CIS 673: Lecture 6: Sept 25
Symbolic Reachability

- Instead of enumerating states, compute with regions (state-sets) that are represented symbolically (eg. $20 \leq x \leq 99$)

- Implicit representation of transition relation and reachable states

- For propositional or enumerated modules, binary decision diagrams serve as a compact representation
  - Binary decision diagrams: representation for boolean functions due to Bryant [87]
  - Application to model checking: tool SMV by McMillan [91]
  - Popular in hardware applications

Symbolic Data Types

- Symbolic regions support following operations
  - $\cup$: $\mathbf{symreg} \times \mathbf{symreg} \mapsto \mathbf{symreg}$
  - $\cap$: $\mathbf{symreg} \times \mathbf{symreg} \mapsto \mathbf{symreg}$
  - $=$: $\mathbf{symreg} \times \mathbf{symreg} \mapsto \mathbf{bool}$
  - $\subseteq$: $\mathbf{symreg} \times \mathbf{symreg} \mapsto \mathbf{bool}$
  - $EmptySet : \mathbf{symreg}$

- Note: No enumeration, so number of states in a region is not an issue

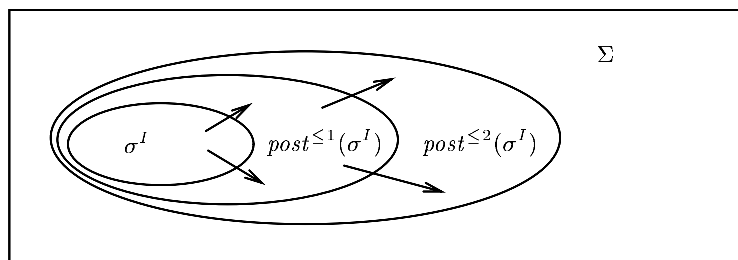
- Symbolic transition graph supports
  - $InitReg : \mathbf{symgraph} \mapsto \mathbf{symreg}$
  - $PostReg : \mathbf{symreg} {\times} \mathbf{symgraph} \mapsto \mathbf{symreg}$

Symbolic Search

Input: a transition graph G, and a region σ^T
Output: answer to reachability problem (G, σ^T).

```
input G: symgraph;  σ^T: symreg;
local σ^R: symreg;
begin
  σ^R := InitReg(G);
  repeat
    if σ^R ∩ σ^T ≠ EmptySet then
      return Yes;
    if PostReg(σ^R, G) ⊆ σ^R then
      return No;
    σ^R := σ^R ∪ PostReg(σ^R, G)
    forever
  end.
```

Symbolic Search



Like breadth-first search, guaranteed to terminate if the diameter of the graph is bounded: there is i such that every reachable state is reachable within i transitions from some initial state

Detour: Fixpoint Theory

- Partial order $\preceq$: reflexive, transitive, and anti-symmetric relation
- Least upper bound a of a set B wrt $\preceq$, denoted $\vee B$
  - $b \preceq a$ for all $b \in B$
  - if $b \preceq c$ for all $b \in B$, then $a \preceq c$
- Similarly, greatest lower bound $\wedge B$ defined
- Complete partial order: all sets have lub and glb
- The powerset of a set with $\subseteq$ is a CPO:
  - lub corresponds to set-union
  - glb corresponds to set-intersection

Fixpoint Theory

- Function f is monotonic: $a \preceq b$ then $f(a) \preceq f(b)$
- a is fixpoint of f: $f(a) = a$
- Knaster-Tarski fixpoint theorem:
  If f is monotonic then f has least and greatest fixpoints: μf and νf
- f is $\vee$-continuous: $f(\vee B) = \vee f(B)$
- Kleene fixpoint theorem: If f is monotonic and $\vee$-continuous, then $\mu f = \vee\{f^{\kappa}(\bot) \mid \kappa \in \mathbb{O}\}$
- For coutable sets: $\mu f = \vee\{f^{i}(\bot) \mid i \in \mathbb{N}\}$
- Summary: Least (or greatest) fixpoints can be approximated by starting from $\bot$, and applying f repeatedly

