



ITI8531: Lecture 4
Module I: Model Checking
Topic: CTL model checking algorithms

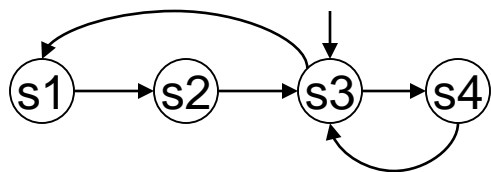
J.Vain
23.02.2017



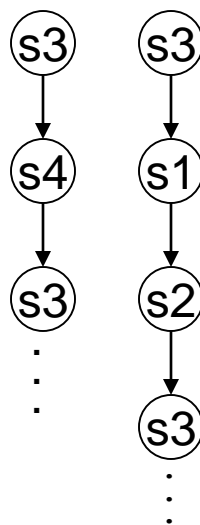
Recall: Linear Time vs. Branching Time

- In linear time logics we look at execution paths individually
- In branching time logics we view the computation alternatives as a tree
 - computation tree unfolds the transition relation

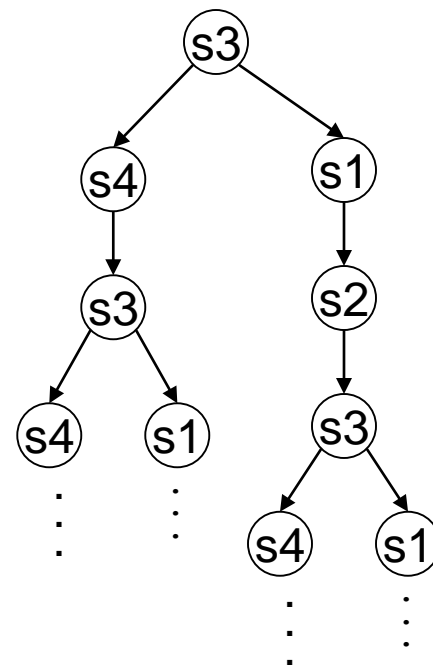
Transition System



Execution Paths



Computation Tree

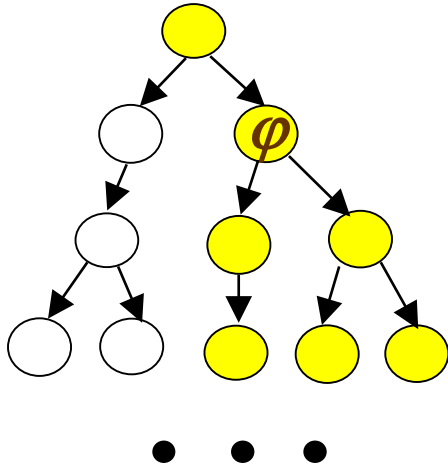


Recall: Computation Tree Logic (CTL)

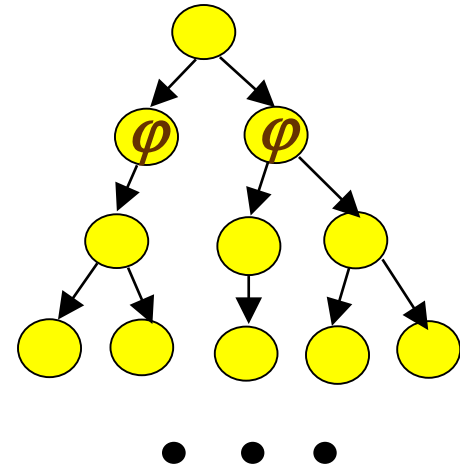


- In CTL we quantify over the paths in the computation tree
- We use the same temporal operators as in LTL: X, G, F, U
- We attach path quantifiers to these temporal operators:
 - A : for all paths
 - E : there exists a path
- We end up with eight temporal operator pairs:
 - AX, EX, AG, EG, AF, EF, AU, EU

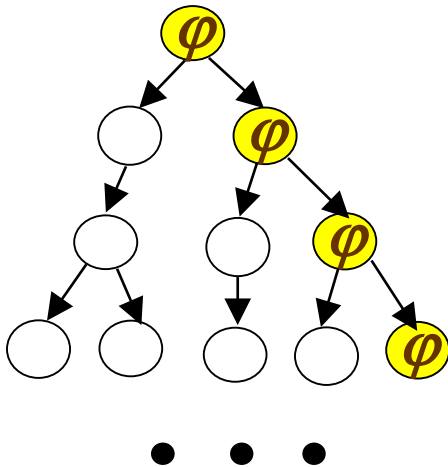
Examples



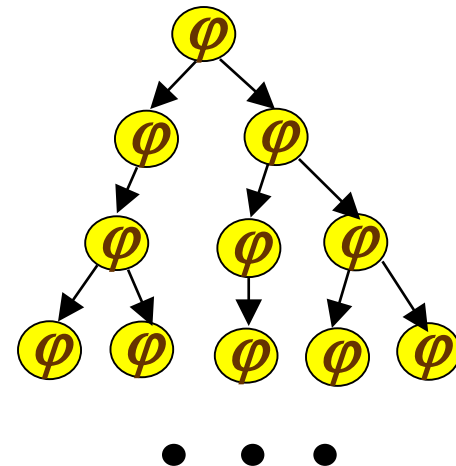
EX φ (exists next)



AX φ (all next)

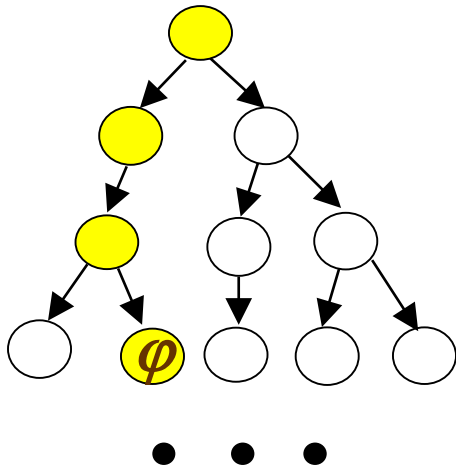


EG φ (exists global)

