Static Analysis and Verification

SAV 2024/2025

Tomáš Vojnar

vojnar@fit.vutbr.cz

Brno University of Technology
Faculty of Information Technology
Božetěchova 2, 612 66 Brno

Binary Decision Diagrams

BDDs

Introduction

- BDDs were introduced by Randal E. Bryant:
  - Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers, C-35(8):677–691, 1986.

- BDDs provide a (usually) very compact and canonical representation of Boolean functions (i.e., functions of the form $\{0,1\}^k \longrightarrow \{0,1\}$, $k \geq 0$), corresponding to propositional formulae (possibly representing finite sets or relations).

- BDDs have a form of rooted, directed, connected, acyclic graph, which consists of internal Boolean decision nodes and terminal Boolean result nodes.

- BDDs may be viewed to arise from Boolean decision trees by removing redundancies from them (merging isomorphic sub-trees, removing useless nodes with isomorphic children).

- Operations on BDDs are done without uncompressing the represented objects.

- Applications: synthesis of circuits, symbolic verification, fault tree analysis, decision procedures, automata with large alphabets in pattern matching, quantum circuit simulation, program synthesis from examples (FlashFill), . . .

From Formulae to BDDs

❖ The propositional formula $\varphi = (a \wedge b \wedge c) \vee (a \wedge b \wedge \neg c)$ may be represented by:

(a) its truth table

a	b	c	φ
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

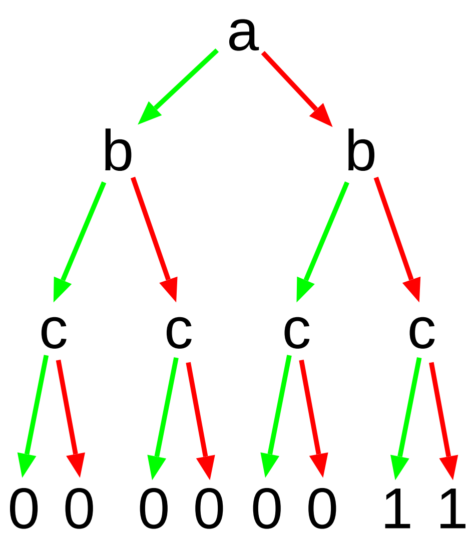
From Formulae to BDDs

❖ The propositional formula $\varphi = (a \wedge b \wedge c) \vee (a \wedge b \wedge \neg c)$ may be represented by:

(a) its truth table

a	b	c	φ
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(b) a decision tree
(a not reduced BDD)



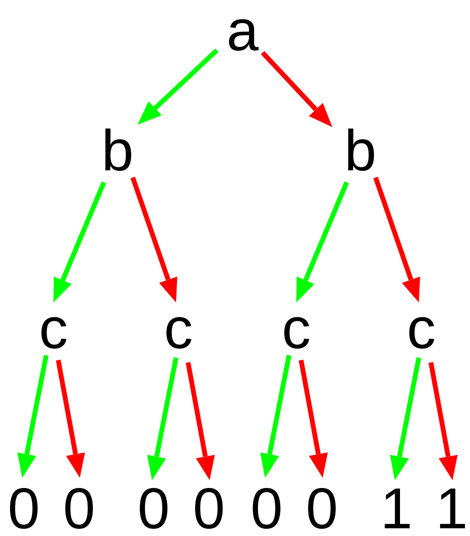
From Formulae to BDDs

❖ The propositional formula $\varphi = (a \wedge b \wedge c) \vee (a \wedge b \wedge \neg c)$ may be represented by:

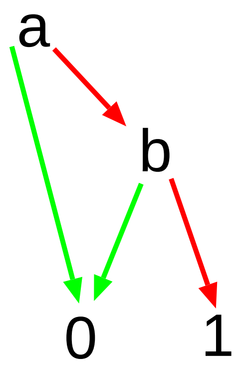
(a) its truth table

a	b	c	φ
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

(b) a decision tree
(a not reduced BDD)



(c) a (reduced) BDD



A Formal Definition of BDDs

❖ A BDD G over a set of Boolean variables Var is defined as a 7-tuple $G = (N, T, var, low, high, root, val)$ where:

- N is a finite set of non-terminal (internal) nodes,
  T is a finite set of terminal nodes (leaves), $N \cap T = \emptyset$.

A Formal Definition of BDDs

❖ A BDD G over a set of Boolean variables Var is defined as a 7-tuple $G = (N, T, var, low, high, root, val)$ where:

- N is a finite set of non-terminal (internal) nodes,
  T is a finite set of terminal nodes (leaves), $N \cap T = \emptyset$.
- $var\colon N \longrightarrow Var$ labels the internal nodes by variables.

A Formal Definition of BDDs

❖ A BDD G over a set of Boolean variables Var is defined as a 7-tuple $G = (N, T, var, low, high, root, val)$ where:

- N is a finite set of non-terminal (internal) nodes,
  T is a finite set of terminal nodes (leaves), $N \cap T = \emptyset$.
- $var\colon N \longrightarrow Var$ labels the internal nodes by variables.
- $low, high\colon N \longrightarrow N \cup T$ define the low and high successors of internal nodes $n \in N$, for the value of $var(n)$ being 0 or 1, respectively.
  - It is required that G is acyclic, i.e., $\nexists n \in N\colon n(low \cup high)^{+} n$.

A Formal Definition of BDDs

❖ A BDD G over a set of Boolean variables Var is defined as a 7-tuple $G = (N, T, var, low, high, root, val)$ where:

- N is a finite set of non-terminal (internal) nodes,
  T is a finite set of terminal nodes (leaves), $N \cap T = \emptyset$.
- $var \colon N \longrightarrow Var$ labels the internal nodes by variables.
- $low, high \colon N \longrightarrow N \cup T$ define the low and high successors of internal nodes $n \in N$, for the value of $var(n)$ being 0 or 1, respectively.
  - It is required that G is acyclic, i.e., $\nexists n \in N \colon n(low \cup high)^+ n$.
- $root \in N \cup T$ is the root node such that $\forall n \in (N \cup T) \setminus \{root\} \colon root(low \cup high)^+ n$ (note that $root$ cannot have any incoming edges without breaking the acyclicity requirement).

A Formal Definition of BDDs

❖ A BDD G over a set of Boolean variables Var is defined as a 7-tuple $G = (N, T, var, low, high, root, val)$ where:

- N is a finite set of non-terminal (internal) nodes,
  T is a finite set of terminal nodes (leaves), $N \cap T = \emptyset$.
- $var : N \longrightarrow Var$ labels the internal nodes by variables.
- $low, high : N \longrightarrow N \cup T$ define the low and high successors of internal nodes $n \in N$, for the value of $var(n)$ being 0 or 1, respectively.
  - It is required that G is acyclic, i.e., $\nexists n \in N : n(low \cup high)^+ n$.
- $root \in N \cup T$ is the root node such that $\forall n \in (N \cup T) \setminus \{root\} : root(low \cup high)^+ n$ (note that $root$ cannot have any incoming edges without breaking the acyclicity requirement).
- $val : T \longrightarrow \{0, 1\}$ labels the leaves by their values.

