Pedant: A Certifying DQBF Solver

Franz-Xaver Reichl
TU Wien, Austria

Friedrich Slivovsky
TU Wien, Austria

Abstract

Pedant is a solver for Dependency Quantified Boolean Formulas (DQBF) that combines propositional definition extraction with Counterexample-Guided Inductive Synthesis (CEGIS) to compute a model of a given formula. Pedant 2 improves upon an earlier version in several ways. We extend the notion of dependencies by allowing existential variables to depend on other existential variables. This leads to more and smaller definitions, as well as more concise repairs for counterexamples. Additionally, we reduce counterexamples by determining minimal separators in a conflict graph, and use decision tree learning to obtain default functions for undetermined variables. An experimental evaluation on standard benchmarks showed a significant increase in the number of solved instances compared to the previous version of our solver.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases DQBF, DQBF Solver, Decision Procedure, Certificates

Digital Object Identifier 10.4230/LIPIcs.SAT.2022.20

Supplementary Material *Software (Source Code)*: `https://github.com/fslivovsky/pedant-solver`

Funding Supported by the Vienna Science and Technology Fund (WWTF) under grant ICT19-060, and the Austrian Science Fund (FWF) under grant W1255.

1 Introduction

The last decades showed steady progress in propositional satisfiability (SAT) solving [13, 14, 9]. This lead to the application of SAT solving to problems of various domains, ranging from AI planning [23], over software verification [17] to electronic design automation [32]. In many of these problems – such as AI planning – SAT solving is used to deal with problems beyond NP [23]. As a consequence of this, the propositional encodings of these problems can grow superpolynomially in the size of the original problem. This motivates research on decision procedures for logics that allow more succinct encodings, such as Quantified Boolean Formulas (QBF) or Dependency Quantified Boolean Formulas (DQBF).

Quantified Boolean Formulas (QBF) extend propositional logic by universal and existential quantification over truth values. A QBF is true if it has a *model*, which is a family of Boolean functions (*Skolem functions*) that satisfies the underlying propositional formula for each assignment to the universally quantified variables. The arguments of these functions are implicitly determined by the structure of the quantifier nesting. Evaluating QBF is PSPACE-complete [31]. Hence, it is believed to be a much harder problem as SAT. On the other hand, QBF allow more succinct encodings for a wide range of problems [25]. In practice this advantage may outweigh the disadvantage of slower decision procedures.

Dependency Quantified Boolean Formulas (DQBF) are a generalization of QBF, where each existentially quantified variable is equipped with a set of *dependencies*. The dependencies are subsets of universally quantified variables and determine the possible arguments of Skolem functions. While evaluating DQBF is NEXPTIME-complete [19], DQBF allow to succinctly encode the existence of Boolean functions subject to a set of constraints, equivalence checking of partial circuit designs and bounded synthesis [21, 10, 8].

The DQBF solver Pedant combines the extraction of propositional definitions [27] with Counter-Example Guided Inductive Synthesis (CEGIS) [30, 29, 16] to construct a candidate model. It can generate certificates for true DQBF with almost no overhead.

This tool paper presents Pedant 2, the successor of Pedant [22], and describes a series of new features:

- *Extended dependencies* allow existentially quantified variables as arguments of Skolem functions when dependencies are suitably nested. Their introduction leads to the detection of additional defined Skolem functions as well as more concise definitions.
- A refined *counterexample reduction* procedure based on minimal separators in a conflict graph results in more compact rules to repair counterexamples.
- *Default functions* obtained by decision tree learning are used instead of default values for undetermined existential variables.

In addition, Pedant 2 comes with several minor improvements such as the support for Aiger [4] certificates for true DQBF.

The remainder of the paper is structured as follows. After briefly covering basic concepts in Section 2, we will discuss the underlying algorithm in Section 3. Subsequently, we will describe the above improvements in Section 4. We discuss experimental results in Section 5 before concluding with an outlook on future work in Section 6.

2 Preliminaries

We use standard notation for propositional logic and refer the reader to the *Handbook of Satisfiability* [5] for an introduction to SAT. For *propositional definability* we refer the reader to [18, 27].

