Counterexample-Guided Model Synthesis

Mathias Preiner(✉), Aina Niemetz, and Armin Biere

Johannes Kepler University, Linz, Austria
mathias.preiner@jku.at

Abstract. In this paper we present a new approach for solving quantified formulas in Satisfiability Modulo Theories (SMT), with a particular focus on the theory of fixed-size bit-vectors. We combine counterexample-guided quantifier instantiation with a syntax-guided synthesis approach, which allows us to synthesize both Skolem functions and terms for quantifier instantiations. Our approach employs two ground theory solvers to reason about quantified formulas. It neither relies on quantifier specific simplifications nor heuristic quantifier instantiation techniques, which makes it a simple yet effective approach for solving quantified formulas. We implemented our approach in our SMT solver Boolector and show in our experiments that our techniques are competitive compared to the state-of-the-art in solving quantified bit-vectors.

1 Introduction

Many techniques in hardware and software verification rely on quantifiers for describing properties of programs and circuits, e.g., universal safety properties, inferring program invariants [1], finding ranking functions [2], and synthesizing hardware and software [3,4]. Quantifiers further allow to define theory axioms to reason about a theory of interest not supported natively by an SMT solver.

The theory of fixed-size bit-vectors provides a natural way of encoding bit-precise semantics as found in hardware and software. In recent SMT competitions, the division for quantifier-free fixed-size bit-vectors was the most competitive with an increasing number of participants every year. Quantified bit-vector reasoning, however, even though a highly required feature, is still very challenging and did not get as much attention as the quantifier-free fragment. The complexity of deciding quantified bit-vector formulas is known to be NExpTime-hard and solvable in ExpSpace [5]. Its exact complexity, however, is still unknown.

While there exist several SMT solvers that efficiently reason about quantifier-free bit-vectors, only CVC4 [6], Z3 [7], and Yices [8] support the quantified bit-vector fragment. The SMT solver CVC4 employs counterexample-guided quantifier instantiation (CEGQI) [9], where a ground theory solver tries to find concrete values (counterexamples) for instantiating universal variables by generating models of the negated input formula. In Z3, an approach called model-based quantifier instantiation (MBQI) [10] is combined with a model finding procedure

Supported by Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

A. Legay and T. Margaria (Eds.): TACAS 2017, Part I, LNCS 10205, pp. 264–280, 2017.
DOI: 10.1007/978-3-662-54577-5_15

based on templates [11]. In contrast to only relying on concrete counterexamples as candidates for quantifier instantiation, MBQI additionally uses symbolic quantifier instantiation to generalize the counterexample by selecting ground terms to rule out more spurious models. The SMT solver Yices provides quantifier support limited to exists/forall problems [12] of the form $\exists \mathbf{x} \forall \mathbf{y} . P[\mathbf{x}, \mathbf{y}]$. It employs two ground solver instances, one for checking the satisfiability of a set of generalizations and generating candidate solutions for the existential variables $\mathbf{x}$, and the other for checking if the candidate solution is correct. If the candidate model is not correct, a model-based generalization procedure refines the candidate models.

Recently, a different approach based on binary decision diagrams (BDD) was proposed in [13]. Experimental results of its prototype implementation Q3B show that it is competitive with current state-of-the-art SMT solvers. However, employing BDDs for solving quantified bit-vectors heavily relies on formula simplifications, variable ordering, and approximation techniques to reduce the size of the BDDs. If these techniques fail to substantially reduce the size of the BDDs this approach does not scale. Further, in most applications it is necessary to provide models in case of satisfiable problems. However, it is unclear if a bit-level BDD-based model can be lifted to produce more succinct word-level models.

In this paper, we combine a variant of CEGQI with a syntax-guided synthesis [14] approach to create a model finding algorithm called *counterexample-guided model synthesis* (CEGMS), which iteratively refines a synthesized candidate model. Unlike Z3, our approach synthesizes Skolem functions based on a set of ground instances without the need for specifying function or circuit templates up-front. Further, we can apply CEGMS to the negation of the formula in a parallel dual setting to synthesize quantifier instantiations that prove the unsatisfiability of the original problem. Our approach is a simple yet efficient technique that does not rely on quantifier specific simplifications, which have previously been found to be particularly useful [11]. Our experimental evaluation shows that our approach is competitive with the state-of-the-art in solving quantified bit-vectors. However, even though we implemented it in Boolector, an SMT solver for the theory of bit-vectors with arrays and uninterpreted functions, our techniques are not restricted to the theory of quantified bit-vectors.

2 Preliminaries

We assume the usual notions and terminology of first-order logic and primarily focus on the theory of *quantified fixed-size bit-vectors*. We only consider many-sorted languages, where bit-vectors of different size are interpreted as bit-vectors of different sorts.

Let Σ be a signature consisting of a set of function symbols $f : n_1, \ldots, n_k \rightarrow n$ with arity $k \geq 0$ and a set of bit-vector sorts with size $n, n_1, \ldots, n_k$. For the sake of simplicity and w.l.o.g., we assume that sort *Bool* is interpreted as a bit-vector of size one with constants $\top$ (1) and $\bot$ (0), and represent all predicate symbols as function symbols with a bit-vector of size one as the sort of the co-domain. We refer to function symbols occurring in Σ as *interpreted*, and those symbols not included in Σ as *uninterpreted.* A *bit-vector term* is either a bit-vector variable or an application of a bit-vector function of the form $f(t_1, \ldots, t_k)$, where $f \in \Sigma$ or $f \notin \Sigma$, and $t_1, \ldots, t_k$ are bit-vector terms. We denote bit-vector term t of size n as $t_{[n]}$ and define its *domain* as $\mathcal{BV}_{[n]}$, which consists of all *bit-vector values* of size n. Analogously, we represent a bit-vector value as an integer with its size as a subscript, e.g., $1_{[4]}$ for 0001 or $-1_{[4]}$ for 1111.

We assume the usual interpreted symbols for the theory of bit-vectors, e.g., $=_{[n]}, +_{[n]}, *_{[n]}, \mathit{concat}_{[n+m]}, <_{[n]}$, etc., and will omit the subscript specifying their bit-vector size if the context allows. We further interpret an $\mathit{ite}(c, t_0, t_1)$ as an *if-then-else* over bit-vector terms, where $\mathit{ite}(\top, t_0, t_1) = t_0$ and $\mathit{ite}(\bot, t_0, t_1) = t_1$.