A Formal Definition of BDDs

❖ A BDD G over a set of Boolean variables Var is defined as a 7-tuple $G = (N, T, var, low, high, root, val)$ where:

- N is a finite set of non-terminal (internal) nodes,
  T is a finite set of terminal nodes (leaves), $N \cap T = \emptyset$.
- $var\colon N \longrightarrow Var$ labels the internal nodes by variables.
- $low, high\colon N \longrightarrow N \cup T$ define the low and high successors of internal nodes $n \in N$, for the value of $var(n)$ being 0 or 1, respectively.
  - It is required that G is acyclic, i.e., $\nexists n \in N\colon n(low \cup high)^+ n$.
- $root \in N \cup T$ is the root node such that $\forall n \in (N \cup T) \setminus \{root\}\colon root(low \cup high)^+ n$ (note that $root$ cannot have any incoming edges without breaking the acyclicity requirement).
- $val\colon T \longrightarrow \{0, 1\}$ labels the leaves by their values.

❖ For convenience, we often assume that Var is indexed by some bijection $f\colon I \leftrightarrow Var$ over the set of indices $I = \{1, \ldots, n\}$, yielding an indexed family of variables denoted $\{v_i\}_{i \in I}$.

Functions Represented by BDDs

❖ A node $x \in N \cup T$ of a BDD $G = (N, T, var, low, high, root, val)$ over an indexed family of variables $\{v_i\}_{i \in I}$, $I = \{1, \ldots, k\}$, $k \geq 0$, represents the Boolean function $f_x \colon \{0,1\}^k \longrightarrow \{0,1\}$ defined as follows:

1. If $x \in T$, then $f_x(v_1, \ldots, v_k) = val(x)$.
2. If $x \in N$ and $var(x) = v_i$ for some $i \in I$, then
   $f_x(v_1, \ldots, v_k) = (\neg v_i \wedge f_{low(x)}(v_1, \ldots, v_k)) \vee (v_i \wedge f_{high(x)}(v_1, \ldots, v_k))$.

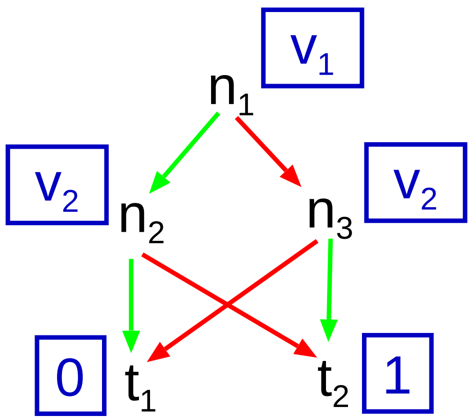
❖ G itself represents the function $f_{root}(v_1, \ldots, v_k)$.

Functions Represented by BDDs

- A node $x \in N \cup T$ of a BDD $G = (N, T, var, low, high, root, val)$ over an indexed family of variables $\{v_i\}_{i \in I}$, $I = \{1, \ldots, k\}$, $k \geq 0$, represents the Boolean function $f_x \colon \{0,1\}^k \longrightarrow \{0,1\}$ defined as follows:
  1. If $x \in T$, then $f_x(v_1, \ldots, v_k) = val(x)$.
  2. If $x \in N$ and $var(x) = v_i$ for some $i \in I$, then $f_x(v_1, \ldots, v_k) = (\neg v_i \wedge f_{low(x)}(v_1, \ldots, v_k)) \vee (v_i \wedge f_{high(x)}(v_1, \ldots, v_k))$.

- G itself represents the function $f_{root}(v_1, \ldots, v_k)$.

- An example:



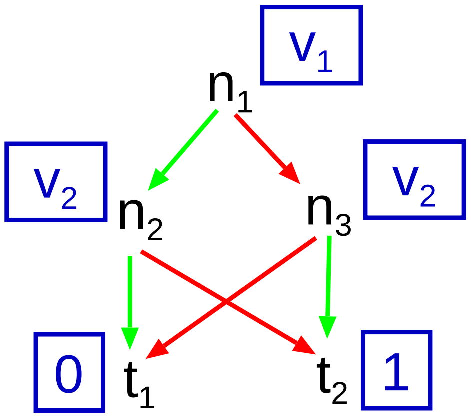
Functions Represented by BDDs

❖ A node $x \in N \cup T$ of a BDD $G = (N, T, var, low, high, root, val)$ over an indexed family of variables $\{v_i\}_{i \in I}$, $I = \{1, \ldots, k\}$, $k \geq 0$, represents the Boolean function $f_x \colon \{0,1\}^k \longrightarrow \{0,1\}$ defined as follows:

1. If $x \in T$, then $f_x(v_1, \ldots, v_k) = val(x)$.
2. If $x \in N$ and $var(x) = v_i$ for some $i \in I$, then $f_x(v_1, \ldots, v_k) = (\neg v_i \wedge f_{low(x)}(v_1, \ldots, v_k)) \vee (v_i \wedge f_{high(x)}(v_1, \ldots, v_k))$.

❖ G itself represents the function $f_{root}(v_1, \ldots, v_k)$.

❖ An example:



$$f_{t_1}(v_1, v_2) = 0,\ f_{t_2}(v_1, v_2) = 1,$$
$$f_{n_2}(v_1, v_2) = (\neg v_2 \wedge f_{t_1}(v_1, v_2)) \vee (v_2 \wedge f_{t_2}(v_1, v_2)) = v_2,$$
$$f_{n_3}(v_1, v_2) = (\neg v_2 \wedge f_{t_2}(v_1, v_2)) \vee (v_2 \wedge f_{t_1}(v_1, v_2)) = \neg v_2,$$
$$f_{n_1}(v_1, v_2) = (\neg v_1 \wedge v_2) \vee (v_1 \wedge \neg v_2).$$

Reduced BDDs

- Two BDDs $G_1 = (N_1, T_1, var_1, low_1, high_1, root_1, val_1)$ and $G_2 = (N_2, T_2, var_2, low_2, high_2, root_2, val_2)$ over the same set of variables are isomorphic iff there exists a bijection $h\colon N_1 \cup T_1 \longleftrightarrow N_2 \cup T_2$ such that:

  1. $H(N_1) = N_2$ and $H(T_1) = T_2$ for the pointwise extension H of h to sets of elements.
  2. $\forall n \in N_1$:
     $h(low_1(n)) = low_2(h(n)) \wedge$
     $h(high_1(n)) = high_2(h(n)) \wedge$
     $var_1(n) = var_2(h(n))$.
  3. $h(root_1) = root_2$.
  4. $\forall t \in T_1 : val_1(t) = val_2(h(t))$.

