Automatic Verification of Industrial Designs

- Based on two papers in: Workshop on Industrial-Strength Formal Specification Techniques, 1995, Boca Raton, Florida, IEEE Computer Society
  - Automatic Verification of Industrial Designs, pages 88-96
  - Timing Analysis of Industrial Real-Time Systems, pages 97-107

Successful formal methods in industry

- Formal methods are mathematical techniques that have been used in the specification and verification of computer systems.
- Want to know: Are we building the product correctly? = Verification (Different from: are we building the right product (= Validation)).

The Meaning of Formal: from Weak to Strong Formal Methods

- Pierre Wolper: International Journal on Software Tools for Technology Transfer.
- Nov. 3, 1997

Abstract

- What makes formal methods “formal”?
- Weak and strong ways of being formal: strong means: formality exploitable and exploited in software tools.

Introduction

- Bring to software development the rigor of mathematical reasoning.
- Formal methods = applied mathematics of software engineering
- Series of criteria that methods should satisfy in order to be formal

Formal methods and syntax

- Start with a high-level description = specification of the intended behavior.
- Choice of notation for expressing specification
  - English not suitable for formal methods because of ambiguity.

Criterion 1

- Decidable syntax: A language has a decidable syntax if its sentences are recognizable algorithmically. A specification language must have a decidable syntax.
- Weak requirement: satisfied by all formal methods.

Formal Methods and Semantics

- Not only syntax needs to be formal, also meaning of language.
- In general, the semantics for a language is given as a mapping from that language to another, usually simpler formalism.
- Semantics of a program: set of possible execution sequences.

Formal Methods and Semantics

- When is such a mapping formal?
- Tempting: mapping must be computable in the Turing sense. Too strong.
- Is too strong: Would imply: semantics of first-order arithmetic is not formal.
- Need something not computable, yet precise.

Criterion 2

- Formal semantics: A language has a formal semantics if deciding semantical questions for this language (e.g. equivalence of sentences) is proven to fall within the arithmetical or the analytical hierarchy.
- Also a weak requirement satisfied by most languages that claim to be formal.

Need third criterion

- Want tool support
- Should require little or no human intervention (otherwise it will not be used)
- Ok if tool sometimes does not terminate

Criterion 3

- Semantical Computational Support: A formal method provides semantical computational support if it allows software tools for checking semantical properties of specifications.
- More fuzzy than first two. But it helps to distinguish formal methods.

Classifying Formal Methods

- Weak formal methods
  - specification only formal methods
  - tool support for syntax checking only
  - write equations of a physical system
- Strong formal methods
  - tool supported semantical analysis
  - with software package to solve equations

A Strong Formal Method

- Model Checking (semantical questions are actually decidable but might have high complexity)
- Model checking without a model

More motivation for model checking

- ISSTA 1998 (March), Model Checking Without a Model:An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft, by 3 researchers from Lucent and Bell Labs.

Microsoft PowerPoint Presentation

Formal methods

- Many different specification languages and proof techniques.
- Some are difficult to apply since computers are not good at proving theorems (they need a lot of human help)
- Exception: Symbolic Model Checking: Fast, based on OBDD techniques (Ordered Binary Decision Diagrams).

Symbolic Model Checking

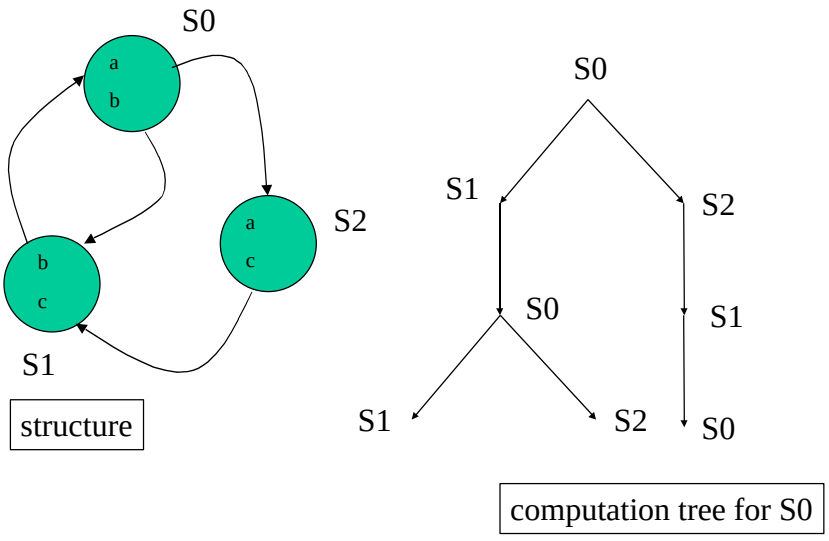
- Determine correctness of finite state systems.
- Developed at Harvard and later at CMU by Clarke/Emerson/Sistla
- Specifications are written as formulas in a propositional temporal logic.
- Temporal logic: expressing ordering of events without introducing time explicitly

Temporal Logic

- A kind of modal logic. Origins in Aristotle and medieval logicians. Studied many modes of truth.
- Modal logic includes propositional logic. Embellished with operators to achieve greater expressiveness.
- A particular temporal logic: CTL (Computation Tree Logic)

Computation Tree Logic

- Used to express properties that will be verified
- Computation trees are derived from the state transition graphs
- State transition graphs unwound into an infinite tree rooted at initial state



Computation Tree Logic

- CTL formulas built from
  - atomic propositions, where each proposition corresponds to a variable in the model
  - Boolean connectives
  - Operators. Two parts
    - path quantifier (A, E)
    - temporal operator (F,G,X,U)

Computation Tree Logic

- Paths in tree represent all possible computations in model.
- CTL formulas refer to the computation tree

$$AG(req\ implies\ AF\ ack)$$

If the signal *req* is high then eventually *ack* will also be high

Computation Tree Logic

- path quantifier (A, E)
  - A: true for all paths from a given state
  - E: true for some paths from a given state
- temporal operator (F,G,X,U)
  - Ff (f holds sometime in the future) is true of a path if there exists a state in the path that satisfies f.

Computation Tree Logic

- temporal operator (F,G,X,U)
  - Ff (f holds sometime in the future) is true of a path if there exists a state in the path that satisfies f.
  - Example: *EF(started and not ready):* It is possible to get to a state where *started* holds but *ready* does not hold.

Computation Tree Logic

- temporal operator (F,G,X,U)
  - Gf (f holds globally) is true of a path if f holds for all states in the path.
  - Example: *AG(req implies AF ack).* It is always the case that if the signal *req* is high then eventually *ack* will also be high.

