Solving Quantified Bit-Vector Formulas Using Binary Decision Diagrams

Martin Jonáš Jan Strejček

Masaryk University, Brno, Czech Republic

Brussels – September 7, 2016

Quantified bit-vector formulas

Example: $\forall x^{32}\, \exists y^{32}\, (y \geqslant_s 0 \wedge x + y = 0)$

Quantified BV formulas naturally arise in software and hardware analysis, e.g.

- invariant synthesis
- ranking function synthesis
- loop summarization
- symbolic state comparision

Traditionally solved by the model-based quantifier instantiation.

When quantifier instantiation fails

Consider the unsatisfiable formula

$$\underbrace{a = 16 \cdot b + 16 \cdot c}_{\varphi} \wedge \forall x \underbrace{(a \neq 16 \cdot x)}_{\psi}.$$

Quantifier instances for all numbers divisible by 16 have to be added to show the unsatisfiability.

When quantifier instantiation fails

Consider the unsatisfiable formula

$$\underbrace{a = 16 \cdot b + 16 \cdot c}_{\varphi} \wedge \forall x \underbrace{(a \neq 16 \cdot x)}_{\psi}.$$

Quantifier instances for all numbers divisible by 16 have to be added to show the unsatisfiability.

Unsatisfiability could be shown by considering the instance $\psi[b + c]$, yielding

$$a = 16 \cdot b + 16 \cdot c \wedge a \neq 16 \cdot (b + c),$$

but considering all possible terms is in general not feasible.

Solution

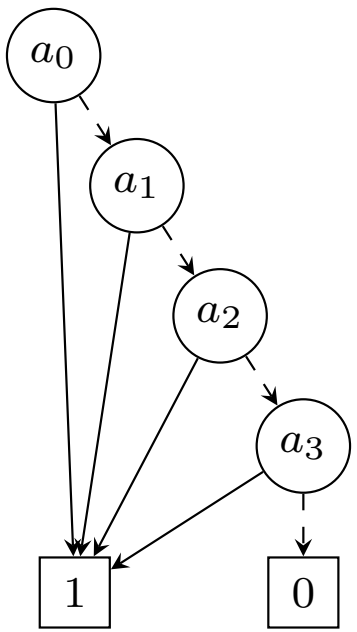
$$\underbrace{a = 16 \cdot b + 16 \cdot c}_{\varphi} \wedge \forall x \underbrace{(a \neq 16 \cdot x)}_{\psi}.$$

BDDs for subformulas can be build bottom-up using the standard BDD operations.

Solution

$$\underbrace{a = 16 \cdot b + 16 \cdot c}_{\varphi} \wedge \forall x \underbrace{(a \neq 16 \cdot x)}_{\psi}.$$

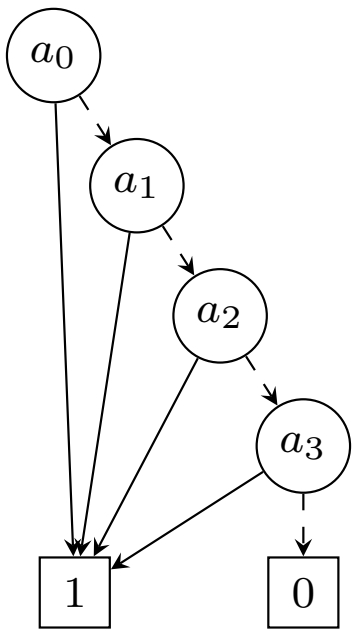
BDDs for subformulas can be build bottom-up using the standard BDD operations. The BDD for $\forall x\, \psi[x]$ is



Solution

$$\underbrace{a = 16 \cdot b + 16 \cdot c}_{\varphi} \wedge \forall x \underbrace{(a \neq 16 \cdot x)}_{\psi}.$$

BDDs for subformulas can be build bottom-up using the standard BDD operations. The BDD for $\forall x\, \psi[x]$ is



The BDD for the whole formula consists only of the node 0.