Reduced BDDs

- Two BDDs $G_1 = (N_1, T_1, var_1, low_1, high_1, root_1, val_1)$ and $G_2 = (N_2, T_2, var_2, low_2, high_2, root_2, val_2)$ over the same set of variables are isomorphic iff there exists a bijection $h\colon N_1 \cup T_1 \longleftrightarrow N_2 \cup T_2$ such that:
  1. $H(N_1) = N_2$ and $H(T_1) = T_2$ for the pointwise extension H of h to sets of elements.
  2. $\forall n \in N_1:$
     $h(low_1(n)) = low_2(h(n)) \wedge$
     $h(high_1(n)) = high_2(h(n)) \wedge$
     $var_1(n) = var_2(h(n))$.
  3. $h(root_1) = root_2$.
  4. $\forall t \in T_1 : val_1(t) = val_2(h(t))$.

- A BDD G is reduced iff
  1. there is no node $n \in N$ such that $low(n) = high(n)$ and
  2. there are no two nodes $x_1, x_2 \in N \cup T$ such that the BDDs obtained from G by making x_1 and x_2 the roots and removing their predecessors are isomorphic.

Ordered BDDs

❖ Given some (strict, total) ordering $\prec$ on Var, a BDD G is ordered wrt $\prec$ iff $\forall n \in N$.

1. $low(n) \in N \Longrightarrow var(n) \prec var(low(n))$ and
2. $high(n) \in N \Longrightarrow var(n) \prec var(high(n))$.

❖ Intuitively, in an ordered BDD, the variables encountered in any path from the root are ordered in an ascending way wrt $\prec$.

❖ We abbreviate ordered BDDs as OBDDs and reduced OBDDs as ROBDDs.

Ordered BDDs

- Given some (strict, total) ordering $\prec$ on Var, a BDD G is ordered wrt $\prec$ iff $\forall n \in N$.
  1. $low(n) \in N \Longrightarrow var(n) \prec var(low(n))$ and
  2. $high(n) \in N \Longrightarrow var(n) \prec var(high(n))$.

- Intuitively, in an ordered BDD, the variables encountered in any path from the root are ordered in an ascending way wrt $\prec$.

- We abbreviate ordered BDDs as OBDDs and reduced OBDDs as ROBDDs.

- **Theorem (canonical representation of Boolean functions by BDDs).** For every Boolean function f over some set of variables Var and every variable ordering $\prec$ on Var, there is a unique (up to isomorphism) ROBDD (wrt $\prec$) G_f which represents f.

- **Corollary.** Checking equivalence of the functions represented by two ROBDDs G_1 and G_2 wrt the same ordering $\prec$ amounts to checking isomorphism of G_1 and G_2.

Ordered BDDs

❖ Given some (strict, total) ordering $\prec$ on Var, a BDD G is ordered wrt $\prec$ iff $\forall n \in N.$

1. $low(n) \in N \implies var(n) \prec var(low(n))$ and
2. $high(n) \in N \implies var(n) \prec var(high(n))$.

❖ Intuitively, in an ordered BDD, the variables encountered in any path from the root are ordered in an ascending way wrt $\prec$.

❖ We abbreviate ordered BDDs as OBDDs and reduced OBDDs as ROBDDs.

❖ **Theorem (canonical representation of Boolean functions by BDDs).** For every Boolean function f over some set of variables Var and every variable ordering $\prec$ on Var, there is a unique (up to isomorphism) ROBDD (wrt $\prec$) G_f which represents f.

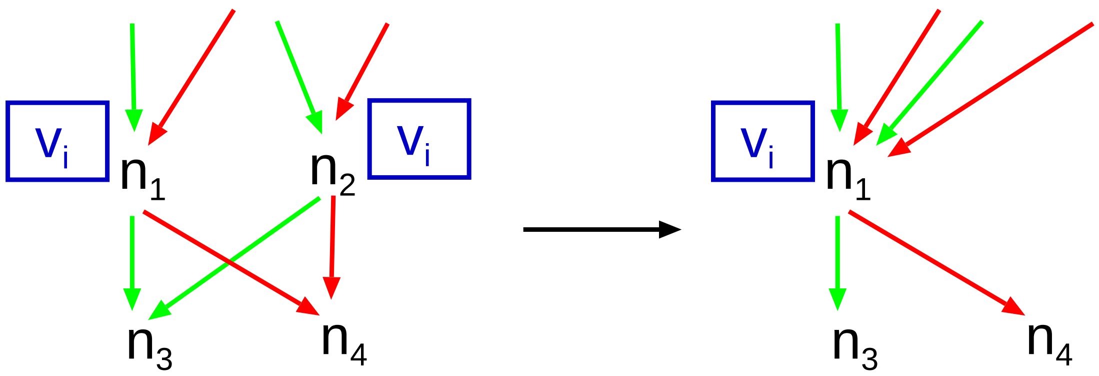
❖ **Corollary.** Checking equivalence of the functions represented by two ROBDDs G_1 and G_2 wrt the same ordering $\prec$ amounts to checking isomorphism of G_1 and G_2.

❖ Moreover, if several Boolean functions are represented by a generalised BDD with multiple roots, the equivalence checking amounts to checking identity of the roots.

Obtaining ROBDDs from OBDDs

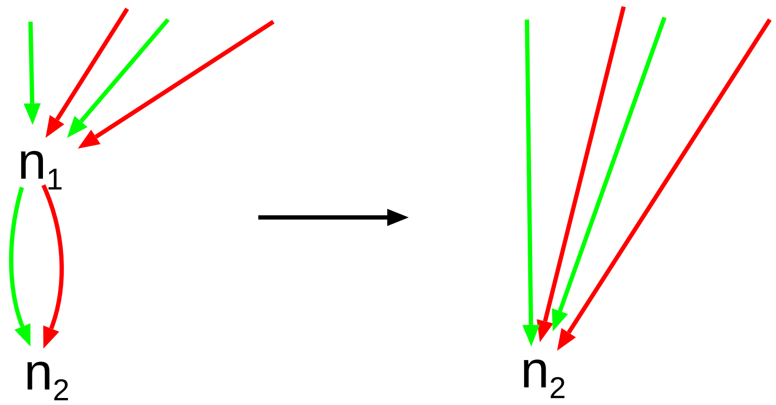
❖ For a fixed ordering $\prec$, the ROBDD can be obtained from an OBDD by a procedure denoted `Reduce` which applies the following three transformation rules until no rule is applicable anymore:

- Rule 1—remove duplicate leaves: merge all equivalued leaves into a single node, which becomes the target of all the edges leading to the merged nodes.
- Rule 2—remove duplicate nonterminals: if there are inner nodes $n_1, n_2 \in N$ such that $n_1 \neq n_2$, but $var(n_1) = var(n_2)$, $low(n_1) = low(n_2)$, and $high(n_1) = high(n_2)$, then merge n_1 and n_2 into a single node being the target of all the edges coming originally into n_1 and n_2.



