Counterexample-Driven Genetic Programming

Krzysztof Krawiec
Poznan University of Technology
Poznan, Poland
krawiec@cs.put.poznan.pl

Iwo Błądek
Poznan University of Technology
Poznan, Poland
ibladek@cs.put.poznan.pl

Jerry Swan
University of York
York, UK
jerry.swan@york.ac.uk

ABSTRACT

Genetic programming is an effective technique for inductive synthesis of programs from training examples of desired input-output behavior (tests). Programs synthesized in this way are not guaranteed to generalize beyond the training set, which is unacceptable in many applications. We present Counterexample-Driven Genetic Programming (CDGP) that employs evolutionary search to synthesize provably correct programs from formal specifications. CDGP employs a Satisfiability Modulo Theories (SMT) solver to formally verify programs in the evaluation phase. A failed verification produces counterexamples that are in turn used to calculate fitness and so drive the search process. When compared against a range of approaches on a suite of state-of-the-art specification-based synthesis benchmarks, CDGP systematically outperforms them, typically synthesizing correct programs faster and using fewer tests.

CCS CONCEPTS

•Software and its engineering → Genetic programming; •Theory of computation → *Evolutionary algorithms; Program verification;*

KEYWORDS

Genetic Programming; Program Synthesis; Formal Verification; Counterexample; SAT; SMT

ACM Reference format:
Krzysztof Krawiec, Iwo Błądek, and Jerry Swan. 2017. Counterexample-Driven Genetic Programming. In *Proceedings of GECCO '17, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3071178.3071224

1 INTRODUCTION

Genetic Programming (GP) proves effective for *test-based* synthesis of programs, where the synthesis task is defined by a set of tests (examples). Each example is an (*input*, *output*) pair, consisting of input to be fed into a program and the corresponding expected (correct) output. There are numerous settings in which this approach proves useful, most of them involving GP as a machine learning tool within the learning-from-examples paradigm.

The main shortcoming of test-based synthesis is that generalization cannot be guaranteed: a program synthesized from a finite training sample cannot be expected to return the correct value for arbitrary admissible input. In GP, this shortcoming can be partially alleviated via *ad-hoc* techniques such as parsimony pressure, but (excepting specific areas like semantic GP) GP lacks a general formal theory of generalization, comparable to the Probably Approximately Correct framework [37] or other algorithmic learning theories. The ability to generalize from the training to the test set remains a matter of chance, rather than something assured by the mechanisms of GP.

Even if GP had such a theory, it would not guarantee perfect generalization, as in the case of any other inductive learning approach. Assuring correct behavior for all admissible program inputs can be achieved only when synthesizing from formal specifications (*spec-based synthesis* in the following). At a systems-level, specifications are expressed via a variety of specification languages, including algebraic [16] and model-based [8, 38]. We are concerned here with a specific simple form of specification, termed a *contract* [19, 33]. Contracts are typically given as a pair of logical clauses defining (i) the inputs that a program can process and (ii) the conditions the response (output) of a program should fulfill. There are a variety of approaches to spec-based synthesis, including the stepwise transformation of the specification into a program (*specification refinement* [12]) or using the specification to constrain the space of program candidates and prioritize a search process conducted in that space (e.g. by first rephrasing the synthesis task as a satisfiability problem [15, 22, 36]).

Crucially, a program produced by spec-based synthesis is guaranteed to adhere perfectly to the specification. In many areas, such guarantees are essential. Examples include security, transportation, safety-critical systems, and costly manufacturing. The list of potential application areas for such methods is growing rapidly, particularly given the increasing level of cyberthreats and degree of responsibility delegated to computer systems.

Given the effectiveness of GP on test-based problems and guarantees offered by programs synthesized from specification, it becomes natural to ask: can the evolutionary paradigm be adapted to solve spec-based synthesis problems? Preliminary attempts on specific classes of executable structures [23] and programs [27], which we review in Section 4, suggest several possibilities. In this paper, we propose Counterexample-Driven Genetic Programming (CDGP), a novel variant of GP for solving spec-based synthesis problems. After presenting the preliminaries of spec-based synthesis and verification in Section 2, we describe the CDGP algorithm (Section 3), review the related work (Section 4), empirically examine its properties on a suite of benchmarks typically used to assess other spec-based synthesis algorithms (Section 5), closing with discussion and conclusions in Sections 6 and 7.

GECCO '17, Berlin, Germany
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. 978-1-4503-4920-8/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3071178.3071224

2 SPEC-BASED SYNTHESIS AND VERIFICATION

Spec-based program synthesis typically proceeds from a *contract*, given by a pair of logical formulas: a *precondition Pre* – the constraint imposed on program input, and a *postcondition Post* – a logical clause that should hold upon program completion. Let p denote a program and $p(in)$ the output produced by p when applied to input in. Solving a synthesis task $(Pre, Post)$ is equivalent to proving that

$$\exists_p \forall_{in} Pre(in) \implies Post(in, p(in)), \tag{1}$$

where $Pre(in)$ is the precondition valuated for the input in, and $Post(in, p(in))$ is the postcondition valuated for the input in and the output produced by p for in. Obviously, the proof has to be constructive, i.e. to produce such a p – merely determining whether or not p exists is not much use for synthesis.

Consider synthesizing a program that calculates the maximum of two integers (x, y). For this synthesis task, the contract can be defined as follows:

$$\begin{aligned} Pre((x,y)) &\iff (x,y) \in \mathbb{Z}^2 \\ Post((x,y),o) &\iff o \in \mathbb{Z} \land o \geq x \land o \geq y \land \\ &\land (o = x \lor o = y) \end{aligned} \tag{2}$$

This is an example of *a complete specification*, which defines the desired behavior of the sought program for *all possible inputs*, the number of which happens to be infinite here.

