Counterexample-Driven Genetic Programming: Heuristic Program Synthesis from Formal Specifications

Iwo Błądek ibladek@cs.put.poznan.pl
Institute of Computing Science, Poznan University of Technology,
Poznań, 60-965, Poland

Krzysztof Krawiec krawiec@cs.put.poznan.pl
Institute of Computing Science, Poznan University of Technology,
Poznań, 60-965, Poland

Jerry Swan jerry.swan@york.ac.uk
Department of Computer Science, University of York, York, YO10 5GH, UK

doi:10.1162/EVCO_a_00228

Abstract

Conventional genetic programming (GP) can guarantee only that synthesized programs pass tests given by the provided input-output examples. The alternative to such a test-based approach is synthesizing programs by formal specification, typically realized with exact, nonheuristic algorithms. In this article, we build on our earlier study on Counterexample-Based Genetic Programming (CDGP), an evolutionary heuristic that synthesizes programs from formal specifications. The candidate programs in CDGP undergo formal verification with a Satisfiability Modulo Theory (SMT) solver, which results in counterexamples that are subsequently turned into tests and used to calculate fitness. The original CDGP is extended here with a fitness threshold parameter that decides which programs should be verified, a more rigorous mechanism for turning counterexamples into tests, and other conceptual and technical improvements. We apply it to 24 benchmarks representing two domains: the linear integer arithmetic (LIA) and the string manipulation (SLIA) problems, showing that CDGP can reliably synthesize provably correct programs in both domains. We also confront it with two state-of-the art exact program synthesis methods and demonstrate that CDGP effectively trades longer synthesis time for smaller program size.

Keywords

Genetic programming, formal verification, counterexamples, SMT.

1 Introduction

Genetic programming (GP) is an inductive program synthesis technique, in which desired program behavior is defined by a set of input-output test cases. While this kind of specification is usually easy to provide, it is by definition incomplete—in general nothing (or at best, very little) can be guaranteed about program behavior for other inputs. Guaranteeing correctness by enumerating all inputs is impossible, except for toy examples. This is limiting, as many applications require a guarantee of correct program behavior for every possible input (which may be infinite in number), or to ensure that a

Manuscript received: 30 November 2017; revised: 27 April 2018; accepted: 9 May 2018.
© 2018 Massachusetts Institute of Technology

certain property is always met. Examples include hardware design, safety-critical systems, and finding mathematical structures with certain properties.

An alternative to test-based specification is to encode the semantics of a program in some formal logic. Program behavior is then reasoned about within that logic, confronting it with a *formal specification* which defines the desired behavior. When thus conducted, the reasoning certifies program correctness. If the answer is positive, the program is guaranteed to behave as desired for *all* inputs allowed by specification, thereby addressing the problem of incomplete testing. Otherwise, a *counterexample* can be constructed that exemplifies a flaw in the program. *Program verification* is nowadays a mature branch of research in computer science, offering a range of efficient tools which facilitate reasoning about program properties (Section 2).

Verification cannot be used directly for synthesis, as it requires a program to work with—it cannot come up with program candidates on its own. In Krawiec et al. (2017), we proposed counterexample-driven genetic programming (CDGP), an approach in which a GP algorithm serves as such a generator. The method, detailed in Section 3, submits the candidate programs to verification, collects the counterexamples produced whenever a program fails to meet the prescribed specification, and uses them as test cases, thereby eliciting the fitness to guide the GP search process. To the best of our knowledge, CDGP is the first GP-based approach utilizing counterexamples obtained via formal verification.

This study extends Krawiec et al. (2017) in several ways. Firstly, the original CDGP occurred in two variants, namely conservative and nonconservative, which varied in the policy used to decide when a (potentially costly) call to the verifier should be made. Here, we unify those variants and control this aspect with a continuous parameter, which also allows us to explore intermediate strategies of engaging the verifier. Secondly, we provide a more rigorous and systematic approach to evaluation modes, properties of the specification-based problems, and their consequences on the workings of CDGP (Section 3.2). Thirdly, we extend CDGP beyond the integer domain, making it applicable to programs that operate on character strings. Fourthly, in the experimental part (Section 5), we apply CDGP to a broader suite of benchmarks, and provide a more in-depth presentation of results and their analysis. Last but not least, we confront CDGP with formal synthesizers (Section 5.7), which leads to interesting insights about their advantages and disadvantages compared to CDGP and GP in general.

2 Formal Verification

In formal approaches to program verification and synthesis, it is usually assumed that the desired behavior of a program is given in the form of a *contract*, 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.

Verifying a given program p then consists of proving that

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

where p denotes a program and $p(in)$ the output produced by p when applied to input in. For instance, the contract for verifying a program that calculates the maximum of two integers (x, y), called hereafter Max2, can be defined as follows:

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

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.

To determine whether a given program p fulfills (1), p and the specification are provided as inputs to a satisfiability (SAT) *solver*. Technically, the solver attempts to *disprove* (1), that is prove that

$$\exists_{in} Pre(in) \implies Post(in, p(in)). \tag{3}$$

For the Max2 problem (2), inputs in to the program would be two integer variables x and y. If the solver produces an assignment to in that validates (3), p does not behave as specified and is thus incorrect. The variable assignment in question, called a *model* in propositional logic, forms a *counterexample*. Otherwise, p meets the contract and is thus correct. Crucially, the solver performs verification without actually running the program, because the properties of the output can be logically inferred from the properties of the input and those of the program code.

In practice, the solver must be equipped with an additional abstraction layer, a *theory*, in order to be able to reason in terms of, for instance, integer arithmetic (which we already assumed in the above Max2 example). This leads to the concept of satisfiability modulo theories (SMT) used both in program verification and synthesis (Jha et al., 2010; Gulwani et al., 2010; Srivastava et al., 2010). For Max2, the theory of linear integer arithmetic (LIA) (Barrett et al., 2016) may be used.

It may be worth mentioning that SMT solvers can be also directly used for program synthesis. A synthesis task $(Pre, Post)$ can be posed as proving that

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

where the source code of the program p is controlled by a set of free 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. If a solver succeeds to prove (4) and the proof is constructive, its transcript produces as a "side effect" such a p that fulfills (4). However, due to the presence of two quantifiers (existential and universal), solving a synthesis task posed in the above way requires much higher computational effort, as compared to the corresponding verification task that uses just one quantifier because the program is already given there. In practice, therefore, only very short programs can be synthesized in this way. For this reason, methods which synthesize programs by direct invocation of solvers are few and far between; more sophistication is needed to make that process effective. Nevertheless, there are a range of methods that rely on some form of SMT-based synthesis (Srivastava et al., 2010; Gulwani et al., 2010).

2.1 Bridging Test-Based and Formal Specifications

Test-based synthesis and spec-based synthesis may seem irreconcilable due to the different ways in which they specify the desired behavior of programs. In fact, they can be united, and illustrating this can be useful in explaining the rationale behind CDGP.

Example 2.1: For the Max2 problem (2), the following set of input-output pairs might form a test-based specification:

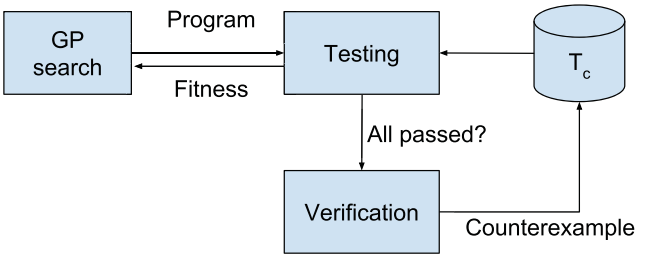

Figure 1: The conceptual diagram of conservative CDGP.

x	y	$max(x, y)$
0	0	0
1	0	1
4	3	4
...	...	...

We can translate this specification into a conjunction of first-order logic constraints and the LIA theory that provides the semantics of the arithmetic operators:

$$[(x = 0 \wedge y = 0) \Longrightarrow max(x, y) = 0] \quad \wedge$$
$$[(x = 1 \wedge y = 0) \Longrightarrow max(x, y) = 1] \quad \wedge$$
$$[(x = 4 \wedge y = 3) \Longrightarrow max(x, y) = 4] \quad \wedge$$
$$\ldots$$

This compound constraint forms a postcondition $Post(x, y)$, which can be fed into a solver together with some candidate program p (i.e. a specific implementation of $max(x, y)$). In response, the solver verifies whether p meets this contract by trying to find a counterexample (3). Success implies that at least one of the constituent implications above is not true; failure implies that they are all fulfilled and the program is correct; however, correct here obviously means "only with respect to the given tests."

3 Counterexample-Driven Genetic Programming

Example 2.1 illustrates that the feedback obtained from verification (whether applied to formal specification or to tests) can be only twofold: success or failure. Program verification is thus of little help, at least in such a simple scenario, for search-based synthesis algorithms (such as GP) that require more graded guidance through the search space. Though in principle the postcondition could be rephrased to calculate also the *number* of passed constraints (cf. Johnson, 2007; see also related work in Section 4), this can be done much more easily (and more efficiently) by just running the program on tests, as practiced in GP. Therefore, rather than trying to elicit richer feedback from verification, in CDGP we rely on conventional GP fitness. Formal verification is used only to decide whether a given candidate program is correct, while the counterexamples become new test cases and provide the algorithm with a search gradient.

Figure 1 presents the conceptual diagram of CDGP. The *GP search* module is a conventional GP algorithm (equivalently, any generate-and-test metaheuristic) equipped

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

```
1:  procedure CDEVAL(P, T_c, (Pre, Post))
2:      T ← ∅                                          ▷ Working set of tests
3:      for all p ∈ P do                               ▷ Evaluation loop
4:          p.eval ← EVAL(p, T_c)
5:          if p.eval ≥ q|T_c| then
6:              c ← VERIFY(p, (Pre, Post))
7:              if c = ∅ then return p                 ▷ Correct program
8:              else
9:                  T ← T ∪ {c}                        ▷ Add counterexample c to T
10:             end if
11:         end if
12:     end for
13:     return (P, T_c ∪ T)
14: end procedure
```

with arbitrary selection and search operators. The generated solutions are evaluated by the *Testing* module by running them on the set T_c of test cases collected during the run. After evaluation, *some* candidate programs are sent to the *Verification* module, which performs verification using an SMT solver and pushes the resulting counterexamples to T_c, gradually increasing the test base. Note that technically counterexamples consist only of program inputs and are not therefore fully-formed tests; we detail and handle this issue in Section 3.2, but for the time being use these terms interchangeably.

