Model Checking

Randal E. Bryant
CS 740
Nov. 17, 1998

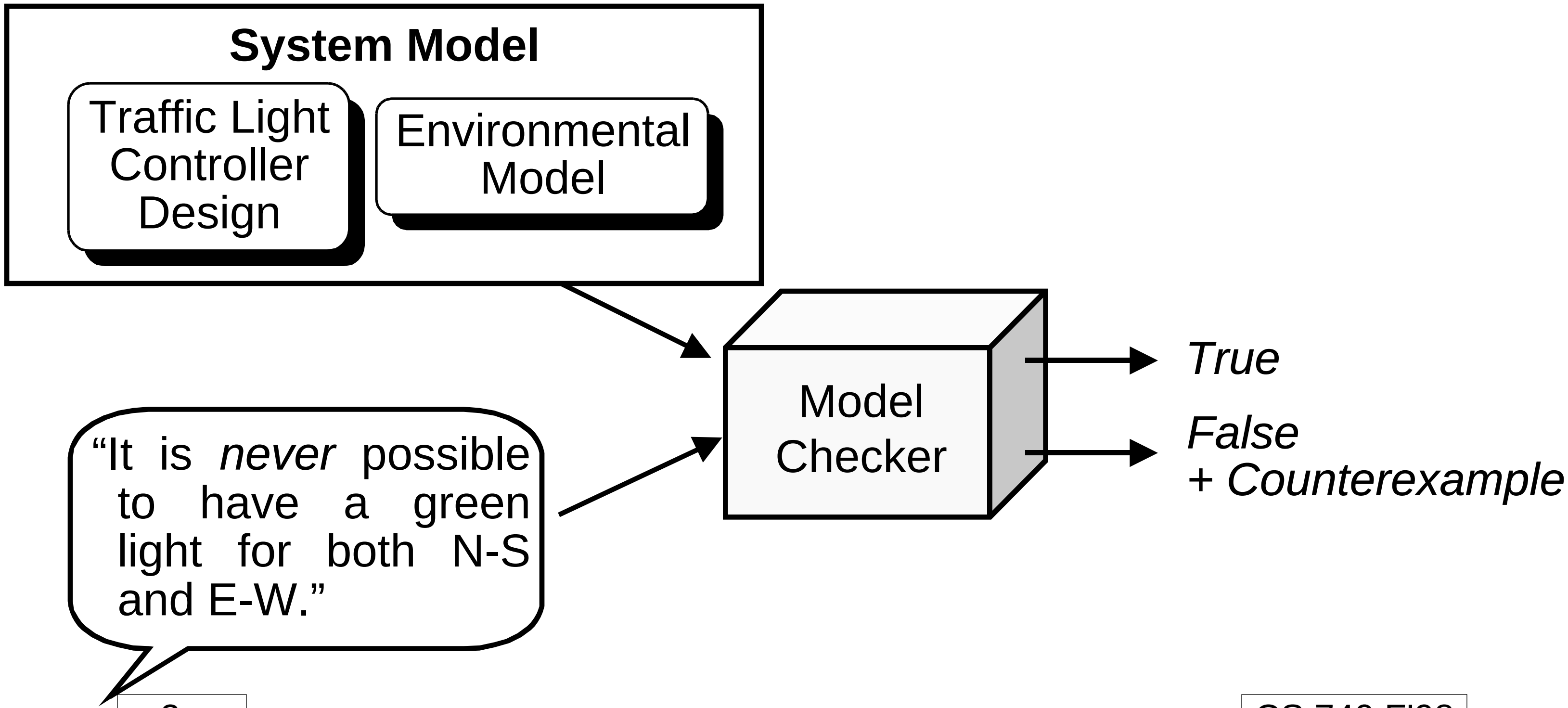
Topics

- **Basics**
  - Model Construction
  - Writing specifications in temporal logic
  - Debugging
- **Model for bus-based cache system**
- **How SMV works**

Reactive System Verification

Temporal Logic Model Checking

- **Construct state machine representation of reactive system**
  - Nondeterminism expresses range of possible behaviors
  - “Product” of component state machines
- **Express desired behavior as formula in temporal logic**
- **Determine whether or not property holds**



System Model

Traffic Light Controller Design

Environmental Model

Model Checker

True

False
+ Counterexample

“It is *never* possible to have a green light for both N-S and E-W.”

Verification with SMV

Language

- **Describe system as hierarchy of modules**
  - Operate concurrently
  - Possibly nondeterministic
- **Describe operating environment as nondeterministic process**
- **Express desired properties by temporal logic formulas**

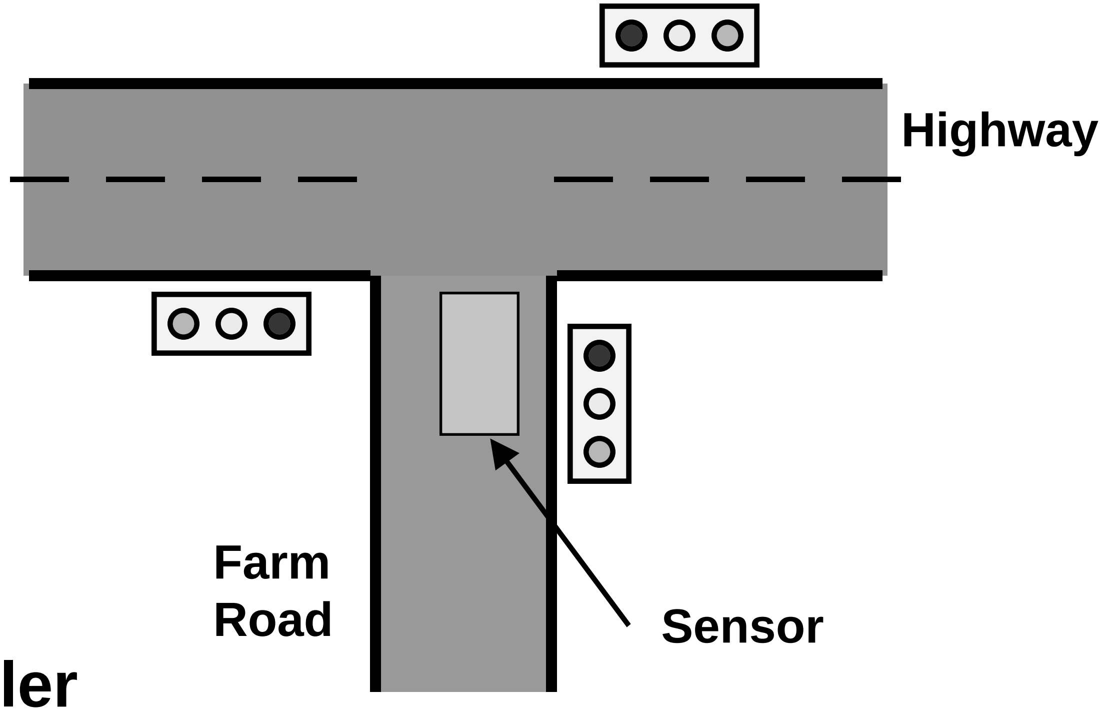
Verifier

- **Constructs BDD representation of system transition relation**
- **Determines whether specification formula satisfied**
  - Generates counterexample if not

Applications

- **Able to verify systems with large (> 10^{20}) state spaces**
- **Widespread interest by industry and researchers**

