Incremental Determinization for Quantifier Elimination and Functional Synthesis

Markus N. Rabe

Google
mrabe@google.com

Abstract. Quantifier elimination and its cousin functional synthesis are fundamental problems in automated reasoning that could be used in many applications of formal methods. But, effective algorithms are still elusive. In this paper, we suggest a simple modification to a QBF algorithm to adapt it for quantifier elimination and functional synthesis. We demonstrate that the approach significantly outperforms previous algorithms for functional synthesis.

1 Introduction

Given a Boolean formula $\exists Y.\, \varphi$ with free variables X, *quantifier elimination* (also called *projection*) is the problem to find a formula $\psi \equiv \exists Y.\, \varphi$ that only contains variables X. Closely related, the *functional synthesis* problem is to find a function $f_y : 2^X \to \mathbb{B}$ for all $y \in Y$, such that $\varphi[Y \mapsto f_y(X)] \equiv \exists Y.\, \varphi$.

Quantifier elimination and functional synthesis are fundamental operations in automated reasoning, computer-aided design, and verification. Hence, progress in algorithms for these problems benefits a broad range of applications of formal methods. For example, typical algorithms for reactive synthesis reduce to computing the safe region of a safety game through repeated quantifier eliminations [1–3] or directly employ functional synthesis [4]. Until today, algorithms for quantifier elimination often involve (reduced ordered) Binary Decision Diagrams (BDDs) [5]. However, BDDs often grow exponentially for applications in verification, and extracting formulas (or strategies, etc.) from BDDs typically results in huge expressions. The search for alternatives resulted in CEGAR-style algorithms [6–10].

In this work, we take look at the closely related field of QBF solving. There pure CEGAR solving [11–13] on the CNF representation is not competitive anymore [14], and it has been augmented by preprocessing [15, 16], circuit representations [17–21], and Incremental Determinization (ID) [22]. It may hence be fruitful to leverage some of the recent developments of QBF.

The contribution of this work is a simple modification of ID to enable quantifier elimination and functional synthesis. Incremental Determinization (ID) is an algorithm for solving quantified Boolean formulas of the shape $\forall X. \exists Y.\, \varphi$, where φ is a propositional formula in conjunctive normal form (CNF), i.e. 2QBF. It

Work partially done at University of California at Berkeley.

follows a proof-theoretic approach, very similar to a SAT solver, alternating between building a model (i.e. Skolem functions for the existential variables Y) and a refutation proof [23]. This allows ID to provide a model (i.e. a Skolem function) when it determines that a formula is true, which sets it apart from other QBF algorithms.

The modification of ID to enable quantifier elimination for a given formula $\exists Y.\, \varphi$ is very simple: We run ID on the formula as if it was a quantified Boolean formula $\forall X.\, \exists Y.\, \varphi$, where X are the free variables, but add φ to the conflict check within ID. This suppresses the UNSAT result in the ID algorithm and it is hence forced to terminate with a model (that is, a function), which is guaranteed to satisfy the functional synthesis requirements. Quantifier elimination is then only a substitution away.

Our experimental evaluation shows that ID significantly outperforms previous algorithms for functional synthesis and quantifier elimination.

This paper is structured as follows: We review related work in Section 2 and introduce standard notation in Section 3. In Section 4 we first review the Incremental Determinization algorithm before introducing the change necessary to lift it to functional synthesis. The experimental evaluation is in Section 5. We summarize the current state of the tool CADET in Section 6 and conclude the paper in Section 7.

2 Related Work

Functional Synthesis. Early works on functional synthesis tried to exploit Craig interpolation, but did not scale well enough [24]. This was followed by first attempts to use CEGAR [6], which failed, however, to surpass the performance of BDDs [7]. More recent works revisited the use of BDDs, e.g. the tools SSyft [25] and RSynth [26, 27]. This motivated the search for alternatives to BDDs [8–10]. At their core, these new algorithms all rely on counter-example guided abstraction refinement (CEGAR) [28], but they apply it in clever, compositional ways. However, they still inherit the well-known weaknesses of CEGAR (as, for example, discussed in the QBF literature): For the simple formula $\varphi = \bigwedge_{i<n} x_i \leftrightarrow y_i$, where $n = |X| = |Y|$ and $x_i \in X$ and $y_i \in Y$, CEGAR needs to browse through 2^n satisfying assignments just to recover that the function we were looking for is $f(x) = x$.

