5. Finite Automata and Temporal Logic

Jacob Abraham

Department of Electrical and Computer Engineering
The University of Texas at Austin

Verification of Digital Systems
Spring 2020

February 6, 2020

Designs with Memory Elements

- At a particular point in time the states of memory elements form a *PRESENT STATE* of the sequential circuit
- Any *INPUT* stimulus can change the status of some of the memory elements. This new status is called the *NEXT STATE*
- Different inputs may cause the PRESENT STATE to respond differently, to become different NEXT STATES; and associated with these responses, different *OUTPUTS*
- The change of states and production of outputs corresponding to a set of inputs is *consistent* and is *predictable*
- The sequential circuit of this kind is referred to as a *Finite State Machine (FSM)*

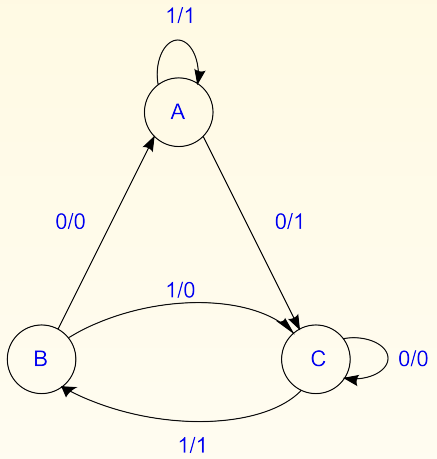
Fundamentals of FSMs will be illustrated using simple examples in the following slides

It is very important to note that only simple FSMs can be verified by enumerating all the states; techniques for analyzing large FSMs without explicit enumeration of the states will be discussed later

Illustration of State Table and State Diagram

Present State	Input 0	Input 1
A	C 1	A 1
B	A 0	C 0
C	C 0	B 1

Next State, Output



Some of the inputs, outputs or transitions may not be specified, resulting in an *Incompletely Specified* FSM

Example of a Finite-State Machine

One input, one output, four states

	0	1
A	A,0	B,0
B	C,0	D,1
C	B,0	A,0
D	D,1	B,1

One implementation (state assignment) using two D-flip-flops

	q_{1a}	q_{2a}
A	0	0
B	0	1
C	1	0
D	1	1

Output function for circuit (C_a) is
$z_a = Z_a(x, q_{1a}, q_{2a}) = xq_{2a} + q_{1a}q_{2a}$
The next state functions are
$Q_{1a} = N_1^a(x, q_{1a}, q_{2a}) = \bar{q}_{1a}q_{2a} + \bar{x}q_{2a}$
$Q_{2a} = N_2^a(x, q_{1a}, q_{2a}) = \bar{x}q_{1a} + x\bar{q}_{1a} + xq_{2a}$

A different state assignment (flip-flop encodings to symbolic states) would result in a different implementation of the same FSM

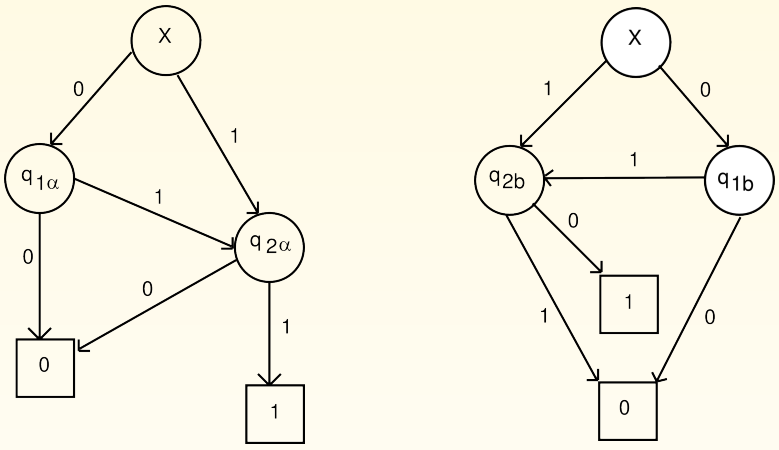
For example, another implementation could use four D-flops (with a *one-hot* assignment), resulting in different hardware

Another Implementation of the FSM

	q_{1b}	q_{2b}
A	1	1
B	0	0
C	0	1
D	1	0

The output function for C_b is
$z_b = Z_b(x, q_{1b}, q_{2b}) = q_{1b}\bar{q}_{2b} + x\bar{q}_{2b}$
and the next state functions for C_b are
$Q_{1b} = N_1^b(x, q_{1b}, q_{2b}) = \bar{x}q_{1b} + x\bar{q}_{1b}$
and
$Q_{2b} = N_2^b(x, q_{1b}, q_{2b}) =$
$\bar{x}\bar{q}_{1b}\bar{q}_{2b} + x\bar{q}_{1b}q_{2b} + \bar{x}q_{1b}q_{2b}$.