Computation Tree Logic

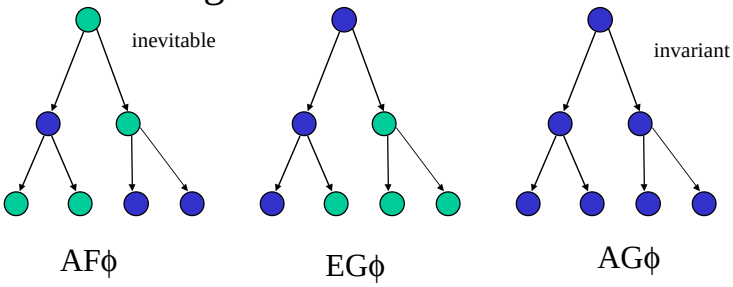
- temporal operator (F,G,X,U)
  - Xf (f holds in the next state) means that f is true in the next state.
  - $f\ Uy$ (f holds until y holds) is satisfied by a path if y is true in some state in the path, and in all preceding states, f holds.
  - Example: *AG(send implies AF[send U recv]).* It is always the case that if *send* occurs, then eventually *recv* is true, and until that time, *send* must remain true.

Computation Tree Logic

AG EF restart: From any state it is possible to get to the restart state.

Computation Tree Logic

- Examples: Dark circle indicates that a specification ϕ is true in corresponding state. Light means false.

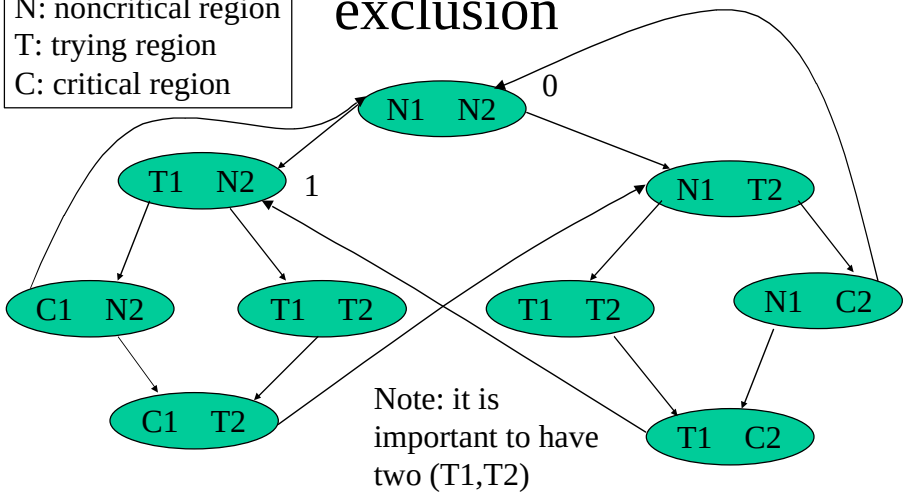


Computation Tree Logic

- Model to be verified: Finite state machine (S,R,P), where S is the finite set of all possible states, R a binary relation on S which defines the possible transitions and P assigns to each state the set of atomic propositions true in that state.
- Can verify systems with more than 10^{120} states (1995).

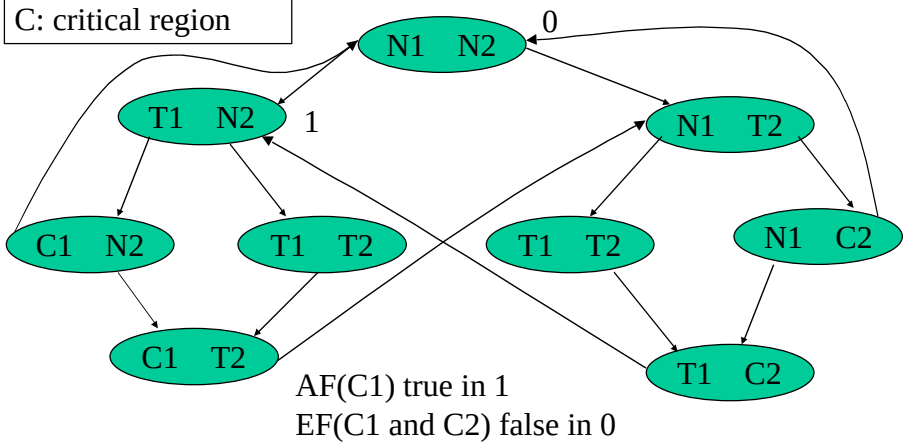
Example: two-process mutual exclusion

N: noncritical region
T: trying region
C: critical region



Example: two-process mutual exclusion

N: noncritical region
T: trying region
C: critical region



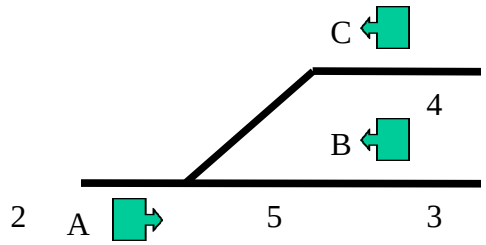
AF(C1) true in 1
EF(C1 and C2) false in 0

Model checking algorithm

- There is an algorithm for determining whether a CTL formula f is true in state s of a structure M = (S,R,P) which runs in time O(length(f))*(card(S) + card(R)))

Computation Tree Logic: Railway Interlocking Control

- Simple Interlocking Model

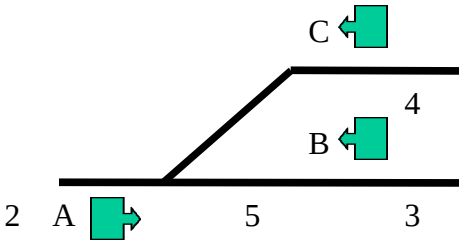


Track sections: 2,3,4,5
Control Signals: A,B,C

Avoid derailments and train crashes

Computation Tree Logic: Railway Interlocking Control

- Simple Interlocking Model



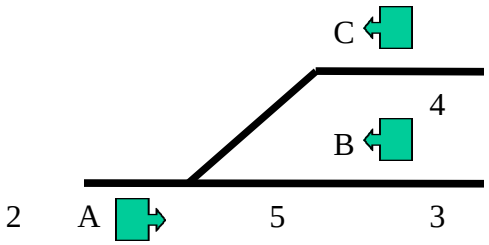
Track sections: 2,3,4,5
Control Signals: A,B,C

Inputs
2T 0 no train in 2
1 2 occupied by train or broken

Finite State Machine not shown

Computation Tree Logic: Railway Interlocking Control

- Simple Interlocking Model



Track sections: 2,3,4,5 (0: unoccupied)
Control Signals: A,B,C(0:red, 1:green)

SPEC
AG!(SignalA=1 and SignalB=1)
AG!(SignalA=1 and SignalC=1)
AG(2T=0 implies AX SignalA=0)

Output from checker

- `Specification AG(SignalA=1 and …) is false as demonstrated by the following execution sequence`
  - `state 1.1`
  - `state 1.2`
  - …
- Gives counterexample if there is one.

Computation Tree Logic: Implementation: BDDs

- Binary Decision Diagrams
  - A canonical representation for Boolean formulas (canonical = in simplest or standard form).
  - Invented by Randal Bryant, now at CMU.
  - Similar to a binary decision tree, but structure is a dag rather than a tree. Allows nodes and substructures to be shared.