3.1 Evaluation and Verification

The complete evaluation stage is realized by function CDEval shown in Algorithm 1. CDEval consists of both testing and verification and is launched once per generation. In contrast to conventional GP evaluation that utilizes only a set of tests T_c (which is initially empty here), CDEval accepts also a formal specification (Pre, $Post$). Upon completion, CDEval returns the evaluated population P and the updated set of tests $T_c \cup T$. Eval(p, T_c) is the conventional fitness function that returns the number of tests from T_c that are passed by p. Verify(p, (Pre, $Post$)) executes the solver and returns the counterexample resulting from verifying p on the specification (Pre, $Post$). We illustrate Verify with the following example.

Example 3.1: Assume the task is to synthesize a program that meets the Max2 specification (Eq. 2), and that the GP algorithm produced the following program p to evaluate:

```
(ite (>= y x) x y),
```

where `ite` is an if-then-else instruction. When p is sent to Verify, CDGP creates the query to the solver shown in Listing 1, formulated in the SMT-LIB language (Barrett et al., 2015, 2016), which we briefly cover in Section 3.3. Notice that p is encapsulated in the body of the `max` function (`define-fun max`). In the lines that follow, free variables x and y are declared (`declare-fun`), and the `assert` statement defines the specification by implementing the formulation of verification from Eq. (3). Once this sequence of commands is passed to the solver and followed by the `check-sat` statement, the solver

```
(set-logic LIA)
(define-fun max ((x Int)(y Int)) Int (ite (>= y x) x y))
(declare-fun x () Int)
(declare-fun y () Int)
(assert (not (and
   (>= (max x y) x)
   (>= (max x y) y)
   (or (= (max x y) x) (= (max x y) y)))))
(check-sat)
(get-value (x y))
```

Listing 1: Example of the verification query for the Max2 problem, formulated in SMT-LIB language ver. 2.6.

will try to find such values of x and y that falsify the specification (notice the `not` in the assertion).

After execution of `check-sat`, the `get-value` statement is used to retrieve the values of x and y found by the solver, and those values form a counterexample. The returned counterexample depends on solver tactics; the Z3 solver (de Moura and Bjørner, 2008) that we use in the experimental section responds here with `(x=-1, y=0)`. The reader can verify that the result of p for this input is indeed incorrect with respect to the specification. When a correct program is verified (i.e., `(ite (>= x y) x y)` or any semantically equivalent program), then the solver returns `unsat`, the search process stops, and the program is returned. □

In the first generation of CDEval (Algorithm 1) $T_c = \emptyset$, so all programs in P receive zero fitness and the attendant selection of parent programs is 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 fine-grained fitness function and more precise search gradient.

Which of the evaluated programs should be subject to verification is an important design choice, to which we pay special attention in this study. In the *conservative* variant which we introduced in Krawiec et al. (2017), we verified only the programs that passed all test cases previously collected in T_c. This variant is not only computationally efficient (verification can be costly), but also arguably most natural, as a program that does not pass all available tests cannot by definition be correct. However, by requiring all tests to be passed, the conservative scheme can lead to stagnation: it can happen that the SMT solver produces a test which is particularly difficult, and the GP algorithm may struggle to generate a program capable of passing it (and all other tests in T_c simultaneously). As a result, many generations may elapse before GP produces a program good enough for the next verification and T_c becomes augmented by the resulting counterexample. Such a course of events can be particularly harmful in the initial stages of a CDGP run, when T_c is small and thus provides little search gradient.

To mitigate this problem, in this study we extend CDGP with the *fitness threshold* q of the ratio of passed tests from T_c required to apply verification (line 5 of Algorithm 1). For the conservative variant, $q = 1.0$. The other extreme is setting $q = 0.0$ (dubbed *non-conservative* in Krawiec et al. (2017)), which causes *all* evaluated programs to undergo verification, regardless of fitness value, and is thus rather costly in computation. We anticipate that setting q to intermediate values can be beneficial, avoiding the above-mentioned stagnation on one hand, and excessive cost of verification on the other.

```
(set-logic LIA)
(define-fun max ((x Int)(y Int)) Int (ite (>= x y) y x))
(define-fun x () Int (- 1))
(define-fun y () Int 0)
(assert (and
   (>= (max x y) x)
   (>= (max x y) y)
   (or (= (max x y) x) (= (max x y) y))))
(check-sat)
```

Listing 2: Example of the query used for evaluation of incomplete tests.

3.2 Turning Counterexamples into Tests

As signaled above and illustrated in Example 3.1, counterexamples are instances of program input (in), and as such do not form tests that require also the associated correct program output (out). Therefore, we allow T_c to hold two types of tests:

Complete tests A *complete test* is a test of the form (in, out). It is equivalent to the notion of test used in conventional GP and can be evaluated by executing the program on in and comparing the obtained result to the expected output out. CDGP uses this mode of evaluation for all complete tests.

Incomplete tests An *incomplete test* is a test of the form $(in, null)$. The expected output for this test is not defined explicitly. This can happen for example if there are many (potentially even infinitely many) correct outputs for in.

In conventional test-based GP, incomplete tests are useless, as they do not say anything about the desired program behavior. In the spec-based CDGP, programs can be still evaluated on them by sending an appropriate query to the SMT solver, which we demonstrate in the following example.

Example 3.2: We use the program $p =$`(ite (>= y x) x y)` from Example 3.1 and the counterexample `(x=-1, y=0)` obtained there, which we express as an incomplete test (`(x=-1, y=0)`, $null$). The query that would evaluate p on this test, presented in Listing 2, differs in two details from the verification query (Listing 1): the postcondition is not negated, and the free variables are bound to constants from the incomplete test.

For this query, the solver returns `unsat`, because p does not meet the specification for the considered input. For some other inputs however (e.g., `(x=0, y=0)`), this incorrect program may pass the specification, causing the solver to return `sat`. □

Though the possibility of testing programs on incomplete tests is appealing, it has a critical downside: calls to the solver are computationally costly, and the above approach becomes prohibitively expensive in the presence of many incomplete tests. Therefore, CDGP transforms incomplete tests into complete ones whenever possible. This, however, requires meeting certain formal properties that we detail in the following.

3.2.1 Single-Output Property

It is typically assumed in GP that for every program input in there is only one correct output out, and that programs returning a value other than out are incorrect. This is, however, not necessarily the case for arbitrary formal specifications. For example, for

```
(set-logic LIA)
(declare-fun out1 () Int)
(declare-fun out2 () Int)
(define-fun max2 ((x Int)(y Int)) Int out1)
(define-fun max2__2 ((x Int)(y Int)) Int out2)
(declare-fun x () Int)
(declare-fun y () Int)

(assert (>= (max2 x y) x))
(assert (>= (max2 x y) y))
(assert (or (= x (max2 x y)) (= y (max2 x y))))

(assert (>= (max2__2 x y) x))
(assert (>= (max2__2 x y) y))
(assert (or (= x (max2__2 x y)) (= y (max2__2 x y))))

(assert (distinct out1 out2))
(check-sat)
```

Listing 3: SMT query that checks whether the Max2 problem has the single-output property.

the following specification of a desired program behavior f:

$$f(in_1) > 0,$$

where $in_i \in \mathbb{Z}$, there are infinitely many acceptable outputs for any given input in_1. Similarly for the specification:

$$(in_2 = 0 \implies f(in_2) = 0) \quad \wedge$$
$$(in_2 > 0 \implies f(in_2) = in_2 + 1).$$

The second case is particularly interesting, since for some of the inputs ($in_2 < 0$) the output of f is undefined, which implies that *any* returned value is correct.

As we aim at collecting complete tests in T_c and so avoiding costly solver costs to determine output's correctness (Example 3.2), we must require that a given input has the *single-output property*, that is it has only one correct corresponding output. If we ignored that aspect and associated an arbitrarily selected correct output out with a given in, the resulting test (in, out) could unfairly fail many programs that return a different correct output for in.

To address this issue, prior to applying CDGP to any given problem, we use the SMT solver to determine whether the single-output property holds for it. To illustrate, Listing 3 presents the SMT query that verifies this property for the Max2 problem. When queried, the solver will search for such values of x, y for which it is possible to find two different outputs $out1$, $out2$ that meet the specification. If the solver returns `sat`, such outputs were found and the problem does not have the single-output property. If the solver returns `unsat`, then the problem has single-output property. Occasionally, the solver may return `unknown`, signaling that either the property cannot be verified in a given logic or that the available computational resources (time, memory) have been exhausted. In such cases, we assume that the single-output property does not hold.

More precisely, the single-output property can be defined in two ways: globally, as a property of a *problem* (i.e. for all inputs), or locally, as a property of an *input* (given problem). Verity of the former implies the latter. However, the absence of the global

single-output property does not imply that single output cannot be determined *for specific inputs*. Therefore, in case the single-output property does not hold globally, CDGP attempts to find out if it holds locally for every new counterexample, which is explained in Section 3.2.3.

3.2.2 Single-Invocation Property

Ensuring that the single-output property holds is not the only issue we must take into account. In the following specification:

$$f(x, y+1) = f(x, y) + 1 \quad \wedge$$
$$f(x+1, y) = f(x, y) + 1,$$

function f is invoked multiple times with different arguments. As a result, whether a program returns the correct output for some input depends on the values it returns for some other inputs. To create a complete test, we need a single input to execute the program on it and then compare the obtained result with the expected output. But for which arguments should the output of f be taken, when there are several of them? In general, we must account for outputs of f for every unique combination of its arguments in the specification, but they again can recursively depend on yet some other invocations of f. Therefore, a single input-output pair is not sufficient to test a program on it, and this necessitates the creation of an incomplete test, like for example $(\texttt{(x=0, y=0)}, null)$, which will be evaluated by the solver.

If the function to be synthesized is called always with the same arguments in formal specification, the problem has the *single-invocation property* (Reynolds et al., 2017). Checking this property is relatively simple and can be done by syntactic analysis of the specification.

