Model Checking

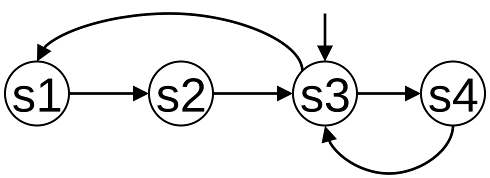
CTL model checking algorithms

Many slides from Tevfik Bultan

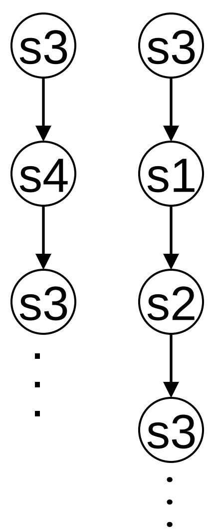
Recall: Linear Time vs. Branching Time

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

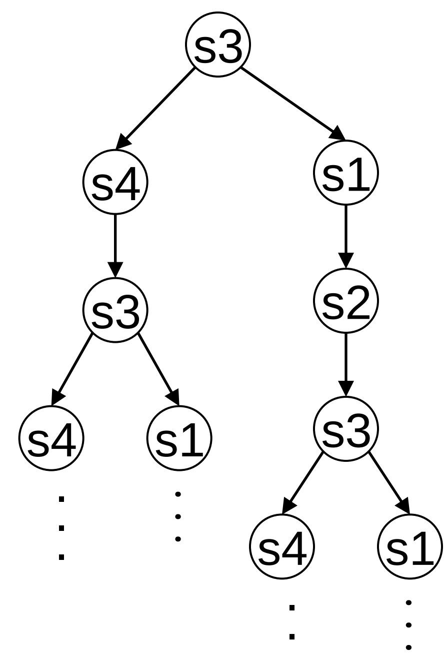
Transition System



Execution Paths



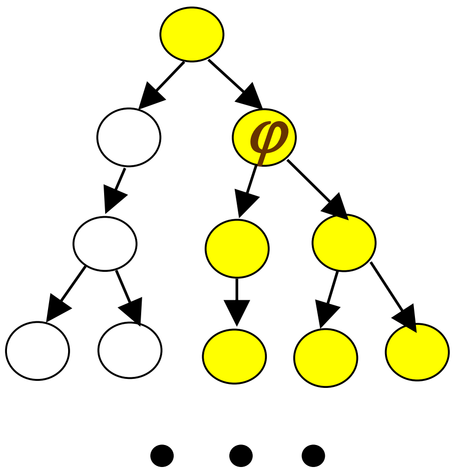
Computation Tree



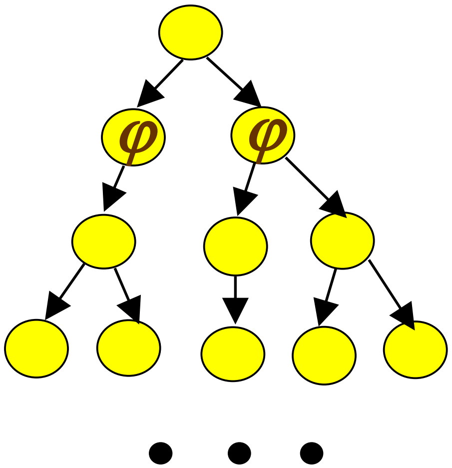
Recall: Computation Tree Logic (CTL)

