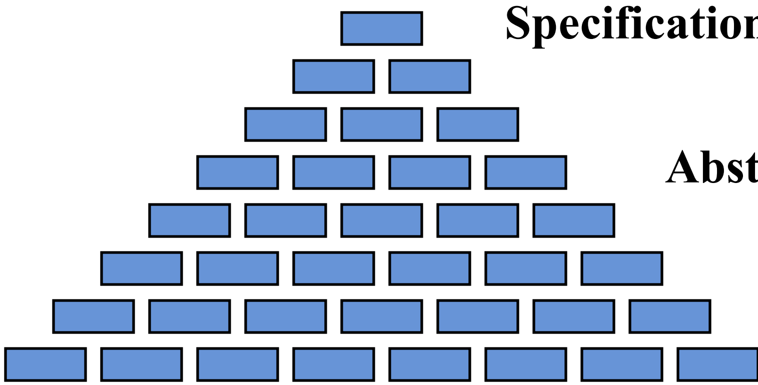
Part II

Concepts

The structure of a design proof

- A proof is a pyramid
  - "Bricks" are assertions, models, *etc*…
  - Each assertion rests on lower-level assertions

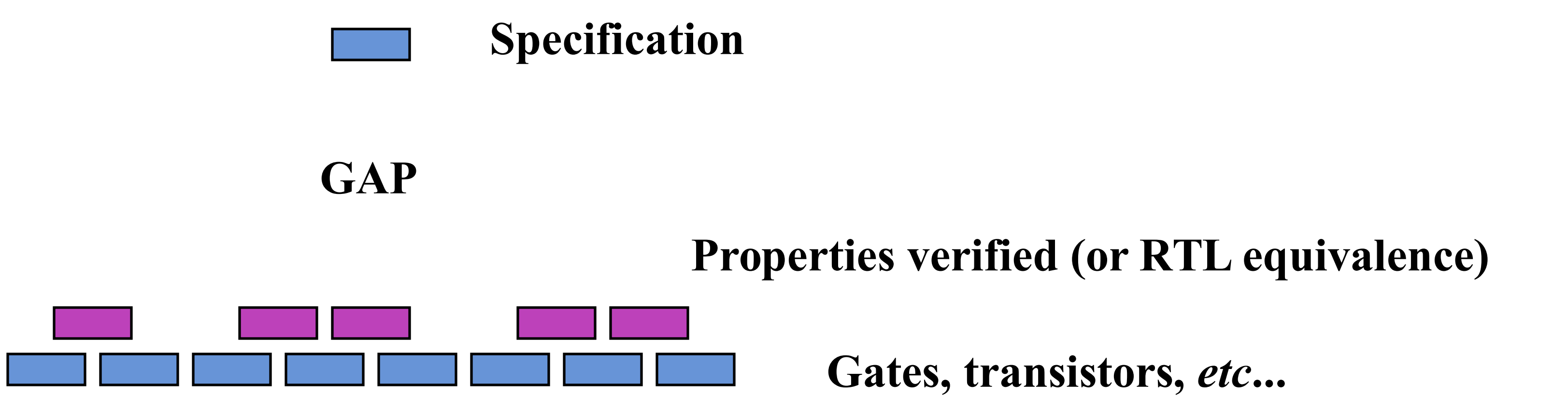
So what happens if we remove some bricks?



Local property verification

- Verify properties of small parts of design, *e.g*…
  - Bus protocol conformance
  - No pipeline hazards
- Like type checking, rules out certain localized errors

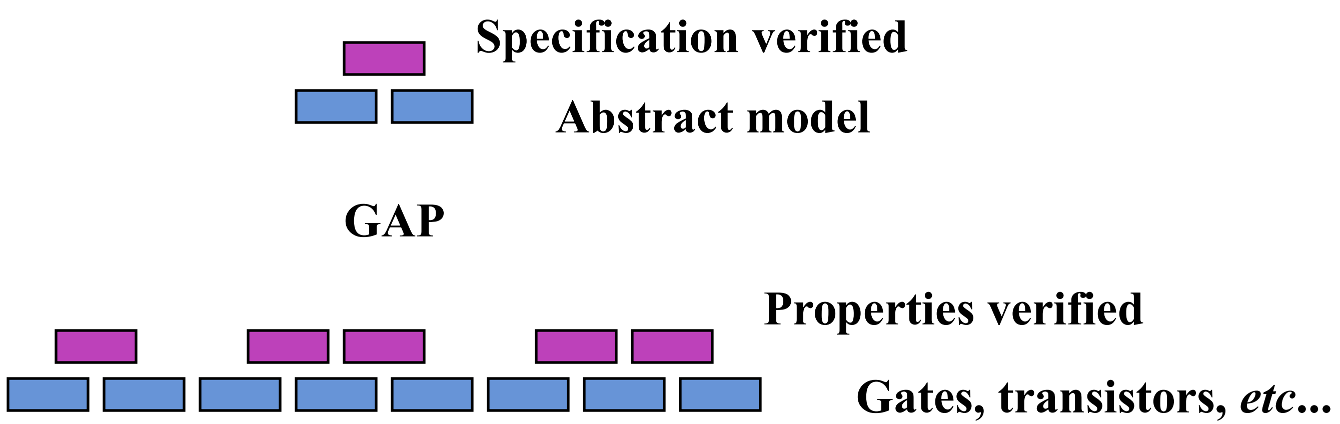
Although this leaves a rather large gap...



Abstract models

- Make an *ad-hoc* abstraction of the design
- Verify that it satisfies specification
- Separate, e.g., protocol and implementation correctness

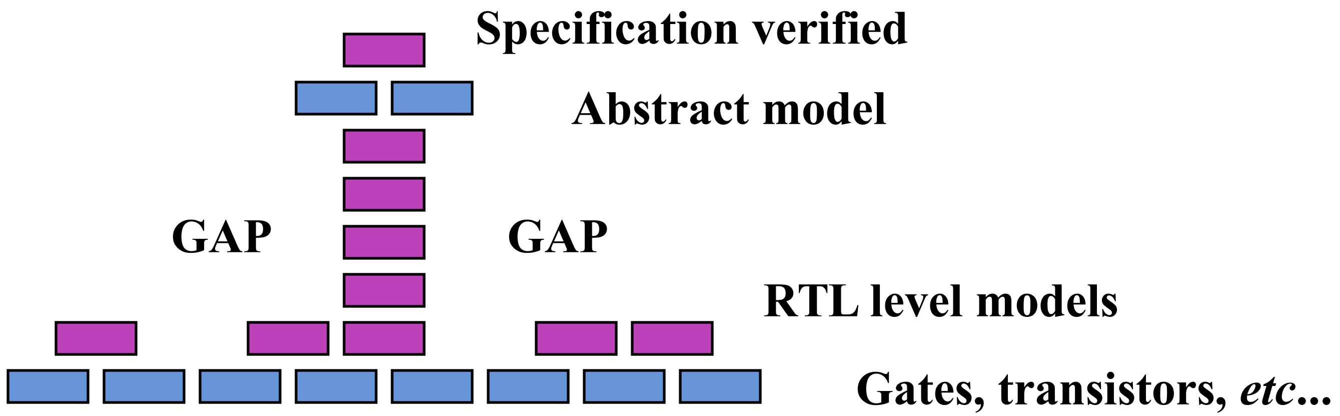
But how do we know we have implemented this abstraction?



Partial refinement verification

- Verify that key RTL components implement abstraction
- Abstract model provides environment for RTL verification
- Make interface assumptions explicit
  - Can transfer interface assumptions to simulation

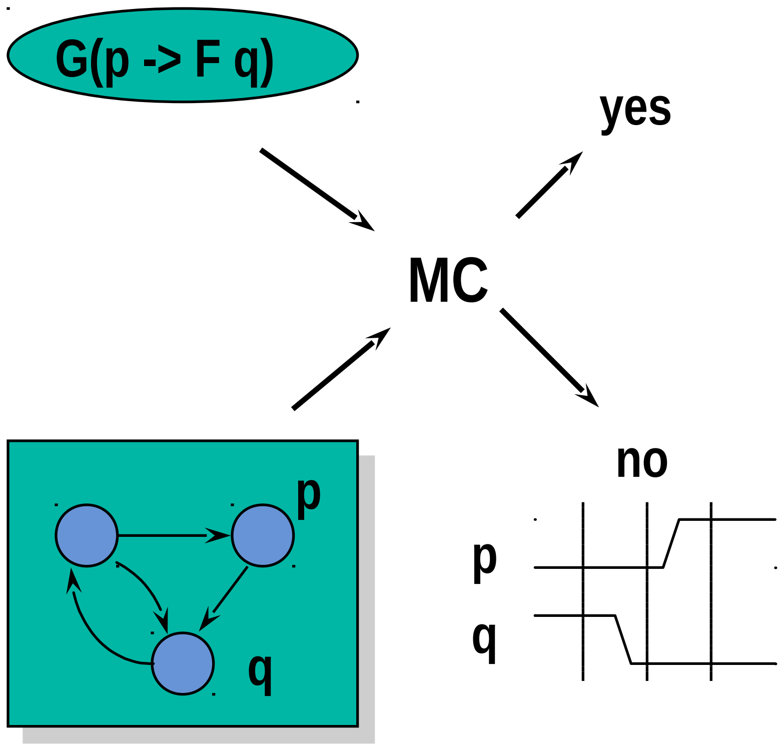
We can rule out errors in certain RTL components, assuming our interface constraints are met.



Overview

- Property specification and verification
  - temporal logic model checking
  - finite automata and language containment
  - symbolic trajectory evaluation
- Abstraction
  - system-level finite-state abstractions
  - abstraction with uninterpreted function symbols
- Refinement verification
  - refinement maps
  - cache coherence example

Model Checking (Clarke and Emerson)



- output
  - yes
  - no + counterexample
- input:
  - temporal logic spec
  - finite-state model

(look ma, no vectors!)