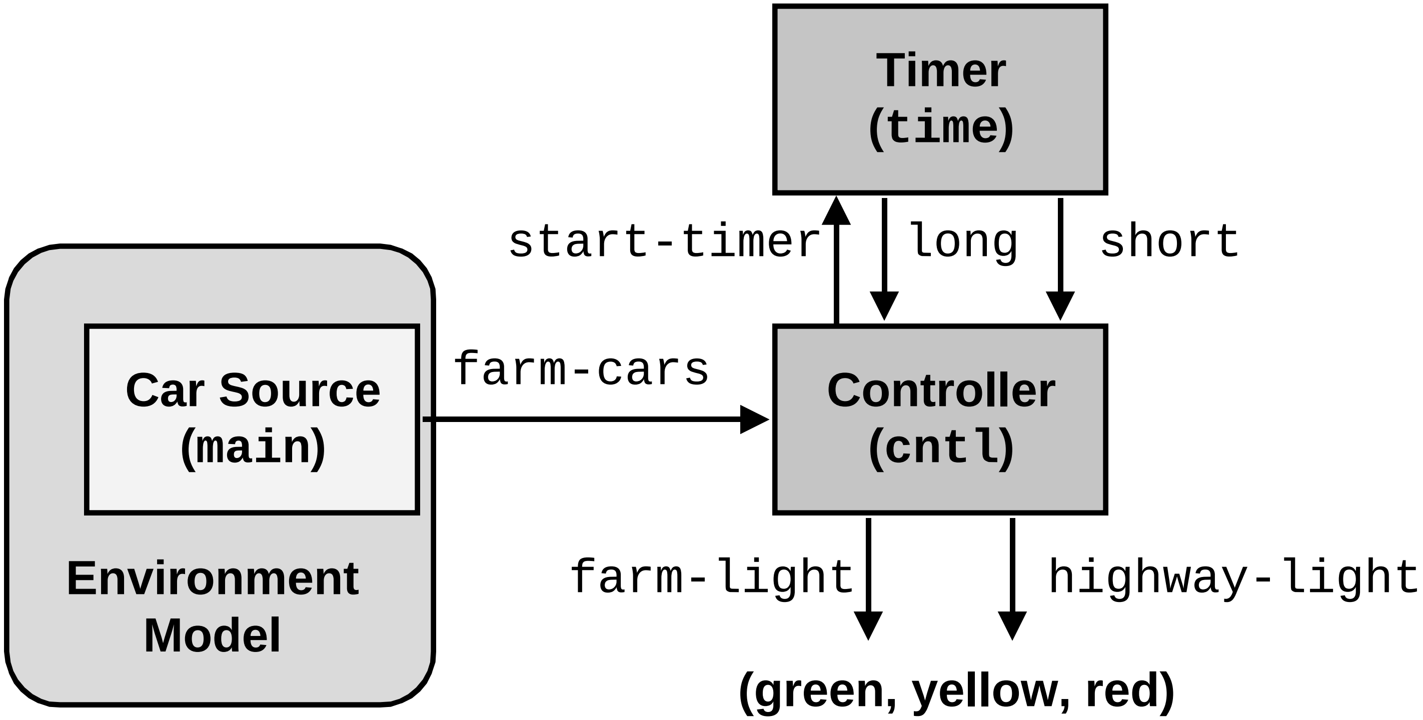
System Example



Traffic Light Controller

- **Mead & Conway, *Introduction to VLSI Systems***
- **Allow highway light(s) to remain green indefinitely**
- **When car sensed on farm road**
  - Wait for delay
  - Cycle to green
  - Hold green until no cars or until maximum delay reached

Model Structure



Model Closed System

- Environment model
- Model of system being verified

Modular Structure

- Each module a (nondeterministic) state machine
- Interacts with other modules via signals

Traffic Controller Main Module

```
-- WARNING: This version has bug(s)

MODULE main
VAR
  farm-cars : boolean;
  cntl : controller(farm-cars, time.long, time.short);
  time : timer(cntl.start-timer);
ASSIGN
  init(farm-cars) := 0;
  -- Nondeterministic driving!
  next(farm-cars) :=  { 0, 1 };
```

State Variables

- **Declared for each module**
  - Boolean (0/1), enumerated, or (finite) integer range
- **Can assign initial and next state**
  - `init(x) :=` …
  - `next(x) :=` …
- **Can reference current and next state**
  - … `:= x`
  - … `:= next(x)`

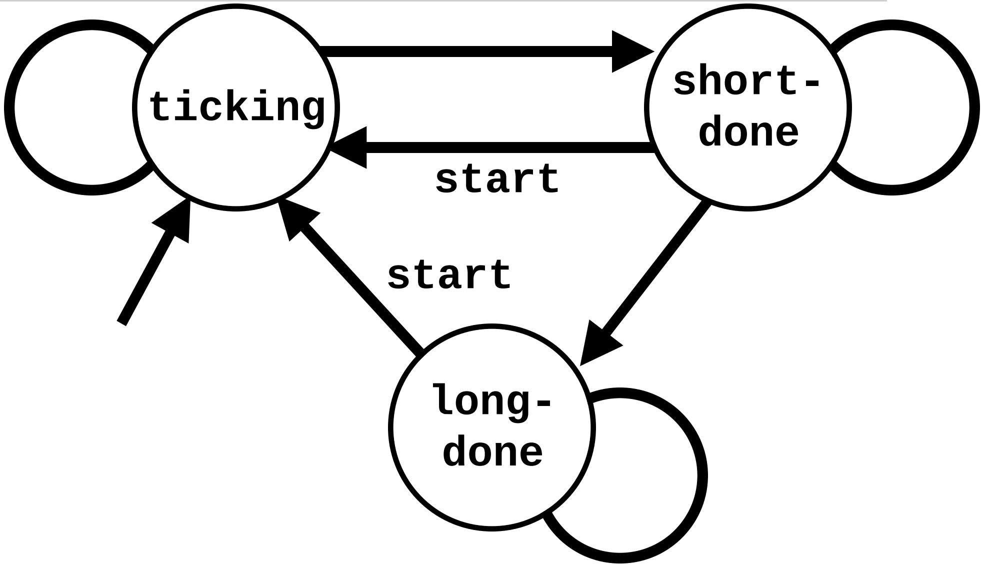
Nondeterministic assignment

- Next state can be any element in set

Timer Module

```
MODULE timer(start)
VAR
  state : { ticking, short-done, long-done };
ASSIGN
  init(state) := long-done;
  next(state) :=
    case
      start : ticking;
      state = ticking : { ticking, short-done };
      state = short-done : { short-done, long-done };
      1 : state;
    esac;
```

- Does not explicitly model time
- Progresses through sequence: ticking, short-done, long-done
- Start acts as reset signal



Timer Module (cont).

```
MODULE timer(start)
VAR
  state : { ticking, short-done, long-done };
ASSIGN

                           • • •

DEFINE
  short := state = short-done;
  long := state = long-done;
```

Defined Signals

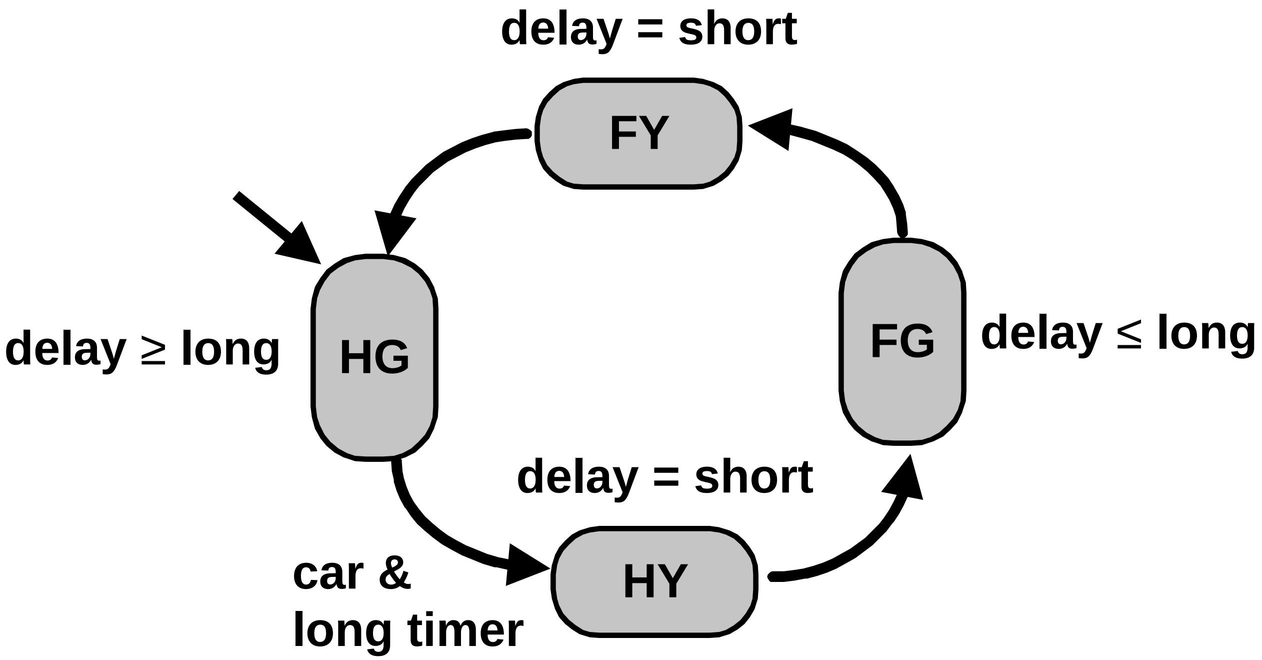
- **Expressions in terms of state variables**
- **Do not introduce additional state variables**
- **More efficient than adding state**

Controller Module

```
MODULE controller(cars, long, short)
VAR
  -- state
  state :
    { highway-yellow, highway-green, farm-yellow, farm-green };
  start-timer : boolean;
  -- outputs
  farm-light : {green, yellow, red};
  highway-light : {green, yellow, red};
```

Controller Module State

```
init(state) := highway-green;
  next(state) :=
    case
      state = highway-green & cars & long : highway-yellow;
      state = highway-yellow & short : farm-green;
      state = farm-green & (!cars | long) : farm-yellow;
      state = farm-yellow & short : highway-green;
      1 : state;
    esac;
```



SMV Case Statement

```
  next(var) :=
    case
      cond1 : expr1;
      cond2 : expr2;
      1 : expr-default;
    esac;
```

- **Sequence of condition / result pairs**
- **First one to match is used**

Other Controller Signals

```
start-timer :=
    state = highway-green & cars & long |
    state = highway-yellow & short |
    state = farm-green & (!cars | long) |
    state = farm-yellow & short;
farm-light :=
  case
    state = farm-yellow : yellow;
    state = farm-green : green;
    1 : red;
  esac;
highway-light :=
  case
    state = highway-yellow : yellow;
    state = highway-green : green;
    1 : red;
  esac;
```

- **Probably should implement as define’s**
  - –Directly assigning current state

