Incremental QBF Solving by DepQBF*

Florian Lonsing and Uwe Egly

Vienna University of Technology
Institute of Information Systems
Knowledge-Based Systems Group
http://www.kr.tuwien.ac.at/

Abstract. The logic of quantified Boolean formulae (QBF) extends propositional logic by explicit existential and universal quantification of the variables. We present the search-based QBF solver DepQBF which allows to solve a sequence of QBFs incrementally. The goal is to exploit information which was learned when solving previous formulae in the process of solving the next formula in a sequence. We illustrate incremental QBF solving and potential usage scenarios by examples. Incremental QBF solving has the potential to considerably improve QBF-based workflows in many application domains.

Keywords: quantified Boolean formulae, QBF, search-based solving, Q-resolution, clause learning, cube learning, incremental solving.

1 Introduction

Propositional logic (SAT) has been widely applied to encode problems from model checking, formal verification, and synthesis. In these practical applications, an instance of a given problem is encoded as a formula. The satisfiability of this formula is checked using a SAT solver. The result of the satisfiability check is then mapped back and interpreted on the level of the problem instance.

Encodings of problems often give rise to sequences of closely related formulae to be solved, in contrast to one single formula. A prominent example is SAT-based bounded model checking (BMC) [1]. Rather than solving each formula in the sequence individually, *incremental solving* [6] aims at employing information that was learned when solving one formula for solving the next formulae. The overall goal is to speed up the solving process of the entire sequence of formulae.

We consider the problem of incrementally solving a sequence of quantified Boolean formulae (QBF). The decision problem of QBF is PSPACE-complete. Existential and universal quantification together with possible quantifier alternations in QBF potentially allow for exponentially more succinct encodings of problems than propositional logic [2]. This property makes QBF an interesting modelling language for practical applications.

Incremental QBF solving was first applied in the context of QBF-based bounded model checking of partial designs [14]. We extended our QBF solver

* Supported by the Austrian Science Fund (FWF) under grant S11409-N23.

DepQBF [11, 12] by *general-purpose* incremental solving capabilities. Our approach adopts ideas from incremental SAT solving, it is *application-independent* and hence applicable to QBF encodings of *arbitrary* problems. Furthermore, our implementation is publicly available, it features APIs in the C and Java languages and thus facilitates the use of incremental QBF solving in practice.[1]

We present incremental QBF solving from a general perspective. During the solving process, QBF solvers learn information about a QBF in terms of restricted inferences in the Q-resolution calculus. Information learned from previous QBFs must be maintained to prevent unsound inferences. Regarding practical applications, we illustrate the API of our incremental QBF solver DepQBF by means of examples to make its use more accessible. Incremental QBF solving has the potential to improve QBF-based workflows in many applications.

2 Quantified Boolean Formulae

A QBF $\psi := \hat{Q}.\,\phi$ in prenex conjunctive normal form (PCNF) consists of a quantifier-free propositional formula ϕ in CNF containing the variables V and a quantifier prefix $\hat{Q}$. The prefix $\hat{Q} := Q_1B_1 \dots Q_nB_n$ contains sets B_i of propositional variables and quantifiers $Q_i \in \{\forall, \exists\}$. We assume that $B_i \neq \emptyset$, $\bigcup B_i = V$ and $B_i \cap B_j = \emptyset$ for $i \neq j$. The sequence of sets B_i introduces a linear ordering of the variables: given two variables x, y, we define $x < y$ if and only if $x \in B_i$, $y \in B_j$ and $i < j$. In the following, we consider QBFs in PCNF.

