



Latest updates: <https://dl.acm.org/doi/10.1145/3321707.3321743>

RESEARCH-ARTICLE

Solving symbolic regression problems with formal constraints

IWO BŁĄDEK, Poznan University of Technology, Poznan, WP, Poland

KRZYSZTOF KRAWIEC, Poznan University of Technology, Poznan, WP, Poland

Open Access Support provided by:

Poznan University of Technology



PDF Download
3321707.3321743.pdf
26 February 2026
Total Citations: 23
Total Downloads: 314

Published: 13 July 2019

Citation in BibTeX format

GECCO '19: Genetic and Evolutionary Computation Conference
July 13 - 17, 2019
Prague, Czech Republic

Conference Sponsors:
SIGEVO

Solving Symbolic Regression Problems with Formal Constraints

Iwo Bładek

Institute of Computing Science,
Poznan University of Technology
Poznan, Poland
ibladek@cs.put.poznan.pl

Krzysztof Krawiec

Institute of Computing Science,
Poznan University of Technology
Center for Artificial Intelligence and Machine Learning
Poznan, Poland
krawiec@cs.put.poznan.pl

ABSTRACT

In many applications of symbolic regression, domain knowledge constrains the space of admissible models by requiring them to have certain properties, like monotonicity, convexity, or symmetry. As only a handful of variants of genetic programming methods proposed to date can take such properties into account, we introduce a principled approach capable of synthesizing models that simultaneously match the provided training data (tests) and meet user-specified formal properties. To this end, we formalize the task of symbolic regression with formal constraints and present a range of formal properties that are common in practice. We also conduct a comparative experiment that confirms the feasibility of the proposed approach on a suite of realistic symbolic regression benchmarks extended with various formal properties. The study is summarized with discussion of results, properties of the method, and implications for symbolic regression.

CCS CONCEPTS

- Theory of computation → Program verification; • Computing methodologies → Genetic programming;

KEYWORDS

genetic programming, symbolic regression, constraints, formal verification, generalization

ACM Reference Format:

Iwo Bładek and Krzysztof Krawiec. 2019. Solving Symbolic Regression Problems with Formal Constraints. In *Genetic and Evolutionary Computation Conference (GECCO '19), July 13–17, 2019, Prague, Czech Republic*. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3321707.3321743>

1 INTRODUCTION

In genetic programming (GP), the behavior of a target program is usually specified using a set of input-output examples, and fitting them well enough with a synthesized program is often considered sufficient to consider the task being solved. There are, however,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GECCO '19, July 13–17, 2019, Prague, Czech Republic

© 2019 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-6111-8/19/07...\$15.00
<https://doi.org/10.1145/3321707.3321743>

scenarios in which one expects the returned solution to have some additional properties other than just the error on test cases.

Consider for instance using symbolic regression (SR) for scientific discovery, more specifically searching for the formula which, given the two resistors r_1, r_2 , determines the *equivalent resistance* r when they are connected in parallel. Assume an experimenter, call her Jane, performed a series of measurements of observed equivalent resistance \hat{r} for different r_1, r_2 , and collected their outcomes in a set of test cases T , with each test of the form $((r_1, r_2), \hat{r})$. Given some domain-specific set of instructions I , Jane could pose this problem as a conventional SR task and apply a ‘vanilla’ GP in order to arrive at the sought formula $\frac{1}{r} = \frac{1}{r_1} + \frac{1}{r_2}$.

However, Jane may know some properties of the problem under consideration in advance. For instance, it is rather obvious that denoting the resistance of the first resistor with r_1 and the second one with r_2 is arbitrary, and those could be swapped without affecting the equivalent resistance. The sought formula for \hat{r} should therefore be *symmetric* with respect to its arguments. Taking this fact into account might arguably (i) improve the predictive accuracy (generalization) of synthesized models, and (ii) reduce the expected runtime of a SR method by constraining the search space.

Unfortunately, the conventional GP framework does not offer means for taking such domain knowledge into account. There are arguably ad-hoc means for implementing the specific property of symmetry: for each test $((r_1, r_2), \hat{r}) \in T$, one could augment T with a test $((r_2, r_1), \hat{r})$. This has, however, several caveats: more computational effort has to be invested in testing the evolving models on the doubled number of tests, and, more importantly, this requires the model to be symmetric *only at the specific points in T* , while ideally one would like it to hold for any (r_1, r_2) pair.

One could argue that Jane might restrict the space of models and so enforce the symmetry, by for instance modifying the grammar of expressions. Unfortunately, while one might imagine designing such a grammar for the symmetry property, it would be rather cumbersome, and often impossible, to do so for other properties without seriously limiting the search space. Consider another commonsense observation that Jane could make: given a single resistor r_1 , attaching $r_2 < \infty$ in parallel can only allow for *more* current to flow through the circuit. Therefore, the equivalent resistance cannot *increase*, i.e., $\forall r_1, r_2 : \hat{r} \leq \min(r_1, r_2)$. There is no easy way (if any) to take this property into account within the conventional GP framework.

We argue that there is only a handful of practically useful formal properties that can be taken into account in standard GP, and attempting to implement them is often cumbersome and/or incomplete. On the other hand, there is the entire gamut of SR usage scenarios where some nontrivial properties of the sought model

are known in advance, so having means to impose them is clearly desirable in practice. To resolve this tension, we present here an approach based on Counterexample-Driven Genetic Programming (CDGP), which was proposed in [2, 10] to synthesize programs from formal specifications. We demonstrate how CDGP can be easily adopted to perform the above-mentioned synthesis of regression models that not only comply with the dataset of observations, but also meet some formal constraints imposed on the model (like monotonicity, symmetry, etc.).

In the following, we first formalize the task of symbolic regression with formal constraints (Section 2), then detail a method for solving such tasks in Sections 3 and 4, and in Section 5 we present the list of some properties it can handle. Sections 6 and 7 contain the description and analysis of computational experiments.

