EasyChair Preprint

№ 5948

DQBDD: An Efficient BDD-Based DQBF Solver

Juraj Síč and Jan Strejček

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

June 28, 2021

DQBDD: An Efficient BDD-Based DQBF Solver*

Juraj Síč[1[0000−0001−7454−3751]] and Jan Strejček[2[0000−0001−5873−403X]]

[1] Brno University of Technology, FIT, Brno, Czech Republic
sicjuraj@fit.vut.cz
[2] Masaryk University, Brno, Czech Republic
strejcek@fi.muni.cz

Abstract. This paper introduces a new DQBF solver called DQBDD, which is based on quantifier localization, quantifier elimination, and translation of formulas to binary decision diagrams (BDDs). In 2020, DQBDD participated for the first time in the *Competitive Evaluation of QBF Solvers (QBFEVAL'20)* and won the *DQBF Solvers Track* by a large margin.

1 Introduction

A *binary decision diagram (BDD)* is a data structure proposed by Bryant [5] to succinctly represent all satisfying assignments of a Boolean formula. Unfortunately, BDDs have limited scalability as there exist formulas such that the corresponding BDDs are exponential in the number of Boolean variables [6]. However, it has been also observed that applying a quantifier to a formula variable often reduces the size of the corresponding BDD [15]. This observation suggests that BDDs could be an appropriate data structure for satisfiability solvers processing formulas with quantifiers. Indeed, recently introduced BDD-based solvers are usually aimed at quantified formulas. For example, eBDD-QBF [21] is a solver for *quantified Boolean formulas (QBFs)* and Q3B [15,16] is an SMT-solver for quantified bit-vector formulas.

This paper introduces another BDD-based solver for quantified formulas, namely the tool called DQBDD deciding satisfiability of *dependency quantified Boolean formulas (DQBFs)*. These formulas are quantified Boolean formulas with existential quantifiers of the form $\exists x(D_x)$, where the value of x can depend only on the values of the universally quantified variables in the *dependency set* D_x. For a precise definition of the syntax and semantics of DQBF, we refer to [11]. While deciding satisfiability of a given Boolean formula is NP-complete, the same problem for QBFs is PSPACE-complete and it is even NEXPTIME-complete for DQBFs [22]. Satisfiability of DQBFs has also some practical applications, in particular the *partial equivalence checking (PEC)* [12] which answers the question

* This work has been supported by the Czech Ministry of Education, Youth and Sports project LL1908 of the ERC.CZ programme, and the FIT BUT internal project FIT-S-20-6427.

of whether a given combinational circuit with unknown parts can be equivalent to a given specification. Another application is the *controller synthesis problem (CSP)* [4] which tries to find a controller that keeps a given system in safe states.

The DQBF satisfiability solving is now a hot research topic. The first algorithm [8], based on DPLL, was introduced in 2012. Since then, several different solving techniques were suggested and implemented in DQBF solvers iDQ [9], iProver [18], HQS [13,34,11], and dCAQE [33]. Further, there exist DQBF preprocessors HQSpre [35] and Unique [30] which can significantly reduce a given formula and HQSpre can even directly solve some of them. Research advances in this area are described in existing overviews [19,28]. Out of the mentioned solvers, the best performing tool is HQS, which won the *DQBF Solvers Track* of the *Competitive Evaluation of QBF Solvers (QBFEVAL)* in 2018 and 2019 [23,24].

The following section briefly explains the basic approach of DQBDD to DQBF solving and compares it to the approach of HQS. Section 3 describes the implementation, installation, and usage of our tool. The performance of our tool is then analyzed in Section 4.

