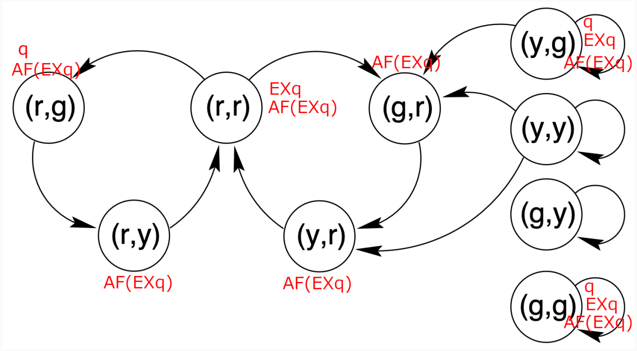

Figure 9: Traffic light state transition graph M [1]

Specifying high level models [1]

Specifying high level models

- High level model serves as system specification
- Implementation is the refinement of high level model (i.e implementation details are abstracted away from high level model)
- Sometimes multiple abstraction layers between high level model and implementation
- Different abstraction approaches (out of scope for this paper)
- Refinement-Checking: Use mathematically sound reasoning to verify that your hardware is an implementation of the model

Combining property-based and high level model based approaches

Combining different techniques to formally verify a SoC [2]

- IP cores can be verified, or come with specification and are pre-verified by vendors
- Standard SoC bus can also be verified using SMV Model-Checker
- IP cores may expect different bus protocol
  $\rightarrow$ "glue" logic required to connect standard bus to IP core
  - collection of abstractions between standard bus other protocols can be model-checked
  - verify that implementation of glue logic is a refinement of previously defined abstraction
- using the IP core specs, glue-abstraction specification, bus-specification, verify that the whole SoC complies to its specification

Conclusions

Conclusions

- Verification is based on mathematical sound properties and requires a formal specification
- Verification is executed on models $\rightarrow$ can't guarantee Hardware was manufactured correctly
- Two main approaches: High Level Model Specification vs Property Based Approach
- Different techniques require different assumptions and have different expressiveness and runtime requirements

References i

References

[1] C. Kern and M. R. Greenstreet, **Formal verification in hardware design: A survey,** ACM Trans. Des. Autom. Electron. Syst., vol. 4, no. 2, pp. 123–193, Apr. 1999, ISSN: 1084-4309. DOI: 10.1145/307988.307989. [Online]. Available: https://doi.org/10.1145/307988.307989.

[2] P. Chauhan, E. Clarke, Y. Lu, and D. Wang, **Verifying ip-core based system-on-chip designs,** in Twelfth Annual IEEE International ASIC/SOC Conference (Cat. No.99TH8454), 1999, pp. 27–31. DOI: 10.1109/ASIC.1999.806467.

[3] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, Model Checking (Cyber Physical Systems Series), 2nd ed. London, England: MIT Press, Dec. 2018.