

An Introduction to Formal Verification

Sandeep K. Shukla

CECS/ICS/UCI

Acknowledgement

- Maciej Ciesielski (Umass, Amherst)
- Kenneth Mcmillan (Cadence Berkeley Labs)

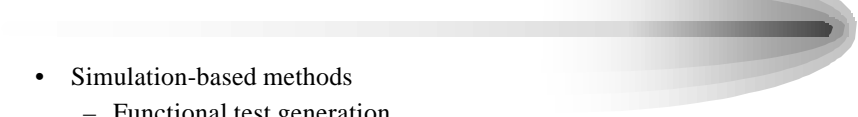
Overview

- Introduction
 - *What* is verification (validation)
 - *Why* do we need it
 - Formal vs. simulation-based methods
- Math background
 - BDD's
 - Symbolic FSM traversal

Overview – Formal Methods

- Equivalence checking
 - Combinational circuits
 - Sequential circuits
- Model checking
 - Problem statement
 - Explicit algorithms (on graphs)
 - Symbolic algorithms (using BDDs)
- Theorem proving
 - Deductive reasoning

Overview – Functional Testing



- Simulation-based methods
 - Functional test generation
 - SAT-based methods, Boolean verification
 - Boolean satisfiability
 - RTL verification
 - Arithmetic/Boolean satisfiability
 - ATPG-based methods
- Emulation-based methods
 - Hardware-assisted simulation
 - System prototyping

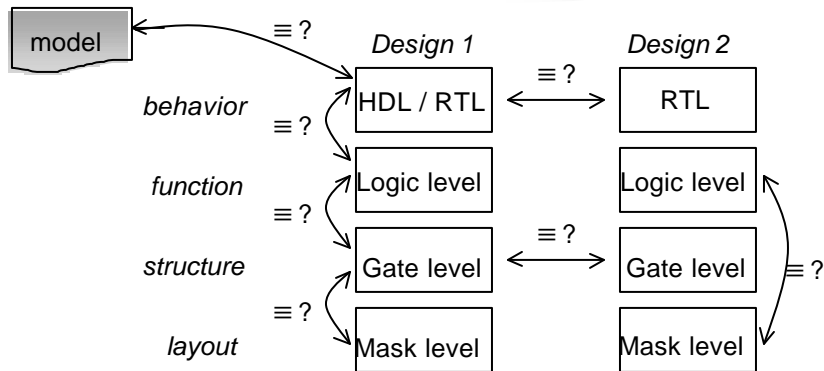
Part I



INTRODUCTION

Verification

- Design verification = ensuring correctness of the design
 - against its implementation (at different levels)
 - against alternative design (at the same level)



Why Verification

- Verification crisis
 - system complexity, difficult to manage
 - more time, effort devoted to verification than to design
 - need automated verification methods, integration
- Examples of undetected errors
 - Ariane 5 rocket explosion, 1996 (exception occurred when converting 64-bit floating number to a 16-bit integer)
 - Pentium bug (multiplier table not fully verified)
 - many more

Verification Methods

- Simulation - performed on the *model*
 - Deductive verification
 - Model checking
 - Equivalence checking
- } *Formal Verification*
- Testing - performed on the actual *product*
(manufacturing test)
 - Emulation, prototyping

Formal Verification

- Deductive reasoning (*theorem proving*)
 - uses axioms, rules to prove system correctness
 - no guarantee that it will terminate
 - difficult, time consuming: for critical applications only
- Model checking
 - automatic technique to prove correctness of concurrent systems: *digital circuits, communication protocols, etc.*
- Equivalence checking
 - check if two circuits are equivalent

Why Formal Verification

- Need for reliable hardware validation
- Simulation, test cannot handle all possible cases
- Formal verification conducts exhaustive exploration of *all* possible behaviors
 - compare to simulation, which explores *some* of possible behaviors
 - if correct, *all* behaviors are verified
 - if incorrect, a *counter-example* (proof) is presented
- Examples of successful use of formal verification
 - SMV system [McMillan 1993]

Part II

BACKGROUND

BDDs, FSM traversal

Binary Decision Diagrams

- Binary Decision Diagram (BDD)
 - compact data structure for Boolean logic
 - can represent sets of objects (states) encoded as Boolean functions
 - reduced ordered BDDs (ROBDD) are canonical
 - canonicity - essential for verification
- Construction of ROBDD
 - remove duplicate terminals
 - remove duplicate nodes (isomorphic subgraphs)
 - remove internal nodes with identical children

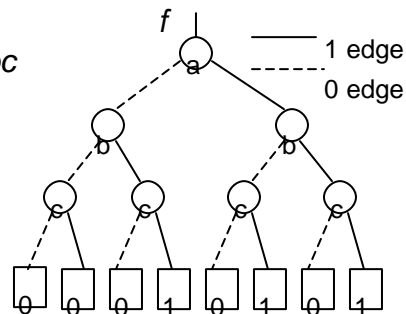
BDD - Construction

- Construction of a Reduced Ordered BDD

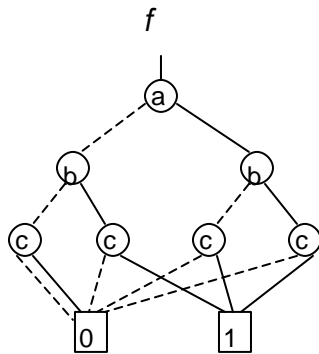
a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Truth table