- 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

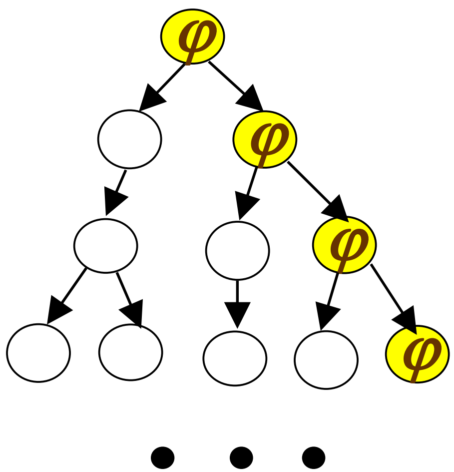
Examples



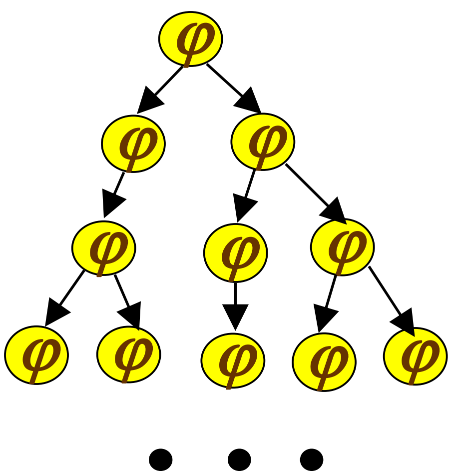
EX*φ* (exists next)



AX*φ* (all next)

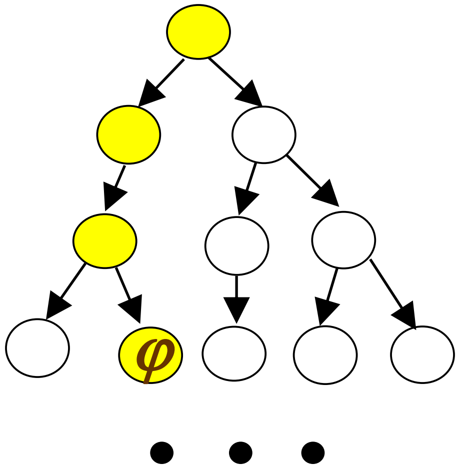


EG*φ* (exists global)

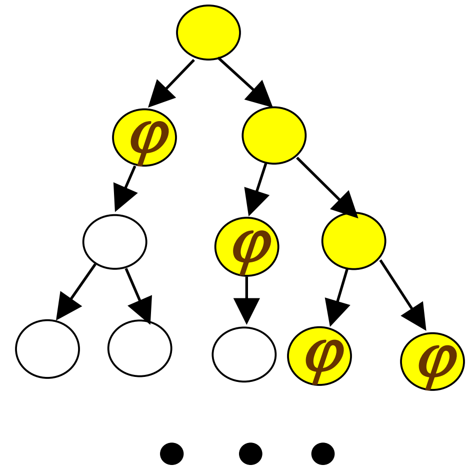


AG*φ* (all global)

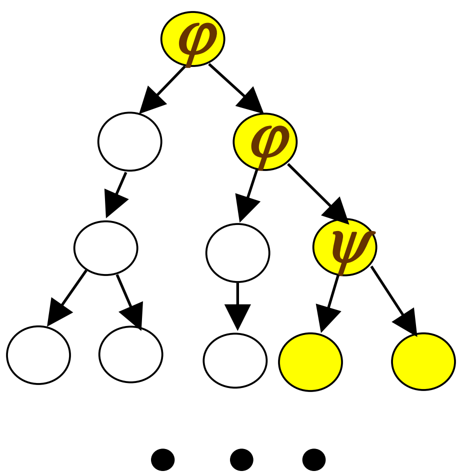
Examples (continued)



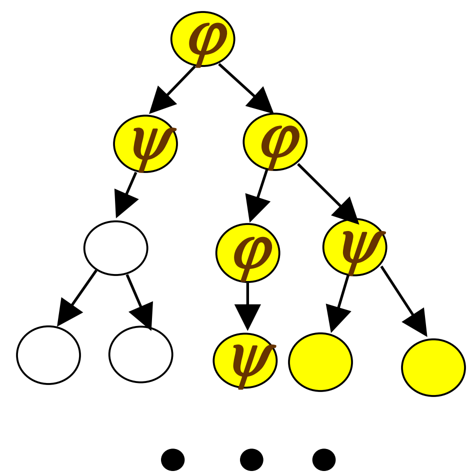
EF*φ* (exists future)



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 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

$\text{EX}\ \varphi, \quad \text{EG}\ \varphi, \quad \varphi\ \text{EU}\ \psi$

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

CTL Model Checking Algorithm (2)

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

CTL Model Checking Algorithms: intuition

- **EX** φ is easy to do 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 paths where <u>each</u> node is labeled with φ and label 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 and 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 $\subseteq$ truth set of φ
- **Falsification**:
  - Find if a state $\in$ (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 states traversal
  - needs a lot of memory
  - causes exponential explosion of required memory
    - the number of states |*S*| in the transition graph *T* is exponential in the number of variables and concurrent processes in the system modelled with LTS.