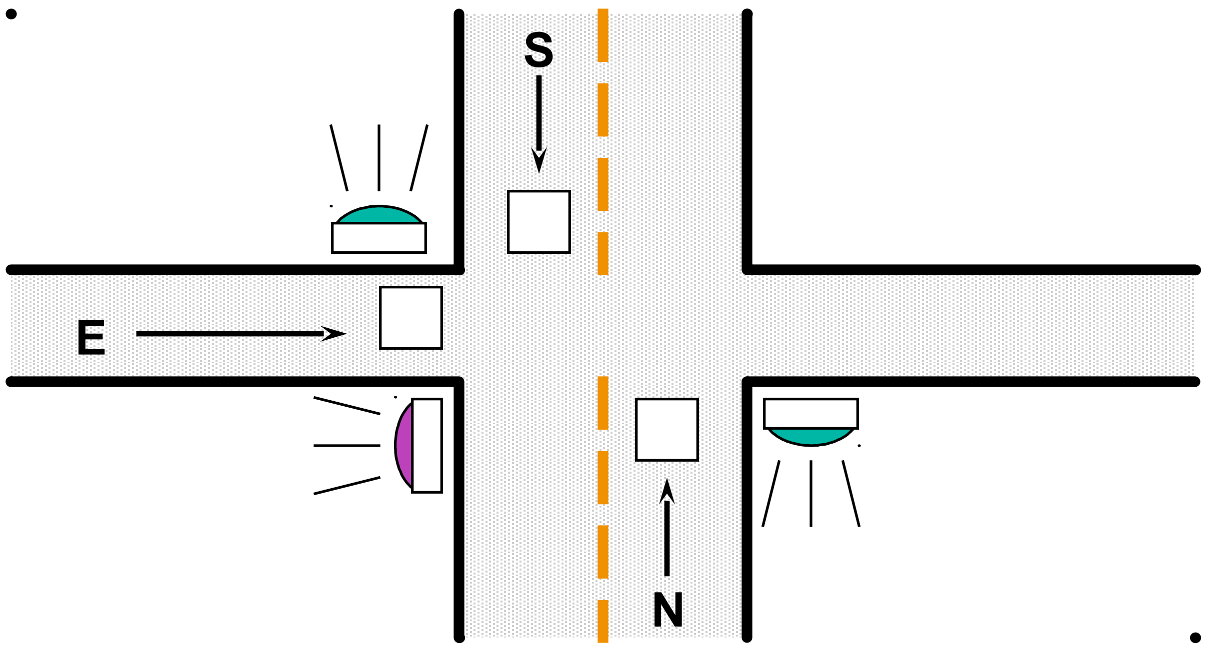
Linear temporal logic (LTL)

- A logical notation that allows to:
  - specify relations in time
  - conveniently express finite control properties
- Temporal operators
  - G p “henceforth p”
  - F p “eventually p”
  - X p “p at the next time”
  - p W q “p unless q”

Types of temporal properties

- Safety (nothing bad happens)
  - G ~(ack1 & ack2) "mutual exclusion"
  - G (req -> (req W ack)) "req must hold until ack"
- Liveness (something good happens)
  - G (req -> F ack) "if req, eventually ack"
- Fairness
  - GF req -> GF ack "if infinitely often req, infinitely often ack"

Example: traffic light controller



- Guarantee no collisions
- Guarantee eventual service

Controller program

```
module main(N_SENSE,S_SENSE,E_SENSE,N_GO,S_GO,E_GO);
  input  N_SENSE, S_SENSE, E_SENSE;
  output N_GO, S_GO, E_GO;
  reg    NS_LOCK, EW_LOCK, N_REQ, S_REQ, E_REQ;

  /* set request bits when sense is high */

  always begin if (!N_REQ & N_SENSE) N_REQ = 1; end
  always begin if (!S_REQ & S_SENSE) S_REQ = 1; end
  always begin if (!E_REQ & E_SENSE) E_REQ = 1; end
```

Example continued...

```
/* controller for North light */
 always begin
    if (N_REQ)
      begin
        wait (!EW_LOCK);
        NS_LOCK = 1; N_GO = 1;
        wait (!N_SENSE);
        if (!S_GO) NS_LOCK = 0;
        N_GO = 0; N_REQ = 0;
      end
  end

/* South light is similar . . . */
```

Example code, cont…

```
/* Controller for East light */
 always begin
    if (E_REQ)
      begin
        EW_LOCK = 1;
        wait (!NS_LOCK);
        E_GO = 1;
        wait (!E_SENSE);
        EW_LOCK = 0; E_GO = 0; E_REQ = 0;
      end
  end
```

Specifications in temporal logic

- Safety (no collisions)

```
G ~(E_Go & (N_Go | S_Go));
```

- Liveness

```
G (~N_Go & N_Sense -> F N_Go);
G (~S_Go & S_Sense -> F S_Go);
G (~E_Go & E_Sense -> F E_Go);
```

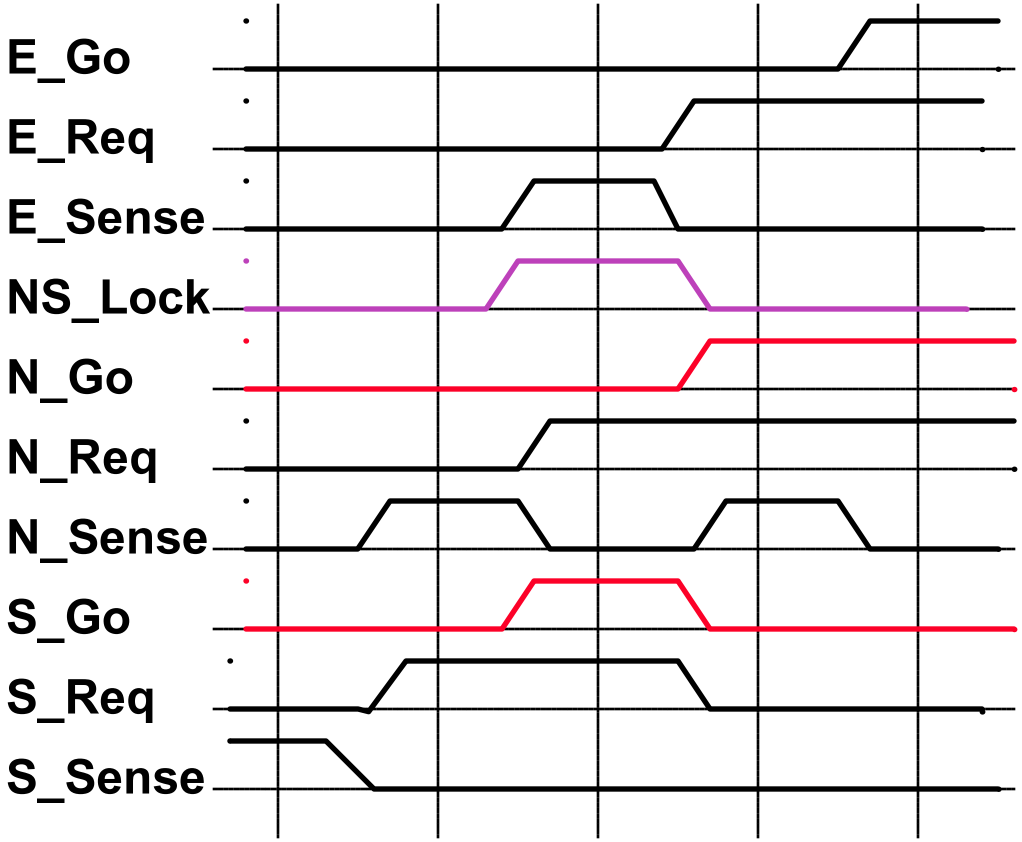
- Fairness constraints

```
GF ~(N_Go & N_Sense);
GF ~(S_Go & S_Sense);
GF ~(E_Go & E_Sense);
```

```
/* assume each sensor off infinitely often */
```

Counterexample

- East and North lights on at same time...



N light goes on at same time S light goes off.

S takes priority and resets NS_Lock

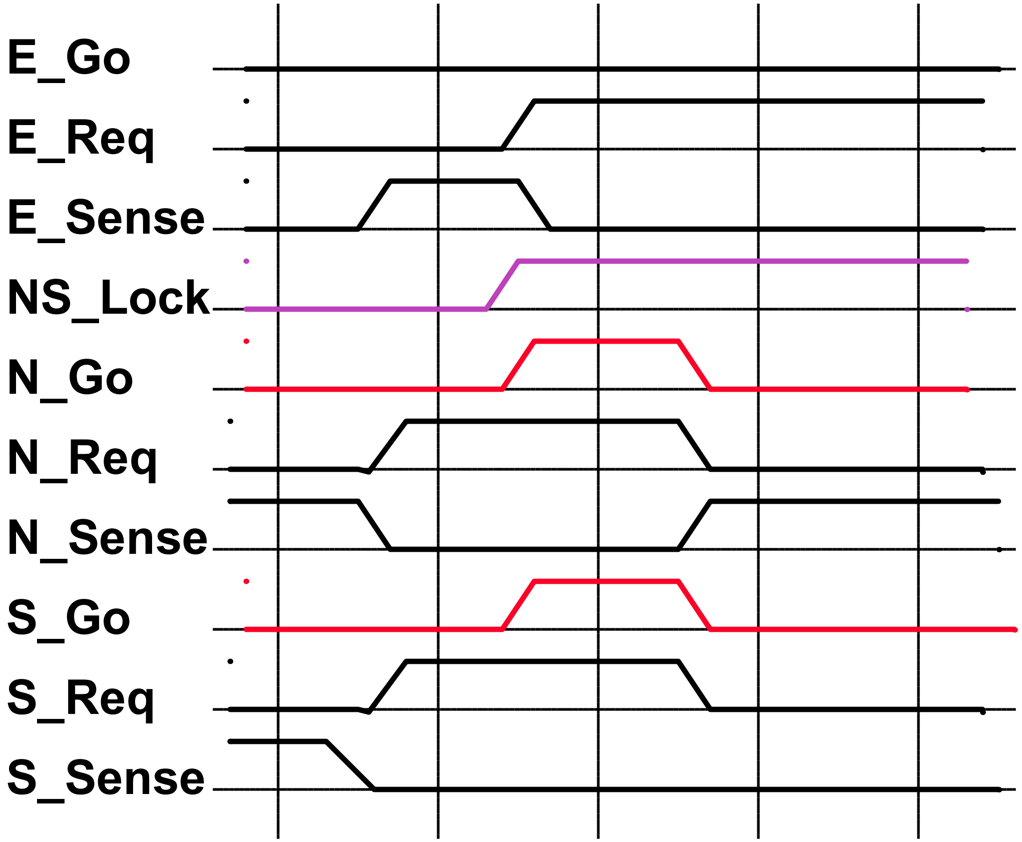
Fixing the error

- Don’t allow N light to go on while south light is going off.

```
always begin
    if (N_REQ)
      begin
        wait (!EW_LOCK & !(S_GO & !S_SENSE));
        NS_LOCK = 1; N_GO = 1;
        wait (!N_SENSE);
        if (!S_GO) NS_LOCK = 0;
             N_GO = 0; N_REQ = 0;
      end
  end
```

Another counterexample

- North traffic is never served...



N and S lights go off at same time

Neither resets lock

Last state repeats forever

Fixing the liveness error

- When N light goes off, test whether S light is also going off, and if so reset lock.