In methods of spec-based synthesis, the content (code) of p is controlled by a set of variables. For instance, programs represented as sequences of n instructions can be encoded with n such variables, each in $[1, k]$, where k is the number of available instructions. To determine the values of variables that cause p fulfill (1) (called a *model* in propositional logic), the synthesis formula parameterized with these variables is passed to a SAT *solver*. The solver either produces a feasible set of variable assignments, and thus yields a correct-by-construction program that is guaranteed to meet the contract, or otherwise states that the specified program does not exist. In practice, the solver is equipped with an additional abstraction layer, a *theory* that enables reasoning in terms of, for instance, integer arithmetic. This leads to the concept of *Satisfiability Modulo Theories* (SMT) used in program synthesis [15, 22, 36].

It is worth emphasizing that the solver achieves this without actually running any program, because the properties of the output can be logically inferred *modulo the theory*, from the properties of the input and the those of the program code. For the above problem (2) to be solved, the theory of Linear Integer Arithmetic (LIA) [7] may be used. Unfortunately, the cardinality of the search space grows exponentially with program length n, so only relatively short programs can be synthesized in this way.

3 COUNTEREXAMPLE-DRIVEN GP

GP is a stochastic generate-and-test technique, where new programs are continuously generated and evaluated on examples. In the test-based setting, the input to GP is a set of tests (fitness cases), i.e. , pairs $(in, out) \in T$ of program input in and the desired output out required to result from applying a correct program to in. A GP algorithm solving a synthesis task maintains a population of programs P. In every generation, each program $p \in P$ is tested on every test $(in, out) \in T$, in which p is applied to in and returns an output $p(in)$ that is confronted with out. If p produces the correct output for t, it is said to *pass t*; otherwise, we say that *p fails t*. The conventional GP fitness that rewards a program for the number of passed tests can be then written as

$$\textsc{Eval}(p, T) = \sum_{(in,out) \in T} [p(in) = out], \tag{3}$$

where [ ] is the Iverson bracket.

Algorithm 1 Evaluation in CDGP, given the current population P, current set of tests T_c, and program specification $(Pre, Post)$, returns the evaluated population and an updated set of tests.

```
procedure CDEval($P$, $T_c$, $(Pre, Post)$)
    $T \leftarrow \emptyset$                                ▷ Working set of tests
    for all $p \in P$ do                                   ▷ Evaluation loop
        $p.eval \leftarrow$ Eval($p$, $T_c$)
        if $p.eval = |T_c|$ then
            $c \leftarrow$ Verify($p$, $(Pre, Post)$)
            if $c = \emptyset$ then return $p$             ▷ Perfect program
            else
                $T \leftarrow T \cup \{c\}$
            end if
        end if
    end for
    return $(P, T_c \cup T)$
end procedure
```

In spec-based synthesis, tests are not available, and so neither is the conventional fitness function. To combine such synthesis with evolutionary search, one must resort to other means of program evaluation. The method presented in this paper relies on *program verification* which consists in proving that, for a given program p,

$$\forall_{in} Pre(in) \implies Post(in, p(in)). \tag{4}$$

The practical difference with respect to spec-based synthesis (1) is that verification can be typically realized using conventional SMT solvers at much lower computational cost, because it is applied to an existing program. The result can be twofold: success when p meets (4), or failure otherwise. Crucially, the latter outcome is accompanied by a *counterexample*, i.e. an input in such that (4) does *not* hold. This characteristic is essential for our method.

The top-level loop of Counterexample-Driven GP (CDGP) proceeds in a similar manner to conventional GP, where in each generation parent programs are selected, modified, and evaluated. The main difference is that evaluation involves *both* formal verification and evaluation on a set of tests T_c, collected from verifications conducted in the previous generations. The evolutionary run starts with an empty T_c. Programs in the working population are evaluated on the tests in T_c in the conventional way, and the resulting fitness drives the search process.

The procedure CDEval presented in Algorithm 1 is launched once per generation. In contrast to conventional GP evaluation, it accepts the formal specification $(Pre, Post)$ as an extra parameter. As in conventional GP, each program p in the current population is first evaluated on the tests currently available in T_c and assigned the conventional fitness (3). If it happens to pass all of them, it is subject to formal verification. A positive outcome of verification terminates search, with p returned as the resulting correct program. Otherwise, the counterexample resulting from verification extends the working set of tests T, which is maintained separately from T_c so that evaluation of successive programs in P remains unaffected. Once all programs have been processed, T_c is extended with the new tests from T. Since both T_c and T are *sets*, adding a test that has been already collected earlier in a run has no effect — duplicate tests are automatically discarded.

In the first generation $T_c = \emptyset$, so all programs in P receive zero fitness and the attendant selection of parent programs is completely random. Nevertheless, this first generation will typically discover a few counterexamples, which provide for some degree of discrimination of programs in the second generation. In this way, the verification outcomes supply CDGP with an increasingly finer-grained fitness function and more precise search gradient.

For brevity, Algorithm 1 omits a technical yet important detail. Eval expects in T_c complete tests of the form (in, out), i.e. program inputs accompanied with corresponding desired outputs. However, the counterexamples resulting from Verify are not tests – each of them is just the input in that caused p to fail verification. Thus, the tests added to T in line 9 have the form $c = (in, \emptyset)$.

As a consequence, Eval needs to evaluate programs on both well-formed tests (which it does in the conventional way, as in (3)), as well as on incomplete tests of the form $(in, \emptyset)$. In the latter case, Eval first applies the evaluated program to in, obtaining the actual output out_a. Then, it validates $Post$ on out_a, i.e. substitutes the variables in the predicate $Post$ with the values from out_a and determines if the resulting Boolean formula is true (cf. (2)). If $Post(in, out_a)$ holds, then out_a is the correct output for in, and Eval replaces $(in, \emptyset)$ in T with (in, out_a).[1] In future evaluations on this tests, Eval can conventionally compare the actual program output with the desired one, which is cheaper than validating $Post$. In this way, the missing information on desired outputs for individual tests is gradually supplemented by the algorithm at moderate computational overhead.