LTS – Labeled Transition System

Inroduction to symbolic state model checking

- How to deal with exponential explosion of the memory space for 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)

value *function* *argument*

$$AG\ \varphi = \varphi \wedge AX\ AG\ \varphi$$

$$EG\ \varphi = \varphi \wedge EX\ EG\ \varphi$$

$$AF\ \varphi = \varphi \vee AX\ AF\ \varphi$$

$$EF\ \varphi = \varphi \vee EX\ EF\ \varphi$$

$$\varphi\ AU\ \psi = \psi \vee (\varphi \wedge AX\ (\varphi\ AU\ \psi))$$

$$\varphi\ EU\ \psi = \psi \vee (\varphi \wedge EX\ (\varphi\ EU\ \psi))$$

Note:

We “unfold” the property by rewriting the CTL temporal operators using op-s themselves and EX and AX operators.

Functionals (mapping of an arbitrary set into a set)

- Given a transition system $T=(S, I, R)$, we will define functions from sets of states to sets of states
  - $f : 2^S \to 2^S$ 2^S – *set of subsets of S*

- For example, one such function is the EX operator (which computes the "pre-image" of a set of states given a relation R)
  - EX : $2^S \to 2^S$

  which can be defined as:

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

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.

Logic	Set
false	$\emptyset$
true	S
$\neg\ \varphi$	$S - \varphi$
$\varphi \wedge \psi$	$\varphi \cap \psi$
$\varphi \vee \psi$	$\varphi \cup \psi$

Temporal Properties as Fixpoints (1)

Based on the equivalence EF φ = $\varphi \vee$ EX EF φ

we observe that EF φ is a fixpoint of the following function:

f y = $\varphi \vee$ EX y,   where y = EF φ

i.e., f y = y

In fact, EF φ is the *least fixpoint* of f, which is written as:

EF φ = μ y . $\varphi \vee$ EX y

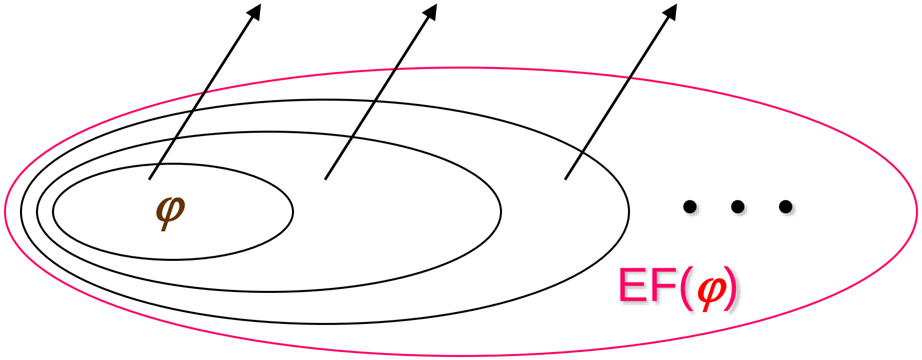
argument

function

Value of the argument that is fp

EF Fixpoint Computation

EF(φ) ≡ states from where φ is reachable ≡ $\varphi \cup$ EX(φ) $\cup$ EX(EX(φ)) $\cup$ …



Temporal Properties as Fixpoints (2)

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

we observe that EG φ is a fixpoint of the following function:

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

i.e., f (EG φ) = EG φ

In fact, EG φ is the *greatest fixpoint* of f, which is written as:

$$\text{EG } \varphi = \nu y . \varphi \wedge \text{EX } y$$

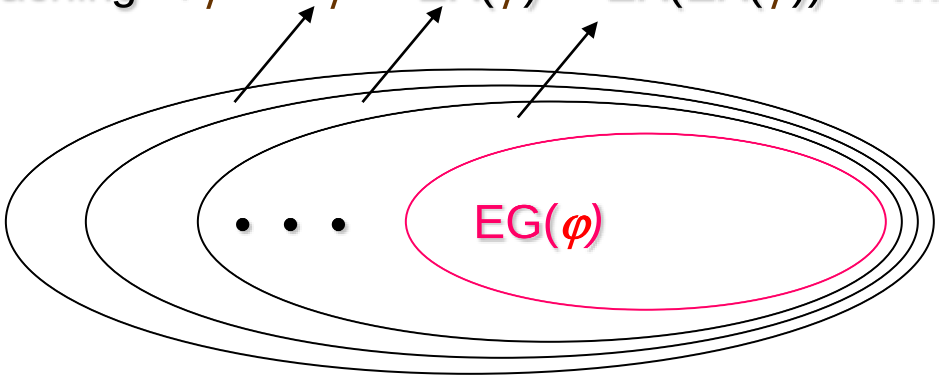
argument

function

Value of argument that is FP

EG Fixpoint Computation

EG(φ) $\equiv$ "states that can avoid reaching $\neg\, \varphi$" $\equiv \varphi \cap$ EX(φ) $\cap$ EX(EX(φ)) $\cap$ ...



μ-Calculus

μ-Calculus is a temporal logic which consist of :

- Atomic properties AP
- Boolean connectives: $\neg$, $\wedge$, $\vee$
- Pre-image operator: EX
- Least and greatest fixpoint operators: μ y. *F* y and ν 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, i.e.
  - pre-condition (EX) computation:

    including existentially bound variable elimination
  - conjunction (intersection), disjunction (union) and negation (set difference), and equivalence check

- Use an efficient data structure for boolean logic formulas
  - Binary Decision Diagrams (BDDs)

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:

    $\neg pc1_0 \wedge \neg pc1_1 \equiv$ `pc1 = out`

    $\neg pc1_0 \wedge pc1_1 \equiv$ `pc1 = wait`

    $pc1_0 \wedge pc1_1 \equiv$ `pc1 = cs`

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

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:

  `(o,o,F,F,F)` becomes (F,F,F,F,F,F,F)

  `(o,c,F,T,F)` becomes (F,F,T,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=o and pc2=o` but other variables may have any value in their domain

I ≡ `{(o,o,F,F,F), (o,o,F,F,T), (o,o,F,T,F), (o,o,F,T,T), (o,o,T,F,F), (o,o,T,F,T), (o,o,T,T,F), (o,o,T,T,T)}`