The Back-and-Forth algorithm explores stronger abstraction using MaxSAT solvers as a means to reduce the number of assignments that CEGAR needs to explore [8]. ParSyn attempts to combat the problem with parallel compute power and a compositional approach [9]. This compositional approach has later been refined using a wDNNF decomposition [10].

QBF Certification. Some solvers and preprocessors for QBF have the ability to not only provide a yes/no answer, but also produce a certificate (i.e. Skolem functions) for their result [13, 22, 29, 30]. While most QBF approaches suffer heavy

performance penalties when asked to provide a certificate, Incremental Determinization naturally computes Skolem functions that can be extracted easily from the final state [22].

3 Preliminaries

Boolean formulas over a finite set of variables $x \in X$ with domain $\mathbb{B} = \{\mathbf{0}, \mathbf{1}\}$ are generated by the following grammar:

$$\varphi := \mathbf{0} \mid \mathbf{1} \mid x \mid \neg\varphi \mid (\varphi) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi$$

Other logical operations, such as implication, XOR, and equality, are considered syntactic sugar with the usual definitions.

An *assignment* $\boldsymbol{x}$ to a set of variables X is a function $\boldsymbol{x} : X \to \mathbb{B}$ that maps each variable $x \in X$ to either $\mathbf{1}$ or $\mathbf{0}$. We denote the space of assignments to some set of variables X with 2^X.

Given formulas φ and φ', and a variable x, we denote the substitution of x by φ' in φ as $\varphi[x \to \varphi']$. We lift substitutions to sets of variables $\varphi[X \mapsto t_x]$ when t_x maps each $x \in X$ to a formula φ'.

A *literal* l is either a variable $x \in X$, or its negation $\neg x$. We use $\bar{l}$ to denote the literal that is the logical negation of l. A disjunction of literals $(l_1 \vee \ldots \vee l_n)$ is called a *clause* and their conjunction $(l_1 \wedge \ldots \wedge l_n)$ is called a *cube*. We denote the variable of a literal by $var(l)$ and lift the notion to clauses $var(l_1 \vee \cdots \vee l_n) = \{var(l_1), \ldots, var(l_n)\}$.

A formula is in *conjunctive normal form* (CNF), if it is a conjunction of clauses. Throughout this exposition, we assume that the input formula is given in CNF. (The output, however, can be a non-CNF formula.) It is trivial to lift the approach to general Boolean formulas: Given a Boolean formula φ over variables X, the Tseitin transformation provides us a formula ψ with $\varphi \equiv \exists Z.\psi$, where Z are fresh variables [31]. Note that eliminating a group of variables $X' \subseteq X$ in φ is then the same as eliminating $X' \cup Z$ in ψ.

Resolution is a well-known proof rule that allows us to merge two clauses as follows. Given two clauses $C_1 \vee v$ and $C_2 \vee \neg v$, we call $C_1 \otimes_v C_2 = C_1 \vee C_2$ their *resolvent* with pivot v. The resolution rule states that $C_1 \vee v$ and $C_2 \vee \neg v$ imply their resolvent. Resolution is *refutationally complete* for Boolean formulas in CNF, i.e. given a formula in CNF that is equivalent to false, we can derive the empty clause using only resolution.

4 Lifting Incremental Determinization

In the sequel, we formally define functional synthesis, review the working principle of Incremental Determinization for 2QBF, discuss how the solver state corresponds to functions, and then introduce the modification to Incremental Determinization to turn it into an algorithm for functional synthesis. The *functional synthesis* problem is to find a function $f_y : 2^X \to \mathbb{B}$ for all $y \in Y$, such

that $\varphi[Y \mapsto f_y(X)] \equiv \exists Y.\, \varphi$. Functional synthesis is closely related to solving 2QBF: Given a true 2QBF problem $\forall X.\, \exists Y.\, \varphi$, any Skolem function that is a model for the formula is also a solution to the functional synthesis problem for variable sets X and Y. Only for false 2QBF there is a difference between the problems: if there is an assignment $\boldsymbol{x}$ to X for which there is no assignment to Y, the 2QBF cannot be proven with a Skolem function, but the functional synthesis problem still requires us to produce a function f. It is clear that for input $\boldsymbol{x}$ the f can produce any output. We will exploit this similarity between 2QBF and functional synthesis in the following to lift the Incremental Determinization algorithm to functional synthesis.