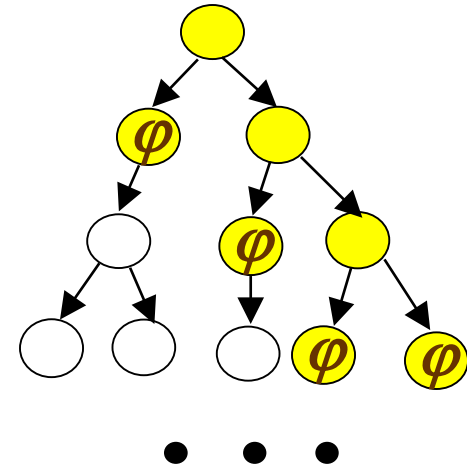


AG φ (all global)

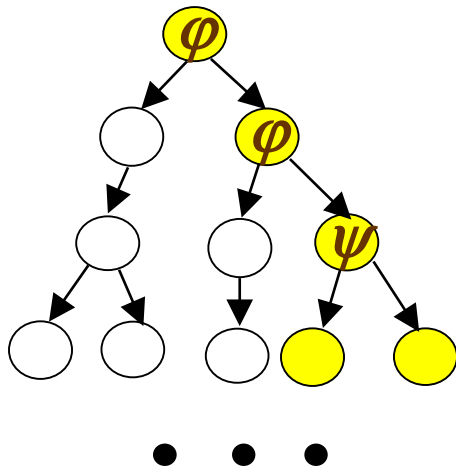
Examples (continued)



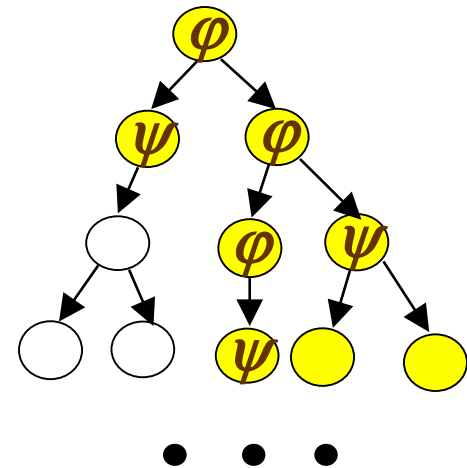
EF φ (exists future)



AF φ (all future)



φ EU ψ (exists until)



φ AU ψ (all until)

Automated Verification of Finite State Systems

[Clarke and Emerson 81], [Queille and Sifakis 82]



- CTL Model checking problem:

Given a transition system $T = (S, I, R)$, and a CTL formula φ , does the transition system T satisfy the property φ ?

CTL model checking problem can be solved in

$$O(|\varphi| \times (|S| + |R|))$$

Note:

- the complexity is linear in the size of the transition system T
 - the complexity is exponential in the number of variables of φ , and S is exponential in the number of concurrent components of T
- This is called the **state space explosion** problem.

CTL Model Checking Algorithm (1)



- Translate the formula to a formula which uses only the basis of CTL

$EX \varphi, EG \varphi, \varphi EU \psi$

- $EF \varphi == E(\text{true} U \varphi)$ (because $F \varphi == [\text{true} U(\varphi)]$)
- $AX \varphi == \neg EX(\neg \varphi)$
- $AG \varphi == \neg EF(\neg \varphi) == \neg E[\text{true} U (\varphi)]$
- $AF \varphi == A[\text{true} U \varphi] == \neg EG(\neg \varphi)$
- $A[\varphi U \psi] == \neg(E[(\neg \psi) U \neg(\varphi \vee \psi)] \vee EG(\neg \psi))$

CTL Model Checking Algorithm (2)



- Key idea of the algorithm
 - Label the states S of KS with atomic propositions from set AP .
 - Label the states S with subformulas of φ that hold in these states (start from the innermost non-atomic subformulas of φ).
- Properties of the algorithm:
 - Each (temporal or boolean) operator has to be processed only once.
 - Graph traversal algorithms (DFS or BFS) are used to find the labeling in KS for each operator.
 - Computation of each sub-formula takes at most $O(|S|+|R|)$ traversal steps.

CTL Model Checking Algorithms: intuition



- **EX φ** can be done in $O(|S|+|R|)$
 - All the nodes which have a next state labeled with φ should be labeled with EX φ
- **φ EU ψ** : Find the states which are the initial states of a path where φ U ψ holds.

Equivalently,

 - find the nodes which reach ψ labeled node by a path where each node is labeled with φ
 - Label such nodes with φ EU ψ

It is a reachability problem which can be solved in $O(|S|+|R|)$

CTL Model Checking Algorithms: intuition



EG φ :

Find the paths where each node is labeled with φ and label the nodes in such paths with EG φ :

- First remove all the states which do not satisfy φ from the transition graph
- Compute the connected components of the remaining graph
- then find the nodes which can reach the connected components. (both of which can be done in $O(|S|+|R|)$)
- Label the nodes with EG φ in the connected components and the nodes that can reach the connected components.

Verification vs. Falsification



- **Verification:**
 - Show that initial states are included in the truth set of φ
- **Falsification:**
 - Find if a state is included in the initial states \cap truth set of $\neg\varphi$
 - Generate a counter-example starting from that state
- CTL model checking algorithm can also generate a counter-example path (if the property is not satisfied) without increasing the complexity
- The ability to find counter-examples is one of the biggest strengths of model checkers

Problems with the previous algorithm



It is named *explicit state* model checking

- All the states and labels associated to the states must be recorded when doing state space traversal
 - needs a lot of memory
 - causes **exponential explosion** of required memory because
 - the number of states $|S|$ in the transition graph T is exponential in the number of variables and the concurrent processes in the system modelled with LTS.

LTS – Labeled Transition System

(KS is a simple form of LTS)

Introduction to symbolic state model checking



- How to deal with exponential explosion of the memory space of CTL model checking???

