Symbolic model checking. Binary decision diagrams

20 oct. 2005

Explicit-state model checking

Need to represent each state individually
⇒ size of the state space severely limits applicability (size of a state determines how many states we can represent in memory)
– typically, limited to a few million states

State space explosion problem: for composed systems, state space is product of component state spaces ⇒ exponential in number of components
⇒ Much of focus in formal verification is scaling to large state spaces

If reachable state set is much smaller than potential complete state space, can try to encode reached states using fewer bits
(*bitstate hashing*, used in SPIN).
However, this is an *approximation*: on reaching an already hashed state, search stops (even though actual state may be different)
⇒ part of state space may remain unexplored
⇒ method is not sound

Exploration with individual states and sets

Problem: compute set of states reachable from initial states
(**EF** *true*)
– by forward traversal of graph starting from initial states
– R: set of explored states; F: *frontier* reached in current step

With *individual states*
$R = \emptyset$; $F = S_0$
while $(F \neq \emptyset)$
  choose $s \in F$;
  $F \leftarrow F \setminus \{s\}$; $R \leftarrow R \cup \{s\}$
  forall s' with $s \to s'$
    if $s' \notin F \cup R$
      $F \leftarrow F \cup \{s'\}$

With *state sets*
$R = \emptyset$; $F = S_0$
while $(F \not\subseteq R)$
  $R \leftarrow R \cup F$
  $F = \{s' \in S | \exists s \in F \,.\, s \to s'\}$
  // or $F = F \setminus R$
  // with test $F \neq \emptyset$

⇒Algorithm can be expressed much easier is *successor* set of a state set can be computed *in a single operation*
⇒set R of reached states grows in each iteration but is finite

Symbolic model checking

- A new approach, based on exploring state sets
  – idea: a set may sometimes be represented (by a forumula) in a much more compact way than individually representing each state
  – need: efficient representation and manipulation for state sets and transition relation
  [McMillan'92]
  – with binary decision diagrams (BDDs) [Bryant'86]
- key idea 1: working with state *sets*
  – used also for infinite state sets (continuous-time or hybrid systems)
- key idea 2: iterative computation until no more change
  ⇒notion of *fixpoint*

Fixpoint representations

Def: $x \in D$ is a *fixpoint* for $f : D \to D$ if $f(x) = x$.
Def: A *lattice* is a partially ordered set in which any finite subset has a least upper bound and a greatest lower bound
Ex: powerset (set of subsets) $\mathcal{P}(S)$ of S, with $\subseteq$ as order

– We work with functions $\tau : \mathcal{P}(S) \to \mathcal{P}(S)$ over the *lattice* $\mathcal{P}(S)$

– We regard $S' \subseteq S$ as a *predicate* over S: $S'(s) = true \Leftrightarrow s \in S'$
în particular: $\emptyset = false$, $S = true$
⇒ $\tau : \mathcal{P}(S) \to \mathcal{P}(S)$ is a *predicate transformer*
Def:

- τ is *monotone* if $P \subseteq Q \Rightarrow \tau(P) \subseteq \tau(Q)$
- τ is *union-continuous* if for any sequence $P_1 \subseteq P_2 \subseteq \ldots$ we have $\tau(\cup_i P_i) = \cup_i \tau(P_i)$
- τ is *intersection-continuous* if for any sequence $P_1 \supseteq P_2 \supseteq \ldots$ we have $\tau(\cap_i P_i) = \cap_i \tau(P_i)$

Fixpoint theorems

A monotone predicate transformer over $\mathcal{P}(S)$ always has
– a minimal fixpoint, denoted $\mu Z.\tau(Z)$
– and a maximal fixpoint, denoted $\nu Z.\tau(Z)$ [Tarski]

If S is finite and τ is monotone, then τ is continuous for union and intersection.