Applications

- VLSI design
- Verification and equivalence checking of sequential machines
- Finding a satisfying assignment for a Boolean formula
- Checking whether two Boolean functions are identical

BDD Definition

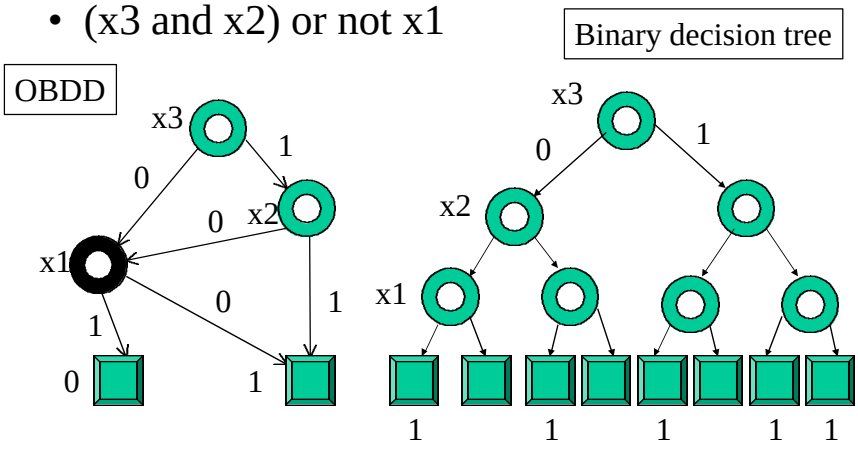
- A BDD is a directed acyclic graph with two terminal nodes (0-terminal, 1-terminal). Each non-terminal node has an index to identify an input variable of the Boolean function and has two outgoing edges, called the 0-edge and the 1-edge.

OBDD Definition

- A OBDD is a BDD where input variables appear in a fixed order in all paths of the graph and no variable appears more than once on a path.

Computation Tree Logic: Implementation: BDDs

- (x3 and x2) or not x1



Reduced ordered BDD: ROBBD

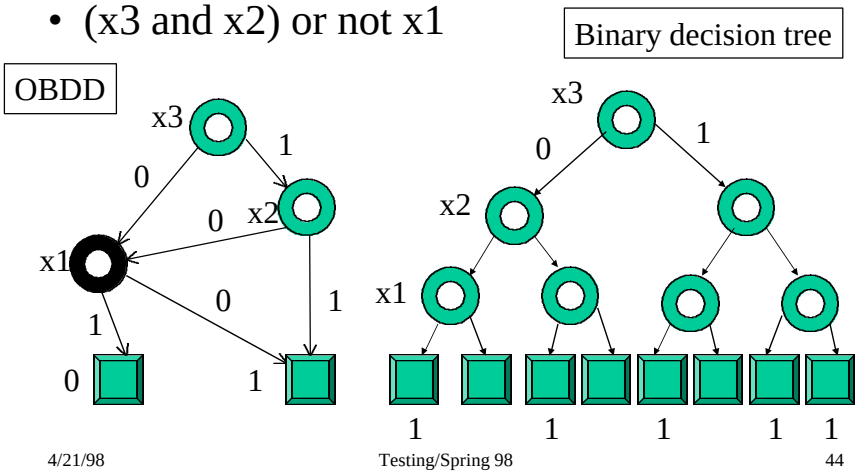
- Three reduction rules: reduced OBBD
  - only two terminal nodes (TERMINAL)
  - eliminate all the redundant nodes whose two edges point to the same node (ELIMINATION)
  - share all the equivalent subgraphs (MERGING)
- ROBDD: canonical form for fixed ordering of variables.
  - Important for equivalence checking
  - BDD now means ROBDD

Reduced OBDD

- Definition: An OBDD is called reduced if none of the three reduction rules (Terminal rule, Elimination rule, Merging rule) can be applied.
- Leads to systematic construction of BDDs from binary decision trees. Terminal rule is applied first.

