Binary Decision Diagrams –II

CS 4271
Abhik Roychoudhury
National University of Singapore

State Explosion Problem

- Our CTL model checking algorithm is linear in size of state space and formula
  - Suffers from State Space Explosion problem since the state space is exponential in the number of system components.
- We need more space efficient representation of sets of state and transition relation
  - Reduced Ordered Binary Decision Diagrams (ROBDD) is a data structure to achieve this.

ROBDD – the context

- To discuss ROBDD, let us first discuss BDD.
- BDD is a data structure for compactly representing boolean functions.
  - Boolean functions have a fundamental role in computing
  - Suitable for directly modeling combinational circuits
  - Can capture the set of states and transition relation of finite state machines corresponding to sequential circuits.
    - **Leads to more space efficient model checking**

In the previous lecture

- Binary decision Diagrams
  - Basic Definitions
  - Reduction of BDDs
  - The importance of variable orders
  - Reduced Ordered BDD as a compact normal form representation of a boolean function.

Today's lecture

- Getting BDDs ready for use in MC
  - Using BDDs for representing sets of states.
  - Using BDDs for representing sets of transitions.
  - Performing binary boolean operations on BDDs
    - Can be used to achieve set operations on sets of states represented by BDDs

Representing sets of states

- Kripke Structure M = (S, S0, $\rightarrow$, L)
  - Represent a set of states S' $\subseteq$ S as a BDD
    - We can then represent the sets of intermediate sets of states St_{φ} constructed during the model checking algo., as a BDD
  - So, represent S' $\subseteq$ S as a boolean function
    - What will be the input variables for this boolean function ?

State Variables

- Assume all state variables are boolean.
  - If not, they can be described as collections of boolean vars.
- Suppose there are n boolean state vars.
  - Hence 2^n possible valuations of these state vars.
  - Definition of Kripke structure allows more than one state to have the same valuation for all variables.
  - If this is the case, we can ensure that each state has a unique valuation of state vars, by
    - Merging states with identical variable valuations, or
    - Simply increasing the set of state variables in order to distinguish the states.

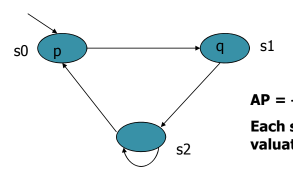
Defining the boolean function

- If we can choose n boolean state variables, such that each state in the Kripke Structure M = (S, S0, $\rightarrow$, L) is a unique valuation of these variables
  - Consider a boolean function with these n inputs.
  - Each state s $\in$ S can be uniquely identified with exactly one row of the truth table for such a function.
  - A subset of states S' $\subseteq$ S is thus uniquely identified with a subset of rows of the truth table.
  - To make the boolean function represent S', we make it output 1 in exactly these rows, output 0 in all other rows.
- OK, that's great, but what are boolean inputs ?

Choice of boolean inputs

- Kripke Structure M = (S,S0,$\rightarrow$,L)
  - L maps S to 2^{AP}, where AP is the set of atomic propositions.
  - Choose AP as the set of boolean state variables.
    - Each state must have a unique valuation of AP, for the AP we choose.
  - A set of states S' $\subseteq$ S is represented as a boolean function $F_{S'}(x_1,\ldots,x_n)$ where
    - AP = $\{x_1,\ldots,x_n\}$ is the set of atomic propositions used in defining the labeling function.

Example



AP = {p,q}

Each state produces a unique valuation of propositions in AP.

Represent the set {s1,s2} as a boolean function F(p,q)

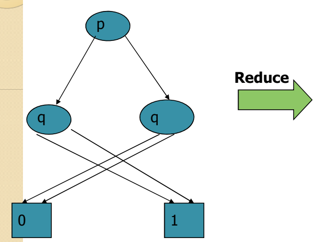
Example

- s1 stands for the valuation
  - p = 0, q=1
- {s1} is represented by f1(p,q) = $\neg$p $\wedge$ q
- s2 stands for the valuation
  - p = 0, q= 0
- {s2} is represented by f1(p,q) = $\neg$p $\wedge$ $\neg$ q
- {s1,s2} is represented by F(p,q) = f1(p,q) $\vee$ f2(p,q)