We only consider *Dependency Quantified Boolean Formulas* (DQBF) in *Prenex Conjunctive Normal Form* (PCNF). A DQBF $\Phi = \mathcal{Q}.\varphi$ in PCNF consists of a *quantifier prefix* $\mathcal{Q}$ and a *matrix* φ. The quantifier prefix $\mathcal{Q}$ has the shape $\forall u_1 \dots \forall u_n \exists e_1(D_1) \dots \exists e_m(D_m)$, where the variables u_i and e_j are pairwise distinct. We refer to the variables $u_1, \dots, u_n$ as the *universal variables* and $e_1, \dots, e_m$ as the *existential variables*. Moreover, each D_i shall be a subset of the universal variables. We denote the set D_i as the *dependencies* of e_i. The matrix φ is a propositional formula in CNF, where each variable is either universally or existentially quantified in $\mathcal{Q}$.

Given a set X of variables, we write $[X]$ for the set of assignments of X. Let Φ be a DQBF and F be a set of functions $\{f_{e_1}, \dots, f_{e_m}\}$ such that $f_{e_i} : [D_i] \to \{\text{TRUE}, \text{FALSE}\}$ for $1 \leq i \leq m$. For an assignment σ to the universal variables we denote the existential assignment $\{e_1 \mapsto f_{e_1}(\sigma|_{D_1}), \dots, e_m \mapsto f_{e_m}(\sigma|_{D_m})\}$ by $F(\sigma)$. F is a *model* (or a *winning* $\exists$*-strategy*) for Φ if, for each assignment σ to the universal variables, the assignment $\sigma \cup F(\sigma)$ satisfies the matrix φ. A DQBF is *true*, if it has a model, and *false* otherwise.

3 Counterexample-Guided DQBF Solving

In this section we will recap the previously presented CEGIS algorithm [22] in Algorithm 1. For the remaining part of this paper let Φ be some DQBF with matrix φ and existential variables E. Moreover, for each $e \in E$ we denote the dependencies of e by $D(e)$.

The core idea of Algorithm 1 is to incrementally construct candidate Skolem functions. In order to do so we try to obtain counterexamples with respect to the candidate model and use them to refine the candidate model. For this purpose we first try to compute definitions for the existential variables in terms of their dependencies. These definitions either describe

Skolem functions or the DQBF is false. Thus, we set the candidate Skolem functions for the defined variables according to their definition. Throughout the remaining steps of the algorithm these functions will not be changed. Note that in contrast to the original algorithm, here we only compute definitions at the beginning. This change was motivated by the observation that in practice most often we do not find definitions for variables that were undefined initially.

For the undefined variables we initially set the candidate Skolem function to some default value. Then we try to iteratively construct a Skolem function for these variables. For this purpose we check if we can find a counterexample to the current candidate under an assignment for the *arbiter variables* – which is initially empty. This means we check if we can find an assignment that satisfies the negation of the matrix and that is consistent with the current candidate model. If we cannot find such a counterexample, our candidate Skolem functions are indeed proper Skolem functions, which means that the DQBF is true.

Otherwise, we obtain a counterexample σ. In general, we are not interested in the complete counterexample, instead we want to reduce it. For this purpose we apply assumption-based SAT solving [7]. This means we check the satisfiability of the matrix under σ. As this SAT check yields that the formula is unsatisfiable under the given assumptions, we can compute a set of failed assumptions. We then use this set as a reduced counterexample, denoted by $\hat{\sigma}$. Subsequently, we will refer to this approach for reducing counterexamples as *reduction by core extraction*.

To repair the counterexample we apply two techniques. First, we check if it suffices to change the assignment for a single existential variable e. This is the case if the reduced counterexample only contains a single existential variable. To rule out the current counterexample in subsequent iterations we now add a *forcing clause* to the candidate model. A forcing clause represents an implication that asserts that under the projection of $\hat{\sigma}$ to $D(e)$ the variable e must be assigned to $\neg\sigma(e)$.