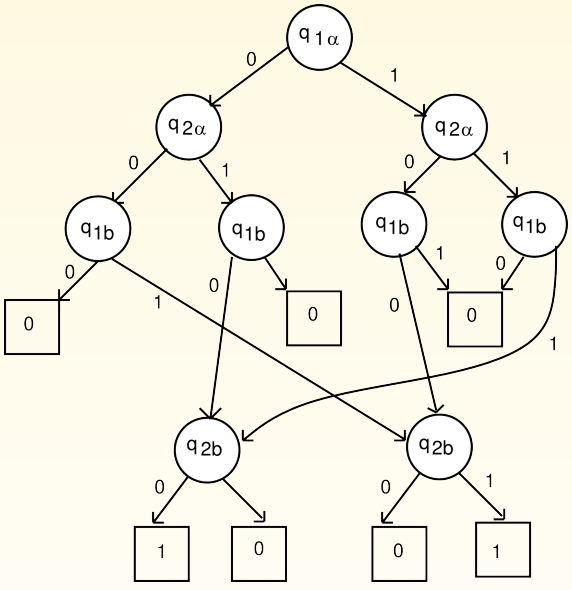
BDD for Output Functions of the Two FSMs



Equivalent States in Two FSMs

q_{1a}	q_{2a}	q_{1b}	q_{2b}	state of C_a	state of C_b
0	0	1	1	A	A
1	0	0	1	C	C
0	1	0	0	B	B
1	1	1	0	D	D

BDD Representing Sets of Equivalent States



Characteristic Functions

Given two sets $\mathcal{A}$ and $\mathcal{B}$, with $\mathcal{B} \subseteq \mathcal{A}$, a characteristic function with respect to $\mathcal{B}$ for an element x is defined to be True if x is in $\mathcal{B}$ and False otherwise

Characteristic functions can be represented by BDDs

For the AND gate of the previous example given by the relation, $y \leftrightarrow (x_1 \wedge x_2)$, the set $\mathcal{B}$ of gate consistent valuations is given by $\mathcal{B} = \{(0,0,0),(0,0,1),(0,1,0),(1,1,1)\}$, ordered by y, x_1, x_2

Formal Definition of FSM

A *Finite-State Machine (FSM)* is a 5-tuple $(S, I, O, \delta, \theta)$, where

S – finite set of *States*

I – finite set of *Inputs*

O – finite set of *Outputs*

δ – *Next-State Function*, $\delta : S \times I \rightarrow S$

θ – *Output Function*, $\theta : S \times I \rightarrow O$

In verification, it is common to specify a set of *Initial States* (in contrast with manufacturing test, where a fault should ideally be detected when the FSM starts from an arbitrary state) – Why?

Finite Automata

A finite deterministic automaton $\mathcal{M}$ is a 6-tuple
$\mathcal{M} := (Q, \Sigma, \Delta, \delta, \lambda, q^0)$, where
Q is the finite set of states
Σ is the input alphabet
Δ is the output alphabet
δ is the transition function, $\delta : Q \times \Sigma \rightarrow Q$
λ is the output function, $\lambda : Q \times \Sigma \rightarrow \Delta$
q^0 is the initial state

Two states q and q' are *equivalent*,
$q \sim q' : \Longleftrightarrow \ \forall a, a \in \Sigma . \lambda(q, a) = \lambda(q', a) . \delta(q, a) \sim \delta'(q', a)$

Two automata are equivalent if their initial states are equivalent

The product automaton of two automata are the state pairs of both, and a transition between two state pairs is only possible if the components make a corresponding transition to the respective states of the target pair with the same input

Finite State Acceptor

Acceptors determine if a given input sequence has a certain property

The set of all input sequences which satisfy a given property is called the accepted language

A deterministic finite acceptor (DFA) $\mathcal{M}^a$ is a 5-tuple
$\mathcal{M}^a := (Q, \Sigma, \delta, q^0, F)$, where
Q is the finite set of states
Σ is the input alphabet
δ is the state transition function, $\delta : Q \times \Sigma \rightarrow Q$, q^0 is the initial state
$F \subseteq Q$ is the set of final (accepting) states

A finite sequence $\vec{a}$ is accepted by $\mathcal{M}^a$ if $\delta^*(q^0, \vec{a}) \in F$

Automata for Infinite Sequences

Have to change the notion of final states in order to deal with infinite sequences

If a Finite Automaton accepts arbitrarily long strings (called an ω automaton), there must be a substring that can be repeated ("pumping lemma")

An automaton can accept an infinite input sequence exactly when there is a resulting path on which an accepting state occurs infinitely often

Example, a Büchi automaton accepts infinite sequences (ω regular languages)

Other automata:

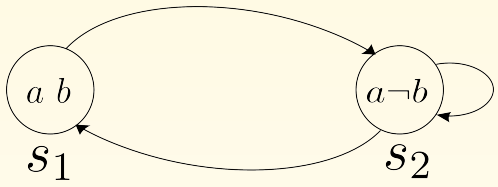
Müller auotmata
Streett automata
Rabin automata

Kripke Structures

- A nondeterministic FSM representing the behavior of a system
- A graph whose nodes represent the reachable states of the system and whose edges represent state transitions
- A labeling function maps each node to a set of properties that hold in the corresponding state
- Temporal logics are traditionally interpreted in terms of Kripke structures

Let $\mathcal{P}$ be a set of atomic propositions.
A Kripke (or temporal) structure $M := (S, I, R, L)$ consists of
1. A finite set of states, S
2. A set of initial states, $I \subseteq S$
3. A transition relation $R \subseteq S \times S$, with
$\forall s \in S \; \exists s' \in S \; . \; (s, s') \in R$ (i.e., R is total)
4. A labeling (or interpretation) function $L : S \rightarrow 2^{\mathcal{P}}$

Example of Kripke Structure



Transition s_1 to s_2 is represented as: $(a \wedge b \wedge a' \wedge \neg b')$

The Kripke structure is a disjunction of the three transitions
$(a \wedge b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge b')$

This can be encoded as a BDD (or other function representation)

Equivalent States

- States S_i and S_j are equivalent *iff* every possible input sequence causes the machine to produce the same output sequence when the initial state is S_i as it does when the initial state S_j
- A machine in which no two states are equivalent is a reduced machine
- Two states are k-distinguishable *iff* there exists an input sequence of length k that yields one output sequence when the machine is started in one state, and a different output sequence when it is started in the other state
- States that are not distinguishable by any experiment of length k or less are called k-equivalent

Determining k-Equivalence

Form partitions of the set of states, P_k of k-equivalent sets
P_1 iff states produce identical output symbols for each possible input symbol
P_2 on each of the partitions of P_1, etc.