Algorithm 1 implements the 'conservative' variant of CDGP, where the time-costly verification is applied only to programs that pass all tests and is thus used sparingly. Attempting verification of a program that is known to fail any tests in T_c may indeed seem pointless, as such a program cannot be correct. However, a counterexample resulting from such verification does not have to be identical to any of the tests already in T_c, because solvers implement complex tactics, and the outcome of verification may depend on the specification and program code in a nontrivial way.

This suggests that incorrect programs in the population may also give rise to *new* counterexamples. Whether the tests built from them are useful at guiding search or not remains an open question. To answer this, we also investigate a *non-conservative* variant of CDGP, where line 5 in Algorithm 1 is skipped, so that each act of evaluating a program is followed by its verification and will produce a counterexample that may extend T_c (unless already present there).

4 RELATED WORK

The application of formal methods to program synthesis precedes heuristically-informed stochastic methods such as GP by several decades [13], and the literature for formal approaches to synthesis (and verification) is vast (for recent overviews, see Boca et al [9] and Almeida et al [1]). However, we are aware of only few approaches which combine formal techniques with heuristic search.

In 2007, Johnson [23] incorporated model-checking (as specified via Computation Tree Logic) into the fitness measure of evolved finite state machines, and used this to learn a controller for a simple vending machine. From 2008, Katz and Peled authored a series of papers combining model-checking and GP [25–27] in which they progressively refine their MCGP tool [26], based on Linear Temporal Logic. They use 'deep model checking' to impose a gradient on the fitness function, for which they report good fitness-distance correlation. The most recent development of their tool [27] applies a $(\mu + \lambda)$ evolutionary strategy to strongly-typed, tree-based tree GP and gives example applications of program synthesis, program improvement and bug-repair.

In common with our approach, Katz and Peled use feedback from the verification process to provide a finer-grained measure of program fitness. In contrast, CDGP formally ensures correctness, whereas their approach does not achieve this in all cases (e.g. in their application to synthesising concurrent programs). Verification in our case is performed via SMT, in theirs via model-checking. Last but not least, in Section 5 we combine CDGP with the recently proposed Lexicase selection [18], and the domain-agnostic effectiveness of CDGP is demonstrated on a wider range of benchmarks.

The possibility of using coevolutionary GP to synthesize programs from formal specifications was researched by Arcuri and Yao [4, 5]. In their approach, populations of both tests and programs are maintained in the competitive coevolution framework. Fitness of programs is calculated using a heuristic that estimates how close a postcondition is from being satisfied by the program's output for specific tests. The population of test cases is initialized randomly and then co-evolves with programs, guided by fitness function that rewards failing as many programs as possible. The approach of Arcuri and Yao, while allowing the synthesis of programs with GP from a formal specification, provides no guarantees that program deemed correct by their method will be consistent with the specification for all possible inputs.

Amongst the dozen or more well-known systems that perform synthesis under the heading of Inductive Logic Programming [34], IGOR II [20] is known to perform well on a range of problems [21]. As extended by Katayama [24], it combines an 'analytic' approach based on analysis of fitness cases with the generate-and-test approach more familiar to the GP community.

In the related area of Genetic Improvement, there have been a number of recent articles incorporating formal approaches. Kocsis et al. [29] report a 10,000-fold speedup of Apache Spark database queries on terabyte datasets. In work by Burles et al [10], a 24% improvement in energy consumption was achieved for Google's Guava collection library by applying the Liskov substitution principle that is the formal cornerstone of object-orientation. Some recent work has also used category theory to perform formal transformations on datatypes [28, 30], in order to join together parts

[1] In general there may be other outputs that satisfy the specification for a given in; however we assume that $(Pre, Post)$ specifies a function in the mathematical sense: for a given input, only one correct output exists, and output depends deterministically solely on the input (i.e. there is no 'hidden' global state).

Table 1: Program synthesis benchmarks. For all of them, the input type is $\mathtt{I}^n$ and the output type is I (I=integer). Some functions were tested in variants with different arities.

Name	*Arity*	*Semantics*
CountPos	2, 3	The number of positive arguments
IsSeries	3	Do arguments form an arithmetic series?
IsSorted	4	Are arguments in ascending order?
Max	2, 4	The maximum of arguments
Median	3	The median of arguments
Range	3	The range of arguments
Search	2, 4	The index of an argument among the other arguments
Sum	2, 4	The sum of arguments

of a program which are otherwise unrelated, a technique that is applicable to 'Grow and Graft Genetic Programming' [17].

An alternative approach to spec-based synthesis is 'program sketching' [35], a technique whereby a program contains 'holes' which are automatically filled in (e.g. using an SMT solver) with values satisfying an executable specification. The method is known to be complete for finite programs (i.e. it can in principle synthesise any required value). However, the approach has limited scalability since the exact search method used is exponential in the number of variables. More recently, Evolutionary Program Sketching (EPS) has been proposed [11]. EPS is presented as a GP alternative that evolves partial programs then uses an SMT solver to complete them, attempting to maximize the number of passing test cases. For the small set of benchmarks under consideration, EPS outperforms conventional GP (e.g. in the number of optimal solutions found).

5 EXPERIMENT

To assess the effectiveness of CDGP, we apply it to a range of spec-based synthesis benchmarks of varying difficulty.

We consider the benchmarks presented in Table 1, all of which belong to the theory of Linear Integer Arithmetic (LIA) [7], where the set of available instructions comprises linear arithmetic, elementary Boolean logic and conditional statements. In all selected benchmarks, the task is to synthesize a certain function with a signature $\mathtt{I}^n \rightarrow \mathtt{I}$, where n is function arity. Max, Search and Sum come from the SyGuS repository maintained for the annual 'Syntax Guided Synthesis' competition [2, 3]; the remaining benchmarks are of our own design. Some benchmarks (IsSeries, IsSorted, Search) interpret input arguments as a fixed-size ordered sequence of type I. In the IsSeries and IsSorted tasks, the program is required to return 1 if the arguments respectively form an arithmetic series or are sorted in ascending order, 0 otherwise.

