System Specification, Verification and Synthesis (SSVS) – CS 4830/7485, Fall 2019

14: Formal Verification:
Binary Decision Diagrams (BDDs)

Stavros Tripakis

Northeastern University
Khoury College of Computer Sciences

BDDs

Binary decision trees

Binary decision tree:

- A tree representing all possible variable assignments, and corresponding truth values of a boolean expression.
- For n variables, the tree has $1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$ nodes (including the leaves).

Let's draw the binary decision tree for

$$(z_1 \wedge z_3) \vee (z_2 \wedge z_3)$$

(assuming the order of variables z_1, z_2, z_3).

From binary decision trees to BDDs

Main idea: make the representation compact (i.e., smaller) by eliminating redundant nodes.

- If two subtrees (including leaves) T_1 and T_2 are identical then keep only T_1. All incoming links to T_2 are redirected to T_1.
- If both the true-branch and the false-branch of a node v lead to the same node v', then node v is redundant: v can be removed, with its incoming links being redirected to v'.

The result is a **reduced ordered binary decision diagram** (ROBDD).
It is a **DAG**: directed acyclic graph.
We often use BDD to mean ROBDD.

From binary decision trees to BDDs

Main idea: make the representation compact (i.e., smaller) by eliminating redundant nodes.

- If two subtrees (including leaves) T_1 and T_2 are identical then keep only T_1. All incoming links to T_2 are redirected to T_1.
- If both the true-branch and the false-branch of a node v lead to the same node v', then node v is redundant: v can be removed, with its incoming links being redirected to v'.

The result is a **reduced ordered binary decision diagram** (ROBDD).
It is a **DAG**: directed acyclic graph.
We often use BDD to mean ROBDD.

Let's try this on the following formulas:

$$a + b, \qquad \text{and} \qquad (z_1 \wedge z_3) \vee (z_2 \wedge z_3)$$

From binary decision trees to BDDs

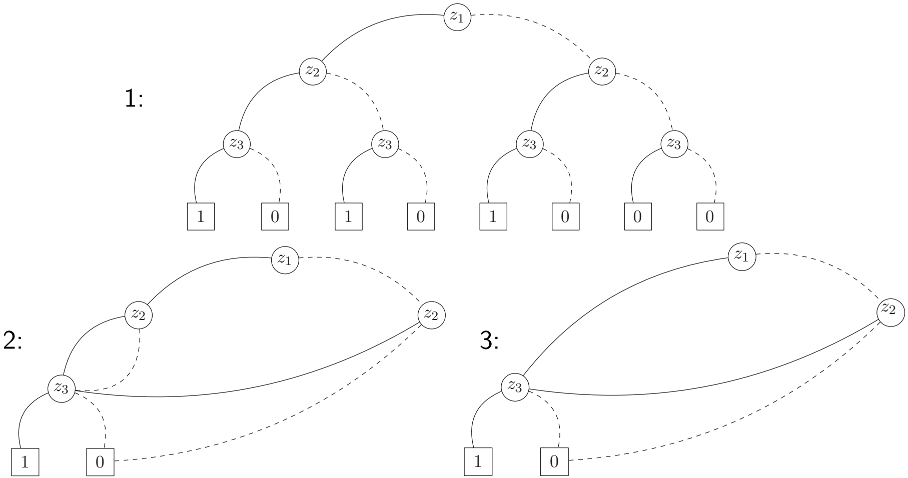

Figure taken from [Baier and Katoen, 2008].

BDDs: a canonical representation of boolean functions

ROBDDs are a **canonical** representation of boolean functions.

This means that two boolean functions (or expressions) f_1 and f_2 are equivalent iff their corresponding ROBDDs (for the same variable ordering) are identical.

BDDs: a canonical representation of boolean functions

ROBDDs are a **canonical** representation of boolean functions.

This means that two boolean functions (or expressions) f_1 and f_2 are equivalent iff their corresponding ROBDDs (for the same variable ordering) are identical.

Is this an important property? What is an example where it is useful?