4.1 Working Principle of Incremental Determinization for 2QBF

ID was originally introduced as an algorithm for 2QBF, the fragment of quantified Boolean formulas with at most one quantifier alternation. Given a formula $\forall X.\, \exists Y.\, \varphi$, ID alternates between constructing a model (i.e. a Skolem function) to prove the formula correct, and constructing a Q-resolution proof to refute the formula [32]. During model construction, ID identifies which variables in Y have unique Skolem functions considering the current set of clauses. When all variables with unique Skolem functions are identified, ID greedily introduces additional clauses to reduce the space of possible Skolem functions, such that the remaining variables may get unique Skolem functions, too. Whenever the model construction ends up in a dead-end (=conflict), ID switches to constructing a refutation proof [32] and derives clauses using resolution. As soon as ID found a clause that prevents the model construction from trying the same partial model again, it switches back to the model search. Since there are only finitely many clauses and models, either the model construction or the refutation proof must eventually finish [22, 23].

Example 1. We will use the following formula as a running example:

$$\begin{aligned}\forall x_1, x_2.\, \exists y_1, y_2, y_3.\ & (x_1 \vee \neg y_1) \wedge (\neg x_1 \vee y_1) \wedge \\ & (y_1 \vee \neg y_2) \wedge (\neg y_1 \vee \neg x_2 \vee y_2) \wedge \\ & (\neg y_1 \vee y_3) \wedge (y_2 \vee \neg y_3) \wedge (x_2 \vee \neg y_3)\end{aligned}$$

Looking at the first two clauses it is clear that y_1 is uniquely determined by x_1 and y_1's Skolem function must be $f_{y_1}(X) = x_1$. For this step, we intentionally ignore all clauses of y_1 that contain y_2 and y_3, as they do not yet have a Skolem function and we have to consider them as undefined. The other clauses containing y_1 will only become relevant when looking for Skolem functions for y_2 and y_3.

Variables y_2 and y_3 do not have *unique* Skolem functions in the formula above. ID would now greedily add a *decision clause*, such as $(x_2 \vee \neg y_2)$, to also make the Skolem function for y_2 unique. The added clause, plus clauses 3 and 4 in the formula define: $f_{y_2}(X) = f_{y_1}(X) \wedge x_2$.

This results in the situation that there is no Skolem function for y_3: For the assignment $x_1 \mapsto \mathbf{1},\ x_2 \mapsto \mathbf{0}$, the functions for y_1 and y_2 assign $y_1 \mapsto \mathbf{1},\ y_2 \mapsto \mathbf{0}$.

Then clauses 4 and 5 cannot be satisfied both by y_3, which means there is a conflict for this assignment to the universals. During conflict analysis, ID would now resolve clauses 5 and 6 to obtain clause $(\neg y_1 \vee y_2)$, and then backtrack to the point before introducing the decision clause. $\triangleleft$

4.2 Representation of Functions

What is particularly interesting about ID is its ability to produce Skolem functions when it has proven a formula correct. Other than previous QBF algorithms, these Skolem functions are produced without any overhead.

ID avoids costly representations of Skolem functions: It maintains a set $D \subseteq Y$ of variables that have a unique Skolem function, and its state includes a formula δ characterizing the input-output behavior of the Skolem functions for variables D. Formula δ satisfies $\forall X. \exists! D. \delta$, where $\exists! D$ means that there exists exactly one assignment to D. We can thus think of δ also as a function f_δ mapping X assignments to D assignments.

Example 2. Back to our running example. After identifying a unique Skolem function for y_1, formula δ consists exactly of the first two clauses of the formula, $(x_1 \vee \neg y_1) \wedge (\neg x_1 \vee y_1)$. After adding the decision clause and identifying a unique Skolem function for y_2, δ consists exactly of the first four clauses and the decision clause. $\triangleleft$

4.3 Conflict Checks in ID

The formulas representing functions have primarily one purpose: to check for the existence of *conflicts*. Whenever we attempt to grow the set D by a variable v, we need to check whether v has a unique Skolem function. This check consists of two parts; given an arbitrary universal assignment $\boldsymbol{x} \in 2^X$,

(1) is there *at most* one legal assignment to v, and
(2) is there *at least* one legal assignment to v?