Figure 1 presents the specification of the Max4 benchmark expressed in the SMT-LIB language [6, 7]. The `synth-fun` statement defines the signature of the function to be synthesized. The `constraint` commands define the specification and are combined with logical conjunction by the solver. In this specific case, each `constraint` clause includes the reference to the synthesized function `max4`, which implies that this specification defines only the postcondition – the precondition is empty, i.e. the inputs to Max4 are only required to belong to the type `Int` (I in our framework).

```
(set-logic LIA)
(synth-fun max4 ((x Int) (y Int) (z Int) (w Int)) Int)

(declare-var x Int)
(declare-var y Int)
(declare-var z Int)
(declare-var w Int)

(constraint (>= (max4 x y z w) x))
(constraint (>= (max4 x y z w) y))
(constraint (>= (max4 x y z w) z))
(constraint (>= (max4 x y z w) w))
(constraint (or (= x (max4 x y z w))
        (or (= y (max4 x y z w))
        (or (= z (max4 x y z w))
        (= w (max4 x y z w))))))

(check-synth)
```

Figure 1: The Max4 benchmark expressed in the SMT-LIB language (fg_max4.sl file in the SyGuS repository). Compare with the specification of the max2 problem in Equation 2.

The specification for Max4 is *complete*, as its postcondition is applicable to all possible inputs (which holds for all benchmarks).

Although the benchmarks' semantics should be clear from Table 1, Search deserves a comment. In that benchmark, a correct program should find the 0-based index of the last argument ('the target') in an 'array' of length n formed by the remaining arguments (which are constrained by precondition to be sorted), such that having the target at that index results in the array being sorted. Hence, for instance `Search2(3,7,1)=0`, `Search2(3,7,4)=1` and `Search2(3,7,10)=2`, where index in the benchmark's name refers to the size of 'array'. [2]

In order to provide a frame of reference, we design a baseline setup called GP Random (GPR). GPR proceeds as CDGP, except for line 9 in Algorithm 1, where it adds to T a randomly generated test rather than the counterexample returned by the solver. In this way, the dynamics of GPR are similar to the conservative variant of CDGP, i.e. tests are added to T_c only when a program in population manages to pass all tests already in T_c. As in CDGP, multiple new tests may be added to T_c in a single generation, duplicates are eliminated, and T_c may grow indefinitely during a run. Comparison between CDGP and GPR allows us to determine whether the directed, spec-based synthesis of tests makes CDGP any better than generating them at random.

The LIA domain includes two types, `Int` (I) and `Boolean` (B), so a typed variant of GP is necessary to maintain their syntactic correctness. We rely on a straightforward approach that guarantees that programs initialized and bred within a run always conform with the grammar shown in Fig. 2. The initialization operator recursively traverses the derivation tree from the starting symbol of the grammar (I) and randomly picks expressions from the right-hand sides of productions. Once the depth of any node of the program tree reaches 4, the operator picks productions that immediately lead to terminals whenever possible. If the depth exceeds 5, it terminates, discards the tree, and starts anew.

[2] A Search$n$ benchmark thus diverges from the naming convention followed in the remaining benchmarks (i.e. the arity of the synthesized program is $n + 1$), but we do not address this for conformance with the SyGuS benchmark suite [2, 3].

```
I ::= I + I | I - I | ite(B,I,I) | -1 | 0 | 1
      | v_1 | v_2 | ... | v_n
B ::= and(B,B) | or(B,B) | not(B) | B = B
      | I < I | I <= I | I = I | I >= I | I > I
```

Figure 2: The grammar defining the set of considered programs (v_i – ith input variable, ite – *if-then-else*, the conditional statement). The starting symbol is I.

The mutation operator picks a random node in a parent tree, and replaces the subtree rooted in that node with a subtree generated in the same way as for initialization. To conform to the grammar, the process of subtree construction starts with the grammar production of the type corresponding to the picked location (e.g. if the return type of the picked node is I, generation of the replacing subtree starts with production I of the grammar).

Crossover draws a random node in the first parent program, and builds the list l of the nodes in the second parent that have the same type. If l is empty, it draws a node from the first parent again and repeats this procedure. Otherwise, it draws a node from l uniformly and exchanges the subtree identified in this way with the subtree drawn from the first parent. This is guaranteed to terminate, as both parent trees always feature at least one node of type (I), i.e. the root node, since it is also permissible for root nodes to be swapped.

A program tree resulting from any of these search operators is considered feasible unless its height exceeds 12. Should that happen, the program is discarded and the search operator is queried again.

Parameters of the evolutionary algorithm common to CDGP and GPR are shown in Table 2. In GPR, we draw tests uniformly from $[-100, 100]^n$. We anticipate that the width of this interval is not critical, given that in most benchmarks (except for Sum and IsSeries) the inputs of the function to be synthesized are interpreted as ordinal variables. While CDGP works with population size 500, we double it for GPR (1000), so that it has more chance of drawing important test cases (e.g. for border cases).

In both CDGP and GPR, the working set of tests T_c may grow slowly. With only a small number of tests, the fitness function can return just a few values and has little discriminatory power, which may hamper population diversity. To address this issue, in addition to the conservative and non-conservative CDGP and GPR, we consider two independent extensions:

1. **Lexicase selection**. The variants marked with '*Lex*' are equipped with *Lexicase selection* [18]. In these configurations, rather than returning the number of tests in T_c passed by p, Eval(p, T_c) returns a vector of $|T_c|$ Booleans indicating the detailed interaction outcomes, i.e. the ith element in that vector states whether p passed the ith test from T_c (cf. the beginning of Section 3). Consistently, line 5 in Algorithm 1 determines whether the number of *true*s in that vector is $|T_c|$. The vector of interaction outcomes is then used in selection phase, where each act of selection implements the algorithm described in [18]: a random test t is drawn from T_c without repetition, and all programs not passing t are discarded. Drawing tests and discarding programs is repeated until only one program is left (in which case it becomes selected); if all tests have been used, the winner is drawn uniformly from the remaining programs.