Symbolic reachability as fixpoint computation

- CPO: set of regions ordered by $\subseteq$
- Consider $f(\sigma) \;=\; \sigma^I \cup post_G(\sigma)$
- f is monotonic, $\cup$-continuous
- Least fixpoint of f: reachable region σ^R
- i-th approximation of reachable region: $f^i(\emptyset)$
- Many of our later symbolic algorithms will correspond to computing fixpoints (μ-calculus)

Symbolic Representation

- Type of variable x: $\mathbb{T}_x$
- Type of state over X: product type $\mathbb{T}_X$
- Type of transition: $\mathbb{T}_{X \cup X'}$
- Transition relation is, thus, a symbolic region over primed and unprimed variables
- Symbolic representation of a transition graph G
  - initial region $\{\sigma^I\}_s$ of type **symreg**$[\mathbb{T}_X]$
  - transition relation $\{\rightarrow\}_s$ of type **symreg**$[\mathbb{T}_{X \cup X'}]$
- We simply need to decide on how to represent regions

Computing Post

- Renaming: $\mathit{Rename}(x, y, \sigma)$ returns the renamed region $\sigma[x := y]$

- Renaming extends to variable-sets

- Existential quantifier elimination: For σ of type **symreg**$[\mathbb{T}_X]$, $\mathit{Exists}(x, \sigma)$ returns the region

$$\{s \in \Sigma_{X \setminus \{x\}} \mid (\exists m.\, s[x := m] \in \sigma)\}$$

- Existential quantifier elimination extends to variable sets

- Computing $\mathit{PostReg}(\sigma)$:
  - conjunct σ with $\{\rightarrow\}_s$ to obtain the set of transitions originating in σ
  - project the result onto the set X'
  - rename each primed variable x' to x

$$\mathit{PostReg}(\sigma, \{G\}_s) \;=\; \mathit{Rename}(X', X, \mathit{Exists}(X, \sigma \cap \{\rightarrow\}_s))$$

Summary

To implement symbolic reachability, we need

- implementation the data type symreg that supports $\cup$, $\cap$, $=$, $\subseteq$, Rename, and Exists
- an efficient way to compute, from the module text, the symbolic representation of
  - the initial region $\{\sigma^I\}_s$
  - the transition region $\{G\}_s$

Symbolic search using propositional formulas

- Represent regions by (parse trees) of boolean expression
- Union, intersection easy
- Renaming: textual substitution
- Quantifier elimination in linear time:

  $$Exists(x, p) \;=\; (p[x := true] \vee p[x := false]).$$

- Inclusion ($\subseteq$) or equivalence ($=$):
  - checking validity of propositional formulas
  - hard: coNP-complete
- Obtaining symbolic representations of σ^I and $\rightarrow$ is easy (no blow-up)

Symbolic reachability for Pete

- For module P_1: $X = \{pc_1, pc_2, x_1, x_2\}$
- Initial region q_1^I: $pc_1 = out$
- Symbolic rep of transition relation q_1^T:

$$
\begin{aligned}
&\vee (pc_1 = out \wedge pc_1' = req \wedge x_1' = x_2) \\
&\vee (pc_1 = req \wedge (pc_2 = out \vee x_1 \neq x_2) \wedge pc_1' = in \wedge x_1' = x_1) \\
&\vee (pc_1 = in \wedge pc_1' = out \wedge x_1' = x_1) \\
&\vee (pc_1' = pc_1 \wedge x_1' = x_1).
\end{aligned}
$$

- One step of fixpoint computation:

$$PostReg(q_1^I) = Rename(X', X, Exists(X, q_1^I \cap q_1^T))$$

- Simplifies to $pc_1 = out \vee pc_1 = req$

Can we improve formula representation?