Writing Specification

Safety Property

- **“Bad things don’t happen”**
- **Either the farm road or the highway always has a red light**

```
AG (cntl.farm-light = red | cntl.highway-light = red)
```

Liveness Property

- **“Good things happen eventually”**
- **If a car appears on the farm road, it will eventually get a green light**

```
AG (farm-cars -> AF cntl.farm-light = green)
```

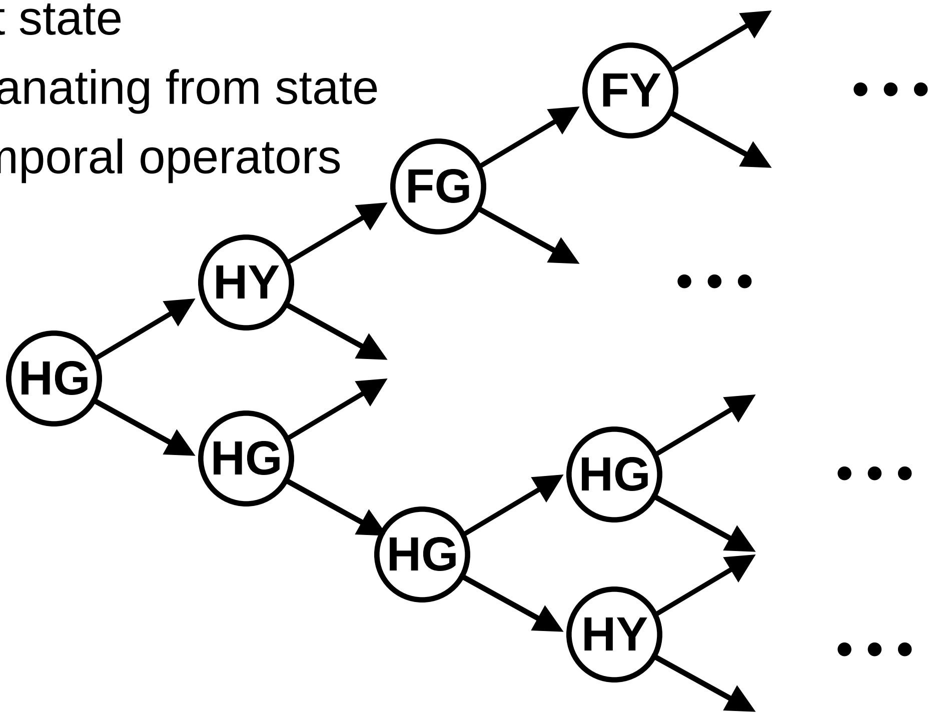
- **The highway light turns green infinitely often**

```
AG (AF cntl.highway-light = green)
```

Computation Tree Logic

Concept

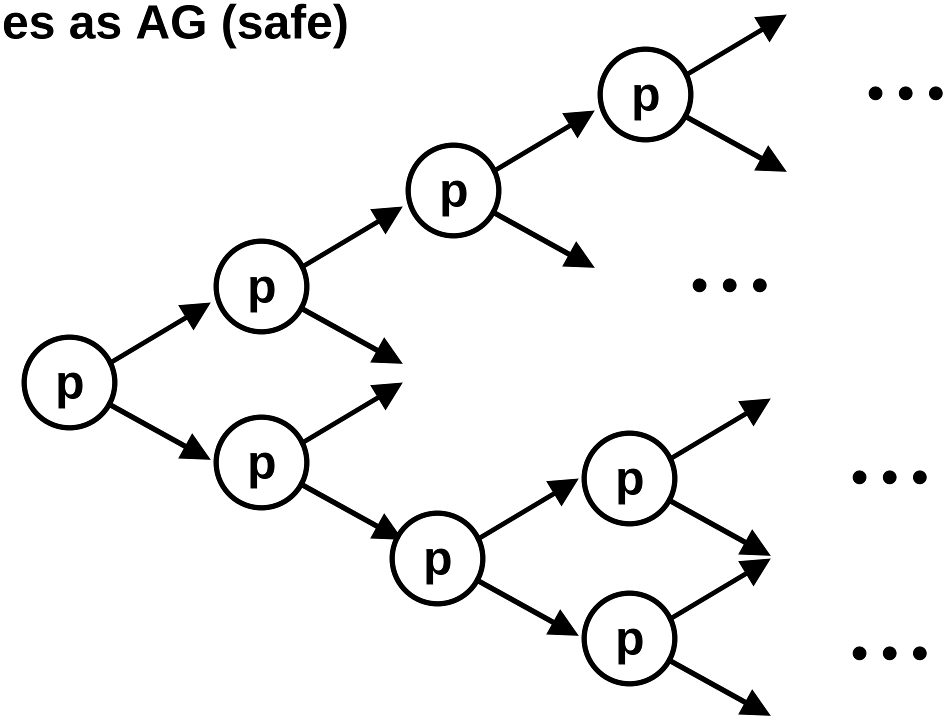
- **Consider unrolling of state graph into infinite tree**
- **Express formulas for state at some node of tree**
  - Usual Boolean connectives
    - » Properties of current state
  - Properties of paths emanating from state
    - » Expressed using temporal operators



Temporal Operators

Always-Globally

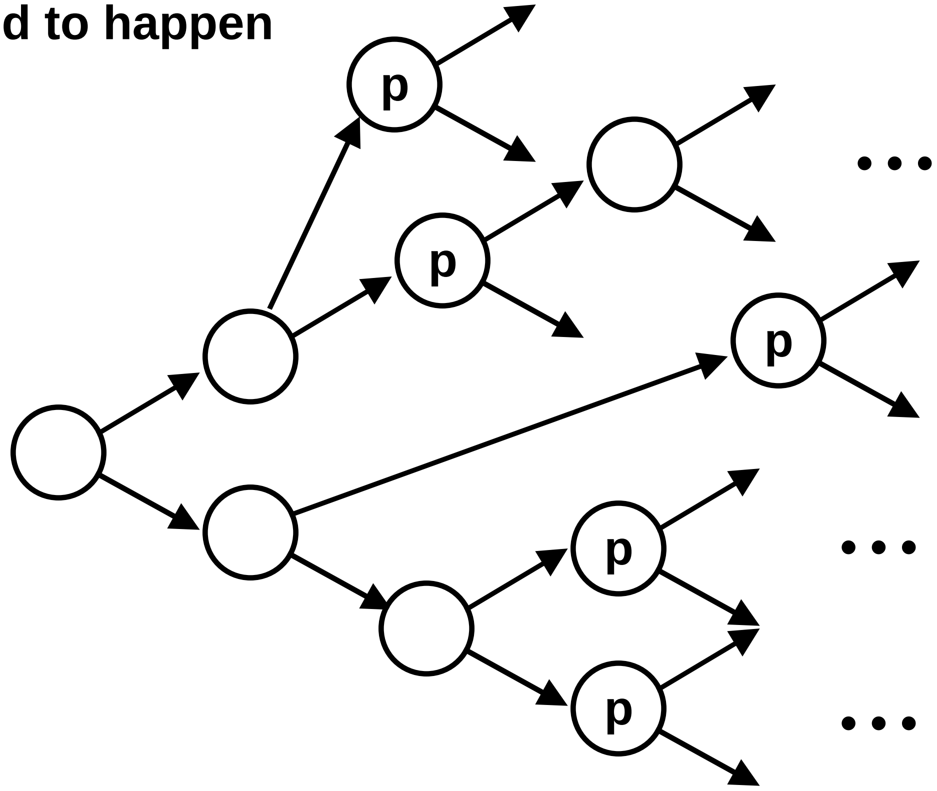
- AG p
- p holds now and forever more
- Regardless of nondeterministic choices
- Express safety properties as AG (safe)



Temporal Operators (cont).

Always-Eventually

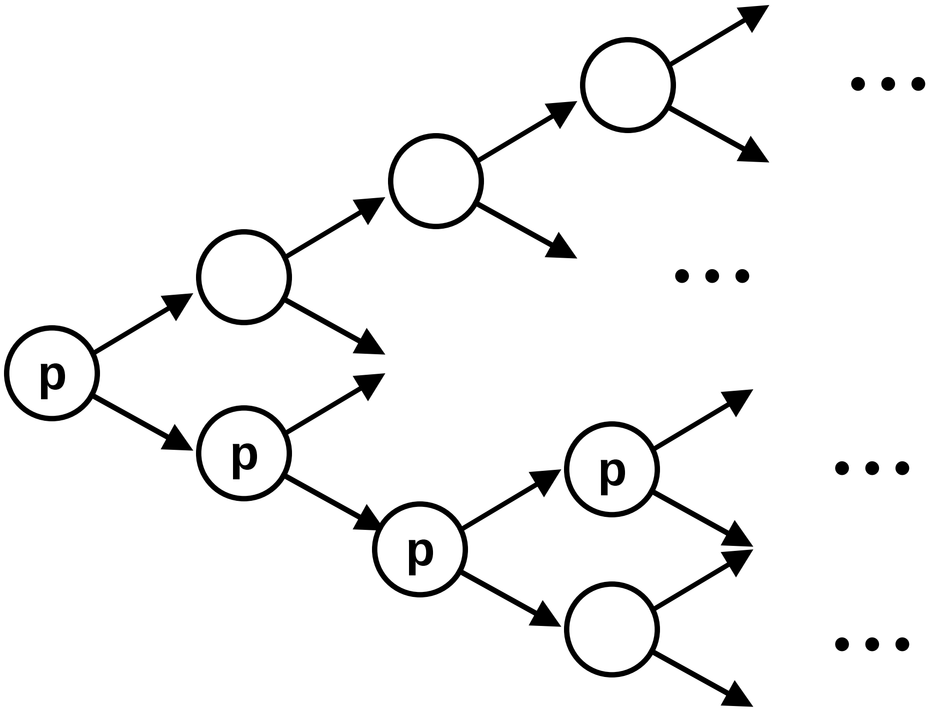
- AF p
- Along every path, p holds somewhere
- Something is guaranteed to happen



