Scalable Algorithms for Abduction via Enumerative Syntax-Guided Synthesis

Andrew Reynolds[1], Haniel Barbosa[2], Daniel Larraz[1], and Cesare Tinelli[1]

[1] Department of Computer Science, The University of Iowa
[2] Department of Computer Science, Universidade Federal de Minas Gerais (UFMG)

Abstract. The abduction problem asks whether there exists a predicate that is consistent with a given set of axioms that when added to these axioms suffices to entail a goal. We propose an approach for solving the abduction problem that is based on syntax-guided enumeration. For scalability, we use a novel procedure that incrementally constructs a solution in disjunctive normal form that is built from enumerated predicates. The procedure can be configured to generate progressively weaker and simpler solutions over the course of a run of the procedure. Our approach is fully general and can be applied over any background logic that is handled by the underlying SMT solver in our approach. Our experiments show our approach compares favorably with other tools for abductive reasoning.

1 Introduction

The abduction problem for theory T, axioms A and goal G asks whether there exists a formula φ such that: (i) $\mathsf{A} \wedge \varphi$ is T-satisfiable and (ii) $\mathsf{A} \wedge \varphi \models_T \mathsf{G}$. In other words, φ is a possible formula that when added to the set of axioms allows the goal to be proven. Abductive reasoning has gained a multitude of applications recently, including for extending knowledge bases for failed verification conditions [16] and invariant generation [17, 20]. Despite the multitude of applications, general tools for automatic abductive inference are not yet mainstream, although some tools have been developed, including GPID [19] and EXPLAIN [15].

Meanwhile, a number of high-performance general purpose tools for syntax-guided synthesis (SyGuS) have also been developed in the past decade. These solvers been applied successfully in a number of domains, including implementation of network protocols [36], data processing [22], and code optimization [29]. The performance of these solvers is evaluated in a yearly competition, where considerable progress has been noted in recent years [4].

In this paper, we investigate scalable approaches to solving the abduction problem using (enumerative) syntax-guided synthesis techniques. We make no assumptions about the background theory, other than it must be one supported by an existing SMT solver and is amenable to syntax-guided synthesis. Our goal is to leverage the scalability of syntax-guided synthesis solvers and apply them to applications where abductive reasoning is required. Our longer term goal is to standardize the interface for these solvers for abduction problems and make them freely available to users of program analysis and automated reasoning who would benefit from high performance automated reasoning systems for abduction.

Contributions

- We introduce a novel procedure for solving abduction problems using enumerative syntax-guided synthesis. Our procedure can be applied for any background theory where syntax-guided synthesis can be applied.
- We give an extension of the procedure that is capable of generating weaker solutions to the abduction problem over the course of a run of the procedure.
- We implement these techniques in the state-of-the-art syntax-guided synthesis solver CVC4 [31], and design several experiments to test its effectiveness. We show that it has compelling advantages with respect to existing approaches for abduction including EXPLAIN [15] and GPID [19].

2 Preliminaries

We work in the context of many-sorted first-order logic with equality ($\simeq$) and assume the reader is familiar with the notions of signature, terms, and so on (see, e.g., [21]). A *theory* is a pair $T = (\Sigma, I)$ where Σ is a signature and I is a non-empty class of Σ-interpretations, the *models of* T, that is closed under variable reassignment (i.e., every Σ-interpretation that differs from one in I only in how it interprets the variables is also in I) and isomorphism. A Σ-formula φ is *T-satisfiable* (respectively, *T-unsatisfiable*) if it is satisfied by some (resp., no) interpretation in I. A satisfying interpretation for φ is a *model* φ. A formula φ is *valid in* T (or *T-valid*), written $\models_T \varphi$, if every model of T is a model of φ. We write $\varphi[\boldsymbol{x}]$ for a tuple $\boldsymbol{x}$ of distinct variables to indicate that the free variables of φ occur in $\boldsymbol{x}$. Given $\varphi[\boldsymbol{x}]$, we write $\varphi[\boldsymbol{t}]$ to denote the result of replacing every occurrence of every variable of $\boldsymbol{x}$ in φ with the corresponding term in $\boldsymbol{t}$.

Syntax-Guided Synthesis (SyGuS) Syntax-guided synthesis [2] is a recent paradigm for automated synthesis that combines semantic and syntactic restrictions on the space of solutions. In detail, a SyGuS problem for a function f in a theory T consists of

1. *semantic restrictions*, a specification given by a (second-order) T-formula of the form $\exists f.\, \forall \boldsymbol{x}.\, \varphi[f, \boldsymbol{x}]$, and
2. *syntactic restrictions* on the solutions for f, given by a context-free grammar $\mathcal{R}$.

The grammar $\mathcal{R}$ is a triple (s_0, S, R) where s_0 is an initial symbol, S is a set of symbols with $s_0 \in S$, and R is a set of *production rules* of the form $s \to t$, where $s \in S$ and t is a term built from the symbols in the signature of theory T, free variables, and symbols from S. The rules define a rewrite relation over such terms, also denoted by $\to$, as expected. We say a term t is *generated* by $\mathcal{R}$ if $s_0 \to^* t$ where $\to^*$ is the reflexive-transitive closure of $\to$ and t does not contain symbols from S. For example, the terms x, $(x + x)$ and $((1 + x) + 1)$ are all generated by the grammar $\mathcal{R} = (\mathsf{I}, \{\mathsf{I}\}, \{\mathsf{I} \to x, \mathsf{I} \to 1, \mathsf{I} \to (\mathsf{I} + \mathsf{I})\})$. A *solution for the SyGuS problem for* f is a lambda term $\lambda \boldsymbol{x}.e$ of the same type as f such that (i) $\forall \boldsymbol{x}.\, \varphi[\lambda \boldsymbol{x}.e,\, \boldsymbol{x}]$ is T-valid and (ii) e is a first-order term generated by $\mathcal{R}$.