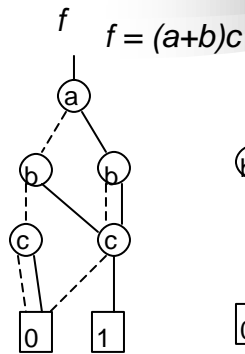
$$f = ac + bc$$



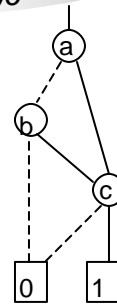
BDD Construction – cont'd



1. Remove duplicate terminals



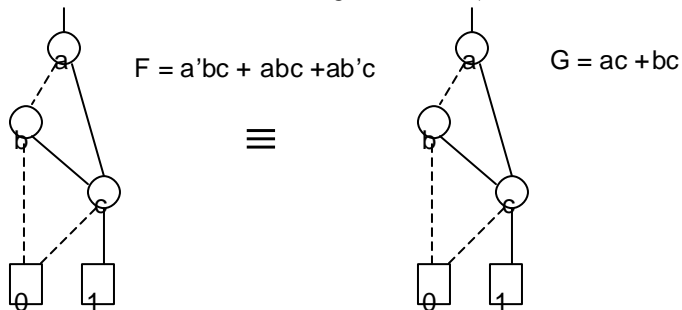
2. Remove duplicate nodes



3. Remove redundant nodes

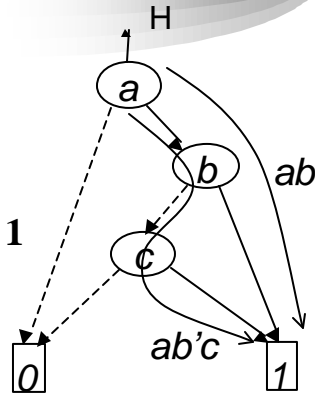
Application to Verification

- Equivalence of *combinational* circuits
- *Canonicity* property of BDDs:
 - if F and G are equivalent, their BDDs are identical (for the same ordering of variables)



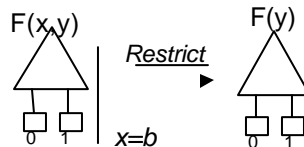
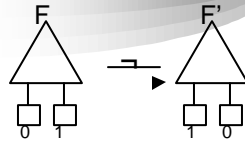
Application to Verification, cont'd

- Functional test generation
 - SAT, Boolean *satisfiability* analysis
 - to test for $H = 1$ (0), find a path in the BDD to terminal **1** (0)
 - the path, expressed in function variables, gives a satisfying solution (test vector)



Logic Manipulation using BDDs

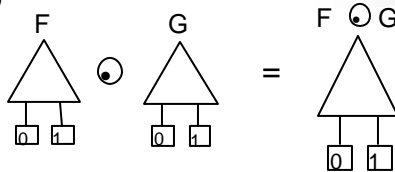
- Useful operators
 - *Complement* $\neg F = F'$
(switch the terminal nodes)
 - *Restrict*: $F|_{x=b} = F(x=b)$
where $b = \text{const}$



Useful BDD Operators - cont'd

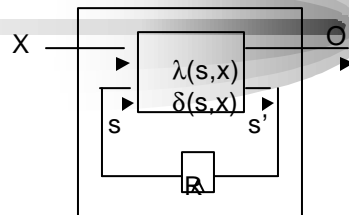
- Apply: $F \odot G$

where \odot stands for any Boolean operator (AND, OR, XOR, \rightarrow)



- Any logic operation can be expressed using only *Restrict* and *Apply*
- Efficient algorithms, work directly on BDDs

Finite State Machines (FSM)

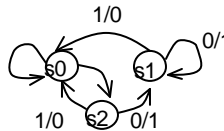


- FSM $M(X, S, \delta, \lambda, O)$

- Inputs: X
- Outputs: O
- States: S
- Next state function, $\delta(s, x) : S \times X \rightarrow S$
- Output function, $\lambda(s, x) : S \times X \rightarrow O$

FSM Traversal

- State Transition Graphs
 - directed graphs with labeled nodes and arcs (transitions)
 - symbolic state traversal methods
 - important for symbolic verification, state reachability analysis, FSM traversal, etc.



Existential Quantification

- Existential quantification (abstraction)

$$\mathcal{S}_x f = f|_{x=0} + f|_{x=1}$$

- Example:

$$\mathcal{S}_x (x y + z) = y + z$$

- Note: $\mathcal{S}_x f$ does not depend on x (smoothing)
- Useful in symbolic image computation (*sets of states*)

Existential Quantification - cont'd

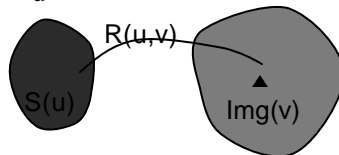
- Function can be existentially quantified w.r.to a vector: $X = x_1x_2\dots$

$$\exists_X f = \exists_{x_1x_2\dots} f = \exists_{x_1} \exists_{x_2} \exists_{\dots} f$$

- Can be done efficiently directly on a BDD
- Very useful in computing *sets* of states
 - Image computation: *next* states
 - Pre-Image computation: *previous* states from a given *set* of initial states

Image Computation

- Computing *set* of next states from a given initial state (or set of states)
- $\text{Img}(S, R) = \exists_u S(u) \bullet R(u, v)$