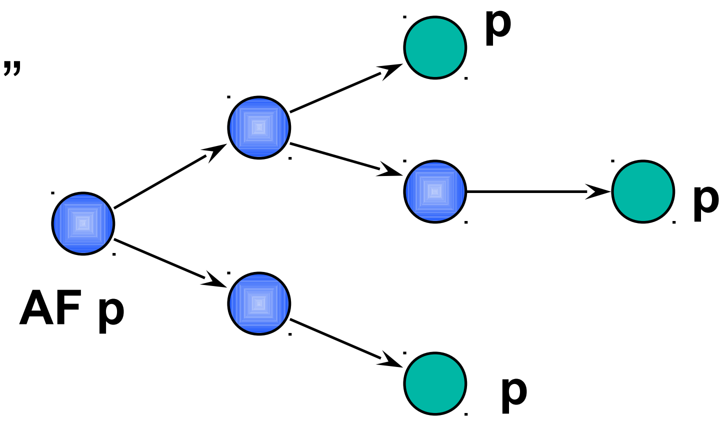
```
always begin
    if (N_REQ)
      begin
        wait (!EW_LOCK & !(S_GO & !S_SENSE));
        NS_LOCK = 1; N_GO = 1;
        wait (!N_SENSE);
        if (!S_GO | !S_SENSE) NS_LOCK = 0;
             N_GO = 0; N_REQ = 0;
      end
  end
```

All properties verified

- Guarantee no collisions
- Guarantee service assuming fairness
- Computational resources used:
  - 57 states searched
  - 0.1 CPU seconds

Computation tree logic (CTL)

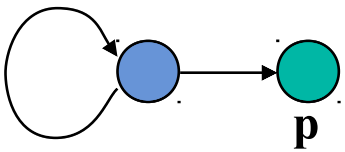
- Branching time model
- Path quantifiers
  - A = “for all future paths”
  - E = “for some future path”
- Example: AF p = “inevitably p”



- Every operator has a path quantifier
  - AG AF p instead of GF p

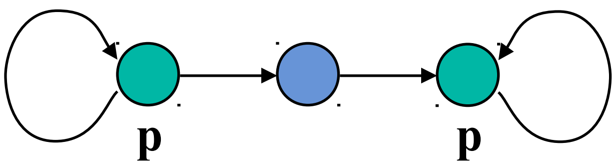
Difference between CTL and LTL

- Think of CTL formulas as approximations to LTL
  - AG EF p is <u>weaker</u> than G F p



Good for finding bugs...

  - AF AG p is <u>stronger</u> than F G p



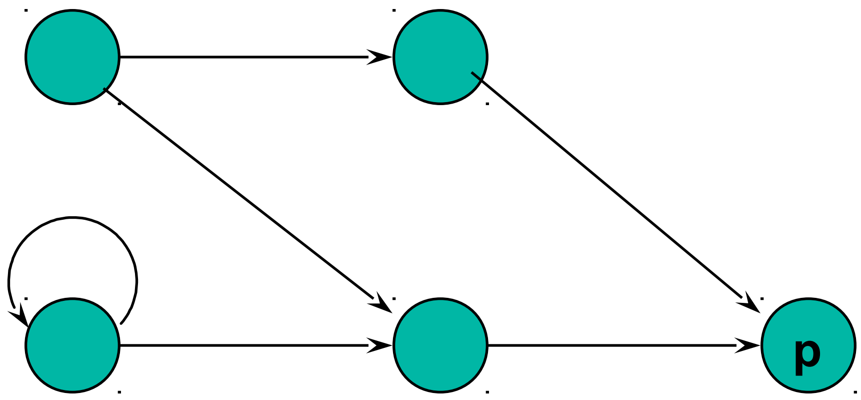
Good for verifying...

- CTL formulas easier to verify

So, use CTL when it applies...

CTL model checking algorithm

- Example: AF p = "inevitably p"

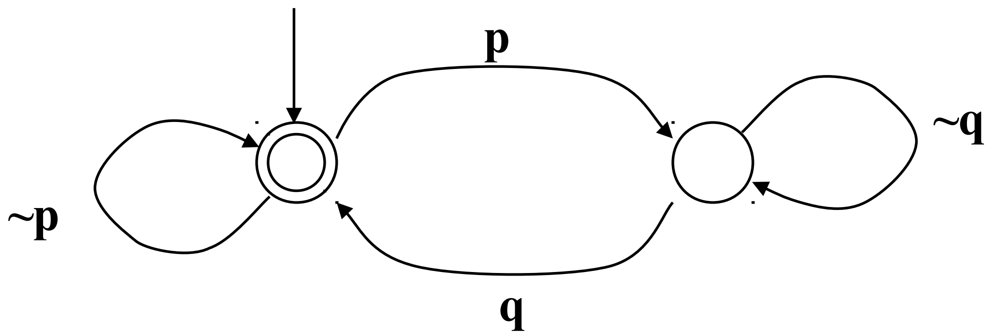


- Complexity
  - linear in size of model (FSM)
  - linear in size of specification formula

Note: general LTL problem is exponential in formula size

Specifying using ω-automata

- An automaton accepting infinite sequences



G (p -> F q)

  - Finite set of states (with initial state)
  - Transitions labeled with Boolean conditions
  - Set of accepting states

Interpretation:

- **A run is accepting if it visits an accepting state infinitely often**
- **Language = set of sequences with accepting runs**

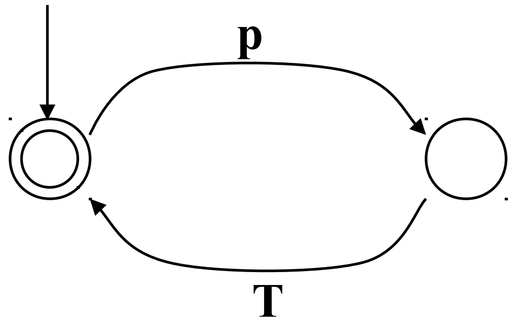
Verifying using ω-automata

- Construct parallel product of model and automaton
- Search for "bad cycles"
  - Very similar algorithm to temporal logic model checking
- Complexity (deterministic automaton)
  - Linear in model size
  - Linear in number of automaton states
  - Complexity in number of acceptance conditions varies

Comparing automata and temporal logic

- Tableau procedure
  - LTL formulas can be translated into equivalent automata
  - Translation is exponential
- ω-automata are strictly more expressive than LTL

Example:



"p at even times"

- LTL with "auxiliary" variables = ω-automata

Example:

G (even -> p)

where:

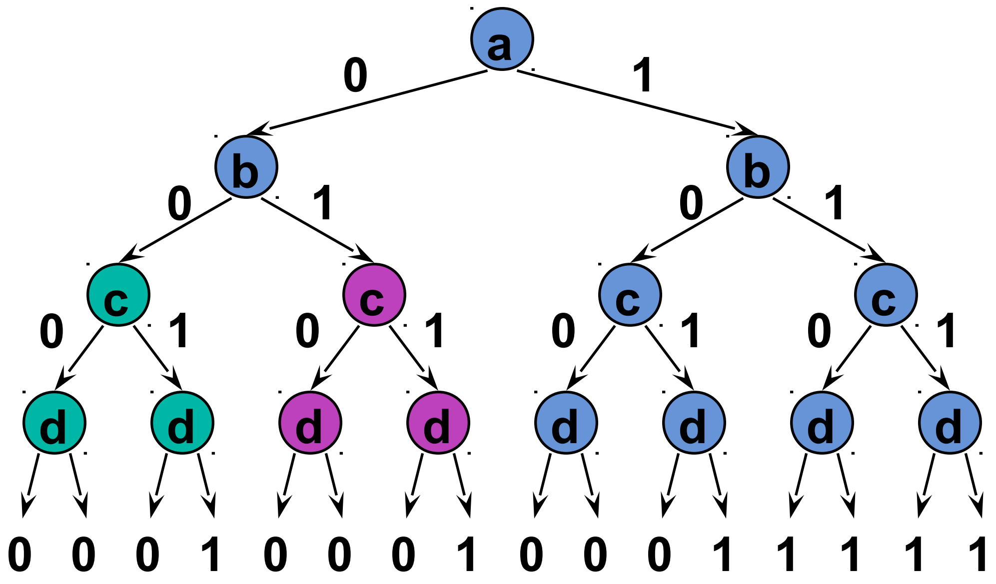
init(even) := 1;
next(even) := ~even;

State explosion problem

- What if the state space is too large?
  - too much parallelism
  - data in model
- Approaches
  - “Symbolic” methods (BDD’s)
  - Abstraction/refinement
  - Exploit symmetry
  - Exploit independence of actions

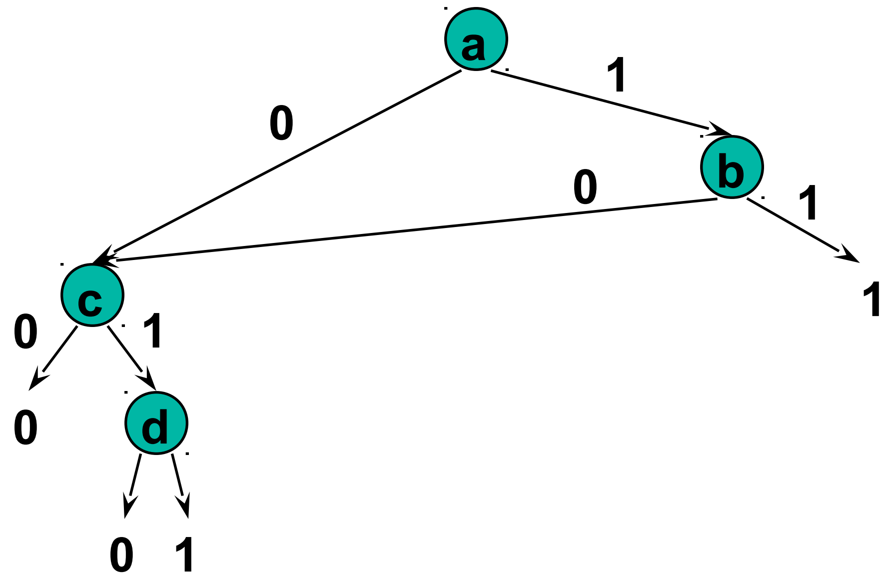
Binary Decision Diagrams (Bryant)

- Ordered decision tree for $f = ab + cd$



OBDD reduction

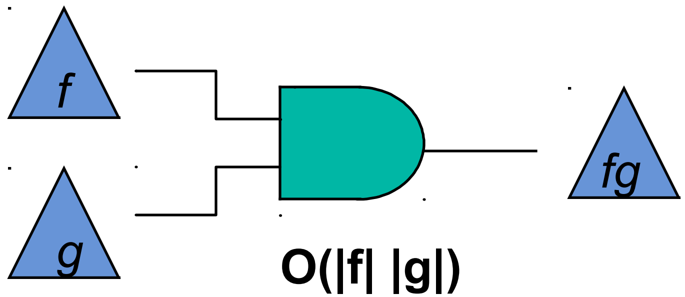
- Reduced (OBDD) form:



- Key idea: combine equivalent sub-cases

OBDD properties

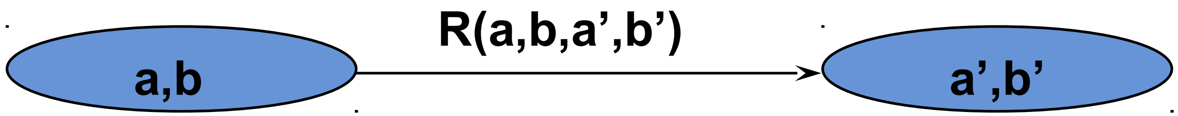
- Canonical form (for fixed order)
  - direct comparison
- Efficient algorithms
  - build BDD’s for large circuits