A number of recent approaches for the syntax-guided synthesis problem exist that target specific fragments, including programming-by-examples [22], single invocation conjectures [32], and pointwise specifications [27, 5]. General purpose methods for solving the syntax-guided synthesis problem are generally based on *enumerative counterexample-guided inductive synthesis* (CEGIS) [34, 35]. An enumerative approach uses a grammar to generate candidate solutions based on some ordering, typically term size (e.g., the

number of non-nullary function applications in the term). These candidate solutions are then tested for correctness using a verification oracle (typically an SMT solver). This process is accelerated by the use of *counterexamples* for previous candidates, i.e., valuations for the input variables $\boldsymbol{x}$, or *points*, that witness the failure of those candidates to satisfy the specification. Despite its simplicity, enumerative CEGIS is the de-facto approach for solving the general class of SyGuS problems, as implemented in a several recent tools, notably CVC4 [31] and the enumerative solver ESOLVER [3]. Its main downside not scaling when the required solution is very large. As we will show in Section 4, we present a scalable procedure for the abduction problem that builds on top of enumerative CEGIS and is capable of quickly finding (conjunctive) solutions to the abduction problem.

3 The Abduction Problem

Informally, the abduction problem for a set A of axioms and a goal G is the problem of finding a formula S that is consistent with A and, together with A, entails the goal. We refine the problem by restricting it to a given background theory T and also considering syntactic restrictions on the solution S. We refer to this as the *syntax-restricted abduction problem*, which we formalize in the following definition.

Definition 1 (Abduction Problem). *The (syntax-restricted) abduction problem for a theory T, a conjunction $\mathsf{A}[\boldsymbol{x}]$ of axioms, a goal $\mathsf{G}[\boldsymbol{x}]$ and a grammar R, where axioms and goal are first-order formulas, is that of finding a first-order formula $\mathsf{S}[\boldsymbol{x}]$ such that:*

1. $\mathsf{A} \wedge \mathsf{S} \models_T \mathsf{G}$,
2. $\mathsf{A} \wedge \mathsf{S}$ *is T-satisfiable, and*
3. S *is generated by grammar* R.

In practice, as in SyGuS, syntactic restrictions on the solution space may be used to capture user-requirements on the desired shape of a solution. They can also be used as a mechanism for narrowing the search space to one where one believes the solver is likely to find a solution. Observe that the formulation of the problem includes the case with no syntactic restriction as a trivial case of a grammar that accepts all formulas in the signature of the theory. In the abduction solver we have developed for this work, the syntax restriction is optional. When it is missing, a grammar generating the full language is constructed internally automatically.

Syntax-restricted abduction bears a strong similarity to SyGuS.[3] In our approach to it, we exploit this similarity by leveraging much of the technology we developed for SyGuS, with the goal of achieving generality and scalability.

Normally, an abduction problem admits many solutions. Thus, it may be useful to look for solutions that optimize certain criteria, such as generality with respect to entailment in T, or minimality with respect to size or number of free variables. Our evaluation contains several case studies where we explore this aspect in further detail.

Recent applications Abduction has a long history in logic and automatic reasoning (see, e.g., [24]). More recently, it has found many useful applications in program analysis.

[3] In fact, it could be readily recast as SyGuS, if one ignored Condition 2 in Definition 1.

It has been used for identifying the possible facts a verification tool is missing to either discharge or validate a verification condition [16], inferring library specifications that are needed for verifying a client program [37], and synthesizing specifications for multiple unknown procedures called from a main program [1]. Other applications includes loop invariant generation [17, 20], where abduction is used for iteratively strengthening candidate solutions until they are inductive and strong enough to verify a program, and compositional program verification [25], where abduction is used for inferring not only loop invariants but preconditions required for the invariants to hold. Abductive inference has also been applied to modular heap reasoning [12], and the synthesis of missing guards for memory safety [18].

4 Abduction via Enumerative Syntax-Guided Synthesis

In this section, we fix a theory T and describe our approach for solving the abduction problem in T using enumerative syntax-guided synthesis. We first present a basic procedure for abduction in the following section, and then extend this procedure so that it generates (conjunctive) solutions in a highly scalable manner. We then describe how either approach can be extended to an incremental onr that constructs solutions that are logically weaker over time. For simplicity, *we restrict ourselves to abduction problems where axioms, goals, and solutions are quantifier-free*.

Requirements on T We assume that the T-satisfiability of quantifier-free formulas is decidable. For each sort of T, we also assume a distinguished set of variable-free terms of that sort which we call *values* (e.g., numerals and negated numerals in the case of integer arithmetic) such that every T-satisfiable formula is satisfied by a valuation of its free variables to sort elements denoted by values. Finally, we require the availability of a computable function Eval that takes a first-order formula $\varphi[\boldsymbol{x}]$ and a tuple $\boldsymbol{p}$ of values of the same length as $\boldsymbol{x}$, and returns $\top$ if $\varphi[\boldsymbol{p}]$ is T-satisfiable and $\bot$ otherwise. This is the case for most theories used in SMT.

4.1 Enumerative Counterexample-Guided Inductive Synthesis for Abduction

We start with a basic CEGIS-style synthesis procedure for solving the syntax-restriction abduction problem where points that represent counterexamples for candidate solutions are cached and used to discard subsequent candidates. The procedure is presented in Figure 1. It takes as input axioms A, goal G and grammar R, and maintains an internally set P of points that satisfy the axioms and falsify the goal. On line 3, we invoke the stateful subprocedure $\mathsf{NextEnum}(\mathsf{R})$ which enumerates the formulas generated by grammar R based on enumerative techniques used in SyGuS solvers. We will refer to the return formula c as the current *candidate solution*. Then, using the (fast) evaluation function Eval, we check online 4 whether c is falsified by all the points in P. If the check fails then we discard c and loop back to line 3 because adding c to A would definitely be not enough to entail G. Otherwise, we check, on line 5, whether $c \wedge \mathsf{A} \wedge \neg\mathsf{G}$ is T-unsatisfiable. If it is T-satisfiable, we obtain a witness point p for the satisfiability, we add it to our set of points P on line 10 and discard c. If the test on line 5 succeeds we check that c is consistent with A before returning it as a possible solutions.

```
GetAbductBasic(axioms A[x], goal G[x], grammar R)
 1: Let P = ∅    // set of points
 2: loop
 3:    Let c[x] = NextEnum(R)
 4:    if Eval(c, p) = ⊥ for all p ∈ P then
 5:       if c ∧ A ∧ ¬G is T-unsatisfiable then
 6:          if c ∧ A is T-satisfiable then
 7:             return c
 8:          end if
 9:       else
10:          P := P ∪ {p} with p such that Eval(c ∧ A ∧ ¬G, p) = ⊤
11:       end if
12:    end if
13: end loop
```

Fig. 1. Basic procedure for the abduction problem for axioms A, goal G and grammar R.