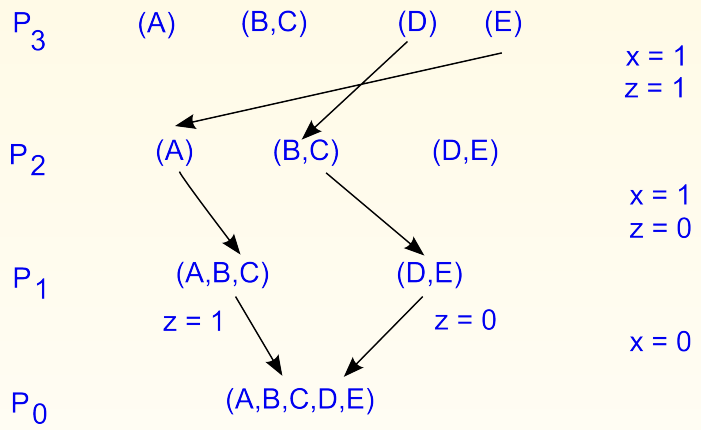
	0	1
A	B 1	C 0
B	C 1	D 0
C	B 1	D 0
D	E 0	C 1
E	D 0	A 1

P_1 = (A, B, C)(D,E)
P_2 = (A)(B, C)(D,E)
P_3 = (A)(B, C)(D)(E)
P_4 = (A)(B, C)(D)(E)

Two states are in the same block of P_k iff, for each input value σ, their σ-successors lie in a single block of P_{k-1}

In the example, B and C are equivalent states

Finding an Input Sequence to Distinguish States



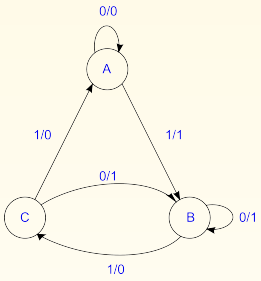
Acceptance and Recognition

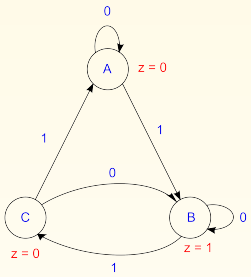
- FSMs can be viewed as devices for *classifying* or *transforming* input sequences
- If a machine is supplied with an arbitrarily long input sequence, the output sequence must ultimately become periodic
  - This property is useful only for small FSMs (such as small controllers)
- One means of classifying FSMs involves the last symbol in the output sequence that results from a given input sequence
  - If, in conjunction with the last input symbol, the machines produces a specific output symbol (generally take as a "1"), it is said to have accepted the given input sequence
- The set composed of all sequences that are accepted by a given machine is called the set that is recognized by the machine
- If a machine only has output symbols 0 and 1, specifying the recognized set is equivalent to specifying the input-output transformation that it performs

Mealy and Moore Machines

Machines in which the outputs depend on the transitions are called Mealy machines

If outputs can be associated only with states (i.e., all transitions entering a state are assigned the same output value), the machine is called a Moore machine



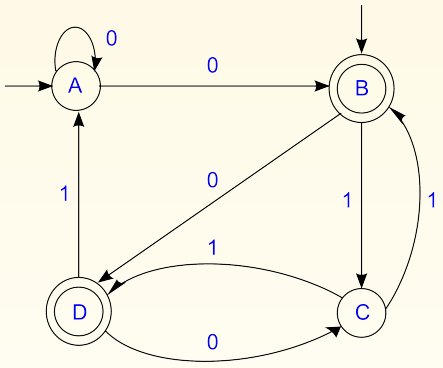


	0	1	z
A	A	B	0
B	B	C	0
C	B	A	0

Mealy machines can be transformed into Moore machines by splitting states that do not have the same output value assigned to all the incoming transitions

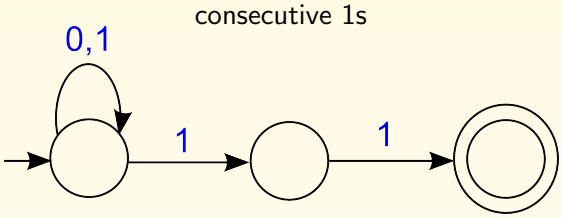
State Transition Graphs

Representation of FSM (when viewing it as describing sets of sequences)

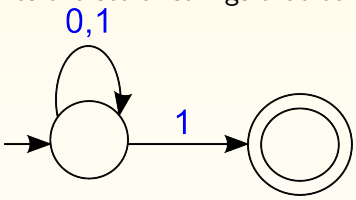


Examples of (Non-Deterministic) Transition Graphs

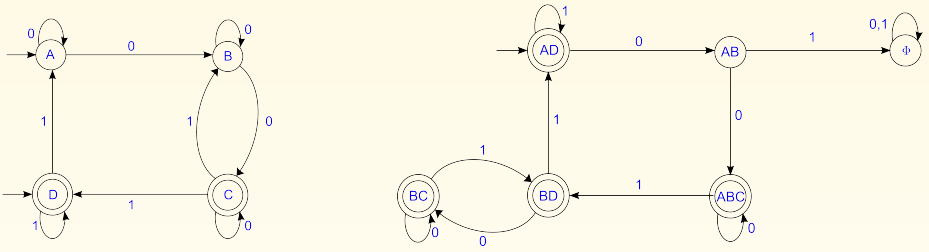
Graph that recognizes the set of strings that end with two consecutive 1s



