

BDDs and their application in SMT solving

Jan Strejček

Masaryk University
Brno, Czechia

MOVEP 2024

- 1 binary decision diagrams (BDDs)
- 2 theory of bitvectors (BV)
- 3 BDD-based SMT solving of BV

- 1 binary decision diagrams (BDDs)
 - definition
 - operations on BDDs
 - libraries and applications
- 2 theory of bitvectors (BV)

- 3 BDD-based SMT solving of BV

Binary decision diagrams (BDDs)



investigated by **Randal Bryant** since 1986

Binary decision diagrams (BDDs)



investigated by **Randal Bryant** since 1986

*"one of the only really fundamental data structures
that came out in the last twenty-five years"*

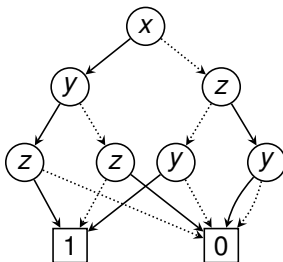
Donald Knuth, 2008



Binary decision diagrams (BDDs)

A **binary decision diagram (BDD)** is a finite rooted directed acyclic graph with two kinds of nodes and two kinds of edges:

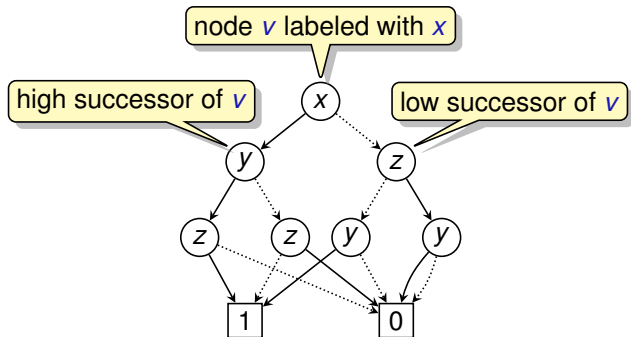
- each **terminal** node is labeled with 0 or 1,
- each **nonterminal** node is labeled with a Boolean variable and has a **low** successor and a **high** successor.



Binary decision diagrams (BDDs)

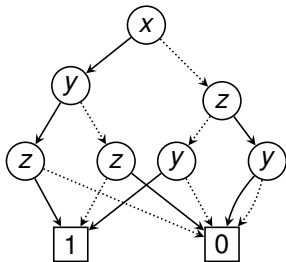
A **binary decision diagram (BDD)** is a finite rooted directed acyclic graph with two kinds of nodes and two kinds of edges:

- each **terminal** node is labeled with 0 or 1,
- each **nonterminal** node is labeled with a Boolean variable and has a **low successor** and a **high successor**.



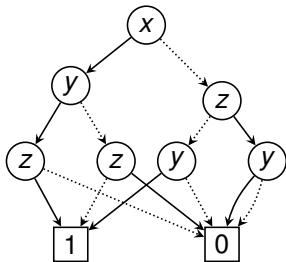
Semantics of BDDs

- a BDD with variables x_1, \dots, x_n describes a Boolean function $f(x_1, \dots, x_n)$
- the value of $f(x_1, \dots, x_n)$ is the value of the terminal node reached from the root by following the high successor whenever the current variable is 1 and the low successor otherwise



Semantics of BDDs

- a BDD with variables x_1, \dots, x_n describes a Boolean function $f(x_1, \dots, x_n)$
- the value of $f(x_1, \dots, x_n)$ is the value of the terminal node reached from the root by following the high successor whenever the current variable is 1 and the low successor otherwise

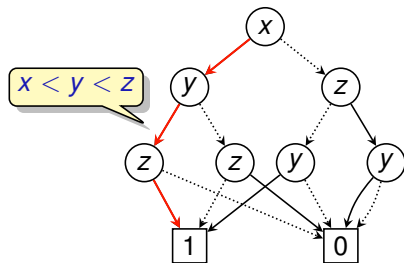


$$f(x, y, z) = \begin{cases} 1 & \text{for } x = 1, y = 1, z = 1 \\ & \text{or } x = 1, y = 0, z = 0 \\ & \text{or } x = 0, z = 0, y = 1 \\ 0 & \text{otherwise} \end{cases}$$

- alternatively, a BDD can represent the set of assignments leading to node 1

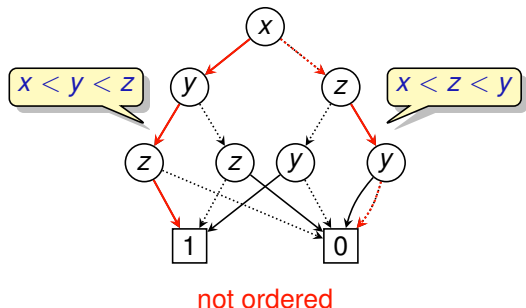
Ordered BDD

A BDD is **ordered** if there exists a linear ordering $<$ on its Boolean variables such that the variables on each path from the root to a terminal node respect the order.



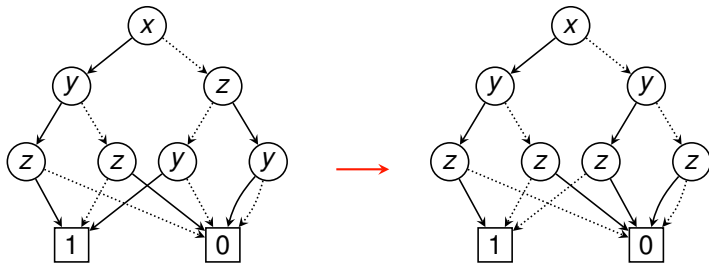
Ordered BDD

A BDD is **ordered** if there exists a linear ordering $<$ on its Boolean variables such that the variables on each path from the root to a terminal node respect the order.



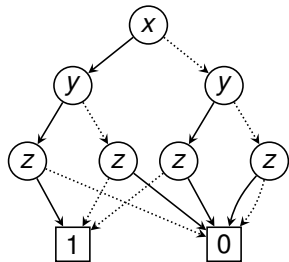
Ordered BDD

A BDD is **ordered** if there exists a linear ordering $<$ on its Boolean variables such that the variables on each path from the root to a terminal node respect the order.



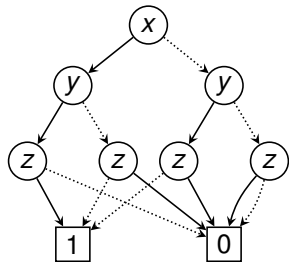
Reduced BDD

A BDD is **reduced** if it does not contain any nonterminal node with identical low and high child and any isomorphic subgraphs.



Reduced BDD

A BDD is **reduced** if it does not contain any nonterminal node with identical low and high child and any isomorphic subgraphs.

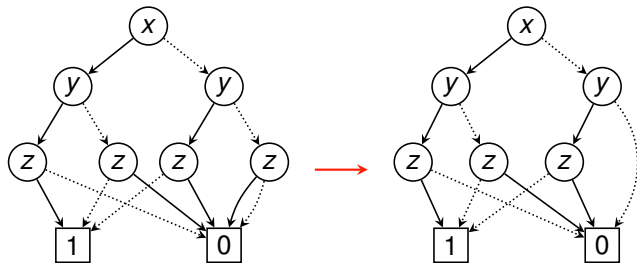


a BDD can be reduced by merging all terminal nodes with the same label and repeated applications of the following steps

- 1 remove each nonterminal node with identical low and high child and redirect all incoming edges to the child
- 2 merge all nodes that have the same low child and the same high child

Reduced BDD

A BDD is **reduced** if it does not contain any nonterminal node with identical low and high child and any isomorphic subgraphs.

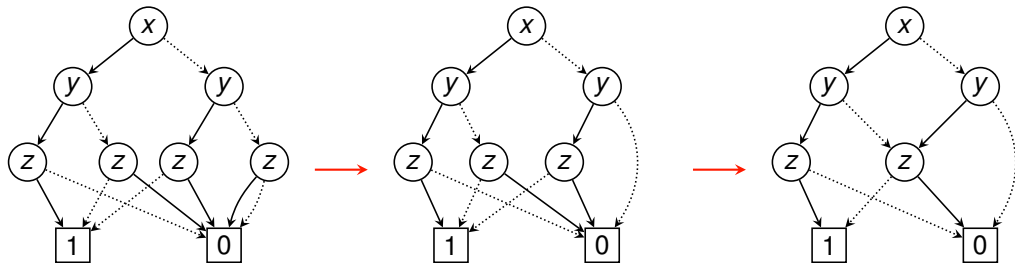


a BDD can be reduced by merging all terminal nodes with the same label and repeated applications of the following steps

- 1 remove each nonterminal node with identical low and high child and redirect all incoming edges to the child
- 2 merge all nodes that have the same low child and the same high child

Reduced BDD

A BDD is **reduced** if it does not contain any nonterminal node with identical low and high child and any isomorphic subgraphs.



a BDD can be reduced by merging all terminal nodes with the same label and repeated applications of the following steps