τ monotone ⇒ $\tau^i(\mathit{False}) \subseteq \tau^{i+1}(\mathit{False})$ şi $\tau^i(\mathit{True}) \supseteq \tau^{i+1}(\mathit{True})$

If τ is monotone and S is finite, there exist $i, j \geq 0$ such that
$\forall k \geq i$, $\tau^k(\mathit{False}) = \tau^i(\mathit{False})$ and $\forall k \geq j$, $\tau^k(\mathit{True}) = \tau^j(\mathit{True})$

If τ is monotone and S is finite, there exist $i, j \geq 0$ such that
$\mu Z.\tau(Z) = \tau^i(\mathit{False})$ and $\nu Z.\tau(Z) = \tau^j(\mathit{True})$

Computing the minimal/maximal fixpoint

function *Lfp*(τ : *Trans*) : *Pred*
  Q := *False*;
  $Q' := \tau(Q)$;
  while $(Q' \neq Q)$ **do**
    $Q := Q'$;
    $Q' := \tau(Q)$;
  return Q;

function *Gfp*(τ : *Trans*) : *Pred*
  Q := *True*;
  $Q' := \tau(Q)$;
  while $(Q' \neq Q)$ **do**
    $Q := Q'$;
    $Q' := \tau(Q)$;
  return Q;

Fixpoint relations for CTL

We identify a CTL formula f with the set of states that satisfy it: $\{s \mid M, s \models f\}$

- **AF** $f = \mu Z \,.\, f \vee$ **AX** Z **EF** $f = \mu Z \,.\, f \vee$ **EX** Z
- **AG** $f = \nu Z \,.\, f \wedge$ **AX** Z **EG** $f = \nu Z \,.\, f \wedge$ **EX** Z
- **A** $[f_1$ **U** $f_2] = \mu Z \,.\, f_2 \vee (f_1 \wedge$ **AX** $Z)$
- **E** $[f_1$ **U** $f_2] = \mu Z \,.\, f_2 \vee (f_1 \wedge$ **EX** $Z)$
- **A** $[f_1$ **R** $f_2] = \nu Z \,.\, f_2 \wedge (f_1 \vee$ **AX** $Z)$
- **E** $[f_1$ **R** $f_2] = \nu Z \,.\, f_2 \wedge (f_1 \vee$ **EX** $Z)$

minimal fixpoint: liveness properties: **F**
maximal fixpoint: safety properties (invariants): **G**

Symbolic model checking algorithm

– works by structural decomposition of the formula
Check(f) returns $\{s \in S \mid M, s \models f\}$ (set of states satisfying f)
Check$(p) = \{s \in S \mid p \in L(s)\}$ atomic propositions
Check$(\neg f) = S \setminus$ *Check*(f) complement
Check$(f \wedge g)$ = *Check*$(f) \cap$ *Check*(g) intersection
Check(**EX** f) = *CheckEX*(*Check*(f))
  CheckEX$(f(\bar{v})) = \exists \bar{v}' \,.\, [f(\bar{v}') \wedge R(\bar{v}, \bar{v}')]$ relational product
Check(**E** $[f$ **U** $g]$) = *CheckEU*(*Check*(f), *Check*(g))
  using **E** $[f_1$ **U** $f_2] = \mu Z \,.\, f_2 \vee (f_1 \wedge$ **EX** $Z)$ și funcția *Lfp*
Check(**EG** f) = *CheckEG*(*Check*(f))
  using **EG** $f = \nu Z \,.\, f \wedge$ **EX** Z and the functional *Gfp*

All of these basic operations can be expressed using BDDs

Binary Decision Diagrams (BDDs)

- a *canonical* and *compact* representation of Boolean functions
- with efficient manipulation algorithms
  [R. Bryant, "Graph-based algorithms for boolean function manipulation", *IEEE Transactions on Computers*, 1986]
- significant impact on formal verification:
  ACM Kanellakis Award for Theory & Practice, 1998
  – Randal E. Bryant: BDDs ('86)
  – Edmund M. Clarke, E. Allen Emerson: model checking ('81)
  – Ken McMillan: symbolic model checking ('92)

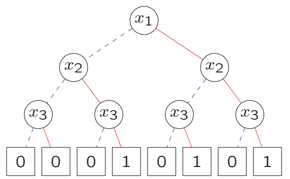
Representations for Boolean functions

$f : B^n \to B$ – can encode both state sets and transition relations

- Usual representations (truth tables, Karnaugh diagrams, canonical sum of minterms) have *exponential size*
- $\Rightarrow$ improvements: reduced sums of products, factorizations, etc.
  – still exponentiale for some common functions (e.g. parity)
- some elementary operations may lead to exponential growth (e.g., negation)
- for non-canonical representations it is difficult to test:
  – equivalence (checking needed after changes in circuit design)
  – satisfiability: $\exists x_1, \cdots x_n \,.\, f(x_1, \cdots, x_n) = 1$?
  $\forall x \,.\, f_1(x) = f_2(x) \equiv \neg \exists x \,.\, f_1(x) \oplus f_2(x) = 1$