Obtaining ROBDDs from OBDDs

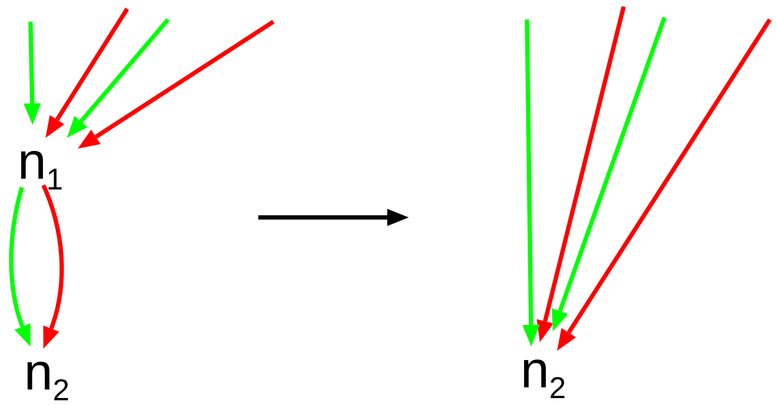
- Rule 3—remove redundant nodes: remove inner nodes $n \in N$ with $low(n) = high(n)$ and redirect all edges coming into n to $low(n)$.



❖ An example: the decision tree from Slide 4 (which is an OBDD but not reduced) can be transformed into the BDD from Slide 4 (which is in fact the appropriate ROBDD).

Obtaining ROBDDs from OBDDs

- Rule 3—remove redundant nodes: remove inner nodes $n \in N$ with $low(n) = high(n)$ and redirect all edges coming into n to $low(n)$.

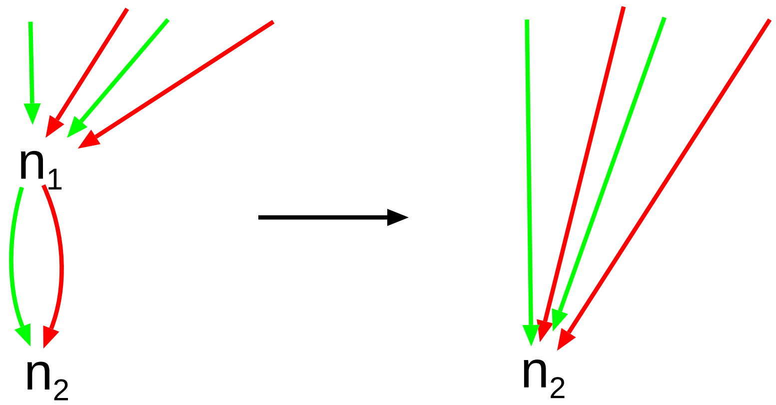


❖ An example: the decision tree from Slide 4 (which is an OBDD but not reduced) can be transformed into the BDD from Slide 4 (which is in fact the appropriate ROBDD).

❖ Constant ROBDDs:

Obtaining ROBDDs from OBDDs

- Rule 3—remove redundant nodes: remove inner nodes $n \in N$ with $low(n) = high(n)$ and redirect all edges coming into n to $low(n)$.



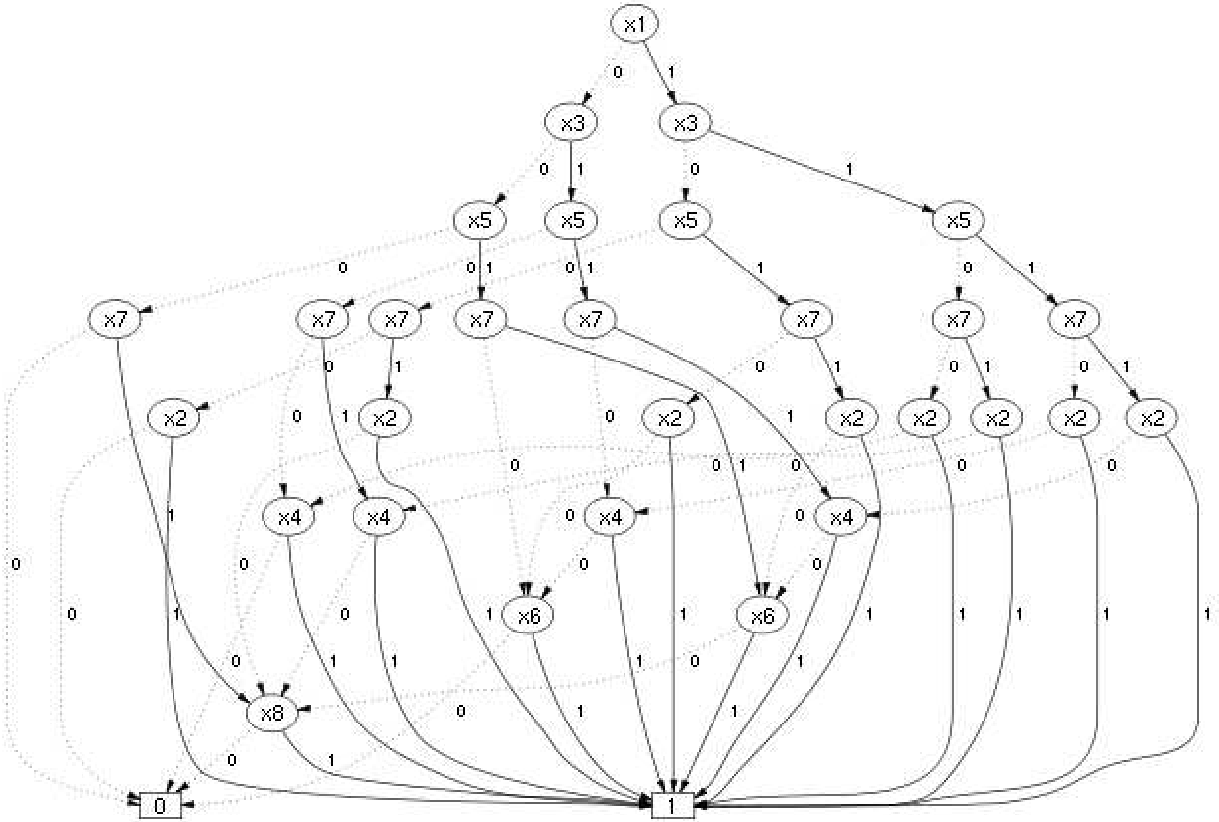
❖ An example: the decision tree from Slide 4 (which is an OBDD but not reduced) can be transformed into the BDD from Slide 4 (which is in fact the appropriate ROBDD).

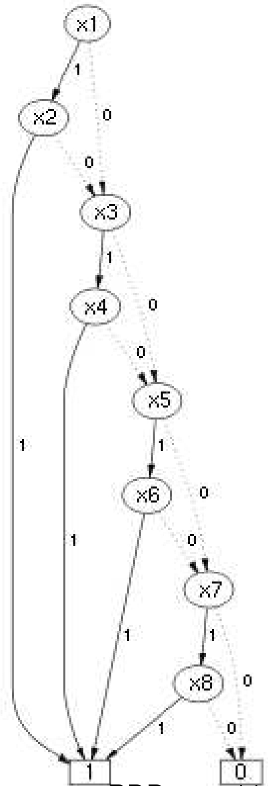
❖ Constant ROBDDs:

- A propositional formula is not satisfiable iff its ROBDD is isomorphic to the “0” ROBDD (a ROBDD consisting of a single 0-valued leaf only).
- A propositional formula is a tautology iff its ROBDD is isomorphic to the “1” ROBDD (a ROBDD consisting of a single 1-valued leaf only).