$\equiv \neg pc1_0 \wedge \neg pc1_1 \wedge \neg pc2_0 \wedge \neg pc2_1$

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;
```

as follows

$$pc1_0 \wedge pc1_1 \wedge \neg pc1_0' \wedge \neg pc1_1' \wedge \neg a' \wedge$$

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

- 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 interprete temporal operators of *p* on symbolic representation of M??

Symbolic Pre-condition Computation

- Recall the pre-image function

  $EX : 2^S \rightarrow 2^S$

  which is defined as:

  $EX(\varphi) = \{ s \mid (s,s') \in R \text{ and } s' \in [|\varphi|]\}$

- We can symbolically compute *pre* as follows

  $EX(\varphi) \equiv \exists V' \, (R \wedge \varphi\,[V' / V])$

  – 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

  – $\exists V' f$: existentially quantify out all the variables in V' from 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*

  $\exists v\, f \equiv f\,[true/v] \vee f\,[false/v]$

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

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

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

$\equiv \exists x'\, (\neg x \wedge y \wedge x' \wedge true \vee \neg x \wedge y \wedge x' \wedge false)$

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

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

$\equiv \neg x \wedge y \wedge true \vee \neg x \wedge y \wedge false$

$\equiv \neg x \wedge y$

An Extremely Simple Example

Variables: x, y: boolean

Set of states:

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

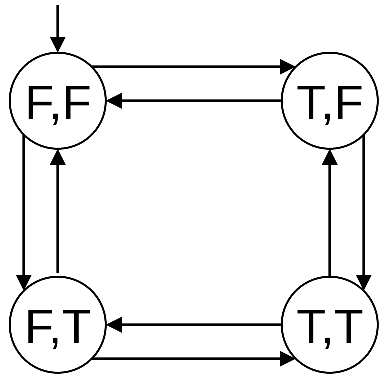
$S \equiv \mathit{true}$

Initial condition:

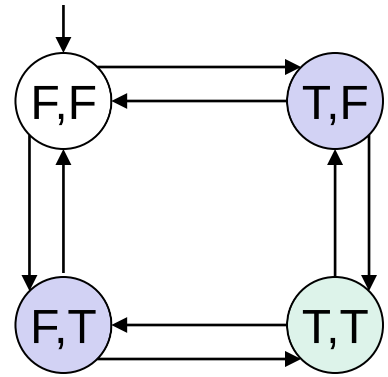
$I \equiv \neg x \wedge \neg y$

Transition relation (negates one variable at a time):

$R \equiv x' = \neg x \wedge y' = y \vee x' = x \wedge y' = \neg y$ (= means $\leftrightarrow$)



An Extremely Simple Example



Given $\varphi \equiv x \wedge y$, compute EX($\varphi$)

$\text{EX}(\varphi) \equiv \exists V'\, R \wedge \varphi[V' / V]$ | *by* substit