Example 1. Let T be the theory of integer linear arithmetic with the usual signature. Let A be the set $\{y \geqslant 0\}$ and let G be the set $\{x + y + z \geqslant 0\}$, and assume R is a grammar generating all linear arithmetic atomic formulas over the variables x, y, z. The results of the procedure are summarized in the table below. We provide, for each iteration, the candidate c generated by syntax-guided enumeration on line 3, the value of the conditions on lines 4,5 and 6 of the procedure when applicable , and the point added to P in when the condition on line 5 evaluates to false. The last column specifies the solution returned on that iteration if any.

#	c	line 4	line 5	$p \in$ P	line 6	return
1	$x \geqslant 0$	$\top$	$\bot$	$(0, 0, -1)$		
2	$x < 0$	$\top$	$\bot$	$(-1, 0, 0)$		
3	$y \geqslant 0$	$\bot$				
4	$y < 0$	$\top$	$\top$	$()$	$\bot$	
5	$z \geqslant 0$	$\bot$				
6	$z < 0$	$\bot$				
7	$x + y \geqslant 0$	$\bot$				
8	$x + y < 0$	$\bot$				
9	$x + z \geqslant 0$	$\top$	$\top$	$()$	$\top$	$x + z \geqslant 0$

On the first iteration, the syntax-guided enumeration generates the predicate $x \geqslant 0$ as the candidate solution c. This fails to imply the goal, namely the goal is false but the axioms and this candidate are true on the point where $(x, y, z) = (0, 0, -1)$. The second candidate fails for similar reasons for the point $(-1, 0, 0)$. The check on line 4 fails for five of the next six candidates, the exception being the candidate $y < 0$. This candidate evaluates to false for both points in P but must be discarded since it is inconsistent with our axioms on line 6. Finally, on the ninth iteration, the candidate $x + z \geqslant 0$ is generated which is a solution for this abduction problem. □

```
GetAbductUCL(axioms A, goal G, grammar R)
 1: Let E, P, U = ∅
 2: loop
 3:    E += {NextEnum(R)}
 4:    Let C = ∅
 5:    while EnsureCexFalsify(C, E, P, U) do
 6:       if C ∧ A ∧ ¬G is T-unsatisfiable then
 7:          Let C_min ⊆ C such that C_min ∧ A ∧ ¬G is T-unsatisfiable
 8:          if C_min ∧ A is T-satisfiable then
 9:             return C_min
10:          else
11:             U += {u} for some u ⊆ C_min such that u ∧ A is T-unsatisfiable
12:             C -= e for some e ∈ u
13:          end if
14:       else
15:          P += {p} where Eval((C ∧ A ∧ ¬G), p) = ⊤
16:       end if
17:    end while
18: end loop
EnsureCexFalsify(candidate C, predicates E, points P, cores U)
 1: while Eval(C, p) = ⊤ for some p ∈ P do
 2:    if Eval(e, p) = ⊥ for some e ∈ E and u ⊈ C ∪ {e} for all u ∈ U then
 3:       C += {e}
 4:    else
 5:       return false
 6:    end if
 7: end while
 8: return true
```

Fig. 2. Procedure for the abduction problem for A, G and R based on unsat core learning.

4.2 A Procedure for Abduction based on Unsat Core Learning

This section extends the procedure from Figure 1 with techniques that make it scalable when the intended solution to the abduction problem is a conjunction of enumerated predicates. The procedure is applicable to cases where the grammar R admits conjunctions of the predicates it enumerates. More precisely, the procedure in this section requires that $s_0 \rightarrow s_0 \wedge s_0$ is a production rule in R where s_0 is the start symbol of R.

This procedure is presented in Figure 2. Similar to the basic procedure from the previous section, this procedure maintains a set of points P that satisfy the axioms and falsify the goal. Additionally, this procedure maintains a set of enumerated predicates E, U is a set of subsets of E that are inconsistent with the axioms. The procedure adds to each of these three sets during the course of its run. Each loop iteration attempts to construct a set of formulas C whose conjunction is a solution to the abduction problem. This is in contrast to the basic procedure from Figure 1 which considers only single predicates as candidate solutions.

To construct the candidate set C, the procedure uses a helper function EnsureCexFalsify which ensures that (i) the conjunction of the predicates in C is false for all points in P and (ii) no subset of C exists in U. If the former condition were to be violated, then C along with our axioms would not suffice to show the goal. If the latter condition were to be violated, then we would know that C is inconsistent with our axioms. If we are able to successfully construct a candidate solution set C, then line 6 checks whether that candidate indeed suffices when added to the axioms to show the goal. If it does not, we add a point to P. If it does, we construct a (ideally minimal) subset of C_{min} of C that also suffices to show the goal. This information can be readily computed by an SMT solver [10] with support for unsatisfiable core generation [13], a feature common to many modern solvers such as CVC4. We then check whether C_{min} is consistent with our axioms. If it is, then it is a solution to the abduction problem. If it is not, then we add some subset of it to U that is also inconsistent with our axioms, where again this can be computed by an SMT solver with support for unsatisfiable cores. In other words, here we have learned that some subset u should never be included in future candidate solutions. To maintain the invariant that no subset of C occurs in U, we remove one enumerated predicate $e \in u$ from C on line 12. In the case where a point is added to P (line 15) or when an unsat cores is added to U (line 11), we run the method EnsureCexFalsify starting from the current resultant set C. This will force the procedure to construct a new candidate solution if possible based on the set E. When this method fails to construct a candidate, the inner loop terminates and the next predicate is added to E based on syntax-guided enumeration.