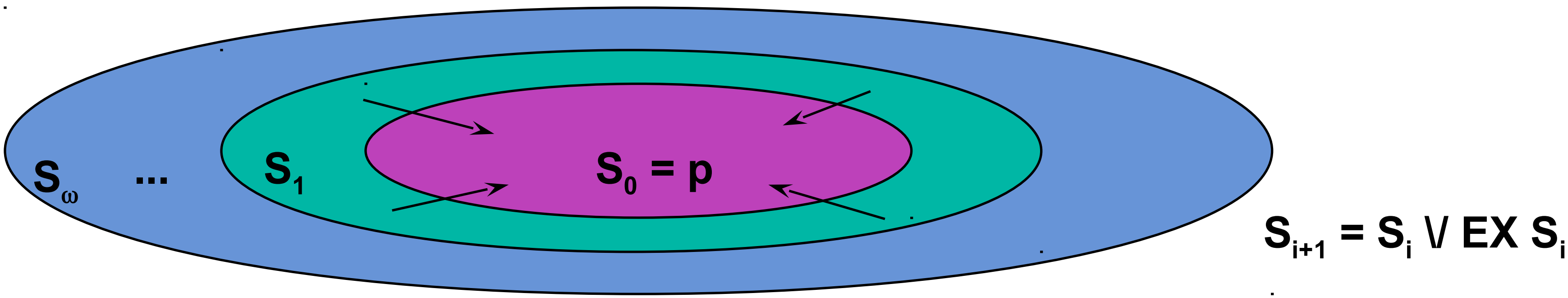
- Variable order strongly affects size

Symbolic Model Checking

- Represent sets and relations with Boolean functions

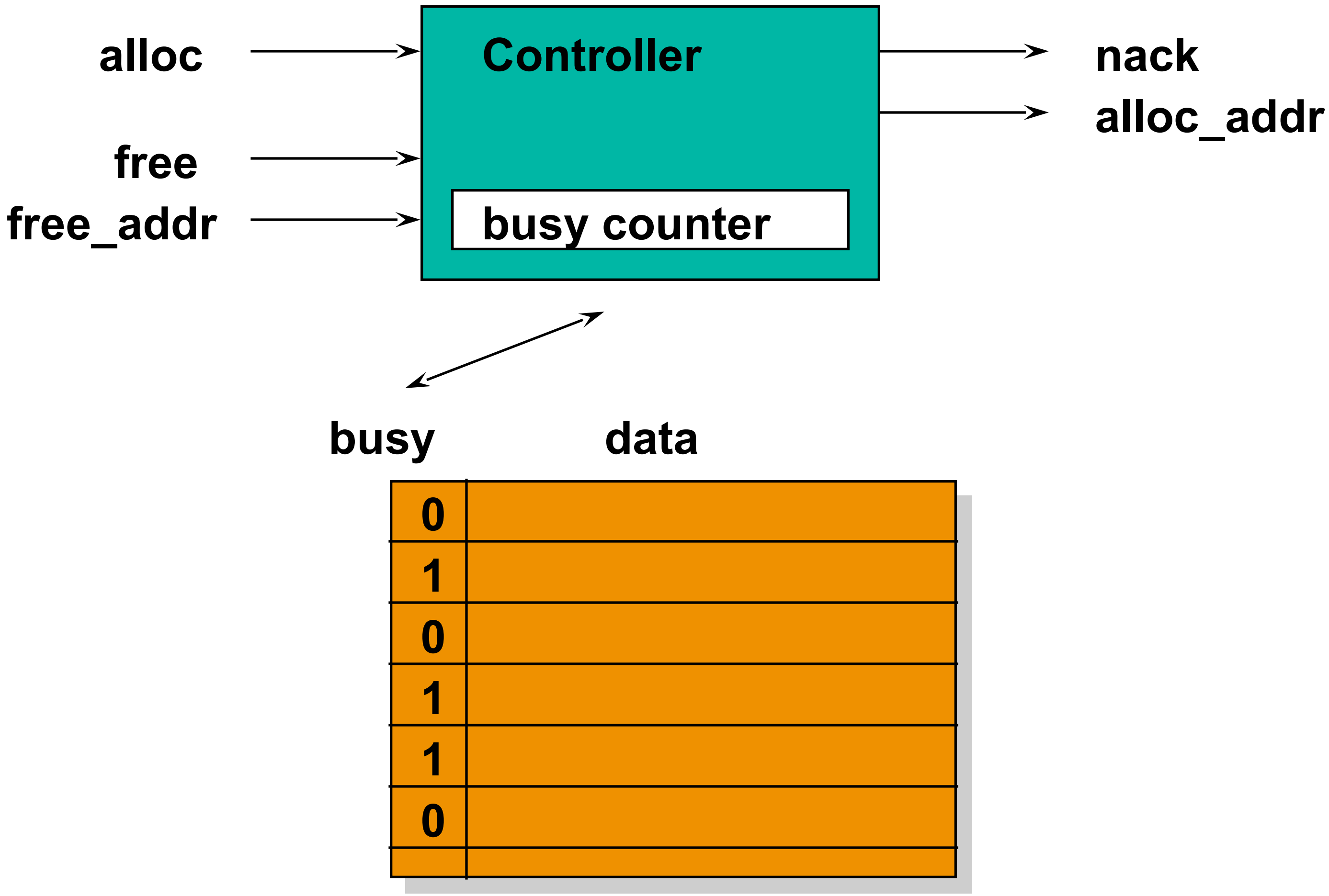


- Breadth-first search using BDD's



  - Enables search of larger state spaces
  - Handle more complex control
  - Can in some cases extend to data path specifications

Example: buffer allocation controller



Verilog description

```
assign nack = alloc & (count == `SIZE);
assign count = count + (alloc & ~nack) - free;

always begin
  if(free) busy[free_addr] = 0;
  if(alloc & ~nack) busy[alloc_addr] = 1;
end

always begin
  for(i = (`SIZE - 1); i >= 0; i = i - 1)
    if (~busy[i]) alloc_addr = i;
end
```

LTL specifications

- Alloc'd buffer may not be realloc'd until freed

```
allocd[i] = alloc & ~nack & alloc_addr = i;
freed[i]  = free & free_addr = i;

G (allocd[i] -> (~allocd[i] W freed[i]);
```

- Must assume the following always holds:
  - `G (free -> busy[free_addr]);`

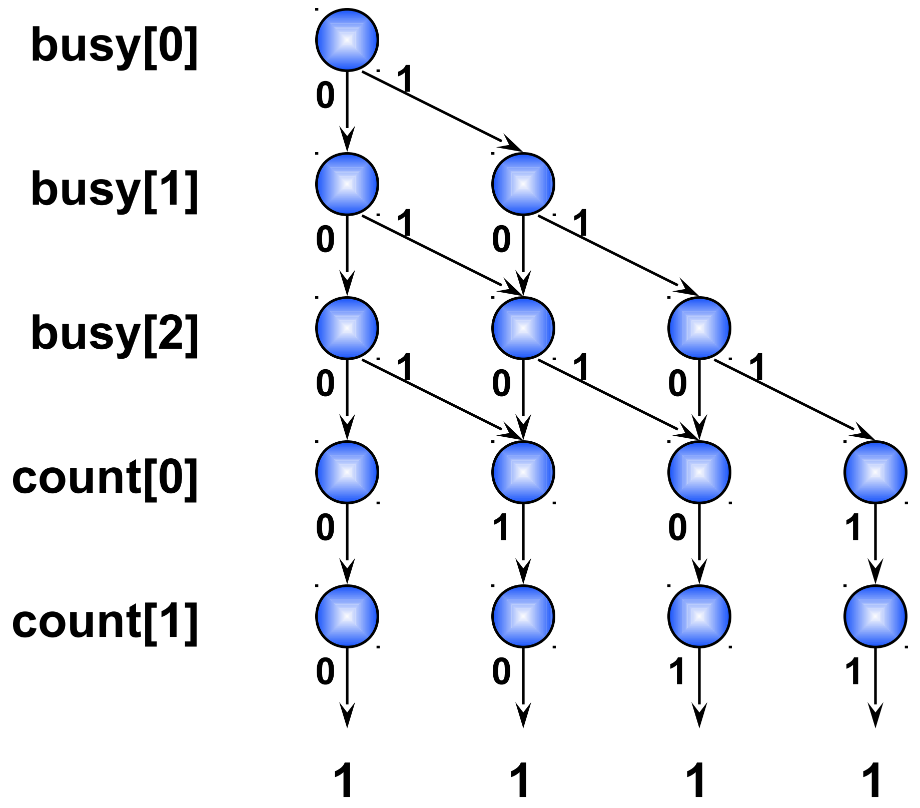
Verification results

SIZE = 32 buffers:

Time	**68 s**
BDD nodes used	
transition relation	**~7000**
reached state set	**~600**
total	**~60000**
Total number of states	**4G**

Why are BDD's effective?

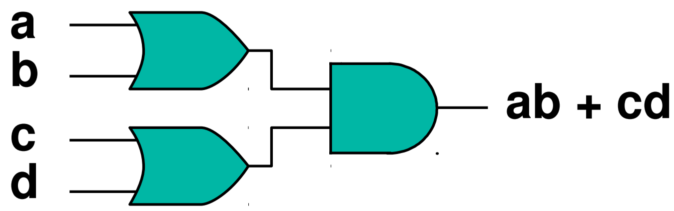
- Combining equivalent subcases:



All cases where sum of busy = x are equivalent

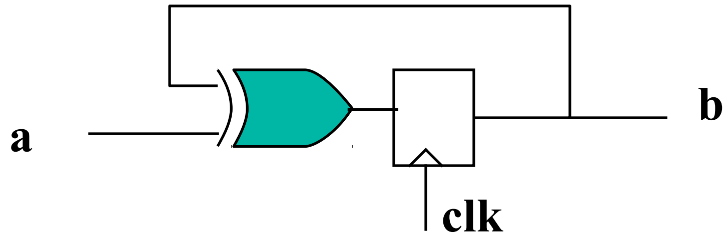
Symbolic simulation

- Simulate with Boolean functions instead of logic values

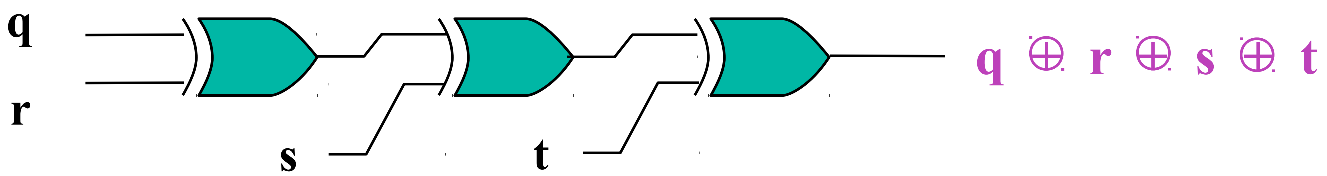


- Use BDD’s to represent functions

Example: sequential parity circuit

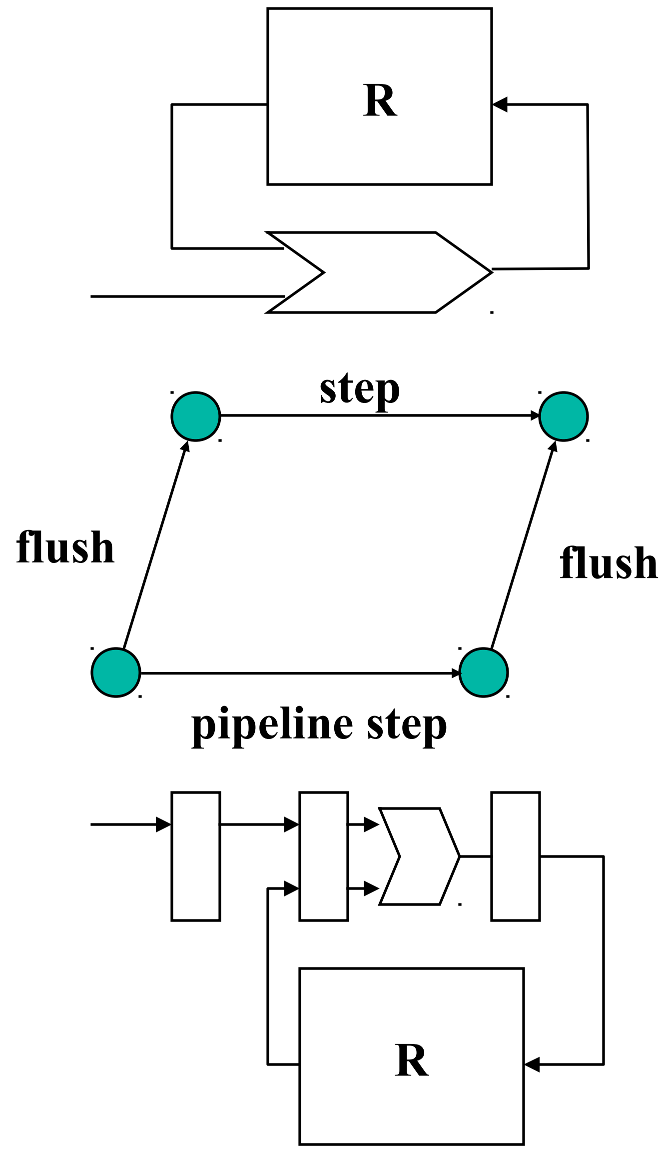


- Specification
  - Initial state — $b_0 = q$
  - Input sequence — $a_0 = r,\ a_1 = s,\ a_2 = t$
  - Final state — $b_3 = q \oplus r \oplus s \oplus t$
- Symbolic simulation = unfolding



Pipeline verification

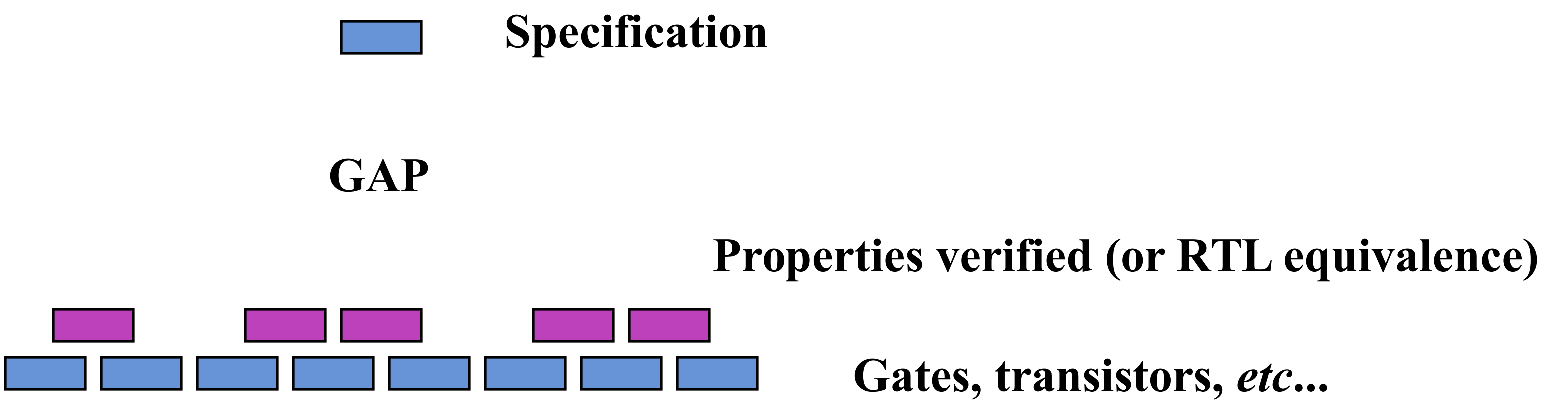
unpipelined



commutative diagram

pipelined

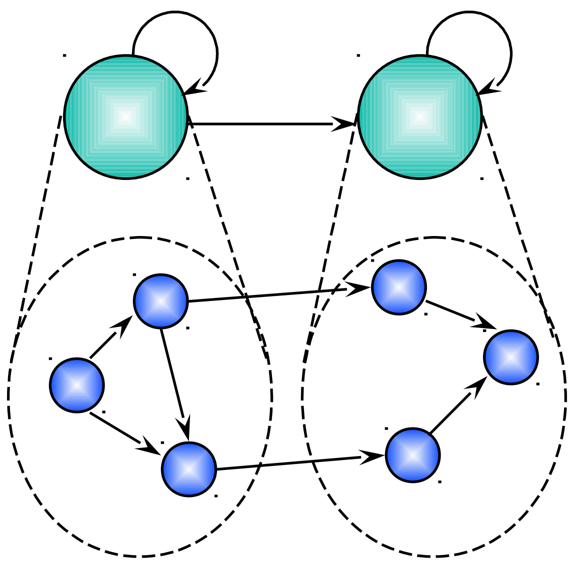
Property verification



- Like type checking…
  - Rules out certain localized errors
  - Static -- requires no vectors
- Does not guarantee correct interaction of components

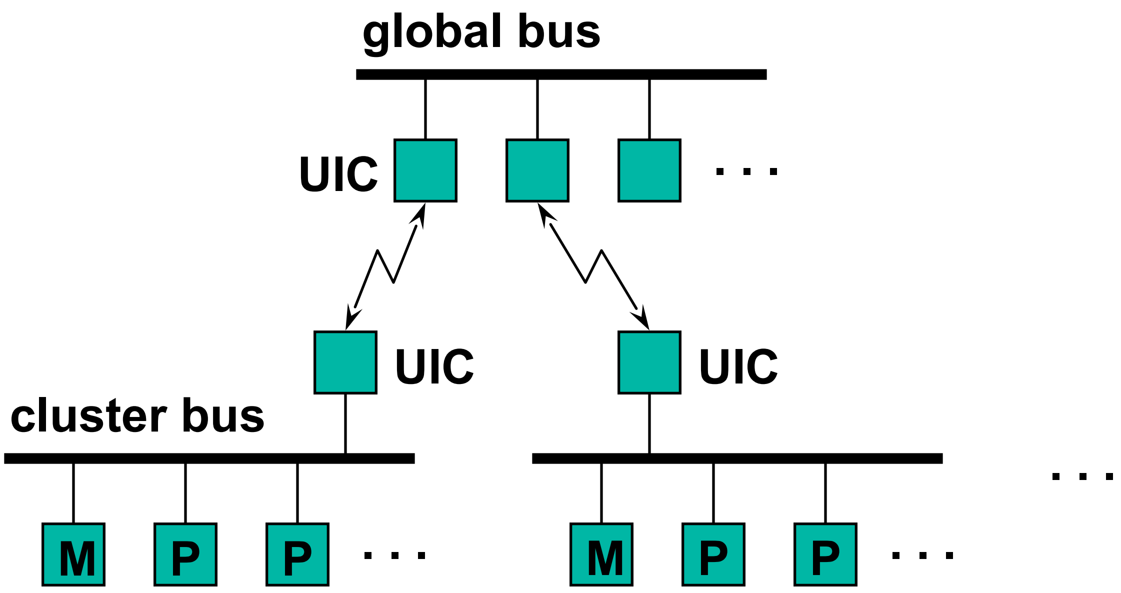
Abstraction

- Reduces state space by hiding some information
- Introduces non-determinism



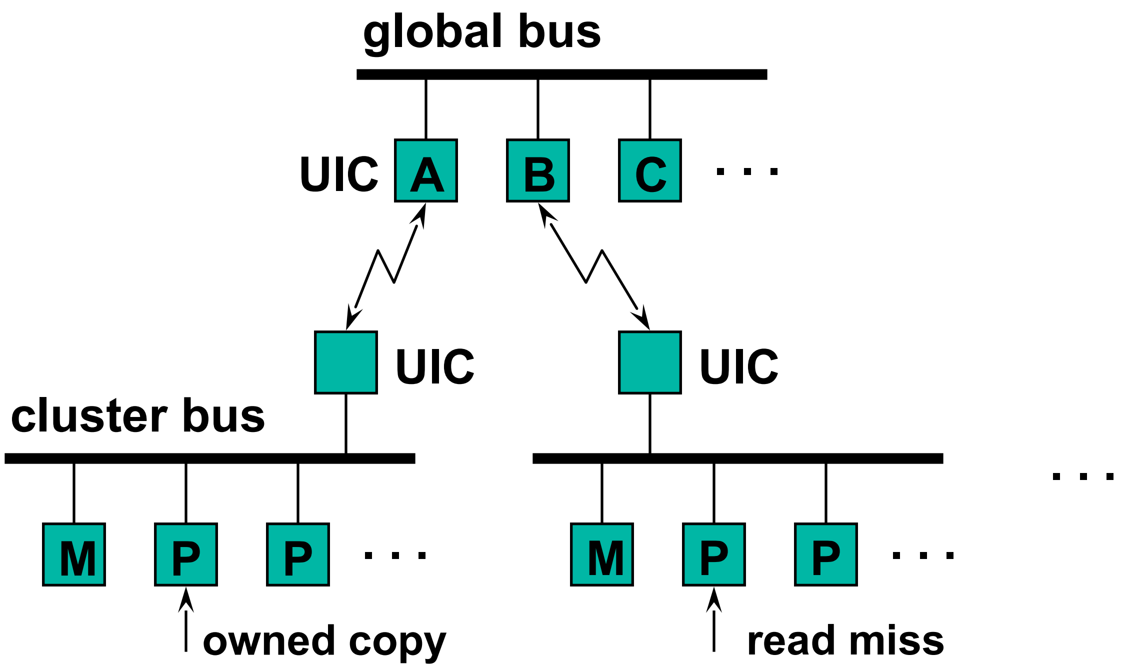
- Allows verification at system level

Example: “Gigamax” cache protocol



- Bus snooping maintains local consistency
- Message passing protocol for global consistency