$\equiv \exists V'\, R \wedge x' \wedge y'$

$\equiv \exists V'\, (x'=\neg x \wedge y'=y \vee x'=x \wedge y'=\neg y) \wedge x' \wedge y'$ | *by distr*

$\equiv \exists V'\, (x'=\neg x \wedge y'=y) \wedge x' \wedge y' \vee (x'=x \wedge y'=\neg y) \wedge x' \wedge y'$ | *by* $\leftrightarrow$

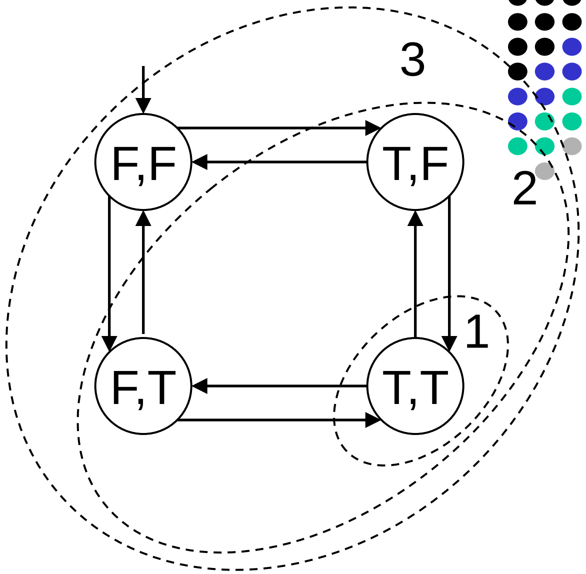
$\equiv \exists V'\, \neg x \wedge y \wedge x' \wedge y' \vee x \wedge \neg y \wedge x' \wedge y'$ | *by* $\exists$ -elimination

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

$\text{EX}(x \wedge y) \equiv \neg x \wedge y \vee x \wedge \neg y$

In other words EX({(T,T)}) ≡ {(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($x \wedge y$) $\equiv$ True

In other words EF({(T,T)}) $\equiv$ {(F,F),(F,T), (T,F),(T,T)}

An Extremely Simple Example

- Based on our results, for extremely simple transition system $T = (S, I, R)$ we have

If

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

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

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

If

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

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

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

An Extremely Simple Example

- Let’s try one more property $AF(x \wedge y)$

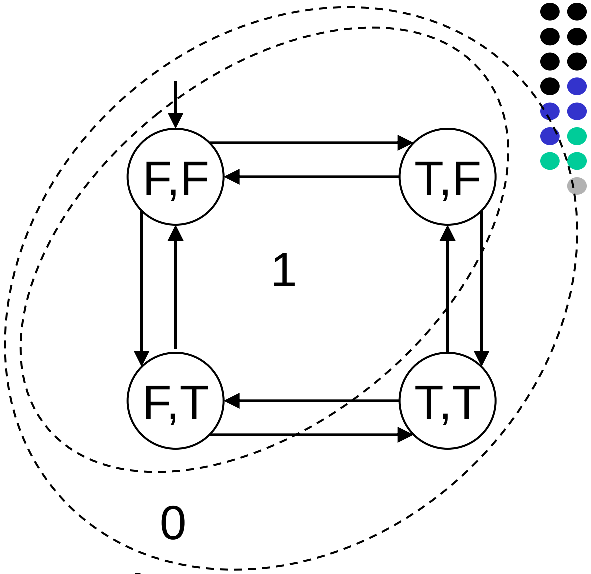
- To check this property we first convert it to a formula which uses only the temporal operators in our basis:

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

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 basis
  - EX φ, EG φ, φ EU ψ
- Atomic formulas can be interpreted directly on the state representation
- For EX φ 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(φ);
case: $f \equiv \varphi \wedge \psi$	return Check(φ) $\wedge$ Check(ψ);
case: $f \equiv \varphi \vee \psi$	return Check(φ) $\vee$ Check(ψ);
case: $f \equiv$ EX φ	return $\exists V' R \wedge$ Check(φ)[V'/V];

Symbolic Model Checking Algorithm

Check(*f*)