Characterization of Temporal operators as Fixpoints

[Emerson & Clarke 80]: Think about temporal op-s as recursive functions on sets



Here are some interesting CTL equivalences (for a state of computation tree)

$$\begin{array}{l} \text{value} \qquad \qquad \text{function} \\ \text{AG } \varphi = \varphi \wedge \text{AX } \text{AG } \varphi \\ \text{EG } \varphi = \varphi \wedge \text{EX } \text{EG } \varphi \end{array}$$

argument

$$\begin{array}{l} \text{AF } \varphi = \varphi \vee \text{AX } \text{AF } \varphi \\ \text{EF } \varphi = \varphi \vee \text{EX } \text{EF } \varphi \end{array}$$

$$\begin{array}{l} \varphi \text{AU } \psi = \psi \vee (\varphi \wedge \text{AX } (\varphi \text{AU } \psi)) \\ \varphi \text{EU } \psi = \psi \vee (\varphi \wedge \text{EX } (\varphi \text{EU } \psi)) \end{array}$$

Note:

We rewrite the CTL temporal operators using the operators themselves and EX and AX operators.

Functionals (mapping a set to a set)



- Given a transition system $T = (S, I, R)$, we will define functions from sets of states to sets of states
 - $f: 2^S \rightarrow 2^S$ 2^S – set of subsets of S
- For example, one such function is the EX operator. It computes the “pre-image” of a set of states given a relation R .
 - $EX: 2^S \rightarrow 2^S$
 - which can be defined as:
 $EX(\varphi) = \{ s \mid (s, s') \in R \text{ and } s' \in [[\varphi]] \}$

Abuse of notation:

Generally, $[[\varphi]]$ denotes the set of states which satisfy the property φ , i.e., the truth set of φ . Here we use just φ in the same sense.



Functionals

- Now, we can think of all temporal operators also as functionals from sets of states to sets of states
- For example,
in **logic notation**:

$$AX \varphi = \neg EX(\neg \varphi)$$

or if we use **set notation**

$$AX \varphi = (S - EX(S - \varphi))$$

Abuse of notation: we will use the set and logic notations interchangeably keeping in mind the correspondence

<u>Logic</u>		<u>Set</u>
<i>false</i>		\emptyset
<i>true</i>		S
$\neg \varphi$	\rightleftarrows	$S - \varphi$
$\varphi \wedge \psi$	\rightleftarrows	$\varphi \cap \psi$
$\varphi \vee \psi$		$\varphi \cup \psi$



Temporal Properties as Fixpoints (1)

Based on the equivalence $\text{EF } \varphi \equiv \varphi \vee \text{EX } \text{EF } \varphi$
we observe that $\text{EF } \varphi$ is a fixpoint of the following function:

$$f y = \varphi \vee \text{EX } y, \quad \text{where } y = \text{EF } \varphi$$

$$\text{i.e., } f y = y$$

In fact, $\text{EF } \varphi$ is the least fixpoint of f , which is written as:

$$\text{EF } \varphi = \mu y . \varphi \vee \text{EX } y$$

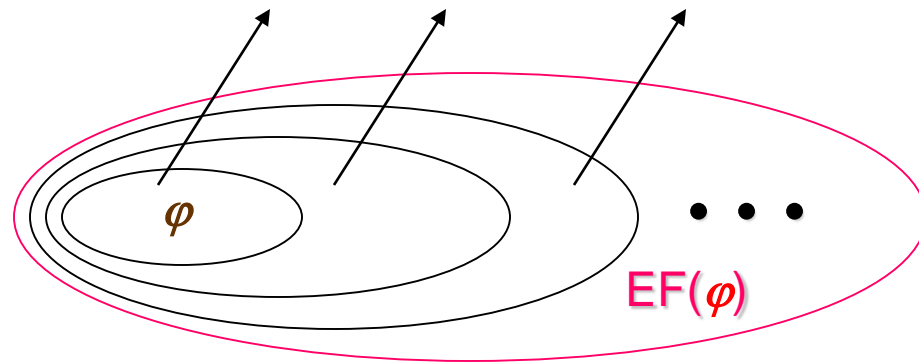
function *argument*

*The value of function is
fixpoint if it equals to the
argument*

EF Fixpoint Computation



$EF(\varphi) \equiv$ states from where φ is reachable $\equiv \varphi \cup EX(\varphi) \cup EX(EX(\varphi)) \cup \dots$



Temporal Properties as Fixpoints (2)



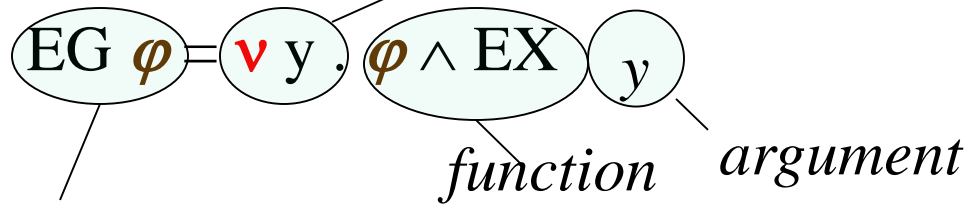
Based on the equivalence $EG \varphi = \varphi \wedge EX EG \varphi$

we observe that $EG \varphi$ is a fixpoint of the following function:

$$f y = \varphi \wedge EX y,$$

$$\text{i.e., } f(EG \varphi) = EG \varphi$$

In fact, $EG \varphi$ is the greatest fixpoint of function $\varphi \wedge EX$, which is written as:

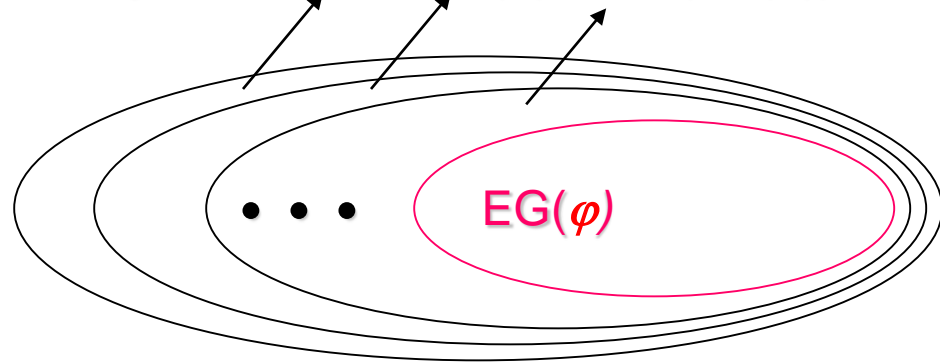


Fixpoint value

EG Fixpoint Computation



$EG(\varphi) \equiv$ “states that can avoid reaching $\neg\varphi$ ” $\equiv \varphi \cap EX(\varphi) \cap EX(EX(\varphi)) \cap \dots$