3.2.3 Finding Outputs for Incomplete Tests

It should be clear from the two previous subsections that the desired output for an incomplete test can be unambiguously determined using a solver if both of the above-mentioned properties hold. We summarize this observation in the following theorem:

THEOREM 1: *To create a complete test* (in, out) *from an incomplete test* $(in, null)$, in *must have the single-output property, and the problem (specification) must have the single-invocation property.*

The practical upshot of the above observation is that a problem without single-invocation property would force calling the solver whenever testing a program on any (incomplete) test. Though CDGP can handle such problems, we do not consider them in the experimental part, because calling the solver for each program generated by GP *and* each test in T_c is prohibitively expensive.[1]

Determining the local single-output property for an input of the incomplete test is realized by the same query which is responsible for searching for the correct output. This query, presented in Listing 4, simply represents the output to be produced by a program as a single variable, and solves for its value for the provided inputs. The resulting value of the output is then combined with the input to form a complete test. The key observation is that we can search for the correct output twice, the second time excluding the answer we obtained the first time. If the answer for the second query is `unsat`,

[1]In contrast, note that the solver is called for verification (Algorithm 1) *at most once per evaluated program* (the number of tests in T_c has no impact on the number of solver's calls used for verification).

```
(set-logic LIA)
(declare-fun out () Int)
(define-fun max ((x Int)(y Int)) Int out)
(define-fun x () Int (- 1))
(define-fun y () Int 0)
(assert (and
   (>= (max x y) x)
   (>= (max x y) y)
   (or (= (max x y) x) (= (max x y) y))))
(check-sat)
(get-value (out))
```

Listing 4: SMT query that finds an output for an incomplete test.

then the input has the single-output property and a complete test is created. The second query has an additional constraint `(assert (distinct out value))`, where `value` is the output obtained in the first query.

It may also happen that the first query returns `unsat`. This means that the specification is contradictory and no program can satisfy it, and thus the search process would end with the appropriate message.

3.3 Representation of Solutions

As programs in CDGP need to be verified by the SMT solver, they can use only the semantics (types, operators, instructions, etc.) available in the background theory supported by the solver, for example LIA or Strings (SLIA). In principle, any programming language could be used to represent programs in CDGP, given the appropriate converter to SMT-LIB language (Barrett et al., 2015, 2016), which is accepted as an input language by most modern SMT solvers. In this study, for simplicity, we decided to evolve programs directly as SMT-LIB expressions. SMT-LIB is a functional language and is similar in many aspects to LISP, so the traditional tree-based GP was most adequate.

4 Related Work

Formal methods for program synthesis have a long history, preceding heuristically informed stochastic methods such as GP by some decades (Cohen, 1994). The literature for formal approaches to synthesis (and verification) is correspondingly vast; for recent overviews see Boca et al. (2009) and Almeida et al. (2011). In contrast, we are aware of only few program synthesis approaches which combine formal techniques with heuristic search. To our knowledge, the earliest work combining the aspects of evolutionary search and formal approaches was that by Haynes et al. (1996), where GP was used to produce entailment proofs, an initial step for potentially using it to automate the specification refinement process.

An approach due to Johnson (2007) incorporated model-checking with the specification of the task expressed via computation tree logic (CTL) to evolve finite state machines, and was used to learn a controller for a simple vending machine. The fitness was computed as the number of CTL properties which were verified to hold for a given program. A similar approach by He et al. (2011), the Hoare logic-based GP, computes fitness as the number of postcondition clauses which can be inferred from the precondition and the program being evaluated. Instead of model-checking, the Hoare logic (Hoare, 1969) is used for the specification of the task and verification.

From 2008, Katz and Peled (2008, 2014, 2016) authored a series of papers combining model-checking and GP in which they progressively refine their MCGP tool based on linear temporal logic (LTL). They use enhanced model-checking to impose a gradient on the fitness function by distinguishing several levels of passing an LTL property (met for all inputs, met for only some inputs, met for no input). Apart from that, this approach is very similar to the two previously described. It is worth noting that Katz and Peled (2014) also considered briefly using SMT solver for verification in the similar way as model-checking, and even similarly to CDGP utilized counterexamples to provide for more granular fitness. However, they only reported trying to solve a simple problem, and seemingly abandoned this line of research after that. By utilizing counterexamples, this initial work is the most similar to CDGP out of all related works mentioned here.

The use of coevolutionary GP to synthesize programs from formal specifications was researched by Arcuri and Yao (2007, 2014). They maintained joint populations of tests and programs within a competitive coevolution framework. Fitness of programs was calculated using a heuristic that estimated how close a postcondition was from being satisfied by the program's output for specific tests. While allowing the synthesis of programs with GP from a formal specification, such an approach provides no guarantees that program deemed correct by their method will be consistent with the specification for all possible inputs.

In the emerging area of genetic improvement (the modification of pre-existing program code via search), there have been a number of recent articles incorporating formal approaches. Kocsis et al. (2016) report a 10,000-fold speedup of Apache Spark database queries on terabyte datasets. In Burles et al. (2015), a 24% improvement in energy consumption was achieved for Google's Guava collection library by applying the "Liskov substitution principle," the formal cornerstone of object-orientation. Some recent work has also used category theory to perform formal transformations on datatypes (Kocsis and Swan, 2014, 2017) in order to join together parts of a program which are otherwise unrelated, a technique applicable to "Grow and Graft Genetic Programming" (Harman et al., 2014).

There are also many well-known systems that perform synthesis under the broad heading of Inductive Logic Programming (Muggleton, 1994). In particular, IGOR II (Hofmann, 2010; Hofmann et al., 2008) is known to perform well on a range of problems. As extended by Katayama (2012), it combines an "analytic" approach based on analysis of fitness cases with the generate-and-test approach more familiar to the GP community.

An alternative to spec-based synthesis is "program sketching" (Solar-Lezama et al., 2006), a technique whereby a program contains "holes" which are automatically filled in (e.g., using an SMT solver) with values satisfying a specification. However, the approach has limited scalability since the exact search method used has exponential runtime in the number of variables. More recently, evolutionary program sketching (EPS) has been proposed (Błądek and Krawiec, 2017). EPS is presented as a GP alternative that evolves partial programs and 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 Experiments

In the following sections, we describe the experimental framework, including information about benchmarks, some implementation choices, tested configurations of CDGP,

Table 1: LIA benchmarks. The input type is $\texttt{I}^n$ and the output type is `I` (`I`=integer). Some functions were tested in variants with different arities.

Name	Arity	Semantics
CountPos	2, 3, 4	The number of positive arguments
IsSeries	3, 4	Do arguments form an arithmetic series?
IsSorted	4, 5	Are arguments in ascending order?
Max	4	The maximum of arguments
Median	3	The median of arguments
Range	3	The range of arguments
Search	2, 3, 4	The index of an argument among the other arguments
Sum	2, 3, 4	The sum of arguments

and baselines. Then, in Section 5.6, we analyze the results of the experiments and the characteristics of CDGP dependent on its settings. In Section 5.7, we confront CDGP with exact, nonheuristic algorithms of program synthesis. The source code of CDGP, along with specifications of problems, is available at: https://github.com/kkrawiec/CDGP.

5.1 Benchmark Suite

We consider a range of spec-based synthesis benchmarks of varying difficulty and characteristics, representing two domains: the theory of linear integer arithmetic (LIA) and the theory of strings (SLIA, strings and linear integer arithmetic) (Barrett et al., 2016). Part of the benchmarks come from the SyGuS repository maintained for the annual "Syntax-Guided Synthesis" competition (Alur et al., 2015, 2013). We detail the domains, grammars, and benchmarks in the following, first for LIA and then for SLIA.

5.1.1 LIA Benchmarks

In LIA benchmarks, presented in Table 1, the task is to synthesize a program with a signature $\texttt{I}^n \rightarrow \texttt{I}$, where `I` stands for integer type and n is program's arity. Max, Search, and Sum come from the SyGuS competition (Alur et al., 2015, 2013); 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 form, respectively, an arithmetic series or are sorted in ascending order, 0 otherwise. In the Search benchmark, a correct program should find the 0-based index of the last argument in an "array" of length n formed by the remaining arguments (which are constrained by a precondition to be 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]

The grammar for LIA programs includes two types, `Int` (`I`) and `Boolean` (`B`) (see Figure 2). To avoid multiplying the input variables by themselves (and so building programs that involve nonlinearity), we introduce an additional type `C` for integer

[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 (Alur et al., 2015, 2013).

```
I ::= C | I + I | I - I | I * C | I / C | I % C | ite(B,I,I)
      | v1 | v2 | ... | vn
C ::= -10 | -9 | ... | -1 | 0 | 1 | ... 9 | 10
B ::= true | false | and(B,B) | or(B,B) | not(B)
      | I < I | I <= I | I = I | I >= I | I > I
```

Figure 2: The grammar defining the domain of LIA programs. v_i is the ith input variable, `ite` is the conditional statement, % is the modulo operator. The starting symbol is `I`.

```
(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)
```

Listing 5: The Max4 benchmark expressed in the SyGuS language (fg_max4.sl file in the SyGuS repository). Compare with the specification of the Max2 problem in Eq. (2).

constants. The corresponding production also explicitly defines the range of integers that can be used in the programs generated by CDGP.[3]

Search k is the only group of LIA benchmarks in our suite without the global single-output property (Section 3.2.1) because the desired output is not defined for arrays which are not sorted, and thus any output is correct for such inputs. The single-invocation property holds for every benchmark in our suite.

To illustrate, Listing 5 presents the specification of the Max4 benchmark expressed in the SyGuS language (Raghothaman and Udupa, 2014).[4] 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. The `declare-var` commands declare universally quantified variables, which can be then used in the constraints. Preconditions can be defined by implications with conditions on the values of the variables. In this example there are no implications, so this specification consists of only the postcondition—the precondition is empty, that is, the inputs to Max4 are only required to belong to the type `Int`.

[3]The original LIA grammar does not define such a range, and formal synthesis methods can thus generate programs with arbitrary integers. This becomes relevant when comparing CDGP against formal methods in Section 5.7.

[4]http://www.sygus.org/

Table 2: SLIA benchmarks. Input type is `S` or S^2 and the output type is `S` (`S`=string).