Derived Temporal Operators

Possibly Globally

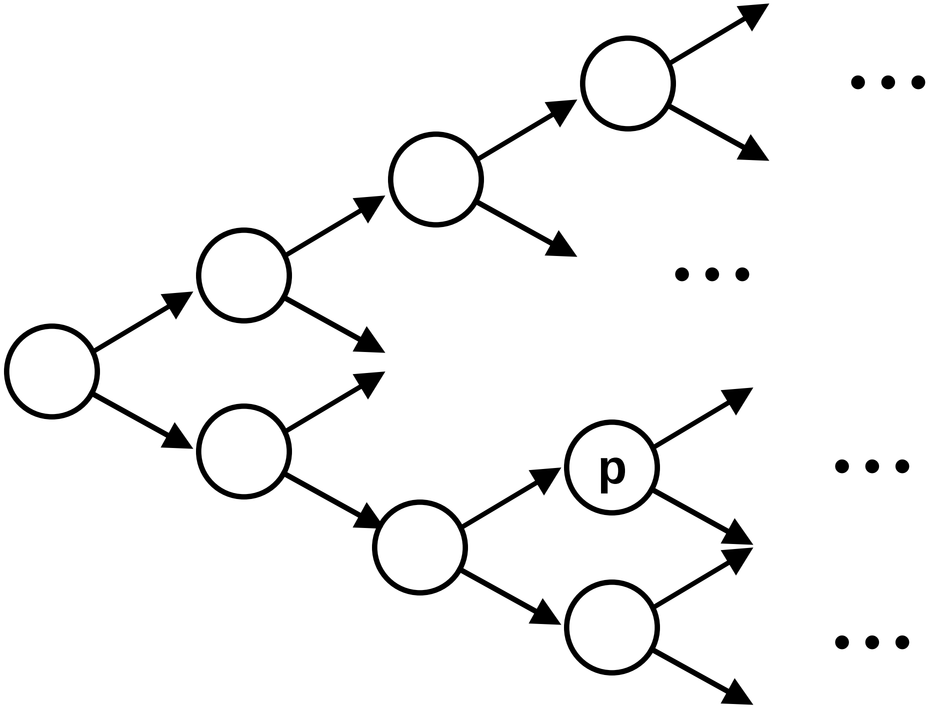
- **EG p**
- **There is some path for which p continually holds**
- **EG p == !AF !p**



Derived Operators (cont).

Possibly Eventually

- EF p
- p holds at some point, as long as correct nondeterministic choices are made
- EF p == ! AG !p

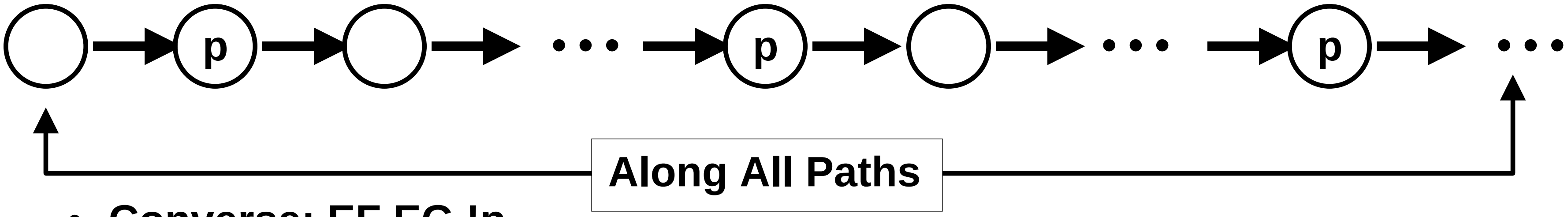


Nested Temporal Operations

- Express properties of paths emanating from states along paths
- Can become hopelessly obscure
- If formula is too complex, it’s almost certainly not what you want, anyhow

Useful Case

- AG AF p
- At any time, p must hold in the future
- p holds infinitely often



- Converse: EF EG !p
  - There is some point in the future, such that from that point onward it is possible for p never to hold

Traffic Light Specification

Safety Property

- "Bad things don't happen"
- Either the farm road or the highway always has a red light

```
AG (cntl.farm-light = red | cntl.highway-light = red)
```

Liveness Property

- "Good things happen eventually"
- If a car appears on the farm road, it will eventually get a green light

```
AG (farm-cars -> AF cntl.farm-light = green)
```

- The highway light turns green infinitely often

```
AG (AF cntl.highway-light = green)
```

SMV Run #1

```
-- specification AG (cntl.farm-light = red | cntl.highway...
-- is false
-- as demonstrated by the following execution sequence
state 1.1:
farm-cars = 0
cntl.state = highway-green
cntl.start-timer = 0
cntl.farm-light = red
cntl.highway-light = green
time.long = 1
time.short = 0
time.state = long-done

state 1.2:
farm-cars = 1
cntl.start-timer = 1

-- loop starts here --
state 1.3:
farm-cars = 0
cntl.state = highway-yellow
cntl.start-timer = 0
cntl.highway-light = yellow
time.long = 0
time.state = ticking
```

Counterexample Facility

- **Shows trace indicating case for which specification is false**
  - Path to state violating safety property
  - Path to cyclic condition violating liveness condition

First Bug Found

- **Timer hung up in “ticking” state**
- **Nothing forces time to progress**

Fixing Timer

```
VAR
  state : { ticking, short-done, long-done };
  progress : boolean;
ASSIGN
  init(state) := long-done;
  next(state) :=
    case
      start : ticking;
      !progress : state;
      state = ticking : short-done;
      state = short-done : long-done;
      1 : state;
    esac;
  next(progress) := {0, 1};
DEFINE
  short := state = short-done;
  long := state = long-done;
FAIRNESS
  progress
```

Modified State

- **Variable `progress` forces transition**
- **Set nondeterministically**

Fairness Property

- **Condition that must hold infinitely often**
- **Model checker considers only fair paths**
- **Timer must keep making progress**
- **Can't reach some point where it stops altogether**

SMV Run #2

- Yields 11 state sequence followed by 3 state loop

Counterexample Condition

- **Farm car #1 approaches, triggering light cycle**
- **Farm car #1 disappears before farm light turns green**
  - Controller designed before right-on-red legal?
- **Farm car #2 appears & disappears at yellow light**
- **Light cycle completes**
- **Highway light stays green indefinitely**

Violated Condition

```
AG (farm-cars -> AF cntl.farm-light = green)
```

- **Didn’t hold for Farm car #2**
- **Went through yellow light**

Specification Fix #1

```
AG (farm-cars -> AF cntl.farm-light in
      { green, yellow })
```

- **Consider yellow light to be good enough**

Counterexample

- **Irrelevant stuff:**
  - Farm car #1 approaches, triggering light cycle
  - Farm car #1 disappears before farm light turns green
  - Light cycle completes
- **Farm car #2 appears, but disappears before long timer interval**
- **Highway light stays green indefinitely**

Violated Condition

- **Farm car #2 never had green or yellow light**

Ways to Fix

Car Fix

```
init(farm-cars) := 0;
next(farm-cars) :=
  case
    -- Wait until light is green
    farm-cars & cntl.farm-light = red : 1;
    1 : {0, 1};
  esac;
```

- **Farm car must stay there as long as light is red**
- **Verifies, but makes strong assumption about environment**

Specification Fix #2

```
AG AF (farm-cars -> cntl.farm-light in
      { green, yellow })
```

- **If a farm car is persistent, it will eventually be allowed to go**

Snoopy Bus-Based Consistency

Caches

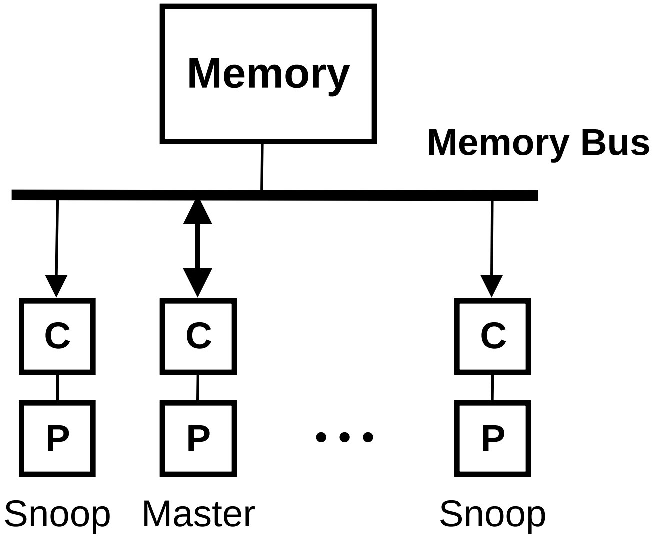
- **Write-back**
  - –Minimize bus traffic
- **Monitor bus transactions when not master**