- Propositional formulas not very satisfactory:
  - Each step of the computation may be expensive (equality test)
  - More importantly, size of $PostReg^{\leq i}(q^I)$ may grow with i, with no good heuristics for simplification

- Desirable properties of representation:
  - All operations (and hence, a single iteration) should be efficient
  - Should maintain compactness whenever possible: canonicity
  - Representing q^I and q^T from module text should be efficient

Ordered Binary Decision Graphs

- Representation for predicates over a set X of boolean vars
- The variables in X are totally ordered by $\prec$
- BDG is a directed acyclic graph
- Paths in the graph encode assignments to variables in X
- Terminal vertices classify paths into accepting and rejecting

Definition of BDG wrt $(X, \prec)$

- Vertices: a finite set V that is partitioned into two sets; internal vertices V^N and terminal vertices V^T

- Root: a root vertex v^I in V

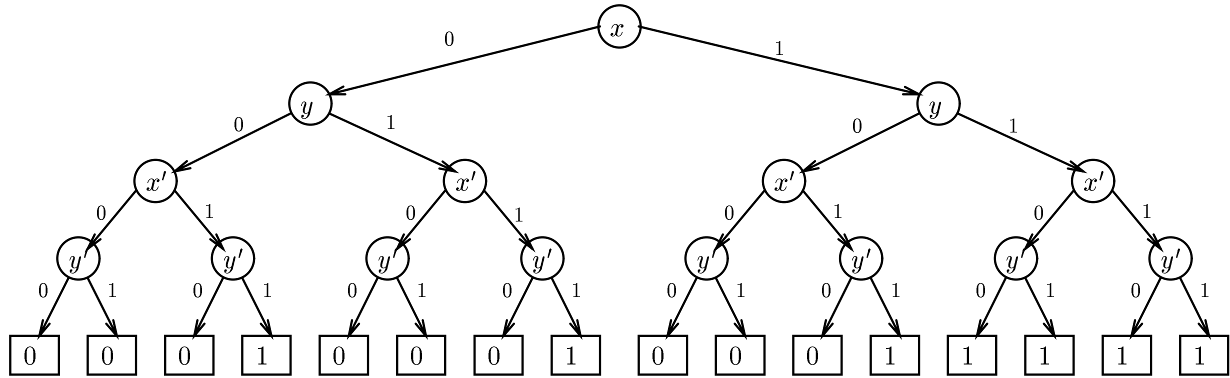
- Labeling: a labeling function *label* that labels each internal vertex with a variable in X, and each terminal vertex with a constant in **bool**

- Left edges: a function $\mathit{left} : V^N \mapsto V$ that maps each internal vertex v to a vertex $\mathit{left}(v)$ such that if $\mathit{left}(v)$ is an internal vertex then $\mathit{label}(v) \prec \mathit{label}(\mathit{left}(v))$

- Right edges: a function $\mathit{right} : V^N \mapsto V$ that maps each internal vertex v to a vertex $\mathit{right}(v)$ such that if $\mathit{right}(v)$ is an internal vertex then $\mathit{label}(v) \prec \mathit{label}(\mathit{right}(v))$.

Sample BDG

Function: $(x \wedge y) \vee (x' \wedge y')$
Ordering: $x \prec y \prec x' \prec y'$



Boolean Function of a BDG

- On every path from root to a terminal vertex, a variable appears 0 or 1 times
- With every vertex v, associate a boolean function $r(v)$
  - if v is terminal then $r(v)$ equals its label (0 or 1)
  - if v is internal then $r(v)$ equals

    $$(\neg label(v) \wedge r(left(v))) \vee (label(v) \wedge r(right(v)))$$

- BDG B represents the function $r(B) = r(v^I)$ asssociated with its root
- To check whether a state s satisfies $r(B)$ simply follow path according to values assigned by s

Isomorphism and Equivalence

- Two BDGs B and C are isomorphic if the corresponding labeled graphs are isomorphic
- Two BDGs B and C are equivalent if the boolean expressions $r(B)$ and $r(C)$ are equivalent.
- Deciding isomorphism is easy
- Deciding equivalence is difficult (so this is not what we want for symbolic reachability)
- Equivalence does not imply isomorphism
- Every vertex v can be viewed as a BDD (subgraph rooted at v), so we can talk about isomorphism and equivalence of vertices