An *assignment* $A : V \rightarrow \{\mathbf{t}, \mathbf{f}\}$ is a (partial) mapping from the set of all propositional variables V to truth values *true* ($\mathbf{t}$) and *false* ($\mathbf{f}$). To allow for simple notation, we represent an assignment A as a set $\{l_1, \dots, l_k\}$ of literals where, for a variable x assigned by A, we have $l_i = x$ ($l_i = \neg x$) if x is mapped to $\mathbf{t}$ ($\mathbf{f}$). Given a QBF ψ, a variable $x \in B_i$ and the assignment $A = \{l\}$ with $l = x$ ($l = \neg x$), the QBF $\psi[A]$ *under the assignment* A is obtained from ψ by replacing every occurrence of x in ψ with the syntactic truth constant $\top$ ($\bot$) denoting *true* (*false*), deleting x from the prefix (along with Q_iB_i if $B_i = \emptyset$) and applying simplifications using the annihilator and identity properties of $\wedge$, $\vee$, $\top$ and $\bot$ of Boolean algebra.

The semantics of QBF is defined recursively based on the syntactic structure. The QBF $\psi = \top$ ($\psi = \bot$), which consists of the syntactic truth constant *true* (*false*), is satisfiable (unsatisfiable). The QBF $\psi = \exists(\{x\} \cup B_1) \dots Q_nB_n.\,\phi$ is satisfiable if and only if $\psi[x]$ or $\psi[\neg x]$ is satisfiable. The QBF $\psi = \forall(\{x\} \cup B_1) \dots Q_nB_n.\,\phi$ is satisfiable if and only if $\psi[x]$ and $\psi[\neg x]$ is satisfiable.

Example 1. The QBF $\forall x \exists y.\,(x \vee \neg y) \wedge (\neg x \vee y)$ is satisfiable. We assign the variables in the ordering of the prefix. Both $\psi[x] = \exists y.\,(y)$ and $\psi[\neg x] = \exists y.\,(\neg y)$ are satisfiable, since $\psi[x, y] = \top$ and $\psi[\neg x, \neg y] = \top$, respectively.

[1] DepQBF is free software: `http://lonsing.github.io/depqbf/`

3 Search-Based QBF Solving with Learning

Modern search-based QBF solvers are based on an extension of the conflict-driven clause learning approach (CDCL), which is applied in SAT solving [15]. In this QBF-specific approach called QCDCL [8, 10, 13, 16], a backtracking search procedure related to the DPLL algorithm [4, 5] is used to generate assignments to control the application of the inference rules in the *Q-resolution calculus* [3, 8]. New learned clauses and cubes are inferred and added to the formula.

In the following, we present the rules of the Q-resolution calculus to derive learned clauses and cubes, called *constraints*, in QCDCL-based QBF solvers. Given a QBF $Q_1B_1 \dots Q_nB_n.\, \phi$ and a literal l of a variable $x \in B_i$, the quantifier type of the variable x of l is denoted by $q(l)$ where $q(l) = \forall$ ($q(l) = \exists$) if $Q_i = \forall$ ($Q_i = \exists$). To allow for a uniform presentation of the rules to derive clauses and cubes in the calculus, we represent clauses and cubes as sets of literals.

$$\frac{C_1 \cup \{p\} \qquad C_2 \cup \{\neg p\}}{C_1 \cup C_2} \quad \begin{array}{l}\text{if } \{x, \neg x\} \not\subseteq (C_1 \cup C_2),\ \neg p \notin C_1,\ p \notin C_2 \text{ and} \\ \text{either (1) } C_1, C_2 \text{ are clauses and } q(p) = \exists \\ \text{or (2) } C_1, C_2 \text{ are cubes and } q(p) = \forall\end{array} \qquad (res)$$

$$\frac{C \cup \{l\}}{\mathrm{C}} \quad \begin{array}{l}\text{if } \{x, \neg x\} \not\subseteq (C \cup \{l\}) \text{ and either} \\ \text{(1) } C \text{ is a clause, } q(l) = \forall \text{ and } \forall l' \in C : q(l') = \exists \rightarrow l' < l \text{ or} \\ \text{(2) } C \text{ is a cube, } q(l) = \exists \text{ and } \forall l' \in C : q(l') = \forall \rightarrow l' < l\end{array} \qquad (red)$$

$$\frac{}{\mathrm{C}} \quad \begin{array}{l}\text{if } \{x, \neg x\} \not\subseteq C \text{ and either (1) } C \in \phi \text{ with } \psi = \hat{Q}.\, \phi \text{ or} \\ \text{(2) } \psi[A] = \top \text{ for an assignment } A \text{ and } C = (\bigwedge_{l \in A} l)\end{array} \qquad (init)$$