BDD reduction example

- (x3 and x2) or not x1

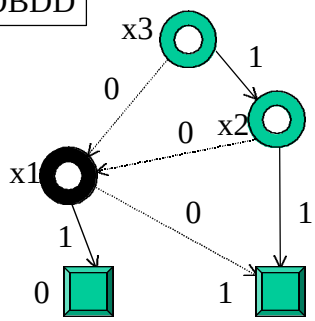


After TERMINAL

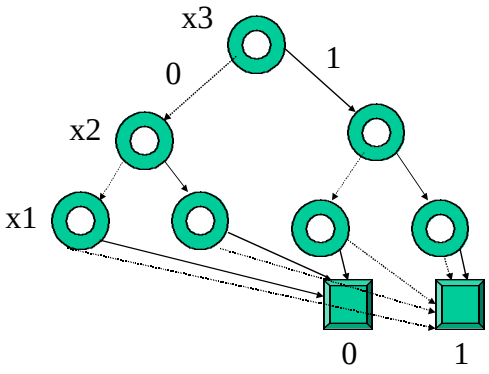
BDD reduction example

- (x3 and x2) or not x1

OBDD



Binary decision diagram

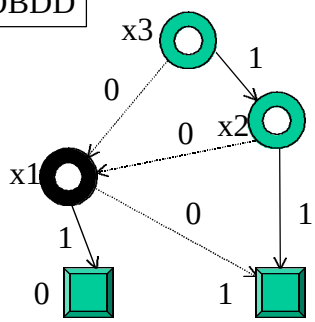


ELIMINATION

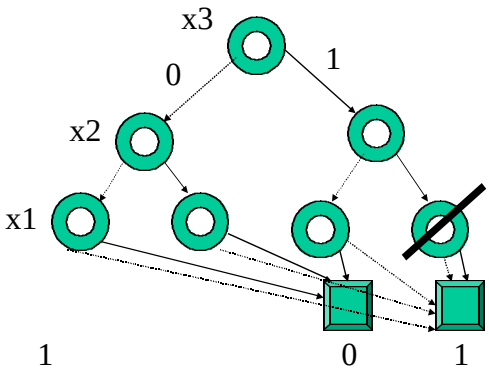
BDD reduction example

- (x3 and x2) or not x1

OBDD



Binary decision diagram

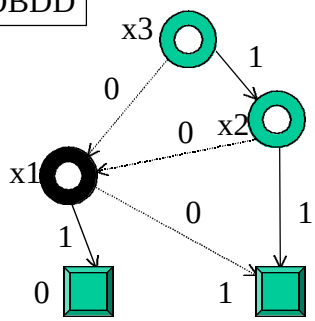


After ELIMINATION

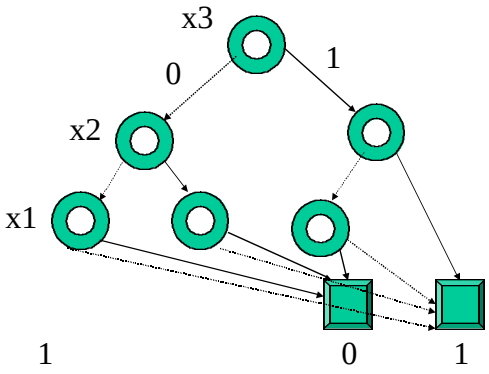
BDD reduction example

- (x3 and x2) or not x1

OBDD



Binary decision diagram

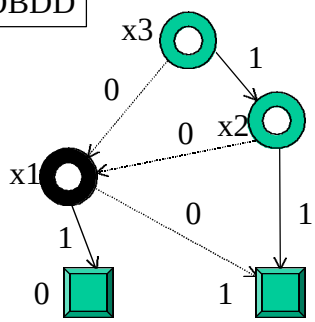


MERGING

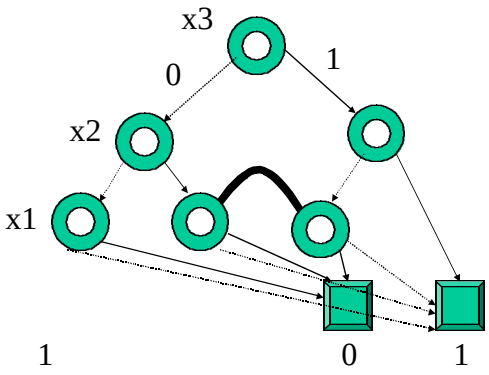
BDD reduction example

- (x3 and x2) or not x1

OBDD



Binary decision diagram

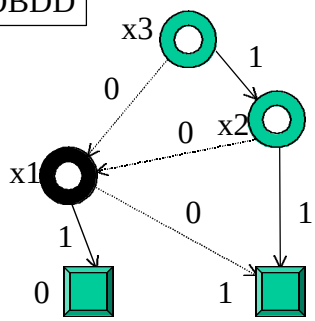


After MERGING

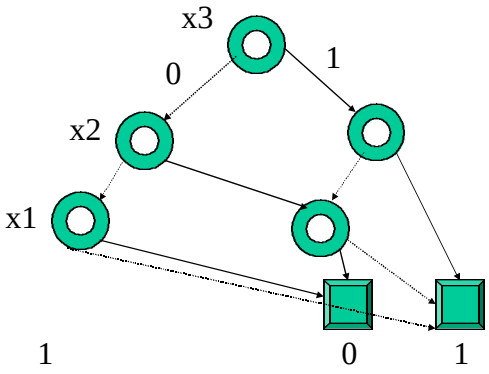
BDD reduction example

- (x3 and x2) or not x1

OBDD



Binary decision diagram

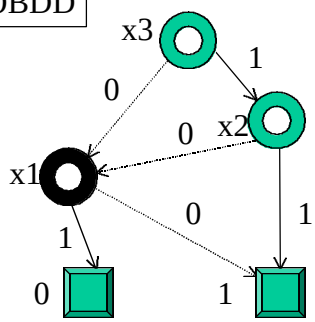


MERGING

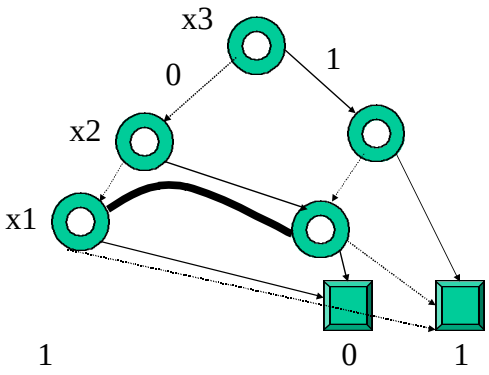
BDD reduction example

- (x3 and x2) or not x1

OBDD



Binary decision diagram

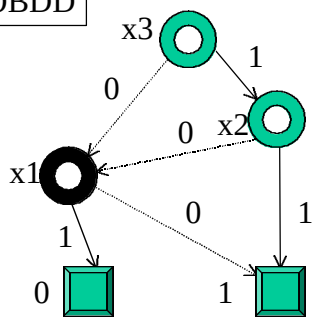


After MERGING

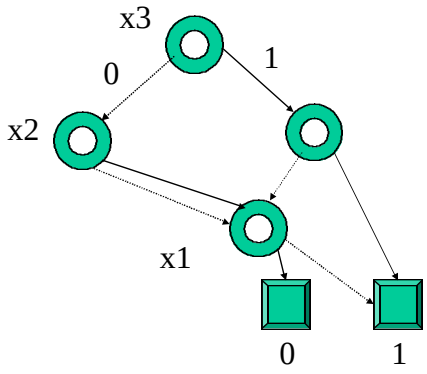
BDD reduction example

- (x3 and x2) or not x1

OBDD



Binary decision diagram

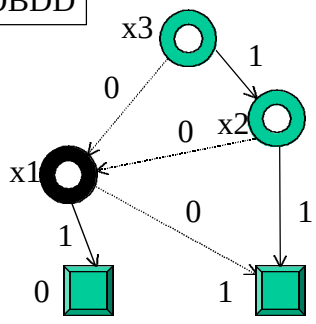


ELIMINATION

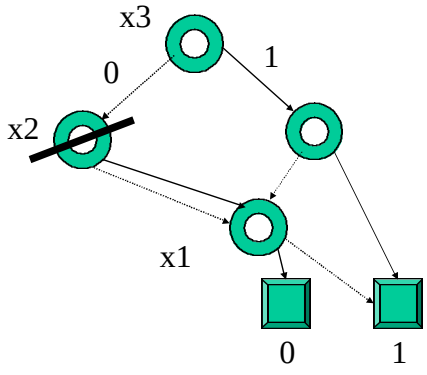
BDD reduction example

- (x3 and x2) or not x1

OBDD



Binary decision diagram

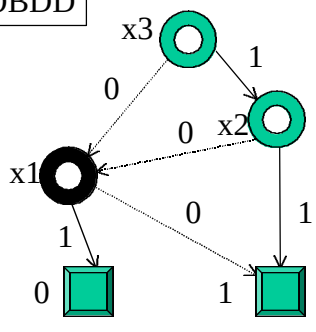


After ELIMINATION

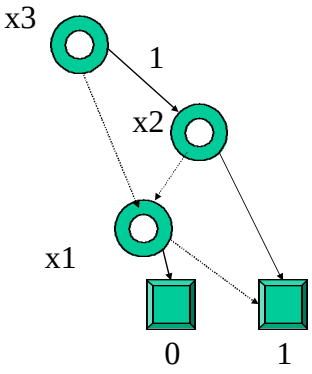
BDD reduction example