To formally define this, let us consider the clauses $(d_1 \vee \cdots \vee d_n \vee l)$ in φ that contain a literal l of variable v and otherwise only contain literals d_i of variables in D and X. We call these the clauses with *unique consequence*, as they can be read as implications $(\neg d_1 \wedge \cdots \wedge \neg d_n \Rightarrow l)$, and we call $\neg d_1 \wedge \cdots \wedge \neg d_n$ the antecedent of that clause. Further, we define $\mathcal{A}_l$ as the disjunction over all antecedents of literal l. (Note that $\mathcal{A}_l$ depends on D and therefore changes as the state of the solver progresses.)

The two checks from above can now be defined as follows:

(1) $\exists X. \delta \wedge \neg \mathcal{A}_v \wedge \neg \mathcal{A}_{\neg v}$
(2) $\exists X. \delta \wedge \quad \mathcal{A}_v \wedge \quad \mathcal{A}_{\neg v}$

Checking for case (1) can be efficiently approximated [22], but checking for case (2) cannot easily be avoided. We thus query a SAT solver with $\delta \wedge \mathcal{A}_v \wedge \mathcal{A}_{\neg v}$ to perform a conflict check.

Example 3. We revisit the conflict described in Example 1. The starting point is the situation when $D = \{y_1, y_2\}$ and δ consists of the first four clauses of the formula as well as the decision clause $(x_2 \vee \neg y_2)$. The antecedents of y_3 are $\mathcal{A}_{y_3} = y_1$ and $\mathcal{A}_{\neg y_3} = \neg y_2 \vee \neg x_2$. It is easy to verify that the universal assignment $x_1 \mapsto \mathbf{1},\ x_2 \mapsto \mathbf{0}, y_1 \mapsto \mathbf{1},\ y_2 \mapsto \mathbf{0}$ satisfies the conflict criterion $\delta \wedge \mathcal{A}_v \wedge \mathcal{A}_{\neg v}$. $\lhd$

4.4 Functional Synthesis

Remember that in the case of functional synthesis for φ over sets of variables X and Y, we search for a function $f : 2^X \to 2^Y$ such that f produces a satisfying assignment whenever it can, but can produce anything when there is no assignment to Y satisfying the formula. In case there are satisfying assignments to Y for all X, we can simply run ID as if it was a QBF $\forall X. \exists. \varphi$ to obtain a Skolem function that also satisfies the functional synthesis criterion. In the other case, that there is an X for which there is no assignment to Y satisfying φ, ID for 2QBF would eventually detect a conflict that did not depend on a decision and return with UNSAT.

In order to lift ID to functional synthesis, we want to ignore universal assignments that have no satisfying assignment to Y. A simple way to suppress these conflicts is to add φ to the conflict check. In order for an assignment to X to remain a conflict, we must now additionally find an assignment to Y that demonstrates that the conflict could be prevented by a different decision.

All other parts of ID, including the extraction of functions, remain untouched. In particular, termination is still guaranteed, as the greedy model construction either results in a function for all variables in Y or in a conflict, upon which at least one model is excluded through resolution.

Example 4. For the conflict in our running example, the universal assignment $x_1 \mapsto \mathbf{1},\ x_2 \mapsto \mathbf{0}$ is excluded in the modified conflict check. Consider the UNSAT core consisting of clauses 2, 5, and 7 for that universal assignment: propagate $y_1 \mapsto \mathbf{1}$ using clause 2; propagate $y_3 \mapsto \mathbf{1}$ using clause 5; and finally propagate $y_3 \mapsto \mathbf{0}$ using clause 7. So, instead of going into conflict analysis and backtracking, ID for functional synthesis concludes that it has found a function for all existential variables and terminates.

4.5 Quantifier Elimination

Given a formula $\exists Y. \varphi$ with free variables X, *quantifier elimination* is the problem to find a formula $\psi \equiv \exists Y. \varphi$ over variables X only. Hence, given a solution f to the functional synthesis problem for φ, we only have to substitute Y by f in φ to obtain the projected formula.