```
...
case: f ≡ EG φ
    Y := True;
    P := Check(φ);
    Y' := P ∧ Check(EX(Y));
    while (Y ≠ Y')
    {
          Y  := Y';
          Y' := P ∧ Check(EX(Y));
    }
    return Y;
```

Symbolic Model Checking Algorithm

Check(*f*)

```
…
case: f ≡ φ EUψ
    Y := False;
    P := Check(φ);
    Q := Check(ψ);
    Y' := Q ∨ [P ∧ Check(EX(Y))];
    while (Y ≠ Y')
    {
        Y := Y';
        Y':= Q ∨ [P ∧ Check(EX(Y))];
    }
    return Y;
```

Binary Decision Diagrams (BDDs)

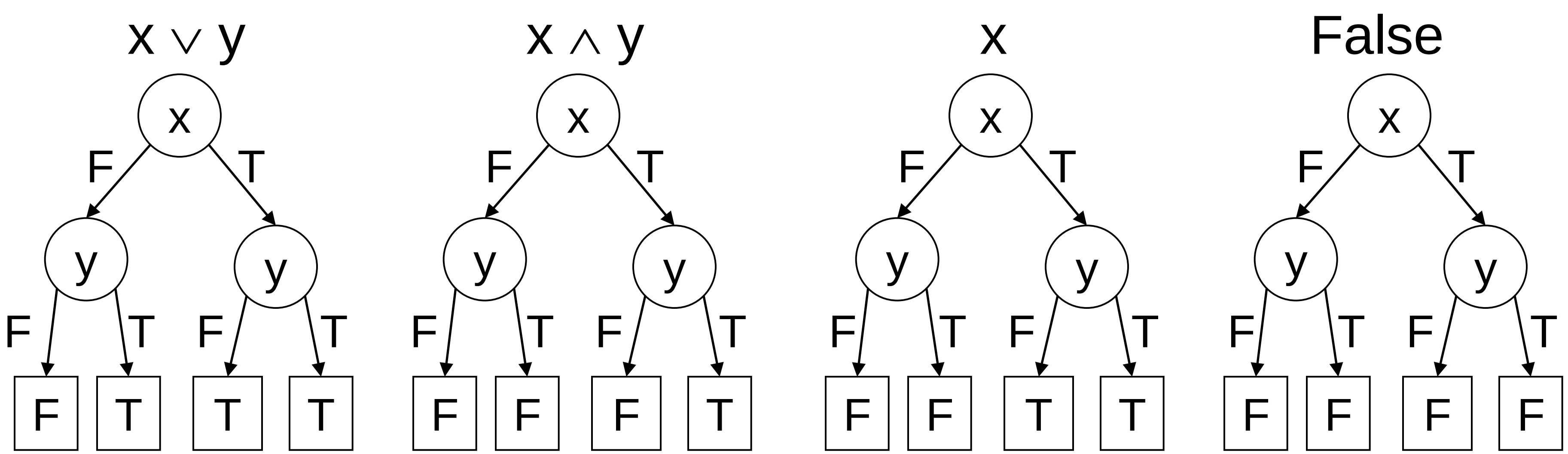
- Binary Decision Diagrams (BDDs)
  - An efficient data structure for boolean formula manipulation.
  - There are BDD packages available, e.g. CUDD from Colorado University http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html
- BDD data structure can be used to implement the symbolic model checking algorithms 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