Name	Arity	Semantics
dr-name	1	Extract first name from full name and prepend it with "Dr."
firstname	1	Extract first name from full name
initials	1	Extract initials name from full name
lastname	1	Extract last name from full name
combine	2	Combine first and last name into full name
combine-2	2	Combine first and last name into first name followed by initial
combine-3	2	Combine first and last name into initial followed by last name
combine-4	2	Combine first and last name into last name followed by initial
phone	1	Extract the first triplet of digits from a phone number
phone-1	1	Extract the second triplet of digits from a phone number
phone-2	1	Extract the third triplet of digits from a phone number
phone-3	1	Put first three digits of a phone number in parentheses
phone-4	1	Change all "-" in a phone number to "."

```
S ::= " " | S ++ S | replace(S,S,S) | charAt(S,I) | fromInt(I)
      | substring(S,I,I) | inputs | constants
I ::= constants | I + I | I - I | len(S) | indexOf(S, S, I)
      | fromString(S)
```

Figure 3: The grammar defining the domain of String programs. *inputs* are the input variables, *constants* is a benchmark-specific set of constants of the same type as the production, ++ is string concatenation. The starting symbol is `S`.

5.1.2 SLIA Benchmarks

The SLIA benchmarks, presented in Table 2, are based on those from the "Programming by Examples" track in SyGuS competition. The original benchmarks are all test-based, and our benchmarks are extended to the simplest formal specification that generalises the original set of tests. For example, the original benchmark "dr-name" included input-output pairs: `("Nancy FreeHafer", "Dr. Nancy"), ("Andrew Cencici"), "Dr. Andrew"), ("Jan Kotas", "Dr. Jan")`. The corresponding formal specification states that: a) the first token of the output is `"Dr."` and b) the second token of the output is equal to the first whitespace-delineated token of the input. The other SLIA benchmarks are similarly defined.

The grammar for SLIA programs, shown in Figure 3, includes two types: `String` (`S`) and `Int` (`I`).[5] To realize the functionality requested by the benchmarks, relatively simple capabilities are required: splitting a string into words; extracting the first letter form a word; concatenating strings, and combining the input string(s) with some constant characters. However, different benchmarks require different character constants; for instance, the *dr-name* benchmark requires the "." character. Therefore, the SLIA

[5]In the original SLIA grammar of the SyGuS competition there was also a production for the `Boolean` (`B`) type, but it was not used in other productions, and consequently it was never utilized in our experiments.

grammar is adapted to individual benchmarks by including the required characters via the *constants* term.

SLIA benchmarks are mostly guarded by some preconditions and thus the global single-output property is not met, the only exception being the *combine* benchmark series. The single-invocation property holds for every SLIA benchmark.

5.2 Search Operators

We guarantee that programs initialized and bred within a run belong to the given domain by using a typed variant of GP and conforming with a theory-specific grammar.

Initialization recursively traverses the derivation tree from the starting symbol of the grammar 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 the productions that immediately lead to terminals whenever possible. If the depth exceeds 5, the tree is discarded and the process starts anew.

Mutation picks a random node in the 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 a 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 rooted there with the subtree drawn from the first parent. This process is guaranteed to terminate, since both parent trees always feature at least one node of the type associated with the root node (`I` for LIA and `S` for SLIA) and the root nodes are also allowed to be swapped.

To control bloat, 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. Additionally, whenever a tournament selection or deselection is used, it includes lexicographic parsimony pressure (Luke and Panait, 2002), that is, in case of a tie on fitness, the smaller program is preferred.

5.3 Communication with the Solver

Communication with the solver is realized via the SMT-LIB standard (Barrett et al., 2015), recognized by most contemporary SMT solvers. We employ the well-known Microsoft Z3 SMT solver (de Moura and Bjørner, 2008), one of the most performant and widely-used noncommercial solvers. This choice was arbitrary; that is, no Z3-specific features were used.

Our implementation of VERIFY$(p, (Pre, Post))$ in Algorithm 1 translates the tree representation of an evolved program p into a function definition in the SMT-LIB language, combines it with the contract $(Pre, Post)$, and calls the solver to verify whether p meets $(Pre, Post)$ (see Section 3.1 for more details). The runtime of the solver may vary with the size of the verified program and its structure. In the experiments conducted here, the average time the solver needed for verifying a single program ranged from 0.03s to 0.35s, depending on CDGP variant and fitness threshold. However, occasionally

Table 3: Parameters of the evolutionary algorithm.

Parameter	Value
Number of runs	25
Population size	500
Maximum height of initial programs	5
Maximum height of trees inserted by mutation	5
Maximum height of programs in population	12
Maximum number of generations	100000
Maximum runtime in seconds	3600
Probability of mutation	0.5
Probability of crossover	0.5
Tournament size	7

it took much longer, up to 30s. For this reason, we cap the time of a single run to 1 hour, which becomes our additional stopping criterion, atop of the maximum number of generations.

5.4 Configurations

Table 3 presents the settings of our evolutionary algorithm. Our default configuration involves ***tournament selection***, a common choice for GP that proved useful in many past studies. However, recall that the working set of tests T_c is initially empty and may grow slowly. With a handful of tests, the fitness function (Eval in Algorithm 1), which counts the fraction of passed tests, can assume only a few distinct values and has thus little discriminatory power. Ties on fitness are likely, which causes tournament selection to act at random and weakens the selective pressure.

Therefore, we consider an alternative setup equipped with *lexicase selection* (Helmuth et al., 2015). In each selection event, lexicase starts with a pool of all programs from the population. A random test t is drawn from T_c without replacement, and programs that do not pass t are discarded from the pool. Drawing tests and discarding programs from the pool is repeated until only one program is left, in which case it is selected; if all tests have been used, or the current test would reduce the pool to the empty set, the winner is drawn uniformly from the remaining pool. We do not use lexicographic parsimony pressure in configurations that involve lexicase selection.

In Section 3, we presented CDGP as a *generational evolutionary algorithm*. Note that for the initial population, the solver is called for *each* program being evaluated, as each such program formally passes all tests in T_c, which is initially empty (no matter what the fitness threshold q is). Given that the population is quite sizeable (500, Table 3), this may lead to high computational cost. More importantly however, each such evaluation will produce a counterexample. Many of them, though unique, can be redundant in T_c, i.e. verify the same property of programs. For instance for the Max4 benchmark, the input-output pairs ((2,1,1,1),2) and ((2,0,1,1),2), even though technically distinct, essentially test the same characteristics of programs, that is, their capability of returning the first argument if it happens to be greater than all the remaining ones.

Arguably, neither of the above is desirable, so we consider the *steady-state evolutionary algorithm* as an alternative to the generational one. In that variant, an iteration (generation) consists in first discarding a poorly-performing program from the population

(using a 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 replaces the removed one in the current population and is subject to evaluation. As a result, it may undergo verification if required by Algorithm 1, and the resulting counterexample t_c is immediately added to T_c. If t_c is new to T_c, the fitness values of all programs in the population are updated by applying them to t_c, so that they are consistent with the contents of T_c. To make the comparison between steady state and generational configurations fair, for steady state we multiply the maximum number of generations by the population size (500), so that the maximum number of evaluated solutions in both cases is the same.

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 in T_c. We anticipate this to make search process more reactive and potentially more efficient.

Additionally, we assess the impact of *fitness threshold* q on evolution by testing CDGP on the range of its values: $\{0.0, 0.25, 0.5, 0.75, 1.0\}$. As was discussed in Section 3.1, high values of q result in lower number of performed verifications, while low values provide for better gradient of the fitness function. We find the trade-off between those two effects worth a closer investigation.

We here summarize the differences in experimental configuration with Krawiec et al. (2017). Differences include the presence of lexicographic parsimony pressure (only for tournament selection), initial population not being verified in the steady-state variant, and smaller interval of integers for drawing random tests in the control approach GPR (described later). In this new series of experiments, all algorithms maintain 500 candidate solutions. Moreover, we also use a newer release of the Z3 solver, which may impact the computing time and characteristic of returned counterexamples.

5.5 Baseline: GPR

Our baseline is GPR (GP Random), which 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; that is, the test base gets extended when a program in population manages to pass all of them. As in CDGP, multiple new tests may be added to T_c in a single generation, duplicates in T_c are eliminated, and T_c may grow indefinitely during a run. We use GPR only with $q = \{0.75, 1.0\}$, because (i) CDGP fares best for these values and (ii) setting q to lower values leads to exorbitant numbers of tests in T_c. By comparing CDGP with GPR, we intend to determine whether synthesizing tests from a specification makes CDGP any better than generating them at random.

In GPR, we create random tests by drawing program inputs at random. For LIA benchmarks, we draw numbers uniformly from $[-100, 100]^n$ where n is the input arity of synthesized function. We anticipate that the width of this interval is not critical, given that in most benchmarks (except for Sum and IsSeries) the functions to be synthesized should interpret their inputs as ordinal variables.

5.6 Performance Analysis

We discuss the results for LIA and SLIA benchmarks separately due to their volume and varying characteristics. In Tables 4, 5, and 6 we present respectively the success rate, the end-of-run size of T_c, and the runtime in seconds, for individual variants of CDGP and GPR on the *LIA benchmarks*. Recall that for spec-based synthesis, a success means

Table 4: Success rate for LIA benchmarks. Cell shading indicates higher values.