2 Approach

Let us first assume that we want to build a BDD-based solver for QBFs. The most straightforward approach is to translate a given formula to the corresponding BDD in a bottom-up manner, i.e., start with atomic subformulas and build BDDs for larger subformulas from previously constructed BDDs for smaller subformulas. The whole formula is satisfiable if and only if the resulting BDD represents at least one satisfying assignment. When processing a quantified subformula $\forall x.\psi$ or $\exists x.\psi$, we handle it as the right side of the corresponding equivalence

$$\forall x.\psi \;\equiv\; \psi[1/x] \wedge \psi[0/x] \qquad \text{or} \qquad \exists x.\psi \;\equiv\; \psi[1/x] \vee \psi[0/x]$$

where $\psi[v/x]$ for $v \in \{0,1\}$ denotes the formula ψ with all the occurrences of x replaced by the value v. Given a BDD for ψ, the BDD for $\psi[1/x] \wedge \psi[0/x]$ or $\psi[1/x] \vee \psi[0/x]$ contains fewer variables than the BDD for ψ (except the case when the BDD for ψ does not contain x; both BDDs are then identical) and the number of its nodes is usually also lower. As mentioned before, the main weakness of BDDs is that they can grow very quickly with an increasing number of variables and the operations on large BDDs get slower. To reduce the size of manipulated BDDs, we first push the quantifiers downwards the syntax tree as far as possible. This process is known as *localization* [11] or *miniscoping* [14].

Now we briefly explain the approach of DQBDD to solving satisfiability of DQBFs. The full description of the algorithm can be found in the master's thesis of Juraj Síč [32]. The approach has basically three steps: preprocessing, quantifier localization, and translation of the input formula to the corresponding BDD.

Formula preprocessing The tool gets an input formula in the DQDIMACS format [9], which implies that the formula is in *prenex conjunctive normal form*

(PCNF). DQBDD then calls HQSpre to reduce the formula. The preprocessed formula is still in prenex normal form, but its matrix (i.e., the part of the formula without the prefix of quantifiers) does not have to be in CNF any more. Alternatively, DQBDD can also read a formula in the prenex cleansed DQCIR format[3] which does not have to be in CNF and HQSpre is thus inapplicable. As the last step of the preprocessing phase, negations are pushed to Boolean variables as the remaining steps of the DQBDD algorithm expect a formula in *negation normal form (NNF)*, where negations appear only in front of variables. Note that NNF has no restrictions on the position of quantifiers, so DQBDD can be easily adjusted to handle DQBFs that are not in prenex normal form.

Quantifier localization In this step, DQBDD applies localization rules [11, Theorems 3 and 4] to push the quantifiers downwards as far as possible. Note that the rule (3d) of Theorem 3 [11] is not valid when applied to subformulas [32,10]. However, the rule can be fixed by additional side conditions [32,10].

Translation to a BDD This step works similarly to the straightforward algorithm for QBFs described at the beginning of this section: the DQBF formula produced by the previous steps is translated to the corresponding reduced ordered BDD in a bottom-up manner. However, handling quantified subformulas is not as simple as for QBF. We use the following quantifier elimination rules.

Universal quantifier elimination We can apply so-called *universal* or *Shannon expansion* to any subformula $\forall x.\psi$ such that all existential quantifiers $\exists y(D_y)$ in ψ satisfy $x \in D_y$. That is, we replace this subformula with

$$\psi_1[0/x] \wedge \psi_2[1/x]$$

where ψ_1 arises from ψ by removing x from all dependency sets D_y and ψ_2 differs from ψ_1 by replacing each variable y existentially quantified inside this formula by a fresh variable y' with the same dependency set. Hence, any universal quantifier can be eliminated as all potential existential quantifiers $\exists y(D_y)$ in ψ violating $x \in D_y$ can be pushed above the subformula. Note that the elimination can increase the number of variables in the subformula.

Existential quantifier elimination The situation for subformulas $\exists y(D_y).\psi$ is different. Roughly speaking, such a subformula can be handled as

$$\psi[0/y] \vee \psi[1/y]$$

but only if ψ contains no quantifiers and each variable in ψ is either a free variable, or a variable from D_y, or an existentially quantified variable y' satisfying $D_{y'} \subseteq D_y$ [11, Theorem 5]. To satisfy these requirements, it may be necessary to first eliminate some universal variable in order to remove it from ψ or from some $D_{y'}$. Recall that the elimination of a universal quantifier can again increase the number of existential quantifiers in the formula.

