CSEP590 – Model Checking and Automated Verification

Lecture outline for July 16, 2003

-Our Model Checker suffers from state space explosion:
  - -We need a more compact representation of the system
  - -A verification procedure to handle this compact system

-Let’s develop this compact representation – will be in terms of Ordered Binary Decision Diagrams (OBDDs)
  - -But first, we need some background on boolean functions

-Boolean functions
  - -Composed of boolean variables and operators (like in prop. logic)
  - -Representations: truth tables or prop. formulas. Both very inefficient in size of representation.

-Better representation: Binary Decision Diagrams (BDDs)
  - -Simpler form is called a Binary Decision Tree (bdt)
    - -Non-terminal nodes labeled with boolean var, terminals with either 0 or 1. Dashed line and solid line links root node x to 2 children (representing valuation of x of 0 and 1 respectively)

- (Example shown in class)
- If T is a bdt, then T represents a unique boolean function of the variables representing non-terminals
- But, bdt is still inefficient in size, in fact it is ~size of a truth table for representing boolean functions
- How improve? Eliminate redundancy!
  - For example, combine all 0-terminals and all 1-terminals
  - No longer have a tree, instead it is called a BDD
  - Defn: a subBDD is part of a BDD below a given node n such that n is the root of the subBDD
  - 3 ways of reducing bdt to its most compact form (called reduced). (Explained in detail via examples in class)
    - C1) Removal of duplicate terminal nodes
    - C2) Removal of redundant tests
    - C3) Removal of duplicate non-terminals
  - BDDs are actually dags (directed non-acyclic graphs)
    - We interpret links in BDDs as pointing down

        - BDDs are dags with a unique initial node
        - A BDD is reduced if no further C1-C3 optimizations can be applied
        - Note: BDDs also represent boolean functions just like bdts
- BDDs can often be compact representations of boolean functions (due to C1-C3 optimizations)
    - Checking satisfiability is easy – just look for path to a 1
- How perform operations of AND, NOT, OR on BDDs?
    - Given B_f and B_g representing functions f and g, then we can construct a BDD B representing f op g.
    - Inefficient methods – we'll see in class. We can improve on these (later in class)

-But we still have a problem: how check equivalency between BDDs?? Ouch.

- Impose an ordering on the variables occurring along ANY path of a BDD

- -Then, impose that ordering on all BDDs under consideration
- -Defn: if [x_1,…x_n] is an ordered list of variables with duplicates and B is a BDD with all these variables, then B has ordering [x_1,…x_n] if all variables of B occur in the list and for every occurrence of x_i followed by x_j along any path in B, we have $i<j$

-Such BDDs with an ordering are called Ordered Binary Decision Diagrams (OBDDs)

- -We'll see some examples in class
- -To perform operations on 2 BDDs, we'll require that they have comparable variable orderings (ie, no inversions in their orderings)
- -Theorem: if 2 reduced OBDDs with comparable variable orderings represent the same boolean function, then they have identical structure! => unique OBDD for every boolean formula
- -OBDDs can yield exponential savings in space (ex in class)

-Note: the variable ordering can make a HUGE difference in the size of the reduced OBDD
- -Price we pay for advantages of OBDDs
- -Luckily, good heuristics exist for find good orderings
- -In-class example of computing reduced OBDD

-We now consider some basic algorithms for operating on reduced OBDDs
- -1) algorithm *reduce*: transforms OBDD to its reduced form
  - -Start with lowest layer (terminals), and work up to root
  - -Let lo(n) denote dashed child of n, hi(n) as solid child of n
  - -Label nodes with an integer id (node n = id(n)) s.t. subOBDDs with root nodes n and m denote same function iff id(n) = id(m)
  - -Assume we've labeled all nodes in layers > i. Given x_i-node n in layer i, there are 3 ways to label it
    - -1) if id(lo(n)) = id(hi(n)), id(n) = that label (C2 test)

-2) if there exists a node m s.t. n and m have the same variable x_i, and id(lo(n)) = id(lo(m)) and id(hi(n)) = id(hi(m)), then id(n) = id(m). (C3 case)

-3) Otherwise, id(n) gets next unused integer

-Algorithm is efficient in the number of nodes in OBDD

-2) algorithm *apply*: implement operations on boolean functions represented as OBDDs

-Given OBDDs B_f and B_g for functions f and g, apply(op,B_f,B_g) returns the reduced OBDD for f op g.

-Defn: a restriction of function f is of the form f[0/x] or f[1/x] which denotes f with all instances of x set to 0 (and 1 respectively)

-Shannon Expansion (really due to Boole):

-Function $f \equiv !x*f[0/x] + x*f[1/x]$

-Allows for recursion on boolean formulas!