We now revisit Example 1. As demonstrated in this example, GetAbductUCL is often capable of generating solutions to the abduction problem faster than the one from Figure 1, albeit those solutions may be logically stronger.

Example 2. We revisit Example 1, where A is the set $\{y \geqslant 0\}$ and G is $\{x + y + z \geqslant 0\}$. A run of the procedure from Figure 2 is summarized in the table below. We list iterations of the outer loop of the procedure (lines 2-18) in the first column of this table. For each iteration, we provide the predicate that is added to our pool E (line 3), the candidate set C we are considering upon a successful call to EnsureCexFalsify. Notice that the inner loop of the procedure may consider multiple candidates C for a single iteration of the outer loop. For each candidate, when applicable, we give the evaluation of the condition on line 6, the point p added to P if that condition is false (line 15), the minimal candidate set C_{min} constructed on line 7, the evaluation of the condition on line 8, the set of predicates added to our set of unsatisfiable cores if that condition is false (line 11), and finally the formula (if any) returned as a solution (line 9).

#	$e \in \mathsf{E}$	C	line 6	$p \in \mathsf{P}$	C_{min}	line 8	$u \in \mathsf{U}$	return
1	$x \geqslant 0$	$\{x \geqslant 0\}$	$\bot$	$(0, 0, -1)$				
2	$x < 0$	$\{x < 0\}$	$\bot$	$(-1, 0, 0)$				
		$\{x < 0, x \geqslant 0\}$	$\top$		C	$\bot$	$\{x < 0, x \geqslant 0\}$	
3	$y \geqslant 0$							
4	$y < 0$	$\{y < 0\}$	$\top$		C	$\bot$	$\{y < 0\}$	
5	$z \geqslant 0$	$\{x \geqslant 0, z \geqslant 0\}$	$\top$		C	$\top$		$x \geqslant 0 \wedge z \geqslant 0$

We assume the same ordered list of predicates enumerated from Figure 1. On the first iteration, we add $x \geqslant 0$ to our pool of enumerated predicates E. The helper function EnsureCexFalsify constructs the set $\mathsf{C} = \{x \geqslant 0\}$ since (vacuously) it is true for all points in P. Similar to the first iteration of Figure 1, on line 6 we learn that $x \geqslant 0$ does not suffice with our axioms to show the goal; a counterexample point is $(x, y, z) = (0, 0, -1)$ which is added to P. Afterwards, EnsureCexFalsify is not capable of constructing another C since there are no other predicates in E. In contrast to Figure 1 which discards the predicate $x \geqslant 0$ at this point, here it remains in E and can be added as part of C in future iterations.

On the second iteration, we add $x < 0$ to our pool. We check the candidate set $\mathsf{C} = \{x < 0\}$, which fails to imply the goal for counterexample point $(x, y, z) = (-1, 0, 0)$. To construct the next candidate set C, we must find an additional predicate from E that evaluates to false on this point (or otherwise we again would fail to imply our goal). Indeed, $x \geqslant 0 \in \mathsf{E}$ evaluates to false on this point, and thus EnsureCexFalsify returns the set $\{x < 0, x \geqslant 0\}$. This set suffices to prove the goal given the axioms, that is, the condition on line 6 succeeds; the unsatisfiable core C_{min} computed for this query is the same as C. However, on line 8, we learn that this set is inconsistent with our axioms (in fact, the set by itself is equivalent to false). On line 11, we add $\{x < 0, x \geqslant 0\}$ to U. In other words, we learn that *any* solution that contains both these predicates is inconsistent with our axioms. Learning this subset will help prune later candidate solutions. The procedure on this iteration proceeds by removing one of these predicates from our candidate solution set C. Subsequently the helper function EnsureCexFalsify cannot construct a new candidate subset due to $\{x < 0, x \geqslant 0\} \in \mathsf{U}$ and since no other predicates occur in E.

On the third iteration, $y \geqslant 0$ is added to our pool. However, no candidate solution can be constructed, where notice that $y \geqslant 0$ evaluates to $\top$ on both points in P. On the fourth iteration, $y < 0$ is added to our pool and the candidate solution set $\{y < 0\}$ is constructed, where notice that this predicate evaluates to $\bot$ on both points in P. This predicate suffices to show the goal from the axioms, but is however inconsistent with our axioms. Thus, $\{y < 0\}$ is added to our set of unsatisfiable cores U. In other words, we have learned that no solution C should include the predicate $y < 0$ since it is alone inconsistent with our axioms.

On the fifth iteration, $z \geqslant 0$ is added to our pool. The only viable candidate that falsifies all points in P and does not contained a subset from U is $\{x \geqslant 0, z \geqslant 0\}$. This set is a solution to the abduction problem and the formula $x \geqslant 0 \wedge z \geqslant 0$ is returned. Due to our assumption that R admits conjunctions, this formula meets the syntax restrictions of our grammar. A run of this procedure required only 5 predicates to be enumerated before finding a solution whereas the basic one in Figure 1 required 9. □

While the solution in the previous example $x \geqslant 0 \wedge z \geqslant 0$ was found in fewer iterations, notice that it is logically stronger than the solution $x + z \geqslant 0$ produced in Example 1, since $x \geqslant 0 \wedge z \geqslant 0$ entails $x + z \geqslant 0$ but not vice versa. We remark that the main advantage of procedure Figure 2 is that is typically capable of generating *any* feasible solution to the abduction problem faster than the procedure from Figure 1. This is especially the case if the only solutions to the abduction problem consist of a large conjunction of literals of small term size $\ell_1 \wedge \ldots \wedge \ell_n$. The basic procedure does

```
GetAbductInc(axioms A, goal G, grammar R)
 1: Let S = ⊥
 2: loop
 3:    Let C = GetAbduct*(A, G, R).
 4:    if C ∧ A ∧ ¬S is T-satisfiable then
 5:       S := S ∨ C
 6:       print Weakest solution so far is S
 7:    else
 8:       // Exclude {u} for some u ⊆ C such that u ∧ A ∧ ¬S is T-unsatisfiable
 9:    end if
10: end loop
```

Fig. 3. Incremental abduction procedure for axioms A, goal G and grammar R.

not scale to this case, since it would require waiting until the conjunction above was enumerated as a predicate.