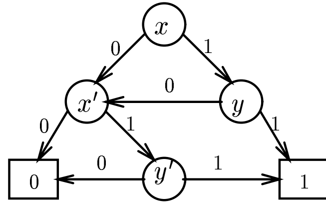
Ordered Binary Decision Diagrams

- BDD is obtained from a BDG by repeatedly applying following transformations:
  - Identify isomorphic subgraphs.
  - Eliminate internal vertices with identical left and right successors.

- An ordered binary decision diagram over a totally ordered set $(X, \prec)$ is an ordered binary decision graph B over $(X, \prec)$ with vertices V and root v^I such that
  - No isomorphic subgraphs: if v and w are two distinct vertices in V, then v is not isomorphic to w
  - No redundancy: for every internal vertex v, the two successors $\mathit{left}(v)$ and $\mathit{right}(v)$ are distinct.

Sample BDD

Function: $(x \wedge y) \vee (x' \wedge y')$
Ordering: $x \prec y \prec x' \prec y'$



Properties of BDDs

- Given an ordered set $(X, \prec)$, every boolean function has a unique, upto isomorphism, representation as a BDD (i.e. existence and canonicity)

- Isomorphism and equivalence are synonyms for BDDs

- Testing equivalence: easy (linear-time)

- $r(B)$ is valid iff B equals a single terminal vertex labeled with 1

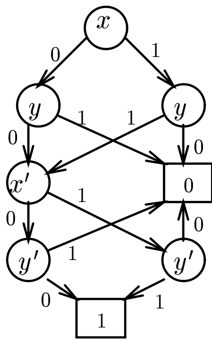
- Testing validity (and satisfiability) is easy (constant-time)

- This implies translating a propositional formula to a BDD can cause an exponential blow-up

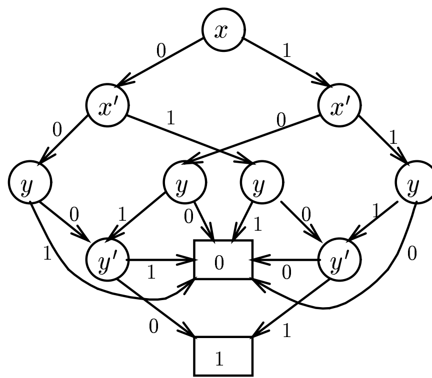
- Minimality: If B is a BDD and C is an equivalent BDG (with same ordering) then C has at least as many vertices as B

Ordering influences size

Two BDDs for $(x \leftrightarrow y) \wedge (x' \leftrightarrow y')$



Ordering: $x \prec y \prec x' \prec y'$



Ordering: $x \prec x' \prec y \prec y'$

More on variable ordering

- The size according to one ordering may be exponentially smaller than another ordering
- Figuring out the optimal ordering of variables is hard
- Theorem: The problem of checking, given a BDD B over $(X, \prec)$, whether the ordering $\prec$ is optimal for $r(B)$, is coNP-complete.
- Some functions have small size independent of ordering, eg. parity
- Some functions have large size independent of ordering, multiplication

BDD Implementation

- Vertices of all BDDs stored in a central pool
- Each BDD is simply an index into this global data structure
- Invariant property: no two vertices in the pool are isomorphic
- Advantages:
  - To check equivalence, compare indices
  - Sharing of subgraphs across nonisomorphic BDDs
- Performance depends on efficient garbage collection

BDD Data Structures

- Type of BDDs: bdd. Pointer to global data structure $BddPool$.