Example

- To construct a ROBDD for representing the set {s1,s2}
  - We need an ordering of the input variables of F(p,q)
  - Choose the order p < q
  - Construct the ROBDD for
  - F(p,q) = f1(p,q) $\vee$ f2(p,q)
  - = ($\neg$p $\wedge$ q) $\vee$ ($\neg$p $\wedge$ $\neg$ q)

Example



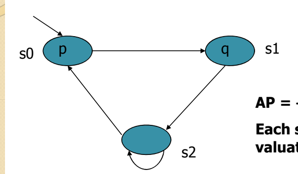
Representing Transition Relation

- M = (S, S0, $\rightarrow$, L)
  - Transition relation $\rightarrow \subseteq S \times S$
- To represent the transition relation as boolean fn
  - How to represent any arbitrary subset of $S \times S$ as a boolean function ?
  - We represented any subset of S as a boolean function $F(x_1,\ldots,x_n)$ where AP = $\{x_1,\ldots,x_n\}$
  - We can represent any subset of $S \times S$ as a boolean function $F(x_1,\ldots,x_n, x'_1,\ldots,x'_n)$ where AP = $\{x_1,\ldots,x_n\}$
  - x'_i denotes the value of atomic proposition x_i after transition

Representing Transition Relation

- Represent transition relation $\rightarrow$ as a boolean function $F(x_1,\ldots,x_n, x'_1,\ldots,x'_n)$ where AP = $\{x_1,\ldots,x_n\}$
- This is function's output is true if and only if
  - Values of $x_1,\ldots,x_n$ correspond to a state $s \in S$
  - Values of $x'_1,\ldots,x'_n$ correspond to a state $s' \in S$
  - $(s, s') \in \rightarrow$
- The 2n-ary function F above can be represented as a ROBDD if we fix an order among $x_1,\ldots,x_n, x'_1,\ldots,x'_n$
- *Note: Keep in mind that M = (S, S0,$\rightarrow$,L) is the Kripke Structure*

Example



AP = {p,q}

Each state produces a unique valuation of propositions in AP.

Represent the transition relation above as a boolean function F(p, q, p', q')

Example

Transitions	**p**	**q**	**p'**	**q'**
(s0, s1)	1	0	0	1
(s1, s2)	0	1	0	0
(s2, s0)	0	0	1	0
(s2, s2)	0	0	0	0

All input valuations above (marked in red) should produce 1 as output.

All other input valuations should produce 0 as output.

In the remaining part today …

- Getting BDDs ready for use in MC
  - Using BDDs for representing sets of states.
  - Using BDDs for representing sets of transitions.
  - Performing binary boolean operations on BDDs
    - Can be used to achieve set operations on sets of states represented by BDDs

Manipulating ROBDDs

- So far we have seen how to represent
  - Sets of states/transitions as boolean function
  - And hence as BDD
- While performing model checking, sets of states represented as BDD
- Set operations now need to be translated as logical operations on BDDs

Manipulating ROBDDs

- $St_{EGf} = St_{EGf} - Temp$
- $St_{EFf} = St_{EFf} \cup Temp$
- …

- $bdd_{EGf} = bdd_{EGf} \wedge !Temp$
- $bdd_{EFf} = bdd_{EFf} \vee Temp$
- …

If sets are represented as boolean functions, set operations as logical operations (in propositional logic).

ROBDDs are a reduced representation of boolean functions.
How to apply boolean functions on ROBDDs?

Manipulating ROBDDs

- $St_{AGf} = St_f$
- do{
- Temp={ s | s $\in St_{AGf}$
- $\exists t\ s \to t \wedge t \notin St_{AGf}$
- }
- $St_{AGf} = St_{AGf}$ - Temp;
- } until no change to St_{AGf}
- Return St_{AGf}

- $bdd_{AGf} = bdd_f$
- do{
- Tmp = $bdd_{AGf} \wedge$ …
- ….

ROBDDs are a reduced representation of boolean functions.
How to apply boolean functions on ROBDDs?