[3] This is the prenex cleansed QCIR format [17] extended with quantifiers `depend(v, v1, ..., vn)` representing existential variable `v` with dependencies `v1, ..., vn`.

Now assume that we need to translate a subformula of the form

$$\forall x_1 \forall x_2 \dots \forall x_n \exists y_1(D_{y_1}) \exists y_2(D_{y_2}) \dots \exists y_m(D_{y_m}).\psi$$

and ψ has already been translated. Note that the order of these quantifiers can be arbitrarily changed without any impact on the formula semantics as long as all variables in each dependency set D_{y_i} are quantified before y_i. We implemented three possible strategies for quantifier elimination called *none*, *simple*, and *all*.

none Instead of elimination, we push the quantifiers upwards using the reverse version of quantifier localization rules. This strategy is equivalent to an algorithm that skips the quantifier localization and keeps the formula in prenex form.

simple We iteratively eliminate all existential quantifiers for which the elimination rule requirements are satisfied and the universal quantifiers that are not in any dependency set D_{y_i} and thus their elimination does not introduce any fresh variable. The remaining quantifiers are pushed up.

all We iteratively eliminate all quantifiers that can be eliminated. More precisely, we first eliminate all existential quantifiers satisfying the requirements, then we eliminate a selected universal quantifier, and then we repeat the process. If we reach the situation that all universal quantifiers are eliminated and the remaining existential quantifiers cannot be eliminated due to a variable quantified outside the considered subformula, then we push these remaining existential quantifiers up.

If the considered subformula is in fact the whole formula, then we have to apply the *all* strategy as we cannot push any quantifier up in the formula. In this strategy, the universal quantifiers can be eliminated in an arbitrary order. We implemented three heuristics to determine the order, namely *at the beginning*, *current lowest*, and *vars in conjuncts*.

at the beginning This heuristics determines the elimination order of universal variables $x_1, \dots, x_n$ at the beginning of the elimination process according to the number of dependency sets each variable appears in (variables with the lowest number are eliminated first). The motivation is to keep the number of variables added by the elimination process low as long as possible.

current lowest This heuristics is similar to the previous one, but the order is updated according to the current situation every time before the next universal variable is selected for elimination.

vars in conjuncts This heuristics is motivated directly by the use of BDDs. Elimination of a universal variable x produces the BDD for $\psi_1[0/x] \wedge \psi_2[1/x]$. As we have the BDD for ψ in hand and instantiation of a variable is very cheap, for each universal variable x we compute the set of variables in the BDDs for $\psi_1[0/x]$ and $\psi_2[1/x]$ and select the variable with the smallest set.

An experimental comparison of all combinations of elimination strategies and elimination order heuristics [32] shows significant differences between strategies and only small differences between heuristics. We selected the *simple* strategy

with the heuristics *at the beginning* as the default setting. The combination of the *simple* strategy with the heuristics *vars in conjuncts* solved the same number of instances (not the same instances) but it was slightly slower.

Our approach is very close to the current approach of HQS, which also applies preprocessing, quantifier localization, and quantifier elimination using the same elimination strategy *simple* as we use by default. However, there are two important differences. First, HQS uses a succinct representation of Boolean formulas called *and-inverter graphs (AIGs)* [20]. Second, after turning the formula back to prenex normal form, HQS uses *dependency elimination* [34] (which removes universal variables only from some dependency sets) and quantifier elimination to simplify the formula until it gets a QBF, which is then sent to a QBF solver.