If we cannot introduce a forcing clause we do not know a priori which existential variables in the counterexample need to be assigned differently. In order to get control over the assignments of these existential variables we introduce *arbiter clauses*. This means we introduce for each variable e in $\hat{\sigma}$ a new auxiliary variable a – which we call *arbiter variable* – and arbiter clauses $a \vee \neg\sigma|_{D(e)} \vee \neg e$ and $\neg a \vee \neg\sigma|_{D(e)} \vee e$. Note that we used the original counterexample and not the reduced one for introducing the arbiter clause. By fixing the assignment for the arbiter variable we can thus fix the assignment of the associated existential variable under the assignment $\sigma|_{D(e)}$. To rule out the current counterexample for the subsequent iterations we first obtain a clause C of arbiter literals by negating the arbiter assignment. Next, we add for each existential variable e in $\hat{\sigma}$ the associated arbiter variable a to C if $\hat{\sigma}$ contains $\neg e$, respectively $\neg a$ if $\hat{\sigma}$ contains e. Then we add C to a SAT solver. If the clauses of arbiter literals can be satisfied, we know that we can find an assignment to the arbiter variables that deals with all encountered counterexamples. By using such a satisfying assignment as the arbiter assignment for the next iteration we ensure that we will not encounter the same counterexample again. If the clauses cannot be satisfied then every assignment to the arbiter variables entails a counterexample. This means that the DQBF is false.

4 Improvements

In this section we will discuss several improvements to the base algorithm presented in the last section.

Algorithm 1 Solving DQBF by CEGIS and Definition Extraction.

```
1:  procedure Solve(Φ)
2:      model ← ComputeDefinitions()
3:      initializeDefaults(model)
4:      arbiterFormula, arbiterAssignment ← ∅
5:      loop
6:          if checkModel(model, arbiterAssignment) then
7:              return true
8:          σ ← getCounterExample(model, arbiterAssignment)
9:          σ̂ ← getCore(model, σ)                          ▷ get failed assumptions
10:         if hasForcingClause(σ) then
11:             addForcingClause(model, σ̂)
12:             continue
13:         failedArbiters ← Clausify(σ̂|_A)                ▷ A the arbiter variables
14:         for ℓ ∈ σ̂|_E do
15:             a ← addArbiter(model, ℓ, σ|_{D(var(ℓ))})
16:             addArbiterLiteral(failedArbiters, a)
17:         arbiterFormula ← arbiterFormula ∪ {failedArbiters}
18:         if arbiterFormula is satisfiable then
19:             arbiterAssignment ← getModel(arbiterFormula)
20:         else
21:             return false
```

4.1 Extended Dependencies

One can often obtain more and more compact definitions by allowing definitions of existential variables not only in terms of their dependencies but also other existential variables. We illustrate this with the following example:

▶ **Example 1.** Consider the DQBF $\forall u_1 \forall u_2 \, \exists e_1(u_1) \exists e_2(u_1, u_2).\, (e_2 \Leftrightarrow (e_1 \wedge u_2)) \wedge (e_1 \vee u_1)$, where neither e_1 nor e_2 is defined in terms of its dependencies. But we can see that e_2 is defined by $e_1 \wedge u_2$.

In order to make use of this we introduce *extended dependencies*. The extended dependencies of an existential variable e – denoted as $ED(e)$ – contain besides the dependencies $D(e)$ also all the existential variables whose dependencies are contained in $D(e)$. To also take variables with the same dependencies into account, we introduce some ordering $<_E$ on the existential variables. In Pedant 2 we order the existential variables according to their index in the DQDIMACS input. If two variables e_1 and e_2 have the same dependencies, we add e_1 to $ED(e_2)$ if $e_1 <_E e_2$. Additionally, if we find a definition ψ for a variable e we also add e to the extended dependencies of all variables that contain $var(\psi)$.