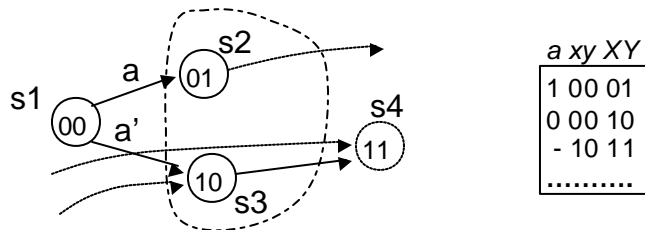
- FSM: when transitions are labeled with input predicates x , quantify w.r.to all inputs (primary inputs and state var)

$$\text{Img}(S, R) = \exists_x \exists_u S(u) \bullet R(x, u, v)$$

Image Computation - example

Compute a set of next states from state s1

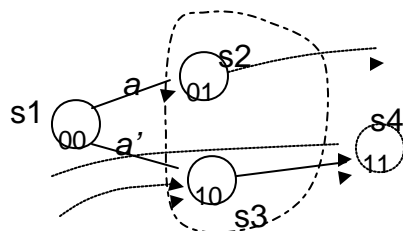
- Encode the states: s1=00, s2=01, s3=10, s4=11
- Write transition relations for the *encoded* states:
 $R = (ax'y'X'Y + a'x'y'XY' + xy'XY + \dots)$



Example - cont'd

- Compute Image from s1 under R

$$\begin{aligned}
 \text{Img}(s1, R) &= \sum_a \sum_{xy} s1(x,y) \cdot R(a,x,y,X,Y) \\
 &= \sum_a \sum_{xy} (x'y') \cdot (ax'y'X'Y + a'x'y'XY' + xy'XY + \dots) \\
 &= \sum_{axy} (ax'y'X'Y + a'x'y'XY') = (X'Y + XY') \\
 &= \{01, 10\} = \{s2, s3\}
 \end{aligned}$$

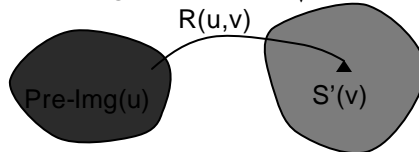


Result: a set of next states for *all* inputs
 $s1 \rightarrow \{s2, s3\}$

Pre-Image Computation

- Computing a *set* of present states from a given next state (or set of states)

$$\text{Pre-Img}(S', R) = \exists_v R(u, v) \bullet S'(v)$$



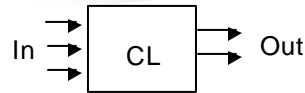
- Similar to Image computation, except that quantification is done w.r. to *next state* variables
- The result: a set of states *backward* reachable from state set S' , expressed in present state variables u
- Useful in computing CTL formulas: AF, EF

Part III

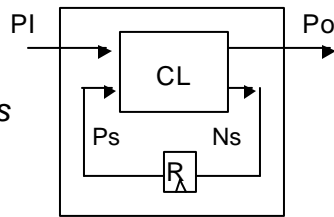
EQUIVALENCE CHECKING

Equivalence Checking

- Two circuits are *functionally equivalent* if they exhibit the same behavior



- Combinational circuits
 - for all possible input *values*
- Sequential circuits
 - for all possible input *sequences*



Combinational Equivalence Checking

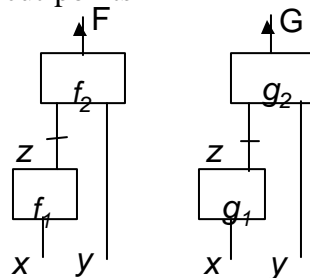
- Functional Approach
 - transform output functions of combinational circuits into a unique (*canonical*) representation
 - two circuits are equivalent if their representations are identical
 - efficient canonical representation: BDD
- Structural
 - identify structurally *similar* internal points
 - prove internal points (cut-points) *equivalent*
 - find implications

Functional Equivalence

- If BDD can be constructed for each circuit
 - represent each circuit as *shared* (multi-output) BDD
 - use the same variable ordering !
 - BDDs of both circuits must be *identical*
- If BDDs are too large
 - cannot construct BDD, memory problem
 - use partitioned BDD method
 - decompose circuit into smaller pieces, each as BDD
 - check equivalence of internal points

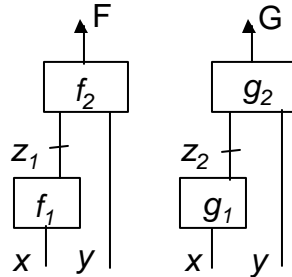
Functional Decomposition

- Decompose each function into *functional blocks*
 - represent each block as a BDD (*partitioned BDD* method)
 - define *cut-points* (z)
 - verify equivalence of blocks at cut-points starting at primary inputs



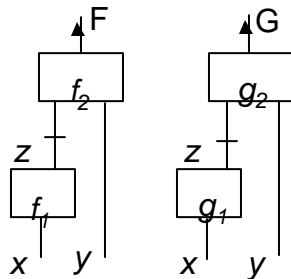
Cut-Points Resolution Problem

- If *all pairs* of cut-points (z_1, z_2) are equivalent
 - so are the two functions, F,G
- If *intermediate* functions (f_2, g_2) are not equivalent
 - the functions (F,G) may still be equivalent
 - this is called *false negative*
- Why do we have false negative ?
 - functions are represented in terms of *intermediate* variables
 - to prove/disprove equivalence must represent the functions in terms of *primary inputs* (BDD composition)