5 Experimental Evaluation

We implemented the modifications to ID in CADET,[1] a competitive 2QBF solver [22]. In this section, we compare CADET experimentally with existing al-

[1] CADET is available at `https://github.com/MarkusRabe/cadet`

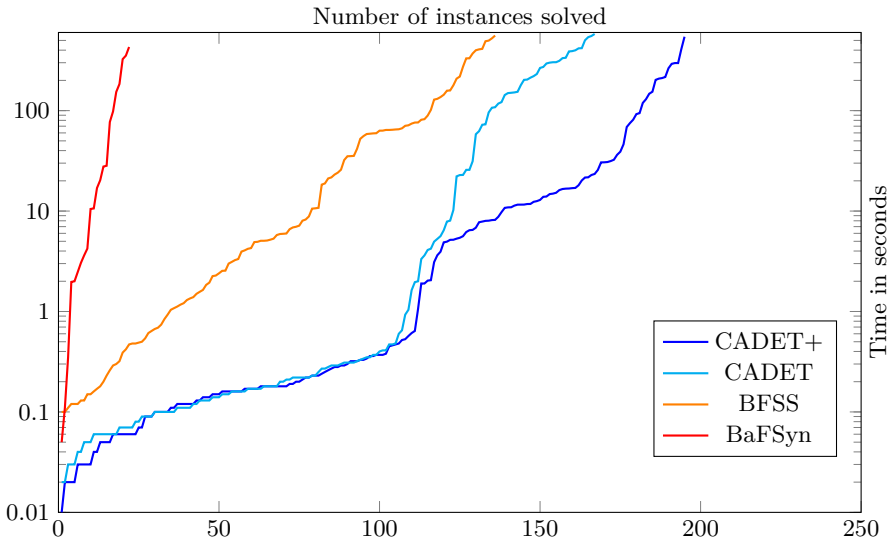

Fig. 1. Log-scale cactus plot comparing the performance over all instances.

gorithms for functional synthesis. Additionally, we implemented a certificate checker for functional synthesis and for quantifier elimination, to make sure that the computed functions are correct. The certificate checker only shares the code for AIGER circuits and the SAT solver (of which we have tried several), but is completely independent otherwise to reduce the chance of correlated bugs. The results of CADET have been checked with the proof checker; running times reported below are excluding the time to check the certificates.

So far, there is no standard benchmark for functional synthesis or quantifier elimination. Like previous works on functional synthesis, we resort to using the 2QBF benchmark from QBFEVAL'17 [14], and re-interpret them as functional synthesis problems. The 2QBF benchmark from QBFEVAL'17 is a collection of 384 formulas from various domains, mostly from software verification, program synthesis, and logical equivalences [33–36].

We compare CADET to the most recent tools on functional synthesis, BaFSyn [8] and BFSS [10], the latter of which has been shown to consistently outperform the earlier, BDD-based tools SSyft [25] and RSynth [26, 27]. We ran CADET in two configurations: with (CADET+) and without (CADET) its CEGAR module [23]. We present the results as a cactus plot, which is obtained by running each tool on all formulas, sorting the running times for each tool separately. A point x, y in this plot means that x formulas were solved in less than time y. Note that the time axis is in log-scale.

CADET shows a clear edge in performance: it is one to two orders of magnitude faster than its strongest competitor, BFSS, and can solve significantly more formulas. But despite the clear performance advantage in this aggregate

view, BaFSyn and BFSS can be faster for individual formulas or subfamilies of QBFEval, as shown in previous works [8, 10].

6 The Current State of CADET

Originally designed as an experimentation platform, CADET has grown to become a performant and versatile tool for the synthesis of Boolean functions. It consistently wins awards at the annual QBFEVAL competitions, and is the only such tool able to prove all its results [14].

CADET reads specifications in the QDIMACS and the QAIGER formats, and now supports the synthesis of Boolean functions for 2QBF, functional synthesis, and quantifier elimination with the command line options `-c [file]`, `-f [file]`, and `-e [file]`. The functions computed by CADET are much smaller compared to those found by CEGAR-based algorithms [22], and in its default configuration, CADET double-checks its results before reporting them. This can be deactivated by the flag `--dontverify`.