In general, we refer to 0-arity function symbols as *constant* symbols, and denote them by a, b, and c. We use f and g for non-constant function symbols, P for predicates, x, y and z for variables, and t for arbitrary terms. We use symbols in bold font, e.g., $\mathbf{x}$, as a shorthand for tuple $(x_1, \ldots, x_k)$, and denote a formula (resp. term) that may contain variables $\mathbf{x}$ as $\varphi[\mathbf{x}]$ (resp. $t[\mathbf{x}]$). If a formula (resp. term) does not contain any variables we refer to it as *ground* formula (resp. term). We further use $\varphi[t/x]$ as a notation for *replacing* all occurrences of x in φ with a term t. Similarly, $\varphi[\mathbf{t}/\mathbf{x}]$ is used as a shorthand for $\varphi[t_1/x_1, \ldots, t_k/x_k]$.

Given a quantified formula $\varphi[\mathbf{x}, \mathbf{y}]$ with universal variables $\mathbf{x}$ and existential variables $\mathbf{y}$, *Skolemization* [15] eliminates all existential variables $\mathbf{y}$ by introducing *fresh* uninterpreted function symbols with arity ≥ 0 for the existential variables $\mathbf{y}$. For example, the *skolemized* form of formula $\exists y_1 \forall \mathbf{x} \exists y_2 . P(y_1, \mathbf{x}, y_2)$ is $\forall \mathbf{x}. P(f_{y_1}, \mathbf{x}, f_{y_2}(\mathbf{x}))$, where f_{y_1} and f_{y_2} are fresh uninterpreted symbols, which we refer to as *Skolem symbols.* The subscript denotes the existential variable that was eliminated by the corresponding Skolem symbol. We write $\mathit{skolemize}(\varphi)$ for the application of Skolemization to formula φ, $\mathit{var}_\forall(\varphi)$ for the set of universal variables in φ, and $\mathit{sym}_{sk}(\varphi)$ for the set of Skolem symbols in φ.

A Σ-structure M maps each bit-vector sort of size n to its domain $\mathcal{BV}_{[n]}$, each function symbol $f : n_1, \ldots, n_k \rightarrow n \in \Sigma$ with arity $k > 0$ to a total function $M(f) : \mathcal{BV}_{[n_1]}, \ldots, \mathcal{BV}_{[n_k]} \rightarrow \mathcal{BV}_{[n]}$, and each constant symbol with size n to an element in $\mathcal{BV}_{[n]}$. We use $M' := M\{x \mapsto v\}$ to denote a Σ-structure M' that maps variable x to a value v of the same sort and is otherwise identical to M. The evaluation $M(x_{[n]})$ of a variable $x_{[n]}$ and $M(c_{[n]})$ of a constant c in M is an element in $\mathcal{BV}_{[n]}$. The evaluation of an arbitrary term t in M is denoted by $M[\![t]\!]$ and is recursively defined as follows. For a constant c (resp. variable x) $M[\![c]\!] = M(c)$ (resp. $M[\![x]\!] = M(x)$). A function symbol f is evaluated as $M[\![f(t_1, \ldots, t_k)]\!] = M(f)(M[\![t_1]\!], \ldots, M[\![t_k]\!])$. A Σ-structure M is a *model* of a formula φ if $M[\![\varphi]\!] = \top$. A formula is *satisfiable* if and only if it has a model.

3 Overview

In essence, our *counterexample-guided model synthesis* (CEGMS) approach for solving quantified bit-vector problems combines a variant of counterexample-guided quantifier instantiation (CEGQI) [9] with the syntax-guided synthesis

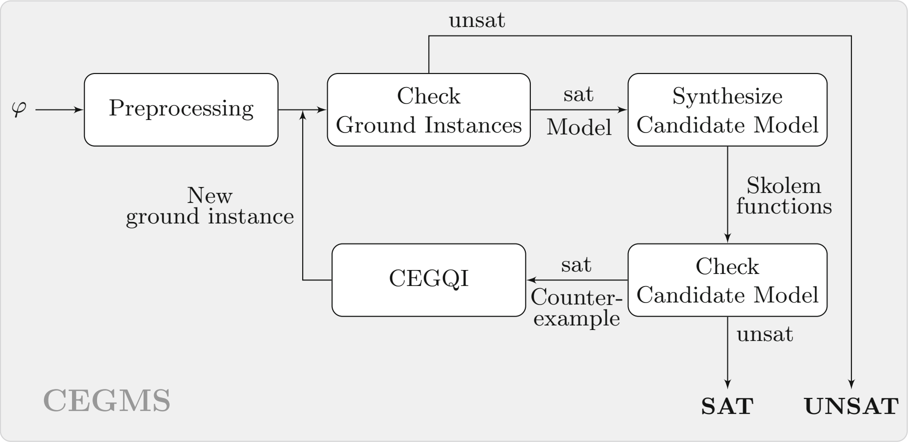

Fig. 1. Basic workflow of our CEGMS approach.

approach in [14] in order to synthesize Skolem functions. The general workflow of our approach is depicted in Fig. 1 and introduced as follows.

Given a quantified formula φ as input, CEGMS first applies Skolemization as a preprocessing step and initializes an empty set of ground instances. This empty set is, in the following, iteratively extended with ground instances of φ, generated via CEGQI. In each iteration, CEGMS first checks for a ground conflict by calling a ground theory solver instance on the set of ground instances. If the solver returns *unsatisfiable*, a ground conflict was found and the CEGMS procedure concludes with UNSAT. If the solver returns *satisfiable*, it produces a model for the Skolem symbols, which serves as a base for synthesizing a candidate model for all Skolem functions. If the candidate model is valid, the CEGMS procedure concludes with SAT. However, if the candidate model is invalid, the solver generates a counterexample, which is used to create a new ground instance of the formula via CEGQI. The CEGMS procedure terminates, when either a ground conflict occurs, or a valid candidate model is synthesized.