Towards BDD based SMT solver

Simple conversion of a formula to BDD is not good enough.

Our algorithm further relies on

- formula simplifications
- precomputed initial BDD variable ordering
- approximations

Approximations

What to do with the formula $\exists x \forall y (x \cdot y = 0)$?

Try to solve a simpler formula instead.

Approximations

What to do with the formula $\exists x \forall y (x \cdot y = 0)$?

Try to solve a simpler formula instead.

Notion of approximations of φ:

Underapproximation

- A formula $\underline{\varphi}$ such that $\underline{\varphi} \models \varphi$.
- If $\underline{\varphi}$ is `sat`, φ is `sat`.

Overapproximation

- A formula $\overline{\varphi}$ such that $\varphi \models \overline{\varphi}$.
- If $\overline{\varphi}$ is `unsat`, φ is `unsat`.

Approximations

Represent some bit-vector variables by fewer bits – effective bit-width.

Variable x of bit-width 6. Possible reductions to 3 effective bits:

zero-extension	0	0	0	x_2	x_1	x_0
sign-extension	x_2	x_2	x_2	x_2	x_1	x_0
right zero-extension	x_5	x_4	x_3	0	0	0
right sign-extension	x_5	x_4	x_3	x_3	x_3	x_3

The algorithm

1. Apply simplifications up to the fixed point and convert the formula to the NNF.

The algorithm

1. Apply simplifications up to the fixed point and convert the formula to the NNF.
2. Compute the initial ordering.

The algorithm

1. Apply simplifications up to the fixed point and convert the formula to the NNF.
2. Compute the initial ordering.
3. Call `computeBDD`(φ).
   If the root is 0 return `unsat`, else return `sat`.

The algorithm

1. Apply simplifications up to the fixed point and convert the formula to the NNF.
2. Compute the initial ordering.
3. Call `computeBDD`(φ).
   If the root is 0 return `unsat`, else return `sat`.
4. If the computation did not finish within 0.1 s, also run in parallel:
   - Sequentially try solving $\underline{\varphi}$ with the effective bit-width 1, 2, 4, 6...
     If any of BDDs has the root distinct from 0, return `sat`.
   - Sequentially try solving $\overline{\varphi}$ with the effective bit-width 1, 2, 4, 6...
     If any of BDDs has the root 0, return `unsat`.

Experimental evaluation

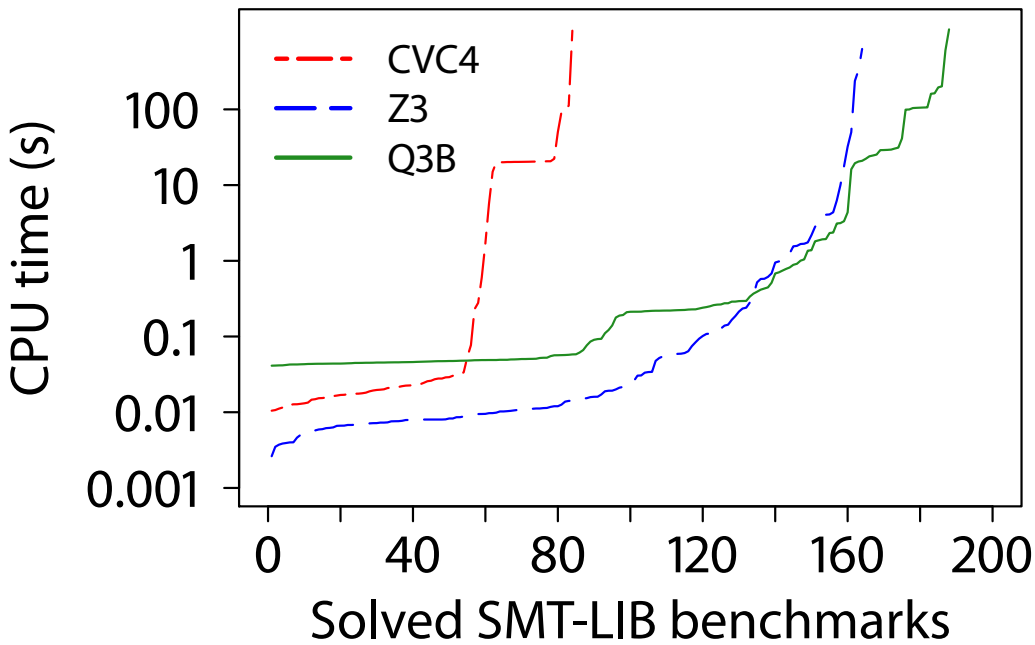
Implemented in a solver called Q3B. Written in C++, using BuDDy to perform BDD operations.

