Counterexample-Driven Genetic Programming for Symbolic Regression with Formal Constraints

Iwo Błądek and Krzysztof Krawiec

***Abstract*—In symbolic regression with formal constraints, the conventional formulation of regression problem is extended with desired properties of the target model, like symmetry, monotonicity, or convexity. We present a genetic programming algorithm that solves such problems using a Satisfiability Modulo Theories solver to formally verify the candidate solutions. The essence of the method consists in collecting the counterexamples resulting from model verification and using them to improve search guidance. The method is exact: upon successful termination, the produced model is guaranteed to meet the specified constraints. We compare the effectiveness of the proposed method with standard constraint-agnostic machine learning regression algorithms on a range of benchmarks, and demonstrate that it outperforms them on several performance indicators.**

***Index Terms*—Symbolic regression, Constraints, Satisfiability Modulo Theories, Genetic Programming**

I. Introduction

Contemporary machine learning continues to be primarily data-centric in assuming that most of relevant information about a problem can be induced from its training sample. This simplifies designing models and training algorithms, but also deprives them of valuable domain knowledge, and increases their proneness to overfitting, especially when data is scarce.

While full account of domain knowledge is usually out of reach in practical settings, parts of it are often available and can be conveniently expressed as *constraints*. The usefulness and expressive power of constraints has been demonstrated by, among others, the theory and practice of continuous and discrete optimization. In regression problems, which are the subject of the study, a simple constraint may for instance require the output variable to be bounded to avoid damage to a controlled hardware component. More complex constraints can engage multiple variables, e.g. the predicted dose of an active substance administered to a patient may need to monotonously increase with patient's weight, or/and decrease with the duration of therapy.

In conventional regression, constraints are implicitly imposed by the choice of the *form* of the model; for instance, a logistic model will be more appropriate for some problems than a linear one. However, just choosing the form of a model is often insufficient to reflect the relevant intricacies of the domain. Moreover, in symbolic regression (SR) that we consider here, the exact form of the model is not mandated: it is constrained only by the grammar of expressions that can be built from the available set of arithmetic operators, elementary functions and constants. The space of models in SR often subsumes the conventional regression (linear, polynomial, and more), and overfitting becomes thus even more likely.

To address this challenge, we propose a SR method that, in addition to the training sample, can incorporate constraints expressed in transparent fashion and produces models that are *guaranteed* to meet them. We coin such augmented task *Symbolic Regression with Formal Constraints* (SRFC), and formalize it in Section II. The constraints can be supplied by the user, who either knows beforehand that they are true of the system in question, or simply finds them desirable or beneficial. In Section III, we enumerate and exemplify a range of types of constraints of practical relevance. In Sections IV and V we present the method and related work. In Section VII we assess it on a suite of benchmarks proposed in Section VI, both in *quantitative* fashion, by measuring test-set generalization error, as well as in a *qualitative* fashion, i.e., in terms of constraints that are satisfied by the synthesized regressor. Computational experiments include analyzing different variants of the method and comparison with the state of the art constraint-agnostic regression algorithms.

II. Symbolic Regression with formal constraints

Following our preliminary study on this topic [1], we define *Symbolic Regression with Formal Constraints* (SRFC) as an extension of the SR task in which, alongside a set of input-output examples (tests), a set of constraints is also given, which the synthesized function is supposed to satisfy. SRFC is a special case of *supervised learning with constraints* [2].

Definition II.1. (*Symbolic Regression with Formal Constraints*) Given (i) a training set T of n examples $(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(n)}, y^{(n)})$, (ii) an error function $L : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, (iii) a set $\mathcal{M}$ of admissible mathematical expressions, and (iv) a set of constraints C, find a function $f \in \mathcal{M}$ that minimizes L and satisfies all constraints in C.