2 SYMBOLIC REGRESSION WITH FORMAL CONSTRAINTS

We pose the problem of Symbolic Regression with Formal Constraints (SRFC) as follows:

Definition 2.1. Given a set of *test cases* (examples) T , a set of *formal constraints* C , and a set/space of models \mathcal{M} , find a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$, $f \in \mathcal{M}$, that minimizes the approximation error on T , while satisfying all constraints in C .

As in standard SR, \mathcal{M} is usually implicitly defined by a set of instructions I and rules of connecting them, expressed, e.g., as a formal grammar. Throughout this study, a model is simply an algebraic formula that defines a candidate function. T is a set of input-output pairs of the form $(x, y) \in (\mathbb{R}^n, \mathbb{R})$ (though it is worth mentioning that SRFC tasks and the method for solving them we propose here can be naturally generalized beyond the realm of real numbers).

Each constraint in C is a logical formula that should be met by the model for a (often infinite) subset of input-output pairs¹, for instance $\forall x : f(x) \geq 0$. When $C = \emptyset$, SRFC reduces to the classical SR problem. If C , \mathcal{M} , and T together unanimously identify the target function f , the SRFC task is *well-defined*. This will usually require the constraints in C to be tight enough, the space of models \mathcal{M} to be somehow limited, and/or f 's approximation error on T to be zero. For instance, if \mathcal{M} is the space of polynomials of degree n , $n + 1$ tests in T are sufficient to define it unanimously at zero approximation error. Though in most scenarios discussed in the following we assume the tasks to be well-defined, that does not have to hold in general. For instance, presence of noise in tests may make it unjustified to assume that there is only one model that solves the problem.

3 COUNTEREXAMPLE-DRIVEN GP

In [2, 10], we proposed Counterexample-Driven GP (CDGP) in order to synthesize programs from formal specifications (rather than from tests). CDGP hybridizes GP with a Satisfiability Modulo Theories (SMT) solver, which is similar to a SAT solver but allows terms and operators belonging to specific theories, e.g., the theory of

¹Technically, nothing prevents a constraint from referring to only a single input-output pair (e.g., $f(3) = 7$), but such requirements can be more conveniently expressed as tests in T .

nonlinear real arithmetic (NRA). A formal specification is assumed to have the form $(Pre, Post)$, where $Pre(in)$ is the *precondition* that must be met by an input in to the program, and $Post(in, out)$ is the *postcondition*, a logical predicate that should hold upon program completion. An SMT solver can be used to verify if a given program p meets the specification by proving that:

$$\forall in \ Pre(in) \implies Post(in, p(in)), \quad (1)$$

where $p(in)$ is the output returned by p for in . In practice, it is common to request the solver to disprove the above implication, i.e., prove that

$$\exists in \ Pre(in) \not\implies Post(in, p(in)). \quad (2)$$

If the solver decides that formula (2) is unsatisfiable, p is guaranteed to meet the specification; otherwise, the solver produces a logical model, an input for which the above implication holds. Since this logical model consists of an input exposing the wrong behavior of the program, it is commonly referred to as a *counterexample*. This capability is the key mechanism behind CDGP.

The only input to CDGP is the $(Pre, Post)$ specification and the space of models \mathcal{M} , given implicitly by a grammar of models that are considered valid (syntactically correct). Internally, CDGP operates as conventional GP, i.e., maintains a randomly initialized population of programs, and performs search by mutating them and crossing them over, while exerting selective pressure using fitness function. The fitness function, as in conventional GP, evaluates the candidate programs on tests, which are stored in a set called T_c . In contrast to GP, however, T_c is initially empty, as the only specification of the task is the $(Pre, Post)$ pair. To supplement T_c with tests, CDGP uses counterexamples harvested from verification. Whenever an evaluated program p passes all tests collected so far in T_c (which is true for all programs in the initial population, as T_c is empty then), it is subject to verification using Eq. (2). If p happens to pass verification, it must be the sought solution and the run terminates with success. Otherwise, the verification process produces a counterexample that is added to T_c . Such counterexamples (which are technically just program *inputs*) are then converted to fully-fledged tests (in, out) using the procedure described in detail in [2]. In this way, T_c is gradually filled with tests, which allow the fitness function to differentiate candidate programs and so provide appropriate search gradient.

We proposed a range of variants of CDGP, including the generative and steady-state version. More importantly, however, in [2] CDGP was extended with a parameter $\alpha \in [0, 1]$ that relaxes the policy concerning verification. In this variant, a program is sent to verification as soon as it passes $\lfloor \alpha \cdot |T_c| \rfloor$ tests from T_c . For $\alpha = 1$, CDGP behaves as in the basic version described above. Experimental analysis conducted in [2] showed that setting $\alpha \approx 0.8$ is indeed beneficial in many cases when compared to $\alpha = 1$, i.e., increases the likelihood of synthesizing a program that is correct w.r.t. specification, although at the cost of more time spent on verification. Another factor that proved to help was the use of Lexicase selection [4] rather than conventional tournament selection.

4 SOLVING SRFC PROBLEMS WITH CDGP

In this section, we describe CDSR, our adaptation of CDGP to SRFC problems.

Our attempt to apply a variant of CDGP to SRFC problems stems from two observations. Firstly, formal specifications used by CDGP (and other methods of program synthesis) allow encoding the required formal properties of the model, like symmetry in the example in Introduction, and many others, which we will present in Section 5. Secondly, CDGP starts a run with an empty set of counterexamples T_c only because it was originally designed to synthesize programs from formal specifications, which normally are not accompanied with examples. There are, however, no technical obstacles for ‘seeding’ T_c with examples that a model should pass.

These two observations incline us to propose the following straightforward procedure to approach SRFC problems with CDGP. Given an SRFC problem (T, C, I) (Section 2), we apply CDGP to it by seeding T_c with T (simply $T_c := T$) and using C as the formal specification. Upon successful completion of CDGP’s run, we obtain a model that meets both the formal constraints in C and passes all tests in T .

