Counterexample-Driven Genetic Programming: Stochastic Synthesis of Provably Correct Programs*

Krzysztof Krawiec[1], **Iwo Błądek**[1], **Jerry Swan**[2], **John H. Drake**[3]
[1] Poznan University of Technology, Poland
[2] University of York, UK
[3] Queen Mary University of London, UK
krawiec@cs.put.poznan.pl, ibladek@cs.put.poznan.pl, jerry.swan@york.ac.uk, j.drake@qmul.ac.uk

Abstract

Genetic programming is an effective technique for inductive synthesis of programs from *tests*, i.e. training examples of desired input-output behavior. 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 thereby drive the search process. When compared with 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.

1 Introduction

1.1 Program Synthesis

In *program synthesis* the aim is to generate (*synthesize*) a program in a defined programming language given the *specification* of its expected behavior. Most commonly, a specification defines an expected program behavior in terms of the return value of the program for some inputs. There are two types of such specifications:

Test-based: Each example is an $(input, output)$ pair, consisting of input to be fed into a program and the corresponding expected (correct) output. For example:

x	y	$max(x,y)$
1	1	1
4	5	5
2	0	2
...	...	...

Formal: The result of a program is constrained by a set of expressions in a certain logic. For example:

$$\begin{aligned} max(x,y) \geq x \quad \wedge \\ max(x,y) \geq y \quad \wedge \\ (max(x,y) = x \ \vee \ max(x,y) = y) \end{aligned}$$

While checking if the program passes all test cases is relatively easy — it suffices to run a program for each input and compare outputs — in the case of formal specification a formal proof must be conducted for the generated program encoded in the same logic as the constraints.

1.2 Genetic Programming

Genetic Programming (GP) is a subtype of the evolutionary algorithm (EA) metaheuristic in which the entities to be evolved are programs, most often represented as expression trees. A *population* of candidate solutions is first initialized with random programs, and then maintained through the subsequent *generations* (iterations) of the algorithm. A *fitness function* assigns a score to the candidate programs based on the estimated proximity of their behavior to the expected one. At the beginning of each new generation, a *selection function* selects programs at random from the population, biased in favour of fitter programs, and recombines them by crossover (exchanging fragments between two programs) or mutation (randomly re-generating part of the program). The old population is discarded, and the selected and modified individuals constitute the new population.

1.3 Motivation

Genetic Programming (GP) proves effective for *test-based* synthesis of programs. 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 (except specific areas like semantic GP) GP lacks a general formal theory of generalization.

*This paper is an abridged version of a paper titled "Counterexample-Driven Genetic Programming" [Krawiec *et al.*, 2017] that won a best-paper award at the GECCO-2017 conference.

Even if GP had such a theory, then as in the case of any other inductive learning approach, it would not guarantee perfect generalization. Assuring correct behavior for all admissible program inputs can be achieved only when synthesizing from formal specifications (*spec-based synthesis* in the following).

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 [Johnson, 2007] and programs [Katz and Peled, 2016], 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.

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. 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:

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

In methods of spec-based synthesis, the content (code) of p is controlled by a set of variables. To determine the values of variables that cause p to be fulfilled (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 [Jha *et al.*, 2010; Gulwani *et al.*, 2010; Srivastava *et al.*, 2010].

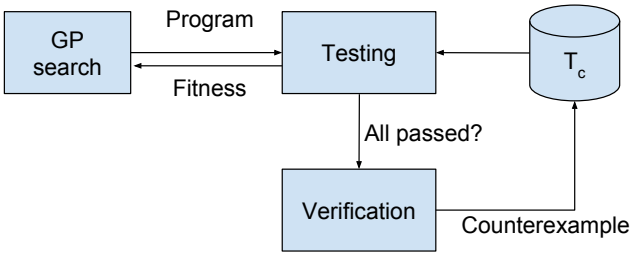

Figure 1: The conceptual diagram of CDGP.

In contrast, GP is a stochastic technique of test-based synthesis. 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 (returns 1 if the proposition is true, 0 otherwise).

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, because we employ the collected counterexamples to compute the fitness according to, amongst others, equation (3).

3 Counterexample-Driven GP

Figure 1 presents the high-level diagram of Counterexample-Driven GP (CDGP). The top-level loop of 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

```
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 = |T_c| then
6:             c ← VERIFY(p, (Pre, Post))
7:             if c = ∅ then return p          ▷ Perfect program
8:             else
9:                 T ← T ∪ {c}
10:            end if
11:        end if
12:    end for
13:    return (P, T_c ∪ T)
14: end procedure
```

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.

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. 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.

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. We also investigate a *non-conservative* variant, 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 [Cohen, 1994], and the literature for formal approaches to synthesis (and verification) is vast (for recent overviews, see Boca et al [Boca *et al.*, 2009] and Almeida et al [Almeida *et al.*, 2011]). However, we are aware of only few approaches which combine formal techniques with heuristic search.

In 2007, Johnson [Johnson, 2007] 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 [Katz and Peled, 2008; Katz and Peled, 2016] in which they progressively refine their MCGP tool 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 [Katz and Peled, 2016] 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.

The possibility of using coevolutionary GP to synthesize programs from formal specifications was researched by Arcuri and Yao [Arcuri and Yao, 2007]. 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 post-condition 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.

Amongst the dozen or more well-known systems that perform synthesis under the heading of Inductive Logic Programming [Muggleton, 1994], IGOR II [Hofmann, 2010] is known to perform well on a range of problems. As extended by Katayama [Katayama, 2012], it combines an 'analytic' approach based on analysis of fitness cases with the generate-and-test approach.