4 Counterexample-Guided Model Synthesis

The main refinement loop of our CEGMS approach is realized via CEGQI [9], a technique similar to the *generalization by substitution* approach described in [12], where a concrete counterexample to universal variables is used to create a ground instance of the formula, which then serves as a refinement for the candidate model. Similarly, every refinement step of our CEGMS approach produces a ground instance of the formula by instantiating its universal variables with a counter example if the synthesized candidate model is not valid. The counterexample corresponds to a concrete assignment to the universal variables for which the candidate model does not satisfy the formula. Figure 2 introduces the main algorithm of our CEGMS approach as follows.

```
1  function CEGMS ($\varphi$)
2      $G := \top$, $\mathbf{x} := \text{var}_\forall(\varphi)$
3      $\varphi_{sk} := \text{skolemize}(\text{preprocess}(\varphi))$      // apply Skolemization
4      $\mathbf{f} := \text{sym}_{\text{sk}}(\varphi_{sk})$                   // Skolem symbols
5      $\varphi_G := \varphi_{sk}[\mathbf{u}/\mathbf{x}]$                     // ground $\varphi_{sk}$ with fresh constants $\mathbf{u}$
6      while true
7          $r, M_G := \text{sat}(G)$                       // check set of ground instances
8          if $r = unsat$ return $unsat$                   // found ground conflict
9          $M_S := \text{synthesize}(\mathbf{f}, G, M_G, \varphi_G)$   // synthesize candidate model
10         $r, M_C := \text{sat}(\neg\varphi_G[M_S(\mathbf{f})/\mathbf{f}])$    // check candidate model
11         if $r = unsat$ return $sat$                     // candidate model is valid
12         $G := G \wedge \varphi_G[M_C(\mathbf{u})/\mathbf{u}]$         // new ground instance via CEGQI
```

Fig. 2. High level view of our CEGMS approach.

Given a quantified bit-vector formula φ, we represent φ as a directed acyclic graph (DAG), with the Boolean layer expressed by means of *AND* and *NOT*. As a consequence, it is not possible to transform φ into negative normal form (NNF) and we therefore apply quantifier normalization as a preprocessing step to ensure that a quantifier does not occur in both negated and non-negated form. For the same reason, an *ite*-term is eliminated in case that a quantifier occurs in its condition. Note that if φ is not in NNF, it is sufficient to keep track of the polarities of the quantifiers, i.e., to count the number of negations from the root of the formula to the resp. quantifier, and flip the quantifier if the number of negations is odd. If a quantifier occurs negative and positive, the scope of the quantifier is duplicated, the quantification is flipped, and the negative occurrence is substituted with the new subgraph. Further note that preprocessing currently does not include any further simplification techniques such as miniscoping or destructive equality resolution (DER) [11].

After preprocessing, Skolemization is applied to the normalized formula, and all universal variables $\mathbf{x}$ in φ_{sk} are instantiated with fresh bit-vector constants $\mathbf{u}$ of the same sort. This yields ground formula φ_G. Initially, procedure CEGMS starts with an empty set of ground instances G, which is iteratively extended with new ground instances during the refinement loop.

In the first step of the loop, an SMT solver instance checks whether G contains a ground conflict (line 7). If this is the case, procedure CEGMS has found conflicting quantifier instantiations and concludes with *unsatisfiable*. Else, the SMT solver produces model M_G for all Skolem symbols in G, i.e., every Skolem constant is mapped to a bit-vector value, and every uninterpreted function corresponding to a Skolem function is mapped to a partial function mapping bit-vector values. Model M_G is used as a base for synthesizing a candidate model M_S that satisfies G. The synthesis of candidate models M_S will be introduced in more detail in the next section. In order to check if M_S is also a model that satisfies φ, we check with an additional SMT solver instance if there exists an assignment to constants $\mathbf{u}$ (corresponding to universal variables $\mathbf{x}$), such that candidate model M_S does not satisfy formula φ (line 10).

If the second SMT solver instance returns unsatisfiable, no such assignment to constants $\mathbf{u}$ exists and consequently, candidate model M_S is indeed a valid model for the Skolem functions and procedure CEGMS returns with *satisfiable*. Else, the SMT solver produces a concrete counterexample for constants $\mathbf{u}$, for which candidate model M_S does not satisfy formula φ. This counterexample is used as a quantifier instantiation to create a new ground instance $g_i := \varphi_G[M_C(\mathbf{u})/\mathbf{u}]$, which is added to $G := G \wedge g_i$ as a refinement (line 12) and considered in the next iteration for synthesizing a candidate model. These steps are repeated until either a ground conflict is found or a valid candidate model was synthesized.

Our CEGMS procedure creates in the worst-case an unmanageable number of ground instances of the formula prior to finding either a ground conflict or a valid candidate model, infinitely many in case of infinite domains. In the bit-vector case, however, it produces in the worst-case exponentially many ground instances in the size of the domain. Since, given a bit-vector formula, there exist only finitely many such ground instances, procedure CEGMS will always terminate. Further, if CEGMS concludes with satisfiable, it returns with a model for the existential variables.

5 Synthesis of Candidate Models

In our CEGMS approach, based on a concrete model M_G we apply synthesis to find general models M_S to accelerate either finding a valid model or a ground conflict. Consider formula $\varphi := \forall xy \exists z \,.\, z = x + y$, its skolemized form $\varphi_{sk} := \forall xy \,.\, f_z(x, y) = x + y$, some ground instances $G := f_z(0, 0) = 0 \wedge f_z(0, 1) = 1 \wedge f_z(1, 2) = 3$, and model $M_G := \{f_z(0, 0) \mapsto 0, f_z(0, 1) \mapsto 1, f_z(1, 2) \mapsto 3\}$ that satisfies G. A simple approach for generating a Skolem function for f_z would be to represent model $M_G(f_z)$ as a lambda term $\lambda xy.ite(x = 0 \wedge y = 0, 0, ite(x = 0 \wedge y = 1, 1, ite(x = 1 \wedge y = 2, 3, 0)))$ with base case constant 0, and check if it is a valid Skolem function for f_z. If it is not valid, a counterexample is generated and a new ground instance is added via CEGQI to refine the set of ground instances G. However, this approach, in the worst-case, enumerates exponentially many ground instances until finding a valid candidate model. By introducing a modified version of a syntax-guided synthesis technique called *enumerative learning* [16], CEGMS is able to produce a more succinct and more general lambda term $\lambda xy \,.\, x + y$, which satisfies the ground instances G and formula φ_{sk}.