In more detail, certain adaptations were necessary to handle SRFC problems accordingly:

- Whether a solution undergoes verification or not is determined by the ratio α of passed *incomplete* tests only, i.e., tests for which the correct output has not been determined yet. Such tests are created out of necessity when, e.g., a symmetry constraint is violated, but the task is not well-defined and no particular desired output can be assigned to given input. The responsibility of determining whether program passes such a test or not is relegated to the solver. This is different from the original CDGP, where the notion of a ‘passed test’ was non-problematic and thus all available tests (with both unknown and known outputs) were taken into account when determining with α if a program was ‘good enough’ to be subject to formal verification.
- A model is decided to be *optimal* when both (i) its aggregated error on complete tests is below the provided error threshold and (ii) it meets the formal constraints. In the original CDGP satisfying formal constraints was sufficient for solution to be considered optimal.

Even though we introduced only these two changes in CDGP, there are few facts worth noting about the operation of CDSR. Firstly, in the original benchmarks considered in [2], specifications were *complete*, i.e., they unanimously identified the target function (program) to be synthesized. Technically, however, formal specifications in CDGP do not have to be complete; in such a case, just *one* of the admissible functions (the number of which can be in general infinite) will be synthesized (provided a CDGP run is successful). In CDSR, we additionally constrain the space of such admissible solutions by seeding T_c with user-provided tests (which may, but does not have to, cause the target function to be unanimously identified).

Notice also that in both CDGP and CDSR, the tests that are gradually being collected in T_c logically follow from the specification C , and therefore do not add anything to problem formulation (i.e., do not make it more specific). For CDGP, where C is the only input to the method, that implies that any program that adheres to specification will by definition pass all tests in T_c . The only purpose of T_c there is to form the basis for calculating fitness function that drives the search. Here, to the contrary, T_c contains user-provided tests

already from the very beginning, and thus additionally constrains the set of admissible functions.

Interestingly, even if a run of CDSR does not complete with success, its outcome may still be of some value. In general, the best program found in the run may:

- (1) pass all tests and meet the constraints,
- (2) pass all tests but do not meet the constraints,
- (3) do not pass all tests but meet the constraints,
- (4) neither pass all tests nor meet the constraints,

where by ‘pass all tests’ we mean that the overall error (mean squared error (MSE) in the experiments that follow) committed by the program on the tests collected in T_c is lower than some threshold. We claim that outcomes of type 2 and 3 may still provide a user with valuable insights into the problem. In practice, tests may be subject to some noise, and it might be impossible to bring the error down to zero while meeting the constraints. As a matter of fact, we typically prefer solutions to generalize well rather than to fit meticulously to the available tests. Thus, it could be claimed that outcome of type 3 are, at least in some situations, more desired than those of type 2.

4.1 Related work

Apart from the CDGP we base our work on, there is only a handful of GP studies that explicitly involve formal constraints into evolutionary search. Johnson et al. [6] incorporated model checking with specifications expressed via Computation Tree Logic (CTL) to evolve finite state machines. Fitness was computed as the number of CTL properties satisfied for a given program, which is similar in spirit to our CDSR_{props} approach introduced in Section 6.5. A similar approach by [3], 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 [5] is used for the specification of the task and verification.

A series of papers by Katz and Peled combined model checking and GP [7–9]. The authors were progressively refining their MCGP tool based on Linear Temporal Logic (LTL), using 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).

5 EXAMPLES OF FORMAL PROPERTIES OF PRACTICAL RELEVANCE

In this section, we present a representative sample of properties that can be handled by CDSR. For each property, we discuss plausible usage scenarios and provide a snippet of specification that defines it, expressed in SMT-LIB [1], the formal notation that is nowadays the standard means of communicating with SAT solvers. In the following, f denotes the function that should meet the constraint in question.

Symmetry with respect to arguments. Many multivariate problems are expected to be symmetric with respect to their arguments. Examples include the equivalent resistance presented in Introduction and the force of gravity that remains the same if the interacting masses are swapped.

In SMT-LIB, such property can be conveniently expressed as:

```
(assert (= (f x y) (f y x)))
```

When applying CDSR, this assertion would be included in the constraint list C , while the tests/examples would be placed in T . However, let us emphasize again that the assertion requires the constraint to be met by f for *all* arguments x and y , not only for those present in T . When a CDSR run ends with success, the synthesized model is guaranteed to be symmetric with respect to its arguments.

Symmetry with respect to domain. For univariate models, it is sometimes desirable to constrain models only to functions that are even ($f(x) = f(-x)$) or odd ($-f(x) = f(-x)$). For instance, the direction of the restoring force of a spring depends on the direction of displacement, which implies that the dependency in question is an odd function $F(x) = -kx$, where k is the spring constant. Expressing symmetry with respect to domain is straightforward in SMT-LIB:

```
(assert (= (f x) (f (- x))))
```

This type of symmetry may be also useful when constraining multivariate models, where it may be selectively applied to individual variables. A bivariate model $f(x, y)$ can be demanded to be even w.r.t. x with the following assertion:

```
(assert (= (f x y) (f (- x) y)))
```

Range. There are multiple scenarios in which domain knowledge excludes certain ranges of values from f 's codomain. In physics (at least the classical one) mass cannot be negative and velocity cannot exceed the speed of light. In econometrics, employee's wage cannot be negative. In medicine, it usually does not make sense to estimate patient's life expectancy to more than, say, 120 years.

Such constraints can be conveniently expressed in SMT-LIB as:

```
(assert (<= (f x y) 120.0))
```

Monotonicity. Monotonicity is one of the most common properties expected from models induced from data. In transport, for that instance, the cost of delivery is almost always a monotonically increasing function of distance (or time).