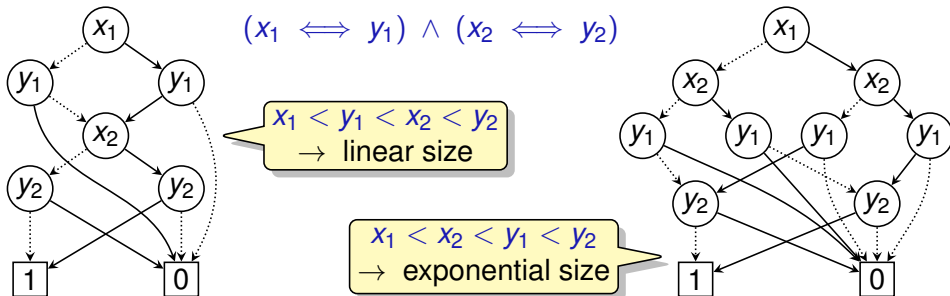
- 1 remove each nonterminal node with identical low and high child and redirect all incoming edges to the child
- 2 merge all nodes that have the same low child and the same high child

Properties of BDDs

- we assume that all BDDs are reduced and ordered
- for a fixed variable order, BDDs are a **canonical representation** of Boolean functions, i.e., two Boolean functions are equivalent (regardless their description) iff the corresponding BDDs are isomorphic

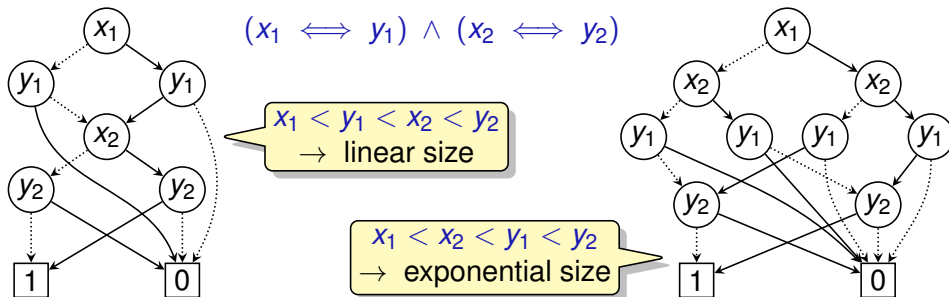
Properties of BDDs

- we assume that all BDDs are reduced and ordered
- for a fixed variable order, BDDs are a **canonical representation** of Boolean functions, i.e., two Boolean functions are equivalent (regardless their description) iff the corresponding BDDs are isomorphic
- BDD size heavily depends on the considered variable order



Properties of BDDs

- we assume that all BDDs are reduced and ordered
- for a fixed variable order, BDDs are a **canonical representation** of Boolean functions, i.e., two Boolean functions are equivalent (regardless their description) iff the corresponding BDDs are isomorphic
- BDD size heavily depends on the considered variable order



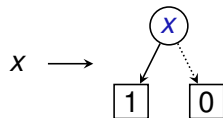
- some BDDs are **exponential** in the number of variables **regardless their order**, e.g., BDDs representing the $\lfloor n/2 \rfloor$ -th bit of the product of two n -bit numbers

Construction of BDDs

BDDs for basic Boolean functions

true \longrightarrow $\boxed{1}$

false \longrightarrow $\boxed{0}$

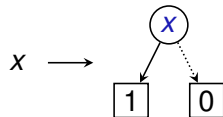


Construction of BDDs

BDDs for basic Boolean functions

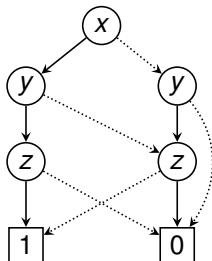
true \rightarrow $\boxed{1}$

false \rightarrow $\boxed{0}$



negation $\neg B$

- replace $\boxed{1}$ with $\boxed{0}$ and vice versa

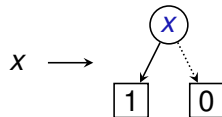


Construction of BDDs

BDDs for basic Boolean functions

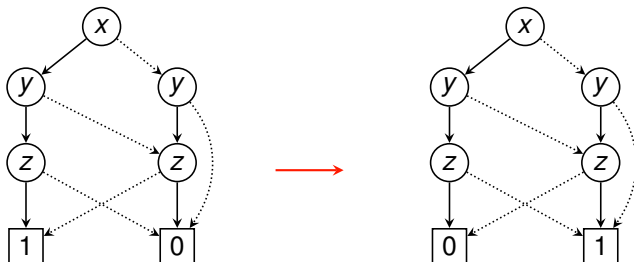
true \longrightarrow $\boxed{1}$

false \longrightarrow $\boxed{0}$



negation $\neg B$

- replace $\boxed{1}$ with $\boxed{0}$ and vice versa



Operations on BDDs

binary operation $B \star D$ for $\star \in \{\wedge, \vee, \text{xor}, \dots\}$

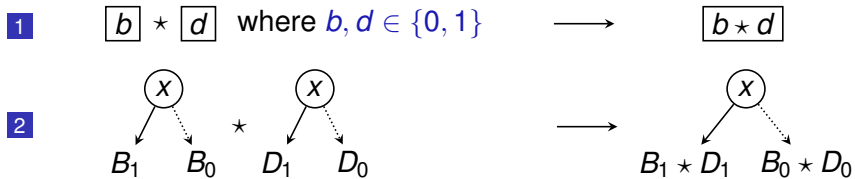
Operations on BDDs

binary operation $B \star D$ for $\star \in \{\wedge, \vee, \text{xor}, \dots\}$

$$1 \quad \boxed{b} \star \boxed{d} \text{ where } b, d \in \{0, 1\} \quad \longrightarrow \quad \boxed{b \star d}$$

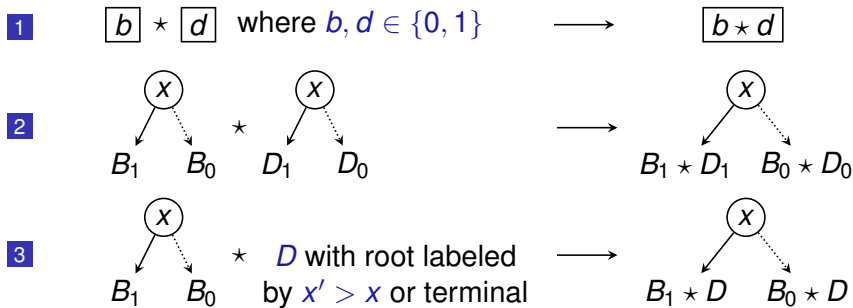
Operations on BDDs

binary operation $B \star D$ for $\star \in \{\wedge, \vee, xor, \dots\}$




Operations on BDDs

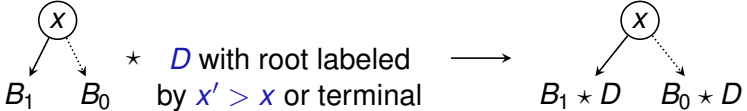
binary operation $B \star D$ for $\star \in \{\wedge, \vee, xor, \dots\}$

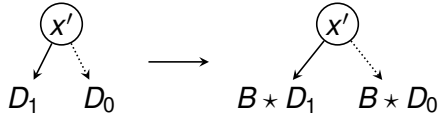


Operations on BDDs

binary operation $B \star D$ for $\star \in \{\wedge, \vee, xor, \dots\}$

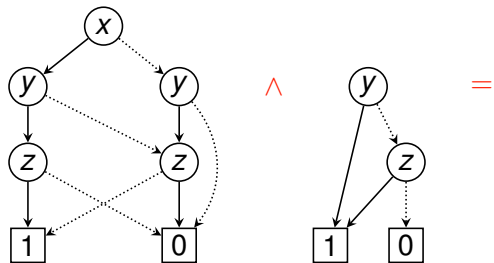
- 1 $\boxed{b} \star \boxed{d}$ where $b, d \in \{0, 1\}$ \longrightarrow $\boxed{b \star d}$
- 2 

$B_1 \star D_1 \quad B_0 \star D_0$
- 3 

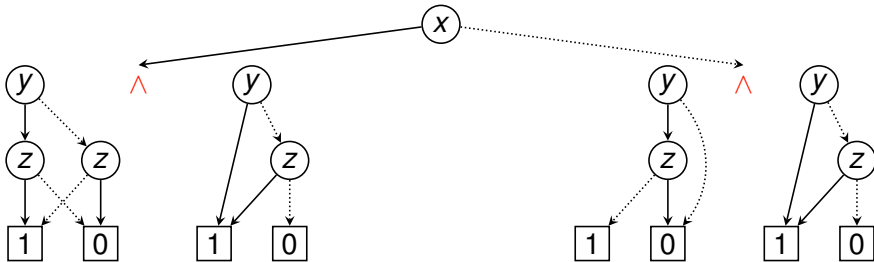
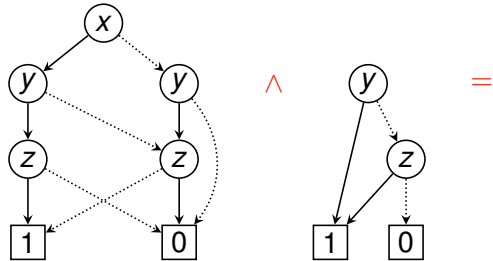
$B_1 \star D \quad B_0 \star D$
- 4 B with root labeled by $x > x'$ or terminal \star 