Enumerative learning as in [16] systematically enumerates expressions that can be built based on a given syntax and checks whether the generated expression conforms to a set of concrete test cases. These expressions are generated in increasing order of a specified complexity metric, such as, e.g., the size of the expression. The algorithm creates larger expressions by combining smaller ones of a given size, which is similar to the idea of dynamic programming. Each generated expression is evaluated on the concrete test cases, which yields a vector of values also denoted as *signature*. In order to reduce the number of enumerated expressions, the algorithm discards expressions with identical signatures, i.e., if two expressions produce the same signature the one generated first will be stored and the other one will be discarded. Figure 3 depicts a simplified version of the enumerative learning algorithm as employed in our CEGMS approach, while a more detailed description of the original algorithm can be found in [16].

```
function enumlearn ($f$, $I$, $O$, $T$, $M$)
  $S := \emptyset$, $E[1] := I$, $\mathit{size} = 0$
  while true
    $\mathit{size} := \mathit{size} + 1$                          // increase expression size to create
    for $t \in$ enumexps($\mathit{size}, O, E$)                   // enumerate all expressions of size $\mathit{size}$
      $s :=$ eval($M, T[t/f]$)                                    // compute signature of $t$
      if $s \notin S$                                             // expression not yet created
        $S := S \cup \{s\}$                                       // cache signature
        if checksig($s$) return $t$                               // $t$ conforms to test cases $T$
        $E[\mathit{size}] := E[\mathit{size}] \cup \{t\}$         // store expression $t$
```

Fig. 3. Simplified version of enumerative learning [16] employed in CEGMS.

Given a Skolem symbol f, a set of inputs I, a set of operators O, a set of test cases T, and a model M, algorithm enumlearn attempts to synthesize a term t, such that $T[t/f]$ evaluates to true under model M. This is done by enumerating all terms t that can be built with inputs I and bit-vector operators O. Enumerating all expressions of a certain size (function enumexps) follows the original enumerative learning approach [16]. Given an expression size *size* and a bit-vector operator o, the size is partitioned into partitions of size $k = arity(o)$, e.g., (1,3) (3,1) (2,2) for $\mathit{size} = 4$ and $k = 2$. Each partition $(s_1, \ldots, s_k)$ specifies the size s_i of expression e_i, and is used to create expressions of size *size* with operator o, i.e., $\{o(e_1, \ldots, e_k) \mid (e_1, \ldots, e_k) \in E[s_1] \times \ldots \times E[s_k]\}$, where $E[s_i]$ corresponds to the set of expressions of size s_i. Initially, for $\mathit{size} = 1$, function enumexps enumerates inputs only, i.e., $E[1] = I$.

For each generated term t, a signature s is computed from a set of test cases T with function eval. In the original algorithm [16], set T contains concrete examples of the input/output relation of f, i.e., it defines a set of output values of f under some concrete input values. In our case, model $M(f)$ may be used as a test set T, since it contains a concrete input/output relation on some bit-vector values. However, we are not looking for a term t with that concrete input/output value behaviour, but a term t that at least satisfies the set of current ground instances G. Hence, we use G as our test set and create a signature s by evaluating every ground instance $g_i \in G[t/f]$, resulting in a tuple of Boolean constants, where the Boolean constant at position i corresponds to the value $M[\![g_i]\!]$ of ground instance $g_i \in G[t/f]$ under current model M. Procedure checksig returns true if signature s contains only the Boolean constant $\top$, i.e., if every ground instance $g_i \in G$ is satisfied.

As a consequence of using G rather than $M(f)$ as a test set T, the expression enumeration space is even more pruned since computing the signature of f

w.r.t. G yields more identical expressions (and consequently, more expressions get discarded). Note that the evaluation via function eval does not require additional SMT solver calls, since the value of ground instance $g_i \in G[t/f]$ can be computed via evaluating $M[\![g_i]\!]$.

Algorithm synthesize produces Skolem function candidates for every Skolem symbol $f \in \mathbf{f}$, as depicted in Fig. 4. Initially, a set of bit-vector operators O is selected, which consists of those operators appearing in formula φ_G. Note that we do not select all available bit-vector operators of the bit-vector theory in order to reduce the number of expressions to enumerate. The algorithm then selects a set of inputs I, consisting of the universal variables on which f depends and the constant values that occur in formula φ_G. Based on inputs I and operators O, a term t for Skolem symbol f is synthesized and stored in model M_S (lines 4–7). If algorithm enumlearn is not able to synthesize a term t, model $M_G(f)$ is used instead. This might happen if function enumlearn hits some predefined limit such as the maximum number of expressions enumerated.

```
1  function synthesize ($\mathbf{f}$, $G$, $M_G$, $\varphi_G$)
2      $M_S := M_G$, $O := \text{ops}(\varphi_G)$            // choose operators $O$ w.r.t. formula $\varphi_G$
3      for $f \in \mathbf{f}$
4          $I := \text{inputs}(f, \varphi_G)$                 // choose inputs for $f$
5          $t := \text{enumlearn}(f, I, O, G, M_S)$    // synthesize term $t$
6          if $t \neq null$
7              $M_S := M_S\{f \mapsto t\}$             // update model
8      return $M_S$
```

Fig. 4. Synthesis of candidate models in CEGMS.

In each iteration step of function synthesize, model M_S is updated if enumlearn succeeded in synthesizing a Skolem function. Thus, in the next iterations, previously synthesized Skolem functions are considered for evaluating candidate expressions in function enumlearn. This is crucial to guarantee that each synthesized Skolem function still satisfies the ground instances in G. Otherwise, M_S may not rule out every counterexample generated so far, and thus, validating the candidate model may result in a counterexample that was already produced in a previous refinement iteration. As a consequence, our CEGMS procedure would not terminate even for finite domains since it might get stuck in an infinite refinement loop while creating already existing ground instances.

The number of inputs and bit-vector operators used as base for algorithm enumlearn significantly affects the size of the enumeration space. Picking too many inputs and operators enumerates too many expressions and algorithm enumlearn will not find a candidate term in a reasonable time, whereas restricting the number of inputs and operators too much may not yield a candidate expression at all. In our implementation, we kept it simple and maintain a set of base operators $\{ite, {\sim}\}$, which gets extended with additional bit-vector operators occurring in the original formula. The set of inputs consists of the constant values occurring in the original formula and the universal variables on which a Skolem symbol depends. Finding more restrictions on the combination of inputs and bit-vector operators in order to reduce the size of the enumeration space is an important issue, but left to future work.