- Type of $BddPool$: **set of bddnode**

- Type of vertices: (k is number of variables)

$$\mathbf{bddnode} = ([1..k] \times \mathbf{bdd} \times \mathbf{bdd}) \cup \mathbf{bool}$$

- **bddnode** supports
  - $Label : \mathbf{bddnode} \mapsto [1..k]$
  - $Left : \mathbf{bddnode} \mapsto \mathbf{bdd}$
  - $Right : \mathbf{bddnode} \mapsto \mathbf{bdd}$

- $BddPool$ has following operations
  - Insert and IsMember
  - $Index : \mathbf{bddnode} \mapsto \mathbf{bdd}$
  - $BddPool[B]$ returns vertex pointed to by B

Insertions into *BddPool*

- *BddPool* is initialized to contain two terminal vertices 0 and 1
- Add vertices only via MakeVertex to ensure that no vertices are isomorphic

function MakeVertex
Input: $i\colon [1..k]$, $B_0, B_1\colon \mathbf{bdd}$.
Output: B : **bdd** such that $r(B)$ equivalent
    to $(\neg x_i \wedge r(B_0)) \vee (x_i \wedge r(B_1))$.
begin
  if $B_0 = B_1$ then return B_0 fi;
  if $\neg IsMember((i, B_0, B_1), BddPool)$ then
    $Insert((i, B_0, B_1), BddPool)$ fi;
  return $Index((i, B_0, B_1))$
  end.

Operations on BDDs

- Algorithms for boolean operations on BDDs
- Conjunction:
  - Input: Two pointers B_0 and B_1 into $BddPool$
  - Output: A pointer B such that the BDD represented by $BddPool[B]$ should be the conjunction of the BDDs represented by $BddPool[B_0]$ and $BddPool[B_1]$
- Recursive function $Conj$ to take conjunction of vertices

Algorithm for Conjunction

function Conj
input B_0, B_1 : **bdd**
output B : **bdd**
local v_0, v_1 : **bddnode**; $i, j : [1 \ldots k]$
    $B, B_{00}, B_{01}, B_{10}, B_{11}$: **bdd**

begin
  $v_0 := BddPool[B_0]$;
  $v_1 := BddPool[B_1]$;
  if $v_0 = 0$ or $v_1 = 1$ then return B_0 fi;
  if $v_0 = 1$ or $v_1 = 0$ then return B_1 fi;
  $i := Label(v_0)$; $j := Label(v_1)$;
  $B_{00} := Left(v_0)$; $B_{01} := Right(v_0)$
  $B_{10} := Left(v_1)$; $B_{11} := Right(v_1)$;
  if $i = j$ then
    $B :=$ MakeVertex(i,Conj(B_{00},B_{10}), Conj(B_{01},B_{11}))
  if $i < j$ then
    $B :=$ MakeVertex(i,Conj(B_{00},B_1),Conj(B_{01},B_1))
  if $i > j$ then
    $B :=$ MakeVertex(j,Conj(B_0,B_{10}),Conj(B_0,B_{11}))
  return B
  end.

Avoiding Recomputation

- Store computed results in a table *Done*
- Keys for *Done*: pairs of the form **bdd** $\times$ **bdd**
- Stores value of the type **bdd**
- Initially empty
- Before recursive calls, $Conj$ checks if $Done[(B_0, B_1)]$ has a value
- Upon termination $Conj$ stores the result in *Done*

Modified Algorithm

```
function Conj
begin
  v_0 := BddPool[B_0];  v_1 := BddPool[B_1];
  if v_0 = 0 or v_1 = 1 then return B_0 fi;
  if v_0 = 1 or v_1 = 0 then return B_1 fi;
  if Done[(B_0, B_1)] ≠⊥ then return Done[(B_0, B_1)]
  if Done[(B_1, B_0)] ≠⊥ then return Done[(B_1, B_0)]
  i := Label(v_0);  j := Label(v_1);
  B_00 := Left(v_0);  B_01 := Right(v_0)
  B_10 := Left(v_1);  B_11 := Right(v_1);
  if i = j then
    B := MakeVertex(i,Conj(B_00,B_10), Conj(B_01,B_11))
  if i < j then
    B := MakeVertex(j,Conj(B_00,B_1),Conj(B_01,B_1))
  if i > j then
    B := MakeVertex(j,Conj(B_0,B_10),Conj(B_0,B_11))
  Done[(B_0, B_1)] := B;
  return B
  end.
```