Cut-Point Resolution – Theory

- Let $f_1(x) = g_1(x) \quad \forall x$
 - if $f_2(z,y) \equiv g_2(z,y), \quad \forall z,y$ then $f_2(f_1(x),y) \equiv g_2(g_1(x),y) \Rightarrow F \equiv G$
 - if $f_2(z,y) \neq g_2(z,y), \quad \forall z,y \not\Rightarrow f_2(f_1(x),y) \neq g_2(g_1(x),y) \Rightarrow F \neq G$



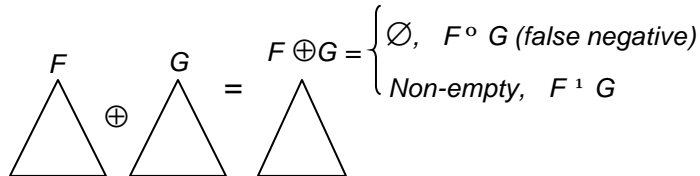
We cannot say if $F \equiv G$ or not

- *False negative*
 - two functions are equivalent, but the verification algorithm declares them as different.

Cut-Point Resolution – cont'd

- **Procedure 2:** create a BDD for $F \oplus G$

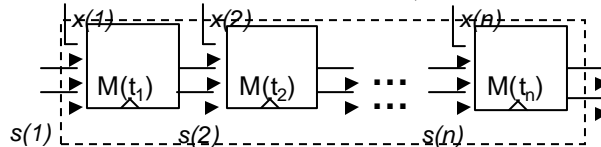
- perform satisfiability analysis (SAT) of the BDD
 - if BDD for $F \oplus G = \emptyset$, problem is *not* satisfiable, *false* negative
 - BDD for $F \oplus G \neq \emptyset$, problem is satisfiable, *true* negative



- the SAT solution, if exists, provides a *test vector* (proof of non-equivalence) – as in ATPG

Sequential Equivalence Checking

- Represent each sequential circuit as an FSM
 - verify if two FSMs are equivalent
- **Approach 1:** reduction to *combinational* circuit
 - unroll FSM over n time frames (flatten the design)

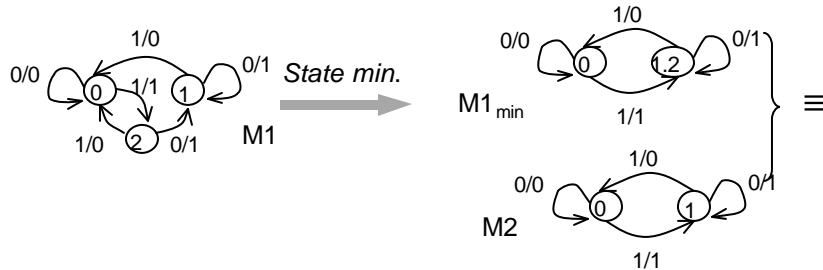


Combinational logic: $F(x(1,2, \dots, n), s(1,2, \dots, n))$

- check equivalence of the resulting combinational circuits
- problem: the resulting circuit can be too large to handle

Sequential Verification

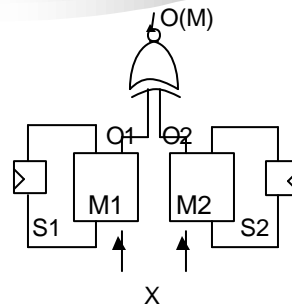
- Approach 2: based on isomorphism of state transition graphs
 - two machines M_1, M_2 are *equivalent* if their state transition graphs (STGs) are *isomorphic*
 - perform state minimization of each machine
 - check if $STG(M_1)$ and $STG(M_2)$ are isomorphic



Sequential Verification

- Approach 3: symbolic FSM traversal of the product machine

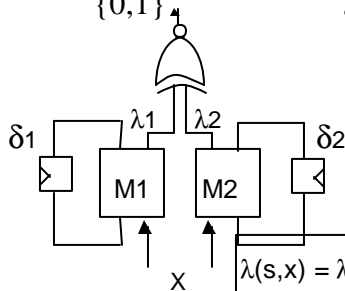
- Given two FSMs: $M_1(X, S_1, \delta_1, \lambda_1, O_1)$, $M_2(X, S_2, \delta_2, \lambda_2, O_2)$
- Create a product FSM: $M = M_1 \times M_2$
 - traverse the states of M and check its output for each transition
 - the output $O(M) = 1$, if outputs $O_1 = O_2$
 - if all outputs of M are 1, M_1 and M_2 are *equivalent*
 - otherwise, an *error state* is reached
 - *error trace* is produced to show: $M_1 \neq M_2$



Product Machine - Construction

- Define the product machine $M(X, S, \delta, \lambda, O)$

- states, $S = S_1 \times S_2$
- next state function, $\delta(s, x) : (S_1 \times S_2) \times X \rightarrow (S_1 \times S_2)$
- output function, $\lambda(s, x) : (S_1 \times S_2) \times X \rightarrow \{0, 1\}$



- Error trace (*distinguishing sequence*) that leads to an error state
 - sequence of inputs which produces 1 at the output of M
 - produces a state in M for which M1 and M2 give different outputs

$$\lambda(s, x) = \lambda_1(s_1, x) \oplus \lambda_2(s_2, x)$$

$$O = \begin{cases} 1 & \text{if } O_1 = O_2 \\ 0 & \text{otherwise} \end{cases}$$