As in ordinary regression, L measures and aggregates the deviation of each $\hat{y}^{(i)} = f(\boldsymbol{x}^{(i)})$ from the corresponding $y^{(i)}$. Each constraint in C is a logical formula that should be satisfied by f for an (often infinite) subset of its domain, for example $\forall x : f(x) \geq 0$, or $\forall x : f(x) = f(-x)$. Technically, a constraint can define function's output for a single input (e.g., $f(3) = 7$), making it similar to the examples in T. However, such *hard constraints* do not allow for any error, and are thus

The authors are with the Institute of Computing Science, Poznan University of Technology, Poznań, Poland, e-mails: ibladek@cs.put.poznan.pl, krawiec@cs.put.poznan.pl.

Manuscript received December 12, 2021; revised June 20, 2022.

0000–0000/00$00.00 © 2022 IEEE

not equivalent to examples in T. When $C = \emptyset$, the above task reduces to conventional SR.

The definition of SRFC is similar in spirit to that of Syntax-Guided Synthesis (SyGuS) [3], where solutions are required to be constructed based on the provided formal grammar. Contrary to SyGuS, however, in SRFC there is a set of training examples, which are not necessarily supposed to be fitted perfectly (as this may lead to overfitting), but rather serve as a basis for the discovery of the model that explains them most adequately in terms of the error L and constraints C.

III. Sources and Types of Constraints

Constraints can be easily deduced from the symbolic form of a model; but how to obtain them for real-world problems, when the underlying model is unknown? We argue that there are several sources of constraints in such scenarios:

- Inference from a training set — training data can be inspected for the satisfaction of a certain set of constraints. Presence of noise can make this harder and require relaxation of constraint satisfiability.
- Desirable or required properties of models — in some applications it is beneficial (and sometimes critical) for models to exhibit certain properties. For example, if one needs to explain the predictions of a model (e.g., regarding credit), then monotonicity w.r.t. the most important features may be required [4]. If the output of a model is meant to control a piece of hardware, that will usually require bounding it to a technically admissible interval.
- Expert knowledge and common sense — sometimes certain properties of a model are known or assumed a priori. For example, constraints appear in the domain of marketing mix modeling [5], where it is common to assume that an increase of advertising cannot negatively impact sales [6]. Another example is the use of constraints representing linguistic knowledge in natural language processing tasks, e.g., identifying roots of Hebrew words [7], or learning named entities and relations between them [8].
- Scientific method — positing the existence of specific constraints is a typical part of the hypothesis-driven scientific process, which is in fact never based only on observation (induction), but also on empirical falsification of hypotheses formulated given the knowledge obtained at the earlier stages of the process.

In the remainder of this section, we present a number of formal constraints that are common in practical applications of SRFC. For each constraint, we discuss plausible usage scenarios and provide its specification in SMT-LIB [9], [10], the standard language of communication with Satisfiability Modulo Theories (SMT) solvers, which we use in our approach.

Symmetry with respect to arguments. Many multivariate models are expected to be symmetric with respect to the order of their arguments. Examples include the equivalent resistance of a number of electrical resistors (chained or arranged in parallel), and the force of gravity that remains the same if the interacting masses are swapped. In SMT-LIB, this can be expressed as:

```
(assert (= (f x1 x2) (f x2 x1)))
```

In SRFC, this assertion would be included in the set of constraints C, while the examples would be placed in T. However, let us emphasize again that the assertion requires `f` to meet the constraint for *all* possible values of `x1` and `x2`, not only for those present in T. Upon successful solving of a SRFC task, the synthesized model is guaranteed to be symmetric with respect to its arguments.

Symmetry with respect to argument's sign. It is sometimes desirable to require models to be even functions ($f(x) = f(-x)$) or odd functions ($-f(x) = f(-x)$). In classical physics, 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 such constraints in SMT-LIB is straightforward:

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

Such 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 with respect to `x` with the following assertion:

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

Bound (Range). There are multiple scenarios in which domain knowledge excludes certain ranges of values from f's codomain. In classical physics, mass cannot be negative and velocity cannot exceed the speed of light. In econometrics, employee's wage cannot be negative. In medicine, it may not make sense to estimate patient's life expectancy to more than 120 years. The last of these constraints can be expressed in SMT-LIB as:

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

Monotonicity. Monotonicity is one of the most common properties required from models. In transportation, the cost of delivery is almost always a monotonically increasing function of distance (or time). Such a constraint can be encoded as:

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

Convexity/concavity. Convex models are often desirable, because they can be later efficiently optimized. Convexity of a univariate function can be defined using Jensen's inequality:

$$\forall_{x,y,t\in[0,1]}\ f(tx + (1-t)y) \leq tf(x) + (1-t)f(y).$$

Similarly as for monotonicity, convexity constraint requires the universal quantifier:

```
(assert (forall ((t Real)(x Real)(x1 Real))
    (=> (and (>= t 0.0) (<= t 1.0))
    (<= (f (+ (* t x) (* (- 1.0 t) x1)))
        (+ (* t (f x)) (* (- 1.0 t) (f x1)))))
))
```

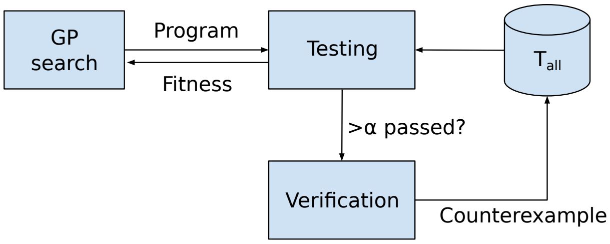

Fig. 1. The conceptual diagram of CDSR.

Changing this constraint to concavity requires replacing <= with >= in the quantified formula; replacing it with < would mandate the function to be strictly convex.

Slope. In a given application, it may be known that the rate of change of model's output with respect to its input cannot exceed a certain threshold. For instance, a body free-falling in Earth's gravitational field cannot accelerate faster than $9.81\ m/s^2$. For SMT solvers, we express that by approximating the derivative with a finite differential $(f(x+\epsilon)-f(x))/\epsilon$. In the following SMT-LIB formulation, we assume that the expected derivative of a function $f(x)$ is 2 at $x=1$, $\epsilon = 10^{-6}$, and the error tolerance of 0.001:

```
(define-fun df ((x Real)) Real
    (/ (- (f (+ x 0.000001)) (f x)) 0.000001))
(assert (=> (= x 1.0)
    (<= (abs (- (df x) 2.0)) 0.001)))
```

Note that this constraint affects only the slope of f at point 1.0, while not determining the desired *value* at that point. Therefore, it cannot be alternatively enforced with input-output tests in T that would implicitly constrain the slope, because such tests would also necessarily fix the values of f.

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

IV. Counterexample-Driven Symbolic Regression

Counterexample-Driven Symbolic Regression (CDSR) [1] allows genetic programming (GP), a heuristic global optimization technique, to produce provably correct solutions to SRFC tasks. It builds upon Counterexample-Driven Genetic Programming (CDGP) [11], [12], and uses an SMT solver to formally verify correctness of candidate solutions and use the resulting counterexamples to augment the training set.

Fig. 1 presents the key components of CDSR, which correspond to the elements of SRFC task (Section II):

- **GP search** – an algorithm responsible for generating candidate programs in $\mathcal{M}$. We use conventional generational GP, with initialization and search operators detailed in the experimental part; however, in principle any generate-and-test search algorithm could be used here.
- **The working set of test cases** $T_{all} = T \cup T_{counter} \cup T_{props}$, initialized with the training set (T) and augmented with additional tests created from counterexamples ($T_{counter}$) and constraints (T_{props}; only in the CDSR$_\text{p}$ variant; see Section IV-C).
- **Testing** – a procedure for evaluating candidate programs and returning their fitness computed on T_{all}. In the simplest scenario, fitness is the error L committed by a program on both the training set and counterexamples.
- **Verification** – an SMT solver that verifies the correctness of programs with respect to constraints in C.