Protocol example



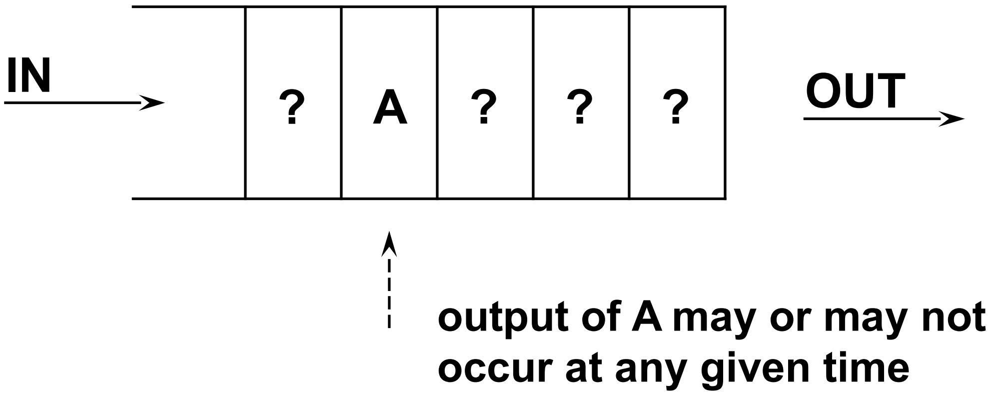
- Cluster B read --> cluster A
- Cluster A response --> B and main memory
- Clusters A and B end shared

Protocol correctness issues

- Protocol issues
  - deadlock
  - unexpected messages
  - liveness
- Coherence
  - each address is sequentially consistent
  - store ordering (system dependent)
- Abstraction is relative to properties specified

One-address abstraction

- Cache replacement is non-deterministic
- Message queue latency is arbitrary



Specifications

- Absence of deadlock

```
SPEC AG (EF p.readable & EF p.writable);
```

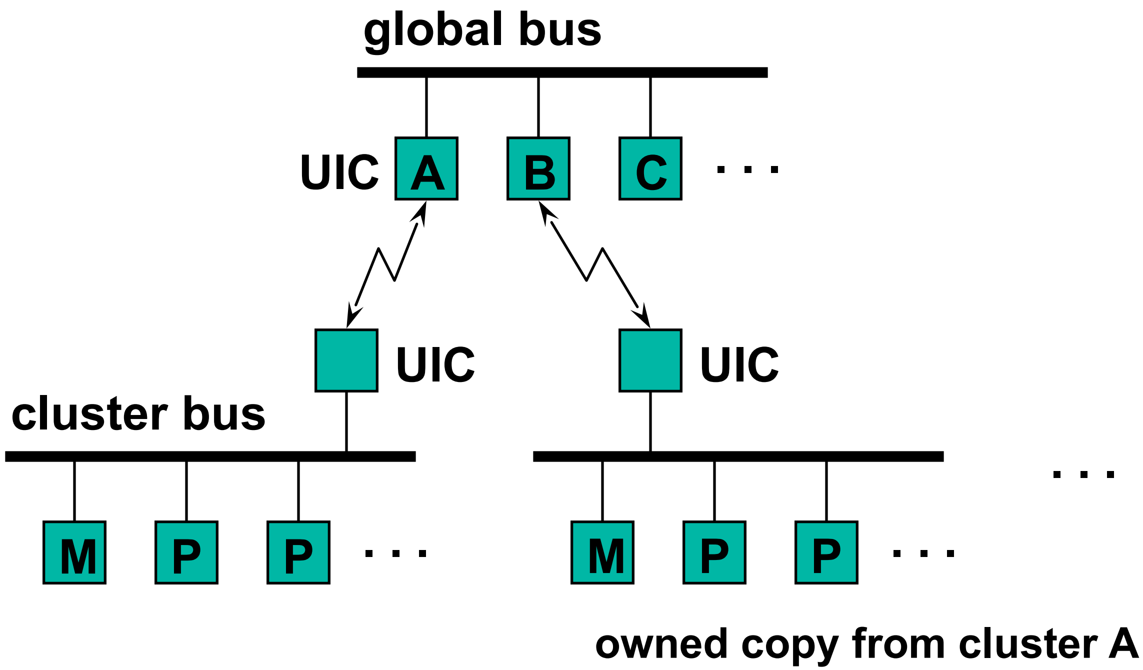
- Coherence

```
SPEC AG((p.readable & bit ->
             ~EF(p.readable & ~bit));
```

Abstraction:

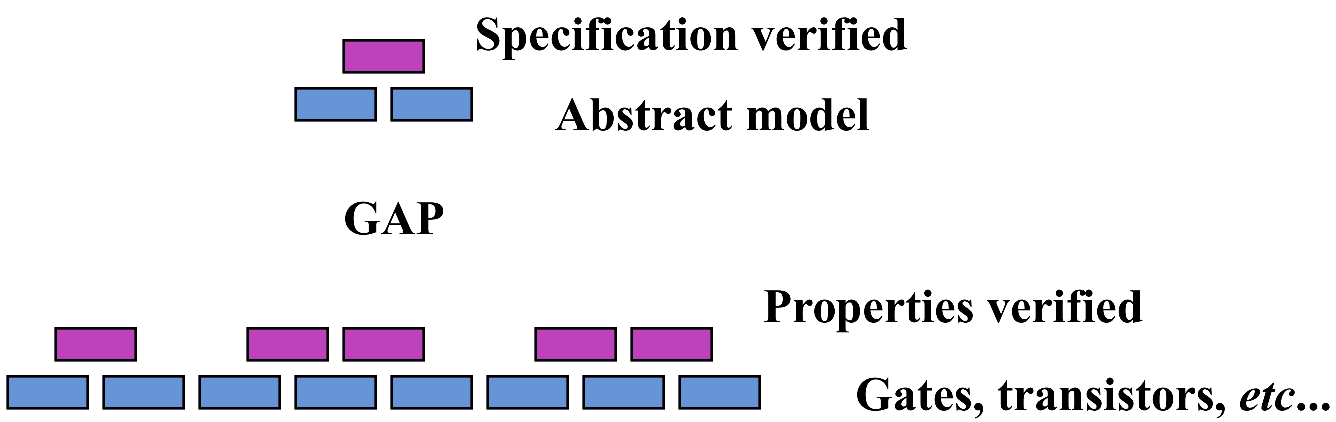
$$\text{bit} = \begin{cases} 0 & \text{if data} < n \\ 1 & \text{otherwise} \end{cases}$$

Counterexample: deadlock in 13 steps



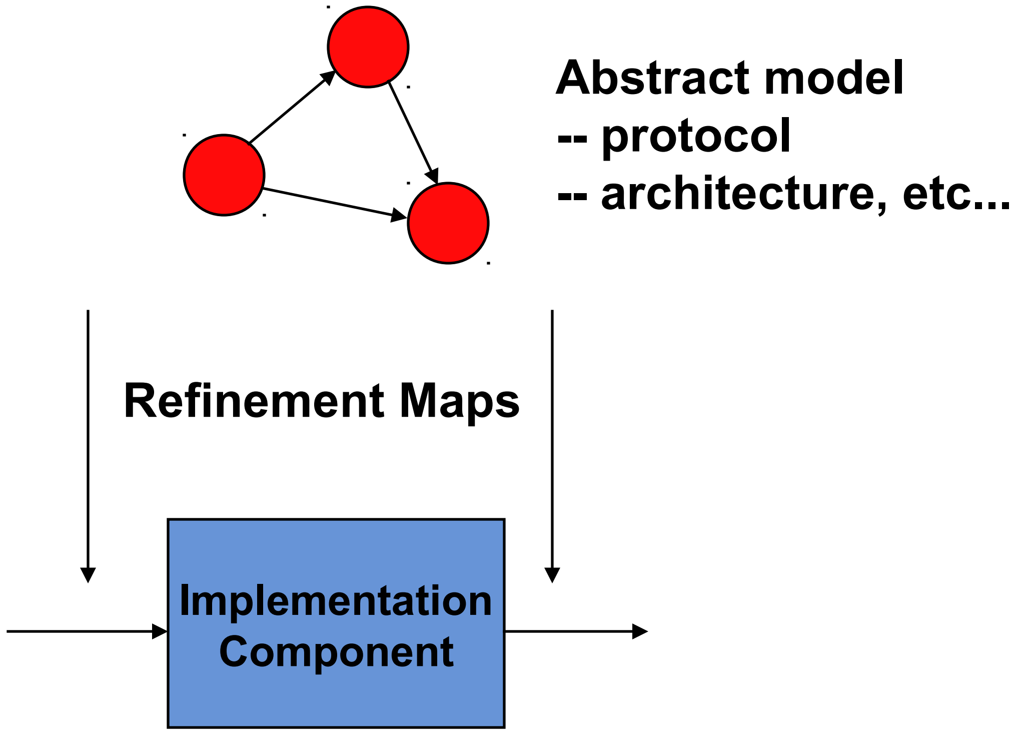
- Cluster A read --> global (waits, takes lock)
- Cluster C read --> cluster B
- Cluster B response --> C and main memory
- Cluster C read --> cluster A (takes lock)

Abstract modeling



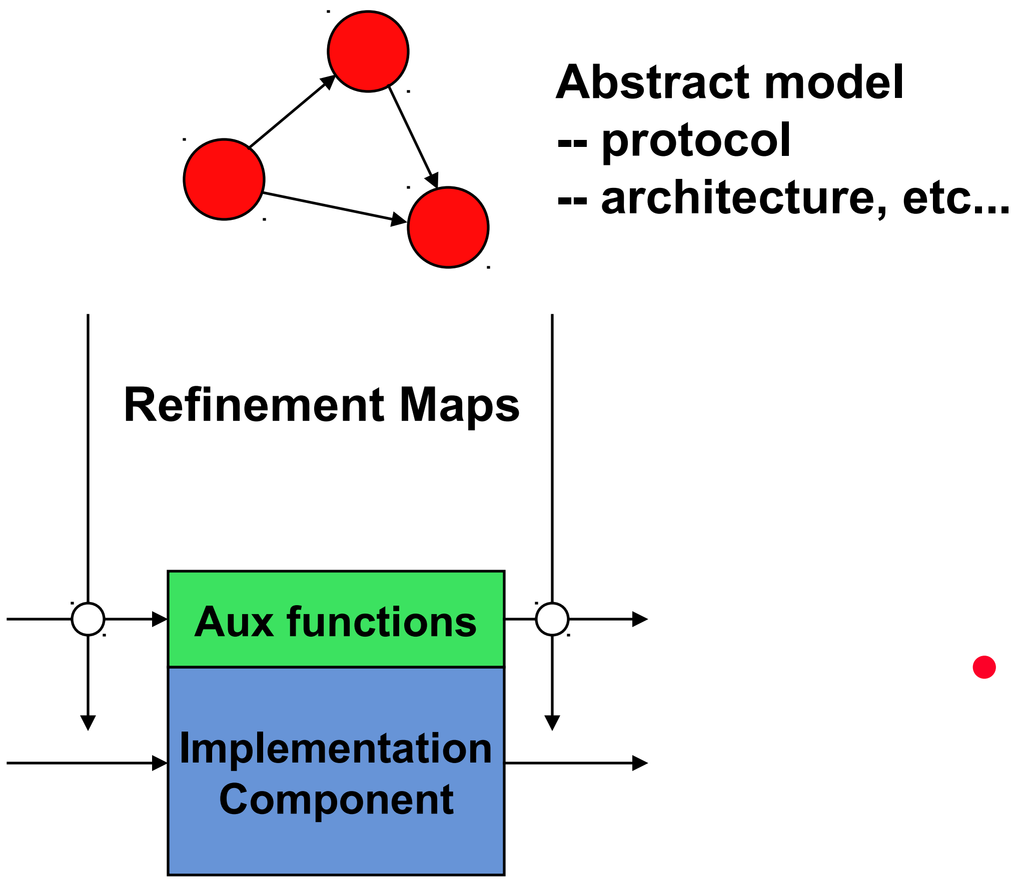
- Model entire system as finite state machine
  - Verify system-level properties
  - Separate protocol/implementation issues
  - Can precede actual implementation
- Doesn't guarantee implementation correctness