The rule *res* defines *Q-resolution* with a *pivot variable* p. The constraints $C_1 \cup \{p\}$ and $C_2 \cup \{\neg p\}$ and the resolvent $C_1 \cup C_2$ must not contain complementary literals and the quantifier type $q(p)$ of the pivot variable is restricted to $\exists$ [3].

The rule *red* defines *constraint reduction* [3, 8], which deletes universal (existential) literals from a clause (cube) C which are maximal among the literals in C with respect to the ordering of the quantifier prefix.

The rule *init* defines the axioms. Any clause $C \in \phi$ of a QBF $\psi = \hat{Q}.\, \phi$ can be used as a start point of a resolution derivation. Given an assignment A such that $\psi[A] = \top$, that is $C'[A] = \top$ for all $C' \in \phi$, the cube $C = (\bigwedge_{l \in A} l)$ can be inferred as a start point of a cube resolution derivation. The application of the rule *init* to infer cubes is also called *model generation* [8, 10, 16].

Due to the soundness of the calculus, a derived learned clause C' is added conjunctively to ψ and has the property that $\hat{Q}.\, \phi \equiv \hat{Q}.\, (\phi \wedge C')$. A derived learned cube C' is added disjunctively to ψ and has the property that $\hat{Q}.\, \phi \equiv \hat{Q}.\, (\phi \vee C')$.

A QBF is unsatisfiable (satisfiable) if and only if the empty clause (cube) can be derived using the rules *res*, *red* and *init*. In this case, the steps in the derivations of the learned clauses (cubes) up to $\emptyset$ correspond to a Q-resolution proof of the unsatisfiability (satisfiability) of ψ. We write $\psi \vdash C$ if the clause (cube) C can be derived from C using the rules of the Q-resolution calculus.

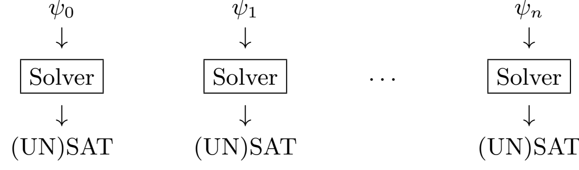


Fig. 1. Solving a sequence $S := \langle \psi_1, \ldots, \psi_n \rangle$ of PCNFs non-incrementally.

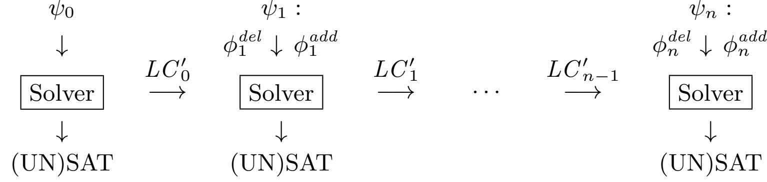


Fig. 2. Solving a sequence $S := \langle \psi_1, \ldots, \psi_n \rangle$ of PCNFs incrementally.

Example 2. Given the satisfiable QBF $\psi = \forall x \exists y. (x \vee \neg y) \wedge (\neg x \vee y)$ from Example 1. By the rule *init*, we generate the cubes $C_1 := (x \wedge y)$ and $C_2 := (\neg x \wedge \neg y)$ using the assignments $A_1 = \{x, y\}$ and $A_2 = \{\neg x, \neg y\}$. Constraint reduction of C_1 and C_2 by rule *red* produces the cubes $C_3 = (x)$ and $C_4 = (\neg x)$, respectively. Finally, resolution by rule *res* of C_3 and C_4 produces the empty cube.