BDDs: a canonical representation of boolean functions

ROBDDs are a **canonical** representation of boolean functions.

This means that two boolean functions (or expressions) f_1 and f_2 are equivalent iff their corresponding ROBDDs (for the same variable ordering) are identical.

Is this an important property? What is an example where it is useful?

Recall the symbolic reachability algorithm stopping criterion:

$$tmp \Leftrightarrow Reachable$$

If B and B' are the BDDs representing tmp and $Reachable$, respectively, then $tmp \Leftrightarrow Reachable$ holds iff B and B' are identical.

The bad news: variable ordering matters greatly

- BDD size depends on variable ordering
  - For the same boolean function, different variable orderings may result BDDs which are very different in size.
  - For example, consider the function

$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$$

    and the two orderings:

$$x_1, y_1, x_2, y_2, x_3, y_3$$

    and

$$x_1, x_2, x_3, y_1, y_2, y_3$$

The bad news: variable ordering matters greatly

- BDD size depends on variable ordering
  - For the same boolean function, different variable orderings may result BDDs which are very different in size.
  - For example, consider the function

$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$$

and the two orderings:

$$x_1, y_1, x_2, y_2, x_3, y_3$$

and

$$x_1, x_2, x_3, y_1, y_2, y_3$$

- Some BDDs have exponential size no matter which ordering we pick.
- Deciding whether a given order is optimal is NP-hard.
- Land of heuristics ...

Operations on BDDs

We want to compute set-theoretic, or equivalently, logical, operations on BDDs:

- Check for emptiness / satisfiability.
- Check for universality / validity.
- Intersection / conjunction.
- Union / disjunction.
- Complementation / negation.

Operations on BDDs

We want to compute set-theoretic, or equivalently, logical, operations on BDDs:

- Check for emptiness / satisfiability.
- Check for universality / validity.
- Intersection / conjunction.
- Union / disjunction.
- Complementation / negation.

Which of these operations are easy to perform on ROBDDs?

Easy operations on BDDs

- Check for emptiness / satisfiability.
  - Check whether the BDD is the leaf 0. If yes $\Rightarrow$ empty / unsat.
- Check for universality / validity.
  - Check whether the BDD is the leaf 1. If yes $\Rightarrow$ valid.
- Complementation / negation.
  - Replace the leaf 0 with 1, and 1 with 0.

Easy operations on BDDs

- Check for emptiness / satisfiability.
  - Check whether the BDD is the leaf 0. If yes $\Rightarrow$ empty / unsat.
- Check for universality / validity.
  - Check whether the BDD is the leaf 1. If yes $\Rightarrow$ valid.
- Complementation / negation.
  - Replace the leaf 0 with 1, and 1 with 0.

We next look at conjunction and disjunction, which are not so trivial.

Shannon expansion

Let f be a boolean expression and let x be a boolean variable.

Recall that

$$f[x \rightsquigarrow 0]$$

is a new formula f' obtained by replacing any occurrence of x in f by 0.

Similarly for $f[x \rightsquigarrow 1]$.

Shannon expansion

Let f be a boolean expression and let x be a boolean variable.

Recall that

$$f[x \rightsquigarrow 0]$$

is a new formula f' obtained by replacing any occurrence of x in f by 0.

Similarly for $f[x \rightsquigarrow 1]$.

$f[x \rightsquigarrow 1]$ and $f[x \rightsquigarrow 0]$ are called the (positive and negative) **cofactors** of f, and are denoted f_x and $f_{\overline{x}}$.

Shannon expansion

Let f be a boolean expression and let x be a boolean variable.

Recall that

$$f[x \rightsquigarrow 0]$$

is a new formula f' obtained by replacing any occurrence of x in f by 0.

Similarly for $f[x \rightsquigarrow 1]$.

$f[x \rightsquigarrow 1]$ and $f[x \rightsquigarrow 0]$ are called the (positive and negative) **cofactors** of f, and are denoted f_x and $f_{\overline{x}}$.