An alternative approach 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 an executable specification. More recently, Evolutionary Program Sketching (EPS) has been proposed [Błądek and Krawiec, 2017]. EPS is 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.

5 Experiment

5.1 Configuration

To assess the effectiveness of CDGP, we apply it to a range of spec-based synthesis benchmarks of varying difficulty, presented in Table 1, all of which belong to the theory of Linear Integer Arithmetic (LIA) [Barrett *et al.*, 2015], 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 repository maintained for the annual

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

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

'Syntax Guided Synthesis' competition [Alur *et al.*, 2015]; the remaining benchmarks are of our own design.

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 order to appropriately handle the two available types (`Int` and `Boolean`), we use a typed variant of GP. The initialization operator (used to populate the initial population) recursively traverses the derivation tree from the starting symbol of the grammar (`I`) and randomly picks expressions from the right-hand sides of productions. 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. Crossover draws a random node in the first parent program, and swaps it with a type-compatible subtree selected at random from the second parent program. 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.

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: *Lexicase selection* [Helmuth *et al.*, 2015] and steady-state workflow.

Communication with the solver is realized via the SMT-LIB standard [Barrett *et al.*, 2015], recognized by most contemporary SMT solvers. We employ the Microsoft Z3 SMT solver [de Moura and Bjørner, 2008], one of the most widely-used non-commercial solvers.

The source code of CDGP, along with specifications of problems, is available at https://github.com/kkrawiec/CDGP.

5.2 Results

The detailed results are presented in [Krawiec *et al.*, 2017]. The key observations are: (i) Compared to GPR's baseline, CDGP offers on average greater likelihood (success rate) of synthesizing a correct program. However, 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. (ii) Lexicase selection boosts the performance of all configurations and brings both CDGP_{Lex} and GPR_{Lex} close to each other. (iii) The conservative 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. (iv) Steady-state did not bring the anticipated benefits, neither when combined with tournament selection nor with Lexicase selection.

In the above experiment, we ignored the actual computational cost of synthesis. Although individual configurations were given the same limit on the number of evaluations ($100{,}000$), their runtimes vary heavily and are typically much higher for CDGP, mainly due to the significant computational overhead of SMT-based formal program verification. To address this issue, 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. This time, CDGP clearly proves more effective than GPR. The non-conservative variants of CDGP that use Lexicase selection have the lead again, though this time the generational variant is noticeably better.

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' [Krawiec, 2015] in a stochastic synthesis process.

6 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. Prospectively, this work may pave 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. SMT solvers support also verification in other domains, such as Reals, Strings or Lists, so there are no fundamental obstacles to use CDGP for spec-based synthesis therein.

Acknowledgements

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

H2020 SAFIRE project (Ref. 723634). JD acknowledges the support of the DAASE project (EP/J017515/1).

References

[Almeida *et al.*, 2011] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. *An Overview of Formal Methods Tools and Techniques*, pages 15–44. Springer London, London, 2011.

[Alur *et al.*, 2015] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Results and analysis of sygus-comp'15. In *Proceedings Fourth Workshop on Synthesis, SYNT 2015, San Francisco, CA, USA, 18th July 2015.*, pages 3–26, 2015.

[Arcuri and Yao, 2007] Andrea Arcuri and Xin Yao. Coevolving programs and unit tests from their specification. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 397–400, New York, NY, USA, 2007. ACM.

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

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

[Błądek and Krawiec, 2017] Iwo Błądek and Krzysztof Krawiec. *Evolutionary Program Sketching*, pages 3–18. Springer International Publishing, Cham, 2017.

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

[de Moura and Bjørner, 2008] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, chapter 24, pages 337–340. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[Gulwani *et al.*, 2010] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Component based synthesis applied to bitvector programs. Technical Report MSR-TR-2010-12, February 2010.

[Helmuth *et al.*, 2015] Thomas Helmuth, Lee Spector, and James Matheson. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643, October 2015.

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

[Jha *et al.*, 2010] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *29th International Conference on Software Engineering (ICSE '10)*, pages 215–224, May 2010.

[Johnson, 2007] Colin Johnson. Genetic programming with fitness based on model checking. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 114–124, Valencia, Spain, 11-13 April 2007. Springer.

[Katayama, 2012] Susumu Katayama. 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, pages 43–52, New York, NY, USA, 2012. ACM.

[Katz and Peled, 2008] Gal Katz and Doron Peled. *Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms*, pages 33–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[Katz and Peled, 2016] Gal Katz and Doron Peled. Synthesizing, correcting and improving code, using model checking-based genetic programming. *International Journal on Software Tools for Technology Transfer*, pages 1–16, 2016.

[Krawiec *et al.*, 2017] Krzysztof Krawiec, Iwo Błądek, and Jerry Swan. Counterexample-driven genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '17, pages 953–960, New York, NY, USA, 2017. ACM.

[Krawiec, 2015] Krzysztof Krawiec. *Behavioral Program Synthesis with Genetic Programming*, volume 618 of *Studies in Computational Intelligence*. Springer International Publishing, 2015.

[Muggleton, 1994] Stephen Muggleton. Inductive Logic Programming: Derivations, successes and shortcomings. *SIGART Bull.*, 5(1):5–11, January 1994.

[Solar-Lezama *et al.*, 2006] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *SIGPLAN Not.*, 41(11):404–415, October 2006.

[Srivastava *et al.*, 2010] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 313–326, New York, NY, USA, 2010. ACM.