- (x3 and x2) or not x1

OBDD



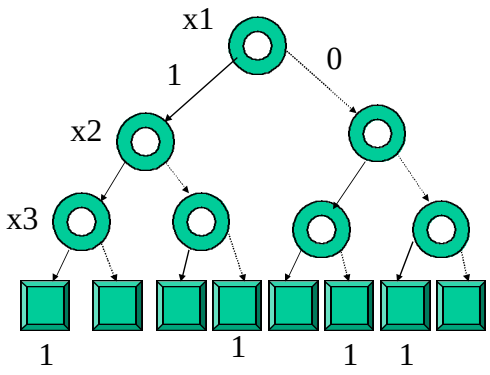
Binary decision diagram



BDD reduction example for exclusive-or function

x1⊕ x2 ⊕ x3
exclusive-or
odd parity function

Binary decision tree

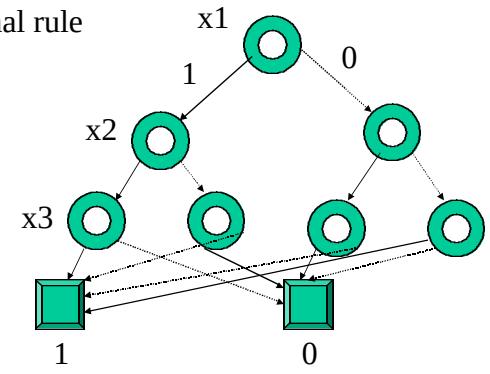


After TERMINAL

BDD reduction example

Binary decision tree

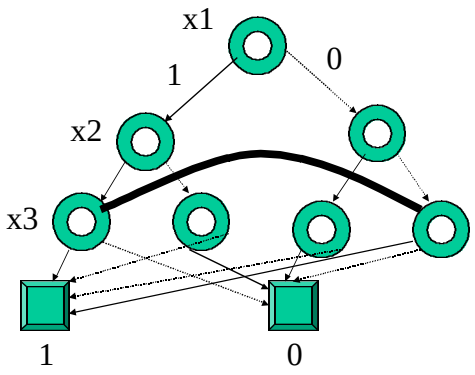
After applying terminal rule



MERGING

BDD reduction example

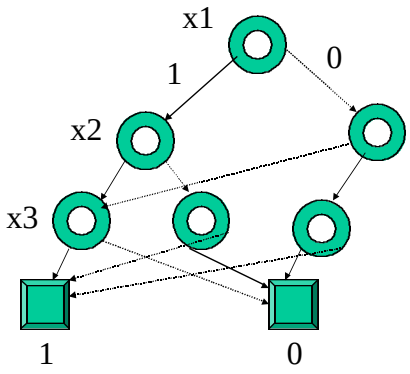
Binary decision tree



After MERGING

BDD reduction example

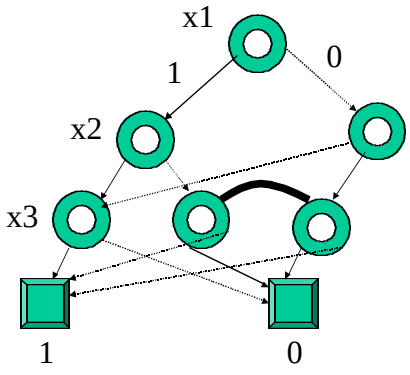
Binary decision tree



MERGING

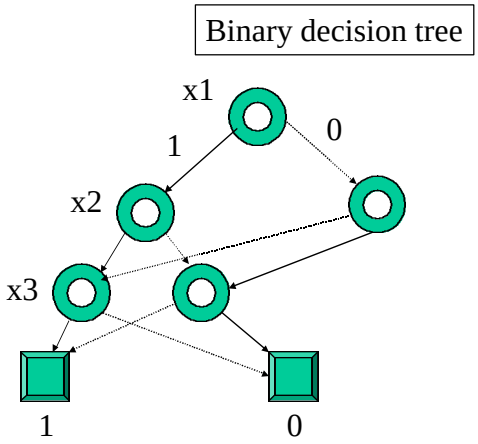
BDD reduction example

Binary decision tree



After MERGING

BDD reduction example



Uniqueness

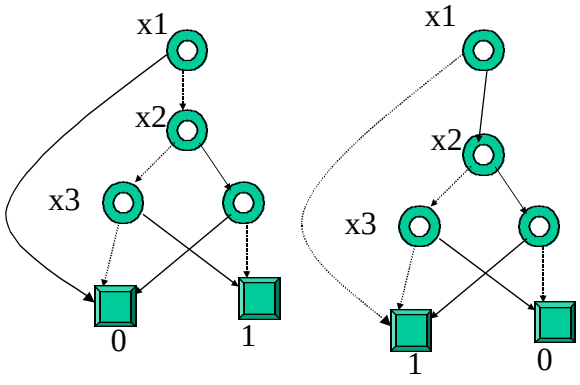
- With respect to each fixed variable order, the reduced OBDD of a Boolean function f is determined uniquely.
- Representations of Boolean functions
  - formulas, based on computation rules: not unique
  - BDDs, based on a decision process: unique if reduced

Automatically recognizing regularities efficiently

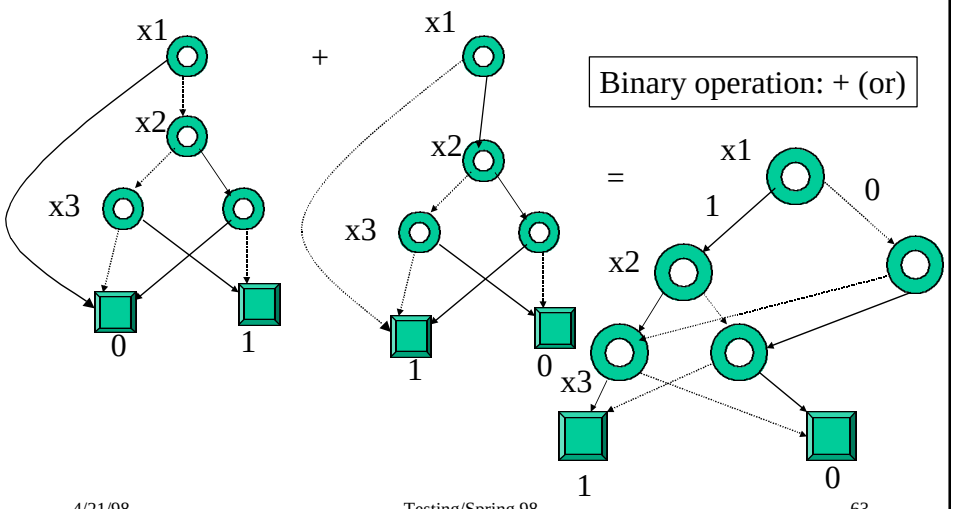
- Construction of BDDs from formulas: use Shannon's expansion:
  - f*g = x and (f(x=1) * g(x=1)) or
    !x and (f(x=0) * g(x=0))

Shannon expansion example

x1⊕ x2 ⊕ x3
exclusive-or
odd parity function
x1⊕ x2 ⊕ x3 =
x1 (1 ⊕ x2 ⊕ x3 +
!x1(0 ⊕ x2 ⊕ x3)

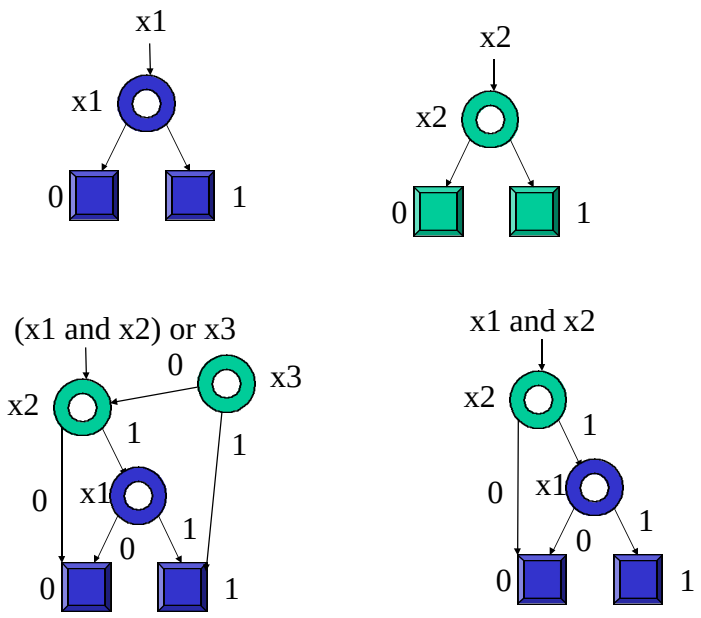


Shannon expansion example



Unary and Binary Operations

- Negation: A BDD for *not f:* exchange 0-terminal and 1-terminal. No increase in size!