Table 2: Parameters of the evolutionary algorithm.

Parameter	*Value*
Number of runs	30
Population size	500/1000
Max. height of initial programs	5
Max. height of trees inserted by mutation	5
Max. height of programs in population	12
Max. number of generations	100
Probability of mutation	0.5
Probability of crossover	0.5
Tournament size	7

2. **Steady-state EA**. In the variants marked by the '*Steady*', the default generational GP is replaced with the steady-state evolutionary algorithm. Therein, a single iteration consists in first discarding a poorly-performing program from the population (using negative tournament selection of size 7), and then breeding a new program with a randomly chosen search operator (mutation or crossover). The program created in this way is immediately added to the current population. Crucially, it also undergoes verification as prescribed by Algorithm 1 (conditionally in conservative CDGP and GPR, always in non-conservative CDGP). Should that process result in a counterexample t_c that is not in T_c, it is immediately added to T_c. Subsequently, the fitness values of all programs in the population are updated by applying them to t_c, so that they are correct for the current contents of T_c. The key feature of the steady state approach is thus that fitness values of all programs in the population are updated promptly, as soon as new tests arrive. It might be anticipated that this would make the search process more reactive and result in faster synthesis.

CDGP and GPR can be extended in both of the above ways independently, so we consider all their combinations: tournament selection ($Tour$) vs. Lexicase selection (*Lex*), and generational evolution (*Gener*) vs. steady-state evolution ($Steady$). These combinations, together with conservative and non-conservative CDGP, give rise to eight configurations of CDGP and four configurations of GPR.

Communication with the solver is realized via the SMT-LIB standard [6], recognized by most contemporary SMT solvers. We employ the well known Microsoft Z3 SMT solver [14], one of the most performant and widely-used non-commercial solvers. This choice was arbitrary and no Z3-specific features were used. Our implementation of Verify($p, (Pre, Post)$) in Algorithm 1 translates our internal representation of program p into a function definition in the SMT-LIB language, combines it with the contract $(Pre, Post)$ retrieved from the benchmark (the constraint clauses in example in Fig. 1), and calls the solver to verify whether p meets $(Pre, Post)$. Solver then either returns a counterexample (inputs leading to failure) if p does not fully meet the specification, or signalizes that p is correct for all inputs. For the benchmarks considered in this study, this process is sufficiently swift that we do not limit the solver's execution time. The source code of CDGP, along with specifications of problems, is available at https://github.com/kkrawiec/CDGP.

Table 3 presents the success rates of particular variants of CDGP and GPR. We define success rate as the ratio of runs that end up with a positively verified (i.e. correct) program. The conservative

Table 3: Success rate of particular variants of CDGP.

	CDGP non-conservative				CDGP conservative				GPR			
	Generational		Steady-state		Generational		Steady-state		Generational		Steady-state	
	Tour	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*
CountPos2	1.00	1.00	1.00	1.00	0.93	1.00	0.83	1.00	0.97	1.00	0.97	1.00
CountPos3	0.77	1.00	0.60	1.00	0.07	0.63	0.03	0.53	0.40	1.00	0.27	1.00
IsSeries3	0.43	1.00	0.43	1.00	0.37	1.00	0.53	0.93	0.00	1.00	0.20	0.00
IsSorted4	0.70	1.00	0.67	1.00	0.27	0.77	0.33	0.93	0.43	1.00	0.33	0.57
Max2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Max4	0.83	1.00	0.97	1.00	0.23	0.97	0.20	0.93	1.00	1.00	0.93	1.00
Median3	0.83	1.00	0.73	1.00	0.20	0.80	0.10	0.87	0.80	1.00	0.80	1.00
Range3	0.47	1.00	0.53	1.00	0.33	0.87	0.17	0.70	0.80	1.00	0.67	0.92
Search2	1.00	1.00	1.00	1.00	0.73	0.97	0.63	1.00	0.57	1.00	0.47	0.65
Search4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Sum2	0.77	1.00	0.73	1.00	0.10	0.27	0.20	0.23	0.67	1.00	0.75	1.00
Sum4	0.00	0.12	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 4: Average runtime of particular variants of CDGP (in seconds).

	CDGP non-conservative				CDGP conservative				GPR			
	Generational		Steady-state		Generational		Steady-state		Generational		Steady-state	
	Tour	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*
CountPos2	37.4	14.7	73.7	49.8	6.5	7.3	30.4	31.1	263.0	636.5	2070.3	3157.1
CountPos3	341.2	98.7	597.7	505.4	39.2	86.7	76.1	281.5	2587.7	4304.5	5102.4	8048.3
IsSeries3	201.4	37.7	447.6	229.4	20.1	20.3	48.2	110.0	783.8	66255.9	2556.3	86402.5
IsSorted4	224.0	77.7	675.4	890.8	29.6	123.6	77.5	179.9	1213.2	13462.2	4256.9	55551.4
Max2	6.0	5.6	29.7	26.9	2.4	2.8	22.4	22.3	16.5	148.2	1532.4	1625.0
Max4	433.8	86.3	1136.2	908.2	33.6	71.3	73.2	192.1	367.2	353.2	2333.0	4383.1
Median3	274.7	82.5	661.7	555.0	34.5	89.4	70.4	229.8	731.7	788.4	2743.3	4143.5
Range3	579.5	100.6	934.9	587.0	23.5	37.1	56.6	158.0	991.0	3466.8	2528.4	20216.0
Search2	109.6	17.9	224.3	90.4	12.9	14.3	40.9	63.8	1218.2	11198.3	3503.4	44266.3
Search4	1326.8	23643.2	1990.8	54181.6	36.7	199.6	71.5	861.9	1539.3	67320.7	2392.8	86402.4
Sum2	151.6	48.5	395.3	330.9	23.7	92.3	51.6	209.9	1049.9	4946.0	2583.9	16110.5
Sum4	5567.0	32720.5	7527.6	50260.7	36.8	224.7	77.6	547.2	1952.0	7575.2	3733.3	22180.8

variants of CDGP are clearly worse than their non-conservative counterparts, which suggests that 'harvesting' new tests from candidate programs that are known to be incorrect is beneficial. Another clear pattern is the superiority of configurations equipped with Lexicase selection – this holds both for CDGP and GPR, and points to the importance of behavioral diversification of programs in the population. Concerning comparison of generational and steady-state configurations, introduction of the latter did not bring the benefits postulated earlier in Section 5, neither when combined with tournament selection nor with Lexicase selection. Regarding all methods' failure on Search4, this should be attributed to the factually higher arity of this problem (five).

