CS 267: Automated Verification

Lectures 5: Symbolic model checking

Instructor: Tevfik Bultan

Symbolic Model Checking

[McMillan et al. LICS 90]

- Basic idea: ***Represent sets of states and the transition relation as Boolean logic formulas***

- Fixpoint computation becomes formula manipulation
  - pre-condition (EX) computation: Existential 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 a critical section without violating mutual exclusion

```
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;
}
```

State Space

- Encode the state space using only boolean variables

- Two program counters: `pc1, pc2` with domains `{out, wait, cs}`
  - Use two boolean variable per program counter:

  $pc1_0$, $pc1_1$, $pc2_0$, $pc2_1$
  - 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 booleans: turn, a , b

State Space

- Each state can be written as a 7-tuple of boolean values:
  ($pc1_0$,$pc1_1$,$pc2_0$,$pc2_1$,turn,a,b)
  - For example:
  `(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)

- There are 2^7=128 possible values for our new representation
  - Our original state space was 3*3*2*2*2=72
  - Note that the following tuples are not in our state space due to our encoding of the program counters:
    (T,F,*,*,*,*,*), (*,*,T,F,*,*,*)
    I used "*" to mean any value

Representing sets of states

- 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:

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

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

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

$\equiv \neg pc1_0 \wedge \neg\, pc1_1 \wedge \neg\, turn \wedge \neg a \wedge (pc2_0 \wedge pc2_1 \leftrightarrow b)$

Initial States

- We can write the initial states as a boolean logic formula
  - recall that, initially: `pc1=o and pc2=o`

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)}`

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

Transition Relation

- We can use boolean logic formulas to encode the transition relation. 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 following 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}

Transition Relation

- We can write a formula for each statement in the program
- Then the overall transition relation is

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

```
    Process 1:
    while (true) {
R1o ⟷ out:  a := true; turn := true;
R1w ⟷ wait: await (b = false or turn = false);
R1c ⟷ cs:   a := false;
    }
    ||
    Process 2:
    while (true) {
R2o ⟷ out:  b := true; turn := false;
R2w ⟷ wait: await (a = false or turn);
R2c ⟷ cs:   b := false;
    }
```

Symbolic Pre-condition Computation

- Remember the function

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

  which is defined as:

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

- We can symbolically compute pre as follows

  $EX(p) \equiv \exists V' \; R \wedge p[V' / V]$

  – V : current-state boolean variables

  – V' : next-state boolean variables

  – p[V' / V] : rename variables in p by replacing current-state variables (V) with the corresponding next-state variables (V')

  – $\exists V' f$: 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 $p \equiv x \wedge y$:

  $p[V' / V] \equiv x \wedge y\,[V' / V] \equiv x' \wedge y'$

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

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

$\equiv \exists x' \; ((\neg x \wedge y \wedge x' \wedge y')[T/y'] \vee (\neg x \wedge y \wedge x' \wedge y')[F/y'])$

$\equiv \exists x' \; (\neg x \wedge y \wedge x' \wedge T \vee \neg x \wedge y \wedge x' \wedge F)$

$\equiv \exists x' \; \neg x \wedge y \wedge x'$

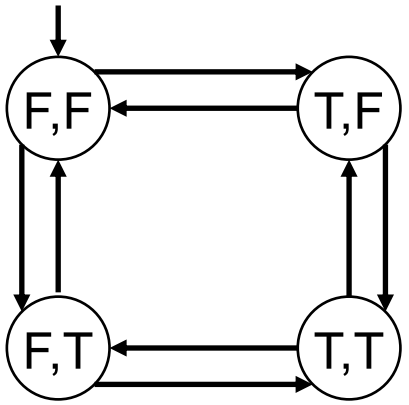
$\equiv (\neg x \wedge y \wedge x')[T/x'] \vee (\neg x \wedge y \wedge x')[F/x'])$

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

$\equiv \neg x \wedge y$

An Extremely Simple Example

Variables: x, y: boolean

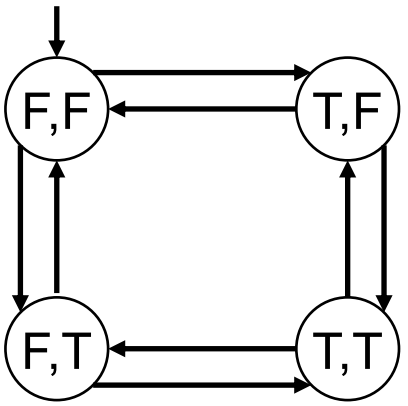


Set of states:
S = {(F,F), (F,T), (T,F), (T,T)}
S ≡ True

Initial condition:
I ≡ ¬ x ∧ ¬ y

Transition relation (negates one variable at a time):
R ≡ x' =¬x ∧ y' =y ∨ x' =x ∧ y' =¬y (= means ↔)