μ -Calculus



μ -Calculus is a temporal logic which consist of :

- Atomic propositions AP
- Boolean connectives: \neg , \wedge , \vee
- Pre-image operator: EX
- Least and greatest fixpoint operators: $\mu y. F y$ and $\nu y. F y$

Any CTL* formula can be expressed in μ -calculus

Symbolic Model Checking

[McMillan et al. LICS 90]



- Represent sets of states S and the transition relation R as Boolean logic formulas
- Fixpoint computation becomes formula manipulation, that includes:
 - pre-condition (EX) computation and existentially bound variable elimination;
 - conjunction (intersection), disjunction (union) and negation (set difference), and equivalence checks;
 - use an efficient data structure (BDD) for boolean logic formulas (BDD stands for Binary Decision Diagram).

Example: Mutual Exclusion Protocol



Two concurrently executing processes are trying to enter their critical section without violating mutual exclusion condition

```
Process 1:
while (true) {
    out:  a := true; turn := true;
    wait: await (b = false or turn = false);
    cs:   a := false;
}
||
Process 2:
while (true) {
    out:  b := true; turn := false;
    wait: await (a = false or turn);
    cs:   b := false;
}
```



Encoding State Space S

- Encode the state space using only boolean variables
- Two program counter variables: $pc1, pc2$ with domains $\{out, wait, cs\}$
 - We need two boolean variables per program counter to encode their 3 values:

$pc1_0, pc1_1, pc2_0, pc2_1$.

- Encoding:

$pc1 = out$	\longrightarrow	$\neg pc1_0 \wedge \neg pc1_1$
$pc1 = wait$	\longrightarrow	$\neg pc1_0 \wedge pc1_1$
$pc1 = cs$	\longrightarrow	$pc1_0 \wedge pc1_1$

- The other three variables $turn, a, b$ are already booleans.

Encoding State Space S



- Each state can be written as a tuple:

$(pc1_0, pc1_1, pc2_0, pc2_1, turn, a, b)$

– After encoding:

$(\overline{0}, \overline{0}, F, F, F)$ becomes $(\overline{F}, \overline{F}, \overline{F}, \overline{F}, F, F, F)$

$(\overline{0}, \overline{c}, F, T, F)$ becomes $(\overline{F}, \overline{F}, \overline{T}, \overline{T}, F, T, F)$

- We can use boolean logic formulas on the variables $pc1_0, pc1_1, pc2_0, pc2_1, turn, a, b$ to represent **sets** of states:

$\{(F, F, F, F, F, F, F)\} \equiv \neg pc1_0 \wedge \neg pc1_1 \wedge \neg pc2_0 \wedge \neg pc2_1 \wedge \neg turn \wedge \neg a \wedge \neg b$

$\{(F, F, T, T, F, F, T)\} \equiv \neg pc1_0 \wedge \neg pc1_1 \wedge pc2_0 \wedge pc2_1 \wedge \neg turn \wedge \neg a \wedge b$

$\{(F, F, F, F, F, F, F), (F, F, T, T, F, F, T)\} \equiv \neg pc1_0 \wedge \neg pc1_1 \wedge \neg pc2_0 \wedge \neg pc2_1 \wedge \neg turn \wedge \neg a \wedge \neg b \vee \neg pc1_0 \wedge \neg pc1_1 \wedge pc2_0 \wedge pc2_1 \wedge \neg turn \wedge \neg a \wedge b$
 $\equiv \neg pc1_0 \wedge \neg pc1_1 \wedge \neg turn \wedge \neg b \wedge (pc2_0 \wedge pc2_1 \leftrightarrow b)$

Encoding Initial States



- We can write the initial states as a boolean logic formula
 - recall that, initially: $pc1=0$ and $pc2=0$ but other variables may have any value in their domain

$$\begin{aligned} I &\equiv \{ (0, 0, F, F, F), (0, 0, F, F, T), (0, 0, F, T, F), \\ &\quad (0, 0, F, T, T), (0, 0, T, F, F), (0, 0, T, F, T), \\ &\quad (0, 0, T, T, F), (0, 0, T, T, T) \} \\ &\equiv \neg pc1_0 \wedge \neg pc1_1 \wedge \neg pc2_0 \wedge \neg pc2_1 \end{aligned}$$

meaning that

$pc1$ and $pc2$ are set to false and other variables may have arbitrary boolean values

Encoding the Transition Relation



- We can use boolean logic formulas and primed variables to encode the transition relation R .
- We will use two sets of variables:
 - Current state variables: $pc1_0, pc1_1, pc2_0, pc2_1, turn, a, b$
 - Next state variables: $pc1_0', pc1_1', pc2_0', pc2_1', turn', a', b'$
- For example, we can write a boolean logic formula for the statement of process 1:

`cs: a := false;`

which describes the effect of executing command symbolically:

$$\overline{pc1_0 \wedge pc1_1 \wedge \neg pc1_0' \wedge \neg pc1_1'} \wedge \overline{\neg a'} \wedge$$

$$(pc2_0' \leftrightarrow pc2_0) \wedge (pc2_1' \leftrightarrow pc2_1) \wedge (turn' \leftrightarrow turn) \wedge (b' \leftrightarrow b)$$

- Call this formula R_{1c}

Encoding the Transition Relation



- Similarly we can write a formula R_{ij} for each statement in the program

- Then the overall transition relation is

$$R \equiv R_{1o} \vee R_{1w} \vee R_{1c} \vee R_{2o} \vee R_{2w} \vee R_{2c}$$

But how to interpret temporal operators of φ on symbolic representation of M ??