FSM Traversal - Algorithm

- Traverse the product machine $M(X, S, \delta, \lambda, O)$
 - start at an initial state S_0
 - iteratively compute symbolic image $Img(S_0, R)$ (set of next states):

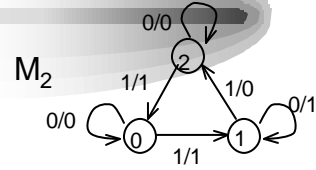
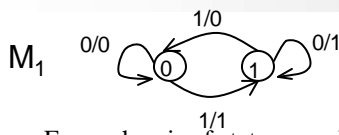
$$Img(S_0, R) = \bigcup_x S_s S_0(s) \cdot R(x, s, t)$$

$$R = \prod_i R_i = \prod_i (t_i \circ \delta_i(s, x))$$

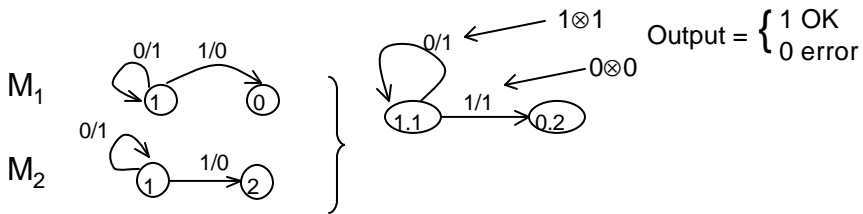
until an *error state* is reached

- transition relation R_i for each next state variable t_i can be computed as $t_i = (t \otimes \delta_i(s, x))$
(this is an alternative way to compute transition relation, when design is specified at gate level)

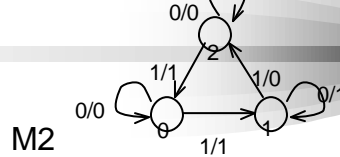
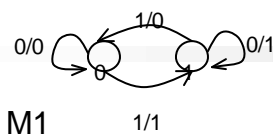
Construction of the Product FSM



- For each pair of states, $s_1 \in M_1, s_2 \in M_2$
 - create a combined state $s = (s_1, s_2)$ of M
 - create transitions out of this state to other states of M
 - label the transitions (*input/output*) accordingly



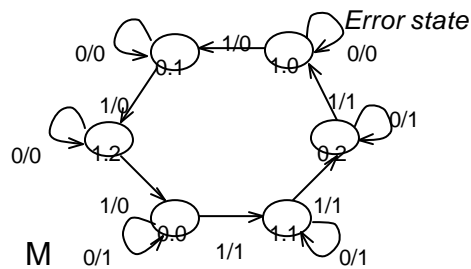
FSM Traversal in Action



Initial states: $s_1=0, s_2=0, s=(0.0)$

State reached	$x=0$	$x=1$	Out(M)
$New^0 = (0.0)$	1	1	
$New^1 = (1.1)$	1	1	
$New^2 = (0.2)$	1	1	
$New^3 = (1.0)$	0	0	

- $New^0 = (0.0)$ 1 1
- $New^1 = (1.1)$ 1 1
- $New^2 = (0.2)$ 1 1
- $New^3 = (1.0)$ 0 0



- STOP - backtrack to initial state to get error trace:
 $x=\{1,1,1,0\}$

Part IV

MODEL CHECKING

Model Checking

- Algorithmic method of verifying correctness of (finite state) concurrent systems against temporal logic specifications
 - A practical approach to *formal verification*
- Basic idea
 - System is described in a *formal model*
 - derived from high level design (HDL, C), circuit structure, etc.
 - The desired *behavior* is expressed as a set of *properties*
 - expressed as *temporal logic* specification
 - The specification is checked against the model

Model Checking

- How does it work
 - System is modeled as a *state transition structure* (Kripke structure)
 - Specification is expressed in *propositional temporal logic* (CTL formula)
 - asserts how system behavior evolves over time
 - Efficient search procedure checks the transition system to see if it satisfies the specification

Model Checking

- Characteristics
 - searches the entire solution space
 - always terminates with YES or NO
 - relatively easy, can be done by experienced designers
 - widely used in industry
 - can be automated
- Challenges
 - state space explosion – use symbolic methods, BDDs
- History
 - Clark, Emerson [1981] USA
 - Quielle, Sifakis [1980's] France

Model Checking - Tasks

- Modeling
 - converts a design into a formalism: *state transition system*
- Specification
 - state the properties that the design must satisfy
 - use logical formalism: *temporal logic*
 - asserts how system behavior evolves over time
- Verification
 - automated procedure (algorithm)

Model Checking - Issues

- Completeness
 - model checking is effective for a *given* property
 - impossible to guarantee that the specification covers *all* properties the system should satisfy
 - writing the specification - responsibility of the user
- Negative results
 - incorrect model
 - incorrect specification (*false negative*)
 - failure to complete the check (too large)

Model Checking - Basics

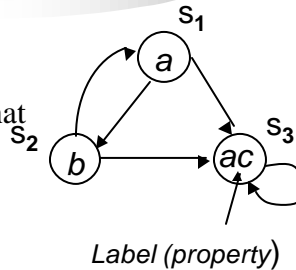
- State transition structure
M(S,R,L) (Kripke structure)