The main loop of CDSR extends the traditional fitness evaluation of GP as follows. The GP search produces a candidate solution p and submits it to Testing. The error committed by p on T_{all} becomes its fitness. If p passes at least the ratio α of tests in $T \cup T_{counter}$, it is submitted to Verification. If p does not satisfy the constraints in C, then a counterexample is found and added to $T_{counter}$. If p satisfies the constraints, the search still continues, since minimization of the error L is one of the objectives in SRFC. We detail these components in the sections that follow.

A. Verification of programs

For formal verification of candidate solutions, CDSR uses a Satisfiability Modulo Theories (SMT) solver [13], [14]. An SMT problem is an extension of the SAT problem that allows for terms and operators from specific theories, e.g., the theory of nonlinear real arithmetic (NRA) used in this work. Crucially, SMT provides decision procedures for proving logical formulas expressed in a given theory.

A formal specification is assumed to have the form $(Pre, Post)$, where Pre and $Post$ are logical formulas over a certain theory. $Pre(x)$ is the *precondition* that must be met by an input x to the program, and $Post(x, y)$ is the *postcondition*, a logical predicate that should hold upon program completion. The constraints presented earlier in Section III are examples of postconditions. An SMT solver can be used to verify if a given program p meets the specification by proving that:

$$\forall_x \ Pre(x) \implies Post(x, p(x)), \tag{1}$$

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

$$\exists_x \ Pre(x) \;\not\Longrightarrow\; Post(x, p(x)). \tag{2}$$

If the solver decides that formula (2) is unsatisfiable, p is guaranteed to meet the specification; otherwise, the solver produces a *logical model*, i.e., an input x 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*.

B. Evaluation of programs

The main evaluation loop in CDSR is presented in Algorithm 1. In evaluation of the population P in a given

Algorithm 1: Evaluation in CDSR, given the current population P, the current sets of tests (T, $T_{counter}$, T_{props}), program specification $Spec \equiv (Pre, Post)$, and verification ratio α, returns the evaluated population together with an updated set of counterexamples. NUMPASSED counts the number of passed tests. VERIFY verifies a program and returns a counterexample when it is incorrect. See Algorithm 2 for EVAL.

```
1:  function CDSREVAL($P$, $T$, $T_{counter}$, $T_{props}$, $Spec$, $\alpha$)
2:      $T_{new} \leftarrow \emptyset$
3:      $T_\alpha \leftarrow T \cup T_{counter}$
4:      for all $p \in P$ do
5:          $p.eval \leftarrow$ EVAL($p$, $T$, $T_{counter}$, $T_{props}$, $Spec$)
6:          if NUMPASSED($p.eval$, $T_\alpha$) $\geq \alpha|T_\alpha|$ then
7:              $x_c \leftarrow$ VERIFY($p$, $Spec$)
8:              if $x_c \neq \emptyset$ then
9:                  $T_{new} \leftarrow T_{new} \cup \{(x_c, null)\}$
10:     $T_{counter} \leftarrow T_{counter} \cup T_{new}$
11:     return $(P, T_{counter})$
```

generation, the newly created test cases are collected in a temporary set T_{new}, and that set is merged into $T_{counter}$ only at the end of a generation, so that all solutions in P are assessed with respect to the same set of tests. Duplicates are discarded in merging.

The verification is invoked in line 6 of the algorithm and produces a counterexample x_c if a program does not satisfy the constraints. We want to use x_c as additional guidance for GP search, in addition to the original examples. However, x_c only defines an *input* that violates the specification, which is incompatible with the (x, y) representation of regular tests. For such inputs, there are usually many (and often infinitely many) corresponding outputs that meet the postcondition. Therefore, we transform x_c into an *incomplete test* of the form $(x_c, null)$. Incomplete tests require different handling than the *complete tests* (x, y) provided in T. We denote the set of all incomplete tests by $T_{counter}$.