Binary operations

- Let the Boolean functions f1 and f2 be represented by reduced OBDDs P1 and P2 with respect to the same variable ordering. For each binary operation * the reduced OBDD P of f1*f2 can be determined in time O(size(P1) * size(P2)).

Size of BDDs

- *n*-input Boolean functions $2^{(2^n)}$
- Require 2^n bits in worst-case
- Truth tables always require 2^n bits
- Many practical functions require much less space in BDD representation.

Regularities in Boolean functions

- A Boolean function has *high* regularity if for some variable ordering its BDD (reduced ordered binary decision diagram) is *small* compared to the size of the decision tree.
- A Boolean function has *high* regularity if for some variable ordering *many* reduction steps can be applied to its decision tree.

Regularities in Boolean functions

- A Boolean function has *high* regularity if for some variable ordering its BDD has a size comparable to the size of its formula representation.
- What is then the benefit of going BDD?
  - Unique representation: easy equality test.
  - Finding a satisfying assignment is easy.

Why BDDs?

- Classic representations: truth tables, disjunctive normal forms, conjunctive normal forms, general Boolean formulas, net-list of gates.
- Testing whether two disjunctive normal forms, conjunctive normal forms, general Boolean formulas, or net-list of gates are equivalent is co-NP complete. NOT GOOD

Regularities in Boolean functions

- Many practically occurring Boolean functions have high regularity.
- Proper variable ordering can make exponential difference.
- Some Boolean functions are not regular: multiplication of two n-bit Boolean numbers has exponential size BDD for every variable ordering.

Regularities in Boolean functions

- Finding optimal variable order is NP-hard.
- Some good heuristics are available.
- Regularities and compact representations are also important in other areas of computer science.

Regularities in two areas of computer science

- **BDD** for **function** f
  - *often high regularity* for functions occurring in practice: BDD is small
  - *sometimes low regularity*: BDD is big
  - *benefit:* excellent algorithmic properties: equivalence, satisfiability, etc. easy
- **Strategy** for **traversal** t in graph G
  - *often high regularity* for traversals occurring in practice: strategy is small
  - *sometimes low regularity*: strategy is big
  - *benefit:* shorter, more flexible programs

Regularities in two areas of computer science

- **BDD** for **function** f
  - *non-compact representation:* truth table
- **Strategy** for **traversal** t in graph G
  - *non-compact representation:* regular expression describing traversal (without needing graph G)

Satisfying assignment

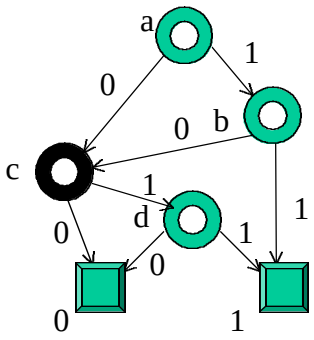
- A path from root to 1-terminal. Can be found in time proportional to the number of input variables.
- Count number of satisfying assignments in time proportional to the number of nodes in the BDD.

Exercise

- Write a BDD for the equality function for n=3 Boolean variables.

Computation Tree Logic: Implementation: BDDs

- Binary Decision Diagrams



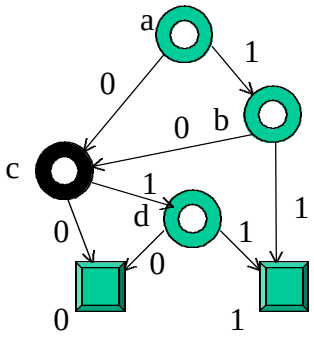

```
a b c d result
1 1       1
      1 1 1
0     1 1 1
1 0 1 1 1
```

What is Boolean formula?

All paths to 1

Computation Tree Logic: Implementation: BDDs

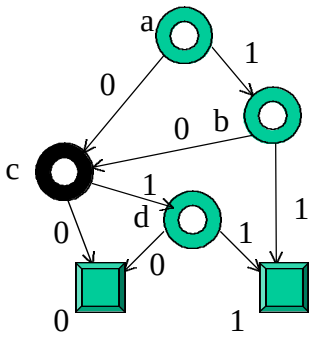
- Binary Decision Diagrams



Given a variable ordering, the BDD for a formula is unique.
There are efficient algorithms to compute the BDD for *not f* and *f or g* given the BDD of *f* and *g*.

Computation Tree Logic: Implementation: BDDs

- Binary Decision Diagrams



For the purpose of model checking also need to compute BDD of restricted formulas. Bryant describes an algorithm for computing the BDD of a restricted formula such as *f, where v=0.*

Summary BDDs

- Many applications in computer-aided design
- Moral of the story: appropriate data structures are very important for efficient algorithms
- The difference can be exponential in size for the currently best algorithms: satisfiability

Summary BDDs

- BDDs don’t always provide a compact representation (2 n-bit multiplier!). But they work well in many cases.
- BDDs improve the performance of many design systems substantially.
- Now back to the CTL application of BDDs.

References

- EATCS bulletin: Survey and tutorial by Christoph Meinel and Thorsten Theobald: Ordered Binary Decision Diagrams and Their Significance in Computer-Aided Design of VLSI Circuits, pages 171-187, year probably 1997, issue unknown.

References

- S. Minato, Binary Decision Diagrams and Applications for VLSI CAD, Kluwer Academic Publisher, 1996.

Computation Tree Logic: Implementation: BDDs

- Binary Decision Diagrams: All Boolean formulas are represented by BDDs. BDDs built in a bottom-up manner.
  - The set of atomic formulas is precisely the set of state variables. (BDD for an atomic variable = one BDD variable)
  - Formulas are built from atomic formulas using Boolean connectives. Allows CTL formulas.

Symbolic Model Checking

- Determine correctness of finite state systems.
- Specifications are written as formulas in a propositional temporal logic.
- Models to be checked are represented by state transition graphs
- Verification is accomplished by an efficient breadth-first search.