Monotonicity can be easily expressed in SMT-LIB as the following postcondition:

```
(assert (forall ((x Real)(x1 Real))
  (=> (> x1 x) (> (f x1) (f x))))
```

This postcondition, however, will be negated per Eq. 2, so it can be from the beginning expressed in the form easier to both solve by the solver and harvest counterexample from (variables bounded in quantifiers are not readily accessible in the logical model). The 'to be negated' form of that modified constraint would look like this, where $x1$ is an additional *free variable*:

```
(declare-fun x1 () Real)
(assert (=> (> x1 x) (> (f x1) (f x))))
```

Convexity/concavity. When searching for a good model explaining the data, it may be beneficial for it to be convex in order to

efficiently perform some later optimization on that model. Convexity of a univariate function is defined as $\forall_{x,y,t \in [0,1]} f(tx+(1-t)y) \leq tf(x) + (1-t)f(y)$.

Similarly as for monotonicity, expressing convexity requires universal quantifier, but for the purpose of the verification the quantifier can be discarded in favor of additional free variables. Below we present the 'to be negated' form of the convexity property:

```
(declare-fun y () Real)
(declare-fun t () Real)
(assert (=> (and (>= t 0.0) (<= t 1.0))
  (<= (f (+ (* t x) (* (- 1.0 t) y)))
    (+ (* t (f x)) (* (- 1.0 t) (f y))))))
))
```

Changing this constraint to concavity would simply require replacing \leq with \geq ; replacing it with $<$ would demand the function to be strictly convex.

Slope. In many applications, it may be known that the rate of change of the model with respect to its input variable cannot exceed certain threshold. For instance, a body free-falling in Earth's gravitational field cannot accelerate faster than 9.81m/s^2 .

The expected value of derivative of a model $f(x)$ at some point x (here $x = 1$) can be approximated in SMT-LIB in the following way:

```
(assert (=> (= x 1.0) (<= (abs (/ (- (f (+ x 0.00001)) (f x)) 0.00001) 2.0)) 0.001)))
```

Here, we ask the slope of f to be equal to 2 ± 0.001 at point $x = 1$. The fidelity of approximation is determined by $\epsilon = 10^{-6}$. Note that this constraint affects only the slope of f at point 1, while not determining the desired *value* of f at that point. Therefore, this requirement cannot be alternatively implemented by providing tests in T that would implicitly constrain the slope.

Discussion. The above list presents only the simplest and most common properties. Other examples include periodicity ($f(x) = f(x+kT)$, $k \in \mathbb{N}$), additivity ($f(x+y) = f(x)+f(y)$), and multiplicativity ($f(x \cdot y) = f(x) \cdot f(y)$). Compound constraints can be easily created by combining the above ones with logical conjunction. Also, all above properties can be defined either globally (i.e., in the entire domain of the considered function) or locally (i.e., in an interval, at a given point, or otherwise constrained part of function's domain).

Arguably, some properties can be imposed by simply constraining the space of models \mathcal{M} and/or their parametrization. For instance, if one limits the instruction set to $\{+, *\}$ and constrains the constants to positive reals, then any polynomial $f(x)$ induced from such a grammar/language is monotonically increasing for $x \geq 0$. However, it is in general hard to assure nontrivial properties just by constraining the syntax of expressions.

6 EXPERIMENTAL SETUP

In the empirical part of this study, we propose to adopt a generalization perspective on SRFC problems. Given an SRFC problem (T, C, I) (Section 2), we expect a method to generalize from test cases in T in such a way that the synthesized model meets the properties expressed by the constraints in C . Typically, generalization is considered in *quantitative* terms, e.g., by checking the

aggregated error on tests that were not used during training. In contrast, here we focus on *qualitative* generalization, i.e., the model having expected properties/features.

In the experiments that follow, we assess thus the qualitative generalization of CDSR by confronting it with conventional tree-based GP, and to that aim we follow the following steps:

- (1) Select a set of test *problems* (Section 6.1), each defining a *target function*.
- (2) For each problem, determine some ‘natural’ formal properties (constraints, C) that could be expected of the model to hold in order to be useful (Section 6.1), and create a range of *benchmarks* by pairing the problem with all combinations of those properties.
- (3) For each benchmark, randomly generate test cases (T) according to the target function defined by the problem (Section 6.2).
- (4) Before each run, apply noise to tests in order to make the benchmark more realistic (Section 6.3) and estimate the error threshold that determines methods’ termination (Section 6.4).
- (5) Apply the compared algorithms (GP and CDSR with variants) to so prepared benchmarks (Section 6.5).

6.1 Test problems

As a base for our benchmarks, we selected three problems based on known laws of physics that are expressed as multivariate algebraic formulas: the law of gravity, equivalent resistance of two resistors in parallel, and equivalent resistance of three resistors in parallel. An individual benchmark is created by combining the problem with at least one of the following formal constraints: **symmetry** with respect to arguments, **bound on output value**, **monotonicity** with respect to an argument, or a domain-specific constraint.

Problem: **gravity**. Target function:

$$f(m_1, m_2, r) = 6.67408 \cdot 10^{-11} \cdot \frac{m_1 m_2}{r^2}$$

Considered constraints:

- s: $f(m_1, m_2, r) = f(m_2, m_1, r)$
- b: $f(m_1, m_2, r) \geq 0$
- m: strict w.r.t. both m_1 and m_2 .