Then

$$f \quad \Leftrightarrow \quad \underbrace{\overline{x} \cdot f_{\overline{x}} \; + \; x \cdot f_x}$$

this is called the Shannon expansion of f

(For brevity, we denote $\wedge$ as $\cdot$ and $\vee$ as $+$, and $\neg x$ as $\overline{x}$.)

Shannon expansion and BDDs

$$f \quad \Leftrightarrow \quad x \cdot f_x \, + \, \overline{x} \cdot f_{\overline{x}}$$

This is the essence of binary decision trees and BDDs: if f is the root, then

- f_x is the sub-tree rooted at the 1-branch ("*true*"-branch) child of f
- $f_{\overline{x}}$ is the sub-tree rooted at the 0-branch ("*false*"-branch) child of f

Recursive application of boolean operations based on Shannon expansion

Suppose $\odot$ is some boolean operation (e.g., conjunction or disjunction).

Let f and g be two boolean expressions, and x be a boolean variable (usually f and g refer to x, but they don't have to).

Then

$$f \odot g \quad \Leftrightarrow \quad \overline{x} \cdot (f_{\overline{x}} \odot g_{\overline{x}}) \; + \; x \cdot (f_x \odot g_x)$$

Recursive application of boolean operations based on Shannon expansion

Suppose $\odot$ is some boolean operation (e.g., conjunction or disjunction).

Let f and g be two boolean expressions, and x be a boolean variable (usually f and g refer to x, but they don't have to).

Then

$$f \odot g \quad \Leftrightarrow \quad \overline{x} \cdot (f_{\overline{x}} \odot g_{\overline{x}}) \;+\; x \cdot (f_x \odot g_x)$$

For instance, if $\odot$ is conjunction:

$$f \cdot g \quad \Leftrightarrow \quad \overline{x} \cdot f_{\overline{x}} \cdot g_{\overline{x}} \;+\; x \cdot f_x \cdot g_x$$

Recursive application of boolean operations based on Shannon expansion

Suppose $\odot$ is some boolean operation (e.g., conjunction or disjunction).

Let f and g be two boolean expressions, and x be a boolean variable (usually f and g refer to x, but they don't have to).

Then

$$f \odot g \quad \Leftrightarrow \quad \overline{x} \cdot (f_{\overline{x}} \odot g_{\overline{x}}) \; + \; x \cdot (f_x \odot g_x)$$

For instance, if $\odot$ is conjunction:

$$f \cdot g \quad \Leftrightarrow \quad \overline{x} \cdot f_{\overline{x}} \cdot g_{\overline{x}} \; + \; x \cdot f_x \cdot g_x$$

This leads to the `apply` function.

The `apply` function

- Takes as input:
  - A boolean operation $\odot$ (e.g., conjunction or disjunction).
  - Two BDDs B_f and B_g (with the same variable ordering) representing two boolean functions f and g.

- Computes as output:
  - A BDD B representing $f \odot g$:

$$B = \texttt{apply}(\odot, B_f, B_g) \quad \text{such that} \quad B \Leftrightarrow B_{f \odot g}$$

The `apply` function

- Takes as input:
  - A boolean operation $\odot$ (e.g., conjunction or disjunction).
  - Two BDDs B_f and B_g (with the same variable ordering) representing two boolean functions f and g.
- Computes as output:
  - A BDD B representing $f \odot g$:

$$B = \texttt{apply}(\odot, B_f, B_g) \quad \text{such that} \quad B \Leftrightarrow B_{f \odot g}$$

- Operates recursively based on Shannon expansion.
- Resulting BDD may not be reduced, so needs to be generally reduced afterwards.

The `apply` function

We are computing $\texttt{apply}(\odot, B_f, B_g)$. Let v_f and v_g be the root nodes of B_f and B_g respectively.