-Idea: based on Shannon expansion for f op g

-$f \text{ op } g \equiv !x_i*(f[0/x_i] \text{ op } g[0/x_i]) + x_i*(f[1/x_i] \text{ op } g[1/x_i])$

-Call this equation '(SE)'

-Algorithm proceeds from root r_f of B_f and root r_g of B_g downwards in a recursive manner based on cases of r_f and r_g:

- -1) if r_f and r_g are terminal nodes with labels l_f and l_g, then compute the value of l_f op l_g and return the resulting OBDD as B_0 or B_1 (depending on value, either 0 or 1)
- -2) if both roots are x_i-nodes, then create an x_i-node n with a dashed line to apply(op,lo(r_f),lo(r_g)) and a solid line to apply(op,hi(r_f),hi(r_g)). This is precisely what equation (SE) tells us to do!
- -3) if r_f is an x_i-node and r_g is either a terminal or x_j-node with $j<i$, then no x_i-node exists in B_g because the 2 OBDDs have a comparable variable ordering and we are proceeding top down. Thus, g is independent of variable x_i. Therefore, we create an x_i-node n with a dashed line to apply(op,lo(r_f),r_g) and a solid line to apply(op,hi(r_f),r_g)
- -4) Symmetric case: r_f takes the role of r_g in 3) and vice versa

-There are also 2 other algorithms, *restrict* and *exists*. But these are straightforward and are covered in the book. They won't be necessary to understand for our understanding of symbolic model checking.

-Now that we have developed the OBDD representation, we can use it as our basic data structure in model checking.

-Recall that our model checking algorithm basically manipulates intermediate sets of states. Operations are performed on these sets.

-We now show how to represent sets of states as OBDDs, and how these same operations on sets can be performed on OBDDs. Thus, we can apply our model checker using the compact OBDD representation instead of the huge, transition system representation.

-This technique, in its basic form, is called Symbolic Model Checking

-Provided a huge breakthrough in the early 1990s

-First, we need to encode sets of states as OBDDs and the transition relation from our model as an OBDD

- -Sets of states encoding:
  - -Each state s is represented by a unique vector of boolean values ($v_1, \ldots v_n$). A subset T of S is a boolean function f_T mapping ($v_1, \ldots v_n$) onto 1 if s is in T and 0 otherwise. f_T is known as the characteristic function.
  - -How do we construct this vector? Use the set of atomic propositions (AP). Let's assume a fixed ordering on the elements of AP, say $x_1, x_2, \ldots, x_n$. Then s is represented by the vector ($v_1, \ldots, v_n$) where for each i, $v_i = 1$ if $x_i \in L(s)$ and 0 otherwise.
  - -We require that vectors are unique, namely that if $L(s_1) = L(s_2)$, then $s_1 = s_2$
    - -If not, we can just add extra, dummy atomic props to our model

-We can then represent s as the OBDD for the boolean function $l_1 * l_2 * \ldots * l_n$ where l_i is x_i if $x_i \in L(s)$ and $!x_i$ otherwise

-A set of states is then just the disjunction of the formulas for each state in the set!

-We'll see an example in class, it really is pretty straightforward ☺

-Now, let's represent the transition relation of a system as an OBDD

-Recall: transition relation $\rightarrow$ is just a subset of S x S

-Think of it as a set of (curr state, next state) pairs

-We then represent such pairs using 2 boolean vectors, one for the curr state and one for the next state in the pair

-Suppose we have the transition $s \rightarrow s'$

-This is represented by the pair of boolean vectors $((v_1,\ldots,v_n),(v_1',\ldots,v_n'))$ where $v_1 = 1$ if $p_i \in L(s)$ and 0 otherwise, and $v_i' = 1$ if $p_i \in L(s')$ and 0 otherwise

-Then, a transition s→s' is simply represented as an OBBD of the boolean function $(l_1 * l_2 * \ldots * l_n) * (l_1' * l_2' * \ldots * l_n')$

-To form the transition relation → for the entire system, it is just the OBDD for the boolean formula that is the disjunction of all such individual transition formulae

-Once again, we'll see an example in class that will make it clear ☺

-What does this mean? We can use our apply algorithm to simulate intersection, union, and complementation of sets on OBDD representation of state sets!

-One last thing though…our model checking algorithm also had 2 trickier routines that it used, namely $pre_\exists(X)$ which given a set of states X, returns all states that can transition into X, and $pre_\forall(X)$ that returns those states that can transition *only* into X. These are used in the routines for operators EX and EU, and AF respectively.

-How do we do those in terms of our OBDD representation?

-We first note that $\mathrm{pre}_{\forall}(X) = S - \mathrm{pre}_{\exists}(S-X)$. Thus, we only need to concern ourselves with $\mathrm{pre}_{\exists}(X)$, which is nice
-In short, it can be done, but it is a confusing equation involving our *exists* and *apply* algorithms. It is given in the textbook on page 357.

-The final part of this lecture with examine the model checker SMV (for Symbolic Model Verifier) developed at CMU and widely considered one of the best model checkers out there. I will be handing out a handout in class describing the language used to program SMV, with examples. We will discuss it, and you will use it for your PS4.