S = finite set of states $\{s_1, s_2, \dots, s_n\}$

R = transition relation

L = set of labels assigned to states, so that

$L(s) = f$ if state s has property f

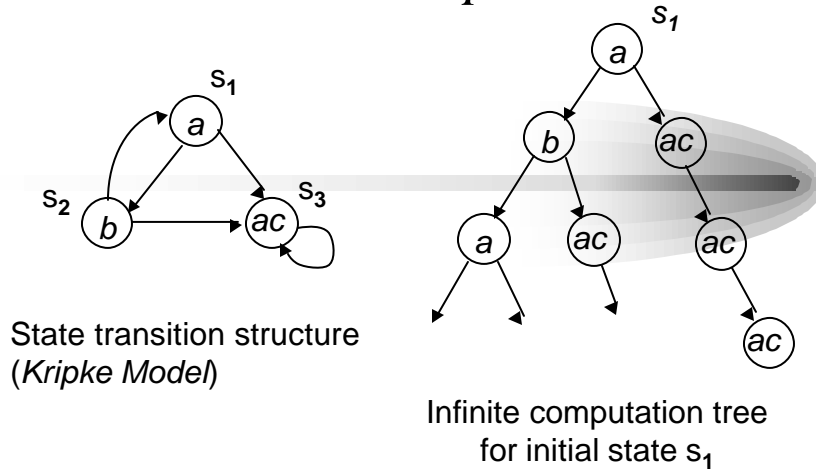


- All properties are composed of *atomic propositions* (basic properties), e.g. the light is green, the door is open, etc.
 - $L(s)$ is a subset of all atomic propositions true in state s

Temporal Logic

- Formalism describing sequences of transitions
- Time is not mentioned explicitly
- The temporal operators used to express temporal properties
 - *eventually*
 - *never*
 - *always*
- Temporal logic formulas are evaluated w.r.to a state in the model
- Temporal operators can be combined with Boolean expressions

Computation Trees



CTL – Computation Tree Logic

- Path quantifiers - describe branching structure of the tree
 - A (for *all* computation paths)
 - E (for *some* computation path = there *exists* a path)
- Temporal operators - describe properties of a path through the tree
 - X (next time, next state)
 - F (eventually, finally)
 - G (always, globally)
 - U (until)
 - R (release, dual of U)

CTL Formulas

- Temporal logic formulas are evaluated w.r.to a state in the model
- State formulas
 - apply to a specific state
- Path formulas
 - apply to all states along a specific path

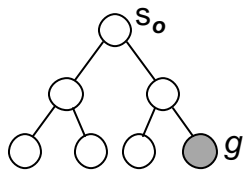
Basic CTL Formulas

- $E X (f)$
 - true in state s if f is true in some successor of s (there exists a next state of s for which f holds)
- $A X (f)$
 - true in state s if f is true for all successors of s (for all next states of s f is true)
- $E G (f)$
 - true in s if f holds in every state along some path emanating from s (there exists a path)
- $A G (f)$
 - true in s if f holds in every state along all paths emanating from s (for all pathsglobally)

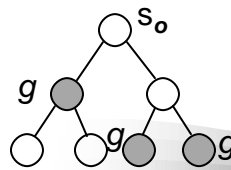
Basic CTL Formulas - cont 'd

- $EF(g)$
 - there *exists* a path which *eventually* contains a state in which g is true
- $AF(g)$
 - for *all* paths, eventually there is state in which g holds
- EF, AF are special case of $E[f U g], A[f U g]$
 - $EF(g) = E[true U g], AF(g) = A[true U g]$
- fUg (f until g)
 - true if there is a state in the path where g holds, and at every previous state f holds

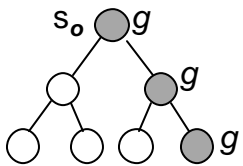
CTL Operators - examples



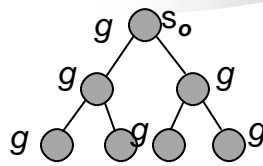
$s_0 \models EFg$



$s_0 \models AFg$



$s_0 \models EGg$



$s_0 \models AGg$

Basic CTL Formulas - cont 'd

- Full set of operators
 - Boolean: $\neg, \dot{U}, \dot{U}, \dot{A}, \rightarrow$
 - temporal: E, A, X, F, G, U, R
- Minimal set sufficient to express any CTL formula
 - Boolean: \neg, \dot{U}
 - temporal: E, X, U
- Examples:
 $f \dot{U} g = \neg(\neg f \dot{U} \neg g), \quad F f = true \ U \ f, \quad A(f) = \neg E(\neg f)$

Typical CTL Formulas

- $E \ F (start \wedge \neg ready)$
 - eventually a state is reached where *start* holds and *ready* does not hold
- $A \ G (req \rightarrow A \ F \ ack)$
 - any time *request* occurs, it will be eventually *acknowledged*
- $A \ G (E \ F \ restart)$
 - from any state it is possible to get to the *restart* state

Model Checking – Explicit Algorithm

Problem: given a structure $M(S,R,L)$ and a temporal logic formula f , find a *set of states* that satisfy f

$$\{s \in S: M, s \models f\}$$

- Explicit algorithm: label each state s with the set $label(s)$ of sub-formulas of f which are true in s .
 - $i = 0$; $label(s) = L(s)$
 - $i = i + 1$; Process formulas with $(i - 1)$ nested CTL operators. Add the processed formulas to the labeling of each state in which it is true.
 - Continue until closure. Result: $M, s \models f$ iff $f \in label(s)$