	CDGP																				GPR							
	Gener.										SteadySt.										Gener.				SteadySt.			
	Tour					*Lex*					*Tour*					*Lex*					*Tour*		*Lex*		*Tour*		*Lex*	
	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.75	1.0	0.75	1.0	0.75	1.0	0.75	1.0
CountPos2	0.92	0.92	0.96	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	1.00	**1.00**	0.72	0.84	**1.00**	**1.00**	0.72	**1.00**	**1.00**	**1.00**	**1.00**	0.96	0.80	0.80	0.84	0.76	0.92	0.60	0.68	0.64
CountPos3	0.32	0.20	0.52	0.48	0.52	0.28	0.52	0.92	**1.00**	0.80	0.20	0.24	0.48	0.64	0.08	0.52	0.44	0.80	0.84	0.52	0.08	0.08	0.36	0.32	0.08	0.08	0.24	0.16
CountPos4	0.00	0.00	0.00	0.12	0.08	0.12	0.16	0.20	0.24	0.16	0.00	0.00	0.04	0.04	0.00	0.04	0.04	0.12	0.16	0.04	0.00	0.00	0.12	0.36	0.00	0.00	0.00	0.00
IsSeries3	0.04	0.16	0.40	0.96	0.92	0.36	0.28	0.28	0.48	0.88	0.24	0.12	0.32	0.92	0.92	0.48	0.36	0.32	0.76	0.40	0.00	0.12	0.00	0.04	0.00	0.08	0.00	0.04
IsSeries4	0.04	0.04	0.08	0.44	0.28	0.08	0.04	0.12	0.24	0.28	0.04	0.00	0.08	0.56	0.36	0.04	0.08	0.00	0.24	0.28	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
IsSorted4	0.48	**1.00**	0.88	**1.00**	**1.00**	0.96	0.96	**1.00**	**1.00**	**1.00**	0.60	0.92	0.96	0.96	0.68	**1.00**	0.88	0.92	**1.00**	0.76	0.00	0.00	0.12	0.44	0.00	0.08	0.08	0.24
IsSorted5	0.32	0.60	0.80	**1.00**	0.92	0.76	0.96	0.96	**1.00**	0.76	0.36	0.68	0.84	0.96	0.76	0.92	0.88	0.88	0.96	0.76	0.00	0.00	0.00	0.04	0.00	0.04	0.00	0.04
Max4	**1.00**	0.96	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.96	**1.00**	0.80	**1.00**	0.96	**1.00**	**1.00**	**1.00**	0.84	**1.00**	0.96	0.96	0.96	**1.00**	0.88	0.80
Median3	0.92	0.84	0.96	**1.00**	0.96	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.60	0.88	0.88	**1.00**	0.88	**1.00**	0.96	0.92	**1.00**	**1.00**	0.96	0.92	0.80	0.72	0.76	0.84	0.76	0.76
Range3	0.72	0.84	0.80	**1.00**	0.96	0.88	0.96	**1.00**	**1.00**	0.96	0.64	0.64	0.88	0.96	0.40	0.92	0.72	0.88	**1.00**	0.88	0.72	**1.00**	0.88	0.96	0.76	0.64	0.80	0.84
Search2	0.96	0.96	**1.00**	**1.00**	**1.00**	0.96	**1.00**	**1.00**	**1.00**	0.96	0.88	0.72	0.88	**1.00**	0.80	0.96	0.96	**1.00**	**1.00**	0.80	0.92	**1.00**	0.96	**1.00**	0.84	0.84	0.96	0.80
Search3	0.84	0.88	0.92	**1.00**	0.96	0.88	0.92	0.92	**1.00**	0.96	0.48	0.44	0.88	0.96	0.72	0.88	0.92	0.96	**1.00**	0.96	0.92	0.80	0.88	0.88	0.52	0.28	0.84	0.72
Search4	0.56	0.76	0.84	0.92	0.80	0.84	0.72	0.96	0.96	0.88	0.40	0.56	0.68	0.76	0.20	0.64	0.56	0.88	**1.00**	0.76	0.80	0.84	0.92	0.76	0.28	0.20	0.76	0.68
Sum2	**1.00**	**1.00**	**1.00**	**1.00**	0.96	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.96	**1.00**	**1.00**	**1.00**	0.96	0.96	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.80	0.96	**1.00**	**1.00**	0.88	0.96
Sum3	0.04	0.24	0.88	**1.00**	0.84	0.92	0.92	0.92	**1.00**	0.80	0.04	0.24	0.76	**1.00**	0.60	0.88	0.84	0.84	**1.00**	0.84	0.24	0.72	0.48	0.84	0.16	0.52	0.20	0.64
Sum4	0.04	0.00	0.04	0.60	0.24	0.12	0.20	0.04	0.80	0.12	0.08	0.00	0.24	0.68	0.00	0.00	0.04	0.32	0.48	0.44	0.00	0.08	0.04	0.32	0.04	0.08	0.08	0.08
All	0.51	0.59	0.69	0.84	0.78	0.70	0.73	0.77	0.86	0.79	0.45	0.52	0.68	0.84	0.56	0.70	0.67	0.74	0.84	0.71	0.46	0.52	0.51	0.58	0.40	0.39	0.45	0.46

Table 5: End-of-run size of T_c for LIA benchmarks. Cell shading indicates higher values.

	CDGP																				GPR							
	Gener.										SteadySt.										Gener.				SteadySt.			
	Tour					*Lex*					*Tour*					*Lex*					*Tour*		*Lex*		*Tour*		*Lex*	
	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.75	1.0	0.75	1.0	0.75	1.0	0.75	1.0
CountPos2	1332	1306	929	368	96	1344	1144	928	274	104	1401	1182	553	138	26	1180	979	665	232	32	6675	815	3318	1205	5813	992	1772	606
CountPos3	4946	4568	3633	1663	100	7261	7315	3583	1283	134	4572	3987	3478	1237	34	4650	4994	3604	1364	74	10447	570	6747	1526	8387	100	5615	597
CountPos4	8014	7183	5083	2397	94	9534	9041	5815	2957	163	8086	6539	5762	1680	33	7045	6469	5538	2377	78	4278	521	5478	954	5001	19	5297	86
IsSeries3	5405	3113	1762	938	114	7652	7472	7393	3839	175	5556	3413	2122	719	29	4717	5808	5556	1903	85	47450	691	6850	958	40245	440	3434	575
IsSeries4	5459	3661	2500	4533	131	9078	9705	8955	5042	249	6549	4223	2582	2921	56	6907	7166	7132	4595	109	48676	40228	3036	3055	42217	35683	2020	2021
IsSorted4	4822	3245	3241	1696	115	3129	3771	3537	1406	146	4706	3414	1836	1169	47	2383	3295	2671	1024	81	48340	576	12219	1531	39895	271	7192	619
IsSorted5	5422	3857	3916	2346	121	5974	5546	4268	2014	207	5118	4366	2854	1443	51	3789	3562	3073	1611	118	41404	502	10564	1275	33883	377	5549	952
Max4	3118	2669	2057	1011	157	1430	1131	1146	747	163	1903	1423	1404	544	31	1784	1487	1475	592	59	5774	608	2109	728	3277	72	2683	604
Median3	3926	3528	2782	913	123	2579	2516	1999	645	156	4986	3317	3341	442	30	2871	2655	2001	409	61	3182	1145	2820	1321	7087	233	2317	615
Range3	4704	4149	2827	700	99	3088	2914	1307	834	120	4527	4135	2684	316	24	3150	3087	2870	695	37	6700	744	2644	655	5370	20	2484	522
Search2	1660	1477	853	307	82	2612	1451	1357	441	94	2791	3529	957	212	23	2174	1847	764	196	46	2982	544	982	500	2463	553	423	116
Search3	4532	3417	2002	314	96	4773	3942	2952	399	107	5424	5195	1794	262	28	3330	2890	1714	293	48	4397	898	2360	715	3937	258	2053	154
Search4	5631	5145	3128	361	86	6156	6776	3885	1292	106	5641	4984	2892	248	17	5260	6033	3268	382	47	6357	596	4421	860	2593	25	2161	140
Sum2	1731	1003	678	414	108	823	782	520	294	107	1673	881	528	192	17	811	672	581	131	25	2992	509	3438	616	1536	169	1430	309
Sum3	5537	4134	3424	874	122	4327	4885	3534	1270	168	6142	4933	3088	538	33	4007	3390	3114	1081	80	20472	636	9451	985	16449	527	7015	614
Sum4	6212	4674	5166	2086	123	9354	9209	7843	3010	178	6854	4600	5536	1879	33	7192	6624	5379	2608	100	9879	506	11667	1078	10043	135	7765	392
All	4528	3571	2749	1308	110	4945	4850	3689	1609	149	4746	3758	2588	871	32	3828	3810	3088	1218	68	16875	3131	5506	1123	14262	2492	3701	558

Table 6: Average runtime for LIA benchmarks (in seconds). Cell shading indicates higher values.

	CDGP																				GPR							
	Gener.										SteadySt.										Gener.				SteadySt.			
	Tour					*Lex*					*Tour*					*Lex*					*Tour*		*Lex*		*Tour*		*Lex*	
	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.75	1.0	0.75	1.0	0.75	1.0	0.75	1.0
CountPos2	707	609	409	146	41	186	167	169	37	83	1109	739	151	159	1466	200	168	130	52	240	1055	1130	1124	1142	874	1754	1269	1583
CountPos3	2884	3354	2695	2350	2350	3039	2873	1377	566	1476	3201	2934	2534	2043	3361	2315	2711	1867	981	2341	3440	3539	2543	2831	3380	3367	3109	3201
CountPos4	3600	3600	3600	3366	3474	3442	3314	3132	3099	3379	3600	3600	3539	3546	3600	3495	3494	3308	3284	3560	3600	3600	3358	2826	3600	3600	3600	3600
IsSeries3	3494	3387	2837	815	559	2818	2965	2890	2407	1365	2976	3259	3070	478	739	2495	2690	2923	1330	2293	3600	3396	3600	3493	3600	3507	3600	3500
IsSeries4	3503	3577	3339	2472	2763	3433	3470	3339	3179	3107	3541	3600	3395	2202	2656	3480	3499	3600	2937	2750	3600	3600	3600	3600	3600	3600	3600	3600
IsSorted4	2518	1054	1251	407	206	576	706	534	184	199	2114	1099	592	284	1265	489	1005	716	238	1033	3600	3600	3504	2218	3600	3360	3454	2889
IsSorted5	3178	2446	1580	675	672	1546	1251	971	389	1486	2713	2212	1129	314	1164	1126	1219	897	381	1326	3600	3600	3600	3530	3600	3511	3600	3510
Max4	642	677	338	157	284	129	96	104	72	66	231	107	231	40	878	320	287	313	60	62	1085	533	299	275	541	305	1148	943
Median3	1276	1053	740	205	384	521	348	231	85	325	1917	957	1118	98	1002	747	822	548	57	200	449	629	949	1180	1312	1097	1286	1189
Range3	2024	1515	1312	330	749	1113	841	274	241	747	1847	1849	1053	294	2400	1377	1586	1324	299	568	1356	345	876	367	1368	1707	1178	746
Search2	433	418	216	218	66	521	133	194	80	296	889	1554	616	145	1003	422	408	85	42	842	499	64	230	52	823	660	164	736
Search3	1499	1255	862	124	455	960	797	737	72	318	2249	2411	783	460	1770	819	874	403	78	321	865	906	628	541	1968	2693	1030	1360
Search4	2318	2034	1479	1069	1336	1424	1637	1029	524	879	2509	2039	1514	1413	3152	1922	2504	1145	142	989	1848	1075	1201	1086	2781	2982	1258	1372
Sum2	583	803	201	82	231	101	114	106	33	33	526	395	101	31	454	224	135	165	15	18	207	158	923	274	53	40	503	347
Sum3	3513	3200	1774	441	1210	1229	1603	991	275	1285	3544	3018	1442	400	2439	1707	1426	1514	451	1010	3222	1798	2573	1102	3140	1976	3266	1934
Sum4	3557	3600	3551	2301	3144	3555	3317	3494	1748	3372	3489	3600	3168	2055	3600	3600	3503	3041	2619	2724	3600	3419	3479	3076	3532	3545	3552	3430
All	2233	2036	1636	947	1120	1537	1477	1223	812	1151	2278	2086	1527	873	1934	1546	1646	1374	810	1267	2226	1962	2030	1725	2361	2357	2226	2121

synthesizing a *provably correct program* that is logically consistent with the specification (in contrast to a conventional GP which is concerned with passing supplied tests).