Cached blocks

- **Clean block can have multiple, read-only copies**
- **To write, must obtain exclusive copy**
  - –Marked as dirty

Getting copy

- **Make bus request**
- **Memory replies if block clean**
- **Owning cache replies if dirty**



Simplifications Made in SMV Model

Single Cache Line

- **No loss of generality, since different cache lines don't interact**
  - Except if some interaction within associative set

Two Tag Values

- **Oversimplification**

Three Processors

- **Oversimplification**

Model Control Only

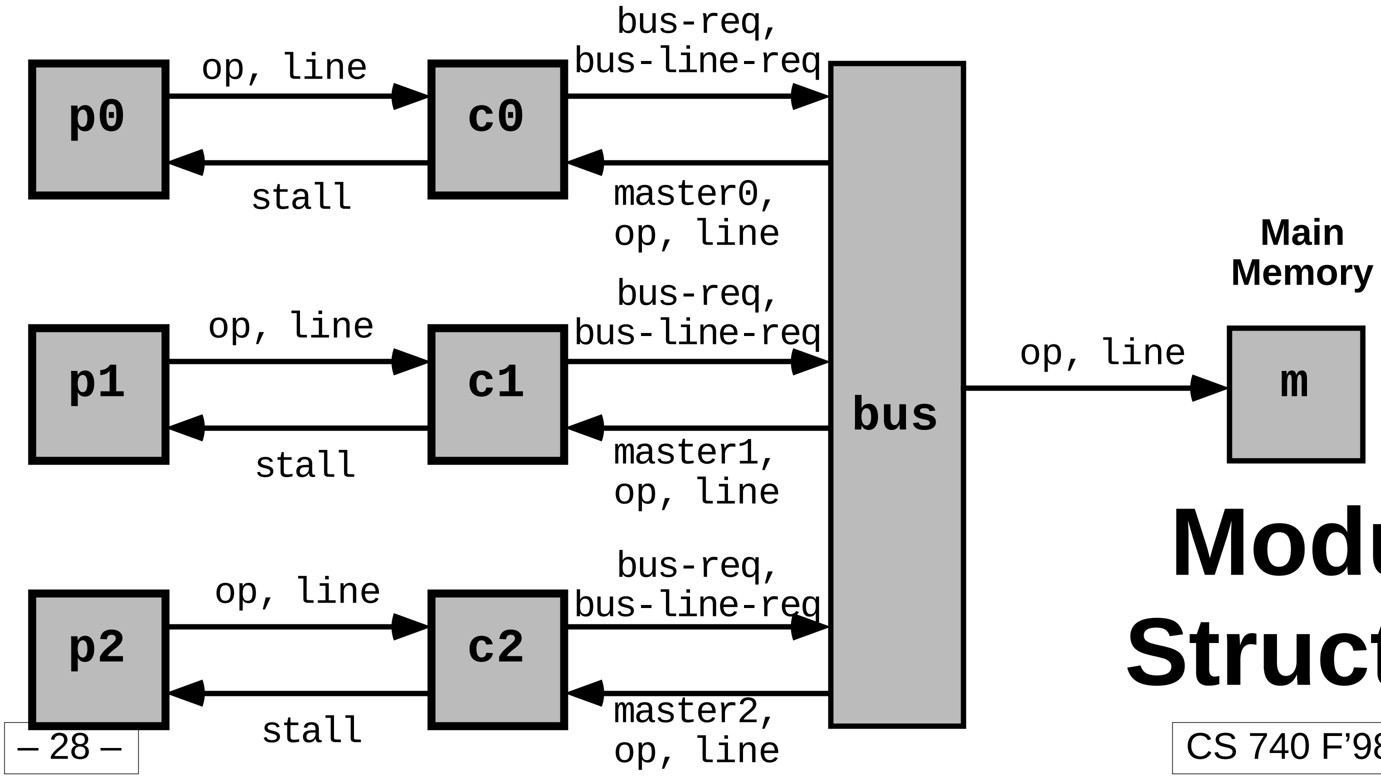
- **No data or data transfers**

Simplistic Processor Model

- **Issues arbitrary sequence of reads, writes, and no-ops**
- **Provides environment model**
- **Captures full generality of operating environment**

Module Structure

```
MODULE main
VAR
  bus : bus(c0.bus-req, c0.bus-line-req,
            c1.bus-req, c1.bus-line-req,
            c2.bus-req, c2.bus-line-req);
  c0 : cache(p0.op, p0.line, bus.master0, bus.op, bus.line);
  p0 : processor(c0.stall);
  c1 : cache(p1.op, p1.line, bus.master1, bus.op, bus.line);
  p1 : processor(c1.stall);
  c2 : cache(p2.op, p2.line, bus.master2, bus.op, bus.line);
  p2 : processor(c2.stall);
  m  : memory(bus.op, bus.line);
```



Processor Module

```
MODULE processor(stall)
VAR
  op : {no-op, read, write };
  line: {lnA, lnB} ;
ASSIGN
  init(op) := no-op;
  next(op) :=
    case
      stall : op;
      !stall : {no-op, read, write};
    esac;
  init(line) := {lnA, lnB} ;
  next(line) :=
    case
      stall : line;
      1 : {lnA, lnB} ;
    esac;
```

- Generates arbitrary sequence of operations to arbitrary addresses
- Holds operation & address persistently as long as stalled

Implementation Details

Bus Operations

- **Read**
  - Get read-only copy
- **XRead**
  - Get writeable copy
  - Like Read + Invalidate
    - » Except that atomic
    - » Required to guarantee eventual success
- **Invalidate**
  - Invalidate all other copies
  - Make local copy writeable
- **Write**
  - Write back dirty block
  - To make room for different block

Operating Principle

- **Every block has “owner”**
- **Responsible for supplying value when needed**

Owned by Main Memory

- **Correct value in main memory**
- **Other copies read-only**
  - May be more than 1 copy

Owned by Cache

- **Held by some cache on behalf of its processor**
  - Allowed to modify
- **Version in memory not valid**
- **Must write back to evict**
- **Must supply if requested by other cache**

Main Memory Module

```
MODULE memory(bus-op, bus-line)
VAR
  ownA : boolean;
  ownB : boolean;
ASSIGN
  init(ownA) := 1;
  next(ownA) :=
    case
      ! bus-line = lnA : ownA;
      -- Gaining ownership
      bus-op = write : 1;
      bus-op = read : 1;
      -- Giving up ownership
      bus-op in {invalidate, xread} : 0;
      1 : ownA;
    esac;
 init(ownB) := 1;
 next(ownB) :=
    • • •
```

Operation

- **Track status of every memory block**
  - Not very realistic
- **Respond to bus requests**
- **A & B blocks handled symmetrically**

Gaining ownership

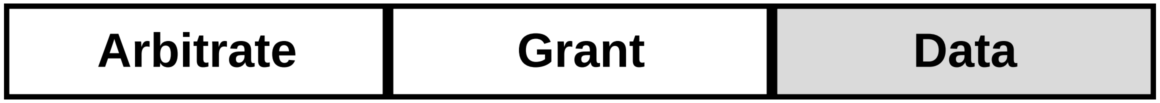
- **when cache writes back**
- **when one cache reads blocked owned by other cache**

Losing ownership

- **Some cache obtains exclusive copy**

Bus Model

Bus Timing



Arbitrate

Grant

Data

- **Arbitrate**
  - Cache controllers specify requested operation & address
- **Grant**
  - Bus designates master & broadcasts requested operation & address
- **Data**
  - Data passed on bus
  - Not modeled in our protocol

Bus State

```
MODULE bus(req0, line0, req1, line1, req2, line2)
VAR
  token : {0, 1, 2}; -- Pass around token
  master : {0, 1, 2, no-one};
  op : {arbitrate, read, xread, write, invalidate, no-op};
  line : {lnA, lnB} ;
```

Token

- Used to guarantee fairness
- Indicates priority among requesters

Master

- Indicates which cache wins arbitration

Op

- “arbitrate” during arbitration phase
- Bus operation during grant phase

Line

- Address for bus operation

Bus Fairness

```
init(token) := {0,1,2};
next(token) :=
  case
    op = arbitrate : token;
    1 : {0, 1, 2};
  esac;
```

```
FAIRNESS
  token = 0
FAIRNESS
  token = 1
FAIRNESS
  token = 2
```

Quasi-Round-Robin

- **Token determines priority**
- **Passed around nondeterministically**
  - on grant phase
- **Everyone guaranteed to get it**

```
init(master) := no-one;
  next(master) :=
    case
      !(op = arbitrate) : no-one;
      -- Arbitrate for new master
      token = 0 :
        case
         !(req0 = no-op) : 0;
         !(req1 = no-op) : 1;
         !(req2 = no-op) : 2;
         1 : no-one;
        esac;
      token = 1 :
        case
         !(req1 = no-op) : 1;
         !(req2 = no-op) : 2;
         !(req0 = no-op) : 0;
         1 : no-one;
        esac;
      1 :        -- token = 2
        case
         !(req2 = no-op) : 2;
         !(req0 = no-op) : 0;
         !(req1 = no-op) : 1;
         1 : no-one;
        esac;
    esac;
```