Replacing dependencies with extended dependencies improves our solver in two ways. First this allows us to compute more and also more compact definitions – as we have seen in the above example. Second this also means that more counterexamples can be resolved by forcing clauses. Remember that in Algorithm 1 we add a forcing clause whenever we only have a single existential variable in the reduced counterexample. By using extended dependencies we can introduce a forcing clause whenever there is an existential variable in the reduced counterexample that contains all the other existential variables in its extended dependencies. The two improvements mentioned above mean that Skolem functions may not

only use universal variables from the dependencies but also existential variables from the extended dependencies. As the used setup allows to replace each existential variable e in a Skolem function by the application of the Skolem function for e, we can still obtain Skolem functions that only depend on $D(e)$.

4.2 Counterexample Reduction

We have already discussed that we want small counterexamples. While counterexamples can be reduced by core extraction, this approach is suboptimal in the sense that the generated reduced counterexamples are not guaranteed to be minimal. In order to obtain a minimal counterexample, a minimal set of failed assumptions could be computed in the core extraction. This can be done by a SAT solver like Picosat [2].

In general the computation of minimal sets of failed assumptions is relatively costly. Additionally, we want to impose certain properties on the reduced counterexamples – which we will discuss below. For this reason we present an alternative technique for reducing counterexamples that is based on computing a minimal separator in a *conflict graph* – our notion of a conflict graph is closely related to *implication graphs* in SAT [5].

If we find a counterexample that is consistent with the current candidate model and that falsifies the matrix, then it falsifies some clause C in the matrix. We want to analyze why this clause was falsified. This is done by considering the conflict graph. The conflict graph is a directed graph that contains a vertex for each universal, existential and arbiter variable. Moreover, if the assignment of an existential variable e is entailed by some rule (i.e. a definition, a forcing clause or an arbiter clause) then the conflict graph contains an edge for each variable v in this rule to e. Subsequently, we refer to the variables in C as *conflict variables*. Moreover, we call the variables that are connected to a conflict variable and only have outgoing edges but no incoming edges, *source variables*.

Based on the conflict graphs we can now give two properties a reduced counterexample shall have. First, the reduced counterexample shall be a separator of the conflict variables and the source variables. This is motivated by the consideration that otherwise the reduced counterexample would be consistent with the candidate model but also satisfy the clause C. Second, we want to ensure that at least one existential source is contained in the reduced counterexample. We know that the assignments of all non-source existentials are entailed by some rule, thus in order to fix the current counterexample we have to assign the existential sources differently.

Next we distinguish between two cases. First, suppose it suffices to assign a single literal in the sources differently to rule out the counterexample. This means we can obtain a forcing clause for a variable e from the sources. To reduce the counterexample we compute a minimal separator of the source and the conflict variables subject to the following constraints:

- The forced variable e shall be contained in the separator.
- Every other existential variable in the separator shall be contained in $ED(e)$.

To compute the separator we use the *Boykov-Kolmogorov max-flow algorithm* [6]. In the second case it does not suffice to assign a single variable differently, this means we have to introduce arbiter clauses. As we do not know which existential variable needs to be assigned differently we use the set of all sources as a reduced counterexample.

4.3 Default Functions

As mentioned above we use default values in order to fix the assignments of undefined variables in case no forcing clause, respectively no arbiter clause applies. Subsequently, we generalize these default values by suitably chosen functions. Actually, we could use any

function that complies with the dependencies to fix the assignment of an existential variable, as defaults only come into play if no other rule applies. We try to obtain a good guess by using decision tree learning based on counterexamples seen so far (cf. Manthan [12]).

For each existential variable e we use a decision tree, whose sample space is given by the set of all assignments to the dependencies of e. The labels for the samples are either true or false and indicate whether the variable e shall be assigned to true or false. The function computed by the decision tree then gives the default function.

We only use counterexamples for decision tree learning if the counterexample can be fixed by a forcing clause for the variable e. In this case we use the projection of the counterexample to the dependencies of e as a sample and label it with $\neg\sigma(e)$ – where σ denotes the counterexample. We only use counterexamples in this case because only if we have a forcing clause we know how to assign e and thus know how to label the sample.