3 Implementation and Usage

DQBDD is implemented in C++ under LGPLv3 license. The current stable version is 1.2. For working with BDDs, our tool uses the library CUDD v3.0.0 [31] which also implements Rudell's sifting algorithm [26] for dynamic reordering of BDD variables to keep the size of BDDs small. Further, DQBDD integrates the DQBF preprocessor HQSpre[4] [35] which uses Easylogging++ v9.96.7 library for logging, and SAT solvers PicoSAT [3] and antom [29]. Finally, DQBDD also uses the library cxxopts v2.2.0 for command-line argument parsing.

The sources of DQBDD including all the mentioned libraries can be found at `https://github.com/jurajsic/DQBDD`. Compilation of the tool requires only a C++ compiler supporting the C++14 standard and CMake v3.5 or higher. DQBDD can be compiled into a dynamically linked executable on Linux and Mac systems while static linking is supported only on Linux systems (and is enabled by default). The executables of DQBDD v1.2 are available in the repository.

The tool is executed from command-line as

```
DQBDD [ARGUMENT...] <input file>
```

where `<input file>` specifies the input formula in DQDIMACS [9] or prenex cleansed DQCIR format. The tool supports the following arguments:

`--preprocess 0/1` turns the preprocessing off/on (not applicable for DQCIR).
`--dyn-reordering 0/1` turns off/on the mentioned sifting algorithm in CUDD.
`--localise 0/1` turns off/on the quantifier localization step. Turning off localization effectively enforces the quantifier elimination strategy *none*.
`--elimination-choice 0/1/2` selects the strategy *none*/*simple*/*all* for quantifier elimination. To select *none*, it is more efficient to switch off the quantifier localization step.
`--uvar-choice 0/1/2` selects the heuristics *at the beginning*/*current lowest*/*vars in conjuncts* determining the order of universal quantifier elimination.

The default value of all these arguments is `1` except the last argument, where the default value is `0`.

[4] We use the version distributed with HQS downloaded on March 18, 2021, from `http://abs.informatik.uni-freiburg.de/src/projectfiles/21/HQS.zip`.

Table 1: For each tool and instance type, the table shows the **total** number of solved instances, the number of solved **sat**isfiable and **unsat**isfiable instances, and the number of instances solved **uniq**uely by the solver. All solved CSP instances are satisfiable.

	PEC 3277 instances				**CSP** 404 instances		**SAT** 22 instances			
	total	**sat**	**unsat**	**uniq**	**total**	**uniq**	**total**	**sat**	**unsat**	**uniq**
dCAQE	818	132	686	2	**41**	**15**	**7**	3	4	0
DQBDD	**3035**	**364**	**2671**	**384**	26	4	1	0	1	**1**
HQS	2625	246	2379	5	24	0	6	**4**	2	0
iDQ	534	48	486	1	7	0	**7**	**4**	3	0
iProver	677	83	594	0	19	0	**7**	2	**5**	**1**

4 Experimental Comparison

We compared the performance of DQBDD v1.2 against DQBF solvers iDQ v1.0, iProver v3.4[5], dCAQE v4.0.1, and the current version of HQS[6].

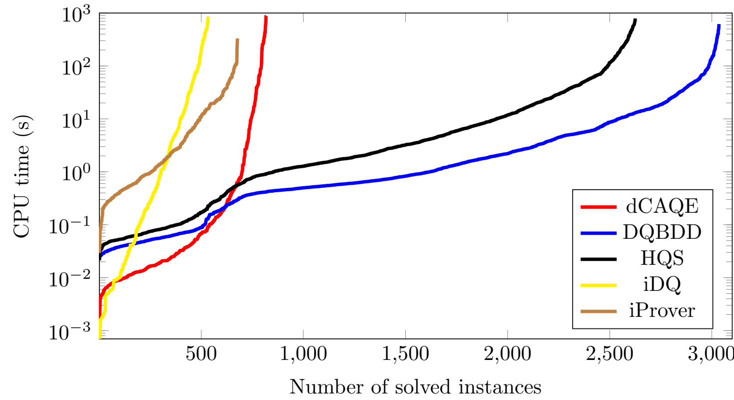
For the experiments, we used the DQBF benchmark set considered also in other recent papers on DQBF [11,10]. The set consists of 4316 instances of partial equivalence checking problem (**PEC**) collected from various sources [9,7,12,27], 461 instances of controller synthesis problem (**CSP**) [4], and 34 instances of **SAT** problem encoded as DQBF with an exponentially smaller number of variables [1].