Verification can be computationally costly, so CDSR verifies only programs that pass at least the ratio α of tests from $T_\alpha = T_{all} \setminus T_{props}$, $\alpha \in [0, 1]$, calculated by NUMPASSED in line 6 of the algorithm. Passing is defined differently for incomplete and complete tests, and in both cases is based on the information returned by the EVAL function (Algorithm 2) called in line 5 that returns an *evaluation vector* $eval$ of errors committed by a program p on all tests in T_{all}. The error on an incomplete test $(t.x, null)$ is determined by calling the SMT solver via ISCORRECT in line 6 of Algorithm 2, which returns 0 to mark passing of a test ($p(t.x)$ satisfies the constraints), and 1 for failing. The error on a complete test $(t.x, t.y)$ is determined in a conventional way, by calculating the difference between the output $p(t.x)$ produced by the program and $t.y$ (line 4), and the corresponding element of $eval$ is set to $|p(t.x) - p.y|$. Solutions with smaller values in $eval$ are preferred, and a zero $eval$ vector is an ideal evaluation.

The evaluation vector $eval$ is used at a few steps of the algorithm, one of them being the determination of the number of passed tests in the NUMPASSED function mentioned earlier: for incomplete tests, we simply count the number of those passed; for complete tests, we use a relative threshold of 5% of the target output of a given test, i.e., a test (x, y) is considered passed if $|\hat{y} - y|/|y| < 0.05$. Another use of $eval$ is to perform selection of candidate solutions (Section IV-D).

It is worth noting that the SMT solver serves two purposes in CDSR: verification of programs (VERIFY) and testing of programs on tests (ISCORRECT, SATPROPERTY).

C. CDSR$_p$: CDSR with properties

In the basic variant of CDSR described above, the outcome of the program's confrontation with the specification in VERIFY does not influence its evaluation vector $eval$: it can only give rise to an incomplete test to be used for evaluating programs in subsequent generations. One may wonder whether allowing programs to be *directly* confronted with constraints could lead to a more informative search guidance.

This observation inclined us to introduce an extended variant, dubbed CDSR$_p$, in which we augment T_{all} with the additional tests T_{props} that verify if a candidate program passes a given constraint. Technically, given a specification $(Pre, Post)$, $Post$ can be represented as a conjunction of one or more *properties* $Post_i$. For each property, we create a test of the form $(Pre, Post_i)$. In EVAL, properties are treated similarly to incomplete tests, i.e., the SMT solver is invoked to verify whether p meets a property, in which case we set the corresponding element of $eval$ to 0, and otherwise to 1. In Algorithm 2 this is represented by calling the function SATPROPERTY in line 8. In the following, we treat properties like incomplete tests, with the exception that they are not counted towards the verification ratio α, since that could stop the counterexample generating process altogether in certain circumstances (e.g., $\alpha = 1$). Unless stated otherwise, further considerations about CDSR apply also to CDSR$_p$.

D. Using evaluation vectors for selection

The basic variant of CDSR uses simple tournament selection, which requires scalar fitness, so we compute the sum of squares of the elements of $eval$ (i.e., the square error), and use the resulting scalar as fitness to be minimized. The impact of failing an incomplete test (and a property in CDSR$_p$) is thus unitary (deteriorates the fitness by 1), while each complete test (x, y) contributes $|\hat{y} - y|^2$ to the fitness. The relative influence of complete and incomplete tests depends thus on the errors committed on complete tests. This is, unfortunately, task-specific: if the output variable y has large magnitude or/and the training set T is large, the relative importance of constraints is low. This could be addressed by weighing the elements of $eval$ that represent the outcomes of complete and incomplete tests. However, tuning that parameter per benchmark would likely be tedious.