Problem: **resistance2**. Target function:

$$f(r_1, r_2) = \frac{r_1 r_2}{r_1 + r_2}$$

Considered constraints:

- s: $f(r_1, r_2) = f(r_2, r_1)$
- c₁: $r_1 = r_2 \implies f(r_1, r_2) = \frac{r_1}{2}$
- c₂: $f(r_1, r_2) \leq r_1 \wedge f(r_1, r_2) \leq r_2$

Problem: **resistance3**. Target function:

$$f(r_1, r_2, r_3) = \frac{r_1 r_2 r_3}{r_1 r_2 + r_1 r_3 + r_2 r_3}$$

Considered constraints:

- s: $f(r_1, r_2, r_3) = \dots = f(r_3, r_2, r_1)$
- c₁: $r_1 = r_2 = r_3 \implies f(r_1, r_2, r_3) = \frac{r_1}{3}$
- c₂: $f(r_1, r_2, r_3) \leq r_1 \wedge f(r_1, r_2, r_3) \leq r_2 \wedge f(r_1, r_2, r_3) \leq r_3$

6.2 Generation of test cases

SRFC can be seen as a (supervised) machine learning problem. Providing a machine learning algorithm with as much training data as possible is usually desirable. Unfortunately, in some usage scenarios training data is scarce, e.g., the cost or time of their collection may be prohibitive, or the very act of data acquisition may influence the phenomenon in question. As a consequence, a learning problem is *underconstrained* – there are many models readily explaining such data, but in an implausible way, or with a poor generalization. CDSR is meant to be of help in such scenarios, by allowing users to augment data with formal constraints, either known to hold for certain, or expressing some desirable or useful properties of the model.

Following this rationale, we decided to conduct experiments with relatively low numbers of tests, namely 3, 5, and 10. Note that those numbers are low by GP standards, particularly given the multivariate nature of the problems. For all benchmarks, a test is generated by first sampling each input variable uniformly from $[0.0001, 20]$ and then querying the target function on those inputs.

6.3 Noise

In practical applications of SR, more often than not there is some source of error affecting data, which can be caused by many factors: imprecise measurements, stochasticity of the process, not accounting for some variables, etc. In order to simulate those conditions, before each run of an algorithm on a benchmark, we disturb the tests in T with noise. Noise is applied both to the inputs and desired outputs of tests. We set the magnitude of noise relative to the standard deviation σ_X of a given variable X (as estimated from T), and apply it independently to each test and variable, according to the formula:

$$\tilde{X} = X + \mathcal{N}(0, \beta\sigma_X),$$

where X is the original (exact) value of variable, $\beta > 0$ determines the magnitude of noise, and \mathcal{N} is zero-mean normal distribution. In our experiments, we set $\beta = 0.1$.

The set of tests T generated in this way, together with a set of formal constraints C and a set of instructions I described later, form an SRFC task (T, C, I) , as defined in Section 2. In general, adding noise to tests may make them inconsistent with one or more constraints in C . Nevertheless, we intentionally do not cater for that, as this may happen also in real-world scenarios like those presented in Introduction: an experimenter may know (or prefer) the sought model to have certain properties, even if the available empirical data (tests in T) say otherwise due to, e.g., measurement errors.

6.4 Error threshold

In presence of noise and rounding errors, exact fit of model to data becomes almost impossible and typically leads to undesirable overfitting, so we terminate runs if the mean squared error (MSE) of model’s output \hat{Y} w.r.t. desired output Y drops below a threshold ϵ . Using the same ϵ for all benchmarks would be questionable due to different magnitudes of Y , so we adjust it automatically based on Y ’s standard deviation σ_Y , according to the formula:

$$\epsilon = (t\sigma_Y)^2,$$

Table 1: Parameters of the genetic programming.

Parameter	Value
Number of runs	25
Population size	500
Maximum height of initial programs	4
Maximum height of trees inserted by mutation	4
Maximum height of programs in population	12
Maximum number of generations	∞
Maximum runtime in seconds	1800
Probability of mutation	0.5
Probability of crossover	0.5

where $t > 0$ is a user-defined *tolerance*, and the square was motivated by using MSE as a measure of model’s error. The user may use t to express his aspiration level concerning model’s accuracy. Experiments were conducted for $t = 0.01$ and $t = 0.1$.

6.5 Algorithms

We compare the following algorithms. The code used for experiments is available at: <https://github.com/kkrawiec/CDGP>.

GP: A ‘vanilla’ tree-based GP [13], meant to serve as a ‘blind baseline’, by which we mean that it cannot use the formal properties from C to guide search, and may thus attain them only by sheer chance (which, as the results will show, is not entirely negligible, given the relative simplicity of our target functions). Our GP implementation uses standard subtree-replacement mutation and subtree-swapping crossover. Other key parameters of GP are listed in Table 1.

A run is terminated when the best model’s $\text{MSE} \leq \epsilon$ (Section 6.4). There is no limit on the number of generations, but we cap the runtime to 0.5h. This was motivated by fairness of comparison, since CDSR tends to spend significant share of its time budget on calling solver that performs formal verification of solutions.

We use ϵ -Lexicase [11] for selection, as it proved much better than the tournament selection in initial experiments. In each selection act, the original Lexicase iterates over tests in a random order and allows solutions to pass to the next iteration only if they perform best on the current test, until only one solution is left (if tests are exhausted before that, one of remaining solutions is selected at random). The ϵ -Lexicase, instead of considering only the candidate solutions with the best result on the currently selected test, considers also those that are sufficiently close to the best ones. More specifically, we use the $\epsilon_{\ell\lambda}$ variant of ϵ -Lexicase, in which the ‘closeness’ is automatically computed based on median absolute deviation (MAD) [12], and solutions with the $|\hat{Y} - Y|$ difference lower than or equal to MAD from the best solution also proceed to the next stage of selection.

Our instruction set is rather limited and covers only the standard arithmetic: $+$, $-$, $*$, $/$. One of the limitations of CDSR is that it requires a solver equipped with so-called *theory* for a given domain, and transcendental functions are not supported by contemporary SMT solvers. The terminals are problem’s input variables (e.g., m_1 , m_2 , r for gravity) and real constants drawn uniformly from $[-1, 1]$.

CDSR: Uses the same parameters as above GP, except for the fact that the formal constraints C ignored by GP are used here to formally verify those solutions that pass all tests collected so far in T_c , while the tests from T seed T_c , as detailed in Section 4. The parameter α specifying the required ratio of passed tests before applying verification was set to 1 in order to limit the number of costly queries to the solver. Preliminary experiments for $\alpha = 0.8$ did not show significant improvement.