Example 1. Consider $\varphi := \forall x \exists y \,.\, (x < 0 \rightarrow y = -x) \wedge (x \geq 0 \rightarrow y = x)$, and its skolemized form $\forall x \,.\, (x < 0 \rightarrow f_y(x) = -x) \wedge (x \geq 0 \rightarrow f_y(x) = x)$, where y and consequently $f_y(x)$ corresponds to the absolute value function *abs(x)*. For synthesizing a candidate model for f_y, we first pick the set of inputs $I := \{x, 0\}$ and the set of operators $O := \{-, \sim, <, ite\}$ based on formula φ. Note that we omitted operators $\geq$ and $\rightarrow$ since they can be expressed by means of the other operators. The ground formula and its negation are defined as follows.

$$\begin{aligned}\varphi_G &:= (u < 0 \rightarrow f_y(u) = -u) \wedge (u \geq 0 \rightarrow f_y(u) = u)\\ \neg\varphi_G &:= (u < 0 \;\wedge\; f_y(u) \neq -u) \vee (u \geq 0 \;\wedge\; f_y(u) \neq u)\end{aligned}$$

For every refinement round i, the table below shows the set of ground instances G, the synthesized candidate model $M(f_y)$, formula $\neg\varphi_G[M_S(f_y)/f_y]$ for checking the candidate model, and a counterexample M_C for constant u if the candidate model was not correct.

i	G	$M_S(f_y)$	$\neg\varphi_G[M_S(f_y)/f_y]$	$M_C(u)$
1	$\top$	$\lambda x.0$	$(u < 0 \wedge 0 \neq -u) \vee (u \geq 0 \wedge 0 \neq u)$	1
2	$f_y(1) = 1$	$\lambda x.x$	$(u < 0 \wedge u \neq -u) \vee (u \geq 0 \wedge u \neq u)$	-1
3	$f_y(-1) = 1$	$\lambda x.ite(x < 0, -x, x)$	$(u < 0 \wedge ite(u < 0, -u, u) \neq -u) \vee$ $(u \geq 0 \wedge ite(u < 0, -u, u) \neq u)$	-

In the first round, the algorithm starts with ground formula $G := \top$. Since any model of f_y satisfies G, for the sake of simplicity, we pick $\lambda x.0$ as candidate, resulting in counterexample $u = 1$, and refinement $\varphi_G[1/u] \equiv f_y(1) = 1$ is added to G. In the second round, lambda term $\lambda x.x$ is synthesized as candidate model for f_y since it satisfies $G := f_y(1) = 1$. However, this is still not a valid model for f_y and counterexample $u = -1$ is produced, which yields refinement $\varphi_G[-1/u] \equiv f_y(-1) = 1$. In the third and last round, $M_S(f_y) := \lambda x.ite(x < 0, -x, x)$ is synthesized and found to be a valid model since $\neg\varphi_G[M_S(f_y)/f_y]$ is unsatisfiable, and CEGMS concludes with satisfiable.

6 Dual Counterexample-Guided Model Synthesis

Our CEGMS approach is a model finding procedure that enables us to synthesize Skolem functions for satisfiable problems. However, for the unsatisfiable case we rely on CEGQI to find quantifier instantiations based on concrete counterexamples that produce conflicting ground instances. In practice, CEGQI is often successful in finding ground conflicts. However, it may happen that way too many quantifier instantiations have to be enumerated (in the worst-case exponentially many for finite domains, infinitely many for infinite domains). In order to obtain better (symbolic) candidates for quantifier instantiation, we exploit the concept of duality of the input formula and simultaneously apply our CEGMS approach to the original input and its negation (the *dual* formula).

Given a quantified formula φ and its negation, the dual formula $\neg\varphi$, e.g., $\varphi := \forall \mathbf{x} \exists \mathbf{y}. P[x, y]$ and $\neg\varphi := \exists \mathbf{x} \forall \mathbf{y}. \neg P[x, y]$. If $\neg\varphi$ is satisfiable, then there exists a model $M(\mathbf{x})$ to its existential variables $\mathbf{x}$ such that $\varphi[M(\mathbf{x})/\mathbf{x}, \mathbf{y}]$ is unsatisfiable. That is, a model in the dual formula $\neg\varphi$ can be used as a quantifier instantiation in the original formula φ to immediately produce a ground conflict. Similarly, if $\neg\varphi$ is unsatisfiable, then there exists no quantifier instantiation in φ such that φ is unsatisfiable. As a consequence, if we apply CEGMS to the dual formula and it is able to synthesize a valid candidate model, we obtain a quantifier instantiation that immediately produces a ground conflict in the original formula. Else, if our CEGMS procedure concludes with unsatisfiable on the dual formula, there exists no model to its existential variables and therefore, the original formula is satisfiable.

Dual CEGMS enables us to simultaneously search for models and quantifier instantiations, which is particularly useful in a parallel setting. Further, applying synthesis to produce quantifier instantiations via the dual formula allows us to create terms that are not necessarily ground instances of the original formula. This is particularly useful in cases where heuristic quantifier instantiation techniques based on E-matching [17] or model-based quantifier instantiation [10] struggle due to the fact that they typically select terms as candidates for quantifier instantiation that occur in some set of ground terms of the input formula, as illustrated by the following example.

Example 2. Consider the unsatisfiable formula $\varphi := \forall x \; . \; a * c + b * c \neq x * c$, where $x = a + b$ produces a ground conflict. Unfortunately, $a + b$ is not a ground instance of φ and is consequently not selected as a candidate by current state-of-the-art heuristic quantifier instantiation techniques. However, if we apply CEGMS to the dual formula $\forall abc \exists x \; . \; a * c + b * c = x * c$, we obtain $\lambda xyz.x + y$ as a model for Skolem symbol $f_x(a, b, c)$, which corresponds to the term $a + b$ if instantiated with (a, b, c). Selecting $a + b$ as a term for instantiating variable x in the original formula results in a conflicting ground instance, which immediately allows us to determine unsatisfiability.