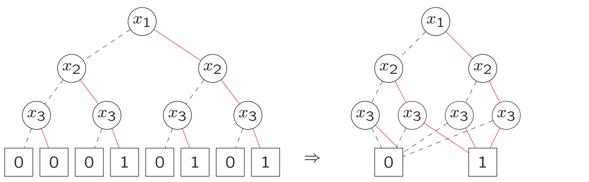
Binary decision trees

- terminal nodes: function value (0 or 1)
- nonterminal nodes: variables
- branches (children): $low(v)$ (left) / $high(v)$ (right): correspond to assignment of 0 or 1 for the variable in the node

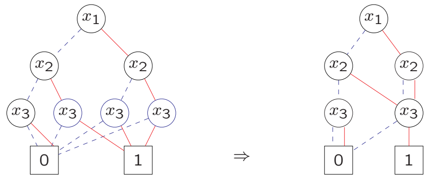


BDDs: obtained from binary decision trees applying 3 reduction rules

Reduction rule 1: Merge terminal nodes

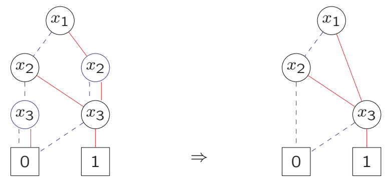


Reduction rule 2: Merge isomorphic nodes



$f(n_1) = f(n_2) \Rightarrow$ merge n_1 and n_2

Reduction rule 3: Eliminate redundant test



$low(n) = high(n) \Rightarrow$ eliminates testing at node n

Basic BDD properties

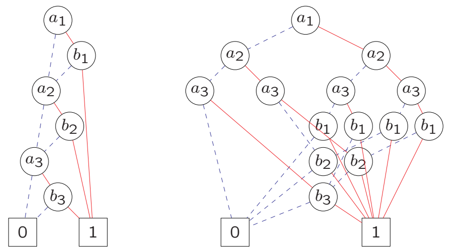
The 3 rules can be applied whatever the variable ordering down the tree.
In an *ordered* BDD (OBDD): one additional condition:
On all paths from root to terminals, variables appear in same order
(there exists a global ordering of variables)

Theorem: For any Boolean function, its representation as an *ordered* BDD, reduced according to rules 1–3 is *unique* up to isomorphism.
$\Rightarrow$ *canonical* representation
$\Rightarrow$ equivalence or satisfiability checking in constant time

Note: A subgraph rooted as a BDD node is also a BDD
$\Rightarrow$ BDDs for several functions may share subgraphs in the same forest

Effect of variable ordering

Consider the function: $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$



Linear growth: $2(n+1)$ Exponential growth: 2^{n+1}

BDD algorithms: Apply

function *Apply*$(f, g$: *OBDD*$, op$: *Operator*$)$: *OBDD*
if *is_leaf*$(f) \wedge$ *is_leaf*(g) **return** $op(f, g)$;
elsif (f, g, op, h) in apply_cache **return** h;
else
  $x :=$ *topvar*(f) // variable at root of f
  $y :=$ *topvar*(g)
  if (*ord*(x) = *ord*(y)) // x = y = same variable
    $h :=$ *find_bdd*$(x,$ *Apply*$(f\,|_{x=0}, g\,|_{x=0}, op),$ *Apply*$(f\,|_{x=1}, g\,|_{x=1}, op))$
    // *find_bdd* creates a new BDD if not already existent
  elsif (*ord*$(x) <$ *ord*(y)) // x before y in ordering
    $h :=$ *find_bdd*$(x,$ *Apply*$(f\,|_{x=0}, g, op),$ *Apply*$(f\,|_{x=1}, g, op))$
  else $h :=$ *find_bdd*$(y,$ *Apply*$(f, g\,|_{y=0}, op),$ *Apply*$(f, g\,|_{y=1}, op))$
  insert (f, g, op, h) in apply_cache
  return h