$B \star D_1 \quad B \star D_0$
- 5 reduce the resulting BDD

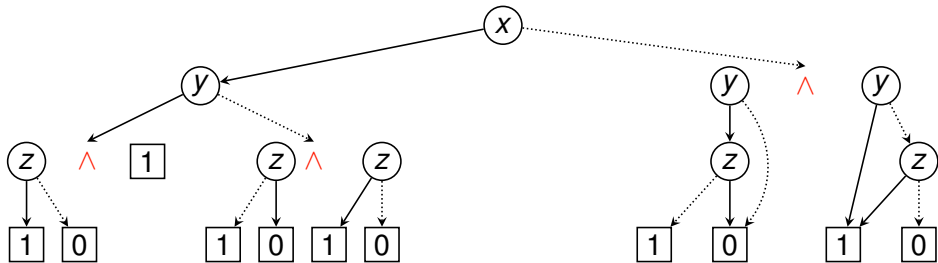
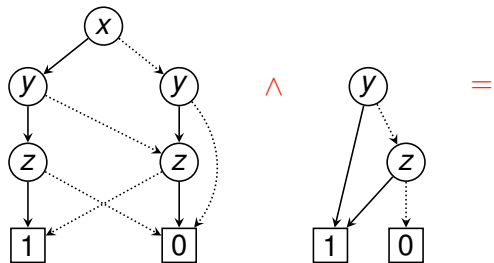
Example



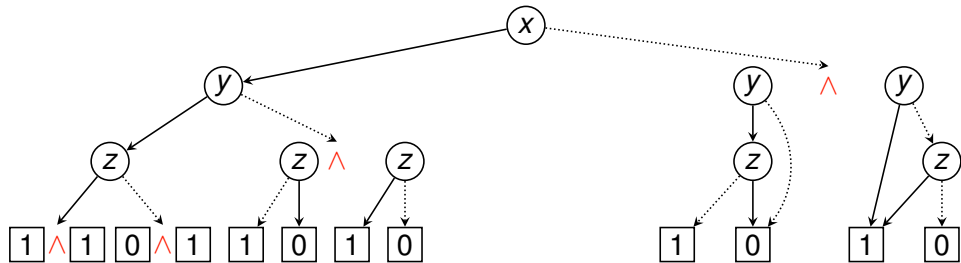
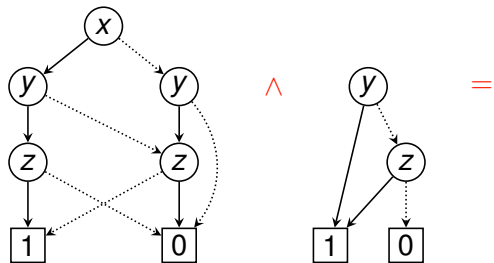
Example



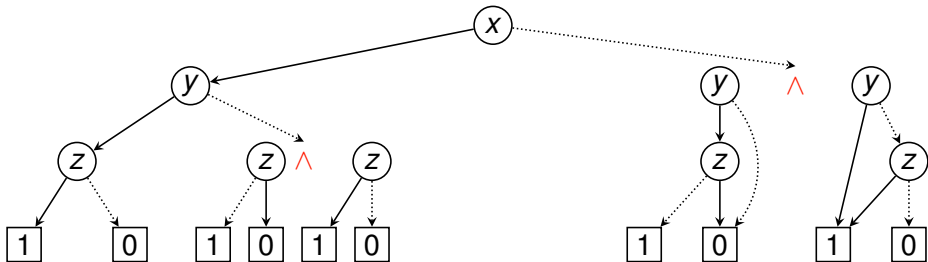
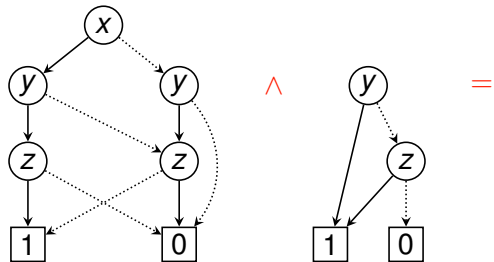
Example



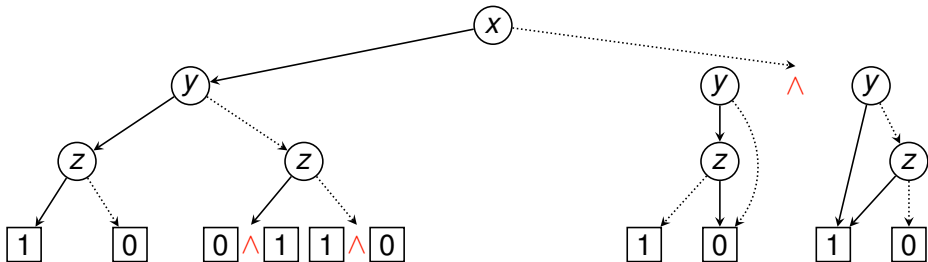
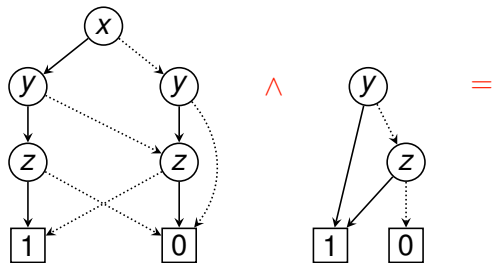
Example



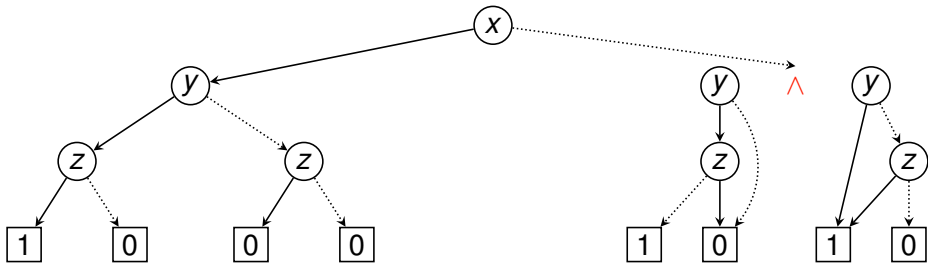
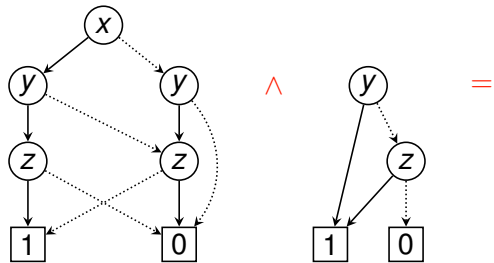
Example



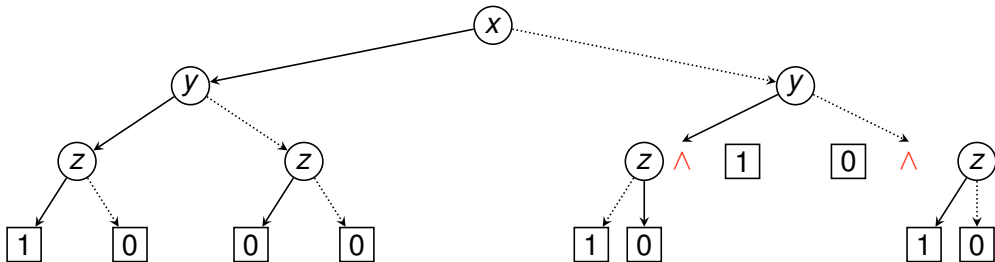
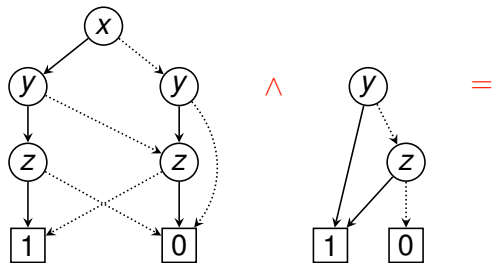
Example



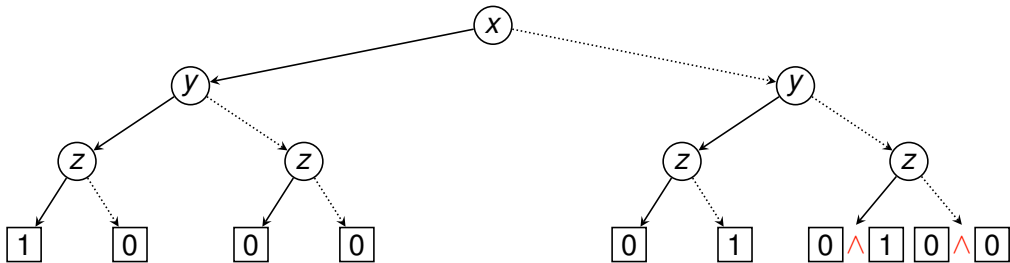
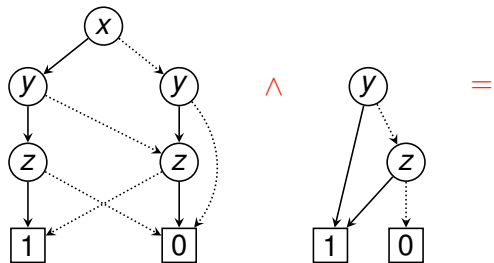
Example