Variable Ordering

- The size of the ROBDD depends very significantly on the chosen variable ordering.
- For example, for the function $f(x_1, ..., x_{2n}) = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \cdots \vee (x_{2n-1} \wedge x_{2n})$,
  - 2^{n+1} ROBDD nodes are needed when using the variable ordering $x_1 < x_3 < \cdots < x_{2n-1} < x_2 < x_4 < \cdots < x_{2n}$, but
  - $2n + 2$ nodes suffice when using the ordering $x_1 < x_2 < x_3 < x_4 < \cdots < x_{2n-1} < x_{2n}$.





Variable Ordering

- Variable ordering is usually fixed at the beginning and maintained throughout all operations with BDDs.

- Finding an optimal ordering is **NP**-hard.

- Various heuristics may be used, e.g., based on putting close to each other the variables which are in some sense closely related (the value of one is computed from the other one or they are together used as an input of some function, etc.).

- Another possibility is the so-called dynamic reordering:
  - It is started when the size of the ROBDD starts to grow.
  - It is based on moving (one-by-one: the so-called sifting) the individual variables to different positions in the ordering by iteratively re-ordering two successive variables v_i and v_{i+1} via swapping the “0-1” and “1-0” successors of nodes labelled with v_i.
    - *Richard Rudell. Dynamic Variable Ordering for OBDDs. In Proc. of CAD 1993. IEEE CS.*

Operations on ROBDDs

❖ Operations on ROBDDs:

- equivalence checking: isomorphism checking (in $\mathcal{O}(\min(|N_1|, |N_2|))$) or just root (pointer) comparison (in $\mathcal{O}(1)$),
- negation: simply invert the value of leaves (in $\mathcal{O}(1)$),
- binary Boolean operations (16 in total)—via a single function `Apply`:
  - uses restriction, Shannon expansion, and dynamic programming,
  - works in $\mathcal{O}(|N_1| \cdot |N_2|)$ as we shall see.

Operations on ROBDDs

❖ Operations on ROBDDs:

- equivalence checking: isomorphism checking (in $\mathcal{O}(\min(|N_1|, |N_2|))$) or just root (pointer) comparison (in $\mathcal{O}(1)$),
- negation: simply invert the value of leaves (in $\mathcal{O}(1)$),
- binary Boolean operations (16 in total)—via a single function `Apply`:
  - uses restriction, Shannon expansion, and dynamic programming,
  - works in $\mathcal{O}(|N_1| \cdot |N_2|)$ as we shall see.

❖ Restriction of a Boolean function f is a Boolean function obtained by fixing some parameter of f to a given value: $f|_{v_i \leftarrow b}(v_1, \ldots, v_n) = f(v_1, \ldots, v_{i-1}, b, v_{i+1}, \ldots, v_n)$.

- On ROBDDs:
  1. for each node $n \in N$ such that $var(n) = v_i$, redirect all edges leading to n to $low(n)$ if $b = 0$ and to $high(n)$ if $b = 1$, respectively, and remove n,
  2. apply `Reduce` (to obtain a canonical form again).

Shannon Expansion and `Apply`

❖ The Shannon expansion of a Boolean function $f(v_1, \ldots, v_i, \ldots, v_n)$ wrt a variable v_i:

$$f(v_1, \ldots, v_n) = (\neg v_i \wedge f|_{v_i \leftarrow 0}(v_1, \ldots, v_n)) \vee (v_i \wedge f|_{v_i \leftarrow 1}(v_1, \ldots, v_n))$$

❖ Using the Shannon expansion as a basis of the `Apply` function:

- $f \; op \; g = (\neg v \wedge (f|_{v \leftarrow 0} \; op \; g|_{v \leftarrow 0})) \vee (v \wedge (f|_{v \leftarrow 1} \; op \; g|_{v \leftarrow 1}))$.
- For example:
  - $f \wedge g = (\neg v \wedge (f|_{v \leftarrow 0} \wedge g|_{v \leftarrow 0})) \vee (v \wedge (f|_{v \leftarrow 1} \wedge g|_{v \leftarrow 1}))$.
  - $f \vee g = (\neg v \wedge (f|_{v \leftarrow 0} \vee g|_{v \leftarrow 0})) \vee (v \wedge (f|_{v \leftarrow 1} \vee g|_{v \leftarrow 1}))$.

❖ Intuitively, the functions are unfolded into their decision trees on whose leaves the appropriate operation is done.

The `Apply` Function

Function `Apply`

Input: a binary Boolean operator op, ROBDDs G_1, G_2 representing Boolean functions f_1, f_2, respectively, over the same indexed family of variables $\{v_i\}_{i \in I}$ ordered wrt $\prec$.

Output: a ROBDD G representing the Boolean function $f_1 \; op \; f_2$ over $\{v_i\}_{i \in I}$.

Method:

1. Call `ApplyFrom`$(op, G_1, G_2, root_1, root_2)$.
2. Apply `Reduce` on the result of step 1 and return the result.

`ApplyFrom` *(part 1/2)*

Function `ApplyFrom`

Input: a binary Boolean operator op, ROBDDs G_1, G_2 over the same indexed family of variables $\{v_i\}_{i \in I}$ ordered wrt $\prec$, and nodes $x_1 \in N_1 \cup T_1$, $x_2 \in N_2 \cup T_2$.

Output: an OBDD G representing the Boolean function $f_1 \; op \; f_2$ over $\{v_i\}_{i \in I}$ where f_1, f_2 are Boolean functions represented by G_1 and G_2, respectively, when x_1 and x_2 are considered as the roots (and their predecessors are ignored).

Method:

`ApplyFrom` (part 1/2)

Function `ApplyFrom`

Input: a binary Boolean operator op, ROBDDs G_1, G_2 over the same indexed family of variables $\{v_i\}_{i \in I}$ ordered wrt $\prec$, and nodes $x_1 \in N_1 \cup T_1$, $x_2 \in N_2 \cup T_2$.

Output: an OBDD G representing the Boolean function $f_1 \; op \; f_2$ over $\{v_i\}_{i \in I}$ where f_1, f_2 are Boolean functions represented by G_1 and G_2, respectively, when x_1 and x_2 are considered as the roots (and their predecessors are ignored).

Method:

1. If $x_1 \in T_1$ and $x_2 \in T_2$ (i.e., both x_1 and x_2 are leaves), return the ROBDD consisting of a single leaf with the value $val(x_1) \; op \; val(x_2)$.

ApplyFrom (part 1/2)

Function `ApplyFrom`

Input: a binary Boolean operator op, ROBDDs G_1, G_2 over the same indexed family of variables $\{v_i\}_{i \in I}$ ordered wrt $\prec$, and nodes $x_1 \in N_1 \cup T_1$, $x_2 \in N_2 \cup T_2$.