Bus Operation

```
init(op) := no-op;
  next(op) :=
    case
      !(op = arbitrate) : arbitrate;
      next(master) = 0 : req0;
      next(master) = 1 : req1;
      next(master) = 2 : req2;
      1 : no-op;
    esac;
  init(line) := {lnA, lnB} ;
  next(line) :=
    case
      !(op = arbitrate) : line;
      next(master) = 0 : line0;
      next(master) = 1 : line1;
      next(master) = 2 : line2;
      1 : {lnA, lnB} ;
  esac;
DEFINE
  master0 := master = 0;
  master1 := master = 1;
  master2 := master = 2;
```

Control

- **Alternate between arbitrate & grant phases**
- **During grant, pass on requested operation**

Address

- **During grant, pass requested line**

Cache State

```
MODULE cache(proc-op, proc-line, master, bus-op, bus-line)
VAR
  state : { invalid , clean, dirty, error };
  tag : {lnA, lnB} ;
```

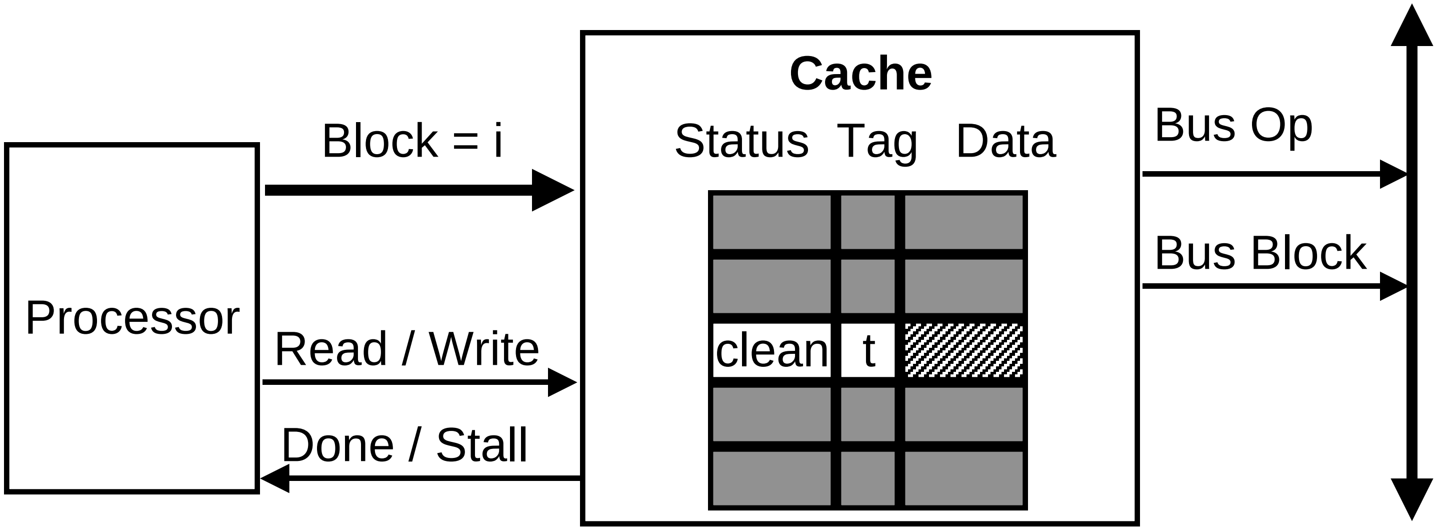
- **Maintained by each cache for each of its blocks**
- **Invalid**
  - Entry not valid
- **Clean**
  - Valid, read-only copy
  - Matches copy in main memory
- **Dirty**
  - Exclusive,writeable copy
  - Must write back to evict
- **Error**
  - Condition that should not arise
  - Added to allow stronger forms of verification

Performing Processor Operations

- **Processor requests cache to perform load or store**
  - – On word in cache block i
- **Cache line currently holds block t**
  - – May or may not have i = t
- **Cache can either:**
  - – Perform operation using local copy
  - – Issue bus request to get block
    - » Stall processor until block ready

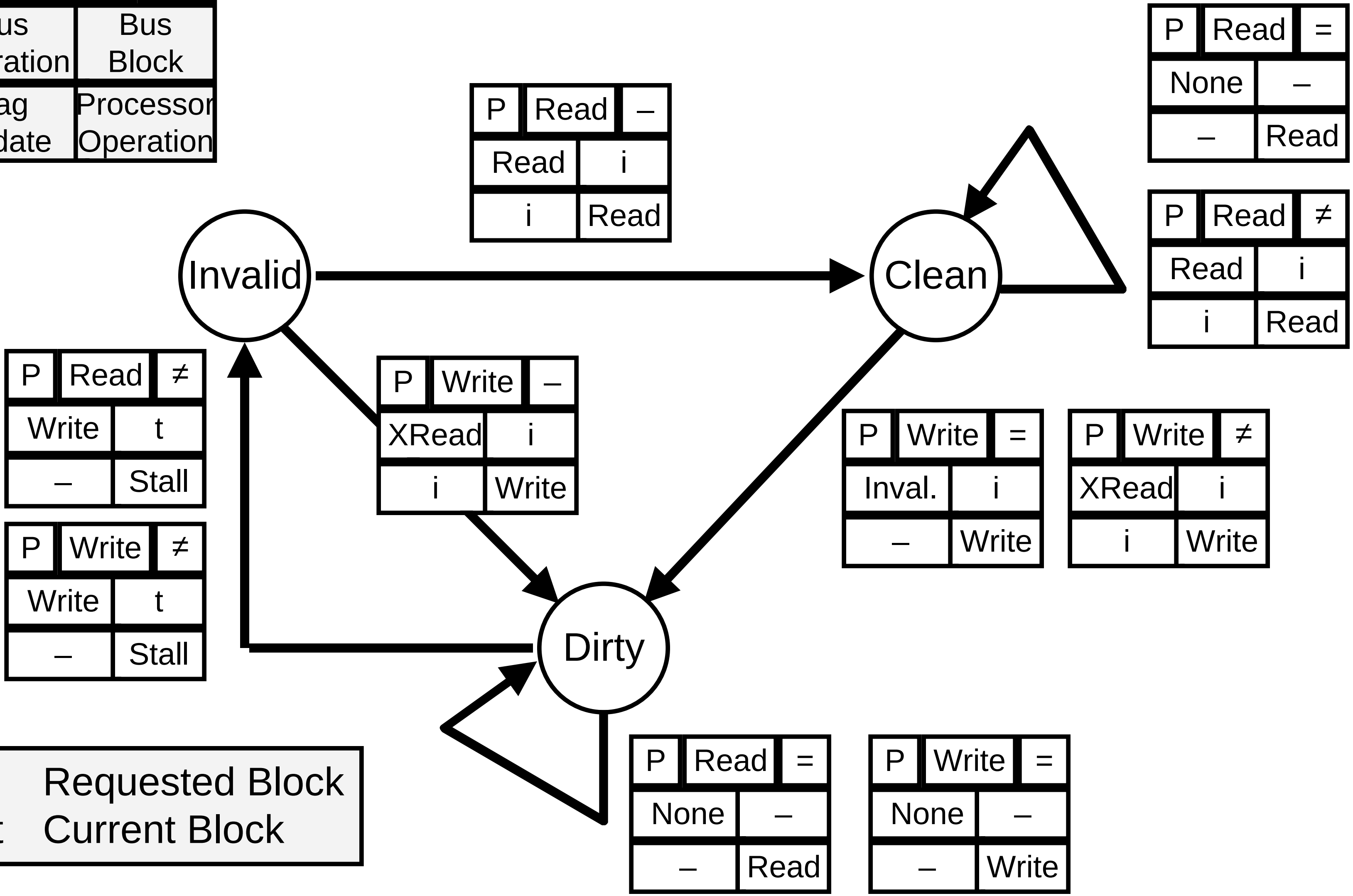
Action Key

P/B	Request Operation	i:t
Bus Operation	Bus Block	
Tag Update	Processor Operation	



Bus Master Actions

P/B	Request Operation	i:t
Bus Operation	Bus Block	
Tag Update	Processor Operation	