CDSR_{props}: This variant extends CDSR by considering each constraint in C as an additional *formal test*. Formal tests do not supplement T_c (and thus do not affect the MSE error), but are taken into account in the Lexicase selection on par with tests from T_c . As a result, the selection algorithm traverses a randomly ordered sequence of *both* tests from T_c and the formal tests. Whenever an iteration of Lexicase concerns a formal test, all candidate solutions are verified with respect to the formal constraint in question, and only those that meet it pass to the next iteration. Obviously, as the outcome of verification is binary, this does not involve the MAD normally used by ϵ -Lexicase.

CDSR_{props} is meant to better inform the search algorithm about detailed characteristics of candidate solutions in terms of individual constraints in C , and so provide for better search effectiveness.

7 RESULTS AND DISCUSSION

As introduced in Section 2, we consider a SRFC benchmark solved when the synthesized model passes both the tests in T (with the error threshold defined in Section 6.4) *and* the constraints/properties in C . The success rate on this joint requirement is thus our main performance indicator, and we present it in Table 2. As can be seen, it was hard for any method to consistently obtain both good MSE and meet the properties. CDSR_{props} fared the best, scoring the average success rate around 0.27 across all numbers of test cases for tolerance 0.1, while CDSR comes second with the average score of 0.17. While these rates may still seem low, they are roughly one order of magnitude higher than for GP; as expected, it is very unlikely for GP to synthesize a model with desirable formal properties, even though the tests implicitly convey some information about them. The fact that CDSR_{props} comes first indicates that informing the algorithm about passing/failing individual formal properties from C helps it perform more efficient search (and probably also provide better diversity in the population, given that the number of objectives considered by Lexicase is in this variant greater on average than in CDSR).

While larger tolerance on error clearly makes the task easier for all methods, the gains for GP are minuscule, while CDSR and CDSR_{props} observe larger increases, to the extent that makes them potentially useful in practice. Still, some combinations of problems and constraints (e.g., gr_m, res3_c2) turn out to be very hard for all configurations. However, we should not necessarily hasten to announce them *inherently* difficult; for instance, some properties are much harder (i.e., take more time) for solvers to verify than others, and thus consume greater share of the allocated time budget, leaving less time for performing the actual search.

It is hard to notice a clear pattern when it comes to combining properties. In some cases meeting several properties at once seems to improve the success rate (e.g., gr_bms for CDSR_{props} and

Table 2: Success rates (MSE below threshold and all formal properties met by the synthesized model). Darker shading marks higher values. Constraint ‘c’ stands for conjunction of constraints/properties c1 and c2.

tolerance	3 tests						5 tests						10 tests						
	GP		CDSR		CDSR _{props}		GP		CDSR		CDSR _{props}		GP		CDSR		CDSR _{props}		
	0.01	0.1	0.01	0.1	0.01	0.1		0.01	0.1	0.01	0.1	0.01	0.1	0.01	0.1	0.01	0.1	0.01	0.1
gr_b	0.16	0.24	0.00	0.12	0.00	0.08	0.00	0.00	0.00	0.04	0.00	0.16	0.00	0.00	0.00	0.04	0.00	0.00	0.00
gr_m	0.00	0.00	0.00	0.00	0.00	0.08	0.00	0.00	0.00	0.08	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
gr_s	0.04	0.04	0.04	0.24	0.16	0.20	0.12	0.16	0.00	0.00	0.00	0.12	0.04	0.20	0.00	0.48	0.00	0.44	0.00
gr_bm	0.00	0.00	0.00	0.00	0.04	0.04	0.00	0.00	0.00	0.04	0.00	0.04	0.00	0.00	0.00	0.04	0.00	0.04	0.12
gr_bs	0.20	0.32	0.00	0.08	0.04	0.36	0.04	0.04	0.00	0.16	0.00	0.16	0.04	0.00	0.00	0.04	0.00	0.04	0.12
gr_ms	0.00	0.00	0.04	0.04	0.00	0.08	0.00	0.00	0.00	0.04	0.00	0.16	0.00	0.00	0.00	0.12	0.00	0.16	0.00
gr_bms	0.00	0.00	0.04	0.04	0.00	0.16	0.00	0.00	0.00	0.04	0.00	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.40
res2_c1	0.00	0.00	0.28	0.52	0.20	0.60	0.00	0.08	0.48	0.48	0.44	0.72	0.00	0.00	0.16	0.68	0.00	0.76	0.00
res2_c2	0.00	0.12	0.04	0.72	0.04	0.96	0.00	0.04	0.00	0.36	0.00	0.56	0.00	0.08	0.00	0.40	0.00	0.60	0.00
res2_s	0.00	0.00	0.00	0.36	0.04	0.60	0.00	0.00	0.60	0.04	0.64	0.00	0.00	0.00	0.36	0.00	0.44	0.00	0.44
res2_sc	0.00	0.04	0.00	0.36	0.08	0.84	0.00	0.04	0.00	0.32	0.00	0.64	0.00	0.08	0.00	0.36	0.00	0.56	0.00
res3_c1	0.00	0.00	0.08	0.08	0.24	0.24	0.00	0.00	0.08	0.04	0.28	0.32	0.00	0.00	0.00	0.12	0.00	0.20	0.00
res3_c2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.04	0.00	0.04	0.00	0.00	0.00	0.08	0.00	0.04	0.04
res3_s	0.00	0.00	0.04	0.04	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.12	0.00	0.08	0.00
res3_sc	0.00	0.00	0.00	0.00	0.04	0.04	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00
mean	0.03	0.05	0.03	0.17	0.06	0.29	0.01	0.03	0.04	0.15	0.05	0.26	0.01	0.02	0.01	0.19	0.00	0.26	0.00

Table 3: Fraction of runs that produced models that meet the formal properties. Constraint ‘c’ stands for conjunction of constraints/properties c1 and c2.