BDD algorithms: relational product

function *Relprod*(f, g : *OBDD*, E : *varset*) : *OBDD*
if f = *false* $\vee$ g = *false* **return** *false*
elsif f = *true* $\wedge$ g = *true* **return** *true*
elsif (f, g, E, h) in relprod_cache **return** h
else
  x := *topvar*(f) // variable at root of f
  y := *topvar*(g)
  z := *topmost*(x, y) // first in variable order
  h_0 := *RelProd*($f\mid_{z=0}, g\mid_{z=0}, E$)
  h_1 := *RelProd*($f\mid_{z=1}, g\mid_{z=1}, E$)
  if $z \in E$ h := *bdd_or*(h_0, h_1) /* $\exists z \,.\, h = h_0 \vee h_1$ */
  else h := *bdd_if_then_else*(z, h_1, h_0)
  insert (f, g, E, h) in relprod_cache
  return h

Complexity of BDD algorithms

• Reduction (to canonical form)	$O(\lvert G\rvert \cdot \log \lvert G\rvert)$
• Apply ($f_1 \langle op\rangle f_2$)	$O(\lvert G_1\rvert \cdot \lvert G_2\rvert)$
• Restrict ($f\mid_{x_i=b}$)	$O(\lvert G\rvert \cdot \log \lvert G\rvert)$
• Compose ($f_1\mid_{x_i=f_2}$)	$O(\lvert G_1\rvert^2 \cdot \lvert G_2\rvert)$
• Satisfy-one (un $\bar{x}$ cu $f(\bar{x}) = 1$)	$O(n)$
• Satisfy-count ($\lvert\{\bar{x} \mid f(\bar{x}) = 1\}\rvert$)	$O(\lvert G\rvert)$

Logarithmic factors can be eliminated
(by more sophisticated algorithms or hashing)

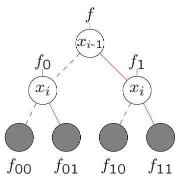
Relational product may have exponential complexity

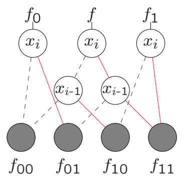
Implementation

- There are mature BDD libraries (packages) (CMU, Cal, CUDD)
- In a typical application, many BDDs have common subgraphs
  $\Rightarrow$ pointers into a graph with unique root
- Memory management: reference counter and garbage collection
- Many optimizations and heuristics
  – memory layout and traversal for efficient caching
  – parallel and distributed algorithms, etc.

Dynamic variable reordering

- Variable ordering is critical for BDD size
- Functions exist with exponential size BDDs regardless of ordering (e.g., middle bit of a multiplier [Bryant'91])
- shape and size of BDDs evolves during computation
  $\Rightarrow$ *dynamic* variable reordering is important
  – transparent for verification algorithms constructed on top
  – reordering adjacent levels does not change pointers into BDD





BDD variants and applications

- choice of other decompositions for Boolean functions:
  – OBDD: Boole-Shannon decomposition $f = \bar{x} \wedge f\mid_{x=0} \vee x \wedge f\mid_{x=1} = \bar{x} \wedge f_{\bar{x}} \vee x \wedge f_x$
  – $f = f_{\bar{x}} \oplus x \wedge f_{\delta x}$ Reed-Muller decomposition
  – $f = f_x \oplus \bar{x} \wedge f_{\delta x}$ positive Davio decomposition
- Multiterminal BDDs: allow arbitrary terminal nodes (typically integers)
- BDDs for arithmetic representations: $f = x_0 + 2 * x_1 + 4 * x_2 + \ldots$

Applications

- Mainly: CAD (equivalence checking) and formal verification
- Compact representations for data with some regularities/repetitions, but difficult to express analytically:
  – coding theory, large data structures, indexing, computational biology

Symbolic model checking with BDDs

System represented as binary encoding for states and atomic propositions
$\Rightarrow$ use BDDs for state sets, transition relation