As we want to incrementally train the tree we use *Hoeffding trees*[1] [15] to represent the decision trees. Hoeffding trees are decision trees that do not keep track of the complete samples in the tree. Instead, they have in each leaf a counter for each possible label. If a sample is added to the tree, it first classifies the sample to the appropriate leaf and increases the corresponding counter. If the new sample changed the majority class in the node, then the label of this leaf is changed. Moreover, a suitable evaluation function like information gain is used to determine whether a split shall be applied.

We illustrate the usage of default functions with the following example:

▶ **Example 2.** Let n be some positive integer then consider the DQBF

$$\forall u_1 \dots \forall u_n \, \exists e_1(u_1, \dots, u_n) \exists e_2(u_2, \dots, u_n). \\ (e_1 \Leftrightarrow \mathrm{XOR}(u_1, \dots, u_n)) \wedge (u_1 \vee u_2 \vee e_1 \vee \neg e_2) \wedge (\neg u_1 \vee \neg u_2 \vee e_1 \vee e_2).$$

While e_1 has a definition e_2 does not. In this example, we will get a large number of counterexamples: A lower bound for the number of counterexamples is given by the number of assignments that assign an even number of universal variables to true, u_1 and u_2 to true and e_2 to false. Each counterexample gives a forcing clause. Moreover, we can see that in each counterexample where we have $\neg e_2$ we also have u_2, respectively that in counterexamples with e_2 we have $\neg u_2$. Thus, after a sufficiently large number of counterexamples the decision tree learning will introduce a split for u_2. We thus obtain the default function $f(u_2, \dots, u_n) = u_2$. By using this function we can then immediately show that the formula evaluates to true. Thus, learned default functions can reduce the number of conflicts.

4.4 Further Improvements

Besides the major changes we described until now Pedant 2 also implements minor changes.

Aiger Certificates. As a core idea of our algorithm is the refinement of candidate Skolem functions, Pedant 2 can compute certificates for true DQBF with almost no overhead. In the previous version only certificates in the form of DIMACS CNF formulas could be generated. In the current version also certificates given as circuits in the AIGER format [4] are supported.

Preprocessing. The experimental evaluation in [22] showed that for certain formulas the preprocessing of formulas can be advantageous for our solver. This motivated the idea of pruning the dependencies of a given formula by means of the *Reflexive Resolution Path*

[1] Note that our setup does not fulfil the requirement of independent samples. Thus, the Hoeffding trees are not necessarily asymptotically arbitrarily close to a decision tree learned by traditional batch learning.

Table 1 Solved Instances QBFEVAL'20 DQBF Track.

		DQBDD		HQS		Pedant		Pedant2	
Family	Total	Solved	PAR2	Solved	PAR2	Solved	PAR2	Solved	PAR2
Balabanov	34	14	2175.3	**18**	1880.4	14	2281.6	**18**	1958.6
Bloem	90	34	2278.8	33	2298.4	37	2163.8	**41**	1976.0
Kullmann	50	**50**	32.5	36	1100.5	33	1397.8	**50**	78.3
Scholl	90	83	283.3	81	406.2	81	362.0	**85**	207.7
Tentrup	90	85	249.0	79	481.5	15	3047.7	**86**	236.2
All	354	266	928.2	247	1146.1	180	1833.6	**280**	814.4

Dependency Scheme (RRS) [33, 28]. We decided to incorporate this preprocessing step directly into our solver. This is motivated by the consideration that on the one hand the introduction of arbiter clauses prefers small dependencies but on the other hand the computation of definitions and the search for forcing clauses prefers large dependencies. By adding the preprocessing step directly to Pedant 2, it has a direct access both to the original dependencies and to the pruned dependencies.

Usage of other SAT Solvers. In the previous version of the solver we did only support the SAT solver CaDiCaL [3] as backend solver. The current version of the solver also supports the solver Glucose [1]. Moreover, we provide an interface which allows to easily extend our solver such that also other SAT solvers can be used.