Rather than that, we propose a non-scalar variant of CDSR, which relies on lexicase selection [15], a selection method that treats each test as a separate objective and so avoids aggregation of solution's performance on individual tests. This characteristic addresses the incomparability of solution's performance on complete and incomplete tests. We rely on ϵ-lexicase [16], which can handle continuous test outcomes, and

Algorithm 2: Evaluation of a single program p in CDSR, given the current sets of tests (T, $T_{counter}$, T_{props}), and program specification $Spec \equiv (Pre, Post)$, returns an evaluation vector. IsCorrect returns 0 if output of a program p for input $t.x$ satisfies $Spec$, and 1 otherwise. SatProperty returns 0 if a program p satisfies the constraint associated with test $t \in T_{props}$, and 1 otherwise.

```
function Eval($p$, $T$, $T_{counter}$, $T_{props}$, $Spec$)
    $eval \leftarrow []$
    for all $t \in T$ do
        $eval.append(|p(t.x) - t.y|)$
    for all $t \in T_{counter}$ do
        $eval.append\big($IsCorrect$(p(t.x), t.x, Spec)\big)$
    for all $t \in T_{props}$ do
        $eval.append\big($SatProperty$(p, t)\big)$
    return $eval$
```

let it directly inspect the evaluation vectors. Given a population P of programs, each holding an evaluation vector $eval$ of length n, a single act of selection proceeds as follows:

1) Let I be the set of indices of tests, $I = \{1, \ldots, n\}$.
2) A random index i is drawn from I without replacement.
3) If i corresponds to an incomplete test in T_{all}, all programs that fail it are discarded from P (unless they *all* fail, in which case P remains intact). If i corresponds to a complete test, programs that commit on it errors greater than the median absolute deviation from the median of errors committed by all programs on that test [16] are discarded.
4) If $|P| = 1$, the only program left in P is returned as the outcome of selection.
5) If $I = \emptyset$, a random element of P is returned. Otherwise, go to step 2.

E. Weighing of properties in CDSR$_\mathrm{p}$

Lexicase selection does not fuse the outcomes of complete and incomplete tests, and is thus immune to the output magnitude problem. Nevertheless, the relative importance of complete tests and properties in CDSR$_\mathrm{p}$ still depends on their *numbers* in T_{all}. The number of properties is usually small (up to 7 in the benchmarks used in this paper) compared to the size of the training set (300). As a result, their impact on selection can be relatively low.

To address this problem, we equip CDSR$_\mathrm{p}$ with a parameter w_p that weighs the contribution of properties to the selection process. When using CDSR with lexicase selection, w_p impacts the odds of drawing incomplete tests in step 2 (Sec. IV-D. Consider T_{all} that holds 9 complete tests and 1 property: with the default setting of $w_p = 1$, the probability of using the property in the first iteration of selection is $^1/_{10}$. Setting $w_p = 3$ increases it to $^3/_{12}$. Analogously, in the CDSR variant with tournament selection, the binary outcomes of testing programs for properties (0 or 1) are multiplied by w_p, so that they effectively become 0 or w_p, and thus not satisfying a constraint is associated with a larger penalty.

F. Stopping condition and calculation of result

To reduce overfitting, CDSR performs early stopping by terminating search when the error of the best-so-far solution on a validation set (comprising 75 examples in our experiments) does not improve in a certain time window (here: 25 generations). The solution with the lowest error on the validation set is updated throughout the run and returned as a final result.

V. Related work

Apart from the CDGP [11], [12] that CDSR is based on, only a handful of studies explicitly introduce formal constraints in evolutionary program synthesis.

Johnson [17] incorporated *model checking* by specifying tasks via Computation Tree Logic (CTL) to evolve finite state machines, and used it to learn a controller for a vending machine. The fitness was the number of satisfied CTL formulas. A similar approach by He et al. [18] computes fitness as the number of postcondition clauses which can be inferred from the precondition and the program being evaluated. Hoare logic was used to specify tasks and verification.

Katz and Peled considered combining model checking and GP [19], [20]. In their method, program specification consists of several independent Linear Temporal Logic (LTL) properties, and several levels of passing a property are defined (i.e., passing for all/some/no input). Other than these levels and the LTL formalism, this approach is very similar to the two described above. For parametric programs (i.e., with unbounded input size), the authors abandoned the idea of providing full correctness guarantees and tested programs on counterexamples found by model checking. In [19] Katz and Peled briefly considered using an SMT solver for verification instead of model checking, and even used counterexamples to provide for more granular fitness in a similar spirit as CDSR.