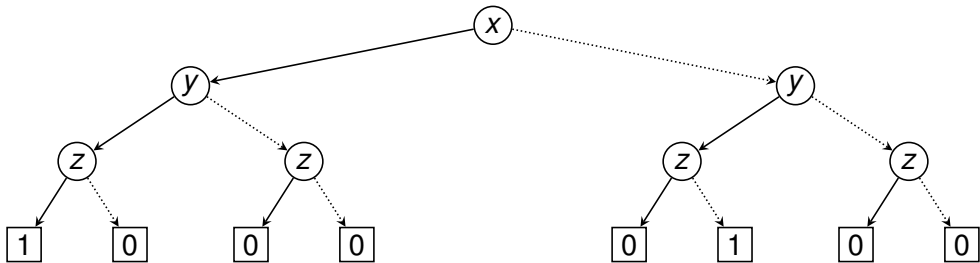
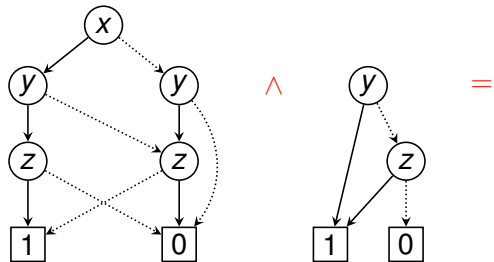
Example



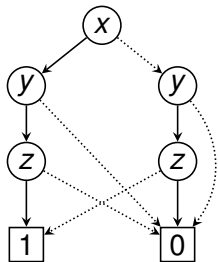
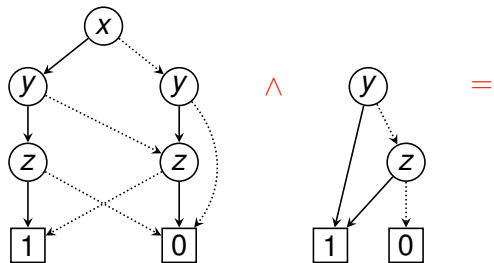
Example



Example



Example



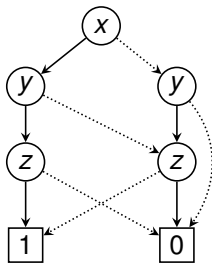
variable instantiation $B[x_i \leftarrow b]$

- 1 if the root is labeled with x_i , then take the high successor as the root if $b = 1$ and the low successor otherwise
- 2 going from top to bottom, each edge leading to a node labeled with x_i is reconnected to its high successor if $b = 1$ and its low successor otherwise
- 3 unreachable nodes are removed and BDD is reduced

Operations on BDDs

variable instantiation $B[x_i \leftarrow b]$

- 1 if the root is labeled with x_i , then take the high successor as the root if $b = 1$ and the low successor otherwise
- 2 going from top to bottom, each edge leading to a node labeled with x_i is reconnected to its high successor if $b = 1$ and its low successor otherwise
- 3 unreachable nodes are removed and BDD is reduced

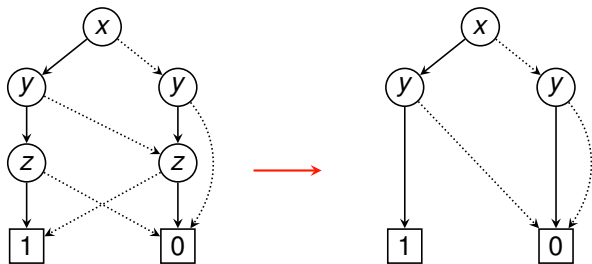


set z to 1

Operations on BDDs

variable instantiation $B[x_i \leftarrow b]$

- 1 if the root is labeled with x_i , then take the high successor as the root if $b = 1$ and the low successor otherwise
- 2 going from top to bottom, each edge leading to a node labeled with x_i is reconnected to its high successor if $b = 1$ and its low successor otherwise
- 3 unreachable nodes are removed and BDD is reduced

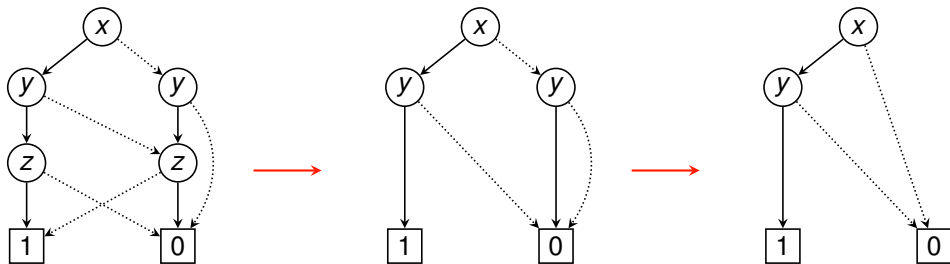


set z to 1

Operations on BDDs

variable instantiation $B[x_i \leftarrow b]$

- 1 if the root is labeled with x_i , then take the high successor as the root if $b = 1$ and the low successor otherwise
- 2 going from top to bottom, each edge leading to a node labeled with x_i is reconnected to its high successor if $b = 1$ and its low successor otherwise
- 3 unreachable nodes are removed and BDD is reduced



set z to 1

Operations on BDDs

quantifiers

$$\blacksquare \exists x.B \equiv B[x \leftarrow 1] \vee B[x \leftarrow 0]$$

$$\blacksquare \forall x.B \equiv B[x \leftarrow 1] \wedge B[x \leftarrow 0]$$

Operations on BDDs

quantifiers

- $\exists x.B \equiv B[x \leftarrow 1] \vee B[x \leftarrow 0]$
- $\forall x.B \equiv B[x \leftarrow 1] \wedge B[x \leftarrow 0]$

complexity and implementation

- all the mentioned operations need only polynomial time when applied to BDDs with the same variable order (and when cache is used)

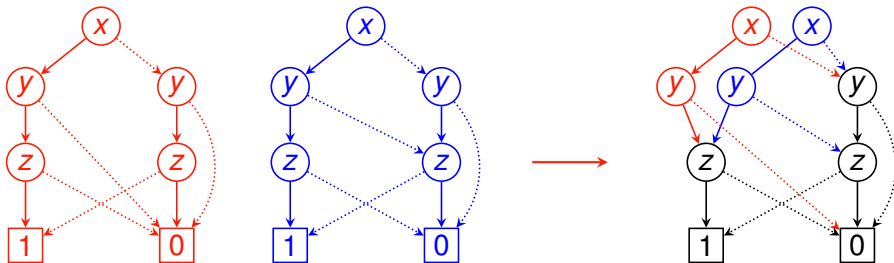
Operations on BDDs

quantifiers

- $\exists x.B \equiv B[x \leftarrow 1] \vee B[x \leftarrow 0]$
- $\forall x.B \equiv B[x \leftarrow 1] \wedge B[x \leftarrow 0]$

complexity and implementation

- all the mentioned operations need only polynomial time when applied to BDDs with the same variable order (and when cache is used)
- in implementations, each node in represented BDDs is stored only once



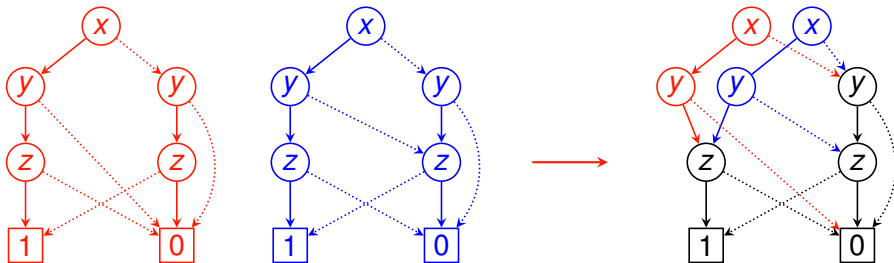
Operations on BDDs

quantifiers

- $\exists x.B \equiv B[x \leftarrow 1] \vee B[x \leftarrow 0]$
- $\forall x.B \equiv B[x \leftarrow 1] \wedge B[x \leftarrow 0]$

complexity and implementation

- all the mentioned operations need only polynomial time when applied to BDDs with the same variable order (and when cache is used)
- in implementations, each node in represented BDDs is stored only once
- equivalence check is constant
- a formula is satisfiable iff the corresponding BDD is not $\boxed{0}$



libraries

- implementations use also negation flags on edges (increases the number of isomorphic subgraphs)
- optimized BDD libraries BuDDy, CUDD, Sylvan, Adiar, . . .
- offer algorithms for automatic improvement of variable order (sifting)
- support of other diagrams, e.g., **zero-suppressed decision diagrams (ZDD)**

libraries

- implementations use also negation flags on edges (increases the number of isomorphic subgraphs)
- optimized BDD libraries BuDDy, CUDD, Sylvan, Adiar, . . .
- offer algorithms for automatic improvement of variable order (sifting)
- support of other diagrams, e.g., **zero-suppressed decision diagrams (ZDD)**

applications

- symbolic model checking
- synthesis of logical circuits
- used in other highly efficient libraries, e.g., in ω -automata library Spot
- SAT and **SMT solving**

- 1 binary decision diagrams (BDDs)
 - definition
 - operations on BDDs
 - libraries and applications
- 2 theory of bitvectors (BV)
 - the theory
 - applications
 - standard approach to SMT solving
- 3 BDD-based SMT solving of BV

SAT solving

- satisfiability of propositional formulae
- Is $((x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge \neg y)) \wedge (z \vee y)$ satisfiable?