Note that if CEGMS concludes unsatisfiable on the dual formula, we currently do not produce a model for the original formula. Generating a model would require further reasoning, e.g., proof reasoning, on the conflicting ground instances of the dual formula and is left to future work.

Further, dual CEGMS currently only utilizes the final result of applying CEGMS to the dual formula. Exchanging intermediate results (synthesized candidate models) between the original and the dual formula in order to prune the search is an interesting direction for future work.

In the context of quantified Boolean formulas (QBF), the duality of the given input has been previously successfully exploited to prune and consequently speed up the search in circuit-based QBF solvers [18]. In the context of SMT, in previous work we applied the concept of duality to optimize lemmas on demand approach for the theory of arrays in Boolector [19].

7 Experiments

We implemented our CEGMS technique and its dual version in our SMT solver Boolector [20], which supports the theory of bit-vectors combined with arrays and uninterpreted functions. We evaluated our approach on two sets of benchmarks (5029 in total). Set BV (191) contains all BV benchmarks of SMT-LIB [21], whereas set BV_{LNIRA} (4838) consists of all LIA, LRA, NIA, NRA benchmarks of SMT-LIB [21] translated into bit-vector problems by substituting every integer or real with a bit-vector of size 32, and every arithmetic operator with its signed bit-vector equivalent.

We evaluated four configurations of Boolector[1]: (1) **Btor**, the CEGMS version without synthesis, (2) **Btor+s**, the CEGMS version with synthesis enabled, (3) **Btor+d**, the dual CEGMS version without synthesis, (4) **Btor+ds**, the dual CEGMS version with synthesis enabled. We compared our approach to the current development versions of the state-of-the-art SMT solvers CVC4[2] [6] and Z3[3] [7], and the BDD-based approach implemented as a prototype called Q3B[4] [13]. The tool Q3B runs two processes with different approximation strategies in a parallel portfolio setting, where one process applies over-approximation and the other under-approximation. The dual CEGMS approach implemented in Boolector is realized with two parallel threads within the solver, one for the original formula and the other for the dual formula. Both threads do not exchange any information and run in a parallel portfolio setting.

All experiments were performed on a cluster with 30 nodes of 2.83GHz Intel Core 2 Quad machines with 8GB of memory using Ubuntu 14.04.5 LTS. We set the limits for each solver/benchmark pair to 7GB of memory and 1200 seconds of CPU time (not wall clock time). In case of a timeout, memory out, or an error, a penalty of 1200 seconds was added to the total CPU time.

Figure 5 illustrates the effect of our model synthesis approach by comparing configurations **Btor** and **Btor+s** on the BV and BV_{LNIRA} benchmark sets. On the BV benchmark set, **Btor+s** solves 22 more instances (21 satisfiable, 1 unsatisfiable) compared to **Btor**. The gain in the number of satisfiable instances is due to the fact that CEGMS is primarily a model finding procedure, which allows to find symbolic models instead of enumerating a possibly large number of bit-vector values, which seems to be crucial on these instances.

[1] Boolector commit 4f7837876cf9c28f42649b368eaffaf03c7e1357.

[2] CVC4 commit d19a95344fde1ea1ff7d784b2c4fc6d09f459899.

[3] Z3 commit 186afe7d10d4f0e5acf40f9b1f16a1f1c2d1706c.

[4] Q3B commit 68301686d36850ba782c4d0f9d58f8c4357e1461.

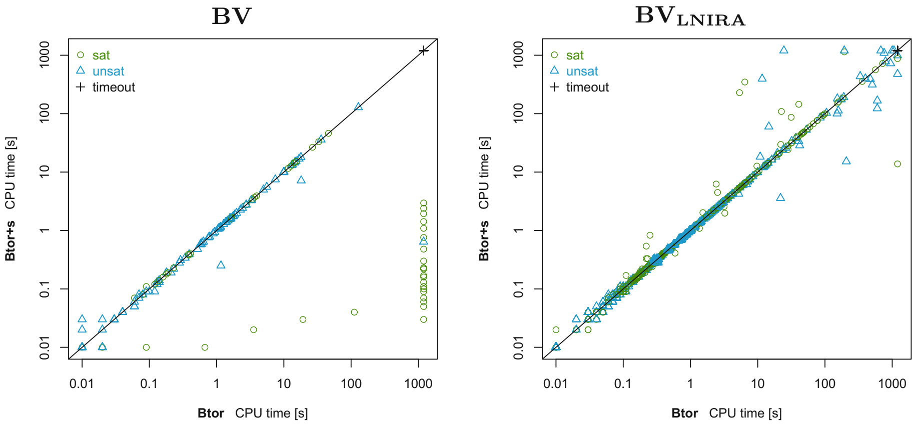

Fig. 5. Comparison of Boolector with model synthesis enabled (**Btor+s**) and disabled (**Btor**) on the BV and BV$_{LNIRA}$ benchmarks.

On set BV$_{LNIRA}$, however, **Btor+s** does not improve the overall number of solved instances, even though it solves two satisfiable instances more than **Btor**. Note that benchmark set BV$_{LNIRA}$ contains only a small number of satisfiable benchmarks (at most 12% = 575 benchmarks[5]), where configuration **Btor** already solves 465 instances without enabling model synthesis. For the remaining satisfiable instances, the enumeration space may still be too large to synthesize a model in reasonable time and may require more pruning by introducing more syntactical restrictions for algorithm *enumlearn* as discussed in Sect. 5.

Figure 6 shows the effect of model synthesis on the dual configurations **Btor+d** and **Btor+ds** on benchmark sets BV and BV$_{LNIRA}$. On the BV benchmark set, configuration **Btor+ds** is able to solve 10 more instances of which all are satisfiable. On the BV$_{LNIRA}$ benchmark set, compared to **Btor+d**, configuration **Btor+ds** is able to solve 132 more instances of which all are unsatisfiable. The significant increase is due to the successful synthesis of quantifier instantiations (133 cases).

Table 1 summarizes the results of all four configurations on both benchmark sets. Configuration **Btor+ds** clearly outperforms all other configurations w.r.t. the number of solved instances and runtime on both benchmark sets. Out of all 77 (517) satisfiable instances in set BV (BV$_{LNIRA}$) solved by **Btor+ds**, 32 (321) were solved by finding a ground conflict in the dual CEGMS approach. In case of configuration **Btor+d**, out of 67 (518) solved satisfiable instances, 44 (306) were solved by finding a ground conflict in the dual formula. As an interesting observation, 16 (53) of these instances were not solved by **Btor**. Note,

[5] Boolector, CVC4, Q3B, and Z3 combined solved 4263 unsatisfiable and 533 satisfiable instances, leaving only 42 instances unsolved.