tolerance	3 tests						5 tests						10 tests						
	GP		CDSR		CDSR _{props}		GP		CDSR		CDSR _{props}		GP		CDSR		CDSR _{props}		
	0.01	0.1	0.01	0.1	0.01	0.1		0.01	0.1	0.01	0.1	0.01	0.1	0.01	0.1	0.01	0.1	0.01	0.1
gr_b	0.20	0.32	0.80	0.76	1.00	1.00	0.00	0.00	0.76	0.80	1.00	1.00	0.20	0.12	0.84	0.80	1.00	1.00	1.00
gr_m	0.00	0.00	0.00	0.00	0.72	1.00	0.00	0.00	0.00	0.08	0.96	0.92	0.00	0.00	0.04	0.00	0.96	0.88	0.88
gr_s	0.04	0.04	0.76	0.92	1.00	1.00	0.12	0.16	0.84	0.92	0.96	1.00	0.16	0.24	0.84	0.88	1.00	1.00	1.00
gr_bm	0.00	0.00	0.00	0.00	0.84	0.80	0.00	0.00	0.00	0.04	0.84	0.96	0.00	0.00	0.00	0.04	0.88	0.68	0.68
gr_bs	0.28	0.36	0.84	0.88	0.92	1.00	0.12	0.08	0.84	1.00	1.00	1.00	0.16	0.16	0.92	0.96	1.00	0.96	0.96
gr_ms	0.00	0.00	0.04	0.04	0.84	0.96	0.00	0.00	0.00	0.04	0.84	0.96	0.00	0.00	0.04	0.12	0.92	0.96	0.96
gr_bms	0.00	0.00	0.04	0.04	0.88	0.96	0.00	0.00	0.00	0.04	1.00	1.00	0.00	0.00	0.00	0.00	1.00	1.00	1.00
res2_c1	0.00	0.00	0.52	0.64	0.80	0.80	0.00	0.08	0.60	0.48	0.76	0.72	0.00	0.00	0.36	0.68	0.68	0.68	0.76
res2_c2	0.00	0.12	0.28	0.72	1.00	1.00	0.00	0.04	0.08	0.44	0.80	0.84	0.00	0.08	0.08	0.40	0.56	0.80	0.80
res2_s	0.00	0.00	0.72	0.68	0.96	1.00	0.00	0.00	0.36	0.68	0.96	1.00	0.00	0.00	0.44	0.72	1.00	1.00	1.00
res2_sc	0.00	0.04	0.08	0.36	1.00	0.88	0.00	0.04	0.08	0.32	0.96	0.76	0.00	0.08	0.08	0.40	0.72	0.64	0.64
res3_c1	0.00	0.00	0.08	0.08	0.24	0.24	0.00	0.00	0.08	0.04	0.28	0.36	0.00	0.00	0.00	0.12	0.24	0.20	0.20
res3_c2	0.00	0.00	0.00	0.00	0.04	0.08	0.00	0.04	0.00	0.04	0.16	0.16	0.00	0.00	0.08	0.16	0.04	0.16	0.04
res3_s	0.00	0.00	0.20	0.36	0.72	0.48	0.00	0.00	0.60	0.52	0.72	0.68	0.00	0.00	0.04	0.16	0.16	0.24	0.24
res3_sc	0.00	0.00	0.00	0.00	0.00	0.08	0.00	0.00	0.00	0.00	0.24	0.12	0.00	0.00	0.00	0.04	0.28	0.04	0.04
mean	0.03	0.06	0.29	0.37	0.73	0.75	0.02	0.03	0.28	0.36	0.77	0.77	0.03	0.05	0.25	0.36	0.70	0.68	0.68

5 tests), while in many other cases it is detrimental (e.g., res3_sc vs. individual properties). One cannot simply say that the likelihood of success increases or decreases monotonically with the set of associated constraints. We hypothesize that this may stem from a (possibly complex) interaction of two factors. On one hand, more constraints means that there are fewer candidate models that meet them in combination. On the other, additional constraints provide more guidance for the search algorithm, in particular in the CDSR_{props} variant that uses them explicitly in the Lexicase selection.

Increasing the number of tests in a benchmark usually makes it harder for all methods, though not systematically. A trade-off analogous to the above may be at play here: more tests provide for better guidance (particularly with Lexicase, which prioritizes search in a multi-objective fashion), but simultaneously make the set of

compatible target models smaller. Recall that presence of noise makes the task significantly more difficult with each additional test.

One would expect the gravity benchmarks to be very hard due to the presence of a very small constant in its target function, but the results suggest otherwise. After the inspection of correct solutions from Table 2 we found out that about half of them are models that always return 0, and still obtain the error below the threshold while trivially meeting some properties (i.e., symmetry and bound; benchmarks with the strict monotonicity property were not affected by this, at least for CDSR configurations)².

To get better insight into the results, in Table 3 we report the fraction of runs in which a model has been found that meets the formal properties regardless of the obtained MSE. Though these numbers by definition cannot be smaller than those in Table 2, GP

²The presence of noise is the reason that 0 was not a good solution for all runs under the assumed tolerances.

Table 4: Ranks for success rates of performance indicators from Tables 2 and 3. On the left: success rate, i.e., MSE below error threshold and properties met (Friedman's test $p = 7.1 \cdot 10^{-25}$). On the right: fraction of models that meet the formal properties ($p = 1.6 \cdot 10^{-40}$).

method	rank	method	rank
CDSR _{props} _0.1	1.6	CDSR _{props} _0.1	1.5
CDSR _0.1	2.5	CDSR _{props} _0.01	1.6
GP _0.1	4.0	CDSR _0.1	3.3
CDSR _{props} _0.01	4.2	CDSR _0.01	4.1
CDSR _0.01	4.3	GP _0.1	5.1
GP _0.01	4.4	GP _0.01	5.4

keeps producing models that meet formal properties only occasionally, which is unsurprising given that it does not take into account any information about the requested properties. On the contrary, CDSR, and particularly CDSR_{props}, had very good results in that regard. Inspection of optimal solution also confirms that many of them, after simplification conducted by the solver, are exactly the target functions. This suggests that formal properties of the model can be successfully used to facilitate generalization and search in the presence of noise.

