CS6610: Software Engineering II

Symbolic Reachability - I
Implicit Representation

- Implicit representation of transition relation and reachable states
- For propositional or enumerated modules, binary decision diagrams (BDDs) serve as a compact representation
  - BDDs: representation for boolean functions due to Bryant
  - Application to model checking: tool SMV by McMillan
  - Popular in hardware applications
- Symbolic model checking recently won ACM's Theory in Practice award
Symbolic Data Types

- Instead of enumerating states, compute with regions (state-sets) that are represented symbolically
  - Example: \(20 \leq x \leq 99\)
- Note: No enumeration, so number of states in a region is not an issue
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.
Search Algorithm

**Input**: a transition graph G, and a region $\sigma^T$

**Output** answer to reachability problem $(G, \sigma^T)$

Input $G$: symgraph; $\sigma^T$: symreg
Local $\sigma^R$: symreg;

$\sigma^R := \text{InitReg}(G)$;
repeat
  if $\sigma^R \cap \sigma^T \neq \text{EmptySet}$ then
    return Yes;
  if $\text{PostReg}(\sigma^R, G) \subseteq \sigma^R$ then //or $\text{PostReg}(\sigma^R, G) \cup \sigma^R = \sigma^R$
    return No;
  $\sigma^R := \sigma^R \cup \text{PostReg}(\sigma^R, G)$
forever
Symbolic Data Types

- Symbolic regions support following operations
  - ∪: symreg × symreg → symreg
  - ∩: symreg × symreg → symreg
  - =: symreg × symreg → bool
  - ⊆: symreg × symreg → bool
  - EmptySet : symreg

- Symbolic transition graph supports
  - InitReg : symgraph → symreg
  - PostReg : symreg×symgraph → symreg
Symbolic Representation

- Symbolic representation of a transition graph
  - Type of variable \( x \): \( T_x \)
  - Type of state over \( X \): product type \( T_X (\prod_{x \in X} T_x) \)
  - Type of transition: \( T_{X \cup X'} \)
    - Transition relation is, thus, a symbolic region over primed and unprimed variables
  - initial region \( \{ \sigma^I \}_s \) of type symreg\([T_X]\]
  - transition relation \( \{ \rightarrow \}_s \) of type symreg\([T_{X \cup X'}]\]
- We need to decide on how to represent regions and compute PostReg
Computing Post

- **Renaming**
  - \( \text{Rename}(x, y, \sigma) \) returns the renamed region \( \sigma[x := y] \)
  - Extends to variable-sets \( \text{Rename}(X, Y, \sigma) \)

- **Existential quantifier elimination**
  - For \( \sigma \) of type \( \text{symreg}[T_X] \), \( \text{Exists}(x, \sigma) \) returns the region \( \{ s \in \Sigma_{x\setminus\{x\}} \mid (\exists m. s[x := m] \in \sigma) \} \)
  - Extends to variable sets \( \text{Exists}(X, \sigma) \)
Computing Post

- Computing PostReg(\(\sigma\)):  
  - conjunct \(\sigma\) with \(\{\rightarrow\}_s\) to obtain the set of transitions originating in \(\sigma\)  
  - project the result onto the set \(X'\)  
  - rename each primed variable \(x'\) to \(x\)  
- \(\text{PostReg}(\sigma, \{G\}_s) = \text{Rename}(X', X, \exists X (X, \sigma \cap \{\rightarrow\}_s))\)
Summary

- To implement symbolic reachability, we need
  - implementation of 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 $\{\sigma^I\}_s$ and the transition region $\{G\}_s$
Symbolic Search Using Propositional Formulas

- Represent regions by boolean expression
  - Obtaining symbolic representations of $\sigma^I$ and $\rightarrow$ is easy (no blow-up)
  - Union, intersection easy
  - Renaming: textual substitution
  - Quantifier elimination in linear time:
    - $\exists x, p = (p[x := true] \lor p[x := false])$
Symbolic Search Using Propositional Formulas

- Inclusion ($\subseteq$) or equivalence ($=$)
  - checking validity of propositional formulas
  - hard: coNP-complete
Can We Improve Formula Representation?