Symbolic Pre-Condition Computation



- Recall the pre-image function

$$\text{EX} : 2^S \rightarrow 2^S$$

which is defined as:

$$\text{EX}(\varphi) = \{ s \mid (s, s') \in R \text{ and } s' \in \llbracket \varphi \rrbracket \}$$

- We can symbolically compute *pre* as follows

$$\text{EX}(\varphi) \equiv \exists V' (R \wedge \varphi[V' / V])$$

- V : values of boolean variables in the current-state
- V' : values of boolean variables in the next-state
- $\varphi[V' / V]$: rename variables in φ by replacing current-state variables with the corresponding next-state variables
- $\exists V' f$: means existentially quantifying variables V' in f

Renaming



- Assume that we have two variables x, y .
- Then, $V = \{x, y\}$ and $V' = \{x', y'\}$
- Renaming example:

Given $\varphi \equiv x \wedge y$:

$$\varphi[V' / V] \equiv x \wedge y [V' / V] \equiv x' \wedge y'$$



Existential Quantifier Elimination

- Given a boolean formula f and a single variable v

$$\exists v f \equiv f[\text{true}/v] \vee f[\text{false}/v]$$

i.e., to existentially quantify out a variable, first set it to true then set it to false and then take the disjunction of the two results.

- Example: $f \equiv \neg x \wedge y \wedge x' \wedge y'$

$$\exists y' f \equiv \exists x' (\exists y' (\neg x \wedge y \wedge x' \wedge y'))$$

$$\equiv \exists x' ((\neg x \wedge y \wedge x' \wedge y')[\text{true}/y'] \vee (\neg x \wedge y \wedge x' \wedge y')[\text{false}/y'])$$

$$\equiv \exists x' (\neg x \wedge y \wedge x' \wedge \text{true} \vee \neg x \wedge y \wedge x' \wedge \text{false})$$

$$\equiv \exists x' (\neg x \wedge y \wedge x')$$

$$\equiv (\neg x \wedge y \wedge x')[\text{true}/x'] \vee (\neg x \wedge y \wedge x')[\text{false}/x']$$

$$\equiv \neg x \wedge y \wedge \text{true} \vee \neg x \wedge y \wedge \text{false}$$

$$\equiv \neg x \wedge y$$

An Extremely Simple Example



Variables: x, y : boolean

Set of explicit states:

$$S = \{(F,F), (F,T), (T,F), (T,T)\}$$

$$S \equiv \text{true}$$

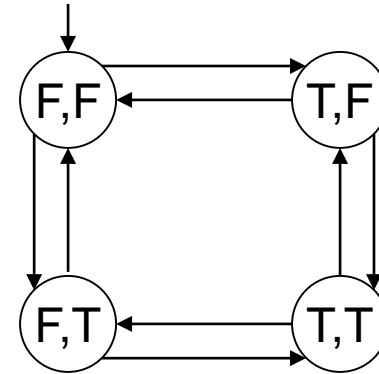
Initial condition:

$$I \equiv \neg x \wedge \neg y$$

Transition relation (after simplification):

$$R \equiv x' = \neg x \wedge y' = y \vee x' = x \wedge y' = \neg y$$

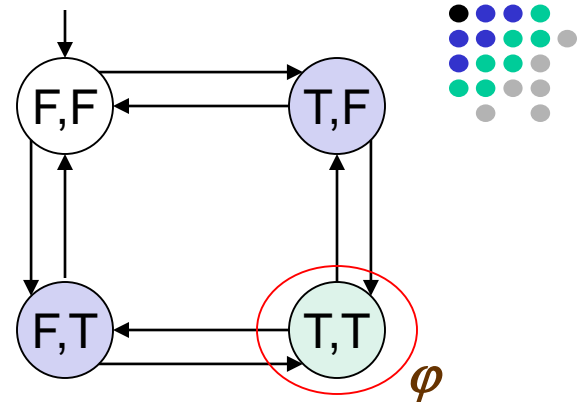
(= means \leftrightarrow)



An Extremely Simple Example

- Given $\varphi \equiv x \wedge y$ and R
- Compute $EX(\varphi)$

$$\begin{aligned}
 EX(\varphi) &\equiv \exists V' R \wedge \varphi[V' / V] && / \text{ by substit} \\
 &\equiv \exists V' (R \wedge x' \wedge y') && / \text{ by substit} \\
 &\equiv \exists V' ((x' = \neg x \wedge y' = y \vee x' = x \wedge y' = \neg y) \wedge x' \wedge y') && / \text{ by distr} \\
 &\equiv \exists V' (x' = \neg x \wedge y' = y) \wedge x' \wedge y' \vee (x' = x \wedge y' = \neg y) \wedge x' \wedge y' && / \text{ by } \leftrightarrow \\
 &\equiv \exists V' \neg x \wedge y \wedge x' \wedge y' \vee x \wedge \neg y \wedge x' \wedge y' && / \text{ by } \exists \text{-elimination} \\
 &\equiv \neg x \wedge y \vee x \wedge \neg y
 \end{aligned}$$



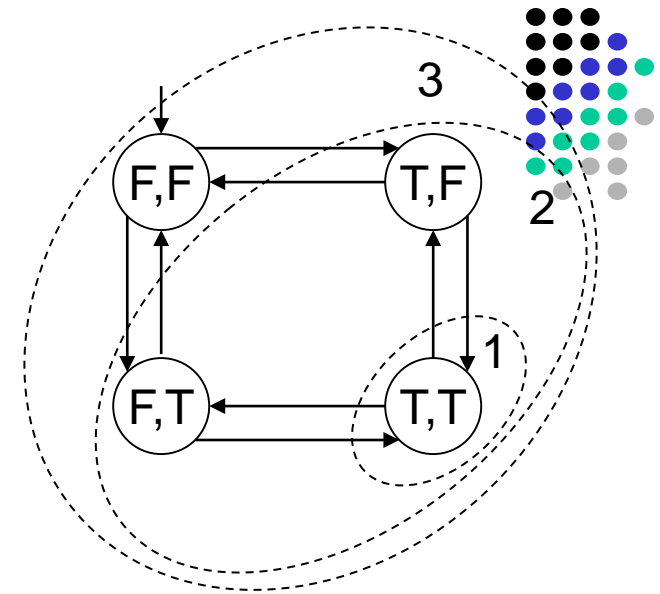
we get

$$EX(x \wedge y) \equiv \neg x \wedge y \vee x \wedge \neg y,$$

in terms of explicit states $EX(\{(T,T)\}) \equiv \{(F,T), (T,F)\}$

An Extremely Simple Example

Let's compute $EF(x \wedge y)$



The fixpoint sequence is

False, $x \wedge y$, $x \wedge y \vee EX(x \wedge y)$, $x \wedge y \vee EX(x \wedge y \vee EX(x \wedge y))$, ...