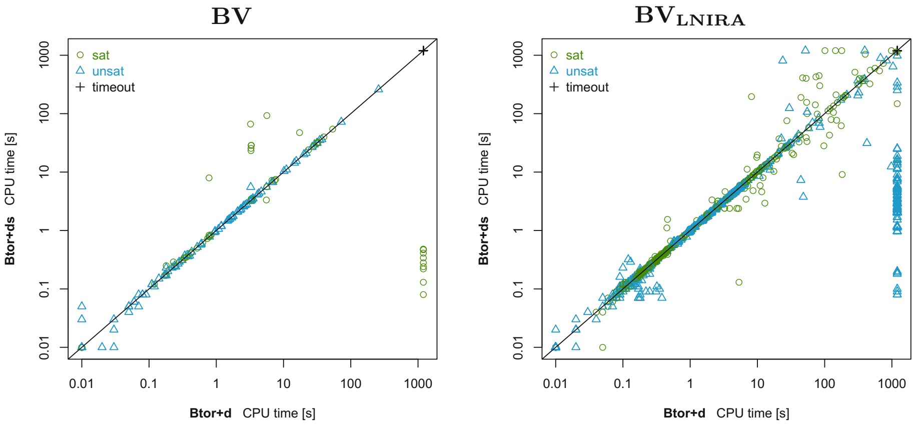

Fig. 6. Comparison of dual CEGMS with model synthesis enabled (**Btor+ds**) and disabled (**Btor+d**) on the BV and BV$_{LNIRA}$ benchmarks.

Table 1. Results for all configurations on the BV and BV$_{LNIRA}$ benchmarks.

	BV (191)					**BV$_{LNIRA}$** (4838)				
	Solved	Sat	Unsat	Time [s]	Uniq	Solved	Sat	Unsat	Time [s]	Uniq
Btor	142	51	91	59529	0	4527	465	4062	389123	3
Btor+s	164	72	92	32996	0	4526	467	4059	390661	1
Btor+d	162	67	95	35877	0	4572	**518**	4054	342412	4
Btor+ds	**172**	**77**	**95**	**24163**	0	**4704**	517	**4187**	**187411**	**135**

however, that **Btor+d** is not able to construct a model for these instances due to the current limitations of our dual CEGMS approach as described in Sect. 6.

On the BV benchmark set, model synthesis significantly reduces the number of refinement iterations. Out of 142 commonly solved instances, **Btor+s** required 165 refinement iterations, whereas **Btor** required 664 refinements. On the 4522 commonly solved instances of the BV$_{LNIRA}$ benchmark set, **Btor+s** requires 5249 refinement iterations, whereas **Btor** requires 5174 refinements. The difference in the number of refinement iterations is due to the fact that enabling model synthesis may produce different counterexamples that requires the CEGMS procedure to sometimes create more refinements. However, as noted earlier, enabling model synthesis on set BV$_{LNIRA}$ does not improve the overall number of solved instances in the non-dual case.

We analyzed the terms produced by model synthesis for both **Btor+s** and **Btor+ds** on both benchmark sets. On the BV benchmark set, mainly terms of the form $\lambda\mathbf{x}\,.\,c$ and $\lambda\mathbf{x}\,.\,x_i$ with a bit-vector value c and $x_i \in \mathbf{x}$ have been synthesized. On the BV$_{LNIRA}$ benchmarks, additional terms of the form $\lambda\mathbf{x}\,.\,(x_i\ op\ x_j)$, $\lambda\mathbf{x}\,.\,(c\ op\ x_i)$, $\lambda\mathbf{x}\,.\,{\sim}(c * x_i))$ and $\lambda\mathbf{x}\,.\,(x_i + (c + {\sim}x_j))$ with a bit-vector operator *op* were synthesized. On these benchmarks, more complex terms did not occur.

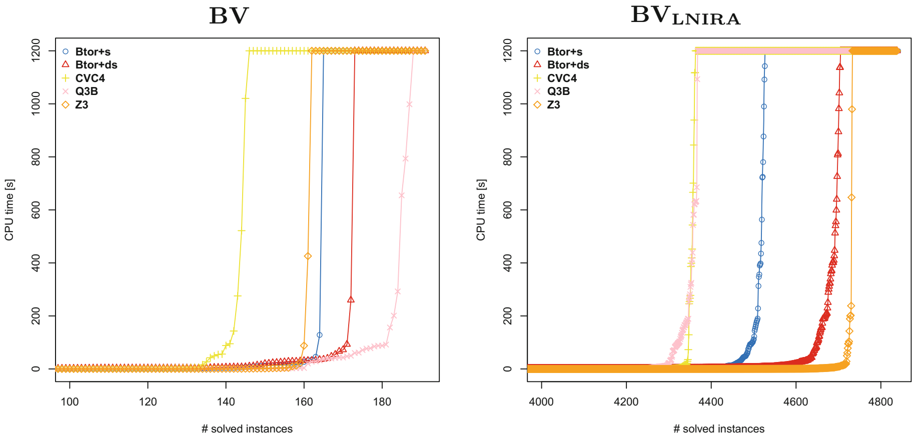

Fig. 7. Cactus plot of the runtime of all solvers on benchmark sets BV and BV_{LNIRA}.

Figure 7 depicts two cactus plots over the runtime of our best configuration **Btor+ds** and the solvers **CVC4**, **Q3B**, and **Z3** on the benchmark sets BV and BV_{LNIRA}. On both benchmark sets, configuration **Btor+ds** solves the second highest number of benchmarks after **Q3B** (BV) and **Z3** (BV_{LNIRA}). On both benchmark sets, a majority of the benchmarks seem to be trivial since they were solved by all solvers within one second.