CDGP is clearly more likely than GPR to synthesize correct programs, which confirms that counterexamples are more useful than random inputs. Due to the curse of dimensionality, covering the input space becomes increasingly difficult in higher dimensions and GPR's performance quickly degrades with the growing cardinality of input. CDGP is affected by this phenomenon too, but to a much lesser extent. We also hypothesize that randomly drawing inputs which test certain "corner cases" (e.g., an array of the same repeated value in IsSorted) is particularly unlikely. In CDGP, to the contrary, the gradually increasing quality of programs forces the solver to come up with more and more sophisticated tests.

Using the fitness threshold q to control when programs should be verified is clearly beneficial when compared to the extremes, that is, to the conservative variant ($q = 1$) and non-conservative one ($q = 0$). Setting q to 0.75 turns out to be optimal here. As anticipated in Section 3, we hypothesize that the conservative approach is too demanding and tends to wait too long for new tests to arrive, depriving itself of potentially valuable search guidance. This is confirmed by the end-of-run size of T_c (Table 5), which is typically between one and two orders of magnitude smaller than for non-conservative variant.

Verifying all evaluated programs in the non-conservative variant ($q = 0$) is also suboptimal. What comes as a bit of a surprise is that this does not seem to lengthen the runtime (Table 6)—presumably, if all programs in a population are being verified, many of them have low fitness, and their incorrectness can be quickly proven by the solver. There must be thus another reason why the success rate for $q \leq 0.5$ is systematically worse than for $q = 0.75$. We posit that many tests collected in these settings may be in fact redundant, i.e. examine the same properties of programs (recall the earlier example of mutually redundant tests for the Max4 benchmark). Because CDGP cannot detect such redundancy, such pairs of tests (and consequently groups of tests) can coexist in T_c. The obvious consequence is that T_c grows large and slows down the evaluation. Even though this does not seem to be challenging given the time budget available here, the presence of many redundant tests decreases the relative importance of the "essential" ones. For the setups equipped with tournament selection, the contribution of nonredundant tests to the overall fitness is low, and so is the likelihood that they affect selection. In lexicase selection, the probability that such tests will reduce the pool of solutions at some point is low.

Though $q = 0.75$ seems to provide the right balance, this is not to say that this value should be considered optimal. We speculate that setting q to values closer to, yet still smaller than 1 may have a similar effect.

Concerning the type of evolutionary algorithm engaged, the results invalidate the hypothesized superiority of steady-state evolution thanks to updating solution fitness online, right after a new counterexample arrives to T_c. The possible reason is that steady-state runs tend to collect noticeably fewer tests than the generational variant.

Statistical analysis corroborates the above observations. We employ the Friedman's test for multiple achievements of multiple subjects (Kanji, 2006) on the success rate of all 28 configurations shown in the tables (20 for CDGP and 8 for GPR). The p-value 2.2×10^{-38} strongly indicates that at least one configuration performs significantly different from the remaining ones. The following table presents them ordered by the average rank, from best to worst (G/S = generational/steady state, T/L = tournament/lexicase):

GL_{75}	SL_{75}	GT_{75}	ST_{75}	GL_5	GL_1	GT_1	GL_{25}	SL_5	GL_0	SL_1	GT_5	SL_0	ST_5
4.6	5.1	5.2	6.9	7.5	8.3	9.0	9.5	10.4	11.2	11.7	12.6	12.7	14.2

SL_{25}	$GPRGL_1$	$GPRGT_1$	GT_{25}	GT_0	$GPRGL_{75}$	ST_{25}	ST_1	$GPRGT_{75}$	ST_0	$GPRST_1$	$GPRSL_{75}$	$GPRSL_1$	$GPRST_{75}$
14.5	16.6	17.1	17.8	18.8	19.2	20.4	20.7	21.0	21.3	21.8	22.2	22.4	23.0

Table 7: Success rate for SLIA benchmarks.

	CDGP																			
	Gener.										SteadySt.									
	Tour					*Lex*					*Tour*					*Lex*				
	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0
dr-name	0.04	0.12	0.52	0.68	0.44	0.00	0.32	0.40	0.56	0.72	0.08	0.60	0.72	0.68	0.80	0.04	0.40	0.60	0.72	0.88
firstname	0.92	0.92	0.96	0.92	**1.00**	0.72	0.84	0.92	**1.00**	0.96	0.88	**1.00**	**1.00**	**1.00**	**1.00**	0.72	0.84	0.80	**1.00**	0.92
initials	0.00	0.00	0.08	0.08	0.20	0.00	0.08	0.04	0.16	0.32	0.04	0.12	0.36	0.64	0.40	0.04	0.28	0.40	0.52	0.60
lastname	0.20	0.28	0.80	0.88	0.92	0.36	0.68	0.72	0.96	0.80	0.16	0.44	0.84	**1.00**	**1.00**	0.56	0.48	0.60	0.68	0.68
combine	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
combine-2	**1.00**	0.96	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.96	0.88	0.92	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
combine-3	0.92	**1.00**	**1.00**	**1.00**	**1.00**	0.96	0.96	0.88	0.96	0.92	0.84	0.96	0.96	**1.00**	**1.00**	0.96	0.96	**1.00**	**1.00**	**1.00**
combine-4	0.88	0.96	0.96	**1.00**	**1.00**	0.40	0.96	0.92	**1.00**	**1.00**	0.76	0.96	**1.00**	0.92	**1.00**	0.92	0.92	0.96	**1.00**	**1.00**
phone	**1.00**	0.96	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.96	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
phone-1	0.96	0.92	**1.00**	**1.00**	**1.00**	0.96	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.96	**1.00**	**1.00**	**1.00**
phone-2	0.76	0.72	0.84	0.96	0.96	0.60	0.80	**1.00**	**1.00**	**1.00**	0.92	0.88	**1.00**	**1.00**	**1.00**	0.92	0.84	0.96	**1.00**	**1.00**
phone-3	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.16	0.12	0.24	0.20	0.00	0.20	0.16	0.24	0.04
phone-4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.12	0.08	0.12	0.00	0.20	0.24	0.20	0.20	0.00	0.08	0.12	0.40	0.16
All	0.59	0.60	0.70	0.73	0.73	0.54	0.66	0.70	0.75	0.75	0.58	0.71	0.78	0.82	0.82	0.63	0.69	0.74	0.81	0.79

All CDGP configurations with $q = 0.75$ clearly rank at the top, followed closely by the CDGP configurations for other values of q. The GPR control configurations, on the other hand, gather at the bottom of the ranking, with a few exceptions of CDGP configurations that use the extreme q of 0 or 1.

To determine the significantly different pairs of configurations, we conduct post-hoc analysis using symmetry test (Hollander et al., 2013). The analysis reveals that all CDGP configurations with $q = 0.75$ are better than all GPR configurations ($p < 0.05$), except for ST_{75} that is not significantly better than $GPRGL_1$ and $GPRGT_1$. A number of other configurations of CDGP (GL_{25}, GL_5, GL_1, GT_1) also tend to be statistically better than four or more GPR configurations (particularly than those GPR configurations that use tournament selection).

Though the average success rates for the optimal $q = 0.75$ are very similar for both tournament and lexicase selection, the latter typically provides better rates for the remaining values of q. We may thus conclude that lexicase has once again proved its usefulness, corroborating many other studies and our results from Krawiec et al. (2017). This is even more impressive given that the lexicase algorithm is actually quite costly in execution compared to tournament selection, which becomes reflected in the average number of generations elapsed—1069 versus 102.5 for the generational variant, and 2.8 million vs. roughly 100 thousand for the steady-state variant. As a consequence, lexicase runs typically evaluate an order of magnitude fewer solutions than the tournament runs—yet, despite that, perform on a par or better.

In Tables 7, 8, and 9, we present the success rate, end-of-run size of T_c, and the runtimes for the *string domain SLIA*. We do not run GPR this time, as there is no obvious way of automatically generating plausible tests for these benchmarks, which are for the most part guarded by preconditions. A character string generated at random is very

Table 8: End-of-run size of T_c for SLIA benchmarks.

	CDGP																			
	Gener.										SteadySt.									
	Tour					*Lex*					*Tour*					*Lex*				
	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0
dr-name	402	233	43	26	24	323	92	32	38	24	325	99	15	4	3	347	152	49	11	6
firstname	136	79	34	29	27	164	87	43	29	28	143	52	14	3	3	130	64	22	5	4
initials	257	112	62	34	28	282	95	54	44	31	297	132	29	8	5	273	136	45	21	10
lastname	272	131	33	27	25	204	94	53	29	26	242	83	15	9	6	111	66	28	13	6
combine	68	48	39	39	39	70	43	40	39	38	50	17	11	5	3	30	19	8	4	5
combine-2	150	119	45	29	29	95	56	35	34	37	239	105	34	11	6	77	67	49	15	19
combine-3	223	96	50	34	30	155	63	61	37	46	231	54	26	12	5	112	63	21	10	7
combine-4	325	95	46	32	25	372	80	48	34	36	382	37	30	11	7	291	79	32	11	6
phone	89	107	40	38	37	76	72	41	40	37	53	11	13	3	2	20	21	20	3	2
phone-1	100	130	50	38	35	87	75	39	40	35	54	18	5	2	2	38	34	4	4	2
phone-2	244	213	98	57	44	253	280	76	45	47	149	139	21	10	3	222	137	31	6	8
phone-3	102	104	15	12	12	197	187	19	11	13	126	63	4	2	2	132	85	6	2	2
phone-4	96	77	11	13	13	192	157	11	12	12	115	40	4	2	2	122	47	4	2	2
All	189	119	43	32	28	190	106	43	33	32	185	65	17	6	4	147	75	25	8	6

Table 9: Average runtime for SLIA benchmarks (in seconds).