Graph that recognizes the set of strings that contain at least one 1



Example: Converting Non-Deterministic Transition Graphs to Deterministic Form

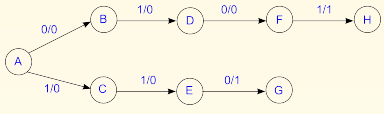


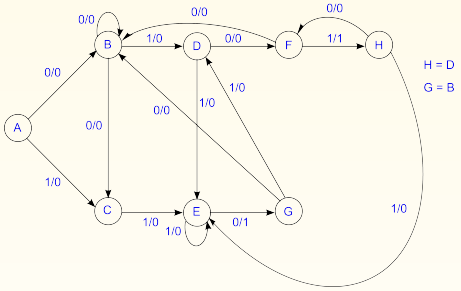
Any set of strings that can be recognized by a finite non-deterministic transition graph can also be recognized by a finite deterministic graph, and hence by a finite-state machine

Example: Design a Machine from Specifications

Machine which accepts an input sequence iff that sequence ends in either the subsequence 0101 or the subsequence 110

Derive design from specifications

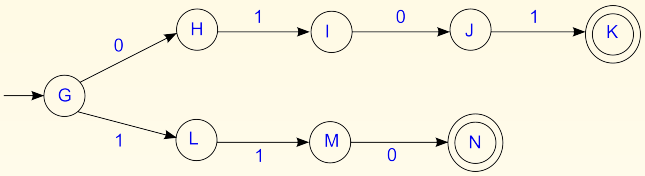




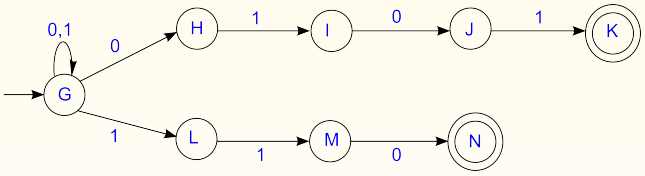
Example: Design a Machine from Specifications, Cont'd

Start with a non-deterministic graph, convert to deterministic

Non-deterministic graph which recognizes sequences



Allowing arbitrary number of symbols to precede desired strings



Sequences Relating to FSMs

- A Synchronizing Sequence is one which forces the FSM into a specific final state
- A sequence X is a Homing Sequence iff knowing the output sequence produced in response to X is always sufficient to uniquely determine the final state
- A Distinguishing Sequence is one which will produce a different output sequence for each initial state
- Given an upper bound on the number of states in a machine, an experiment which determines whether the machine corresponds to a given state description (specification) is called a Checking Experiment
  - Used also to test for faults in machines
  - First studied by Moore in 1956

Regular Expressions and Sets

Let $A = \{\sigma_1, \ldots, \sigma_k\}$ be a finite set of symbols. Then

(a) Each of the expressions $\sigma_1, \ldots, \sigma_k$ is a regular expression on A, as are λ (the null string) and ϕ (the empty set)

(b) If P and Q are regular expressions on A, then so is their *union* $P + Q$ and their *concatenation* $(P)(Q)$

(c) If P is a regular expression on A, then so is its *closure* (P)*

(d) Only those expressions that can be obtained by a finite number of applications of (a), (b) and (c) are regular expressions on A

The set A is usually called the *alphabet* of the expressions based on it

Regular expressions are used to represent sets of strings

Can you think of a set of strings that cannot be described by regular expressions?

Example Regular Expressions on the Alphabet {a, b, c}

Regular Expression R	Members of Set Represented by R	Examples of Strings in Set	Examples of Strings Not in Set
ab	the single string "ab"	ab	a, ba abb, cccb, λ
a+b	strings "a" and "b"	a, b	aa, ab, c, λ
a*	strings containing only a's	λ, a, aaaa	b, aaaca, baaa
ab*	strings beginning with one a, followed by b's	a, ab, abbbbb	abab, bbbb, aabbb, λ
(ab)*	strings consisting of repetitions of ab	λ, ab, abab abababababab	abba, aabb, ababc
(a+b)*	strings not containing c's	λ, abbbaba, bbbaaba, a	aaaaaac, abababc,
(a+b+c)*	all strings	λ, a, bcaac, cab	(none)
(a+b+c)*a	strings ending with a	a, cbbaca, aaaa	λ, abcabc, bbbb
(a+bc)*	every b followed by c, every c preceded by b	λ, abcbcaabc, bc, bcbcabca, aa	abcb, acbc, abcbabc
(a+b)*c(a+b)*	strings containing exactly one c	c, abac, cbbab, aabacbabbb	λ, aacbc, abbaaa

Examples of Regular Expressions

From the set {a, b, c}, what is the expression for strings containing an even number of c's?