Table 2 summarizes the results of all solvers on both benchmark sets. On the BV benchmark set, **Q3B** solves with 187 instances the highest number of benchmarks, followed by **Btor+ds** with a total of 172 solved instances. Out of all 19 benchmarks unsolved by **Btor+ds**, 9 benchmarks are solved by **Q3B** and **CVC4** through simplifications only. We expect Boolector to also benefit from introducing quantifier specific simplification techniques, which is left to future work. On the BV_{LNIRA} set, **Z3** solves the most instances (4732) and **Btor+ds** again comes in second with 4704 solved instances. In terms of satisfiable instances, however, **Btor+ds** solves the highest number of instances (517). In terms of unsatisfiable instances, **Z3** clearly has an advantage due to its heuristic

Table 2. Results for all solvers on the BV and BV_{LNIRA} benchmarks with a CPU time limit of 1200 seconds (not wall clock time).

	BV (191)					**BV**$_{LNIRA}$ (4838)				
	Solved	Sat	Unsat	Time [s]	Uniq	Solved	Sat	Unsat	Time [s]	Uniq
Btor+ds	172	77	**95**	24163	2	4704	**517**	4187	187411	**19**
CVC4	145	64	81	57652	0	4362	339	4023	580402	3
Q3B	**187**	**93**	94	**9086**	**9**	4367	327	4040	581252	5
Z3	161	69	92	36593	0	**4732**	476	**4256**	**130405**	11

quantifier instantiation techniques and solves 69 instances more than **Btor+ds**, out of which 66 were solved within 3 seconds. The BDD-based approach of **Q3B** does not scale as well on the BV_{LNIRA} set as on the BV set benchmark set and is even outperformed by **Btor+s**. Note that most of the benchmarks in BV_{LNIRA} involve more bit-vector arithmetic than the benchmarks in set BV.

Finally, considering **Btor+ds**, a wall clock time limit of 1200 seconds increases the number of solved instances of set BV_{LNIRA} by 11 (and by 6 for **Q3B**). On set BV, the number of solved instances does not increase.

8 Conclusion

We presented CEGMS, a new approach for handling quantifiers in SMT, which combines CEGQI with syntax-guided synthesis to synthesize Skolem functions. Further, by exploiting the duality of the input formula dual CEGMS enables us to synthesize terms for quantifier instantiation. We implemented CEGMS in our SMT solver Boolector. Our experimental results show that our technique is competitive with the state-of-the-art in solving quantified bit-vectors even though Boolector does not yet employ any quantifier specific simplification techniques. Such techniques, e.g., miniscoping or DER were found particularly useful in Z3. CEGMS employs two ground theory solvers to reason about arbitrarily quantified formulas. It is a simple yet effective technique, and there is still a lot of room for improvement. Model reconstruction from unsatisfiable dual formulas, symbolic quantifier instantiation by generalizing concrete counterexamples, and the combination of quantified bit-vectors with arrays and uninterpreted functions are interesting directions for future work. It might also be interesting to compare our approach to the work presented in [22–25].
Binary of Boolector, the set of translated benchmarks (BV_{LNIRA}) and all log files of our experimental evaluation can be found at http://fmv.jku.at/tacas17.

References

1. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 120–135. Springer, Heidelberg (2008). doi:10.1007/978-3-540-93900-9_13
2. Cook, B., Kroening, D., Rümmer, P., Wintersteiger, C.M.: Ranking function synthesis for bit-vector relations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 236–250. Springer, Heidelberg (2010). doi:10.1007/978-3-642-12002-2_19
3. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, 17–23 January 2010, pp. 313–326. ACM (2010)
4. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: 6th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2006, San Jose, California, USA, 12–16 November 2006, Proceedings, pp. 117–124. IEEE Computer Society (2006)

5. Kovásznai, G., Fröhlich, A., Biere, A.: Complexity of fixed-size bit-vector logics. Theory Comput. Syst. **59**(2), 323–376 (2016)
6. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22110-1_14
7. Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78800-3_24
8. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 737–744. Springer, Heidelberg (2014). doi:10.1007/978-3-319-08867-9_49
9. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 198–216. Springer, Cham (2015). doi:10.1007/978-3-319-21668-3_12
10. Ge, Y., Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02658-4_25
11. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. In: Bloem, R., Sharygina, N. (eds.) Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, 20–23 October, pp. 239–246. IEEE (2010)
12. Dutertre, B.: Solving exists/forall problems in Yices. In: Workshop on Satisfiability Modulo Theories (2015)
13. Jonáš, M., Strejček, J.: Solving quantified bit-vector formulas using binary decision diagrams. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 267–283. Springer, Heidelberg (2016). doi:10.1007/978-3-319-40970-2_17
14. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, 20–23 October 2013, pp. 1–8. IEEE (2013)
15. Robinson, J.A., Voronkov, A. (eds.): Handbook of Automated Reasoning, vol. 2s. Elsevier and MIT Press, Cambridge (2001)
16. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: TRANSIT: specifying protocols with concolic snippets. In Boehm, H., Flanagan, C., eds.: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013, Seattle, WA, USA, 16–19 June 2013, pp. 287–296. ACM (2013)
17. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
18. Goultiaeva, A., Bacchus, F.: Exploiting QBF duality on a circuit representation. In: Fox, M., Poole, D. (eds.) Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, 11–15 July 2010. AAAI Press (2010)
19. Niemetz, A., Preiner, M., Biere, A.: Turbo-charging lemmas on demand with don't care reasoning. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, 21–24 October 2014, pp. 179–186. IEEE (2014)
20. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 system description. J. Satisfiability Boolean Model. Comput. **9**, 53–58 (2014) (2015, published)
21. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016). www.SMT-LIB.org

22. Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Automated discovery of simulation between programs. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 606–621. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48899-7_42
23. John, A.K., Chakraborty, S.: A layered algorithm for quantifier elimination from linear modular constraints. Formal Methods Syst. Des. **49**(3), 272–323 (2016)
24. Bjørner, N., Janota, M.: Playing with quantified satisfaction. In: Fehnker, A., McIver, A., Sutcliffe, G., Voronkov, A., (eds.) 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations, LPAR 2015. EPiC Series in Computing, Suva, Fiji, 24–28 November 2015, vol. 35, pp. 15–27. EasyChair (2015)
25. Farzan, A., Kincaid, Z.: Linear arithmetic satisfiability via strategy improvement. In: Kambhampati, S. (ed.) Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9–15 July 2016, pp. 735–743. IJCAI/AAAI Press (2016)