If we do the EX computations, we get:

$$\underbrace{\text{False}}_0, \quad \underbrace{x \wedge y}_1, \quad \underbrace{x \wedge y \vee \neg x \wedge y \vee x \wedge \neg y}_2, \quad \underbrace{\text{True}}_3$$

$EF(x \wedge y) \equiv \text{True}$

in terms of explicit states $EF(\{(T,T)\}) \equiv \{(F,F), (F,T), (T,F), (T,T)\}$

An Extremely Simple Example



- Based on our results, shown on example transition system $T = (S, I, R)$ we have

If

$I \subseteq \text{EF}(x \wedge y)$ (\subseteq corresponds to implication) hence:

$$T \models \text{EF}(x \wedge y)$$

(i.e., there exists a path from the initial state s.t. eventually x and y become true in the same state)

If

$I \not\subseteq \text{EX}(x \wedge y)$ hence:

$$T \not\models \text{EX}(x \wedge y)$$

(i.e., there does not exist a path from the initial state where in the next state x and y both become true)

An Extremely Simple Example



- Let's try one more property $AF(x \wedge y)$
- To check this property we first convert it to a formula which uses only temporal operators in our basis:

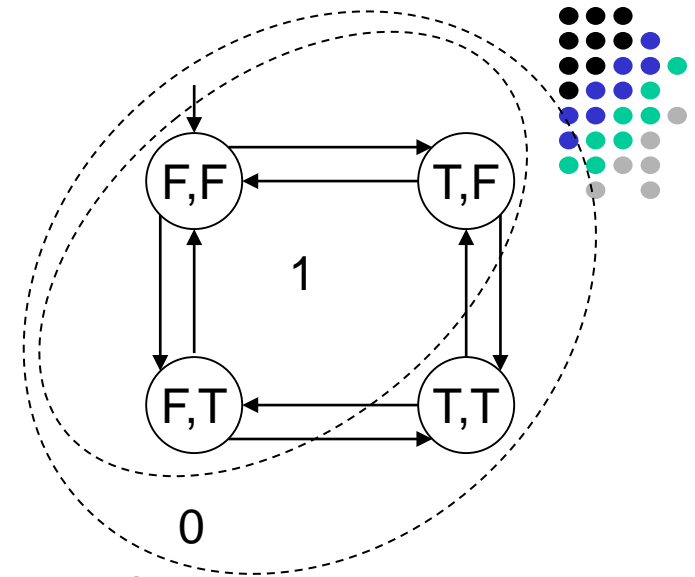
$$AF(x \wedge y) \equiv \neg EG(\neg(x \wedge y))$$

i.e.,

if we can find an initial state which satisfies $EG(\neg(x \wedge y))$, then we know that the transition system T does not satisfy the property $AF(x \wedge y)$

An Extremely Simple Example

Let's compute $EG(\neg(x \wedge y))$



The fixpoint sequence is:

true, $\neg x \vee \neg y$, $(\neg x \vee \neg y) \wedge EX(\neg x \vee \neg y)$, ...

If we do the EX computations, we get:

$\underbrace{\text{True}}_0$, $\underbrace{\neg x \vee \neg y}_1$, $\underbrace{\neg x \vee \neg y}_2$,

$$EG(\neg(x \wedge y)) \equiv \neg x \vee \neg y$$

Since $I \cap EG(\neg(x \wedge y)) \neq \emptyset$ we conclude that $T \not\models AF(x \wedge y)$

Symbolic CTL Model Checking Algorithm (in general)



- Translate the formula to a formula which uses the CTL basis
 - $EX \varphi, EG \varphi, \varphi EU \psi$
- Atomic formulas can be interpreted directly on the state representation
- For $EX \varphi$ compute the pre-image using existential variable elimination as we discussed
- For EG and EU compute the fixpoints iteratively

Symbolic Model Checking Algorithm



Check(f : CTL formula) : boolean logic formula
(here we use logic encoding of sets of states)

```
case:  $f \in AP$            return  $f$ ;  
case:  $f \equiv \neg \varphi$      return  $\neg$ Check( $\varphi$ );  
case:  $f \equiv \varphi \wedge \psi$  return Check( $\varphi$ )  $\wedge$  Check( $\psi$ );  
case:  $f \equiv \varphi \vee \psi$  return Check( $\varphi$ )  $\vee$  Check( $\psi$ );  
case:  $f \equiv EX \varphi$      return  $\exists V.R \wedge$  Check( $\varphi$ ) [ $V' / V$ ]
```

;

Symbolic Model Checking Algorithm



Check(f)

...

case: $f \equiv \mathbf{EG} \varphi$

$Y := \text{True};$

$P := \text{Check}(\varphi);$

$Y' := P \wedge \text{Check}(\mathbf{EX}(Y));$

while ($Y \neq Y'$)

{

$Y := Y';$

$Y' := P \wedge \text{Check}(\mathbf{EX}(Y));$

}

return Y ;

Symbolic Model Checking Algorithm



Check(f)

...

case: $f \equiv \varphi \text{ EU } \psi$

$Y := \text{False};$

$P := \text{Check}(\varphi);$

$Q := \text{Check}(\psi);$

$Y' := Q \vee [P \wedge \text{Check}(\text{EX}(Y))];$

while ($Y \neq Y'$)

{

$Y := Y';$

$Y' := Q \vee [P \wedge \text{Check}(\text{EX}(Y))];$

}

return Y ;

Binary Decision Diagrams (BDDs)