What set does this regular expression describe?
1*+1*00(0+11*00)*(λ+11*)

What about
0*10*(10*10*)*
and (0*10*1)*0*10*

Some identities

- $\alpha(\beta\alpha)^* = (\alpha\beta)^*\alpha$
- $\alpha\beta + \alpha\gamma = \alpha(\beta + \gamma)$
- $\alpha^* = \lambda + \alpha\alpha^* = \lambda + \alpha^+$
- $(\alpha + \beta)^* = (\alpha^*\beta^*)^*$
- $(\alpha^*\beta)^* = \lambda + (\alpha + \beta)^*\beta$

Eample of Representing Circuits Using ROBDDs

- Consider a synchronous modulo–8 counter, and let $V = v_0, v_1, v_2$ and $V' = v'_0, v'_1, v'_2$
- The transitions of the modulo–8 counter are given by the following equations:

$$v'_0 = \neg v_0$$
$$v'_1 = v_0 \oplus v_1$$
$$v'_2 = (v_0 \wedge v_1) \oplus v_2$$

  - The above equations can be used to define the relations

$$N_0(V, V') = (v'_0 \Leftrightarrow \neg v_0)$$
$$N_1(V, V') = (v'_1 \Leftrightarrow v_0 \oplus v_1)$$
$$N_2(V, V') = (v'_2 \Leftrightarrow (v_0 \wedge v_1) \oplus v_2)$$

- Transition relation is given by

$$N(V, V') = N_0(V, V') \wedge N_1(V, V') \wedge N_2(V, V')$$

.

Reachability

- Reachable state computations are at the heart of any formal verification approach for sequential circuits
- States reachable from S_0
  $S_1 = S_0 \cup \{s' | \exists s[s \in S_0 \wedge (s, s') \in N]\}$
  - Using ROBBDs:
    $S_1(V') = S_0(V') \vee \exists_{v \in V \cup \Sigma}[S_0(V) \wedge N(V, V', \Sigma)]$
- States reachable in at most $k+1$ steps are represented by

$$S_{k+1}(V') = S_0(V') \vee \exists_{v \in V \cup \Sigma}[S_k(V) \wedge N(V, V', \Sigma)]$$

- Computations can be viewed as finding a **least fixed point**
- Predicate transformer F:
  $F(S) = S_0 \cup \{s' | \exists s[s \in S \wedge (s, s') \in N]\}$
  - Using ROBDDs we have:
    $F(S)(V') = S_0(V') \vee \exists_{v \in V \cup \Sigma}[S(V) \wedge N(V, V', \Sigma)]$
  - we have $F(S_i)(V') = S_{i+1}(V')$
  - the sequence of state sets $0, F(0), F^2(0)$, etc., converges to the least fixed point of F under the set containment ordering, and the fixed point is exactly the set of reachable states

Image Operators

- Given a set $Z \subseteq S$, define the following operators:

$$Image(N, Z) = \{v | \exists u[u \in Z \wedge N(u, v)]\}$$

$$PreImage(N, Z) = \{u | \exists v[v \in Z \wedge N(u, v)]\}$$

$$BackImage(N, Z) = \{u | \forall v[N(u, v) \Rightarrow v \in Z]\}$$

- Using ROBDDs
  - Image:

$$T(V') = \exists_{v \in V \cup \Sigma}[S(Z) \wedge N(V, V', \Sigma)]$$

$$I_{N,Z}(V) = \forall_{v' \in V'}[\exists_{v \in V}[(v \Leftrightarrow v') \wedge T(V')]]$$

  - PreImage :

$$T(V') = \forall_{v \in V}[\exists_{v' \in V'}[(v \Leftrightarrow v') \wedge S(Z)]]$$

$$T_1(V, \Sigma) = \forall_{v' \in V}[T(V') \otimes N_i(V, V', \Sigma)]$$

$$PI_{N,Z}(V) = \exists_{\sigma \in \Sigma}[T_1(V, \Sigma)]$$

Image Operators, Cont'd

- BackImage:

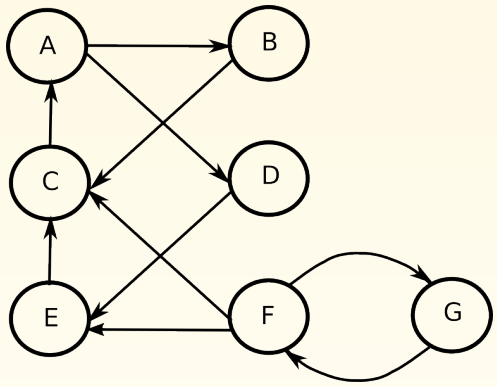
$$T(V') = \forall_{v \in V}[\exists_{v' \in V'}[(v \Leftrightarrow v') \wedge S(Z)]]$$
$$T_1(V, \Sigma) = \forall_{v' \in V}[T(V') \otimes N_i(V, V', \Sigma)]$$
$$PI_{N,Z}(V) = \forall_{\sigma \in \Sigma}[T_1(V, \Sigma)]$$

$BackImage(N, Z) = \neg PreImage(N, \neg Z)$
$BackImage(N, Y \wedge Z) =$
$BackImage(N, Y) \wedge BackImage(N, Z).$

Intuition

- Image gives the set of states that can be reached in one transition from a state in Z
- PreImage gives the set of states that in one transition can reach a state in Z
- BackImage gives the set of states that in one transition must end up in Z.