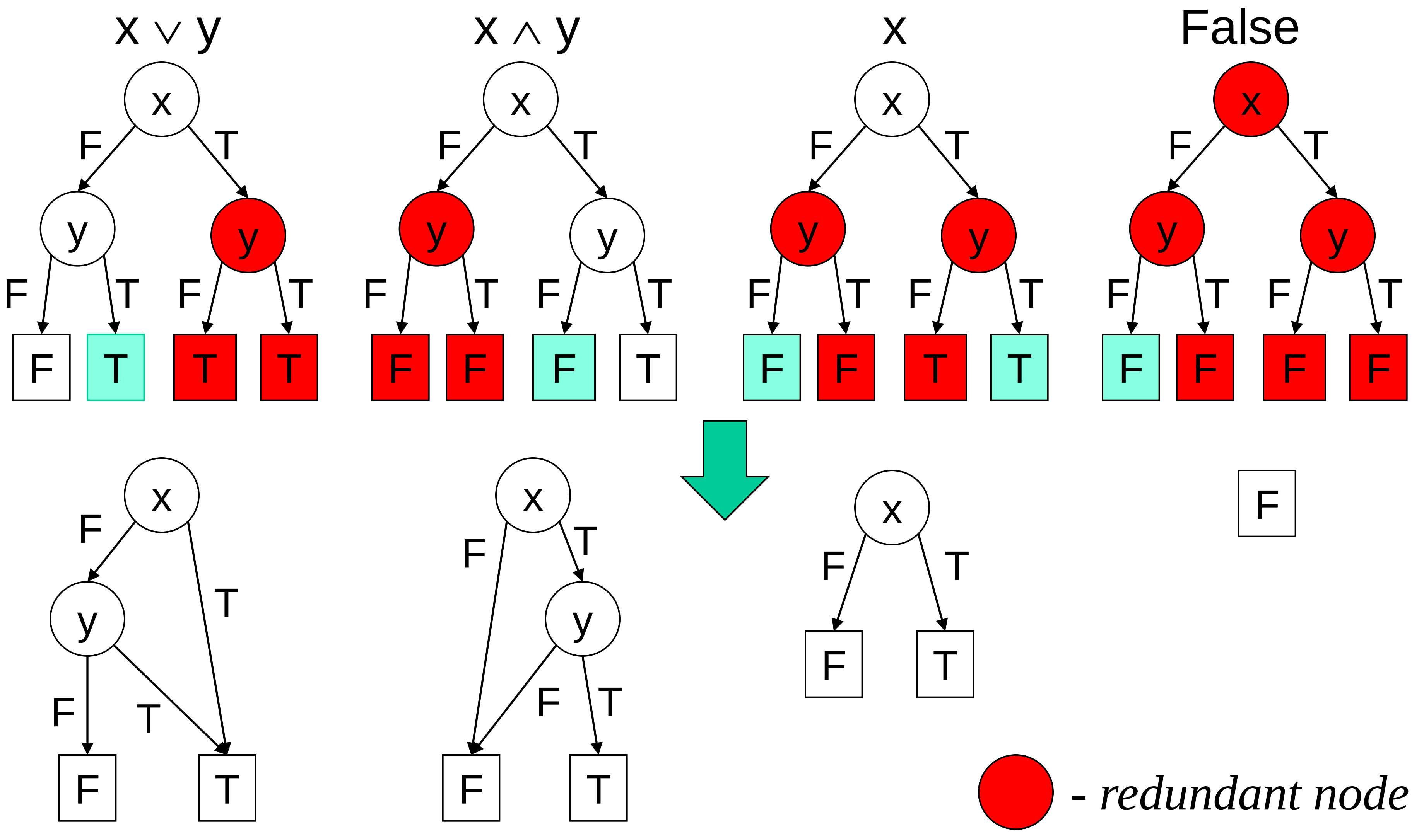
Transforming BDT to BDD

- 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



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
    - But, this does not mean that one can solve SAT in constant time (it is NP-complete problem).

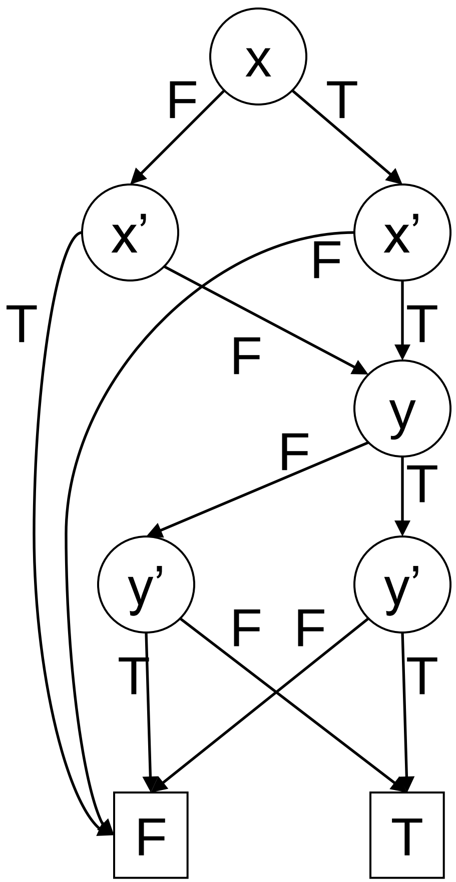
Bad News About BDDs

- The size of a BDD can be exponential in the number of boolean variables
- The sizes of the BDDs are very <u>sensitive to the ordering of variables</u>. 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

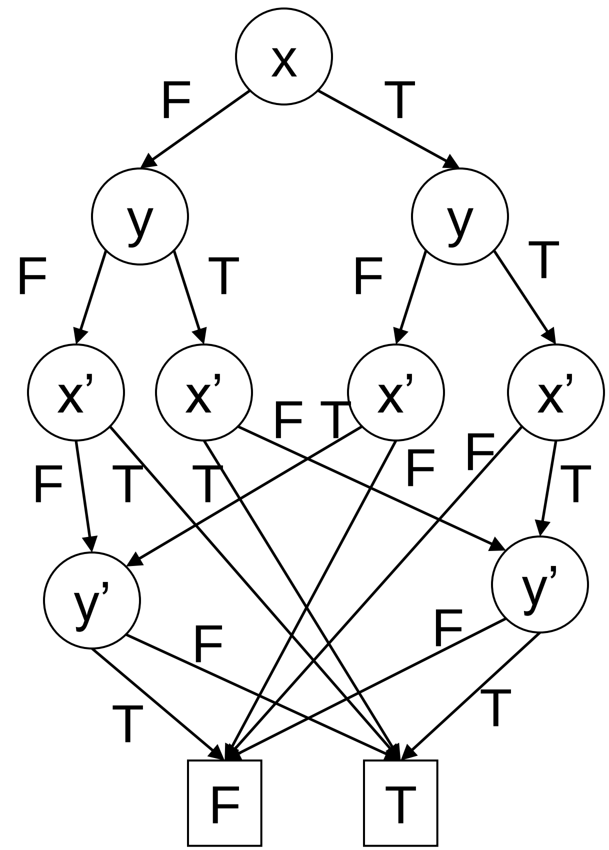
Identity relation for two variables: $(x' \leftrightarrow x) \wedge (y' \leftrightarrow y)$

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
  - XF φ and FX φ are equivalent in LTL
  - AXAF φ and AFAX φ are not equivalent in CTL