Regardless, the user may be interested in obtaining a solution to abduction problem that maximizes some criteria and is not necessarily the first one discovered by (either of) the aforementioned procedures. In the next section, we describe an extension to our approach for abduction that maintains the advantage of returning solutions quickly while still seeking to generate the best solution in the long run. Namely, we extend these procedures so that it generates solutions to the abduction problem based on the above procedures, and moreover generates additional solutions over time that are maximize some criteria such as logical weakness.

4.3 Incremental Weakening for Abduction

We remark that it is straightforward to extend enumerative syntax-guided approaches for abduction to generate *multiple* solutions. In particular, we are interested in an approach that generates solutions over time that are progressively better in terms of some metric. We briefly give an overview of how the above procedures can be extended in this way and discuss some relevant details regarding this extension. We focus on the problem of generating the *logically weakest* solution to the abduction problem in this section.

Figure 3 presents an incremental procedure for generating (multiple) solutions to a given abduction problem. The procedure requires that R admits disjunctions, i.e. that $s_0 \rightarrow s_0 \vee s_0$ is a production rule in R where s_0 is the start symbol of R. It maintains a formula S, which when not $\bot$, represents the logically weakest solution to the abduction problem known so far. In the main loop of the procedure, it calls one of the procedures for generating single solutions to the abduction problem (written GetAbduct*) as a subprocedure on line 3. Line 4 of the procedure then checks whether a new solution can be constructed that is logically weaker with respect to the axioms than the current one. In particular, this is the case if C ∧ A ∧ ¬S is T-satisfiable. In other words, there is at least one point for which the current candidate satisfies that is not satisfied by the current solution S. If this is the case, the current solution S is updated to S ∨ C, which is by construction guaranteed to also be a solution to the abducion problem. If no such point

can be found, then C is redundant with respect to the current candidate solution since it does not weaken the current solution. Optionally, we may learn a subset u of C that is also redundant with respect to the current candidate solution. This subset can be learned as an unsatisfiable core in the case where we are using the procedure GetAbductUCL as a subprocedure on line 3.

4.4 Implementation Details

We implemented the above procedures in CVC4 [8], state-of-the-art SMT solver which has also been extended with several strategies for enumerative syntax-guided synthesis [31]. It supports inputs both in the SMT-LIB version 2.6 format [9], and synthesis problems in SyGuS version 2.0 format [30]. To specify an abduction problem, we extend its SMT-LIB version 2.6 parser. SMT-LIB version 2.6 is a scripting language where assertions may be provided via commands (assert F) where F is a formula. The solver is invoked to check for satisfiability of its current assertions with the command (check-sat). We extended CVC4's parser for this format to support commands of the form (get-abduct p G R) where p is a symbol (the name of the solution predicate), G is a formula (the goal of the abduction problem), and (optionally provided) R is a grammar in the SyGuS version 2.0 format. This command asks the solver to find a predicate that is a solution to an abduction problem, where the axioms is the solver's current assertions. The expected response from the solver is (define-fun p () Bool S) where p matches the symbol name provided in the first argument of get-abduct and S is a formula that is the solution to the abduction problem.

Internally, invoking a get-abduct command causes a synthesis conjecture to be constructed and passed to the SyGuS solver of CVC4. The SyGuS solver of CVC4 traditionally accepts conjectures of the form $\exists f.\, \forall \bar{x}.\, \varphi[f, \bar{x}]$ where φ is quantifier-free. Thus, we must pass the abduction problem in two parts: (i) the *conjecture* $\exists P.\, \forall \bar{x}.\, \neg(P(\bar{x}) \wedge A \wedge \neg G)$ where $\bar{x}$ are the free variables of A, G[4], stating that P along with our axioms must imply the goal, and (ii) a *side condition* $\exists \bar{x}.\, P(\bar{x}) \wedge A$ stating that P must be consistent with our axioms. The conjecture above is of a form that can be readily handled by the existing SyGuS solver of CVC4 and processed using its existing techniques. We have added additionally techniques so that the side condition is considered during solving, as described in Figures 1 and 2.

The procedure in Figure 2 is implemented as a strategy on top of the basic enumerative CEGIS loop of CVC4. We give some important implementation details here. Firsty, we use a data structure for efficiently checking whether any subset of C occurs in our set of unsatisfiable cores U, which keeps the sets in U in an index and is traversed dynamically as predicates are added to C. We chose enumerated predicates on line 2 of EnsureCexFalsify by selecting first the most recently generated predicate, and then a random one amongst those that meet the criteria to be included in C. Finally, since the number of candidate solutions can be exponential in the worst case for a given iteration of the inner loop of this procedure, we use a heuristic where predicates cannot be added to C more than once on the same iteration of the loop, making the number of candidate sets tried on a given iteration linear in the size of E in the worst case.

[4] We assume that all free symbols in A and G are variables.

5 Evaluation

We implemented our approach in the enumerative syntax-guided synthesis solver of CVC4 [31] and evaluated[5] it in comparison with CVC4's enumerative CEGIS, a general purpose synthesis approach, as well as with GPID [19] and EXPLAIN [15], state-of-the-art solvers for similar abduction problems as the one defined here. In the comparison below, we refer to the basic procedure from Figure 1 as CVC4+B and the one from Figure 2 as CVC4+U. Experiments ran on a cluster with Intel E5-2637 v4 CPUs, Ubuntu 16.04. Each execution of a solver on a benchmark was provisioned one core, 300 seconds and 8 GB RAM.