Example(s)



Verification Methodology

- Three requirements for Invariant Checking:
  - The initial state(s) is contained in G
  - All states reachable from G are contained in G
  - G and Z_0 are disjoint
- Forward traversal:
  - Initialize R_0 to the set of the initial states
  - Compute $R_{i+1} = R_0 \vee Image(N, R_i)$.
- Backward traversal:
  - Initialize $G_0 = G$
  - Compute $G_{i+1} = G_0 \wedge BackImage(N, G_i)$.
- A Third Method
  - Start from the set of states Z_0 rather than S_0
  - By reverse reachability analysis compute the set of states Z from which some state in Z_0 can be reached
  - Thus Z is the set of states that can reach a "bad" state

Design Verification

- If the specification is not formal, how do we verify the correctness of the design?
  - derive *properties* the design should have, based on the specification
  - check if the design satisfies the properties
- Temporal logic and variations have been used to specify properties for design verification
- Digital systems similar to reactive programs
- Digital systems receive inputs and produce outputs in a continuous interaction with their environment
- Behavior of digital systems is concurrent because each gate in the system simultaneously evaluating its output as a function of its inputs

Design Verification

Properties need to be specified formally, but general enough to be applicable to a wide variety of designs

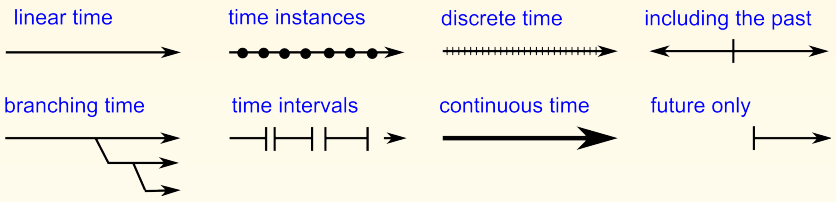
Check Properties of Design

- Since specification is usually not formal, check design for properties that would be consistent with the specification
- Safety "something bad will never happen"
- Liveness Property: "something good will eventually happen"
- Temporal Logic and variations commonly used to specify properties
- Example: Linear Temporal Logic (LTL) or Computation Tree Logic (CTL)

Introduction to Temporal Logic

- Operators of temporal propositional logic augmented by tense operators
- Tense operators used to form assertions about changes in time
- Temporal system provides a complete set of axioms and inference rules for providing all validities in the logic for a given model of time
- Models of time: partially ordered time, linearly ordered time, branching time, etc.
- Temporal logic can define semantics for programs for Floyd-Hoare style program proving
- Temporal logic can also prove properties like termination, possible termination and termination under fair scheduling of concurrent processes
  - For instance using this logic one can express the assertion that if proposition p holds in the present, then proposition q holds at some instant in the future
  - Temporal modalities can be combined to express complex statements about the past, present and the future

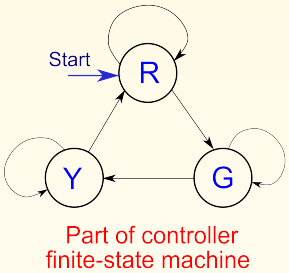
Modeling Time in Different Ways



Ref: Kropf, *Introduction to Formal Hardware Verification*, Springer, 1999

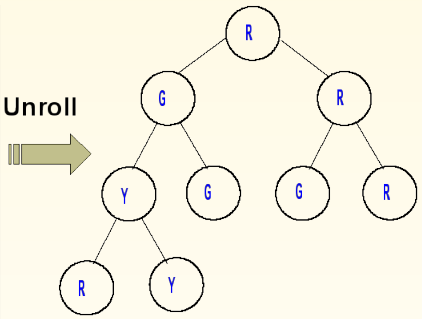
Example of Computation Tree

Traffic light controller



Part of controller
finite-state machine

Unroll



Computation Tree

Computation Tree Logic (CTL)

- CTL is a subset of modal branching time logic defined by Clarke and Emerson
- Temporal operators occur in pairs consisting of A or E followed by F, G, U or X
- Past time operators are not allowed
- Tense operators cannot be combined directly with the propositional connectives
- A and E are called *path quantifiers* and Fp, Gp, Xp and pUq are called path formulas from Propositional Linear Temporal Logic (PLTL)

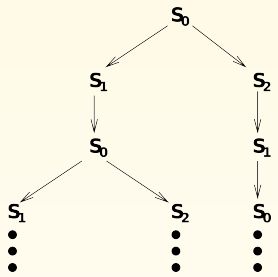
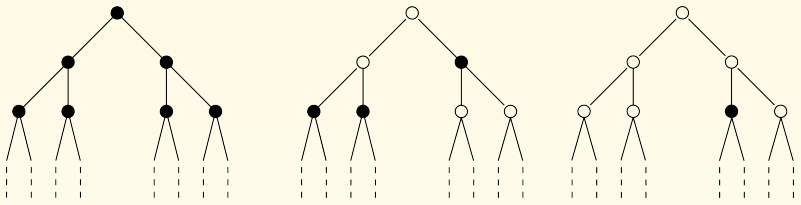


Illustration of Some CTL Operators

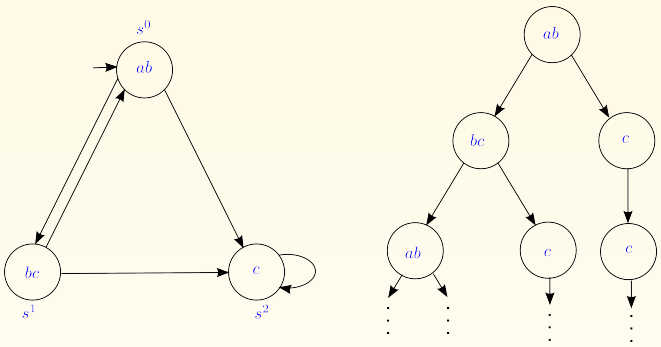


AGp **AFp** **EFp**

● **p is true**

○ **p is false**

Kripke Structure and Infinite Computation Tree



The labeling denotes which variables are true at each time instance

Formulas of CTL

Every temporal operator, F, G, X and U must be directly preceded by a path quantifier A or E, and every path quantifier must be followed by a temporal operator

Thus, only formulas like AXϕ or E(ϕ U ψ) are allowed, but NOT A(X $\phi \vee$ Fψ)