Example 3. Given the unsatisfiable QBF $\psi = \forall x \exists y. (x \vee \neg y) \wedge (\neg x \vee y) \wedge (x \vee y)$. Resolution of the clauses $(x \vee \neg y)$ and $(x \vee y)$ by the rule *res* produces the clause $C_1 := (x)$. Finally, constraint reduction by rule *red* results in the empty clause.

4 Incremental QBF Solving

Let $S := \langle \psi_1, \ldots, \psi_n \rangle$ be a sequence of QBFs to be solved where $\psi_i = \hat{Q}_i. \phi_i$. The QBF $\psi_{i+1} = \hat{Q}_{i+1}. \phi_{i+1}$ is obtained from the previous QBF ψ_i by adding and deleting the sets ϕ_{i+1}^{add} and ϕ_{i+1}^{del} of clauses, respectively: $\phi_{i+1} = (\phi_i \setminus \phi_{i+1}^{del}) \cup \phi_{i+1}^{add}$. Similarly, the quantifier prefix $\hat{Q}_{i+1}$ of ψ_{i+1} is obtained from $\hat{Q}_i$ by adding and deleting quantifiers, provided that in ψ_{i+1} still all the variables are quantified.

In *non-incremental* solving (Fig. 1), the solver tackles each QBF ψ_i in S from scratch. The entire formula is parsed and solving starts without using any learned constraint that was inferred when solving previous QBFs ψ_j with $j < i$.

In *incremental* solving (Fig. 2), the solver retains in a correctness preserving way a subset LC'_{i-1} of the constraints that were learned from previously solved QBFs in S in order to solve the current QBF ψ_i. The constraints in LC'_{i-1} can be used for inferences by the Q-resolution calculus. The choice of the set LC'_{i-1} depends on the sets ϕ_i^{add} and ϕ_i^{del} of clauses that were added to and deleted from the previous QBF ψ_{i-1}, respectively. For all constraints $C \in LC'_{i-1}$, it must hold that C can be derived from ψ_i and hence $\psi_i \vdash C$. Due to the soundness of

Q-resolution, in this case we have that (1) $\hat{Q}_i.\,\phi_i \equiv \hat{Q}_i.\,(\phi_i \wedge C)$ if $C \in LC'_{i-1}$ is a clause and (2) $\hat{Q}_i.\,\phi_i \equiv \hat{Q}_i.\,(\phi_i \vee C)$ if $C \in LC'_{i-1}$ is a cube.

Compared to non-incremental solving, incremental solving has several advantages. First, the solver has to parse only the clauses ϕ_i^{add} which are added to ψ_{i-1} to obtain the new QBF ψ_i rather than the entire QBF ψ_i. In practice, an incremental solver typically is called as a library from another application program which generates the sequence $S := \langle \psi_1, \ldots, \psi_n \rangle$ of QBFs to be solved and retrieves the result returned by the solver. The solver is configured to solve the next QBF in S through its API. In contrast to that, a non-incremental solver is called as a standalone program to solve each QBF in S, where the QBFs are first written to hard disk, accessed by the solver and then parsed. This may result in I/O overhead, which is avoided in incremental solving.

The addition of ϕ_i^{add} to the previous QBF ψ_{i-1} can make the derivations of cubes learned from ψ_{i-1} invalid with respect to the current QBF ψ_i. Similarly, the deletion of ϕ_i^{del} from the previous QBF ψ_{i-1} can make the derivations of learned clauses invalid. The reason is that the side conditions of the rule *init*, which held with respect to ψ_{i-1}, might no longer hold with respect to ψ_i.