Compared to GPR's baseline, CDGP offers on average greater likelihood of synthesizing a correct program. Yet, the quantitative differences are largely disappointing: the success rates on individual benchmarks are usually only slightly better for CDGP than for GPR, and on a few occasions GPR is better than the corresponding CDGP variant. What is more, Lexicase selection boosts the performance of all configurations and brings both CDGP$_{Lex}$ and GPR$_{Lex}$ close to each other, with both of them failing only on the Search4 benchmark and the latter yielding to the former only on Sum4.

On the face of it, this might seem to undermine the value of CDGP and the usefulness of counterexamples produced by verification. However, the results presented in Table 3 ignore the actual computational cost of synthesis. Although individual configurations were given the same limit on the number of evaluations (100,000), their runtimes reported in Table 4 vary heavily. There are several causes for this. On one hand, SMT-based formal program verification used in CDGP incurs significant computational overhead, being typically more expensive than running a program on tests (where the latter takes place in both by CDGP and GPR). On the other hand, the tests generated by GPR are random and thus less likely to be duplicates of the tests already in T_c, which usually makes this set grow faster than in CDGP, and make evaluation in subsequent generations more expensive.

In order to take into account these and other potential factors affecting the computational expenditure, we conduct another experiment, where each configuration of CDGP and GPR is given the same time budget equal to the average runtime *of successful runs* of all configurations from the first experiment, i.e. 158 seconds for Max2, 531s for CountPos2, 953s for Median3, 971s for Max4, and 1639s for IsSeries3. The average runtimes for the remaining benchmarks were higher and we decided to cap them to 1,800 seconds.

Table 5 presents the success rates for particular configurations in the fixed-time setting. This time, CDGP clearly proves more effective than GPR. Similarly as in Table 3, the non-conservative variants of CDGP that use Lexicase selection have the lead, though this time the generational variant is noticeably better.

The tests obtained from counterexamples thus prove more effective as a basis for a fitness function. This result is strengthened by Table 6, where we present the average size of T_c at the end of runs. The sizes are typically much smaller for CDGP, often several times. Yet, in spite of working with such smaller test bases, CDGP is more likely to synthesize a correct program.

Table 5: Success rate of particular variants of CDGP in the fixed-time setting.

	CDGP non-conservative				CDGP conservative				GPR			
	Generational		Steady-state		Generational		Steady-state		Generational		Steady-state	
	Tour	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*
CountPos2	1.00	1.00	1.00	1.00	1.00	1.00	0.97	0.97	0.83	0.60	0.00	0.00
CountPos3	0.80	1.00	0.63	1.00	0.67	0.97	0.60	0.87	0.13	0.67	0.00	0.00
IsSeries3	0.93	1.00	0.73	1.00	0.90	1.00	0.97	0.97	0.27	0.00	0.03	0.00
IsSorted4	0.97	1.00	0.90	0.97	0.97	1.00	0.97	0.93	0.50	0.03	0.07	0.00
Max2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.93	0.03	0.00
Max4	0.83	1.00	0.33	0.63	1.00	1.00	0.97	1.00	0.83	0.90	0.00	0.00
Median3	0.83	1.00	0.70	0.73	0.83	1.00	0.67	0.93	0.77	0.97	0.00	0.00
Range3	0.73	1.00	0.73	0.97	0.63	1.00	0.63	0.83	0.50	0.53	0.07	0.00
Search2	1.00	1.00	1.00	1.00	1.00	1.00	0.97	1.00	0.60	0.10	0.00	0.00
Search4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Sum2	0.97	1.00	1.00	1.00	0.50	0.43	0.13	0.27	0.43	0.10	0.20	0.03
Sum4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 6: End-of-run size of the set of tests T_c for particular variants of CDGP in the fixed-time setting.

	CDGP non-conservative				CDGP conservative				GPR			
	Generational		Steady-state		Generational		Steady-state		Generational		Steady-state	
	Tour	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*	*Tour*	*Lex*
CountPos2	60.6	55.9	59.2	48.2	24.9	23.1	24.1	24.7	1018.0	990.8	988.3	986.6
CountPos3	429.6	305.3	443.4	283.3	53.0	56.6	54.5	57.1	1000.1	1041.2	1000.0	999.9
IsSeries3	335.4	262.4	361.7	218.8	40.6	45.9	37.3	44.7	1109.4	1027.2	1001.3	1000.8
IsSorted4	753.9	520.8	647.5	409.1	64.8	75.5	60.7	74.1	1837.9	1000.0	1006.3	1000.0
Max2	36.5	41.4	30.2	28.2	24.6	23.6	23.2	23.3	987.6	988.6	987.9	987.7
Max4	1092.5	674.2	847.5	685.9	63.7	60.9	63.7	72.6	1077.2	1074.5	1000.0	1000.0
Median3	474.1	288.5	371.2	290.3	51.1	54.4	51.2	55.0	1023.1	1086.5	1000.0	999.9
Range3	527.9	277.0	482.3	274.7	39.6	42.9	39.6	41.1	1050.0	1007.3	999.9	999.9
Search2	143.1	108.5	149.7	104.0	29.1	29.9	28.6	29.1	1003.2	999.9	999.9	999.9
Search4	1137.3	805.9	988.3	578.8	85.1	237.3	77.0	78.0	1000.0	1000.0	1000.0	1000.0
Sum2	396.5	286.0	303.0	273.7	34.7	35.2	31.1	31.1	991.8	987.7	990.3	989.0
Sum4	1625.3	4045.1	1183.8	1362.7	52.5	85.9	55.4	70.7	1000.0	1000.0	1000.0	1000.0

6 DISCUSSION

The experimental outcomes corroborate our main hypothesis: the counterexamples collected from verification in CDGP prove more useful as tests than the inputs constructed at random in GPR. On one hand, this was expected – as opposed to counterexamples, random tests are not derived from the problem specification and are in this sense knowledge-free. On the other hand, this result is nontrivial, because counterexamples constructed by an SMT solver reflect its sophisticated search tactics, which are reportedly built on years of expert experience, and as such involve certain search biases. It is thus not obvious that counterexamples they identify should be effective when used as 'search drivers' [31] in a stochastic synthesis process.