Manipulation of ROBDDs

- We now give an algorithm Apply
  - $2^{2^2} = 16$ logical operations on boolean func.
- Do not see model checking as graph traversal
  - Transitions systems represented as BDDs
  - Traversal achieved by logical operations
  - "Symbolic" Model Checking (Next class)
- Let us look at the Apply algorithm.

Applying boolean operations

- ROBDDs are a compact representation of boolean functions
- How to efficiently apply boolean operations to two ROBDDs, e.g.
  - $F1(x,y,z) = x \wedge z$
  - $F2(x,y,z) = x \vee y$
  - Define $F(x,y,z) = F1(x,y,z) \oplus F2(x,y,z)$
  - How to construct the ROBDD for F from the ROBDD for F1 and F2 (given a variable order, of course !)

Binary Ops on Boolean Fn

F1	F2	F
0	0	?
0	1	?
1	0	?
1	1	?

2^4 = 16 such logical operations exist.

Let * denote any such logical operation.

Co-factoring

- $F(x_1,\ldots,x_n)|_{x_1}$ is the boolean function F with x_1 replaced by True
- Similarly define $F(x_1,\ldots,x_n)|_{\neg x_1}$
  - Shorthand written as $F|_{x_1}$ and $F|_{\neg x_1}$ when the set of input variables of F are understood from the context.
- **Shannon's expansion defines F in terms of its cofactors**
  - $F(x_1,\ldots,x_n) = (x_1 \wedge F|_{x_1}) \vee (\neg x_1 \wedge F|_{\neg x_1})$
  - $= (x_2 \wedge F|_{x_2}) \vee (\neg x_2 \wedge F|_{\neg x_2})$
  - $= \ldots$
  - $= (x_n \wedge F|_{x_n}) \vee (\neg x_n \wedge F|_{\neg x_n})$

Co-factoring is distributive

- … for any binary boolean operation
- $F(x_1,\ldots,x_n) = F1(x_1,\ldots,x_n) * F2(x_1,\ldots,x_n)$
  - $F|_{x_1} = (F1 * F2)|_{x_1} = F1|_{x_1} * F2|_{x_1}$
  - $F|_{\neg x_1} = (F1 * F2)|_{\neg x_1} = F1|_{\neg x_1} * F2|_{\neg x_1}$
  - Similarly for $F|_{x_2}$, $F|_{\neg x_2}, \ldots, F|_{x_n}, F|_{\neg x_n}$
- Can be shown by expanding as follows
  - $F = x_1 \wedge F|_{x_1} \vee \neg x_1 \wedge F|_{\neg x_1}$
  - $= F1 * F2 =$
  - $= (x_1 \wedge F1|_{x_1} \vee \neg x_1 \wedge F1|_{\neg x_1}) * (x_1 \wedge F2|_{x_1} \vee \neg x_1 \wedge F2|_{\neg x_1})$
  - $= x_1 \wedge (F1|_{x_1} * F2|_{x_1}) \vee \neg x_1 \wedge (F1|_{\neg x_1} * F2|_{\neg x_1})$

Binary Operations

- Let the variable order be $x_1 < x_2 < \ldots < x_n$
- $F(x_1,\ldots,x_n) = F1(x_1,\ldots,x_n) * F2(x_1,\ldots,x_n)$
- Derivation:
  - $F(x_1,\ldots,x_n) = (x_1 \wedge F|_{x_1}) \vee (\neg x_1 \wedge F|_{\neg x_1})$
  - $= (x_1 \wedge (F1*F2)|_{x_1}) \vee (\neg x_1 \wedge (F1*F2)|_{\neg x_1})$
  - $= (x_1 \wedge (F1|_{x_1} * F2|_{x_1})) \vee$
  - $(\neg x1 \wedge (F1|_{\neg x_1} * F2|_{\neg x_1}))$

Computing F1*F2

- Compute F1*F2 by recursively computing
  - $F1|_{x_1} * F2|_{x_1}$
  - $F1|_{\neg x_1} * F2|_{\neg x_1}$
  - Where x_1 is the top variable in the variable order
    - $x_1 < x_2 < \ldots < x_n$
  - among all variables appearing in F1, F2
- Clearly x_2 is the top variable in the variable order among all variables in $F1|_{x_1} * F2|_{x_1}$
  - Hence computation of $F1|_{x_1} * F2|_{x_1}$ involves co-factoring w.r.t. x_2 and so on.