P	Read	–
Read	i	
i	Read	

P	Read	=
None	–	
–	Read	

P	Read	≠
Read	i	
i	Read	

P	Write	–
XRead	i	
i	Write	

P	Write	=
Inval.	i	
–	Write	

P	Write	≠
XRead	i	
i	Write	

P	Read	≠
Write	t	
–	Stall	

P	Write	≠
Write	t	
–	Stall	

P	Read	=
None	–	
–	Read	

P	Write	=
None	–	
–	Write	

i	Requested Block
t	Current Block

Bus Master State Update

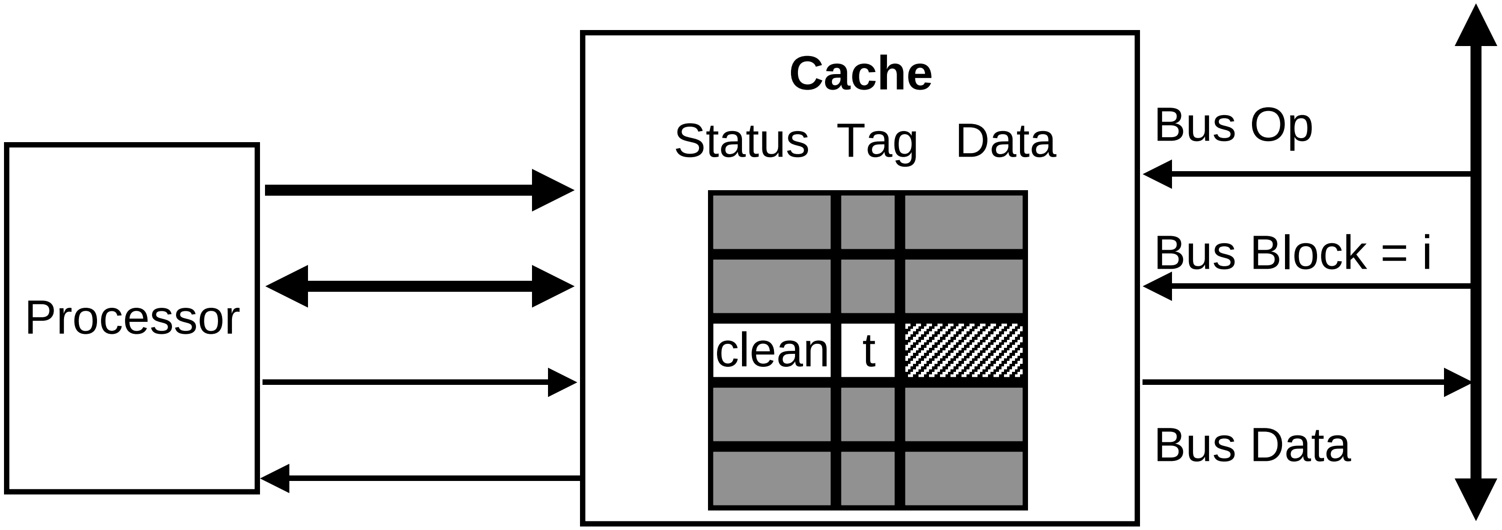
```
init(state) := invalid;
  next(state) :=
    case master :
      case
        state in { invalid, clean } :
          case
            proc-op = read : clean;
            proc-op = write : dirty;
            1 : state;
          esac;
        state = dirty :
          case
            proc-line = tag : dirty;
            proc-op in { read, write } : invalid;
            1 : state;
          esac;
        1 : state;
      esac;
```

Bus Monitoring

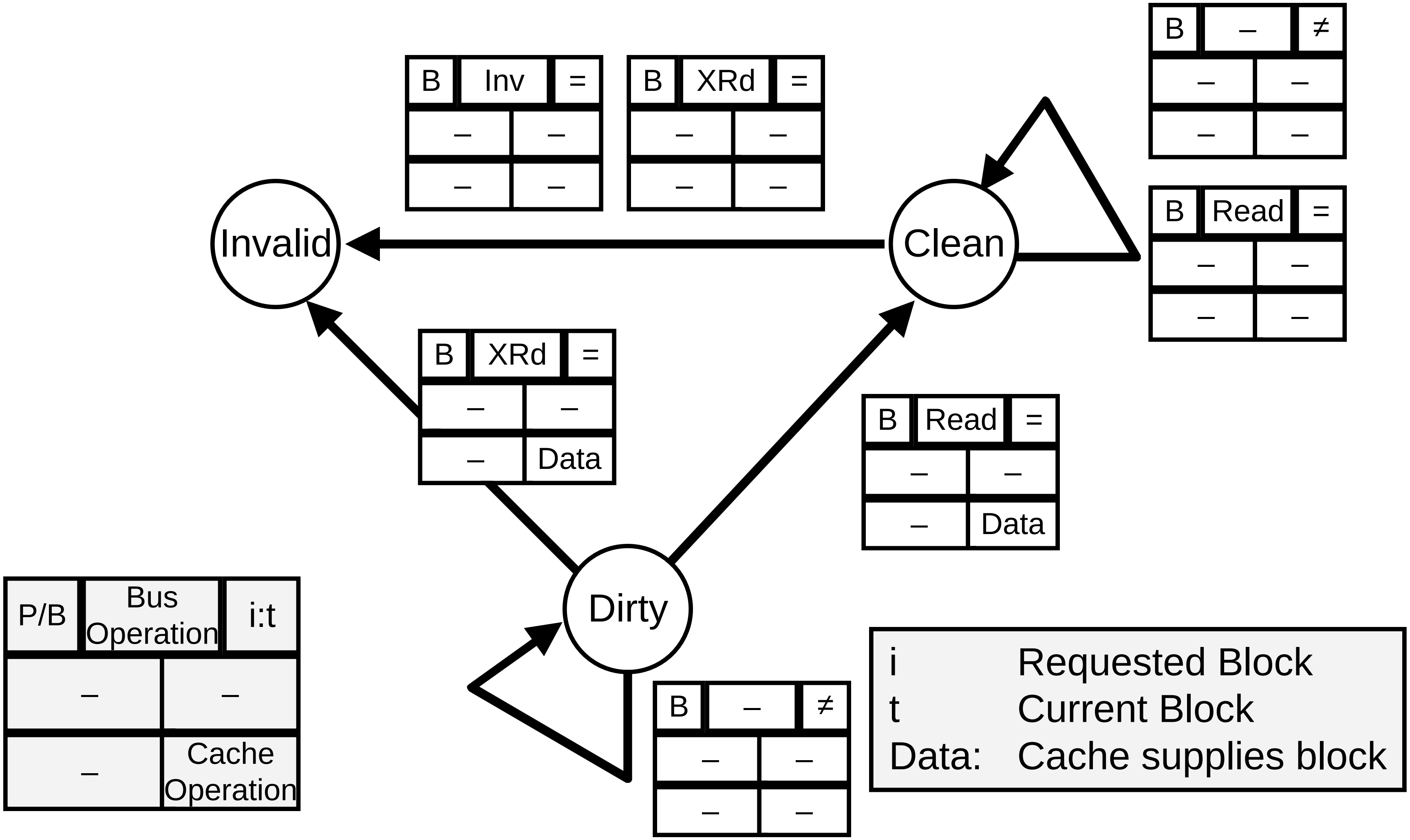
- **Cache monitors bus traffic when not master**
  - – Looks for operations on blocks matching cache entries
- **Possible actions**
  - – Invalidate entry
  - – Allow sharing of exclusively held block
    - » Supply data on bus

Action Key

P/B	Bus Operation	i:t
–		–
–		Cache Operation



Bus Snoop Actions



P/B	Bus Operation	i:t
–	–	
–	Cache Operation	

i	Requested Block
t	Current Block
Data:	Cache supplies block

Bus Snoop State Update

```
1 :   -- ! master
      case
        state = clean :
          case
            !(bus-line = tag) : clean;
            bus-op in { invalidate, xread } : invalid;
            bus-op = write : error;
            1 : state;
          esac;
        state = dirty :
          case
            !(bus-line = tag) : dirty;
            bus-op = read : clean;
            bus-op = xread : invalid;
            bus-op in { write, invalidate } : error;
            1 : state;
          esac;
        1 : state ;
      esac;
    esac;
```

Maintaining Tag

```
init(tag) := {lnA, lnB} ;
  next(tag) :=
    case
      !master : tag;
      -- When bus master, operate on behalf of processor
      state in { invalid, clean } &
        proc-op in {read, write} : proc-line;
      1 : tag;
    esac;
```

- **Only update when loading new block**
- **Due to processor read or write operation**

Defining Bus Request

```
bus-req :=
    case bus-op = arbitrate :
      case
        state = invalid :
          case
            proc-op = read : read;
            proc-op = write : xread;
            1 : no-op;
          esac;
        state = clean :
          case
            proc-op = read & ! proc-line = tag : read;
            proc-op = write & proc-line = tag : invalidate;
            proc-op = write : xread;
            1 : no-op;
          esac;
        state = dirty :
          case
            proc-op in {read, write} &
               ! proc-line = tag : write;

            1 : no-op;
          esac;
        1 : no-op;
      esac;
      1 : no-op;
    esac;
```

Other Defines

Processor Stall Signal

```
stall :=
            proc-op in {read, write} &
            ! proc-line = tag |
            proc-op = read & ! state in {clean, dirty} |
            proc-op = write & ! state = dirty;
```

Address for Bus Request

```
bus-line-req :=
    case
      state in {invalid, clean} : proc-line;
      state = dirty : tag;
      1 : {lnA, lnB} ;
    esac;
```

Ownership Conditions

```
ownA := state = dirty & tag = lnA;
ownB := state = dirty & tag = lnB;
```