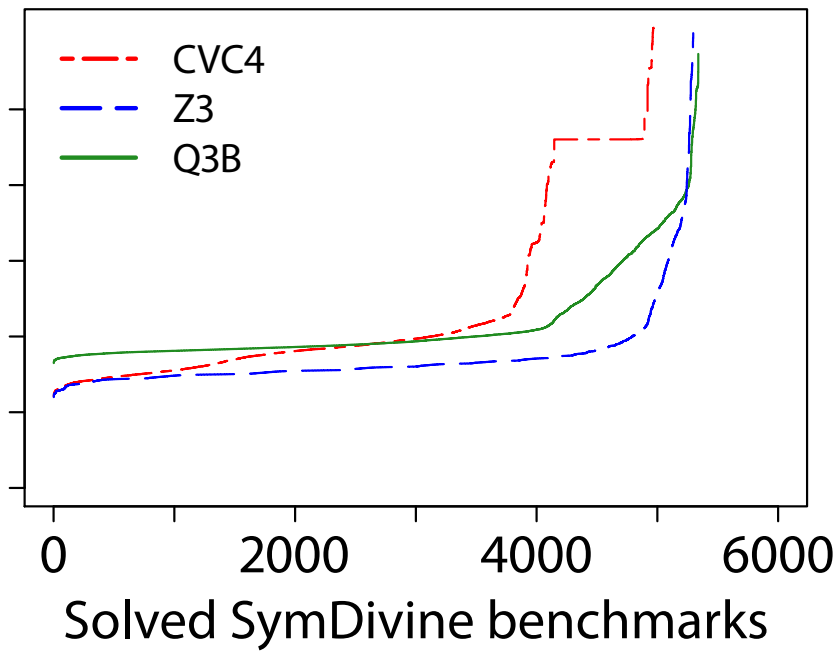
Available at `https://github.com/martinjonas/Q3B`

The solver was evaluated on

- all 191 benchmarks from the BV category of SMT-LIB
- 5 461 formulas generated by the symbolic model checker SymDivine when run on SV-COMP benchmarks

Experimental evaluation – comparison





Experimental evaluation – comparison

	SMT-LIB			
	sat	unsat	unknown	timeout
CVC4	29	55	32	75
Z3	71	93	5	22
Q3B	94	94	0	3

	SymDivine			
	sat	unsat	unknown	timeout
CVC4	1 124	3 845	2	490
Z3	1 135	4 162	22	142
Q3B	1 137	4 202	0	122

SMT competition

Q3B is the winner of BV category of SMT-COMP 2016.

	Known status		Unknown status		
	solved	avg CPU	solved	avg CPU	avg WALL
Boolector	85	1.635	89	11 431	11 422
CVC4	85	1.576	56	29 464	29 453
Q3B	85	0.138	99	12 111	4 059
Z3	85	0.339	78	16 721	16 713

Conclusion and future work

Conclusion

- new algorithm for the SMT solving of quantified bit-vector formulas
- relies on BDDs, simplifications, tailored initial ordering, and approximations
- outperforms state-of-the art SMT solvers Z3, CVC4, and Boolector

Future work

- finer refinement of approximations
- approximate functions and predicates, not only variables
- add uninterpreted functions and arrays