SAT solving

- satisfiability of propositional formulae
- Is $((x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge \neg y)) \wedge (z \vee y)$ satisfiable?
- YES: $x = 0, y = 0, z = 1$
- decidable, NP-complete

SAT solving

- satisfiability of propositional formulae
- Is $((x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge \neg y)) \wedge (z \vee y)$ satisfiable?
- YES: $x = 0, y = 0, z = 1$
- decidable, NP-complete

SMT solving

- satisfiability of predicate formulae
- Is $\forall y (y \geq 0 \implies \exists x . y = x \cdot x)$ satisfiable/valid?

SAT solving

- satisfiability of propositional formulae
- Is $((x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge \neg y)) \wedge (z \vee y)$ satisfiable?
- YES: $x = 0, y = 0, z = 1$
- decidable, NP-complete

SMT solving

- satisfiability of predicate formulae
- Is $\forall y (y \geq 0 \implies \exists x . y = x \cdot x)$ satisfiable/valid?
- NO for integers, YES for reals
- **SMT** = satisfiability modulo theory
- decidability and complexity depends on the theory

Theory of fixed-size bitvectors

- **BV** = theory of bitvectors / bitvector logic
- multi-sorted first-order logic
- for each $n > 0$, sort $[n]$ = bitvectors of length n

Theory of fixed-size bitvectors

- **BV** = theory of bitvectors / bitvector logic
- multi-sorted first-order logic
- for each $n > 0$, sort $[n]$ = bitvectors of length n
- functions
 - constants: $0_{[n]}, 1_{[n]}, \dots$
 - bit-wise logic operations: $\text{not}_{[n]}, \text{and}_{[n]}, \text{or}_{[n]}$
 - arithmetic operations with overflows: $+_{[n]}, *_{[n]}, /_{[n]}^s, /_{[n]}^u \dots$
 - bit-wise left/right shift: $\text{shl}_{[n]}, \text{shr}_{[n]}$
 - concatenation: $\text{concat}_{[m,n]}$
 - extraction: $\text{extract}_{[m,i,j]}$
 - [uninterpreted functions]

Theory of fixed-size bitvectors

- **BV** = theory of bitvectors / bitvector logic
- multi-sorted first-order logic
- for each $n > 0$, sort $[n]$ = bitvectors of length n

- functions
 - constants: $0_{[n]}, 1_{[n]}, \dots$
 - bit-wise logic operations: $\text{not}_{[n]}, \text{and}_{[n]}, \text{or}_{[n]}$
 - arithmetic operations with overflows: $+_{[n]}, *_{[n]}, /_{[n]}^s, /_{[n]}^u \dots$
 - bit-wise left/right shift: $\text{shl}_{[n]}, \text{shr}_{[n]}$
 - concatenation: $\text{concat}_{[m,n]}$
 - extraction: $\text{extract}_{[m,i,j]}$
 - [uninterpreted functions]

- predicates
 - equality: $=_{[n]}$
 - signed/unsigned less than (or equal): $<_{[n]}^s, <_{[n]}^u, \leq_{[n]}^s, \leq_{[n]}^u$

Applications of the theory

```
unsigned char x = input();  
if (x > 100) {  
    unsigned char y = 4 * x + 1;  
    assert(x < y);  
}
```

the assertion can be violated \iff

$x_{[8]} >_{[8]}^u 100_{[8]} \wedge \neg(x_{[8]} <_{[8]}^u 4_{[8]} *_{[8]} x_{[8]} +_{[8]} 1_{[8]})$ is satisfiable

Applications of the theory

```
unsigned char x = input();  
if (x > 100) {  
    unsigned char y = 4 * x + 1;  
    assert(x < y);  
}
```

the assertion can be violated \iff

$x >^u 100 \wedge \neg(x <^u 4 * x + 1)$ is satisfiable

```
unsigned char x = input();  
if (x > 100) {  
    unsigned char y = 4 * x + 1;  
    assert(x < y);  
}
```

the assertion can be violated \iff

$x >^u 100 \wedge \neg(x <^u 4 * x + 1)$ is satisfiable

- **QF_BV** = quantifier-free fragment of BV
- verification of safety properties
- automatic test generation
- ...

Applications of the theory

```
unsigned char x = input();  
while (x != 0) {  
    x = x - 3;  
}
```

the program always terminates \iff

$\forall x_{[8]} \exists y_{[8]} . x_{[8]} -_{[8]} 3_{[8]} *_{[8]} y_{[8]} =_{[8]} 0_{[8]}$ is satisfiable/valid

Applications of the theory

```
unsigned char x = input();  
while (x != 0) {  
    x = x - 3;  
}
```

the program always terminates \iff

$\forall x \exists y . x - 3 * y = 0$ is satisfiable/valid


```
unsigned char x = input();  
while (x != 0) {  
    x = x - 3;  
}
```

the program always terminates \iff

$\forall x \exists y . x - 3 * y = 0$ is satisfiable/valid

- proving termination
- computation and application of loop summaries
- program synthesis
- ...

Complexity of SMT solving of BV

	uninterpreted functions	encoding of bitvector lengths	
		unary	binary
QF_BV	no	NP	NEXPTIME
	yes	NP	NEXPTIME
BV	no	PSPACE	AEXP(poly)
	yes	NEXPTIME	2-NEXPTIME

- **-complete** in all cases
- **AEXP(poly)** = problems solvable by alternating Turing machines with polynomial number of alternations in exponential time
- **NEXPTIME** \subseteq **AEXP(poly)** \subseteq **EXPSpace**

- bit-blasting and SAT solving
- each bitvector variable $x_{[n]}$ can be seen as a sequence of n Boolean variables $x_{n-1}x_{n-2} \dots x_1x_0$
- bitvector functions and relations can be transformed into Boolean operations