5.1 Benchmarks

Since abduction tools are generally focused on specific application domains, there is no standard language or benchmark library for evaluation. As here we did not target a specific application but rather the abduction problem as a whole, we had to generate our own general benchmark sets. We did so using benchmarks relevant for verification from SMT-LIB [9], the standard test suite for SMT solvers. We chose as a basis the SMT-LIB logics QF_LIA, QF_NIA, and QF_SLIA due to their relevance for verification. For QF_NIA, we focus on the benchmark family VeryMax and on kaluza for QF_SLIA. In QF_LIA we excluded benchmark families whose benchmarks explode in size without the let operator. This was necessary to allow a comparison with EXPLAIN, whose parser does not fully support let, on let-free benchmarks. We considered both benchmarks that were (annotated as) satisfiable and unsatisfiable for generating abduction problems, according to the following methodology.

Given a *satisfiable* SMT-LIB problem $\varphi = \psi_1 \wedge \cdots \wedge \psi_n$[6] in the theory T, we see it as an encoding of a validity problem $\psi_1 \wedge \cdots \wedge \psi_{n-1} \models \neg\psi_n$ that could not be proven. We consider the abduction problem where G is $\neg\psi_n$, A is $\psi_1 \wedge \cdots \wedge \psi_{n-1}$, and R is a grammar that generates any quantifier-free formula in the language of T over the free variables of G and A. A solution S to this problem allows the validity of φ to be proven, since $\varphi \wedge \mathsf{S}$ is unsatisfiable.

Given an *unsatisfiable* SMT-LIB problem φ, let $U = \{\psi_1, \ldots, \psi_n\}$ be a *minimal* unsatisfiable core for this formula, i.e. any conjunctive set $U \setminus \{\psi\}$, for some $\psi \in U$, is satisfiable. Let $\psi_{\max}$ be U's component with maximal size. We will call $\psi_{\max}$ the *reference* to the abduction problem. We consider the abduction problem whose G is $\neg\psi_{\mathsf{G}}$, for some $\psi_{\mathsf{G}} \in U$ and $\psi_{\mathsf{G}} \neq \psi_{\max}$, whose axioms A are $U \setminus \{\psi_{\mathsf{G}}, \psi_{\max}\}$ and R as before is a grammar that generates any predicate in the language of T over the free variables of G and A. A solution S to this problem allows proving the validity of $U \setminus \{\psi_{\mathsf{G}}, \psi_{\max}\} \models \psi_{\mathsf{G}}$, since $U \setminus \{\psi_{\max}\} \cup \{\mathsf{S}\}$ is unsatisfiable. Solving this abduction problem amounts to "completing" the original unsatisfiable core with the further restriction that this completion is at least as weak as the reference, as well as consistent with all but one of the other core components, seen as axioms for the abduction problem.

[5] Full material at http://cvc4.cs.stanford.edu/papers/abduction-sygus/

[6] SMT-LIB problems are represented as sequences of assertions. Here we considered each ψ_i as one of these assertions.

From satisfiable SMT-LIB benchmarks we generated 2025 abduction problems in QF_LIA, 12214 in QF_NIA and 11954 in QF_SLIA. For unsatisfiable benchmarks we were limited not only by the benchmark annotations but also by being able to find minimal unsatisfiable cores. We used the Z3 SMT solver [14] to generate minimal unsatisfiable cores with a 120s timeout. Excluding benchmarks whose cores had less than three assertions (so we could have axioms, a goal and a reference), we ended up with 97 problems in QF_LIA, 781 in QF_NIA and 2546 in QF_SLIA.

		CVC4+B			CVC4+U		
Logic	#	Solved	Unique	Weaker	Solved	Unique	Weaker
QF_LIA	2025	721	261	183	594	134	2
QF_SLIA	11954	10902	3	466	10980	81	0
QF_NIA	12214	1492	171	671	1712	391	45
Total	26593	13329	435	1320	13628	606	47

Table 1. Comparison of abduction problems from originally SAT SMT-LIB benchmarks.

5.2 Finding missing assumptions in SAT benchmarks

In this section we evaluate how effective CVC4+B and CVC4+U are in (i) finding any solution to the abduction problem and (ii) finding logically weak solutions. The evaluation is done on the abduction problems produced from satisfiable SMT-LIB benchmarks as above. Results are summarized in Table 1. The number of solved problems corresponds to the problems for which a given CVC4 configuration could find a solution within 300s. CVC4+U solves a significant number of problems more than CVC4+B in all logics but QF_LIA. In both QF_LIA and QF_NIA we can see a significant orthogonality between both approaches. We attribute these both to the fragility of integer arithmetic reasoning, where the underlying ground solver checking the consistency of candidate solutions is greatly impacted by the shape of the problems it is given. Overall, the procedure in CVC4+U leads to a better success rate than the basic procedure in CVC4+B. Solution strength was evaluated considering the solutions produced according to the incremental procedures shown in Section 4.3 on commonly solved problems. As expected, CVC4+U is able to solve more problems but at the cost of often producing stronger solutions than CVC4+B. This is particularly the case in QF_SLIA and QF_NIA, in which CVC4+U both solves many more problems and often finds stronger solutions.

5.3 Completing UNSAT cores

Here we evaluate how effective CVC4+B and CVC4+U are in solving the abduction problem with the extra restriction of finding a solution that is at least as weak as a given reference formula. We use the abduction problems produced from unsatisfiable SMT-LIB benchmarks following the methodology of Section 5.1 as the basis for this evaluation.

The results are summarized in Table 2. CVC4+B significantly outperforms CVC4+U in QF_SLIA, in which the references are very simple formulas (generally with size

		CVC4+B		CVC4+U	
Logic	#	Solved	Unique	Solved	Unique
QF_LIA	97	6	0	6	0
QF_SLIA	2546	2546	32	2514	0
QF_NIA	781	86	49	41	4
Total	3424	2638	81	2561	4

Table 2. Comparison of abduction problems from originally UNSAT SMT-LIB benchmarks.

below 3), for which the specialized procedure of CVC4+U is not necessary. Overall, as in the previous section when checking who finds the weakest solution, CVC4+B has as advantage over CVC4+U for finding solutions as weak as the reference.