	CDGP																			
	Gener.										SteadySt.									
	Tour					*Lex*					*Tour*					*Lex*				
	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0	0.0	0.25	0.5	0.75	1.0
dr-name	3570	3288	2121	1607	2179	3600	2721	2536	2086	1572	3481	2008	1362	1856	1414	3583	2817	2145	1408	1095
firstname	1186	978	357	434	214	1617	1041	1003	450	451	1300	564	109	270	56	1698	1144	1290	154	441
initials	3600	3600	3440	3520	3118	3600	3456	3547	3306	3205	3558	3494	2514	1751	2314	3556	3086	2805	2267	2033
lastname	3110	2749	1350	934	619	2994	1983	1504	682	1140	3176	2356	978	278	302	2001	2605	1771	1301	1537
combine	148	110	52	47	48	202	65	54	47	48	112	35	22	5.3	2.3	88	78	42	13	32
combine-2	216	316	38	48	44	117	83	33	97	168	647	413	37	22	2.5	92	230	160	19	34
combine-3	1116	556	168	93	140	785	483	797	302	458	1137	314	242	49	47	606	598	171	73	86
combine-4	1583	534	270	93	74	2863	556	521	332	372	1827	349	149	397	31	1609	697	370	43	136
phone	664	920	394	398	334	464	399	269	206	248	286	66	210	6.4	4.3	80	120	158	9.1	6.4
phone-1	1003	1218	483	372	306	671	547	308	263	209	324	70	21	4.9	5.5	209	230	57	118	9.0
phone-2	1868	1859	1127	745	806	2163	2145	657	525	542	982	983	145	50	9.1	1462	960	299	41	49
phone-3	3600	3600	3600	3600	3600	3600	3600	3580	3600	3600	3600	3164	3215	2911	3004	3600	3097	3153	2992	3516
phone-4	3600	3600	3600	3600	3600	3600	3600	3305	3467	3315	3600	2930	2945	3189	2982	3600	3379	3354	2537	3205
All	1943	1794	1308	1192	1160	2021	1591	1393	1182	1179	1848	1288	919	830	782	1706	1465	1213	844	937

unlikely to pass the precondition, and consequently test *any* program property that would relate to a given task.

The overall success rates for the SLIA benchmarks turn out to be slightly smaller than for the LIA domain. We observe a similar pattern of sensitivity to the q threshold as for LIA benchmarks: best success rates for fitness thresholds around 0.75, and smaller sizes of T_c for higher fitness thresholds. Interestingly, this time the steady-state variant is noticeably better, which is striking, as the number of tests collected there is often very small, in single digits. This indicates that the good performance of these configurations is more due to visiting a large number of candidate solutions (again, often one or more orders of magnitude more than for the generational variant) than to usefulness of tests elicited by CDGP from formal verification.

We scrutinize these results statistically using the same apparatus as for the LIA benchmarks, running the Friedman's test (Kanji, 2006) on the success rate of configurations shown in the tables, the number of which is this time 20. The p-value amounts to 3.5×10^{-14} and so indicates significant differences. The configurations rank as follows (G/S = generational/steady state, T/L = tournament/lexicase):

ST_1	SL_{75}	ST_{75}	SL_1	ST_5	GL_{75}	GT_1	SL_5	GL_1	GT_{75}	ST_{25}	GT_5	GL_5	SL_{25}	GL_{25}	SL_0	GT_{25}	ST_0	GT_0	GL_0
5.4	5.8	6.2	7.0	7.5	8.6	9.0	9.3	9.4	9.5	10.3	10.7	11.0	12.1	12.7	13.6	15.3	15.3	15.3	15.8

As for LIA, the CDGP configurations with $q = 0.75$ tend to rank at the top, however, this time accompanied by a few configurations with $q = 1$. The superiority of the steady-state approach is evident. However, post-hoc analysis using the symmetry test (Hollander et al., 2013) reveals that most of pairwise differences are statistically insignificant. For instance, even though ST_1 and ST_{75} top the ranking, each of them significantly outranks only the four CDGP configurations from the very bottom of the ranking (GL_0, GT_0, GT_{25}, ST_0). The moderate number of pairwise significant differences was however expected, given that there are no dramatic differences between success rates for SLIA benchmarks—the average success rates range in [0.54, 0.82] (Table 7).

In summary, CDGP equipped with tournament selection and admitting programs for verification only if they pass at least 75% of tests is the configuration that tops the success rate on our benchmark suite. This holds for both the generational and steady-state variant, though the latter is noticeably faster than the former on SLIA problems and thus may be preferred in practice.

5.7 Comparison with Formal Approaches

We compare CDGP with two exact solvers for program synthesis: EUSolver (Alur et al., 2017), and CVC4 (Barrett et al., 2011). CVC4 is the latest in the "Cooperating Validity Checker" series of SAT-based solvers, developed over the last 30 years. It is well-known that SMT solvers do not perform well in proving universally quantified expressions to be satisfiable. CVC4 therefore supports *refutation-based synthesis*, for which a model of the function to be synthesized is obtained from the proof that the negation of the synthesis formula is unsatisfiable.

Since naïve enumerative approaches to program synthesis do not scale, EUSolver seeks to provide scalable enumeration via a divide and conquer approach that separately enumerates a) predicates for partitioning the inputs and b) small expressions which are correct on a subset of inputs. The problem of combining predicates and expressions is then treated as a multilabel decision tree learning problem. By working with a probability distribution over labels, EUSolver can take advantage of standard information-gain heuristics to induce compact trees.

We apply CVC4 and EUSolver to the LIA benchmarks, and CVC4 to the SLIA benchmarks (EUSolver cannot handle formal String specifications). As the exact algorithms are deterministic, we run them only once on each benchmark. Both methods need only a fraction of a second to synthesize a correct program for all LIA benchmarks; the average runtime of EUSolver is 0.4s (max 1.5s), and for CVC4 it is hardly measurable (below 0.01s). We may conclude that, in terms of efficiency, CDGP is no match for the exact algorithms in the LIA domain. In the SLIA domain, however, purely formal string specifications proved hard for CVC4, which managed to find a correct program only for 2 (*name-combine*, *name-combine-3*) out of 13 benchmarks.

There are, however, other metrics that make the comparison more interesting. Figure 4 juxtaposes the best-of-run programs produced by the exact methods to the average sizes of those synthesized by CDGP (in the generational variant with tournament selection and $q = 0.75$). We factor these results by benchmark class and present them as a function of instance size, i.e. the number of inputs. The sizes of programs produced by CVC4 and EUSolver grow very fast with instance size, close to exponentially (note

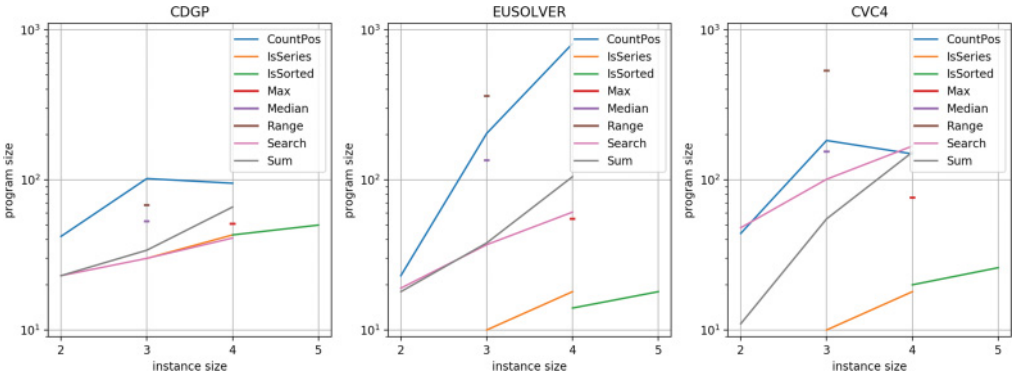

Figure 4: Program sizes for CDGP (left), EUSolver (middle), and CVC4 (right). For each group of benchmarks that represent the same problem class, we form a dataseries in function of input arity (instance size). Notice the logarithmic scale of the program size axis.

the log scale of the vertical axis). For CDGP, on the other hand, the growth is moderate and closer to linear. However, we should add at this point that CDGP *simplifies* the best-of-run solution using the SMT solver in a semantic-preserving manner. While the simplification utility is not a part of the SMT-LIB standard (Barrett et al., 2015), it is present in some SMT solvers. In Z3, we use the `simplify` command that checks locally whether a subexpression can be replaced with a shorter one (e.g., `(+ 1 1)` would be rewritten as 2). The results shown in Figure 4 count that aspect in, so one might argue that they can be biased in favor of CDGP. On the other hand, the preference for shorter programs is to some extent built into CVC4 and EUSolver by design.

To illustrate the differences in length, we present the shortest programs found for the CountPos2 problem, in which a program was supposed to count the number of arguments (`a`, `b`) greater than zero. The programs produced by EUSolver and CVC4 were additionally passed through simplification by Z3, in the same way as it is done in CDGP. Though the programs produced by CDGP for this problem were generally rather large on average (Fig. 4), the following elegant solution was found in one of the evolutionary runs:

```
(+ (ite (>= 0 b) 0 1) (ite (>= 0 a) 0 1)).
```

In comparison, EUSolver returned a solution of the following simplified form:

```
(ite (and (<= a 0) (<= b 0)) 0 (ite (and (not (<= b 0))
(not (<= a 0))) 2 1)),
```

and the simplified solution synthesized by CVC4 was even longer:

```
(ite (and (not (>= a 1)) (not (>= b 1))) 0 (ite (and (or (not (>= a 1))
(>= b 1)) (or (>= a 1) (not (>= b 1)))) 2 (ite (or (not (>= a 1))
(not (>= b 1))) 1 0))).
```

6 Discussion

The results indicate that 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, because, in contrast to counterexamples, random tests are not derived from the problem specification and in this sense convey less problem-specific knowledge. On the other hand, SMT solvers follow sophisticated search tactics, reportedly built on years of expert experience and as such involving certain search biases. It is thus not obvious that counterexamples they identify should be effective when used as "search drivers" (Krawiec, 2016) in a stochastic synthesis process.