Explicit Algorithm - cont'd

- To check for arbitrary CTL formula f
 - successively apply the state labeling algorithm to the sub-formulas
 - start with the shortest, most deeply nested
 - work outwards

- Example: $E F \neg (g \dot{\cup} h)$

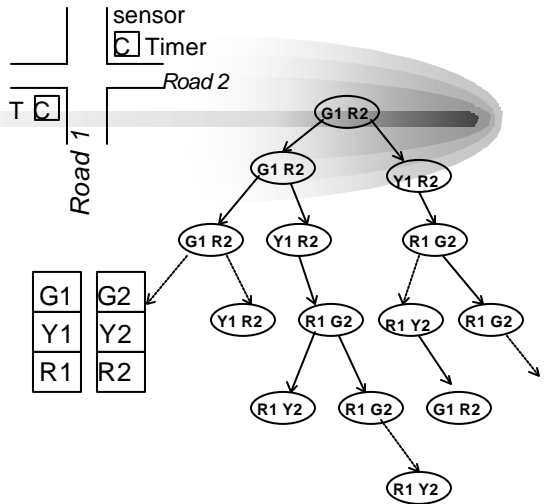
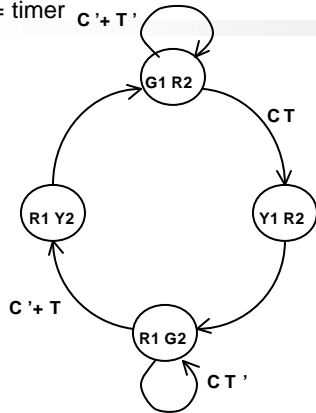
$\underbrace{\quad}_{T1} = \text{states in which } g \text{ and } h \text{ are true}$
 $\underbrace{\quad}_{T2} = \text{complement of } T1$

$T3 = \text{predecessor states to } T2$

Model Checking Example

Traffic light controller (simplified)

C = car sensor
T = timer

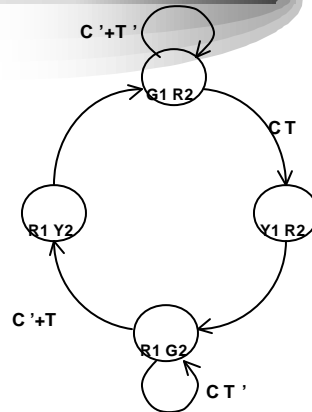


Traffic light controller - Model Checking

- Model Checking task: check
 - safety condition
 - fairness conditions
- *Safety condition*: no green lights on both roads at the same time

$$A G \neg (G1 \wedge G2)$$
- *Fairness condition*: eventually one road has green light

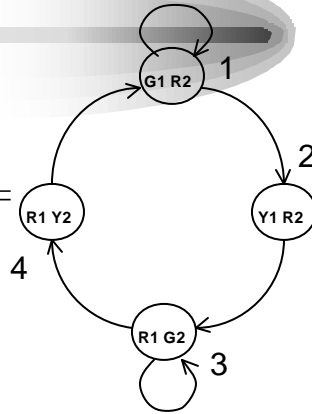
$$E F (G1 \vee G2)$$



Checking the Safety Condition

$$A G \neg (G1 \dot{\cup} G2) = \neg E F (G1 \dot{\cup} G2)$$

- $S(G1 \dot{\cup} G2) = S(G1) \dot{\cap} S(G2) = \{1\} \cap \{3\} = \emptyset$
- $S(EF (G1 \dot{\cup} G2)) = \emptyset$
- $S(\neg EF (G1 \dot{\cup} G2)) = \neg \emptyset = \{1, 2, 3, 4\}$



Each state is included in $\{1,2,3,4\}$ \bar{P}
 the safety condition is true (for each state)

Checking the Fairness Condition

$$E F (G1 \dot{\cup} G2) = E(true U (G1 \dot{\cup} G2))$$

- $S(G1 \dot{\cup} G2) = S(G1) \dot{\cap} S(G2) = \{1\} \cap \{3\} = \{1,3\}$
- $S(EF (G1 \dot{\cup} G2)) = \{1,2,3,4\}$
 (going *backward* from $\{1,3\}$, find predecessors)

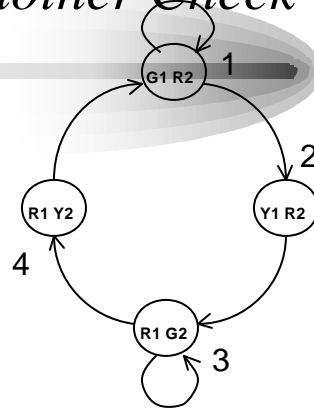


Since $\{1,2,3,4\}$ contains all states, the condition is true for all the states

Another Check

$$E X^2 (Y1) = E X (E X (Y1))$$

(starting at $S_1 = G1R2$, is there a path s.t. Y1 is true in 2 steps ?)