Example 4. Let $\psi' = \forall x \exists y.\,(x \vee \neg y) \wedge (\neg x \vee y) \wedge (x \vee y)$ be obtained from the satisfiable QBF ψ in Example 2 by adding the clause $(x \vee y)$. The QBF ψ' is unsatisfiable. Consider the cubes C_1 to C_4 inferred from ψ as shown in Example 2. We have $\psi' \nvdash C_2$ and $\psi' \nvdash C_4$ but $\psi' \vdash C_1$ and $\psi' \vdash C_3$ because the assignment $A_1 = \{x, y\}$ is still a model of ψ' whereas $A_2 = \{\neg x, \neg y\}$ is not.

Example 5. Let $\psi' = \forall x \exists y.\,(x \vee \neg y) \wedge (\neg x \vee y)$ be obtained from the unsatisfiable ψ in Example 3 by deleting the clause $(x \vee y)$. The QBF ψ' is satisfiable. We have $\psi' \nvdash C_1$ and no clauses can be derived from ψ' by the rules *red* and *res*.

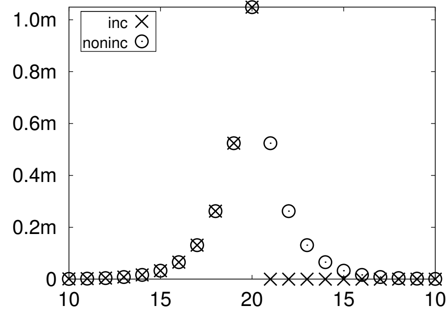
If the set LC'_{i-1} of constraints which is retained contains constraints C for which $\psi_i \nvdash C$ then the solver might perform unsound inferences on ψ_i by the Q-resolution calculus, using the constraints in LC'_{i-1}.

Keeping learned constraints in incremental solving might give speedups in the solving time, as illustrated in the following experiment.

Proposition 1 ([9, 10]). $\psi_n^C := \forall x_1 \exists y_2 \ldots \forall x_{2n-1} \exists y_{2n}.\, \bigwedge_{i=0}^{n-1} [(x_{2i+1} \vee \neg y_{2i+2}) \wedge (\neg x_{2i+1} \vee y_{2i+2})]$ *is a class of* satisfiable *QBFs. For each QBF* ψ_n^C*, the length of the shortest cube resolution proof of satisfiability of* ψ_n^C *is exponential in* n*.*

Let $S := \langle \psi_{10}^C, \ldots, \psi_{20}^C \rangle$ be the sequence of QBFs by Proposition 1 for $n = 10, 11, \ldots, 20$. The left part of the plot on the right shows that the number of learned cubes (y-axis, in millions) carried out by DepQBF when solving S incrementally ("inc") and non-incrementally ("noninc") scales exponentially with the size parameter n ($10 \ldots 20$ on the x-axis). Consider the reversed sequence $S' = \langle \psi_{20}^C, \ldots, \psi_{10}^C \rangle$ and the right part of the plot ($20 \ldots 10$ on the x-axis). When solving S' incrementally, then *all* the cubes learned when solving ψ_i^C in S' can be fully retained and used to solve the next QBF ψ_j^C with $j < i$. No new cubes are inferred. This is possible because clauses are only

deleted from ψ_i^C to obtain ψ_j^C for $j < i$ in S' but not added. Therefore, for all cubes C' derived from ψ_i^C, it holds that $\psi_j^C \vdash C''$ for a subcube $C'' \subseteq C'$. The subcube C'' is obtained from C' by removing any literals which no longer occur in ψ_j^C. For further experiments, we refer to the technical reports related to DepQBF [12] and QBF-based conformant planning by incremental QBF solving [7].