5.4 Comparison with Explain

EXPLAIN [15] is a tool for abductive inference based on quantifier elimination. It accepts as input a subset of SMT-LIB and we extended it to support abduction problems as generated in Section 5.1. However, EXPLAIN imposes more restrictions to their solutions, only producing those with a *minimal* number of variables and for which every other solution with those variables is not stronger than it. Their rationale is finding "simple" solutions, according to the above criteria, which are more interesting to their applications. Since we do not apply these restrictions in CVC4, nor is in the scope of this paper incorporating them into our procedure, it should be noted that comparing CVC4 and EXPLAIN puts the latter at a disadvantage. We considered satisfiable SMT-LIB problems in the QF_LIA logic for our evaluation, as QF_LIA is better supported by EXPLAIN.

	Solved	Unique	Total time
CVC4+B	721	261	418849s
CVC4+U	594	125	449424s
EXPLAIN	33	0	532839s

Table 3. Comparison with EXPLAIN in 2025 abduction problems in QF_LIA

All problems solved by EXPLAIN are solved by CVC4+U. Of these 33 problems, CVC4+U, in incremental mode, finds a solution with the same minimal number of variables as EXPLAIN for 25 of them. Of the 8 problems to which it only finds solutions with more variables, in 4 of them the difference is of a single variable. All other 4 are in the slacks benchmark family, which contains crafted problems. A similar comparison occurs with CVC4+B. This shows that even though CVC4 is not optimized to minimize the number of variables it its solutions, it can still often finds solutions that are optimal (or close to optimal) according to EXPLAIN's criteria, while solving a much larger number of problems with a fully general approach.

5.5 Comparison with GPiD

We also compared CVC4 with GPID [19], a framework for generating implicates, i.e. logical consequences of formulas. As Echenim et al. say in their paper, negating the

implicate of a satisfiable formula φ yields the "missing hypothesis" for making φ unsatisfiable. Therefore GPID solves a similar problem to that of Section 5.2, differing by they always considering an empty set of axioms and the whole original formula as the goal. Given this similarity, we compare the performance of GPID in generating implicates for satisfiable benchmarks and of of CVC4+B and CVC4+U in solving abduction problems generated from those same benchmarks. We did not consider the benchmarks from the previous sections because we were not able to produce *abduces*, which are the syntactic components GPID uses to find implicates, for other logics using the tools in GPID public repository[7]. Thus we restricted our analysis to 400 abduction problems produced, as per the methodology of Section 5.1, from satisfiable QF_UFLIA benchmarks that were used in [19]. Note however that the CVC4 configurations will require solutions to be consistent with all but the last assertion in the problems (which are the axioms in the respective abduction problem). Since that, as far as we know, this is not a requirement in GPID, effectively CVC4+B and CVC4+U are solving a harder problem than GPID. We formulated the abduction problem this way, rather than as with all assertions as goals, to avoid trivializing the abduction problem, for which the negation of the goal would always be a solution. Also note that the presence of uninterpreted functions in the abduction problem requires solutions to be generated in a higher-order background logic, which CVC4 supports after a recent extension [7]. As in [19], we used GPID's version with the Z3 backend. We present their results with (GPID-1) and without (GPID) the restriction to limit the set of abduces to size 1.

	Solved	Unique	Total time
CVC4+B	214	0	57290s
CVC4+U	342	0	18735s
GPID	193	0	69s
GPID-1	398	54	1188s

Table 4. Comparison with GPID on 400 abduction problems in the QF_UFLIA logic.

Results are summarized in Table 4. CVC4+U significantly outperforms CVC4+B, both in the number of problems solved and in total time, besides being almost 20% faster on commonly solved problems. We also see that solution finding in GPID is heavily dependent on which abduces are considered when building solutions, as it solves almost all benchmarks when limited to abduces of size 1 but barely half when unrestricted. It should also be noted that GPID takes *pre-computed* abduces, whose production time is not accounted for in the evaluation. Despite this, CVC4+U is only on average 30% slower on commonly solved problems than GPID-1 and solves many more problems than GPID. The big variation of GPID results in terms of what pre-determined set of candidates can be used in the computation is a severe limitation of their tool. Similarly, while the method proposed in [19] is theory agnostic, their tooling for producing abduces imposes strong limitations on the usage of GPID for theories other than QF_UFLIA.

[7] At https://github.com/sellamiy/GPiD-Framework .

6 Related Work

The procedure introduced in Section 4.2 based on unsat core learning follows a recent trend in enumerative syntax-guided synthesis solving that aims to improve scalability by applying divide-and-conquer techniques, where candidate solutions are built from smaller enumerated pieces rather than being directly enumerated. While previous approaches, both for pointwise [27, 5] and for unrestricted specifications [6], have targeted general-purpose function synthesis, we specialize divide and conquer for solving the abduction problem with a lean (see Section 4.4) and effective (see Section 5) procedure.

Abductive inference tools for the propositional case include the AbHS and AbHS+ tools [26, 33], based on SAT solvers [11] and hitting set procedures [23]. More general approaches, to which our work bears more resemblance and to which we provided an experimental comparison in Section 5, are GPiD [19] and Explain [15]. GPiD uses an off-the-shelf SMT solver as a black box to generate ground implicates. It can be used with any theory supported by the underlying SMT solver, similarly to our SyGuS-based approach. While we enumerate predicates that compose the solution for the abduction problem they use *abducibles*, which are equalities and disequalities over the variables in the problem. They similarly build candidates in a refinement loop by combining abducibles according to consistency checks performed by an underlying SMT solver. They use an order on abducibles to guide the search, which is analogous to the enumeration order in enumerative synthesis. Explain on the other hand is built on top of an SMT solver for the theories of linear integer arithmetic and of equality with uninterpreted functions, but their abduction inference procedure in principle can work with any theory that admits quantifier elimination. Their method is based on first determining a subset of the variables in the abduction problem and trying to build the weakest solution over these variables via quantifier elimination, while computing minimal satisfying assignments to ensure that a found solution covers a minimal subset. Their method however is not complete, as it can miss solutions. Their tool also allows the user to specify costs for each variable, so that a given minimal set may be favored.