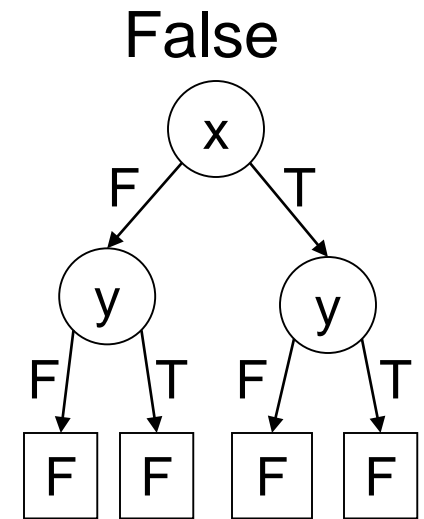
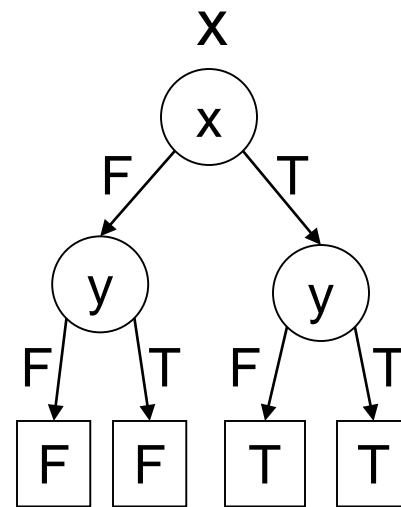
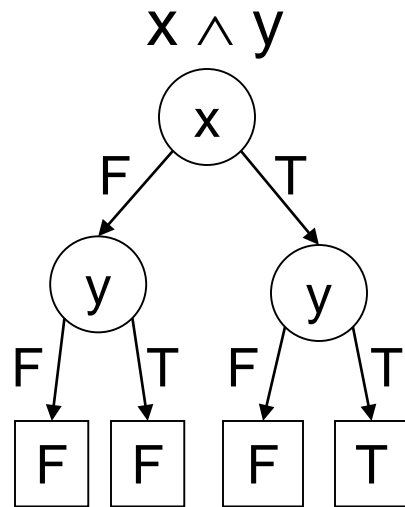
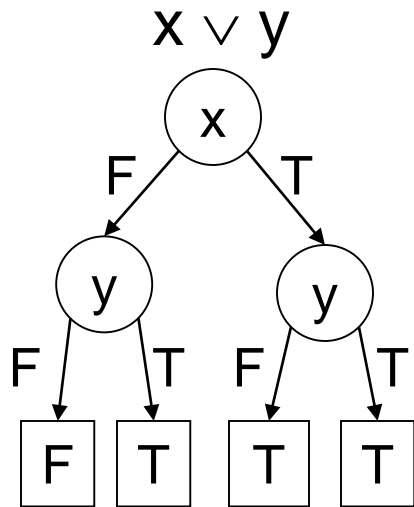
- Binary Decision Diagrams (BDDs)
 - An efficient data structure for boolean formula manipulation.
 - There are BDD packages available, e.g.
https://github.com/johnyf/tool_lists/blob/master/bdd.md
- BDD data structure can be used to implement the symbolic model checking algorithms discussed above.
- BDDs are *canonical representation* for boolean logic formulas, i.e.
 - given formulas F and G , they are $F \Leftrightarrow G$ if their BDD representations will be identical.

Binary Decision Trees (BDT)



Fix a variable order, in each level of the tree branch one value of the variable in that level.

- Examples of BDT-s for boolean formulas on two variables:
Variable order: x, y

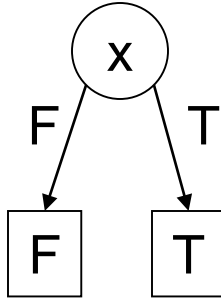
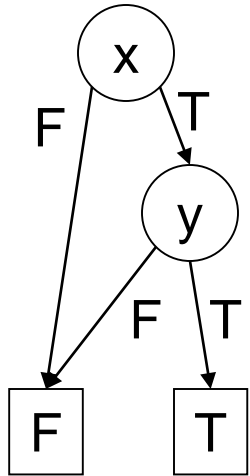
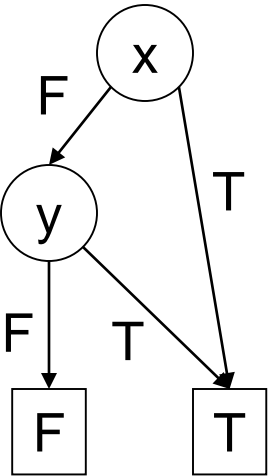
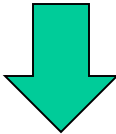
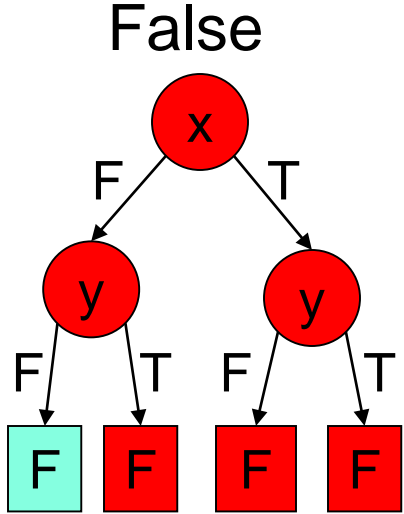
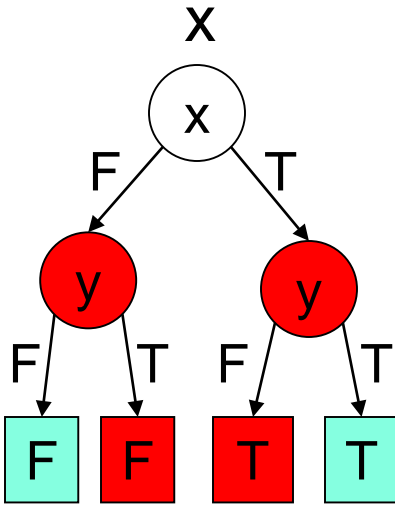
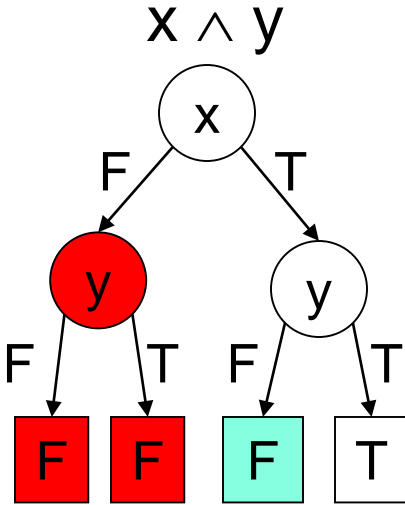
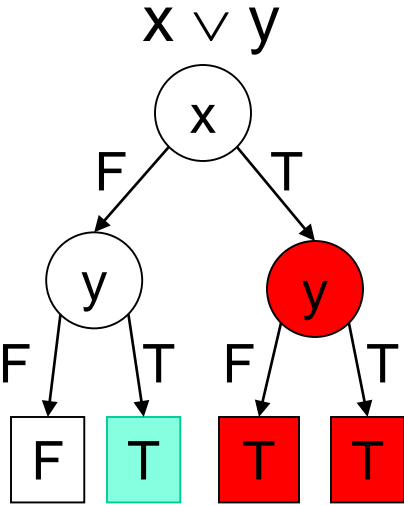