$$x_{[2]} +_{[2]} y_{[2]} =_{[2]} 1_{[2]}$$



$$"x_1x_0 + y_1y_0 = 01"$$

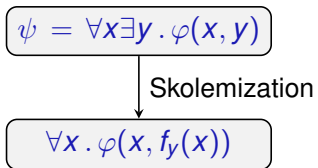


$$(x_0 \iff \neg y_0) \wedge (x_1 \iff y_1)$$

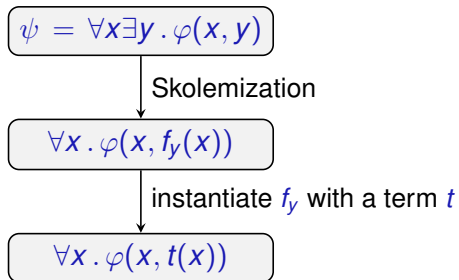
quantifier instantiation

$$\psi = \forall x \exists y . \varphi(x, y)$$

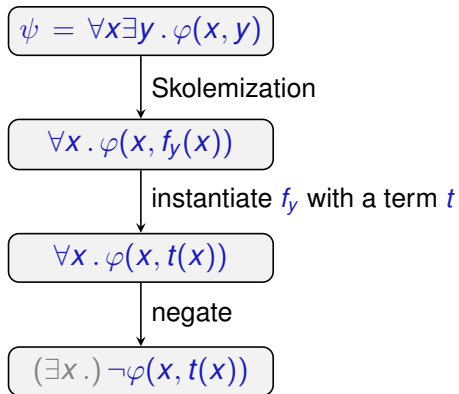
quantifier instantiation



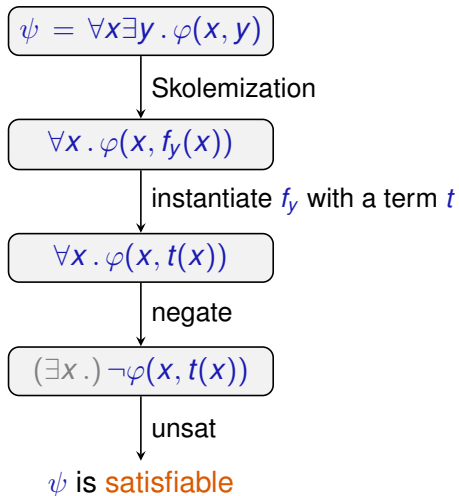
quantifier instantiation



quantifier instantiation

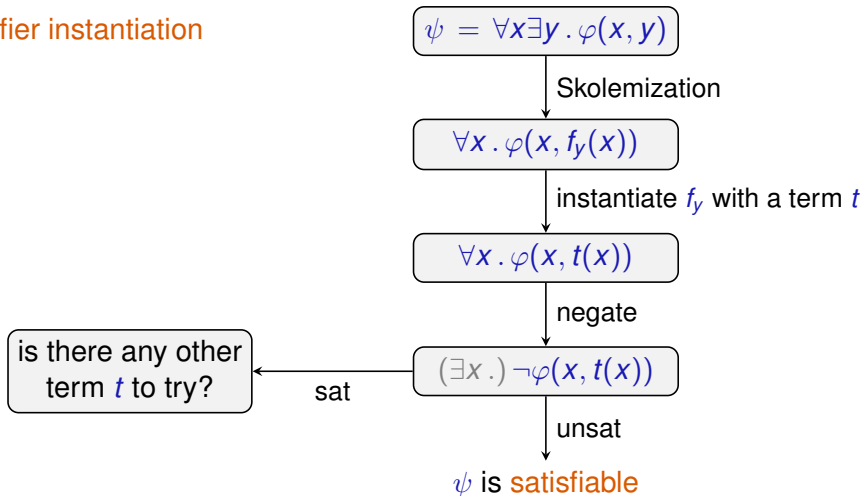


quantifier instantiation



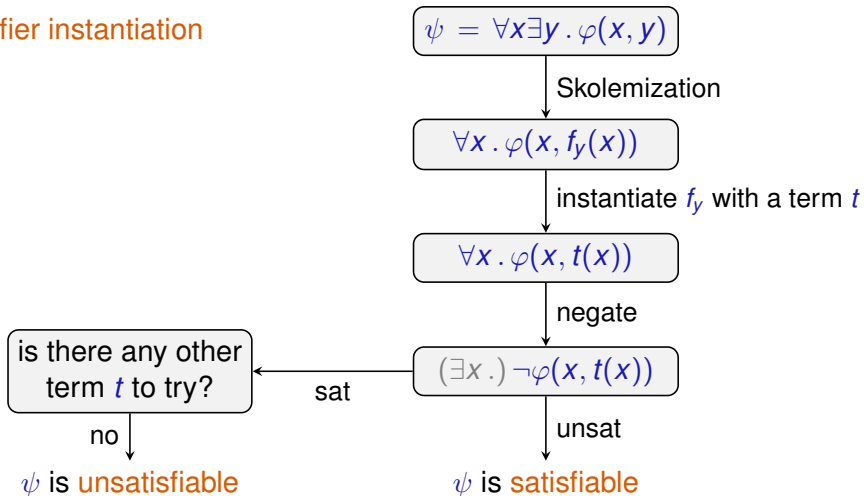
Traditional approach to SMT solving of BV

quantifier instantiation



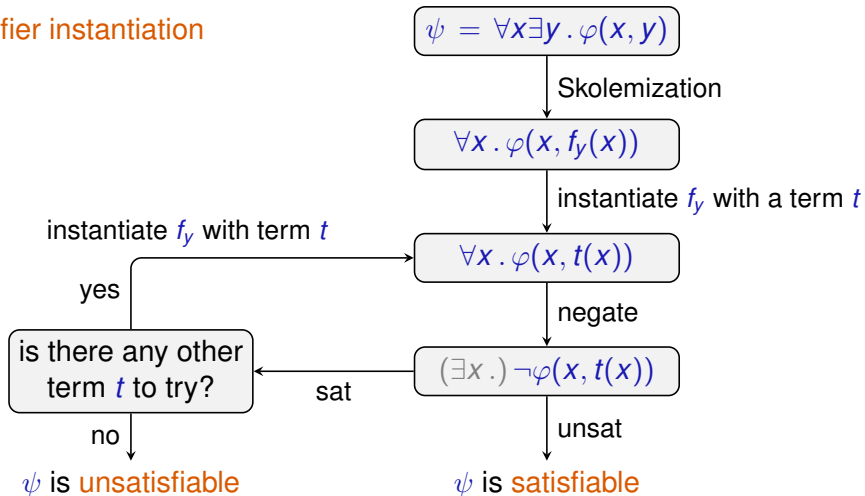
Traditional approach to SMT solving of BV

quantifier instantiation



Traditional approach to SMT solving of BV

quantifier instantiation



- 1 binary decision diagrams (BDDs)
 - definition
 - operations on BDDs
 - libraries and applications
- 2 theory of bitvectors (BV)
 - the theory
 - applications
 - standard approach to SMT solving
- 3 **BDD-based SMT solving of BV**
 - naïve algorithm
 - algorithm improvements
 - Q3B
 - based on joint work with **Martin Jonáš**
[SAT'16, SAT'17, ICTAC'18, IPL'18, CAV'19]



- 1 translate BV formula φ to BDD B_φ representing its models
- 2 check if B_φ represents some model, i.e., $B_\varphi \neq \boxed{0}$
 - YES: φ is **satisfiable**
 - NO: φ is **unsatisfiable**

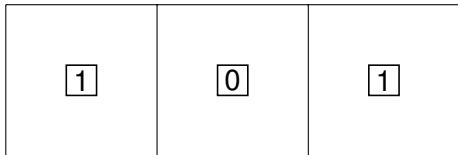
BV formula to BDD bottom-up translation

- a translate each term of type $[n]$ into a **vector** of n of BDDs that represents the function given by the term

BV formula to BDD bottom-up translation

- a translate each term of type $[n]$ into a **vector** of n of BDDs that represents the function given by the term

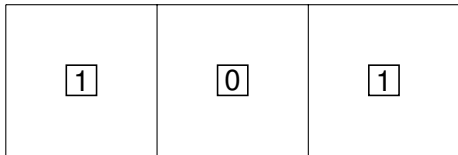
$5_{[3]}$
"101"



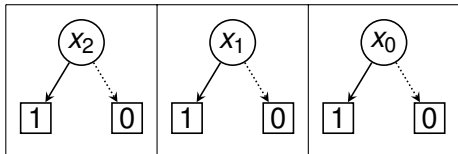
BV formula to BDD bottom-up translation

- a translate each term of type $[n]$ into a **vector** of n of BDDs that represents the function given by the term

$5_{[3]}$
"101"



$x_{[3]}$
" $x_2 x_1 x_0$ "

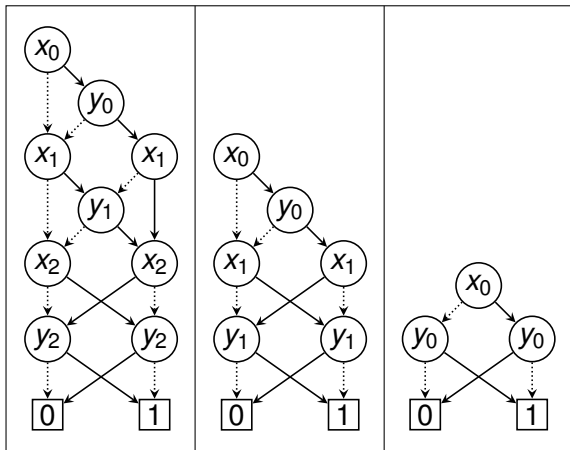


BV formula to BDD bottom-up translation

- a translate each term of type $[n]$ into a **vector** of n of BDDs that represents the function given by the term

$$x_{[3]} +_{[3]} y_{[3]}$$

“ $x_2x_1x_0 + y_2y_1y_0$ ”

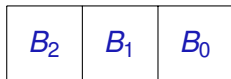
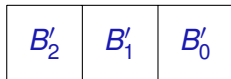


BV formula to BDD bottom-up translation

- b translate each atomic subformula to a **single** BDD representing its models

BV formula to BDD bottom-up translation

- b translate each atomic subformula to a **single** BDD representing its models

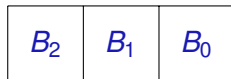
 $t_{[3]}$  $t'_{[3]}$ 

$$t_{[3]} =_{[3]} t'_{[3]}$$

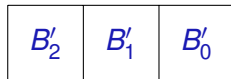
BV formula to BDD bottom-up translation

- b translate each atomic subformula to a **single** BDD representing its models

$t_{[3]}$



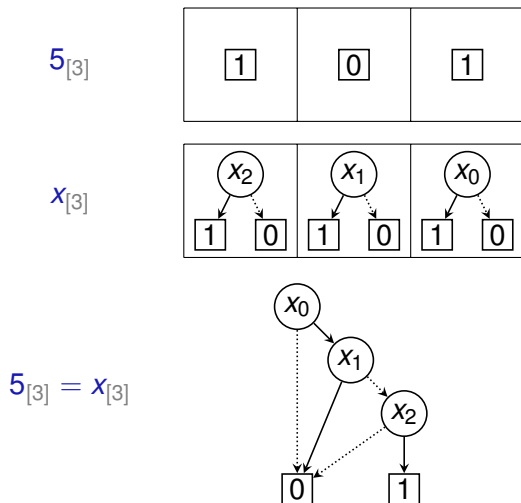
$t'_{[3]}$



$$t_{[3]} =_{[3]} t'_{[3]} \quad (B_2 \iff B'_2) \wedge (B_1 \iff B'_1) \wedge (B_0 \iff B'_0)$$

BV formula to BDD bottom-up translation

- b translate each atomic subformula to a **single** BDD representing its models

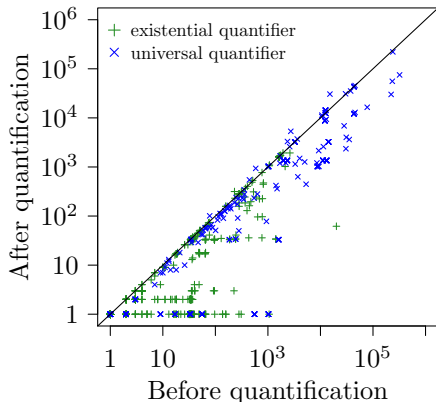


- c apply logical connectives and quantifiers of the formula to BDDs corresponding to subformulae

$\forall x_{[3]} . \psi$ process as $\forall x_2 \forall x_1 \forall x_0 . B_\psi$

Observations

- the algorithm works well as long as the constructed BDDs are small
- quantification of variables usually reduces the BDD size



sizes of BDDs corresponding to all quantified subformulas in SMT-LIB benchmarks for BV logic