$AX\phi \leftrightarrow \neg EX(\neg\phi)$
$AG\phi \leftrightarrow \neg EF(\neg\phi)$
$AF\phi \leftrightarrow \neg EG(\neg\phi)$
$EF\phi \leftrightarrow E(\text{true } U\ \phi)$
$A(\phi\ U\ \psi) \leftrightarrow \neg E(\neg\psi\ U\ \neg\phi \wedge \neg\psi) \wedge \neg EG(\neg\psi)$

Examples of CTL Formulas for Verification

- It is possible to reach a state in which ϕ holds, but not ψ
  $EF(\phi \wedge \neg\psi)$
- When a request *req* occurs, then it will be eventually acknowledged by *ack*
  $AG(req \rightarrow AF\ ack)$
- The right for a computation *granted* holds infinitely often on every computation
  $AG(AF\ granted)$
- The *reset* state is reachable from every state
  $AG(EF\ reset)$

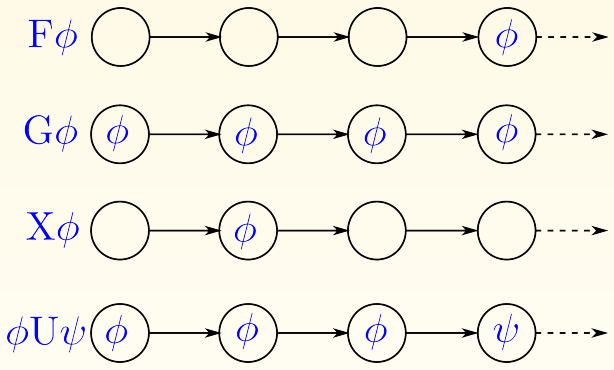
Fairness Constraints

- In many cases a CTL formula should be considered only with regard to a restricted set of paths – *fair paths*

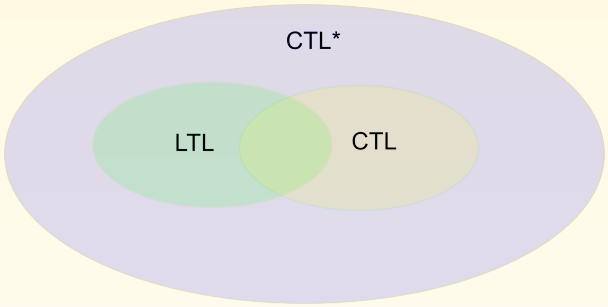
Linear Time Propositional Temporal Logic – LTL

Logic with regard to single, non-branching paths (proper subset of CTL*)

Semantics of the LTL Operators



Expressiveness of CTL*, CTL and LTL



CTL* formula $A(FG\phi) \vee AG(EF\phi)$ not contained in CTL or LTL

CTL formula $AG(p \rightarrow ((AXq) \vee) AX \neg q)$ cannot be expressed in LTL (used in database transactions)

LTL formula FGp cannot be precisely expressed in CTL
$AF\ AG\ p$ is stronger, and $AF\ EG\ p$ is weaker

Clarity and Complexity of CTL versus LTL

- CTL is more expressive than LTL – but this does not mean that it is more clear and intuitive
  - Most properties written are very simple – safety properties relating to bad states, for example
  - LTL more intuitive for most people
- FXp and XFp mean the same thing
- AFAXp and AXAFp do not
- CTL model checking algorithms run in $O(nm)$ time, where n is the size of the transition system and m is the size of the temporal formula
- LTL model checking algorithms run in $n.2^{O(m)}$ time $(m << n)$

Comparison between CTL and LTL, Continued

- Better error traces from LTL
  - CTL is branching time, and if some properties are disproved, there is no linear trace
  - All LTL property failures can produce a single linear trace
- More difficult to do Semiformal Verification (combining formal verification and simulation) with CTL
  - Practical verification is necessarily semiformal
- Compositional verification works more easily with LTL
- Abstractions can be mapped to language containment, which LTL can handle
  - To verify if a design P_1 is a refinement of P_2, just have to check $L(P_1) \subseteq L(P_2)$
- LTL is not able to express all assumptions about the environment in modular verification
- Commercial assertion languages and verification tools are primarily based on LTL

Power of Reachability Computations

Map CTL Operators to Reachability

- Safety Properties can be checked with Reachability analysis
- Industry verification tends to focus on reachability
- ATPG algorithms (test for a stuck-at fault) can be used to check reachability
- Can transform liveness checking problems to safety checking problems*

Industry Verification

- Temporal logic formulas are the basis for assertions used in industry
- Example, System Verilog Assertions

* Biere, Artho and Shuppan, "Liveness Checking as Safety Checking," *Electronic Notes in Theoretical Computer Science*, vol. 66, no. 2 (2002).

Temporal Logic Assertions in Practise: System Verilog

- A hardware description and verification language
- Superset of Verilog
- Extensive set of enhancements to IEEE 1364 Verilog-2001 standards
  - Developed originally by Accellera to improve productivity in the design of large gate-count, IP-based, bus-intensive chips
  - Features inherited from Verilog-HDL, C++, etc.
  - Targeted primarily at the chip implementation and verification flow, with links to system-level design flow
  - Adds several new keywords to Verilog $\Longrightarrow$ use *compatibility switches* to avoid errors with identifiers
- Web resources
  - `www.eda.org/sv/`
  - `www.asic-world.com/systemverilog/index.html`
- Focus on verification aspects in this course

Capabilities of System Verilog

- New data types (e.g., logic)
- Object-oriented programming support
- Constrained randomization
- Easy C model integration
- Assertions
- Coverage support
- Narrows the gap between design and verification engineer