Transforming BDT to BDD



- Repeatedly apply the following transformations to a BDT:
 - Remove duplicate terminals & redraw connections to remaining terminals that have same name as deleted ones
 - Remove duplicate non-terminals & ...
 - Remove redundant tests
- These transformations transform the tree to a directed acyclic graph – binary decision diagram (BDD).

Binary Decision Trees vs. BDDs



- *redundant node*

Good News About BDDs



- Given BDDs for two boolean logic formulas F and G ,
 - the BDDs for $F \wedge G$ and $F \vee G$ are of size $|F| \times |G|$ (and can be computed in that time)
 - the BDD for $\neg F$ is of size $|F|$ (and can be computed in that time)
 - Equivalence $F \equiv? G$ can be checked in constant time
 - Satisfiability of F can be checked in constant time

Bad News About BDDs



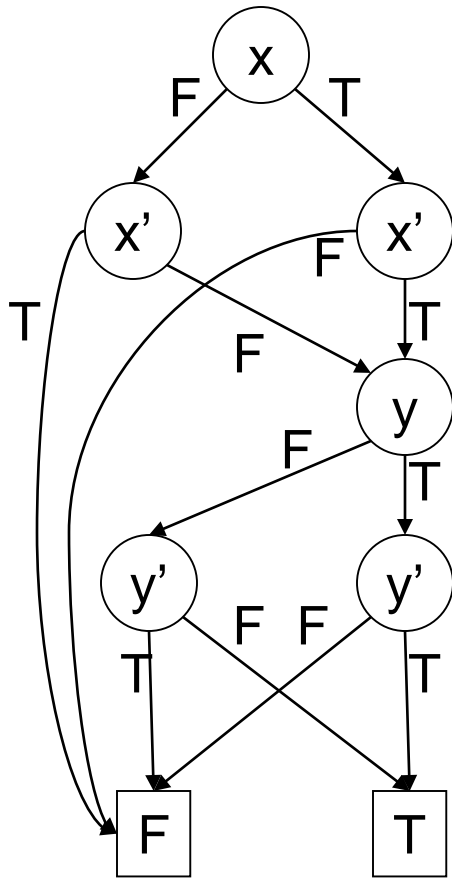
- The size of a BDD can be exponential in the number of boolean variables
- The sizes of the BDDs are very sensitive to the ordering of variables. Bad variable ordering can cause exponential increase in the size of the BDD
- There are functions which have BDDs that are exponential for any variable ordering (for example binary multiplication)
- Pre-condition computation requires existential variable elimination
 - Existential variable elimination can cause an exponential blow-up in the size of the BDD

BDDs are Sensitive to Variable Ordering



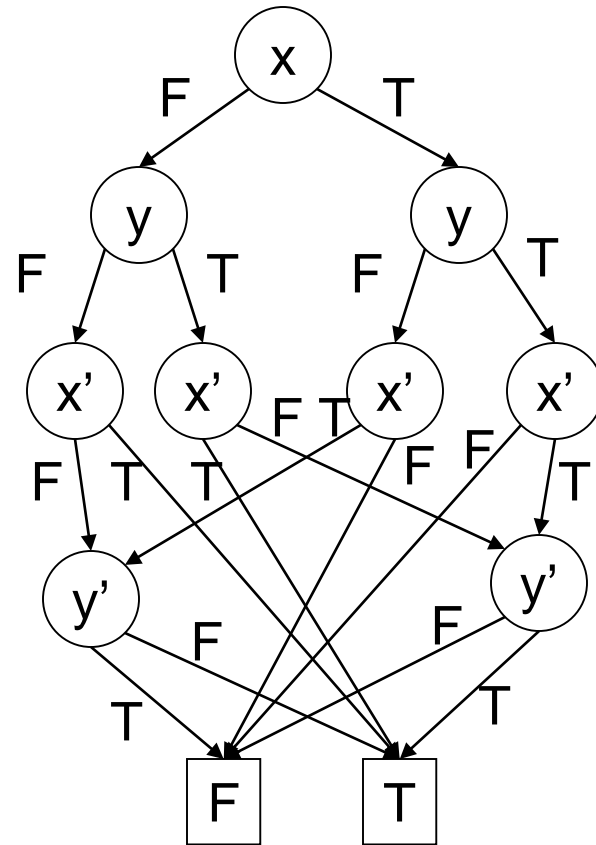
Identity relation for two variables: $(x' \leftrightarrow x) \wedge (y' \leftrightarrow y)$

Variable order: x, x', y, y'



For n variables, $3n+2$ nodes

Variable order: x, y, x', y'



For n variables, $3 \times 2^n - 1$ nodes

What About LTL and CTL* Model Checking?



- The complexity of the model checking problem for LTL and CTL* is:
 - $(|S|+|R|) \times 2^{O(|f|)}$
where $|f|$ is the number of logic connectives in f .
- Typically the size of the formula is much smaller than the size of the transition system
 - So the exponential complexity in the size of the formula is not very significant in practice
- LTL properties are intuitive and easy to write correctly
 - $X\varphi$ and $FX\varphi$ are equivalent in LTL
 - $AXAF\varphi$ and $AFAX\varphi$ are not equivalent in CTL