On the technical side, it is interesting to see that SMT-based verification is efficient enough to form a part of a fitness function, called tens of thousands of times within a single evolutionary run. At least for the range of benchmarks considered here, that is not prohibitively expensive. SMT solvers support also verification in other domains, such as Reals, Strings or Lists. There are thus no fundamental obstacles to adding SMT solvers and spec-based synthesis to the GP toolbox.

As in other studies [18], Lexicase selection proved helpful, significantly boosting CDGP's performance on most tasks, and never leading to a deteriorated success rate. The likely explanation is that better exploration of the search space is afforded by the diversification of program behavior of this selection method.

7 CONCLUSION AND FUTURE WORK

We have presented CDGP, a method for specification-based program synthesis, via a hybrid of Genetic Programming and formal verification, in which the traditional evaluation phase of Genetic Programming is augmented using new test cases obtained via counterexamples generated from an SMT solver. The application of verification to a pre-existing program means that our formal/stochastic hybrid may be capable of performing synthesis at much lower computational cost than via conventional SMT-based methods.

The overall positive conclusion of this work paves the way for effective hybridization of heuristic search methods like GP with spec-based synthesis. We find this possibility promising, given the limitations of contemporary exact methods of program synthesis that struggle to scale well with the length of synthesized programs.

In operation, CDGP could be said to resemble the software engineering methodology of *test-driven development* ([32]), where a software developer iteratively constructs tests of gradually increasing difficulty, aimed at detecting flaws in the current implementation. This analogy holds also for other counterexample-driven methods [22, 35], and naturally brings to mind the coevolutionary metaphor, as posited in related works [27]. Indeed, a natural follow-up of this study could involve borrowing the developments from coevolutionary algorithms, in particular coevolving tests or using measures like distinctions or informativeness to maintain them.

However, though such extensions are likely to improve success rates, we do not expect them to bring a qualitative breakthrough. CDGP in its current form does not scale well on all problems (cf.

Search4, arity $n = 5$). Though CDGP involves a solver that ‘understands’ the source code of candidate programs, the way it exploits that knowledge is far from sophisticated, to say the least. The search operators, taken verbatim from standard GP, are largely ignorant about the strengths and weaknesses of a program they modify, abstract from the nature of tests used to evaluate a program, and do not distinguish whether a test originated randomly or via counterexample. It seems thus desirable to make search operators better-informed about the characteristics of parent programs. The Lexicase selection we used here is a small step in that direction, and we find applying analogous ideas to search operators the most promising further direction.

Acknowledgments KK and IB acknowledge support from grant 2014/15/B/ST6/05205 funded by the National Science Centre, Poland.

REFERENCES

[1] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. 2011. *An Overview of Formal Methods Tools and Techniques.* Springer London, London, 15–44. DOI:http://dx.doi.org/10.1007/978-0-85729-018-2_2

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit Seshia, Rajdeep Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD), 2013.* IEEE, 1–8. DOI:http://dx.doi.org/10.1109/fmcad.2013.6679385

[3] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2015. Results and Analysis of SyGuS-Comp’15. In *Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015.* 3–26. DOI:http://dx.doi.org/10.4204/EPTCS.202.3

[4] Andrea Arcuri and Xin Yao. 2007. Coevolving Programs and Unit Tests from Their Specification. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE ’07).* ACM, New York, NY, USA, 397–400. DOI:http://dx.doi.org/10.1145/1321631.1321693

[5] Andrea Arcuri and Xin Yao. 2014. Co-evolutionary Automatic Programming for Software Development. *Inf. Sci.* 259 (Feb. 2014), 412–432. DOI:http://dx.doi.org/10.1016/j.ins.2009.12.019

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2015. *The SMT-LIB Standard: Version 2.5.* Technical Report. Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org. (2016).

[8] Dines Bjørner and Cliff B. Jones (Eds.). 1978. *The Vienna Development Method: The Meta-Language.* Springer-Verlag, London, UK, UK.

[9] Paul P. Boca, Jonathan P. Bowen, and Jawed I Siddiqi. 2009. *Formal Methods: State of the Art and New Directions* (1st ed.). Springer Publishing Company, Incorporated.

[10] Nathan Burles, Edward Bowles, Alexander E. I. Brownlee, Zoltan A. Kocsis, Jerry Swan, and Nadarajen Veerapen. 2015. *Search-Based Software Engineering: 7th International Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015, Proceedings.* Springer International Publishing, Cham, Chapter Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava, 255–261. DOI:http://dx.doi.org/10.1007/978-3-319-22183-0_20

[11] Iwo Błądek and Krzysztof Krawiec. 2017. *Evolutionary Program Sketching.* Springer International Publishing, Cham, 3–18. DOI:http://dx.doi.org/10.1007/978-3-319-55696-3_1

[12] A. Cavalcanti, A. Sampaio, and J. Woodcock. 2006. *Refinement Techniques in Software Engineering: First Pernambuco Summer School on Software Engineering, PSSE 2004, Recife, Brazil, November 23-December 5, 2004, Revised Lectures.* Springer Berlin Heidelberg. https://books.google.co.in/books?id=aa1qCQAAQBAJ

[13] B Cohen. 1994. A Brief History of Formal Methods. *Formal Aspects of Computing* 1, 3 (1994).