7 Conclusion

We have described approaches for solving the abduction problem using a modern enumerative solver for syntax-guided synthesis. Our evaluation shows that procedures based on enumerative CEGIS scale for several non-trivial abduction tasks, and have several compelling advantages with respect to other approaches like those used in EXPLAIN and GPID. In several cases, it suffices to use a basic procedure for enumerative CEGIS to generate solutions to abduction problems that are optimal according to certain metrics. Moreover, the generation of feasible solutions can be complemented and accelerated via a procedure for generating conjunctions of enumerated predicates as shown in Figure 2.

We see a number of promising applications of the new abduction capabilities described in this paper. For example, we plan to use abduction to develop *conditional rewrite rules* in CVC4. Abduction can be used to generalize a recent approach for the semi-automated development of rewrite rules [28] by synthesizing (most general) conditions under which two terms are equivalent. This in turn can be used to develop new solving strategies in the SMT solver based on those rewrite rules.

References

[1] A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal specification synthesis. *ACM SIGPLAN Notices*, 51(1):789–801, 2016.

[2] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods In Computer-Aided Design (FMCAD)*, pages 1–8. IEEE, 2013.

[3] R. Alur, P. Cerný, and A. Radhakrishna. Synthesis through unification. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification (CAV)*, volume 9207 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2015.

[4] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama. Sygus-comp 2017: Results and analysis. In *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017*, pages 97–115, 2017.

[5] R. Alur, A. Radhakrishna, and A. Udupa. Scaling enumerative program synthesis via divide and conquer. In A. Legay and T. Margaria, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 10205 of *Lecture Notes in Computer Science*, pages 319–336, 2017.

[6] H. Barbosa, A. Reynolds, D. Larraz, and C. Tinelli. Extending enumerative function synthesis via SMT-driven classification. In C. W. Barrett and J. Yang, editors, *Formal Methods In Computer-Aided Design (FMCAD)*, pages 212–220. IEEE, 2019.

[7] H. Barbosa, A. Reynolds, D. E. Ouraoui, C. Tinelli, and C. W. Barrett. Extending SMT solvers to higher-order logic. In P. Fontaine, editor, *Proc. Conference on Automated Deduction (CADE)*, volume 11716 of *Lecture Notes in Computer Science*, pages 35–54. Springer, 2019.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[9] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[10] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, 2009.

[11] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[12] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 289–300, 2009.

[13] A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *J. Artif. Intell. Res. (JAIR)*, 40:701–728, 2011.

[14] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[15] I. Dillig and T. Dillig. Explain: A tool for performing abductive inference. In N. Sharygina and H. Veith, editors, *Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 684–689. Springer, 2013.

[16] I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. *ACM SIGPLAN Notices*, 47(6):181–192, 2012.

[17] I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive invariant generation via abductive inference. *Acm Sigplan Notices*, 48(10):443–456, 2013.

[18] T. Dillig, I. Dillig, and S. Chaudhuri. Optimal guard synthesis for memory safety. In *International Conference on Computer Aided Verification*, pages 491–507. Springer, 2014.

[19] M. Echenim, N. Peltier, and Y. Sellami. A generic framework for implicate generation modulo theories. In D. Galmiche, S. Schulz, and R. Sebastiani, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 10900 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2018.

[20] M. Echenim, N. Peltier, and Y. Sellami. Ilinva: Using abduction to generate loop invariants. In A. Herzig and A. Popescu, editors, *Frontiers of Combining Systems (FroCoS)*, volume 11715 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2019.

[21] H. B. Enderton. *A mathematical introduction to logic*. Academic Press, 2 edition, 2001.

[22] S. Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 9706 of *Lecture Notes in Computer Science*, pages 9–14. Springer, 2016.

[23] A. Ignatiev, A. Morgado, and J. Marques-Silva. Propositional abduction with implicit hitting sets. In *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, pages 1327–1335, 2016.

[24] J. R. Josephson and S. G. Josephson, editors. *Abductive Inference: Computation, Philosophy, Technology*. Cambridge University Press, 1994.

[25] B. Li, I. Dillig, T. Dillig, K. McMillan, and M. Sagiv. Synthesis of circular compositional program proofs via abduction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 370–384. Springer, 2013.

[26] E. Moreno-Centeno and R. M. Karp. The implicit hitting set approach to solve combinatorial optimization problems with an application to multigenome alignment. *Operations Research*, 61(2):453–468, 2013.

[27] D. Neider, S. Saha, and P. Madhusudan. Synthesizing piece-wise functions by learning classifiers. In M. Chechik and J. Raskin, editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 9636 of *Lecture Notes in Computer Science*, pages 186–203. Springer, 2016.

[28] A. Nötzli, A. Reynolds, H. Barbosa, A. Niemetz, M. Preiner, C. W. Barrett, and C. Tinelli. Syntax-guided rewrite rule enumeration for SMT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, pages 279–297, 2019.

[29] P. M. Phothilimthana, A. Thakur, R. Bodík, and D. Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 297–310, 2016.

[30] M. Raghothaman, A. Reynolds, and A. Udupa. The sygus language standard version 2.0, 2019.

[31] A. Reynolds, H. Barbosa, A. Nötzli, C. Barrett, and C. Tinelli. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In I. Dillig and S. Tasiran, editors, *Computer Aided Verification (CAV), Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 74–83, Cham, 2019. Springer International Publishing.

[32] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification (CAV)*, volume 9207 of *Lecture Notes in Computer Science*, pages 198–216. Springer, 2015.

[33] P. Saikko, J. P. Wallner, and M. Järvisalo. Implicit hitting set algorithms for reasoning beyond NP. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, pages 104–113, 2016.

[34] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In V. Sarkar and M. W. Hall, editors, *Conference on Programming Language Design and Implementation (PLDI)*, pages 281–294. ACM, 2005.

[35] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In J. P. Shen and M. Martonosi, editors, *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415. ACM, 2006.

[36] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. TRANSIT: specifying protocols with concolic snippets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 287–296, 2013.

[37] H. Zhu, T. Dillig, and I. Dillig. Automated inference of library specifications for source-sink property verification. In *Asian Symposium on Programming Languages and Systems*, pages 290–306. Springer, 2013.