Output: an OBDD G representing the Boolean function $f_1 \ op \ f_2$ over $\{v_i\}_{i \in I}$ where f_1, f_2 are Boolean functions represented by G_1 and G_2, respectively, when x_1 and x_2 are considered as the roots (and their predecessors are ignored).

Method:

1. If $x_1 \in T_1$ and $x_2 \in T_2$ (i.e., both x_1 and x_2 are leaves), return the ROBDD consisting of a single leaf with the value $val(x_1) \ op \ val(x_2)$.
2. Otherwise (at least one of x_1 and x_2 is an inner node):

`ApplyFrom` (part 1/2)

Function `ApplyFrom`

Input: a binary Boolean operator op, ROBDDs G_1, G_2 over the same indexed family of variables $\{v_i\}_{i \in I}$ ordered wrt $\prec$, and nodes $x_1 \in N_1 \cup T_1$, $x_2 \in N_2 \cup T_2$.

Output: an OBDD G representing the Boolean function $f_1 \; op \; f_2$ over $\{v_i\}_{i \in I}$ where f_1, f_2 are Boolean functions represented by G_1 and G_2, respectively, when x_1 and x_2 are considered as the roots (and their predecessors are ignored).

Method:

1. If $x_1 \in T_1$ and $x_2 \in T_2$ (i.e., both x_1 and x_2 are leaves), return the ROBDD consisting of a single leaf with the value $val(x_1) \; op \; val(x_2)$.
2. Otherwise (at least one of x_1 and x_2 is an inner node):
   (a) If $var(x_1) = var(x_2) = v$ for some variable v,
   - let $G'_1 = \texttt{ApplyFrom}(op, G_1, G_2, low_1(x_1), low_2(x_2))$, i.e., compute $f_1|_{v \leftarrow 0} \; op \; f_2|_{v \leftarrow 0}$ using the fact that $f_i|_{v \leftarrow 0} = low_i(x_i)$ for $i \in \{1, 2\}$,

`ApplyFrom` (part 1/2)

Function `ApplyFrom`

Input: a binary Boolean operator op, ROBDDs G_1, G_2 over the same indexed family of variables $\{v_i\}_{i \in I}$ ordered wrt $\prec$, and nodes $x_1 \in N_1 \cup T_1$, $x_2 \in N_2 \cup T_2$.

Output: an OBDD G representing the Boolean function $f_1 \; op \; f_2$ over $\{v_i\}_{i \in I}$ where f_1, f_2 are Boolean functions represented by G_1 and G_2, respectively, when x_1 and x_2 are considered as the roots (and their predecessors are ignored).

Method:

1. If $x_1 \in T_1$ and $x_2 \in T_2$ (i.e., both x_1 and x_2 are leaves), return the ROBDD consisting of a single leaf with the value $val(x_1) \; op \; val(x_2)$.
2. Otherwise (at least one of x_1 and x_2 is an inner node):

   (a) If $var(x_1) = var(x_2) = v$ for some variable v,

   - let $G'_1 = \texttt{ApplyFrom}(op, G_1, G_2, low_1(x_1), low_2(x_2))$, i.e., compute $f_1|_{v \leftarrow 0} \; op \; f_2|_{v \leftarrow 0}$ using the fact that $f_i|_{v \leftarrow 0} = low_i(x_i)$ for $i \in \{1, 2\}$,
   - let $G'_2 = \texttt{ApplyFrom}(op, G_1, G_2, high_1(x_1), high_2(x_2))$,
   - return the OBDD constructed from G'_1 and G'_2 having roots $root'_1$ and $root'_2$, resp., by uniting their sets of terminals and non-terminals (assumed to be disjoint), the var, low, $high$, and val functions, and by adding a new root node n such that $var(n) = v$, $low(n) = root'_1$, and $high(n) = root'_2$.

`ApplyFrom` *(part 2/2)*

Continuation of step 2:

(b) Otherwise, if $var(x_1) = v$ for some variable v and either $x_2 \in T_2$ or $x_2 \in N_2$ and $v \prec var(x_2)$ (meaning that f_2 is independent of v, i.e., $f_2|_{v \leftarrow 0} = f_2|_{v \leftarrow 1} = f_2$),

`ApplyFrom` (part 2/2)

Continuation of step 2:

(b) Otherwise, if $var(x_1) = v$ for some variable v and either $x_2 \in T_2$ or $x_2 \in N_2$ and $v \prec var(x_2)$ (meaning that f_2 is independent of v, i.e., $f_2|_{v \leftarrow 0} = f_2|_{v \leftarrow 1} = f_2$),

- let $G'_1 = \texttt{ApplyFrom}(op, G_1, G_2, low_1(x_1), x_2)$,
- let $G'_2 = \texttt{ApplyFrom}(op, G_1, G_2, high_1(x_1), x_2)$,
- return the OBDD constructed from G'_1 and G'_2 having roots $root'_1$ and $root'_2$, respectively, by uniting their sets of terminals and non-terminals (assumed to be disjoint), the var, low, $high$, and val functions, and by adding a new root node n such that $var(n) = v$, $low(n) = root'_1$, and $high(n) = root'_2$.

`ApplyFrom` (part 2/2)

Continuation of step 2:

(b) Otherwise, if $var(x_1) = v$ for some variable v and either $x_2 \in T_2$ or $x_2 \in N_2$ and $v \prec var(x_2)$ (meaning that f_2 is independent of v, i.e., $f_2|_{v \leftarrow 0} = f_2|_{v \leftarrow 1} = f_2$),

- let $G'_1 = \texttt{ApplyFrom}(op, G_1, G_2, low_1(x_1), x_2)$,
- let $G'_2 = \texttt{ApplyFrom}(op, G_1, G_2, high_1(x_1), x_2)$,
- return the OBDD constructed from G'_1 and G'_2 having roots $root'_1$ and $root'_2$, respectively, by uniting their sets of terminals and non-terminals (assumed to be disjoint), the var, low, $high$, and val functions, and by adding a new root node n such that $var(n) = v$, $low(n) = root'_1$, and $high(n) = root'_2$.

(c) Otherwise $var(x_2) = v$ for some variable v and either $x_1 \in T_1$ or $x_1 \in N_1$ and $v \prec var(x_1)$ and a symmetric step to step 2(b) is taken:

- let $G'_1 = \texttt{ApplyFrom}(op, G_1, G_2, x_1, low_2(x_2))$,
- let $G'_2 = \texttt{ApplyFrom}(op, G_1, G_2, x_1, high_2(x_2))$,
- return the OBDD constructed from G'_1 and G'_2 having roots $root'_1$ and $root'_2$, respectively, by uniting their sets of terminals and non-terminals (assumed to be disjoint), the var, low, $high$, and val functions, and by adding a new root node n such that $var(n) = v$, $low(n) = root'_1$, and $high(n) = root'_2$.

`ApplyFrom`: Further Remarks

❖ An example: use `Apply` over the ROBDDs representing $v_1 \wedge \neg v_2$ and $\neg v_1 \wedge v_2$, respectively, with op being either $\wedge$ or $\vee$.