All our experiments were computed on a 24 core machine with 2.10 GHz Intel Xeon CPU. We set the runtime limit to 900 s of CPU time and the memory consumption limit to 4 GB for each tool and input formula. We employed the framework for reliable benchmarking and resource measurement called BenchExec v2.2 [2] to enforce these limits. BenchExec also isolates the measured processes such that they can run in parallel with minimum interference between each other.

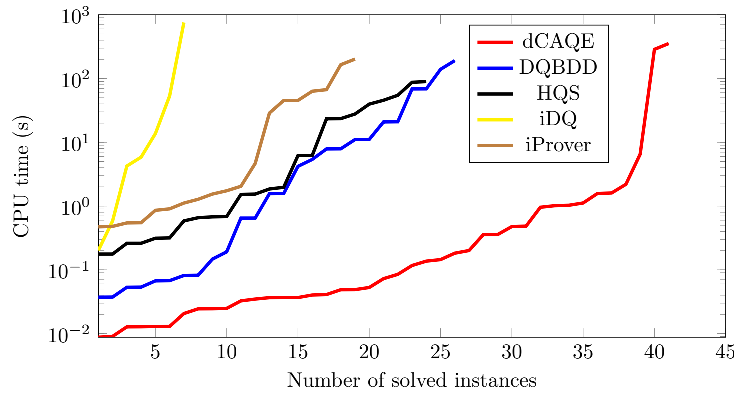
First, we run preprocessor HQSpre on all benchmarks and removed the solved instances from our benchmark set. This leaves us with 3277 PEC instances, 404 CSP instances, and 22 SAT instances. Then we run solvers dCAQE, iDQ, and iProver on the remaining instances in the preprocessed form. We run HQS and DQBDD on the remaining instances in their original form as both these tools call HQSpre in their solving routine. All the considered benchmarks with the corresponding BenchExec definitions and obtained results from the solvers can be found at `https://github.com/jurajsic/DQBFbenchmarks`.

The results are presented in Table 1. DQBDD dominates on PEC instances. This can be also seen in Figure 1a which shows the cactus plot comparing running

[5] Called with "`--qbf_mode true --inst_out_proof false --res_out_proof false`".
[6] Downloaded from `http://abs.informatik.uni-freiburg.de/src/projectfiles/21/HQS.zip` on March 18, 2021.



(a) PEC instances

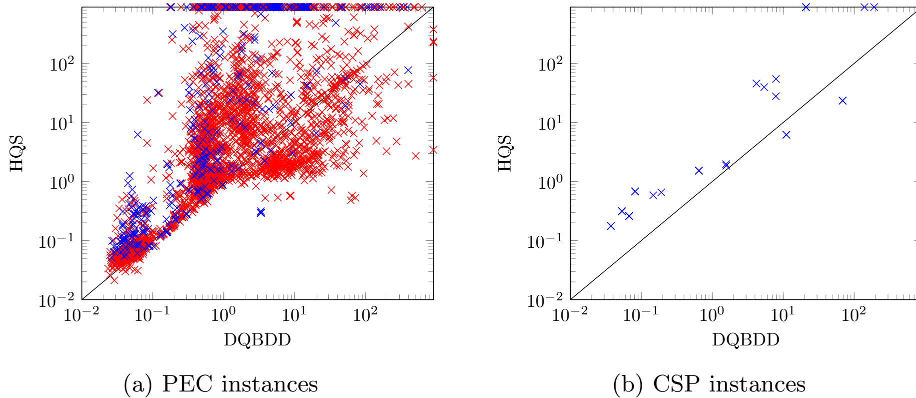


(b) CSP instances

Fig. 1: Cactus plots showing on the x axis the numbers of PEC and CSP instances solved by individual tools for the runtime limit set to values on the y axis.