Relaxed Data Type Rules in System Verilog

Verilog

- Strict about usage of wire, reg data type
- Variable types are 4 state 0,1,X,Z

System Verilog

- Logic data type can be used so no need to worry about reg, wire
- 2 state data type added – 0, 1 state
- 2 state variable can be used in test benches, where X,Z are not required
- 2 state variable in RTL model may enable simulators to be more efficient

Memory Management

Verilog

- Memories in verilog are static in nature
- Example:
  `reg[7:0] X[0:127];`
  128 bytes of memory

System Verilog

- Memories are dynamic in nature
- Allocated at runtime
- Better memory management (for queues, for example)
- Example:
  `Logic[3:0] length[$];`
  an empty queue with an unbounded size of logic data type

Verilog	System Verilog
Uses `always` to represent • Sequential logic • Combinational logic • Latched logic	Uses three new procedures • `always_ff` – sequential logic • `always_comb` – combinational logic • `always_latch` – latched logic
• Ports are connected using either named instance or positional instance	• Ports are connected using `Design DUT(.*);` which means: connect all port to variables or nets with the same name as the ports

Currently, limited synthesis support for System Verilog; it can be used for verification, but not design

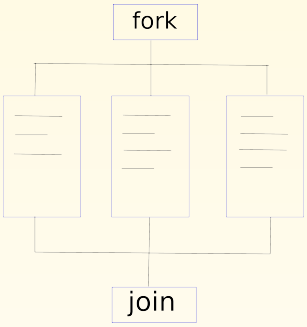
Data Types and Implication

```
reg r;          // 4-state Verilog-2001
logic w;        // 4-valued logic, see below
bit b;          // 2-state bit 0 or 1

integer i;      // 4-state, 32-bits, signed Verilog-2001
byte b8;        // 8 bit signed integer
int i;          // 2-state, 32-bit signed integer
shortint s;     // 2-state, 16-bit signed integer
longint l;      // 2-state, 64-bit signed integer
```

`logic` has a single driver (procedural assignments or a continuous assignment), can replace `reg` and single driver `wire`

```
$rose(start) |-> ##2 $rose(done);  // Overlapping
$rose(start) |=> ##1 $rose(done);  // Non-Overlapping
    // Two cycles after start, signal done must be high
```




```
Initial
Begin
  Clk = 0;
  #5
  Fork
    #5a = 0;
    #10b = 0;
  Join
  Clk = 1; //Clk is 1 at t=15
end
```

`fork/join_any`: Clk becomes 1 at t=10

`fork/join_none`: Clk becomes 1 at t=5

System Verilog Concepts

Tasks and Functions

- No begin end required
- Return can be used in task
- Function return values can have a "void return type"
- Functions can have any number of inputs, outputs and inouts including none

Direct Programming Interface (DPI)

- DPIs are used to call C, C++, System C functions
- System Verilog has a built in C interface
- Simple to use as compared to PLIs
- Values can be passed directly

Example: USB code (usbf_utmi_ls.v)

```
module usbf_utmi_ls( clk, rst,
resume_req, ... )
// UTMI Interface
...
// Main State Machine
//
always @(state or mode_hs or ...)
   begin
next_state = state;
...
```

```
case(state)
   RESUME: begin
  suspend_clr = 1'b1;
  if(ls_se0)
     ...
     end

   RESUME_REQUEST: begin
suspend_clr = 1'b1;
if(T2_wakeup)
      ...
     end
...
```


Sample Property for the Main State Machine

- Description: if the suspend bit is not cleared, then the machine is not resuming from the SUSPEND state in the main state machine
- Formula: $P4 : G(\neg(suspend\_clr) \implies X((state = \neg RESUME) \wedge (state = \neg RESUME\_REQUEST)))$

System Verilog Asseriton for P4

```
module assertions (
//signals for p4
input p4_suspend_clr,
input[14:0] p4_state,
input p4_clk,
input p4_rst
);
```

System Verilog Assertion for P4, Cont'd

```
parameter       [14:0]  // synopsys enum state1
        POR             = 15'b000_0000_0000_0001,
        NORMAL          = 15'b000_0000_0000_0010,
        RES_SUSP        = 15'b000_0000_0000_0100,
        SUSPEND         = 15'b000_0000_0000_1000,
        RESUME          = 15'b000_0000_0001_0000,
        RESUME_REQUEST  = 15'b000_0000_0010_0000,
        RESUME_WAIT     = 15'b000_0000_0100_0000,
        RESUME_SIG      = 15'b000_0000_1000_0000,
        ATTACH          = 15'b000_0001_0000_0000,
        RESET           = 15'b000_0010_0000_0000,
        SPEED_NEG       = 15'b000_0100_0000_0000,
        SPEED_NEG_K     = 15'b000_1000_0000_0000,
        SPEED_NEG_J     = 15'b001_0000_0000_0000,
        SPEED_NEG_HS    = 15'b010_0000_0000_0000,
        SPEED_NEG_FS    = 15'b100_0000_0000_0000;
```


System Verilog Assertion for P4, Cont'd

```
p4: assert property (
@(posedge p4_clk) disable iff (!p4_rst)
(!p4_suspend_clr) |=> ((p4_state != RESUME)
                        && (p4_state != RESUME_REQUEST)));

endmodule: assertions

module bindings;
//binding for p4
bind usbf_utmi_ls assertions req4 (
.p4_state(state),
.p4_suspend_clr(suspend_clr),
.p4_clk(clk),
.p4_rst(rst)
);
endmodule
```