5 Implementation of DepQBF

In incremental solving, the set of learned constraints must be maintained across different calls of the solver. Regarding the learned clauses, the implementation of DepQBF [12] is based on the idea of *selector variables* from incremental SAT solving [6]. Thereby, a *fresh* variable v is added to *each* clause in the QBF $\psi = Q_1B_1 \ldots Q_nB_n.\, \phi$ so that the clause $C \cup \{v\}$ is added to ψ instead of C. The selector variables are existentially quantified in a separate block V' at the left end of the quantifier prefix, which has the form $\exists V' Q_1B_1 \ldots Q_nB_n.\, \phi$. If new clauses are derived using the rule *res* of the Q-resolution calculus, then the selector variables are always transferred to the derived clauses. In order to remove a clause C from the CNF of ψ including all learned clauses derived from C, the solver assigns the selector variable $v \in C$ to *true*. This causes v to be replaced by $\top$ in every clause, which effectively removes the clauses. They can no longer be used to make inferences by the Q-resolution calculus.

Regarding the learned cubes, we keep only cubes derived by applications of the rule *init*. For every cube C which is kept, the side condition of this rule with respect to the *current* QBF ψ' must hold: $\psi'[A] = \top$ for the assignment A which was used to derive $C = (\bigwedge_{l \in A} l)$.

The API of DepQBF provides the user with functions to manipulate the input formula by incrementally adding and deleting clauses and variables. As an additional API feature, the user can add and delete sets of clauses by means of push and pop operations. This way, the set of clauses of the input formula is organized as a sequence of frames on a stack. The same selector variable is added to all clauses of a particular stack frame. As a unique feature, DepQBF maintains the selector variables internally, which are invisible to the user. This design increases the usability of the solver from a user's perspective.

We illustrate the API of DepQBF by the code example in Fig. 3. The source release of DepQBF comes with further examples.[2] A solver object is created using the function `qdpll_create`. We create the quantifier prefix $\forall x \exists y$ by calling `qdpll_new_scope_at_nesting` followed by `qdpll_add` to add the variables x

[2] DepQBF tutorial: `http://lonsing.github.io/depqbf/depqbf-in-practice.pdf`

```
int main (int argc, char ** argv) {
  QDPLL *s = qdpll_create();
  ...
  qdpll_new_scope_at_nesting
    (s,QDPLL_QTYPE_FORALL,1);
  qdpll_add(s,1); qdpll_add(s,0);
  qdpll_new_scope_at_nesting
    (s,QDPLL_QTYPE_EXISTS,2);
  qdpll_add(s,2); qdpll_add(s,0);

  qdpll_add(s,1); qdpll_add(s,-2);
    qdpll_add(s,0);

  qdpll_push(s);
  ...//continues on right column.
```

```
  ...//continued from left column.
  qdpll_add(s,2); qdpll_add(s,0);

  QDPLLResult res = qdpll_sat(s);
  assert(res == QDPLL_RESULT_UNSAT);
  assert(qdpll_get_value (s,1) ==
           QDPLL_ASSIGNMENT_FALSE);

  qdpll_reset(s);
  qdpll_pop(s);

  res = qdpll_sat(s);
  assert(res == QDPLL_RESULT_SAT);

  qdpll_delete (s); }
```

Fig. 3. DepQBF API usage example. Some configuration code was omitted for brevity.

and y which we encode by the unsigned integers `1` and `2`, respectively.[3] Then we add the clauses $(x \vee \neg y)$ by `qdpll_add` where the negative integer `-2` encodes the negative literal $\neg y$ and `qdpll_add(s,0)` closes the clause. A new frame of clauses is allocated by `qdpll_push`. We add the clause (y) to the new frame (right column in Fig. 3). The call of `qdpll_sat` starts the solver given the current QBF $\psi = \forall x \exists y. (x \vee \neg y) \wedge (y)$. The function `qdpll_get_value` returns a partial countermodel of the QBF: x was assigned to *false* which explains the unsatisfiability of ψ since $\psi[\neg x] = (\neg y) \wedge (y)$. By calling `qdpll_pop` we remove the clauses of the most recently added frame, which contains only (y). Thus the new QBF is $\psi = \forall x \exists y. (x \vee \neg y)$, which is satisfiable as found out by `qdpll_sat`.