Check(p) = $\{s \in S \mid p \in L(s)\}$	*bdd_if_then_else*($p, 1, 0$)
Check($\neg f$) = $S \setminus$ *Check*(f)	*bdd_not*
Check($f \wedge g$) = *Check*(f) $\cap$ *Check*(g)	*bdd_and*
Check(**EX** f) = *CheckEX*(*Check*(f))	
CheckEX($f(\bar{v})$) = $\exists \bar{v}' \,.\, [f(\bar{v}') \wedge R(\bar{v}, \bar{v}')]$	*RelProd*($f, R, \bar{v}'$)
Check(**E** [f **U** g]) = *CheckEU*(*Check*(f), *Check*(g))	
E [f_1 **U** f_2] = $\mu Z \,.\, f_2 \vee (f_1 \wedge$ **EX** $Z)$	algorithm *Lfp*
Check(**EG** f) = *CheckEG*(*Check*(f))	
EG $f = \nu Z \,.\, f \wedge$ **EX** Z	algorithm *Gfp*

Partitioning the transition relation

Monolithic transition relation – grows – can become major obstacle in building system model to fit in memory

- *disjunctive* partitioning (asynchronous systems)
  $$R(\overline{v},\overline{v}') = R_1(\overline{v},\overline{v}') \vee \cdots \vee R_n(\overline{v},\overline{v}')$$
  because of distributivity $\exists\overline{v}'[f(\overline{v}') \wedge R(\overline{v},\overline{v}')] =$
  $$= \exists\overline{v}'[f(\overline{v}') \wedge R_1(\overline{v},\overline{v}')] \vee \cdots \vee \exists\overline{v}'[f(\overline{v}') \wedge R_n(\overline{v},\overline{v}')]$$
- conjunctive partitioning (for synchronous systems)
  $\exists$ does not distribute over $\wedge$, but may exploit locality
  (if R_i does not depend on all next-state variables $\overline{v}'$):
  $$R(\overline{v},\overline{v}') = R_1(\overline{v},v_1') \wedge \cdots \wedge R_n(\overline{v},v_n')$$
  $\exists\overline{v}'[f(\overline{v}') \wedge R(\overline{v},\overline{v}')] =$
  $$= \exists v_n'[\cdots \exists v_1'[f(\overline{v}') \wedge R_0(\overline{v},v_1'] \wedge R_1(\overline{v},v_1')] \cdots \wedge R_n(\overline{v},v_n')]$$

(perform conjunction and quantification successively for each component)

Symbolic model checking with fairness

Recall: fairness constraint is : $F = \{P_1, P_2, \cdots, P_n\}$, with $P_i \subseteq S$

EG f is true in the maximal set Z such that:
– all states of Z satisfy f
– $\forall P_k \in F, s \in Z$ there is a path from s to a state of $Z \cap P_k$
(passing only through states that satisfy f)

$\Rightarrow$ can be expressed as fixpoint and thus computed symbolically

$$\mathbf{EG}_{\,fair} f = \nu Z \,.\, f \wedge \bigwedge_{i=1}^{n} \mathbf{EX}\,\mathbf{E}\,[f\,\mathbf{U}\,(Z \wedge P_k)]$$

Likewise for the other fundamental operators:

$$\mathbf{EX}_{\,fair}\, f = \mathbf{EX}\,(f \wedge fair)$$
$$\mathbf{EU}_{\,fair}\,(f,g) = \mathbf{EU}\,(f, g \wedge fair)$$

Counterexample generation

Main advantages of model checking:
– completely automated
– generates counterexamples that identify errors

- for existential formulas (**E**) : produces a *witness* path for which the formula is true
- for universal formulas (**A**): produces a counterexample
- counterexample for a universal formula is witness for its negation (its dual existential formula)

Witness for **EF** f

– minimal fixpoint: **EF** $f = \mu Z \,.\, f \vee \mathbf{EX}\, Z$
– compute and retain successive approximations $f = Q_0 \subseteq Q_1 \subseteq \ldots \subseteq Q_k$
– Q_k: set of states from which f can be reached in at most k steps
– find intersection $Q_k \cap S_0 \neq \emptyset$
(first traversal: backwards, symbolic)
– choose $s_k \in S_0 \cap Q_k$
– compute set *Succ*(s_k) of successors for s_k
– must have nonempty intersection Q_{k-1} (from s_k f is reachable in at most k steps, so there is a successor reaching it in $k-1$ steps)
– choose $s_{k-1} \in$ *Succ*$(s_k) \cap Q_{k-1}$, etc. until $Q_0 = f$
(second traversal, forward, through individual states)
– we have found path $s_k \to \ldots \to s_0$ reaching f