On the other hand, it is fair to say that the effectiveness of GPR is quite high, particularly on the simpler benchmarks. The success rate of this baseline approach could form a measure of problem difficulty, which does not seem to trivially correlate strongly with input arity; compare for instance the staggering differences in success rates for CountPos4 and Search4 (Table 4). This is, however, not to say that GPR could form a competitive alternative to CDGP.

The reader familiar with contemporary software engineering has likely noticed that CDGP can be seen as an automatic analog to *test-driven software development* (Beck, 2002), where a software developer iteratively constructs tests of gradually increasing difficulty that detect flaws in the current implementation and so help improving it. This analogy holds also for other counterexample-driven methods (Jha et al., 2010; Solar-Lezama et al., 2006), and naturally brings to mind the coevolutionary metaphor, as posited in related works (Katz and Peled, 2016). Indeed, a natural follow-up of this study could involve borrowing the developments from coevolutionary algorithms, in particular coevolving the tests alongside with programs, and using measures like *distinctions* or *informativeness* to maintain them (Ficici and Pollack, 2001).

7 Conclusions

This contribution builds upon our original study on counterexample-driven genetic programming (Krawiec et al., 2017), a method for synthesizing programs from specifications. We extended CDGP with a fitness threshold parameter that controls the frequency of program verification, found that setting it to a non-extreme value (0.75) tends to systematically improve the odds of successful synthesis, and proposed an explanation for this observation. We introduced a rigorous conceptual framework for turning counterexamples into tests, based on the well-defined notions of single-output property and single-invocation property. We updated and improved several technical internals of the method and applied it to a larger suite of benchmarks, showing, among others, its capability to synthesize both integer-based (LIA) and string-processing (SLIA) programs. Last but not least, we compared CDGP to two state-of-the-art exact methods of formal program synthesis; CDGP, albeit slower, has been shown to produce shorter programs.

With this work, we also hope to help bridge the gap between the test-based and spec-based synthesis. As we argued in Section 2.1, these two paradigms, though often perceived as disparate, have certain commonalities and their marriage can be mutually beneficial. Test-based synthesis, by opening to formal specifications, may gain correctness guarantees. Spec-based synthesis, faced with the combinatorial explosion of systematic search, may benefit from including heuristics as a search guidance, and thanks to that scale better. From a broader perspective, such "middle ground" approaches address one of the fundamental—if not existential—questions of program synthesis, that is, how should user intent (Gulwani, 2010) be expressed? It is clear that tests and specifications are just the extremes of a conceivably rich spectrum.

CDGP in its current form is admittedly not free from certain challenges. The main shortcoming of the approach presented in this article are search operators. The way

CDGP exploits knowledge obtained through the use of a solver is far from sophisticated, to say the least. The search operators, taken verbatim from the standard GP, are unaware of how programs interact with tests. It seems thus desirable to make search operators better informed by the verification process, which we find the most promising further research direction.

Acknowledgments

I. Błądek and K. Krawiec acknowledge support from grant 2014/15/B/ST6/05205 funded by the National Science Centre, Poland. J. Swan acknowledges the support of the EU H2020 SAFIRE Factories project.

References

Almeida, J. B., Frade, M. J., Pinto, J. S., and Melo de Sousa, S. (2011). *An overview of formal methods tools and techniques* (pp. 15–44). London: Springer.

Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. (2013). Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design*, pp. 1–8.

Alur, R., Fisman, D., Singh, R., and Solar-Lezama, A. (2015). Results and analysis of SyGuS-comp'15. In *Proceedings Fourth Workshop on Synthesis*, pp. 3–26.

Alur, R., Radhakrishna, A., and Udupa, A. (2017). Scaling enumerative program synthesis via divide and conquer. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems—23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software Part I*, pp. 319–336.

Arcuri, A., and Yao, X. (2007). Coevolving programs and unit tests from their specification. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 397–400.

Arcuri, A., and Yao, X. (2014). Co-evolutionary automatic programming for software development. *Information Science*, 259:412–432.

Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., and Tinelli, C. (2011). CVC4. In *Proceedings of Computer Aided Verification—23rd International Conference*, pp. 171–177.

Barrett, C., Fontaine, P., and Tinelli, C. (2015). The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa. Retrieved from http://www.SMT-LIB.org

Barrett, C., Fontaine, P., and Tinelli, C. (2016). The Satisfiability Modulo Theories Library (SMT-LIB). Retrieved from http://www.SMT-LIB.org

Beck (2002). *Test driven development: By example*. Boston: Addison-Wesley.

Błądek, I., and Krawiec, K. (2017). Evolutionary program sketching. In *Proceedings of the 20th European Conference on Genetic Programming*, pp. 3–18. Lecture Notes in Computer Science, Vol. 10196.

Boca, P. P., Bowen, J. P., and Siddiqi, J. I. (2009). *Formal methods: State of the art and new directions*. 1st ed. New York: Springer.

Burles, N., Bowles, E., Brownlee, A. E. I., Kocsis, Z. A., Swan, J., and Veerapen, N. (2015). Object-oriented genetic improvement for improved energy consumption in Google Guava. In *Proceedings of Search-Based Software Engineering: 7th International Symposium*, pp. 255–261.

Cohen, B. (1994). A brief history of formal methods. *Formal Aspects of Computing*, 1(3).

de Moura, L., and Bjørner, N. (2008). Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof (Eds.), *Tools and algorithms for the construction and analysis of systems* (pp. 337–340). Lecture Notes in Computer Science, Vol. 4963. Berlin, Heidelberg: Springer.

Ficici, S. G., and Pollack, J. B. (2001). Pareto optimality in coevolutionary learning. In *6th European Conference Advances in Artificial Life*, pp. 316–325. Lecture Notes in Computer Science, Vol. 2159.

Gulwani, S. (2010). Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pp. 13–24. Invited talk.

Gulwani, S., Jha, S., Tiwari, A., and Venkatesan, R. (2010). *Component based synthesis applied to bitvector programs*. Technical Report MSR-TR-2010-12.

Harman, M., Jia, Y., and Langdon, W. B. (2014). *Babel Pidgin: SBSE can grow and graft entirely new functionality into a real world system* (pp. 247–252). Cham: Springer.

Haynes, T., Gamble, R., Knight, L., and Wainwright, R. (1996). Entailment for specification refinement. In *Proceedings of the 1st Annual Conference on Genetic Programming*, pp. 90–97.

He, P., Kang, L., Johnson, C. G., and Ying, S. (2011). Hoare logic-based genetic programming. *SCIENCE CHINA Information Sciences*, 54(3):623–637.

Helmuth, T., Spector, L., and Matheson, J. (2015). Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

Hofmann, M. (2010). Igor II—An analytical inductive functional programming system. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pp. 29–32.

Hofmann, M., Kitzelmann, E., and Schmid, U. (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*, pp. 78–86.

Hollander, M., Wolfe, D. A., and Chicken, E. (2013). *Nonparametric statistical methods*, Vol. 751. New York: John Wiley & Sons.

Jha, S., Gulwani, S., Seshia, S. A., and Tiwari, A. (2010). Oracle-guided component-based program synthesis. In *29th International Conference on Software Engineering*, pp. 215–224.

Johnson, C. (2007). Genetic programming with fitness based on model checking. In *Proceedings of the 10th European Conference on Genetic Programming*, pp. 114–124. Lecture Notes in Computer Science, Vol. 4445.

Kanji, G. K. (2006). *100 statistical test*. Thousand Oaks, CA: Sage.

Katayama, S. (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*, pp. 43–52.

Katz, G., and Peled, D. (2008). *Genetic programming and model checking: Synthesizing new mutual exclusion algorithms* (pp. 33–47). Berlin, Heidelberg: Springer.

Katz, G., and Peled, D. (2014). Synthesis of parametric programs using genetic programming and model checking. In *Proceedings 15th International Workshop on Verification of Infinite-State Systems*, pp. 70–84. Electronic Proceedings in Theoretical Computer Science, Vol. 140.

Katz, G., and Peled, D. (2016). Synthesizing, correcting and improving code, using model checking-based genetic programming. *International Journal on Software Tools for Technology Transfer*, pp. 1–16.

Kocsis, Z., and Swan, J. (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*.

Kocsis, Z. A., Drake, J. H., Carson, D., and Swan, J. (2016). Automatic improvement of Apache Spark queries using semantics-preserving program reduction. In *Genetic Improvement 2016 Workshop*, pp. 1141–1146.

Kocsis, Z. A., and Swan, J. (2017). Genetic programming + proof search = automatic improvement. *Journal of Automated Reasoning*, 60:157–176.

Krawiec, K. (2016). *Behavioral program synthesis with genetic programming*, Vol. 618 of *Studies in computational intelligence*. Retrieved from http://www.cs.put.poznan.pl/kkrawiec/bps

Krawiec, K., Błądek, I., and Swan, J. (2017). Counterexample-driven genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 953–960.

Luke, S., and Panait, L. (2002). Lexicographic parsimony pressure. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 829–836.

Muggleton, S. (1994). Inductive logic programming: Derivations, successes and shortcomings. *SIGART Bulletin*, 5(1):5–11.

Raghothaman, M., and Udupa, A. (2014). Language to specify syntax-guided synthesis problems. *Computing Research Repository*, abs/1405.5590.

Reynolds, A., Kuncak, V., Tinelli, C., Barrett, C., and Deters, M. (2017). Refutation-based synthesis in SMT. *Formal Methods in System Design*.

Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V. (2006). Combinatorial sketching for finite programs. *SIGPLAN Notices*, 41(11):404–415.

Srivastava, S., Gulwani, S., and Foster, J. S. (2010). From program verification to program synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 313–326.