There are the following cases to consider:

1. Both v_f and v_g are leaves (i.e., 0 or 1). Then, `apply` returns the leaf BDD with truth value $v_f \odot v_g$.
2. Both v_f and v_g are internal x-*nodes*, i.e., corresponding to variable x. Then, let B_f^x, B_g^x be the *positive sub-BDDs* (i.e., positive cofactors, i.e., BDDs rooted at the *true*-branch children) of v_f and v_g, respectively; and similarly with $B_f^{\overline{x}}, B_g^{\overline{x}}$. Then:
    1. Recursively compute BDD $B_x := \texttt{apply}(\odot, B_f^x, B_g^x)$.
    2. Recursively compute BDD $B_{\overline{x}} := \texttt{apply}(\odot, B_f^{\overline{x}}, B_g^{\overline{x}})$.
    3. Create and return a new BDD with root x and B_x as positive sub-BDD and $B_{\overline{x}}$ as negative sub-BDD.

    The justification for this comes directly from

$$f \odot g \quad \Leftrightarrow \quad \overline{x} \cdot (f_{\overline{x}} \odot g_{\overline{x}}) \, + \, x \cdot (f_x \odot g_x)$$

The apply function (continued)

3. v_f is an internal x-node, but v_g is either a leaf (0 or 1) or an internal y-node, with $y > x$, i.e., variable y is *after* x in the ordering (y is lower in the tree). Then we know, since B_f and B_g must follow the same variable ordering, that B_g is independent from x at this point in the tree. So we proceed as follows:
   1. Recursively compute BDD $B_x := \texttt{apply}(\odot, B_f^x, B_g)$.
   2. Recursively compute BDD $B_{\overline{x}} := \texttt{apply}(\odot, B_f^{\overline{x}}, B_g)$.
   3. Create and return a new BDD with root x and B_x as positive sub-BDD and $B_{\overline{x}}$ as negative sub-BDD.

The `apply` function (continued)

3. v_f is an internal x-node, but v_g is either a leaf (0 or 1) or an internal y-node, with $y > x$, i.e., variable y is *after* x in the ordering (y is lower in the tree). Then we know, since B_f and B_g must follow the same variable ordering, that B_g is independent from x at this point in the tree. So we proceed as follows:
    1. Recursively compute BDD $B_x := \texttt{apply}(\odot, B_f^x, B_g)$.
    2. Recursively compute BDD $B_{\overline{x}} := \texttt{apply}(\odot, B_f^{\overline{x}}, B_g)$.
    3. Create and return a new BDD with root x and B_x as positive sub-BDD and $B_{\overline{x}}$ as negative sub-BDD.

Do you see room for optimization here?

The `apply` function (continued)

3. v_f is an internal x-node, but v_g is either a leaf (0 or 1) or an internal y-node, with $y > x$, i.e., variable y is *after* x in the ordering (y is lower in the tree). Then we know, since B_f and B_g must follow the same variable ordering, that B_g is independent from x at this point in the tree. So we proceed as follows:
   1. Recursively compute BDD $B_x := \texttt{apply}(\odot, B_f^x, B_g)$.
   2. Recursively compute BDD $B_{\overline{x}} := \texttt{apply}(\odot, B_f^{\overline{x}}, B_g)$.
   3. Create and return a new BDD with root x and B_x as positive sub-BDD and $B_{\overline{x}}$ as negative sub-BDD.

   Do you see room for optimization here?

   E.g., when $\odot$ is $+$ and v_g is 0 or 1. If 0, return v_f. If 1, return 1.

The `apply` function (continued)

3. v_f is an internal x-node, but v_g is either a leaf (0 or 1) or an internal y-node, with $y > x$, i.e., variable y is *after* x in the ordering (y is lower in the tree). Then we know, since B_f and B_g must follow the same variable ordering, that B_g is independent from x at this point in the tree. So we proceed as follows:

    1. Recursively compute BDD $B_x := \texttt{apply}(\odot, B_f^x, B_g)$.
    2. Recursively compute BDD $B_{\overline{x}} := \texttt{apply}(\odot, B_f^{\overline{x}}, B_g)$.
    3. Create and return a new BDD with root x and B_x as positive sub-BDD and $B_{\overline{x}}$ as negative sub-BDD.

    Do you see room for optimization here?

    E.g., when $\odot$ is $+$ and v_g is 0 or 1. If 0, return v_f. If 1, return 1.