In order to investigate significance of the results, we applied the Friedman statistical test, separately to the data presented in Tables 2 and 3. In both cases, the statistical test was conducted for all 15 benchmarks and three different numbers of test cases (45 entries in total) and three methods combined with two values of tolerance (6 entries in total). The average ranks computed and the p-values resulting from the test are presented in Table 4. The observed differences proved statistically significant, and the post-hoc analysis revealed that the top two methods in each ranking were significantly better than all the other methods in their respective tables. Additionally, when it comes to meeting only formal properties (the right subtable in Table 4), both CDSR configurations were significantly better than the GP baseline.

8 CONCLUSIONS

In this paper, we have introduced CDSR, a method allowing synthesis of regression models that satisfy arbitrary formal properties alongside with the supplied set of tests. In the experiments, CDSR proved its capability to produce models that meet both tests and properties, and performed significantly better than the baseline. The variant that proved particularly good at conforming with formal properties was CDSR_{props}, which considers meeting/failing an individual property as a separate objective during selection process. We anticipate CDSR to be particularly useful in usage scenarios where the number of available tests is low or/and they are subject to strong noise.

We delineated here the class of SRFC tasks and conducted experiments on SR benchmarks of the form $\mathbb{R}^n \rightarrow \mathbb{R}$. However, it should be clear at this point that the possibility of extending the conventional specification (i.e., tests in SR) with additional constraints

is not necessarily limited to the domain of reals. Complex- and integer-valued functions are, for that instance, another promising application areas. More broadly, one could envision generalizing CDSR to a method capable of synthesizing a model from arbitrary ‘hybrid’ specifications, i.e., composed of both tests (examples) and formal properties. In this respect, CDSR is essentially limited only by the availability of SMT solver theories.

As an interesting conceptual side-effect, we posit that generalization should not only be considered quantitatively, but also qualitatively, the topic we only touched upon here. Some avenues of research regarding CDSR remain open, among others capability of solving problems requiring the use of transcendental functions.

ACKNOWLEDGMENTS

I. Bładek acknowledges support from grant 2018/29/N/ST6/01646 funded by the National Science Centre, Poland.

REFERENCES

- [1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2015. *The SMT-LIB Standard: Version 2.5*. Technical Report. Department of Computer Science, The University of Iowa. Available at www.smt-lib.org.
- [2] Iwo Bładek, Krzysztof Krawiec, and Jerry Swan. 2018. Counterexample-Driven Genetic Programming: Heuristic Program Synthesis from Formal Specifications. *Evolutionary Computation* 26, 3 (Fall 2018), 441–469. https://doi.org/doi:10.1162/evo_a_00228
- [3] Pei He, Lishan Kang, Colin G. Johnson, and Shi Ying. 2011. Hoare logic-based genetic programming. *SCIENCE CHINA Information Sciences* 54, 3 (March 2011), 623–637. <https://doi.org/doi:10.1007/s11432-011-4200-4>
- [4] Thomas Helmuth, Lee Spector, and James Matheson. 2015. Solving Uncompromising Problems with Lexicase Selection. *IEEE Transactions on Evolutionary Computation* 19, 5 (Oct. 2015), 630–643. <https://doi.org/doi:10.1109/TEVC.2014.2362729>
- [5] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. <https://doi.org/10.1145/363235.363259>
- [6] Colin Johnson. 2007. Genetic Programming with Fitness based on Model Checking. In *Proceedings of the 10th European Conference on Genetic Programming (Lecture Notes in Computer Science)*, Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar (Eds.), Vol. 4445. Springer, Valencia, Spain, 114–124. https://doi.org/doi:10.1007/978-3-540-71605-1_11
- [7] Gal Katz and Doron Peled. 2008. *Genetic Programming and Model Checking: Synthesizing New Mutual Exclusion Algorithms*. Springer Berlin Heidelberg, Berlin, Heidelberg, 33–47. https://doi.org/10.1007/978-3-540-88387-6_5
- [8] Gal Katz and Doron Peled. 2014. Synthesis of Parametric Programs using Genetic Programming and Model Checking. In *Proceedings 15th International Workshop on Verification of Infinite-State Systems*, Hanoi, Vietnam, 14th October 2013 (*Electronic Proceedings in Theoretical Computer Science*), Lukas Holik and Lorenzo Clemente (Eds.), Vol. 140. Open Publishing Association, 70–84. <https://doi.org/10.4204/EPTCS.140.5>
- [9] Gal Katz and Doron Peled. 2016. Synthesizing, correcting and improving code, using model checking-based genetic programming. *International Journal on Software Tools for Technology Transfer* (2016), 1–16. <https://doi.org/10.1007/s10009-016-0418-1>
- [10] Krzysztof Krawiec, Iwo Bładek, and Jerry Swan. 2017. Counterexample-driven Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*. ACM, New York, NY, USA, 953–960. <https://doi.org/10.1145/3071178.3071224>
- [11] William La Cava, Lee Spector, and Kourosh Danai. 2016. Epsilon-lexicase Selection for Regression. In *GECCO '16: Proceedings of the 2016 Annual Conference on Genetic and Evolutionary Computation*, Tobias Friedrich (Ed.). ACM, Denver, USA, 741–748. <https://doi.org/doi:10.1145/2908812.2908898>
- [12] T. Pham-Gia and T. L. Hung. 2001. The Mean and Median Absolute Deviations. *Math. Comput. Model.* 34, 7–8 (Oct. 2001), 921–936. [https://doi.org/10.1016/S0895-7177\(01\)00109-1](https://doi.org/10.1016/S0895-7177(01)00109-1)
- [13] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. 2008. *A Field Guide to Genetic Programming*. Published via <http://lulu.com>. <http://www.gp-field-guide.org.uk/>