Symbolic Model Checking

- View transition system as model of logic.
- Verify whether specifications are satisfied for model.
- Advantages:
  - completely automatic
  - provides counterexamples (execution trace which shows why formula is not true)
  - verify partially specified systems

Symbolic Model Checking

- Model checkers achieve great efficiency through the use of symbolic implementation techniques
- represent states and transitions through Boolean formulas in BDD form

Symbolic Model Checking

- Representing the Model
  - Labeled state-transition graph M.
  - Use BDDs to represent graph and check whether formula holds.
  - Behavior determined by variables V

$$V = v_0, \ldots, v_{n-1} \quad V' = v_0', \ldots, v_{n-1}'$$

Symbolic Model Checking

- Representing the Model
  - Behavior determined by variables V
    - current state
  - V' = Second copy of variables
    - next state

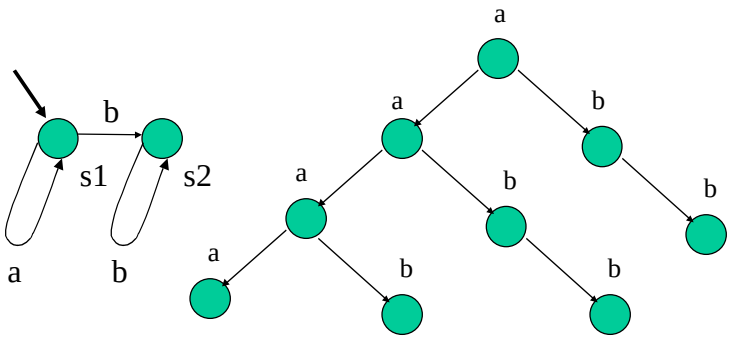
$$V = v_0, \ldots, v_{n-1} \quad V' = v_0', \ldots, v_{n-1}'$$

Symbolic Model Checking

- Representing the Model: Relationship between variables in the current state and the next states is written as a formula using V and V'. Boolean formula N representing transition relation. Convert to BDD.

$$V = v_0, \ldots, v_{n-1} \quad V' = v_0', \ldots, v_{n-1}'$$

Computation Tree Logic



State transition graph and corresponding computation tree
Paths in tree represent all possible computations

Computation Tree Logic

- Used to express properties that will be verified
- Computation trees are derived from the state transition graphs
- State transition graphs unwound into an infinite tree rooted at initial state

Design and synthesis of synchronization skeletons

- Edmund Clarke and Allen Emerson, Logics of Programs 1981, LNCS 131, page 52-71.
- Synthesize synchronization skeleton from a temporal logic specification.
- Skeleton: detail irrelevant to synchronization is suppressed.

Exercise

- Design a finite state machine with start state *s* and final state *t* and prove that for all transitions from *s* to *t* any encounter of state *y* is preceded by encountering first state *x*.
- Run your model and specification with the model checker on the CMU model checking home page.

Application of CTL: Traversal specifications and CTL

- What are the connections, if any? How can CTL ideas be used for traversals?
- F modal operator has flavor of structure-shyness. *"When starting in state A eventually we will get to state B":* sounds like "from A to B".

Result

- Can use a subset of CTL to express graph constraints corresponding to traversal specifications.
- Need to modify class graph so that every node has an outgoing edge. CTL works with infinite paths.
- Model synthesis algorithm for CTL might be useful for type checking adaptive programs.

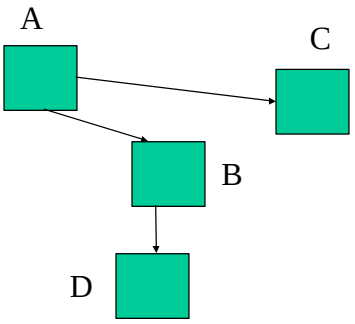
CTL for defining path sets in a graph

- Atomic variable for each state s
  - s true: we are in state s
  - s false: we are not in s
- Exists path from s to t: AG(s=>EF(t))
  - if false: no path from s to t
  - if true: describes set of state transitions leading from s to t = path set from s to t

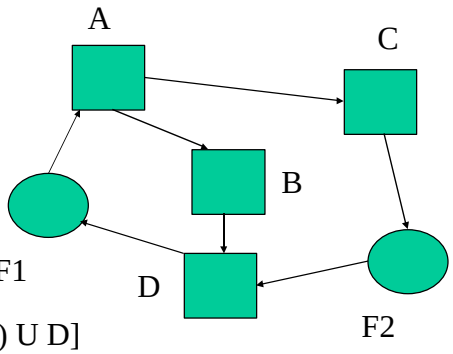
CTL for defining path sets in a graph

- Idea: express traversals with E quantifier.
- Quantifier claims existence of paths and defines set of paths.
- CTL formula both as constraint and as definer of a set of paths (all paths satisfying constraint).

Problem: state transition relation must be total in CTL



Make graph cyclic
Graph M



Graph must satisfy:

M,A Æ E[(not F1 and not F2) U D]
From A bypassing {F1,F2} to D

CTL for defining path sets in a graph

- Exist path from s to t: AG(s=>EF(t))
  - if false: no path from s to t
  - if true: describes set of state transitions leading from s to t = path set from s to t
  - there is also s0 involved: M,s0 Æ AG(s=>EF(t))
  - simpler: M,s Æ EF(t)

CTL for defining path sets in a graph

- Exists path from s bypassing y to t : AG(s=>EF(!y U t))
  - if s is true then on some path eventually t is true and until that time y must be false.
  - is a constraint on graphs
  - (given a set of CTL formulas, there is an algorithm to construct a model from formulas (Clarke/Emerson 81)).

CTL for defining path sets in a graph

- Exists path from s bypassing y to t :
  - M,s Æ EF(!y U t)
  - on some path from s eventually t is true and until that time y must be false.
  - is a constraint on graphs

CTL for defining path sets in a graph

- Exists path from s to t: M,s Æ EF(t)
- Exists path from t to u: M,t Æ EF(u)
- Exists path from s via t to u:
  - M,s Æ EF(t) and M,t Æ EF(u)
- Following is different: Exists path from s via t to u:
  - AG(s=>EF(t)) and AG(t=>EF(u))

End of expressing traversals with CTL formulas

- An interesting connection between temporal logic and compact representation of path sets in graphs.

Next: a more precise definition of CTL

- CTL very useful for verifying finite state systems

Definition of CTL

- Formulas
  - Every atomic proposition p in AP (atomic propositions) is a CTL formula.
  - If f1 and f2 are CTL formulas, then so are not f1, f1 and f2, f1 or f2, AXf1, EXf1,A[f1 U f2], E[f1 U f2].
  - X next-time operator
  - U until operator

Definition of CTL

- Formulas
  - AXf1: f1 holds in **every** immediate successor of the current program state
  - EXf1: f1 holds in **some** immediate successor of the current program state

Definition of CTL

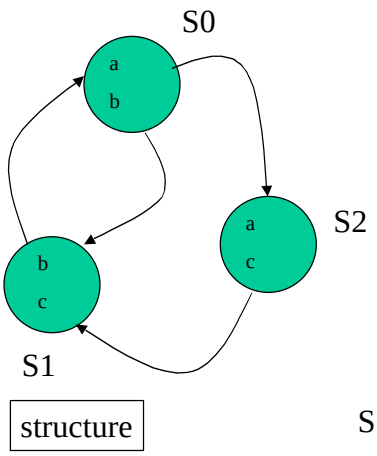
- Formulas
  - A[f1 U f2] : for **every** computation path there exists an initial prefix such that f2 holds at the last state of the prefix and f1 holds at all other states along the prefix.
  - E[f1 U f2] : for **some** computation path there exists an initial prefix such that f2 holds at the last state of the prefix and f1 holds at all other states along the prefix.