times of individual solvers on PEC instances. Only HQS can solve a similar number of PEC instances, but it is significantly slower. The total running time of DQBDD on solved PEC instances is 28 081 s, while for HQS, which solved 410 instances less, it is 58 154 s. The scatter plot in Figure 2a compares running times of DQBDD and HQS on individual PEC instances. Furthermore, there was a discrepancy for 7 PEC instances. All these were determined as satisfiable by dCAQE while at least one other solver determined them as unsatisfiable. We believe that all these instances are unsatisfiable as we were able to find a simple unsatisfiable DQBF that dCAQE solves incorrectly [32, Appendix D].

For CSP instances, dCAQE solved the most instances. The comparison of running times of all tools can be found in Figure 1b. As dCAQE sometimes returns an incorrect result, we rather focus on the comparison of the two next



(a) PEC instances (b) CSP instances

Fig. 2: Scatter plots comparing CPU times of DQBDD and HQS on individual satisfiable (blue) and unsatisfiable (red) instances of PEC and CSP.

best solvers, that is DQBDD and HQS. DQBDD needed 190 s to solve the 22 instances solved by both DQBDD and HQS, while HQS needed 238 s. The detailed comparison of running times is shown in Figure 2b.

Finally, DQBDD solved only one SAT instance, but other solvers were not able to solve this instance.

QBF As QBF is a special case of DQBF, DQBDD is also a QBF solver. We tried DQBDD on the QBF benchmarks from the QBFEVAL'20 [25] competition. Out of the 521 QBFs considered in the *Prenex CNF Track*, DQBDD solved 250 instances. However, 214 of them were actually solved by the preprocessor. For the 339 *Prenex non-CNF Track* benchmarks, DQBDD solved 109 instances. As HQSpre works only on CNF benchmarks, no preprocessing was involved. A comparison of these results with the results of QBFEVAL'20 reveals that our solver is currently not competitive with leading QBF solvers.

5 Conclusion

We have presented a new DQBF solver called DQBDD. The tool uses a similar approach based on quantifier localization and elimination as the solver HQS, but DQBDD essentially translates a given formula to the equivalent BDD, which other DQBF solvers do not. Our experimental comparison shows that DQBDD runs significantly faster on instances of the *partial equivalence checking* problem, which is currently the principal application of DQBF solving. The good performance of DQBDD has also been confirmed by winning the *DQBF Solvers Track* of QBFEVAL'20.