GOAL: reduce the size of constructed BDDs

- modification of the input formula
 - move quantifiers downwards
 - eliminate variables or lower their bitwidth
 - simplify the formula
- approximations using less Boolean variables
- abstractions of bitvector operations with BDDs of limited size

Algorithm improvements: modifications of input formula

- push quantifiers in the formula downwards (miniscoping)

$$\begin{array}{ll} \forall x. \varphi(x) \vee \psi \rightsquigarrow \forall x (\varphi(x)) \vee \psi & \forall x. \varphi(x) \wedge \psi(x) \rightsquigarrow \forall x (\varphi(x)) \wedge \forall x (\psi(x)) \\ \exists x. \varphi(x) \wedge \psi \rightsquigarrow \exists x (\varphi(x)) \wedge \psi & \exists x. \varphi(x) \vee \psi(x) \rightsquigarrow \exists x (\varphi(x)) \vee \exists x (\psi(x)) \end{array}$$

Algorithm improvements: modifications of input formula

- push quantifiers in the formula downwards (miniscoping)

$$\begin{array}{ll} \forall x. \varphi(x) \vee \psi \rightsquigarrow \forall x (\varphi(x)) \vee \psi & \forall x. \varphi(x) \wedge \psi(x) \rightsquigarrow \forall x (\varphi(x)) \wedge \forall x (\psi(x)) \\ \exists x. \varphi(x) \wedge \psi \rightsquigarrow \exists x (\varphi(x)) \wedge \psi & \exists x. \varphi(x) \vee \psi(x) \rightsquigarrow \exists x (\varphi(x)) \vee \exists x (\psi(x)) \end{array}$$

- eliminate variables in the formula by equality resolution

$$\forall x. \neg(x = t) \vee \varphi(x) \rightsquigarrow \varphi(t) \qquad \exists x. x = t \wedge \varphi(x) \rightsquigarrow \varphi(t)$$

Algorithm improvements: modifications of input formula

- push quantifiers in the formula downwards (miniscoping)

$$\begin{aligned}\forall x. \varphi(x) \vee \psi &\rightsquigarrow \forall x (\varphi(x)) \vee \psi & \forall x. \varphi(x) \wedge \psi(x) &\rightsquigarrow \forall x (\varphi(x)) \wedge \forall x (\psi(x)) \\ \exists x. \varphi(x) \wedge \psi &\rightsquigarrow \exists x (\varphi(x)) \wedge \psi & \exists x. \varphi(x) \vee \psi(x) &\rightsquigarrow \exists x (\varphi(x)) \vee \exists x (\psi(x))\end{aligned}$$

- eliminate variables in the formula by equality resolution

$$\forall x. \neg(x = t) \vee \varphi(x) \rightsquigarrow \varphi(t) \qquad \exists x. x = t \wedge \varphi(x) \rightsquigarrow \varphi(t)$$

- simplify the formula with unconstrained or partly constrained terms

- **unconstrained term** = term that can have an arbitrary value
- let u be a free or existentially quantified unconstrained variable
- let v be a fresh variable

$$u + 5 * (y + z) \rightsquigarrow v \qquad u_{[n]} *_{[n]} 4_{[n]} \rightsquigarrow \text{concat}_{[n-2,2]}(v_{[n-2]}, 0_{[2]})$$

Algorithm improvements: modifications of input formula

- push quantifiers in the formula downwards (miniscoping)

$$\begin{aligned}\forall x. \varphi(x) \vee \psi &\rightsquigarrow \forall x (\varphi(x)) \vee \psi & \forall x. \varphi(x) \wedge \psi(x) &\rightsquigarrow \forall x (\varphi(x)) \wedge \forall x (\psi(x)) \\ \exists x. \varphi(x) \wedge \psi &\rightsquigarrow \exists x (\varphi(x)) \wedge \psi & \exists x. \varphi(x) \vee \psi(x) &\rightsquigarrow \exists x (\varphi(x)) \vee \exists x (\psi(x))\end{aligned}$$

- eliminate variables in the formula by equality resolution

$$\forall x. \neg(x = t) \vee \varphi(x) \rightsquigarrow \varphi(t) \qquad \exists x. x = t \wedge \varphi(x) \rightsquigarrow \varphi(t)$$

- simplify the formula with unconstrained or partly constrained terms

- **unconstrained term** = term that can have an arbitrary value
- let u be a free or existentially quantified unconstrained variable
- let v be a fresh variable

$$u + 5 * (y + z) \rightsquigarrow v \qquad u_{[n]} *_{[n]} 4_{[n]} \rightsquigarrow \text{concat}_{[n-2,2]}(v_{[n-2]}, 0_{[2]})$$

- let u be universally quantified unconstrained variable

$$t <^u u \rightsquigarrow \text{false} \qquad t \leq^u u \rightsquigarrow t = 0$$

Algorithm improvements: approximations

- approximate bitvector variables by using less Boolean variables
- reduce the number of propositional variables representing $x_{[n]}$

$$\begin{array}{l} x_{n-1}x_{n-2} \dots x_3x_2x_1x_0 \rightsquigarrow 00 \dots 0x_2x_1x_0 \quad \text{zero-extension} \\ x_{n-1}x_{n-2} \dots x_3x_2x_1x_0 \rightsquigarrow x_2x_2 \dots x_2x_2x_1x_0 \quad \text{sign-extension} \end{array}$$

Algorithm improvements: approximations

- approximate bitvector variables by using less Boolean variables
- reduce the number of propositional variables representing $x_{[n]}$

$$\begin{aligned}x_{n-1}x_{n-2} \dots x_3x_2x_1x_0 &\rightsquigarrow 00 \dots 0x_2x_1x_0 && \text{zero-extension} \\x_{n-1}x_{n-2} \dots x_3x_2x_1x_0 &\rightsquigarrow x_2x_2 \dots x_2x_2x_1x_0 && \text{sign-extension}\end{aligned}$$

- **underapproximation**

$\underline{\varphi}$ = formula φ with reduced existentially quantified variables

$\underline{\varphi}$ is satisfiable $\implies \varphi$ is satisfiable

Algorithm improvements: approximations

- approximate bitvector variables by using less Boolean variables
- reduce the number of propositional variables representing $x_{[n]}$

$$\begin{aligned}x_{n-1}x_{n-2} \dots x_3x_2x_1x_0 &\rightsquigarrow 00 \dots 0x_2x_1x_0 && \text{zero-extension} \\x_{n-1}x_{n-2} \dots x_3x_2x_1x_0 &\rightsquigarrow x_2x_2 \dots x_2x_2x_1x_0 && \text{sign-extension}\end{aligned}$$

- **underapproximation**

$\underline{\varphi}$ = formula φ with reduced existentially quantified variables

$$\underline{\varphi} \text{ is satisfiable} \implies \varphi \text{ is satisfiable}$$

- **overapproximation**

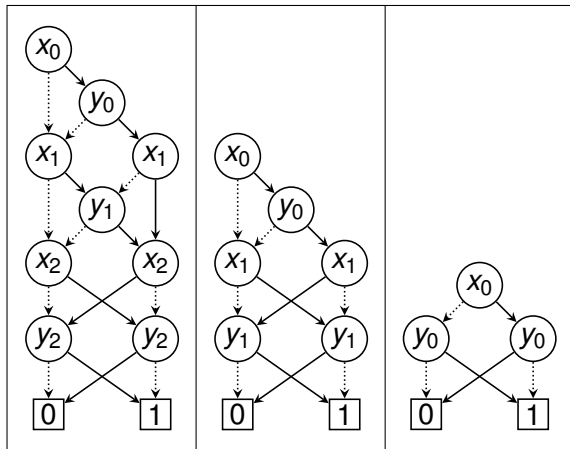
$\overline{\varphi}$ = formula φ with reduced universally quantified variables

$$\overline{\varphi} \text{ is unsatisfiable} \implies \varphi \text{ is unsatisfiable}$$

Algorithm improvements: abstractions of bitvector operations

- satisfiability can be sometimes decided without costly computations
- abstract bitvector operations by computing only BDDs under a given node limit

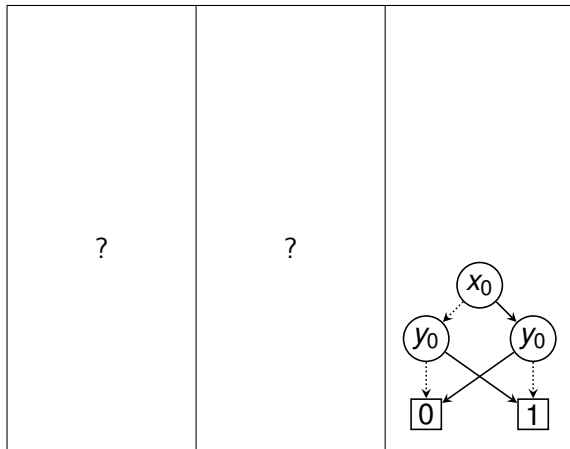
$$x_{[3]} +_{[3]} y_{[3]}$$



Algorithm improvements: abstractions of bitvector operations

- satisfiability can be sometimes decided without costly computations
- abstract bitvector operations by computing only BDDs under a given node limit

$$x_{[3]} +_{[3]} y_{[3]}$$



Algorithm improvements: abstractions of bitvector operations

- satisfiability can be sometimes decided without costly computations
- abstract bitvector operations by computing only BDDs under a given node limit
 - adopt BDD operations to handle ? correctly

$$? \wedge B = \begin{cases} \boxed{0} & \text{if } B = \boxed{0} \\ ? & \text{otherwise} \end{cases}$$