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

- Desirable properties of representation:
  - All operations should be efficient
  - Should maintain compactness whenever possible
  - Representing $q^I$ and $q^T$ from module text should be efficient
Representations of Boolean Functions?

- We often reduce problems to the manipulation of Boolean Functions
  - Truth Tables
  - Disjunctive Normal Form
    - Sum-of-Products
  - Conjunctive Normal Form
    - Product-of-Sums
Truth Table

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f ((a,b,c))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
DNF and CNF

- $f = abc + ab'c + a'bc$
- $f = (a + b + c) (a + b + c') (a + b' + c) (a' + b + c) (a' + b' + c)$
- $f = (a+b) c$
Review

- How about DNF?
  - $F = ab$
  - $F = abc + abc'$
  - $F = ab' + bc' + ca'$
  - $F = a'b + b'c + c'a$
Ordered Binary Decision Graphs

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

- A finite set $V$ of vertices
  - Internal vertices $V^I$ and terminal vertices $V^T$
  - Root: a root vertex $v^I$ in $V$
- Labeling: a labeling function that labels each internal vertex with a variable in $X$, and each terminal vertex with a constant 0 or 1
- Left edges: a function $left : V^I \rightarrow V$
  - If $left(v)$ is an internal vertex then $label(v) < label(left(v))$
  - Right edges similar
Sample BDG

Function: \((x \land y) \lor (x' \land y')\)
- Ordering: \(x < y < x' < y'\)
- On every path from root to a terminal vertex, a variable appears 0 or 1 times
Boolean Function of a BDG

- With vertex $v$, associate a function $r(v)$
  - if $v$ is terminal then $r(v)$ equals its label (0 or 1)
  - if $v$ is internal then $r(v)$ equals
    \[ (-\text{label}(v) \land r(\text{left}(v))) \lor (\text{label}(v) \land r(\text{right}(v))) \]
- BDG $B$ represents the function $r(B) = r(v^I)$
- 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.
- Deciding isomorphism is easy

Two BDGs B and C are equivalent if the boolean expressions r(B) and r(C) are equivalent.
- Equivalence does not imply isomorphism
- Deciding equivalence is difficult (so this is not what we want for symbolic reachability)
Binary Decision Tree

\[ f = (a+b) \cdot c \]
Binary Decision Diagrams

\[ f = (a+b) \cdot c \]
Binary Decision Diagrams

\[ f = (a+b) \cdot c \]
Binary Decision Diagrams

\[ f = (a+b) \cdot c \]
Binary Decision Diagrams

\[ f = (a+b) \cdot c \]
Binary Decision Diagrams

Reduced Ordered Binary Decision Diagram
Reduction Rules

- If the two successors of a node $v$ are identical, then remove $v$.

- If two nodes represent the same function, remove one and share other.
ROBDD

- **Reduced**
  - No more reduction rules can be applied
  - Represent each sub-function only once
  - Remove nodes where both successors are identical

- **Ordered**
  - Same variable ordering on all paths
  - Size of BDD depends on variable ordering
Example: Influence of Variable Ordering

\[ f = x_1 x_2 + x_3 x_4 + x_5 x_6 \]

(a) Variable order \( x_1, \ldots, x_6 \)  
(b) Variable order \( x_1, x_3, x_5, x_2, x_4, x_6 \)
How do we build a BDD?

- Idea:
  - Given a Boolean Function
  - Generate Decision Tree
  - Apply reduction to obtain BDD

- Bad idea!

- Why?
BDD Operations

- BDDs are compact
- Operations on BDDs are efficient
- How do they work?
BDD_AND

BDD_AND (BDD F, BDD G)
    if (terminal case)
        return (r = trivial result)
    else
        if computed_table_has_entry (F, G, r)
            return r
        else
            x = top_variable (F,G)
            t = BDD_AND (Fx, Gx)
            e = BDD_AND (Fx', Gx')
            if (t==e) return t
            r = find_or_add_unique_table (x, t, e)
            insert_computed_table (F, G, r)
            return r
Run-time Analysis

- Operations are carried out in polynomial time
  - Run-time can be bound by size(F) * size(G)
- Why?
- Idea: Count number of calls to BDD_AND
- Each call is a unique pair of one node from first BDD (F) and one node from second BDD (G)
  - ‘repeats’ are caught by computed table