Operations on BDDs

- If B_0 contains m nodes and B_1 has n nodes then $Conj$ has time complexity $O(mn)$
- Quadratic bound tight: conjunction of two BDDs with n nodes can have n^2 nodes (like finite automata)
- Similar algorithms for union, negation, simplification (i.e. computing $B[x := 0]$)
- Now we can translate a propositional formula to its BDD representation
- Translation can cause exponential blow-up

Multi-valued Decision Diagrams (MDD)

- Just like BDD, except choice need not be binary

- Variables have enumerated types

- Branching degree at an internal vertex x is k if x can take k values:
  - An internal vertex labeled with x has a successor edge labeled with m for every $m \in \mathbb{T}_x$

- Terminal vertices have 0 or 1 label as before

- Useful for verification of enumerated modules

Symbolic Invariant Verification

To solve the invariant verification problem (P, p)

1. Choose variable ordering over $X_P \cup X'_P$
2. Build the BDDs for
   - Initial region of P
   - Target region (negation of invariant p)
   - Transition relation of P
3. Execute symbolic search algorithm

Partitioned Transition Relation

- Transition relation is conjunction of predicates
- Do not compute the conjunction a priori, but only as needed (on-the-fly symbolic representation)
- A conjunctively-partitioned representation of a boolean expression p is a set $\{B_1, \ldots B_k\}$ of BDDs such that p is equivalent to the conjunction $r(B_1) \wedge \cdots \wedge r(B_k)$
- The transition relation q^T is maintained as a set $\{q_1^T, \ldots q_k^T\}$ can have much smaller BDD representation
- In each iteration, if q is the current reachable region, we need to compute $q \wedge q^T$
- In each iteration, we need compute the conjunction $q \wedge q_1^T \wedge \cdots \wedge q_k^T$

- This is typically much smaller than BDD for q^T: only reachable transitions are considered (the domain of various conjuncts q_i^T is constrained, or simplified by q)

- For us, for every atom, there is a conjunct representing update command of the atom

- Further partitioning possible: eg. each conjunct is a disjunction of guarded commands, maintain this in a disjunctively partitioned format

Early Quantifier Elimination

- If p does not depend upon x then

$$\exists x.\, p \wedge q \;\equiv\; p \wedge \exists x.\, q$$

- BDD for $\exists x.\, q$ is likely to be smaller than BDD of q

- In computing *PostReg*, we need to compute

$$\exists X.\, (q \wedge q_1^T \wedge \cdots \wedge q_k^T)$$

- Strategy: take one conjunction at a time, eliminate (via existential quantification) as many variables as possible

- Quantifier elimination has a precedence over conjunction

Ordering of Conjuncts

- Effectiveness of early quantifier elimination depends on the ordering of conjuncts
- In 3-bit counter example, we want to compute

$$\exists \{out_0, out_1, out_2\}. \, (p \wedge q_0^T \wedge q_1^T \wedge q_2^T)$$

- If we compute $p \wedge q_0^T$ first, no existential quantification can be applied
- If we compute $p \wedge q_2^T$ first, out_2 can be eliminated
- Optimal computing strategy:

$$\exists \, out_0. \, (q_0^T \wedge \exists \, out_1. \, (q_1^T \wedge \exists \, out_2. \, (q_2^T \wedge p))).$$

Dynamic Variable Reordering

- Symbolic computation computes successive approximation q_i to the reachable region
- What if, at some iteration, BDD representation of q_i becomes too large?
- One solution: try to reorder variables
- Swapping adjacent variables, say x_i and x_{i+1}, can be done reasonable efficiently:
  - BDD structure and above level i and below level $i + 1$ stays unchanged
- Greedy heuristic: find the level at which the BDD is fattest, and try swapping with adjacent level. Repeat if desired
- All BDDs of current relevance must be updated: expensive step