The API of DepQBF allows to add new variables at any position in the prefix and provides functions to inspect the prefix. Variables and clauses can not explicitly deleted. Instead, a garbage collection phase can be triggered through the API which deletes all the clauses, variables and quantifiers which have been effectively removed by previous calls of `qdpll_pop`. The push/pop functionality of DepQBF is particularly useful for sequences S of QBFs where a large part of the CNFs is shared between the individual QBFs in S.

Originally, DepQBF is written in C. The release of version 3.03 (or later) comes with the Java API *DepQBF4J*, which allows to call DepQBF as a library from Java programs and thus makes incremental QBF solving more accessible.

6 Conclusion

We presented an overview of incremental QBF solving and our incremental QBF solver DepQBF. Incremental solving is useful to solve sequences of related formulae. Information learned from previously solved formulae in terms of derived clauses and cubes can be employed to solve the next formulae in the sequence.

[3] DepQBF takes input in QDIMACS format: `http://www.qbflib.org/qdimacs.html`

We implemented a simple approach to keep only particular cubes derived by model generation across incremental solver calls. As future work, we consider more sophisticated approaches to also keep cubes derived by Q-resolution.

Another important direction is the combination of incremental QBF solving with advanced techniques such as preprocessing and the generation of proofs and certificates. Currently, these techniques are implemented in separate tools. It is necessary to efficiently integrate them into a uniform framework to leverage the full power of the state of the art of QBF reasoning in practical applications.

References

1. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. TACAS*, volume 1579 of *LNCS*. Springer, 1999.
2. U. Bubeck and H. Kleine Büning. Encoding Nested Boolean Functions as Quantified Boolean Formulas. *JSAT*, 8(1/2):101–116, 2012.
3. H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for Quantified Boolean Formulas. *Inf. Comput.*, 117(1):12–18, 1995.
4. M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An Algorithm to Evaluate Quantified Boolean Formulae and Its Experimental Evaluation. *J. Autom. Reasoning*, 28(2):101–142, 2002.
5. M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
6. N. Eén and N. Sörensson. Temporal Induction by Incremental SAT Solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.
7. U. Egly, M. Kronegger, F. Lonsing, and A. Pfandler. Conformant Planning as a Case Study of Incremental QBF Solving. *CoRR (submitted)*, 2014.
8. E. Giunchiglia, M. Narizzano, and A. Tacchella. Clause/Term Resolution and Learning in the Evaluation of Quantified Boolean Formulas. *J. Artif. Intell. Res. (JAIR)*, 26:371–416, 2006.
9. M. Janota, R. Grigore, and J. Marques-Silva. On QBF Proofs and Preprocessing. In *Proc. LPAR*, volume 8312 of *LNCS*. Springer, 2013.
10. R. Letz. Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In *Proc. TABLEAUX*, volume 2381 of *LNCS*. Springer, 2002.
11. F. Lonsing and A. Biere. DepQBF: A Dependency-Aware QBF Solver. *JSAT*, 7(2-3):71–76, 2010.
12. F. Lonsing and U. Egly. Incremental QBF Solving. *CoRR*, abs/1402.2410, 2014.
13. F. Lonsing, U. Egly, and A. Van Gelder. Efficient Clause Learning for Quantified Boolean Formulas via QBF Pseudo Unit Propagation. In *Proc. SAT*, volume 7962 of *LNCS*. Springer, 2013.
14. P. Marin, C. Miller, M. D. T. Lewis, and B. Becker. Verification of Partial Designs using Incremental QBF Solving. In *Proc. DATE*. IEEE, 2012.
15. J. P. Marques Silva, I. Lynce, and S. Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, volume 185 of *FAIA*. IOS Press, 2009.
16. L. Zhang and S. Malik. Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In *Proc. CP*, volume 2470 of *LNCS*. Springer, 2002.