`ApplyFrom`: Further Remarks

❖ An example: use `Apply` over the ROBDDs representing $v_1 \wedge \neg v_2$ and $\neg v_1 \wedge v_2$, respectively, with op being either $\wedge$ or $\vee$.

❖ Every call to `ApplyFrom` can result in two new calls to `ApplyFrom`: hence exponential complexity!

`ApplyFrom`: Further Remarks

❖ An example: use `Apply` over the ROBDDs representing $v_1 \wedge \neg v_2$ and $\neg v_1 \wedge v_2$, respectively, with op being either $\wedge$ or $\vee$.

❖ Every call to `ApplyFrom` can result in two new calls to `ApplyFrom`: hence exponential complexity!

❖ Use dynamic programming:

- store results of finished invocations of `ApplyFrom` in a hash table together with the appropriate arguments,
- use the hash table to avoid re-computation of `ApplyFrom` over arguments on which it has already been applied.

`ApplyFrom`: Further Remarks

❖ An example: use `Apply` over the ROBDDs representing $v_1 \land \neg v_2$ and $\neg v_1 \land v_2$, respectively, with op being either $\land$ or $\lor$.

❖ Every call to `ApplyFrom` can result in two new calls to `ApplyFrom`: hence exponential complexity!

❖ Use dynamic programming:

- store results of finished invocations of `ApplyFrom` in a hash table together with the appropriate arguments,
- use the hash table to avoid re-computation of `ApplyFrom` over arguments on which it has already been applied.

❖ The number of subgraphs in ROBDDs depends on the number of vertices $V = N \cup T$,

- hence we have $\mathcal{O}(|V_1| \cdot |V_2|)$ ways how to call `ApplyFrom`,
- so the complexity becomes $\mathcal{O}(|V_1| \cdot |V_2|)$.

Other Flavours of Decision Diagrams

❖ BDDs with complemented edges: low-edges can be tagged as complemented, i.e., they represent negation of the subformula

Other Flavours of Decision Diagrams

❖ BDDs with complemented edges: low-edges can be tagged as complemented, i.e., they represent negation of the subformula

❖ MTBDDs (multi-terminal BDDs): represent a function $\{0,1\}^k \to \mathbb{D}$ for an arbitrary domain $\mathbb{D}$

Other Flavours of Decision Diagrams

- BDDs with complemented edges: low-edges can be tagged as complemented, i.e., they represent negation of the subformula

- MTBDDs (multi-terminal BDDs): represent a function $\{0,1\}^k \to \mathbb{D}$ for an arbitrary domain $\mathbb{D}$

- ZDDs (zero-suppressed DDs): a missing node corresponds to a node with the $high$-edge going to 0 (and the low-edge continuing)

Other Flavours of Decision Diagrams

- BDDs with complemented edges: low-edges can be tagged as complemented, i.e., they represent negation of the subformula

- MTBDDs (multi-terminal BDDs): represent a function $\{0,1\}^k \to \mathbb{D}$ for an arbitrary domain $\mathbb{D}$

- ZDDs (zero-suppressed DDs): a missing node corresponds to a node with the $high$-edge going to 0 (and the low-edge continuing)

- TBDDs, CBDDs/CZDDs, ESRBDDs, QMDDs, . . .

Other Flavours of Decision Diagrams

- BDDs with complemented edges: low-edges can be tagged as complemented, i.e., they represent negation of the subformula

- MTBDDs (multi-terminal BDDs): represent a function $\{0,1\}^k \to \mathbb{D}$ for an arbitrary domain $\mathbb{D}$

- ZDDs (zero-suppressed DDs): a missing node corresponds to a node with the $high$-edge going to 0 (and the low-edge continuing)

- TBDDs, CBDDs/CZDDs, ESRBDDs, QMDDs, . . .

- (B)DD packages: BuDDy, CUDD, Sylvan, Adiar, . . .

BDDs in Symbolic Verification

Symbolic Model Checking

- In symbolic model checking, one does not work with individual states, exploring them one by one.

- Instead, (possibly large, sometimes even infinite) sets of states are represented using some formalism and handled at the same time.

- This is, one, e.g., does not compute the successor/predecessor of one state at a time but of all the states in the set, leading to a set of successor/predecessor states.

- The sets of states can be represented as automata, formulae, graphs with summary nodes, BDDs, ...

- One needs to be able to perform transitions on the symbolic representation (unfolding it as little as possible), possibly leading to representing also transitions or Kripke structures symbolically.

Encoding Kripke Structures by BDDs

❖ For symbolic model checking, we need to represent Kripke structures and sets of their states satisfying some formulae using BDDs.

Encoding Kripke Structures by BDDs

❖ For symbolic model checking, we need to represent Kripke structures and sets of their states satisfying some formulae using BDDs.

❖ Hence, we need to use BDDs to encode sets of states and relations on states:

- the labelling function L of Kripke structures can be encoded by encoding separately, for each atomic proposition $p \in AP$, the set of states in which p holds.

Encoding Kripke Structures by BDDs

❖ For symbolic model checking, we need to represent Kripke structures and sets of their states satisfying some formulae using BDDs.

❖ Hence, we need to use BDDs to encode sets of states and relations on states:

- the labelling function L of Kripke structures can be encoded by encoding separately, for each atomic proposition $p \in AP$, the set of states in which p holds.

❖ Having a finite set S of states:

- We may code each state using a binary vector with $\lceil log_2 \ |S| \rceil$ bits.
- An i-th bit may be assigned a Boolean variable v_i and sets of the states may be coded as propositional formulae and hence BDDs:

Encoding Kripke Structures by BDDs

❖ For symbolic model checking, we need to represent Kripke structures and sets of their states satisfying some formulae using BDDs.

❖ Hence, we need to use BDDs to encode sets of states and relations on states:

- the labelling function L of Kripke structures can be encoded by encoding separately, for each atomic proposition $p \in AP$, the set of states in which p holds.

❖ Having a finite set S of states:

- We may code each state using a binary vector with $\lceil log_2 \ |S| \rceil$ bits.
- An i-th bit may be assigned a Boolean variable v_i and sets of the states may be coded as propositional formulae and hence BDDs:
  - For example, for $S = \{s_1, s_2, s_3\}$,
    - we may use 2 bits;
    - encode s_1 as 00, s_2 as 01, s_3 as 10;
    - associate the most-significant bit with v_1, the least-significant bit with v_2;
    - code S as $\neg v_1 \ \vee \ (v_1 \wedge \neg v_2)$; and use the corresponding ROBDD.