5 Experimental Evaluation

In this section we will on the one hand give a comparison of Pedant 2 with other state-of-the-art DQBF solvers and on the other hand we will illustrate the impact of the presented techniques by giving a small ablation study. For evaluating the solvers we used instances from the DQBF track of QBFEVAL'20 [20]. To compare the solvers we use on the one hand the number of solved instances within a given timeout and on the other hand the PAR2 score[2]. For all experiments described below we used a cluster with Intel Xeon E5649 processors at 2.53 GHz running 64-bit Linux. The presented results are based on single runs of the respective solvers where we imposed a time and memory limit of 1800 seconds and 8 GB, by using RunSolver [24]. We run each solver with its default configuration, except for the evaluations for the ablation study where we used the appropriate configurations to disable the respective features.

We compared Pedant 2 with the previous version of Pedant [22], the DQBF solver DQBDD 1.2 [26] and the solver HQS 2 [11]. The results are given in Table 1. In particular these results show that Pedant 2 could significantly improve on the instances from the *Tentrup* family. Not only Pedant 2 could solve more instances than all the other solvers, it could also generate certificates for all true DQBF. We validated these certificates by using a tool which is available as part of the Pedant 2 system.

The results for the ablation study are given in Table 2. We compare four different configurations:

[2] The Penalized Average Runtime (PAR) is the average runtime, with the time for each unsolved instance calculated as a constant multiple of the timeout.

Table 2 Solved Instances QBFEVAL'20 DQBF Track – Ablation Study.

		Pedant		noSep + noML		noML		Pedant2	
Family	Total	Solved	PAR2	Solved	PAR2	Solved	PAR2	Solved	PAR2
Balabanov	34	14	2281.6	**20**	1776.0	18	1957.3	18	1958.6
Bloem	90	37	2163.8	31	2385.7	**41**	1979.7	**41**	1976.0
Kullmann	50	33	1397.8	40	816.9	**50**	76.2	**50**	78.3
Scholl	90	81	362.0	75	604.3	80	402.7	**85**	207.7
Tentrup	90	15	3047.7	24	2785.3	**86**	220.2	**86**	236.2
All	354	180	1833.6	190	1754.3	275	860.4	**280**	814.4

1. Disable extended dependencies, separator-based counterexample reduction and machine learning. Due to their deep integration into Pedant 2, extended dependencies cannot be disabled. For this reason this configuration is represented by the previous version of Pedant.
2. Disable machine learning and separator-based counterexample reduction.
3. Disable machine learning.
4. Default configuration.

The table shows that the technique with the by far most significant impact is the separator-based counterexample reduction. Still, we have to remember that the separator based counterexample reduction requires the usage of extended dependencies. Thus, the significance of using extended dependencies is not fully captured in the table. A closer analysis of the results shows that in most of the instances that could not be solved by Pedant 2 there is only a relatively small number of forcing clauses. For this reason no, respectively only few shallow decision trees, can be learned. Thus, with a more reliable source for samples for the decision trees, the learning of default functions may further improve the solver.

6 Conclusion

We detailed several improvements introduced in Pedant 2. Jointly, these resulted in significantly improved performance on benchmark instances compared to the initial release. In particular, the combination of extended dependencies and counterexample reduction lead to many counterexamples being resolved by forcing clauses. Generally, fixing a counterexample through a forcing clause is more efficient than the introduction of arbiter variables, since only a single candidate Skolem function is modified. However, it requires that variables occurring in a counterexample have dependencies than can be linearly ordered. While this is the case for many benchmark formulas, it appears to be less common in "genuine" DQBF instances, where counterexamples often involve multiple existential variables with incomparable dependency sets. In such cases, Pedant 2 still has to create arbiter variables that define the values of existential variables for complete assignments of their dependency sets, negating the effect of counterexample reduction. Finding a more efficient way of dealing with such counterexamples is the most important challenge for future work.

References

1 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 399–404, 2009.

2 Armin Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4(2-4):75–97, 2008. doi:10.3233/sat190039.

3 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

4 Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

5 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

6 Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26:1124–1137, September 2004. URL: http://www.csd.uwo.ca/~yuri/Abstracts/pami04-abs.shtml.