It has also been integrated in py-aiger [37], a Python package for the convenient handling of circuits due to Marcell Vazquez-Chanlatte, which enables us to easily model and prototype new approaches. For example, we can write:

```
import aiger_analysis as aa
import aigerbv as bv
x = bv.atom(32, 'x')  # Create a 32 bit variable
y = bv.atom(32, 'y')
expr = (x != y)
result = aa.eliminate(expr, ['y'])
assert aa.is_equal(x, result)
```

CADET also has an experimental reinforcement learning interface that allows us to automatically learn decision heuristics with the help of graph neural networks. A recent effort shows that there is huge potential in learning better branching heuristics from scratch [38].

7 Conclusions

In this work, we extended ID with the ability to solve functional synthesis and quantifier elimination problems. The extension is very simple—we only need to add the clauses of the original formula to its conflict check. The resulting algorithm significantly outperforms previous algorithms for functional synthesis.

Acknowledgements The author wants to thank to Shubham Goel, Shetal Shah, and Lucas Tabajara for insightful discussions and for their assistance with running their functional synthesis tools. In particular, I want to express my gratitude to Supratik Chakraborty for inspiring me to work on the topic in a discussion in the summer of 2016.

References

1. R. Ehlers, "Symbolic bounded synthesis," in *Proceedings of CAV*, T. Touili, B. Cook, and P. Jackson, Eds., Berlin, Heidelberg, 2010, pp. 365–379.
2. R. Brenguier, G. A. Pérez, J. Raskin, and O. Sankur, "Abssynthe: abstract synthesis from succinct safety specifications," in *Proceedings of SYNT*, 2014, pp. 100–116.
3. S. Jacobs, N. Basset, R. Bloem, R. Brenguier, M. Colange, P. Faymonville, B. Finkbeiner, A. Khalimov, F. Klein, T. Michaud *et al.*, "The 4th reactive synthesis competition (syntcomp 2017): Benchmarks, participants & results," *arXiv preprint arXiv:1711.11439*, 2017.
4. S. Zhu, L. M. Tabajara, J. Li, G. Pu, and M. Y. Vardi, "Symbolic LTLf synthesis," in *Proceedings of IJCAI*, ser. IJCAI'17. AAAI Press, 2017, pp. 1362–1369.
5. R. E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," *ACM Comput. Surv.*, vol. 24, no. 3, pp. 293–318, Sep. 1992.
6. E. Goldberg and P. Manolios, "Quantifier elimination by dependency sequents," *Formal Methods in System Design*, vol. 45, no. 2, pp. 111–143, Oct 2014. [Online]. Available: https://doi.org/10.1007/s10703-014-0214-z
7. E. Goldberg and P. Manolios, "Quantifier elimination via clause redundancy," in *2013 Formal Methods in Computer-Aided Design*, Oct 2013, pp. 85–92.
8. S. Chakraborty, D. Fried, L. M. Tabajara, and M. Y. Vardi, "Functional synthesis via input-output separation," in *Proceedings of FMCAD*. IEEE, 2018, pp. 1–9.
9. S. Akshay, S. Chakraborty, A. K. John, and S. Shah, "Towards parallel Boolean functional synthesis," in *Proceedings of TACAS*. Springer, 2017, pp. 337–353.
10. S. Akshay, S. Chakraborty, S. Goel, S. Kulal, and S. Shah, "What's hard about Boolean functional synthesis?" in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds., 2018, pp. 251–269.
11. M. Janota, W. Klieber, J. Marques-Silva, and E. M. Clarke, "Solving QBF with counterexample guided refinement." in *Proceedings of SAT*, 2012, pp. 114–128.
12. M. Janota and J. Marques-Silva, "Solving QBF by clause selection," in *Proceedings of IJCAI*. AAAI Press, 2015, pp. 325–331.
13. M. N. Rabe and L. Tentrup, "CAQE: A certifying QBF solver," in *Proceedings of FMCAD*, 2015, pp. 136–143.
14. "QBFEVAL: QBF solver evaluation portal," http://www.qbflib.org/index_eval.php, accessed: Jan 2018.
15. A. Biere, F. Lonsing, and M. Seidl, "Blocked clause elimination for QBF," in *Proceedings of CADE*, 2011, pp. 101–115.
16. R. Wimmer, S. Reimer, P. Marin, and B. Becker, "Hqspre–an effective preprocessor for qbf and dqbf," in *Proceedings of TACAS*, 2017.
17. W. Klieber, S. Sapra, S. Gao, and E. Clarke, "A non-prenex, non-clausal QBF solver with game-state learning," in *Proceedings of SAT*. Springer, 2010, pp. 128–142.
18. C. Jordan, W. Klieber, and M. Seidl, "Non-CNF QBF solving with QCIR," in *AAAI Workshop: Beyond NP*, 2016.
19. V. Balabanov, J.-H. R. Jiang, C. Scholl, A. Mishchenko, and R. K. Brayton, "2QBF: Challenges and solutions," in *Proceedings of SAT*. Springer, 2016, pp. 453–469.
20. L. Tentrup, "Non-prenex QBF solving using abstraction," in *Proceedings of SAT*, N. Creignou and D. Le Berre, Eds. Cham: Springer International Publishing, 2016, pp. 393–401.