Encoding Kripke Structures by BDDs

- For symbolic model checking, we need to represent Kripke structures and sets of their states satisfying some formulae using BDDs.
- Hence, we need to use BDDs to encode sets of states and relations on states:
  - the labelling function L of Kripke structures can be encoded by encoding separately, for each atomic proposition $p \in AP$, the set of states in which p holds.
- Having a finite set S of states:
  - We may code each state using a binary vector with $\lceil log_2\ |S| \rceil$ bits.
  - An i-th bit may be assigned a Boolean variable v_i and sets of the states may be coded as propositional formulae and hence BDDs:
    - For example, for $S = \{s_1, s_2, s_3\}$,
      - we may use 2 bits;
      - encode s_1 as 00, s_2 as 01, s_3 as 10;
      - associate the most-significant bit with v_1, the least-significant bit with v_2;
      - code S as $\neg v_1\ \vee\ (v_1 \wedge \neg v_2)$; and use the corresponding ROBDD.
    - In practice, the encoding schema may reflect the internal structure of states (e.g., if states contain one 8-bit integer encoding a line number, two 8-bit integer variables, and 2 Boolean flags, we may use 26 bits by concatenating the bit representations of all the mentioned state variables).

Encoding Kripke Structures by BDDs

❖ A transition relation $R \subseteq S \times S$ for S coded on n bits, associated with Boolean variables $v_1, \ldots, v_n$, may be coded using $2n$ bits, associated with the Boolean variables $v_1, \ldots, v_n$ and also Boolean variables $v'_1, \ldots, v'_n$ constraining future values of the state variables.

❖ For example, for the set $S = \{s_1, s_2, s_3\}$ and the encoding of s_1 as 00, s_2 as 01, and s_3 as 10 from the previous slide,

- the relation $R = \{(s_1, s_2), (s_1, s_3), (s_2, s_3), (s_3, s_3)\}$ may be encoded as
- $(\neg v_1 \land \neg v_2 \land ((\neg v'_1 \land v'_2) \lor (v'_1 \land \neg v'_2))) \lor$
  $(\neg v_1 \land v_2 \land v'_1 \land \neg v'_2) \lor$
  $(v_1 \land \neg v_2 \land v'_1 \land \neg v'_2)$,
- which can in turn be represented as a ROBDD over 4 variables.

❖ The encoding of the transition relation may again reflect the internal structure of the states and the bitwise implementation of the transitions on the components of states.

CTL Predicate Transformers

❖ Consider a Kripke structure $M = (S, S_0, R, L)$. The meaning of the CTL operators (including atomic formulae viewed as nullary operators) over M can be defined in terms of predicate transformers as follows (for $S', S_1, S_2 \subseteq S$):

$$\begin{aligned}
\tau_p() &= [\![p]\!] \\
\tau_\neg(S') &= S \setminus S' \\
\tau_\vee(S_1, S_2) &= S_1 \cup S_2 \\
\tau_{EX}(S') &= \{s \in S \mid \exists s' \in S'.\ (s, s') \in R\} \\
\tau_{EG}(S') &= \nu Z.\ S' \cap \tau_{EX}(Z) \\
\tau_{E[.U.]}(S_1, S_2) &= \mu Z. S_2 \cup (S_1 \cap \tau_{EX}(Z))
\end{aligned}$$

❖ Going along the syntax tree of a given CTL formula φ from its leaves to the root, the above can be used to compute the sets of states satisfying all subformulae of φ and at last the entire formula φ —this allows one to perform CTL model checking by just checking that S_0 is included in the final computed set.

The CTL Fixpoint Semantics and BDDs

- The operations used within the CTL fixpoint semantics include:
  - set operations on sets of states (like union, intersection, and set complement) which directly map to the corresponding operations on propositional formulae representing the sets wrt. some bit-vector encoding of the states (disjunction, conjunction, negation) and which are easy to implement on BDDs,

The CTL Fixpoint Semantics and BDDs

- The operations used within the CTL fixpoint semantics include:
  - set operations on sets of states (like union, intersection, and set complement) which directly map to the corresponding operations on propositional formulae representing the sets wrt. some bit-vector encoding of the states (disjunction, conjunction, negation) and which are easy to implement on BDDs,
  - fixpoint computations which can be implemented by iteratively applying the appropriate transformers starting from $true$ (νf) or $false$ (μf) till the result stops changing,

The CTL Fixpoint Semantics and BDDs

❖ The operations used within the CTL fixpoint semantics include:

- set operations on sets of states (like union, intersection, and set complement) which directly map to the corresponding operations on propositional formulae representing the sets wrt. some bit-vector encoding of the states (disjunction, conjunction, negation) and which are easy to implement on BDDs,
- fixpoint computations which can be implemented by iteratively applying the appropriate transformers starting from $true$ (νf) or $false$ (μf) till the result stops changing,
- application of the transition relation which maps to a conjunction of the formulae representing a set of states and the relation,

The CTL Fixpoint Semantics and BDDs

- The operations used within the CTL fixpoint semantics include:
  - set operations on sets of states (like union, intersection, and set complement) which directly map to the corresponding operations on propositional formulae representing the sets wrt. some bit-vector encoding of the states (disjunction, conjunction, negation) and which are easy to implement on BDDs,
  - fixpoint computations which can be implemented by iteratively applying the appropriate transformers starting from $true$ (νf) or $false$ (μf) till the result stops changing,
  - application of the transition relation which maps to a conjunction of the formulae representing a set of states and the relation,
  - quantification on Boolean variables (we are dealing with quantified Boolean formulae, abbreviated as QBF)—can be done easily on ROBDDs using restriction and `Apply`:

The CTL Fixpoint Semantics and BDDs

- The operations used within the CTL fixpoint semantics include:
  - set operations on sets of states (like union, intersection, and set complement) which directly map to the corresponding operations on propositional formulae representing the sets wrt. some bit-vector encoding of the states (disjunction, conjunction, negation) and which are easy to implement on BDDs,
  - fixpoint computations which can be implemented by iteratively applying the appropriate transformers starting from $true$ (νf) or $false$ (μf) till the result stops changing,
  - application of the transition relation which maps to a conjunction of the formulae representing a set of states and the relation,
  - quantification on Boolean variables (we are dealing with quantified Boolean formulae, abbreviated as QBF)—can be done easily on ROBDDs using restriction and `Apply`:
    - $\exists v.f \equiv f|_{v\leftarrow 0} \vee f|_{v\leftarrow 1}$,
    - $\forall v.f \equiv f|_{v\leftarrow 0} \wedge f|_{v\leftarrow 1}$.

The CTL Fixpoint Semantics and BDDs

❖ The operations used within the CTL fixpoint semantics include:

- set operations on sets of states (like union, intersection, and set complement) which directly map to the corresponding operations on propositional formulae representing the sets wrt. some bit-vector encoding of the states (disjunction, conjunction, negation) and which are easy to implement on BDDs,
- fixpoint computations which can be implemented by iteratively applying the appropriate transformers starting from $true$ (νf) or $false$ (μf) till the result stops changing,
- application of the transition relation which maps to a conjunction of the formulae representing a set of states and the relation,
- quantification on Boolean variables (we are dealing with quantified Boolean formulae, abbreviated as QBF)—can be done easily on ROBDDs using restriction and `Apply`:
  - $\exists v.f \equiv f|_{v \leftarrow 0} \vee f|_{v \leftarrow 1}$,
  - $\forall v.f \equiv f|_{v \leftarrow 0} \wedge f|_{v \leftarrow 1}$.
- renaming of primed variables to unprimed (after quantification): trivial.