Cache Specification

Safety

```
-- Block A has unique owner
      AG (c0.ownA + c1.ownA + c2.ownA + m.ownA = 1)
   -- Block B has unique owner
    & AG (c0.ownB + c1.ownB + c2.ownB + m.ownB = 1)
   -- No error states
    & AG (!(c0.state = error) & !(c1.state = error)
    &     !(c2.state = error))
```

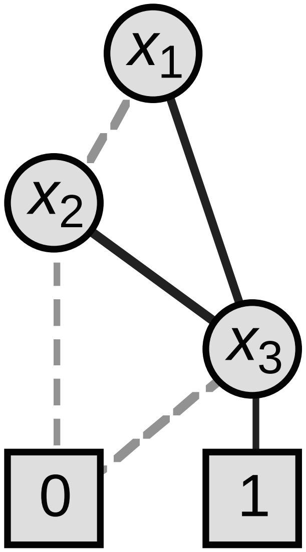
Liveness

```
& AG (AF !c0.stall & AF !c1.stall & AF !c2.stall)
```

Boolean Manipulation with OBDDs

- Ordered Binary Decision Diagrams
- Data structure for representing Boolean functions
- Widely used for other VLSI CAD tasks

Example:



$(x_1 + x_2) \cdot x_3$

- ◆ **Nodes represent variable tests**
- ◆ **Branches represent variable values**

  Dashed for value 0

  Solid for value 1

Representing Circuit Functions

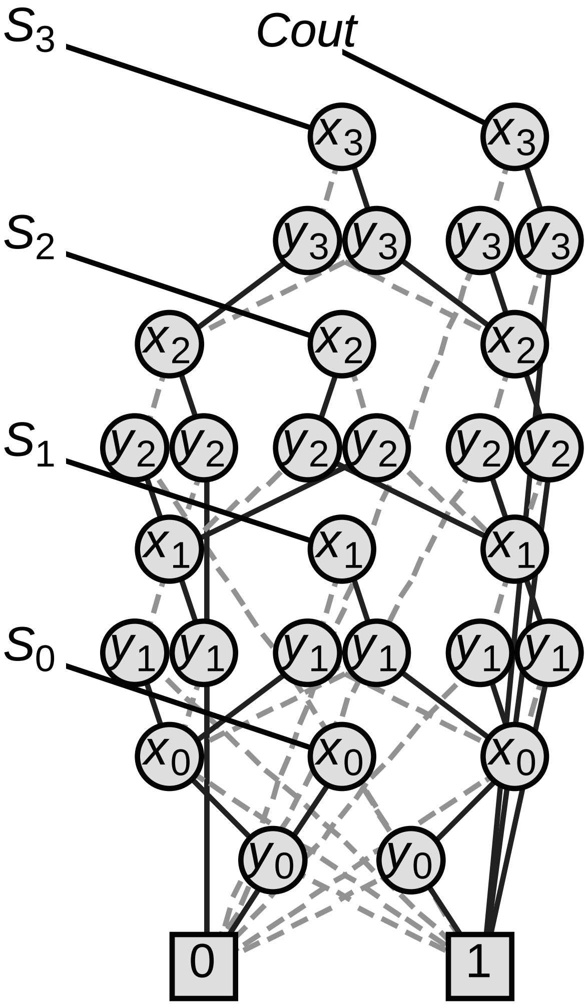
Functions

- All outputs of 4-bit adder
- as functions of data and carry inputs

Shared Representation

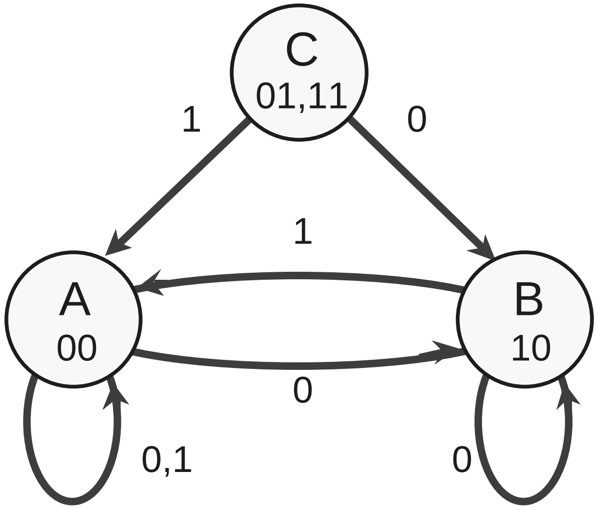
- Graph with multiple roots
- 31 nodes for 4-bit adder
- 571 nodes for 64-bit adder

* *Linear growth*

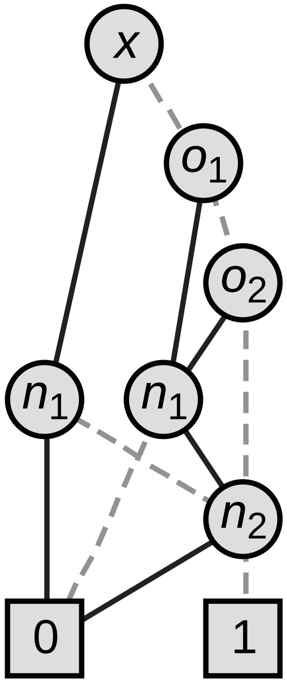


Symbolic FSM Representation

Nondeterministic FSM



Symbolic Representation



x input

o_1,o_2 encoded old state

n_1, n_2 encoded new state

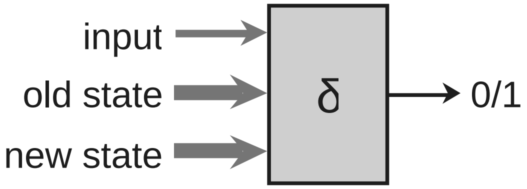
- **Represent set of transitions as function δ(*x*, *o*, *n*)**
  - Yields 1 if input *x* can cause transition from state *o* to state *n*.
- **Represent as Boolean function**
  - Over variables encoding states and inputs

Example: Reachability Analysis

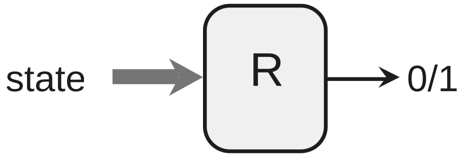
Task

- **Compute set of states reachable from initial state Q0**
- **Represent as Boolean function R(*s*).**
- **Never enumerate states explicitly**

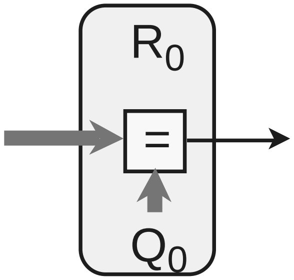
Given



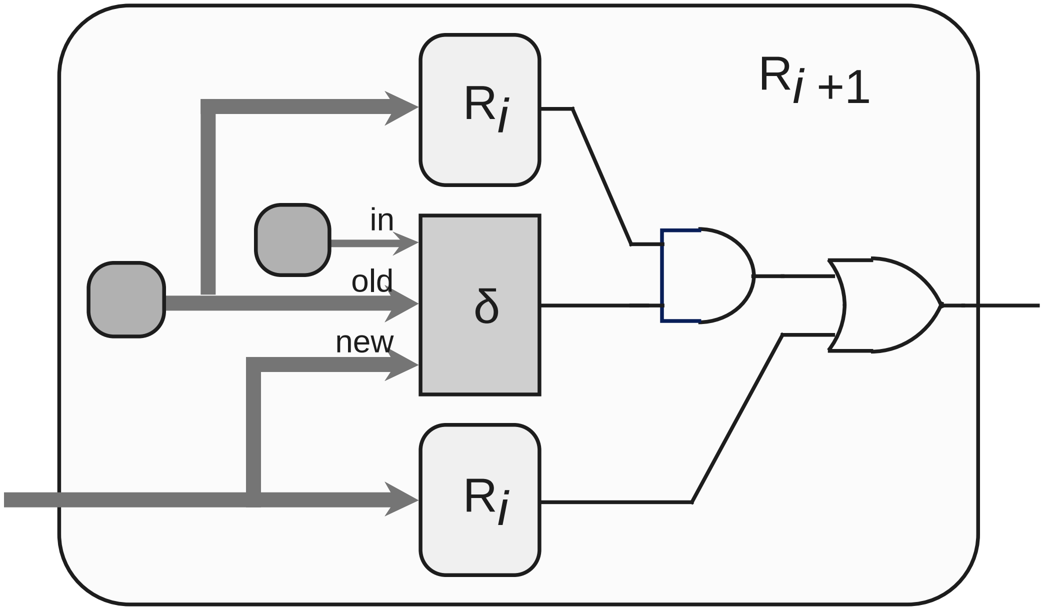
Compute



Initial



Iterative Computation



- ***R*$_{i+1}$ – set of states that can be reached *i* +1 transitions**
  - Either in R_i
  - or single transition away from some element of R_i
  - for some input
- **Continue iterating until $R_i = R_{i+1}$**

The Symbolic Advantage

Handle Large State Spaces

- **Single 32-bit register has over 4 billion states**
- **As combine modules, states increase multiplicatively**

Why BDDs?

- **Often remain compact, even though state spaces very large**
- **Algorithmic way to compose functions, project relations, test for convergence**
  - Never explicitly enumerate states