An Extremely Simple Example



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

$EX(p) \equiv \exists V' \; R \wedge p[V' / V]$

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

$\equiv \exists V' \; (x' = \neg x \wedge y' = y \vee x' = x \wedge y' = \neg y) \wedge x' \wedge y'$

$\equiv \exists V' \; (x' = \neg x \wedge y' = y) \wedge x' \wedge y' \vee (x' = x \wedge y' = \neg y) \wedge x' \wedge y'$

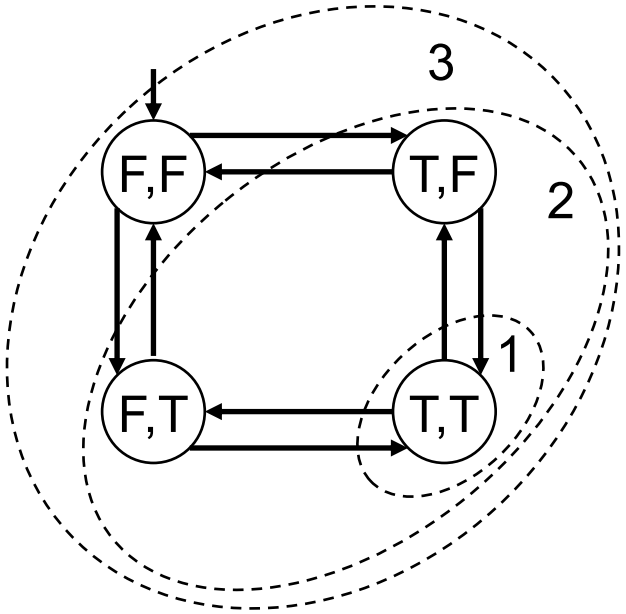
$\equiv \exists V' \; \neg x \wedge y \wedge x' \wedge y' \vee x \wedge \neg y \wedge x' \wedge y'$

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

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

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

An Extremely Simple Example



Let's compute compute EF(x ∧ y)

The fixpoint sequence is

False, x∧y , x∧y ∨ EX(x∧y) , x∧y ∨ EX (x∧y ∨ EX(x∧y)) , ...

If we do the EX computations, we get:

$\underbrace{\text{False}}_{0}$, $\underbrace{x \wedge y}_{1}$, $\underbrace{x \wedge y \vee \neg x \wedge y \vee x \wedge \neg y}_{2}$, $\underbrace{\text{True}}_{3}$

EF(x ∧ y) ≡ True

In other words EF({(T,T)}) ≡ {(F,F),(F,T), (T,F),(T,T)}

An Extremely Simple Example

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

$I \subseteq EF(x \wedge y)$ hence:

$T \models EF(x \wedge y)$

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

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

$T \not\models EX(x \wedge y)$

(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

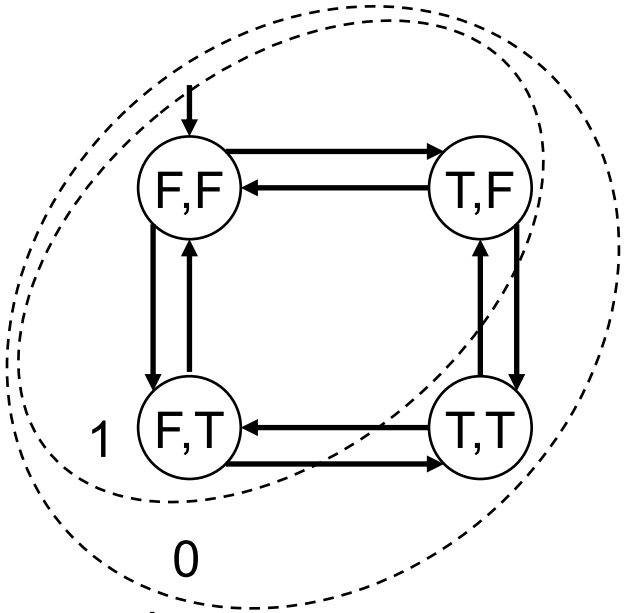
- Let's try one more property $AF(x \wedge y)$

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

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

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

Symbolic CTL Model Checking Algorithm

- Translate the formula to a formula which uses the basis
  - EX p, EG p, p EU q
- Atomic formulas can be interpreted directly on the state representation
- For EX p compute the precondition 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

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

Symbolic Model Checking Algorithm

```
Check(f)
    …
    case: f ≡ EG p
        Y := True;
        P := Check(p);
        Ynew := P ∧ Check(EX(Y));
        while (Y ≠ Ynew) {
                Y := Ynew;
                Ynew := P ∧ Check(EX(Y));
        }
        return Y;
```

Symbolic Model Checking Algorithm

```
Check(f)
    …
    case: f ≡ p EU q
          Y := False;
          P := Check(p);
          Q := Check(q);
          Ynew := Q ∨ P ∧ Check(EX(Y));
          while (Y ≠ Ynew) {
                Y := Ynew;
                Ynew := Q ∨ P ∧ Check(EX(Y));
          }
          return Y;
```