Refinement maps



- Maps translate abstract events to implementation level
- Allows verification of component in context of abstract model

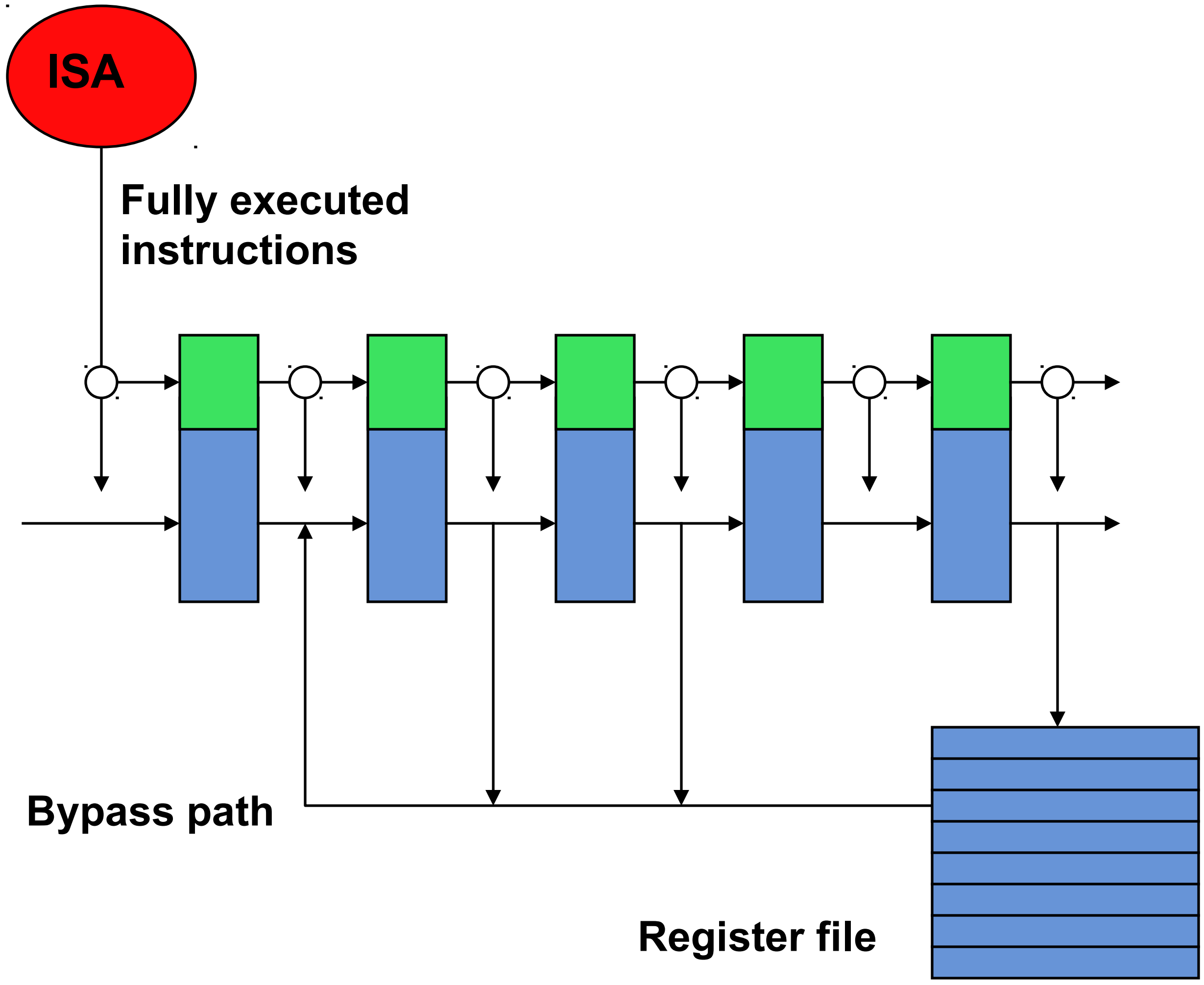
Auxiliary signals



- Imaginary signals:
  - identifying tags
  - future values

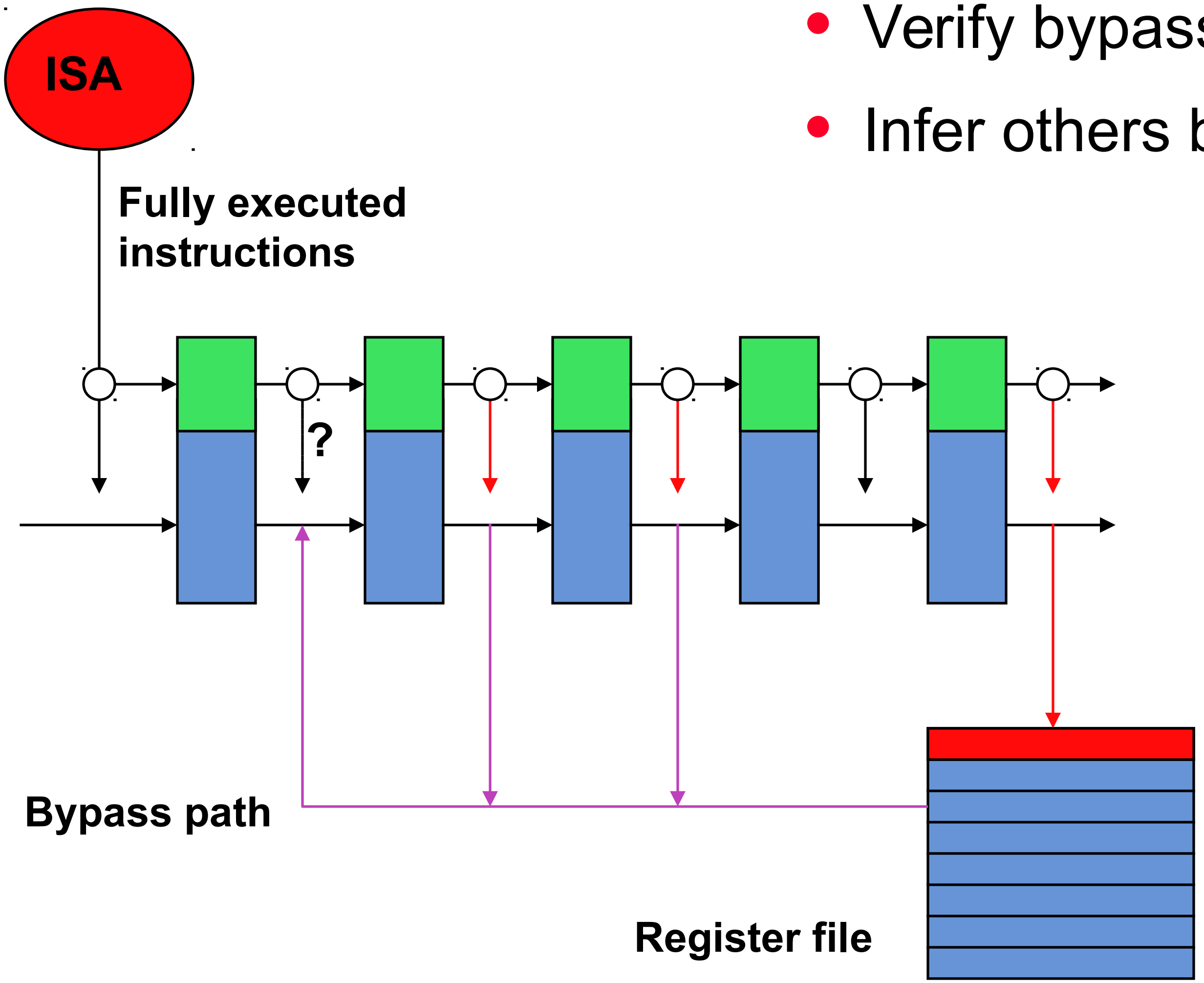
  to relate high/low level

Example -- pipelines

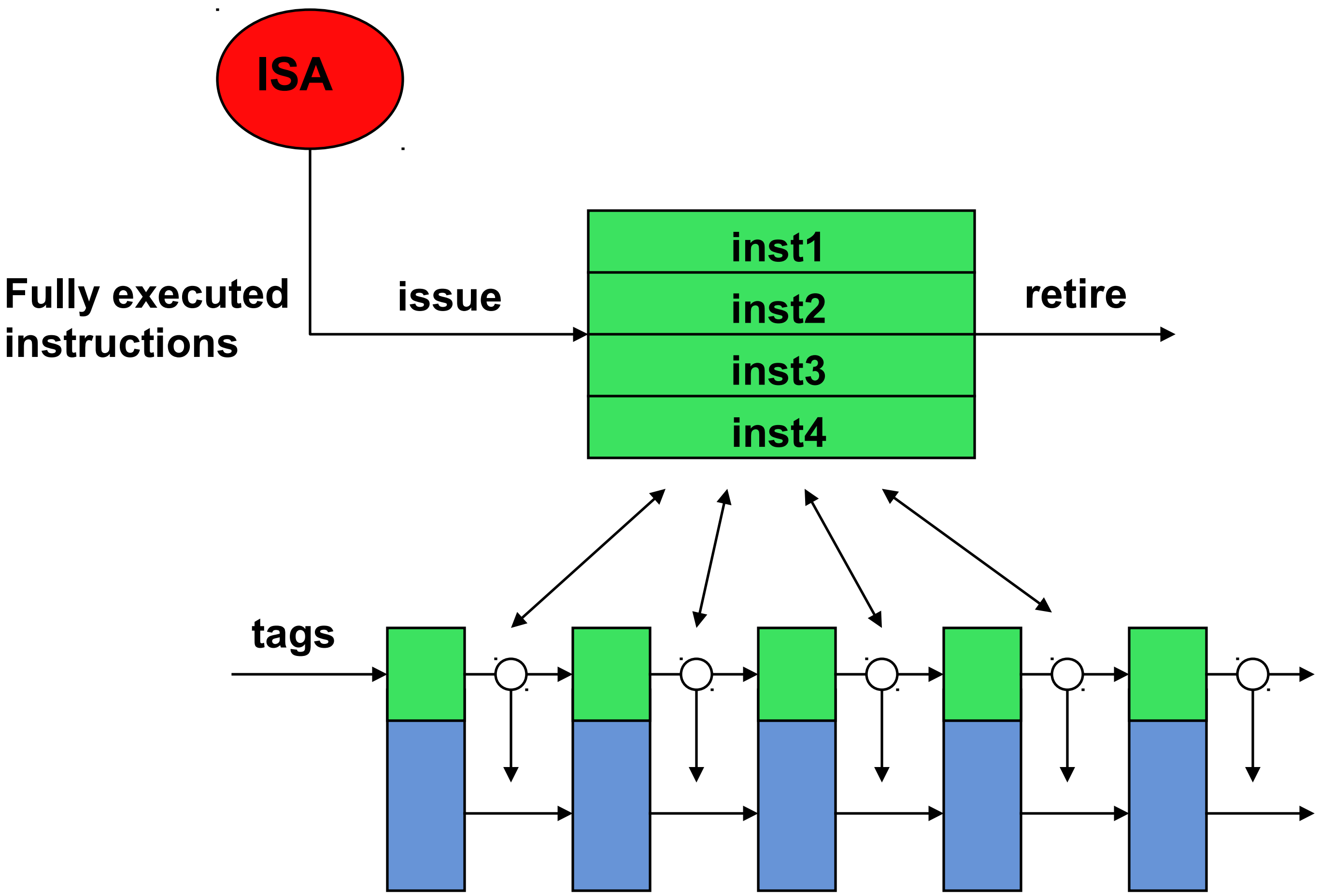


Decomposition

- Verify bypass for register 0
- Infer others by symmetry

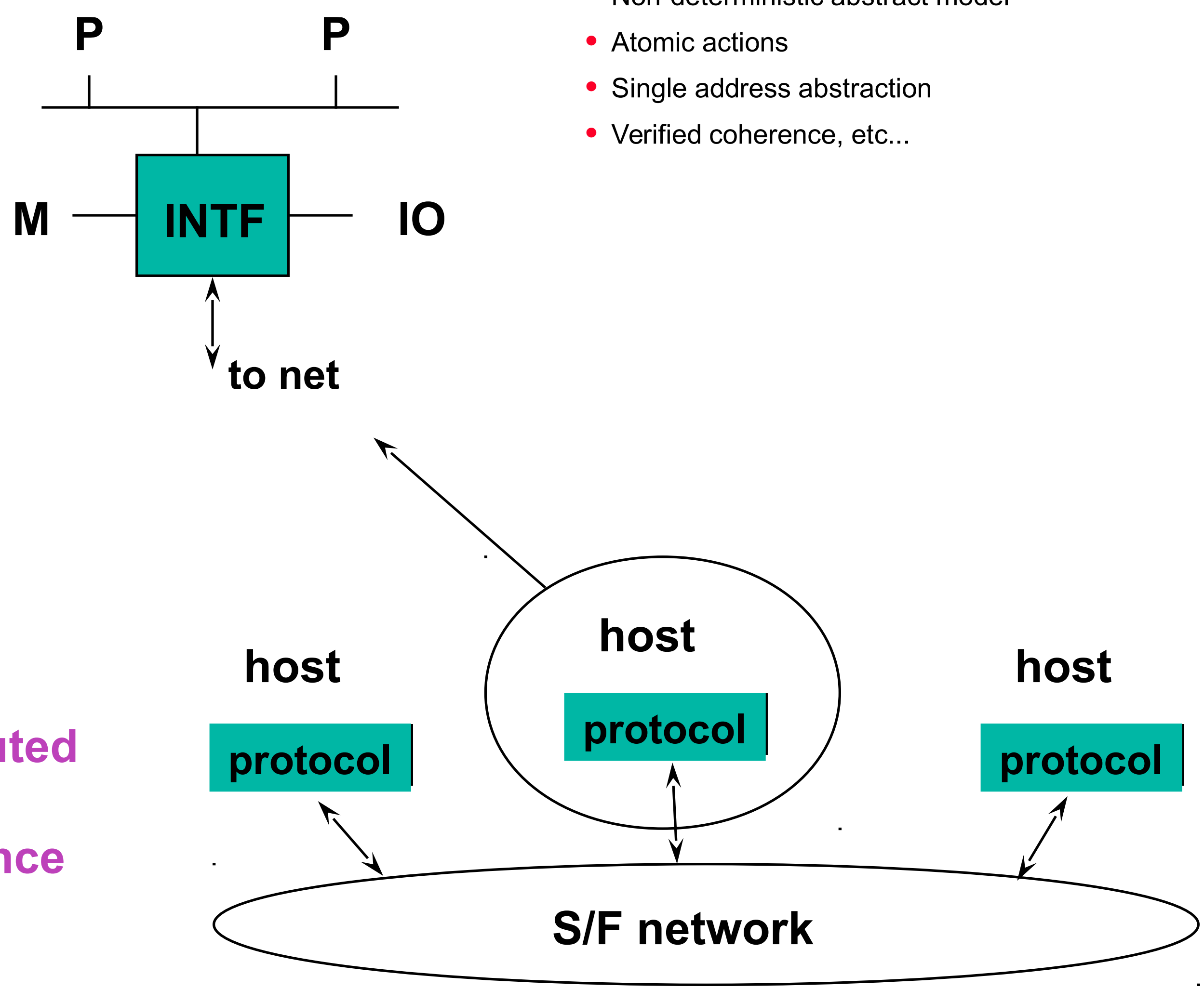


Out of order processors



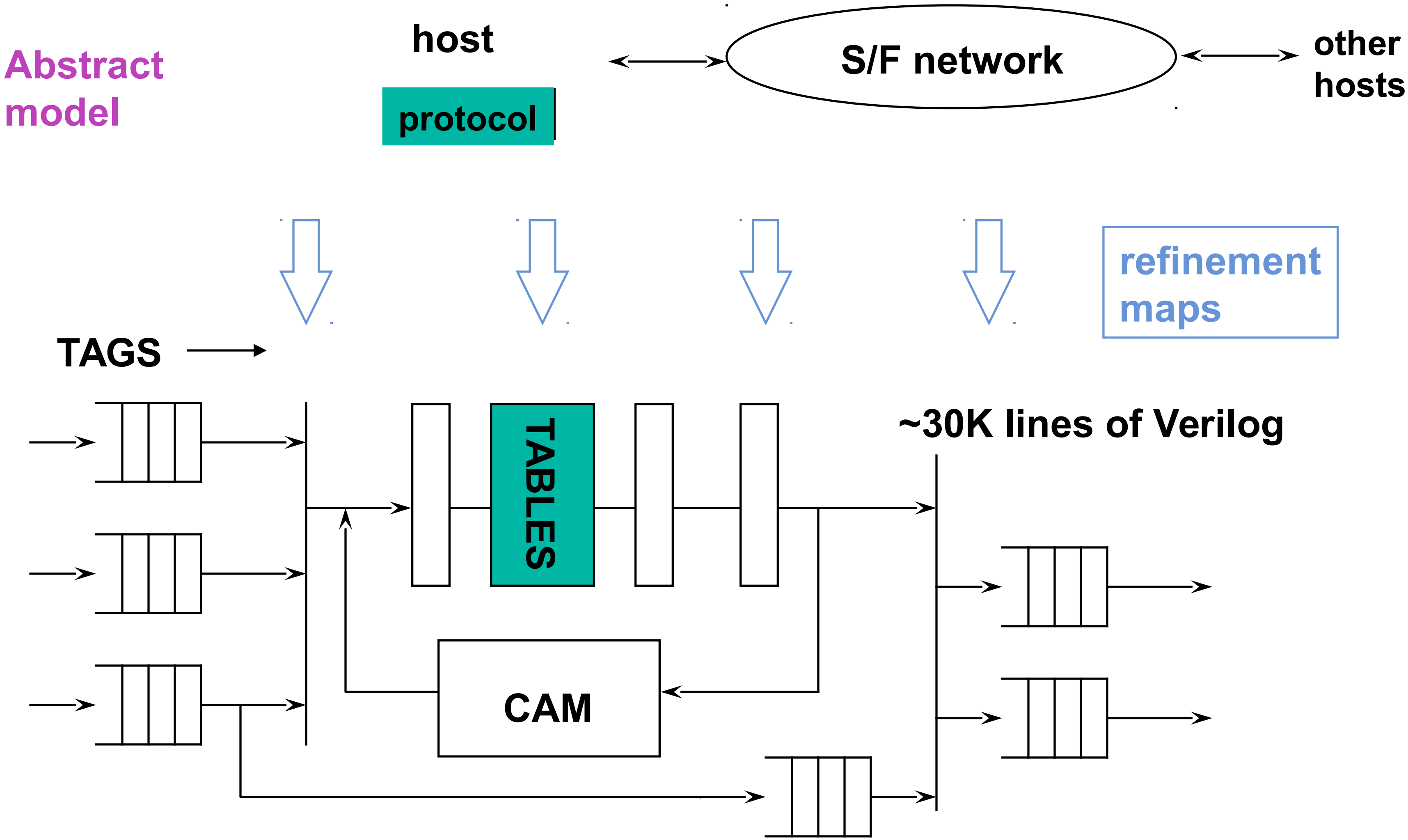
Refinement of cache protocol

- Non-deterministic abstract model
- Atomic actions
- Single address abstraction
- Verified coherence, etc...

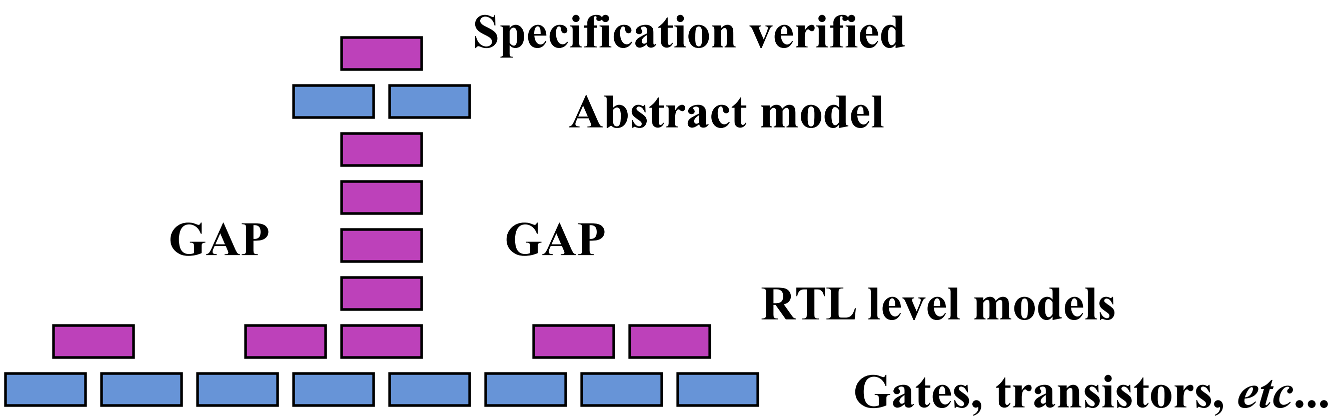


Distributed cache coherence

Mapping protocol to RTL



Local refinement verification



- Specifying refinement maps allows
  - use of abstract model as verification context
  - explicit interface definitions (can transfer to simulation)
  - formal verification of RTL units, without vectors
- System correctness at RTL level not guaranteed

And note, this is not a highly automated process...

Summary

- Basic specification and verification techniques
  - Temporal logic model checking
  - Finite automata
  - Symbolic simulation
- Application at different levels
  - Local property verification
  - Abstract model verification
  - Local refinement verification
- Benefits
  - Find design errors (negative results)
  - Make assumptions explicit
  - Systematically rule out classes of design errors