21. M. Janota, "Circuit-based search space pruning in QBF," in *Proceedings of SAT*, O. Beyersdorff and C. M. Wintersteiger, Eds. Cham: Springer International Publishing, 2018, pp. 187–198.
22. M. N. Rabe and S. A. Seshia, "Incremental determinization," in *Proceedings of SAT*. Berlin, Heidelberg: Springer-Verlag, 2016.
23. M. N. Rabe, L. Tentrup, C. Rasmussen, and S. A. Seshia, "Understanding and extending incremental determinization for 2QBF," in *Proceedings of CAV*, 2018, pp. 256–274.
24. J.-H. R. Jiang, "Quantifier elimination via functional composition," in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 383–397.
25. S. Zhu, L. M. Tabajara, J. Li, G. Pu, and M. Y. Vardi, "A symbolic approach to safety LTL synthesis," in *Haifa Verification Conference (HVC)*. Springer, 2017, pp. 147–162.
26. L. M. Tabajara and M. Y. Vardi, "Factored Boolean functional synthesis," in *Proceedings of FMCAD*. IEEE, 2017, pp. 124–131.
27. D. Fried, L. M. Tabajara, and M. Y. Vardi, "BDD-based Boolean functional synthesis," in *Proceedings of CAV*. Springer, 2016, pp. 402–421.
28. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proceedings of CAV*, 2000, pp. 154–169.
29. F. Lonsing and A. Biere, "DepQBF: A dependency-aware QBF solver," *JSAT*, vol. 7, no. 2-3, pp. 71–76, 2010.
30. M. Heule, M. Seidl, and A. Biere, "A unified proof system for QBF preprocessing," in *Proceedings of IJCAR*, ser. LNCS, vol. 8562. Springer, 2014, pp. 91–106.
31. G. S. Tseitin, "On the complexity of derivation in propositional calculus," *Studies in constructive mathematics and mathematical logic, Reprinted in [?]*, vol. 2, no. 115-125, pp. 10–13, 1968.
32. H. Buning, M. Karpinski, and A. Flogel, "Resolution for quantified Boolean formulas," *Information and Computation*, vol. 117, no. 1, pp. 12 – 18, 1995.
33. A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu, "Programming by sketching for bit-streaming programs," in *Proceedings of PLDI*, 2005, pp. 281–294.
34. B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger, "Ranking function synthesis for bit-vector relations." in *Proceedings of TACAS*, 2010, pp. 236–250.
35. C. M. Wintersteiger, Y. Hamadi, and L. De Moura, "Efficiently solving quantified bit-vector formulas," *Proceedings of FMSD*, vol. 42, no. 1, pp. 3–23, 2013.
36. C. Jordan and Ł. Kaiser, "Experiments with reduction finding," in *Proceedings of SAT*. Springer, 2013, pp. 192–207.
37. M. Vazquez-Chanlatte, "mvcisback/py-aiger," Aug. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1326224
38. G. Lederman, M. N. Rabe, E. A. Lee, and S. A. Seshia, "Learning heuristics for automated reasoning through deep reinforcement learning," *arXiv preprint arXiv:1807.08058*, 2018.