Algorithm improvements: abstractions of bitvector operations

- satisfiability can be sometimes decided without costly computations
- abstract bitvector operations by computing only BDDs under a given node limit
 - two modes for evaluation of atomic subformulae

$t_{[3]}$

B_2	B_1	B_0
-------	-------	-------

$t'_{[3]}$

B'_2	B'_1	B'_0
--------	--------	--------

$$t_{[3]} =_{[3]} t'_{[3]} \quad (B_2 \iff B'_2) \wedge (B_1 \iff B'_1) \wedge (B_0 \iff B'_0)$$

Algorithm improvements: abstractions of bitvector operations

- satisfiability can be sometimes decided without costly computations
- abstract bitvector operations by computing only BDDs under a given node limit
 - two modes for evaluation of atomic subformulae

$t_{[3]}$

?	?	B_0
---	---	-------

$t'_{[3]}$

B'_2	B'_1	B'_0
--------	--------	--------

$$t_{[3]} =_{[3]} t'_{[3]}$$

optimist

pesimist

$$(B_2 \iff B'_2) \wedge (B_1 \iff B'_1) \wedge (B_0 \iff B'_0)$$
$$\begin{array}{lll} true & \wedge & true \quad \wedge \quad (B_0 \iff B'_0) \\ false & \wedge & false \quad \wedge \quad (B_0 \iff B'_0) \end{array}$$

Algorithm improvements: abstractions of bitvector operations

- satisfiability can be sometimes decided without costly computations
- abstract bitvector operations by computing only BDDs under a given node limit
 - two modes for evaluation of atomic subformulae

$t_{[3]}$

?	?	B_0
---	---	-------

$t'_{[3]}$

B'_2	B'_1	B'_0
--------	--------	--------

$$t_{[3]} =_{[3]} t'_{[3]}$$

optimist

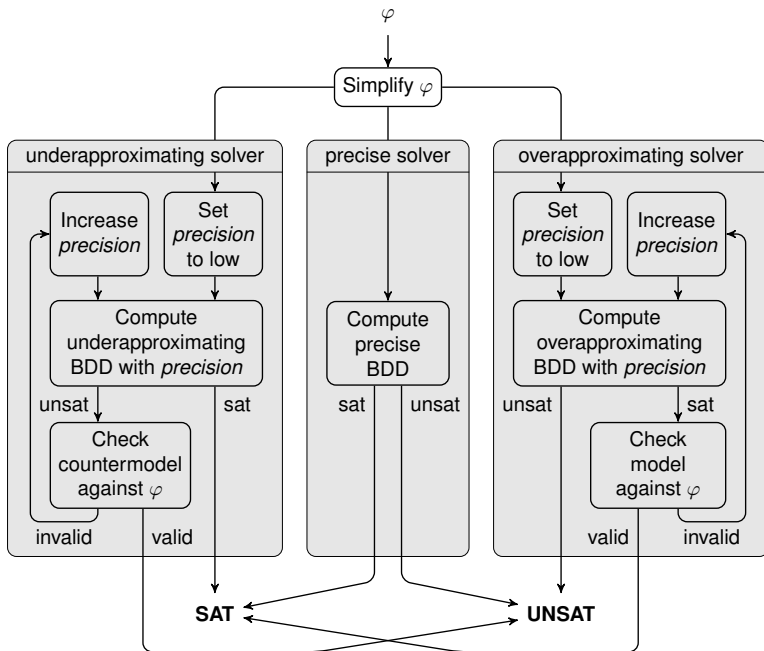
pesimist

$$(B_2 \iff B'_2) \wedge (B_1 \iff B'_1) \wedge (B_0 \iff B'_0)$$

$$true \wedge true \wedge (B_0 \iff B'_0)$$

$$false \wedge false \wedge (B_0 \iff B'_0)$$

- pesimist claims satisfiability \implies formula is satisfiable
- optimist claims unsatisfiability \implies formula is unsatisfiable

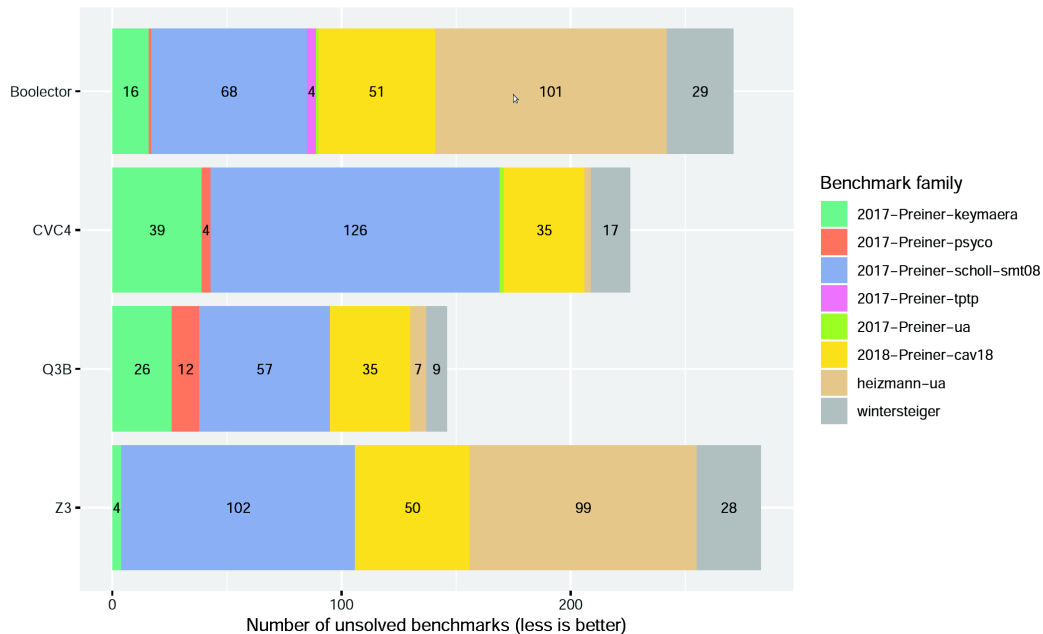


Evaluation of SMT solvers for BV

- results from 2019
- CPU time limit 20 minutes and memory limit 16 GiB per formula

family	total	solved by			
		Boolector	CVC4	Q3B	Z3
2017-Preiner-keymaera	4035	4019	3996	4009	4031
2017-Preiner-psyco	194	193	190	182	194
2017-Preiner-scholl-smt08	374	306	248	317	272
2017-Preiner-tptp	73	69	73	73	73
2017-Preiner-UAutomizer	153	152	151	153	153
20170501-Heizmann-UAutomizer	131	30	128	124	32
2018-Preiner-cav18	600	549	565	565	550
wintersteiger	191	162	174	182	163
Total	5751	5480	5525	5605	5468

Evaluation of SMT solvers for BV



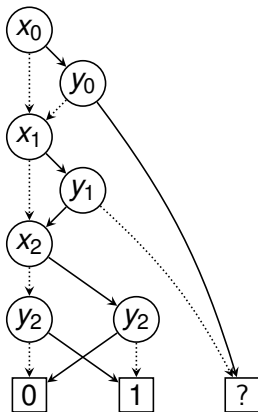
- 2016 1st in the Main Track of the BV division
in Sequential Performance and Parallel Performance
- 2017 1st in the Main Track of the BV division
in Sequential Performance and Parallel Performance
- 2019 2nd in the Single Query Track of the BV division
in Sequential Performance, Parallel Performance, SAT Performance,
and 24 seconds Performance
- 2022 2nd in the Single Query Track of the BV division
in Sequential Performance, Parallel Performance, SAT Performance,
UNSAT Performance, and
1st in 24 seconds Performance

Ongoing research and development of Q3B

- improved caching and static analysis of BV formulae
- advanced sifting algorithms
- use of **partial BDDs**

Ongoing research and development of Q3B

- improved caching and static analysis of BV formulae
- advanced sifting algorithms
- use of **partial BDDs**



- BDDs have known limitations, but also great advantages and efficient libraries
- in the context of SAT/SMT-solving, BDDs like quantifiers

- BDDs have known limitations, but also great advantages and efficient libraries
- in the context of SAT/SMT-solving, BDDs like quantifiers

- BDDs have known limitations, but also great advantages and efficient libraries
- in the context of SAT/SMT-solving, BDDs like quantifiers

[SAT'16] M. Jonáš and J. S.: *Solving Quantified Bit-Vector Formulas Using Binary Decision Diagrams*, SAT 2016.

[SAT'17] M. Jonáš and J. S.: *On Simplification of Formulas with Unconstrained Variables and Quantifiers*, SAT 2017.

[ICTAC'18] M. Jonáš and J. S.: *Abstraction of Bit-Vector Operations for BDD-Based SMT Solvers*, ICTAC 2018.

[IPL'18] M. Jonáš and J. S.: *On the Complexity of the Quantified Bit-Vector Arithmetic with Binary Encoding*, Inf. Process. Lett. 135:57-61, 2018.

[CAV'19] M. Jonáš and J. S.: *Q3B: An Efficient BDD-Based SMT Solver for Quantified Bit-Vectors*, CAV 2019.