4. Symmetric to case 3 above, but with v_g being higher in the tree than v_f instead of lower.

The apply function: example

Let's try apply(+) on the two BDDs below:

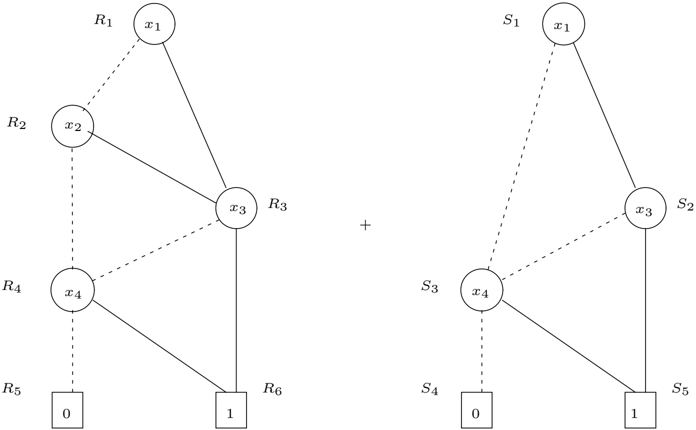

Figure taken from [Huth and Ryan, 2004].

Existential quantifier elimination

Recall that if x is a boolean variable then:

$$\exists x : f \quad \Leftrightarrow \quad f[x \rightsquigarrow 0] \vee f[x \rightsquigarrow 1] \quad \Leftrightarrow \quad f_{\overline{x}} \vee f_x$$

Let B_f be the BDD for f. How to compute the BDD for $\exists x : f$?

We know how to compute disjunction of BDDs already. It suffices to be able to compute substitutions like $f[x \rightsquigarrow 0]$.

Existential quantifier elimination

Recall that if x is a boolean variable then:

$$\exists x : f \quad \Leftrightarrow \quad f[x \rightsquigarrow 0] \vee f[x \rightsquigarrow 1] \quad \Leftrightarrow \quad f_{\overline{x}} \vee f_x$$

Let B_f be the BDD for f. How to compute the BDD for $\exists x : f$?

We know how to compute disjunction of BDDs already. It suffices to be able to compute substitutions like $f[x \rightsquigarrow 0]$.

This is simple:

- For every x-node v in B_f, eliminate v and redirect all incoming links to the 0-child of v.
- (If we wanted $f[x \rightsquigarrow 1]$ instead, we would redirect them to the 1-child of v.)
- We must then reduce the resulting BDD.

Putting it all together

Recall: Symbolic Reachability Analysis Algorithm

```
Reachable := Init;
terminate := false;
repeat
    tmp := Reachable ∨ succ(Reachable);
    if tmp ⇔ Reachable then
        terminate := true;
    else
        Reachable := tmp;
    end if
until terminate
return Reachable;
```

where

$$\mathbf{succ}\big(\phi(\vec{x})\big) := \big(\exists \vec{x} : \phi(\vec{x}) \wedge \mathit{Trans}(\vec{x}, \vec{x}')\big)[\vec{x}' \leadsto \vec{x}]$$

We have all the ingredients to implement this algorithm using BDDs:

- $\mathit{Init}, \mathit{Reachable}, \mathit{tmp}$ are each represented as a BDD on state variables $\vec{x}$.
- Trans is represented as another BDD on $\vec{x}, \vec{x}'$.
- We know how to compute $\wedge, \vee, \exists$ on BDDs.
- Renaming variables $[\vec{x}' \leadsto \vec{x}]$ is straightforward also.
- We know how to check $\Leftrightarrow$ on BDDs.

Bibliography

Baier, C. and Katoen, J.-P. (2008).
Principles of Model Checking.
MIT Press.

Bryant, R. (1986).
Graph-based algorithms for boolean function manipulation.
IEEE Trans. on Computers.

Clarke, E., Grumberg, O., and Peled, D. (2000).
Model Checking.
MIT Press.

Huth, M. and Ryan, M. (2004).
Logic in Computer Science: Modelling and Reasoning about Systems.
Cambridge University Press.