Semantics of CTL

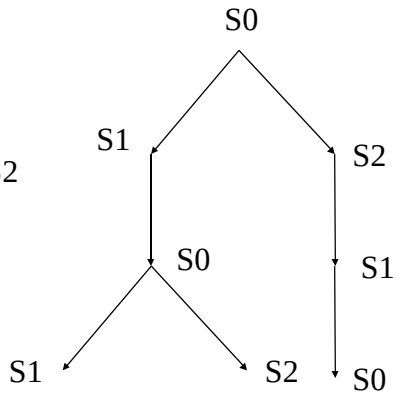
- With respect to a labeled state transition graph. A CTL structure is a triple M = (S,R,P) where
  - S a finite set of states
  - R is a binary relation on S ($R \subseteq S \times S$) which must be total: $\forall x \in S \exists y \in S[(x,y) \in R]$
  - P: $S \rightarrow 2^{AP}$ assigns to each state the set of atomic propositions true in that state

Semantics of CTL

- A path is an infinite sequence of states (s0,s1, …) such that for all i $[(si, si+1) \in R]$.
- For any structure M=(S,R,P) and state s0 in S, there is an infinite computation tree with root labeled s0 such that $s \rightarrow t$ is an arc in the tree iff $(s,t) \in R$.



structure



computation tree for S0

Semantics of CTL

- M,s0Æ f means that formula f holds at state s0 in structure M.
- When M is understood: s0Æ f
- Inductive definition for Æ
  - s0 Æ p iff p ∈ P(s0)
  - s0 Æ not p iff not(s0 Æ p)
  - s0 Æ f1 and f2 iff s0 Æ f1 and s0 Æ f2

Semantics of CTL

- Inductive definition for Æ
  - s0 Æ AX f1 iff for all states t such that (s0,t)∈R, t Æ f1
  - s0 Æ EX f1 iff for some state t such that (s0,t)∈R, t Æ f1

Semantics of CTL

- Inductive definition for Æ
  - s_0 Æ A[f_1 U f_2] iff for all paths (s_0,s_1,…), ∃ i [i>=0 and s_i Æ f_2 and ∀ j[0<=j<i => s_j Æ f_1]]
  - s_0 Æ E[f_1 U f_2] iff for some path (s_0,s_1,…), ∃ i [i>=0 and s_i Æ f_2 and ∀ j[0<=j<i => s_j Æ f_1]]

Abbreviations

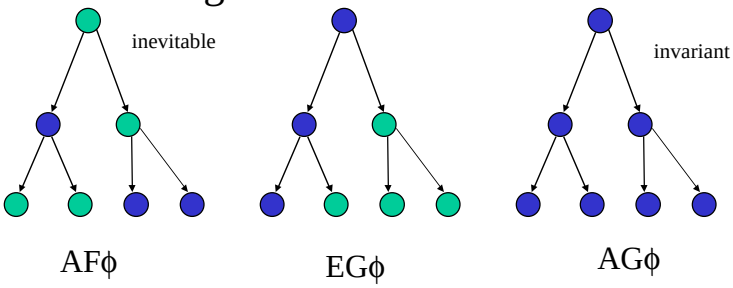
- AE(f) = A[True U f]
  - intuition: f holds sometime in the future along every path from s0: f is *inevitable*.
  - True: true in all states
- EF(f) = E[True U f]
  - intuition: there is some path from s0 that leads to a state at which f holds: f *potentially* holds.

Abbreviations

- EG(f) = not AF[not f]
  - intuition: there is some path from s0 on which formula f holds at every state.
- AG(f) = not EF[not f]
  - intuition: on all paths from s0 formula f holds at every state.

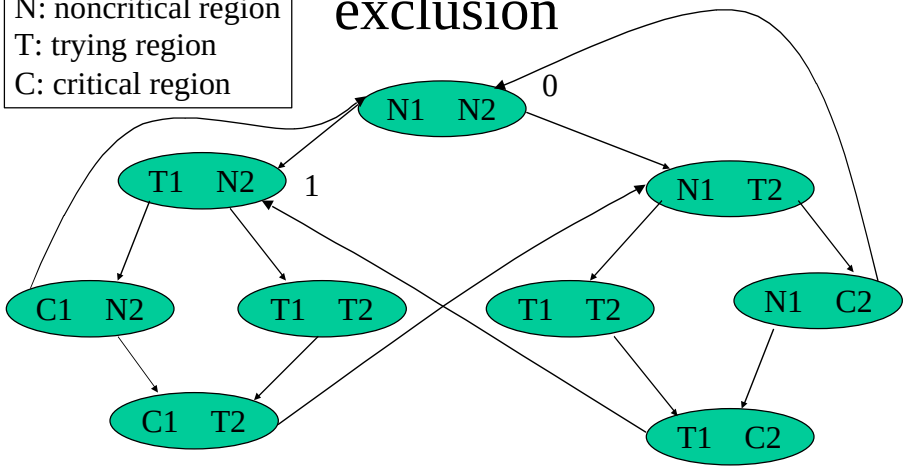
Computation Tree Logic

- Examples: Dark circle indicates that a specification ϕ is true in corresponding state. Light means false.



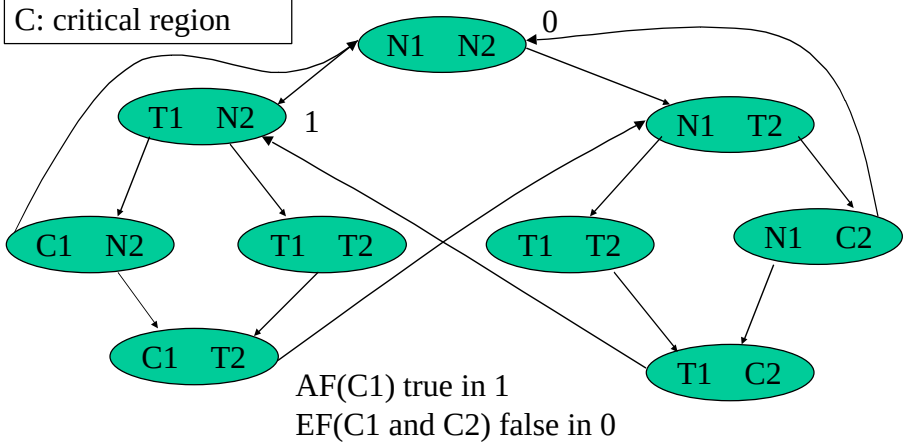
Example: two-process mutual exclusion

N: noncritical region
T: trying region
C: critical region



Example: two-process mutual exclusion

N: noncritical region
T: trying region
C: critical region



AF(C1) true in 1
EF(C1 and C2) false in 0

Expressing deadlock

- AG(no_next_state => finished)
- no_next_state = AX False
- False is false in all states
- AG(AX False => finished)