Efficiency Issues

- Computation of F1*F2 generates two recursive calls.
  - Use dynamic programming to remember results that have been computed already.

F1 * F2

$F1|_{x1} * F2|_{x1}$ $\quad$ $F1|_{\neg x1} * F2|_{\neg x1}$

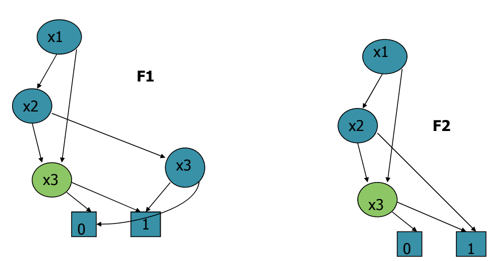
$F1|_{x1,x2} * F2|_{x1,x2}$ $\quad$ $F1|_{x1,\neg x2} * F2|_{x1,\neg x2}$ $\quad$ $F1|_{\neg x1,x2} * F2|_{\neg x1,x2}$

$F1|_{x1,x2} * F2|_{x1,x2}$

Pattern of recursive calls

Efficiency issues

- $F1|_{x1,\neg x2} * F2|_{x1,\neg x2}$ can re-use the result of
  - $F1|_{x1,x2} * F2|_{x1,x2}$ for example.



Efficiency Issues

- Any recursive call (f*f') in the computation of F1*F2 corresponds to
  - A pair of vertices, one each from the ROBDD of F1 and F2
  - If any such call was made before, the cached answer can be directly used.
  - Tremendous savings, since (f*f') might have generated many other recursive calls.
- Note: After (F1*F2) is computed, the resultant BDD must be Reduced as before.

Formal Def. of Apply

- Applying a logical operation * to two OBDDs representing boolean functions f, f'
- Defined as a recursive procedure Apply(F1, F2)
- Case 0: If Apply(F1,F2) has already been computed then return result from result-cache.
- Else
  - Let v1, v2 be the root nodes of the OBDD for F1, F2
  - Employ Case 1 or Case 2 or Case 3 or Case 4

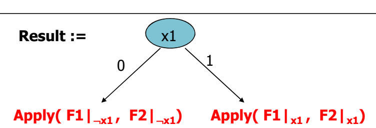
Case 1: Terminal Vertex

- If v1 and v2 are terminal vertices
- then result := value(v1) * value(v2)
- add result to result-cache;
- return(result)

Cases 2, 3, 4

- Let var(v1) = x1 and var(v2) = x2
  - x1 and x2 are the root vars. of OBDDs for F1,F2
- Since we are working with OBDDs, we can locate x1, x2 in the variable order
  - Case 2,3,4 differ in the positions of x1, x2 in the ordering of variables of F1, F2

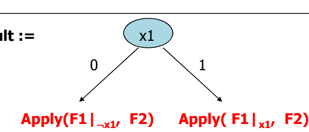
Case 2: x1 = x2



Add Result to result-cache;

Return result;

Case 3: x1 < x2

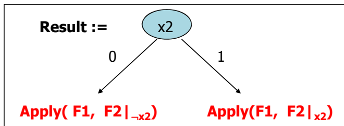


Add Result to result-cache;

Return result;

Note: x1 does not even appear in the OBDD for F2

Case 3: x2 < x1



Add Result to result-cache;
Return result;

Note: x2 does not even appear in the OBDD for F1

Summary

- BDD :A space efficient data structure for representing transition system.
- Can also represent intermediate sets of states during model checking in a space efficient fashion.
- Set operations performed during model checking can be achieved through boolean operations on BDD.
- Canonical form to detect fixed points.
- Next class:
  - Symbolic model checking algorithm which proceeds by manipulating BDDs.