- $S(Y1) = \{2\}$
- $S(EX(Y1)) = \{1\}$
(predecessor of 2)
- $S(EX(EX(Y1))) = \{1,4\}$
(predecessors of 1)

Property $E X^2 (Y1)$ is true for states $\{1,4\}$, hence true

Symbolic Model Checking

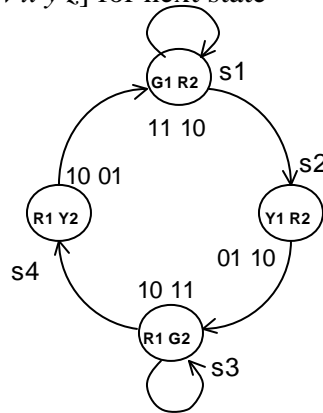
- Symbolic
 - operates on entire *sets* rather than individual states
- Uses BDD for efficient representation
 - represent Kripke structure
 - manipulate Boolean formulas
 - *RESTRICT* and *APPLY* logic operators
- Quantification operators
 - Existential: $\sum_x f = f|_{x=0} + f|_{x=1}$ (smoothing)
 - Universal: $\prod_x f = f|_{x=0} \cdot f|_{x=1}$ (consensus)

Symbolic Model Checking - example

Traffic Light Controller

- Encode the atomic propositions (G1,R1,Y1, G2,Y2,R2):
use $[a\ b\ c\ d]$ for present state, $[v\ x\ y\ z]$ for next state

	a	b	c	d
G1	1	1	--	
Y1	0	1	--	
R1	1	0	--	
G2	--	--	1	1
Y2	--	--	0	1
R2	--	--	1	0

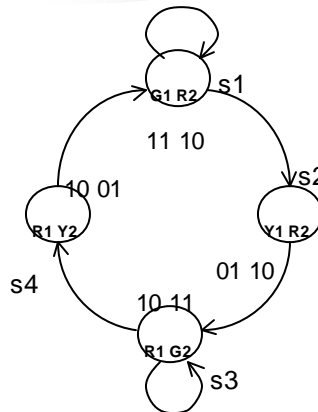


Example - cont'd

- Represent the set of states as Boolean formula
 $Q: \quad Q = abcd' + a'bcd' + ab'cd + ab'c'd$

- Store Q in a BDD

(It will be used to perform logic operations, such as $S(G1) \cup S(G2)$)

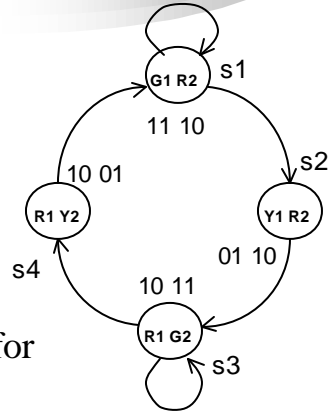


Example - cont'd

- Write a characteristic function R for the transition relation $R = abcd'vxyz' + abcd'v'xyz' + \dots + ab'c'dvxyz'$

(6 terms)

abcd	vxyz	R
1110	1110	1
1110	0110	1
0110	1011	1
1011	1011	1
1011	1001	1
1001	1110	1



- Store R in a BDD. It will be used for *Pre-Image* computation for EF.

Example - Fairness Condition

- Check fairness condition: $E F (G1 \dot{\cup} G2)$
- Step 1: compute $S(G1)$, $S(G2)$ using *RESTRICT* operator
 - $S(G1)$: $ab \cdot \text{Restrict } Q(G1) = ab Q|_{ab} = abcd' = \{s1\}$
 - $S(G2)$: $cd \cdot \text{Restrict } Q(G2) = cd Q|_{cd} = ab'cd = \{s3\}$
- Step 2: compute $S(G1) \dot{\cup} S(G2)$ using *APPLY* operator
 - Construct BDD for $(abcd' + ab'cd) = \{s1, s3\}$, set of states labeled with G1 or G2

Example – cont'd

- Step 3: compute $S(EF(G1 \cup G2))$ using *Pre-Image computation* (quantify w.r.to *next state variables*)

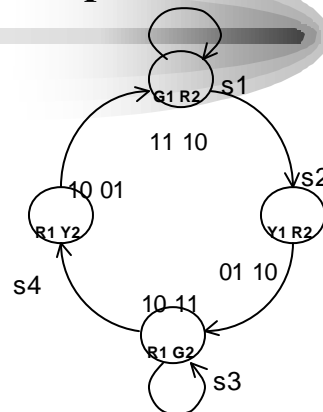
- Recall: $R = \underbrace{abcd'vxyz' + abcd'v'xyz' + \dots + ab'c'dvxyz'}_{\downarrow}$

- $S_s. \{s1',s3'\} \bullet R(s,s') =$
 $= S_{vxyz}(vxyz' + vx'yz) \bullet R(a,b,c,d;v,x,y,z)$
 $= S_{vxyz}(abcd'vxyz' + a'bcdvx'yz + ab'cdvx'yz + ab'c'dvxyz')$
 $= (abcd' + a'bcd + ab'cd + ab'c'd) = \{s1, s2, s3, s4\}$
- Compare to the result of explicit algorithm $\bar{0}$

Example – Interpretation

- Pre-Img($s1',s3',R$) eliminates those transitions which do not reach $\{s1,s3\}$

	abcd	vxyz	R
X	1110	1110	1
	1110	0110	1
	0110	1011	1
X	1011	1011	1
	1011	1001	1
	1001	1110	1



- Quantification w.r.to *next state variables* (v,x,y,z) gives the encoded *present states* $\{s1,s2,s3,s4\}$