[14] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. Ramakrishnan and Jakob Rehof (Eds.). Lecture Notes in Computer Science, Vol. 4963. Springer Berlin / Heidelberg, Berlin, Heidelberg, Chapter 24, 337–340. DOI:http://dx.doi.org/10.1007/978-3-540-78800-3_24

[15] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2010. *Component Based Synthesis Applied to Bitvector Programs.* Technical Report MSR-TR-2010-12. http://research.microsoft.com/apps/pubs/default.aspx?id=119146

[16] John V. Guttag and James J. Horning. 1993. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag New York, Inc., New York, NY, USA.

[17] Mark Harman, Yue Jia, and William B. Langdon. 2014. *Babel Pidgin: SBSE Can Grow and Graft Entirely New Functionality into a Real World System.* Springer International Publishing, Cham, 247–252. DOI:http://dx.doi.org/10.1007/978-3-319-09940-8_20

[18] Thomas Helmuth, Lee Spector, and James Matheson. 2015. Solving Uncompromising Problems with Lexicase Selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (Oct. 2015), 630–643. DOI:http://dx.doi.org/doi:10.1109/TEVC.2014.2362729

[19] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. DOI:http://dx.doi.org/10.1145/363235.363259

[20] Martin Hofmann. 2010. Igor II - an analytical inductive functional programming system. In *In Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation.* 29–32.

[21] Martin Hofmann, Emanuel Kitzelmann, and Ute Schmid. 2008. Analysis and Evaluation of Inductive Programming Systems in a Higher-Order Framework. In *Proceedings of the 31st Annual German Conference on Advances in Artificial Intelligence (KI ’08).* Springer-Verlag, Berlin, Heidelberg, 78–86. DOI:http://dx.doi.org/10.1007/978-3-540-85845-4_10

[22] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *29th International Conference on Software Engineering (ICSE ’10).* 215–224. DOI:http://dx.doi.org/10.1145/1806799.1806833

[23] Colin Johnson. 2007. Genetic Programming with Fitness based on Model Checking. In *Proceedings of the 10th European Conference on Genetic Programming (Lecture Notes in Computer Science),* Marc Ebner, Michael O’Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar (Eds.), Vol. 4445. Springer, Valencia, Spain, 114–124. DOI:http://dx.doi.org/doi:10.1007/978-3-540-71605-1_11

[24] Susumu Katayama. 2012. An Analytical Inductive Functional Programming System That Avoids Unintended Programs. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation (PEPM ’12).* ACM, New York, NY, USA, 43–52. DOI:http://dx.doi.org/10.1145/2103746.2103758

[25] Gal Katz and Doron Peled. 2008. *Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms.* Springer Berlin Heidelberg, Berlin, Heidelberg, 33–47. DOI:http://dx.doi.org/10.1007/978-3-540-88387-6_5

[26] Gal Katz and Doron Peled. 2010. MCGP: A Software Synthesis Tool Based on Model Checking and Genetic Programming. In *8th International Symposium on Automated Technology for Verification and Analysis, ATVA 2010 (Lecture Notes in Computer Science),* Ahmed Bouajjani and Wei-Ngan Chin (Eds.), Vol. 6252. Springer, Singapore, 359–364. DOI:http://dx.doi.org/doi:10.1007/978-3-642-15643-4_28

[27] Gal Katz and Doron Peled. 2016. Synthesizing, correcting and improving code, using model checking-based genetic programming. *International Journal on Software Tools for Technology Transfer* (2016), 1–16. DOI:http://dx.doi.org/10.1007/s10009-016-0418-1

[28] Zoltan Kocsis and Jerry Swan. 2014. Asymptotic Genetic Improvement Programming with Type Functors and Catamorphisms. In *Workshop on Semantic Methods in Genetic Programming, Parallel Problem Solving from Nature, Ljubljana, Slovenia.*

[29] Zoltan A. Kocsis, John H. Drake, Douglas Carson, and Jerry Swan. 2016. Automatic Improvement of Apache Spark Queries using Semantics-preserving Program Reduction. In *Genetic Improvement 2016 Workshop,* Justyna Petke, David R. White, and Westley Weimer (Eds.). ACM, Denver, 1141–1146. DOI:http://dx.doi.org/doi:10.1145/2908961.2931692

[30] Zoltan A. Kocsis and Jerry Swan. 2017 (to appear). Genetic Programming + Proof Search = Automatic Improvement. *Journal of Automated Reasoning* (2017 (to appear)).

[31] Krzysztof Krawiec. 2015. *Behavioral Program Synthesis with Genetic Programming.* Studies in Computational Intelligence, Vol. 618. Springer International Publishing. DOI:http://dx.doi.org/10.1007/978-3-319-27565-9

[32] Lech Madeyski. 2010. *Test-Driven Development: An Empirical Evaluation of Agile Practice* (1st ed.). Springer Publishing Company, Incorporated.

[33] Bertrand Meyer. 1986. *Design by Contract.* Technical Report TR-EI-12/CO. Interactive Software Engineering Inc.

[34] Stephen Muggleton. 1994. Inductive Logic Programming: Derivations, Successes and Shortcomings. *SIGART Bull.* 5, 1 (Jan. 1994), 5–11. DOI:http://dx.doi.org/10.1145/181668.181671

[35] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. *SIGPLAN Not.* 41, 11 (Oct. 2006), 404–415. DOI:http://dx.doi.org/10.1145/1168918.1168907

[36] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’10).* ACM, New York, NY, USA, 313–326. DOI:http://dx.doi.org/10.1145/1706299.1706337

[37] L. G. Valiant. 1984. A Theory of the Learnable. *Commun. ACM* 27, 11 (Nov. 1984), 1134–1142. DOI:http://dx.doi.org/10.1145/1968.1972

[38] Jim Woodcock and Jim Davies. 1996. *Using Z: Specification, Refinement, and Proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.