References

1. Valeriy Balabanov and Jie-Hong Roland Jiang. Reducing satisfiability and reachability to DQBF, 2015. Talk given at International Workshop on Quantified Boolean Formulas – QBF 2015.
2. Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transf.*, 21(1):1–29, 2019.
3. Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008.
4. Roderick Bloem, Robert Könighofer, and Martina Seidl. SAT-based synthesis methods for safety specs. In *Verification, Model Checking, and Abstract Interpretation*, pages 1–20, 2014.
5. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
6. Randal E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. Computers*, 40(2):205–213, 1991.
7. Bernd Finkbeiner and Leander Tentrup. Fast DQBF refutation. In *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 243–251, 2014.
8. Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. A DPLL algorithm for solving DQBF. In *Pragmatics of SAT (PoS 2012, aff. to SAT 2012)*, 2012.
9. Andreas Fröhlich, Gergely Kovásznai, Armin Biere, and Helmut Veith. iDQ: Instantiation-based DQBF solving. In Daniel Le Berre, editor, *POS-14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria*, volume 27 of *EPiC Series in Computing*, pages 103–116. EasyChair, 2014.
10. Aile Ge-Ernst, Christoph Scholl, Juraj Síč, and Ralf Wimmer. Solving dependency quantified Boolean formulas using quantifier localization. *Theoretical Computer Science*, 2021. Submitted. Preprint available as arXiv:1905.04755v2.
11. Aile Ge-Ernst, Christoph Scholl, and Ralf Wimmer. Localizing quantifiers for DQBF. In Clark W. Barrett and Jin Yang, editors, *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, pages 184–192. IEEE, 2019.
12. Karina Gitina, Sven Reimer, Matthias Sauer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. Equivalence checking of partial designs using dependency quantified Boolean formulae. In *2013 IEEE 31st International Conference on Computer Design, ICCD 2013, Asheville, NC, USA, October 6-9, 2013*, pages 396–403. IEEE Computer Society, 2013.
13. Karina Gitina, Ralf Wimmer, Sven Reimer, Matthias Sauer, Christoph Scholl, and Bernd Becker. Solving DQBF through quantifier elimination. In Wolfgang Nebel and David Atienza, editors, *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 1617–1622. ACM, 2015.
14. John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.
15. Martin Jonáš and Jan Strejček. Solving Quantified Bit-Vector Formulas Using Binary Decision Diagrams. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 267–283. Springer, 2016.

16. Martin Jonáš and Jan Strejček. Q3B: an efficient BDD-based SMT solver for quantified bit-vectors. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 64–73. Springer, 2019.
17. Charles Jordan, Will Klieber, and Martina Seidl. Non-CNF QBF solving with QCIR. In *AAAI Workshop: Beyond NP*, 2016.
18. Konstantin Korovin. iProver - an instantiation-based theorem prover for first-order logic (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer, 2008.
19. Gergely Kovásznai. What is the state-of-the-art in DQBF solving. In *MaCS-16. Joint Conference on Mathematics and Computer Science*, 2016.
20. Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS Dept., UC Berkeley, 2005.
21. Oswaldo Olivo and E. Allen Emerson. A more efficient BDD-based QBF solver. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 675–690. Springer, 2011.
22. Gary Peterson, John Reif, and Salman Azhar. Lower bounds for multiplayer noncooperative games of incomplete information. *Computers & Mathematics with Applications*, 41(7):957–992, 2001.
23. Luca Pulina and Martina Seidl. QBF evaluation 2018, 2018.
24. Luca Pulina, Martina Seidl, and Ankit Shukla. QBF evaluation 2019, 2019.
25. Luca Pulina, Martina Seidl, and Ankit Shukla. QBF evaluation 2020, 2020.
26. Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, pages 42–47, 1993.
27. Christoph Scholl and Bernd Becker. Checking equivalence for partial implementations. In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 238–243, 2001.
28. Christoph Scholl and Ralf Wimmer. Dependency quantified Boolean formulas: An overview of solution methods and applications - extended abstract. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 3–16. Springer, 2018.
29. Tobias Schubert, Matthew Lewis, and Bernd Becker. Antom – solver description, 2010.
30. Friedrich Slivovsky. Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 508–528. Springer, 2020.
31. Fabio Somenzi. CUDD: CU decision diagram package release 3.0.0, 2015.
32. Juraj Síč. Satisfiability of DQBF using binary decision diagrams. Master's thesis, Masaryk University, Faculty of Informatics, 2020.

33. Leander Tentrup and Markus N. Rabe. Clausal abstraction for DQBF. In Mikoláš Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 388–405. Springer, 2019.
34. Ralf Wimmer, Andreas Karrenbauer, Ruben Becker, Christoph Scholl, and Bernd Becker. From DQBF to QBF by dependency elimination. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 2017.
35. Ralf Wimmer, Christoph Scholl, and Bernd Becker. The (D)QBF preprocessor HQSpre – underlying theory and its implementation. *Journal on Satisfiability, Boolean Modeling and Computation*, 11:3–52, 2019.