7 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

8 Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe, and Leander Tentrup. Encodings of bounded synthesis. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*, volume 10205 of *Lecture Notes in Computer Science*, pages 354–370, 2017.

9 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artificial Intelligence*, 301:103572, 2021. doi:10.1016/j.artint.2021.103572.

10 Karina Gitina, Sven Reimer, Matthias Sauer, Ralf Wimmer, Christoph Scholl, and Bernd Becker. Equivalence checking of partial designs using dependency quantified Boolean formulae. In *IEEE 31st International Conference on Computer Design, ICCD 2013,*, pages 396–403. IEEE Computer Society, 2013.

11 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*, pages 1617–1622. ACM, 2015.

12 Priyanka Golia, Subhajit Roy, and Kuldeep S. Meel. Manthan: A data-driven approach for Boolean function synthesis. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020*, volume 12225 of *Lecture Notes in Computer Science*, pages 611–633. Springer, 2020.

13 Marijn J. H. Heule, Matti Järvisalo, and Martin Suda. SAT competition 2018. *J. Satisf. Boolean Model. Comput.*, 11(1):133–154, 2019.

14 Marijn J.H. Heule, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Report Series B*. Department of Computer Science, University of Helsinki, Finland, 2019.

15 Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 97–106, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/502512.502529.

16 Susmit Jha and Sanjit A. Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54(7):693–726, 2017.

17 Daniel Kroening. Software verification. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 16, pages 505–532. IOS Press, Amsterdam, 2009.

18 Jérôme Lang and Pierre Marquis. On propositional definability. *Artif. Intell.*, 172(8-9):991–1017, 2008. doi:10.1016/j.artint.2007.12.003.

19 G. Peterson, J. Reif, and S. Azhar. Lower bounds for multiplayer noncooperative games of incomplete information. *Computers & Mathematics with Applications*, 41(7):957–992, 2001.

20 Luca Pulina and Martina Seidl. The 2016 and 2017 QBF solvers evaluations (qbfeval'16 and qbfeval'17). *Artif. Intell.*, 274:224–248, 2019.

21 Markus N. Rabe. A resolution-style proof system for DQBF. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017*, volume 10491 of *Lecture Notes in Computer Science*, pages 314–325. Springer, 2017.

22 Franz-Xaver Reichl, Friedrich Slivovsky, and Stefan Szeider. Certified DQBF solving by definition extraction. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing – SAT 2021*, pages 499–517, Cham, 2021. Springer International Publishing.

23 Jussi Rintanen. Planning and sat. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 15, pages 483–504. IOS Press, Amsterdam, 2009.

24 Olivier Roussel. Controlling a solver execution with the runsolver tool. *J. Satisf. Boolean Model. Comput.*, 7(4):139–144, 2011.

25 Ankit Shukla, Armin Biere, Luca Pulina, and Martina Seidl. A survey on applications of quantified Boolean formulas. In *ICTAI*, pages 78–84. IEEE, 2019.

26 Juraj Síč. Satisfiability of DQBF using binary decision diagrams. Master's thesis, Masaryk University, Brno, Czech Republic, 2020.

27 Friedrich Slivovsky. Interpolation-based semantic gate extraction and its applications to QBF preprocessing. In *CAV (1)*, volume 12224 of *Lecture Notes in Computer Science*, pages 508–528. Springer, 2020.

28 Friedrich Slivovsky and Stefan Szeider. Soundness of Q-resolution with dependency schemes. *Theor. Comput. Sci.*, 612:83–101, 2016.

29 Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 136–148. ACM, 2008.

30 Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006*, pages 404–415. ACM, 2006.

31 Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong, editors, *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, pages 1–9. ACM, 1973.

32 Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proc. IEEE*, 103(11):2021–2035, 2015.

33 Ralf Wimmer, Christoph Scholl, Karina Wimmer, and Bernd Becker. Dependency schemes for DQBF. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016*, volume 9710 of *Lecture Notes in Computer Science*, pages 473–489. Springer, 2016.