The use of coevolutionary GP to synthesize programs from formal specifications in first order logic (augmented with arrays and arithmetic operators) was researched by Arcuri and Yao [21]. They maintained separate populations of tests (generated from the specification) and programs within a competitive coevolution framework. Programs were rewarded for passing tests and tests for failing programs. The 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, so there is no guarantee that the returned program is consistent with the specification for all possible inputs.

In formal approaches to program synthesis, the closest approach to CDSR is Counterexample Guided Inductive Synthesis (CEGIS), introduced by Solar-Lezama et al. [22], [23]. CEGIS is a general scheme of combining an inductive program synthesizer with a formal verification procedure. One starts with a randomly generated test case, from which the synthesizer produces a program. The program is verified, which returns a counterexample that is added to the set of test cases. This cycle is repeated until a globally correct program is found. From that perspective, CDSR is an instance of CEGIS, where the inductive program synthesizer is GP, and the verification is realized by an SMT solver.

TABLE I
THE LIST OF SRFC BENCHMARKS. $\Delta f(x) > 0$ MEANS f IS MONOTONOUSLY INCREASING WITH x.

Benchmark: gravity **Solution:** $f(m_1, m_2, r) = \frac{6.674 \cdot 10^{-11} m_1 m_2}{r^2}$ **Precondition:** $m_1, m_2, r > 0$ **Constraints:** $f(m_1, m_2, r) = f(m_2, m_1, r)$ $f(m_1, m_2, r) \geq 0$ $\Delta f(m_1) > 0$ $\Delta f(m_2) > 0$	**Benchmark:** keijzer5 **Solution:** $f(x, y, z) = \frac{30xz}{(x-10) \cdot y^2}$ **Precondition:** $y \neq 0, x \neq 10$ **Constraints:** $x = z = 0 \implies f(x, y, z) = 0$ $x = y = z \ \wedge \ x > 10 \implies f(x, y, z) > 0$ $x = y = z \ \wedge \ x < 10 \implies f(x, y, z) < 0$	**Benchmark:** keijzer12 **Solution:** $f(x, y) = x^4 - x^3 + \frac{y^2}{2} - y$ **Precondition:** none **Constraints:** $x \geq 0 \implies f(x, y) \leq f(-x, y)$ $y \geq 0 \implies f(x, y) \leq f(x, -y)$ $x = y = 0 \implies f(x, y) = 0$ $x \leq 0 \implies \Delta f(x) < 0$ $y \geq 1 \implies \Delta f(y) > 0$ $y \leq 1 \implies \Delta f(y) < 0$
Benchmark: keijzer14 **Solution:** $f(x, y) = \frac{8}{2 + x^2 + y^2}$ **Precondition:** none **Constraints:** $f(x, y) \geq 0$ $f(x, y) \leq 4$ $f(x, y) \leq f(0, 0)$ $f(x, y) = f(y, x)$	**Benchmark:** keijzer15 **Solution:** $f(x, y) = \frac{x^3}{5} + \frac{y^3}{2} - y - x$ **Precondition:** none **Constraints:** $x = y = 0 \implies f(x, y) = 0$ $x = -y \ \wedge \ x \leq 0 \implies f(x, y) \geq 0$ $x = -y \ \wedge \ x \geq 0 \implies f(x, y) \leq 0$	**Benchmark:** nguyen1 **Solution:** $f(x) = x^3 + x^2 + x$ **Precondition:** none **Constraints:** $x > 0 \implies f(x) \geq 0$ $x < 0 \implies f(x) \leq 0$ $x > 0 \implies f(x) \geq f(-x)$
Benchmark: nguyen3 **Solution:** $f(x) = \sum_{k=1}^{5} x^k$ **Precondition:** none **Constraints:** $x > 0 \implies f(x) \geq 0$ $x < 0 \implies f(x) \leq 0$ $x > 0 \implies f(x) \geq f(-x)$	**Benchmark:** nguyen4 **Solution:** $f(x) = \sum_{k=1}^{6} x^k$ **Precondition:** none **Constraints:** $x > 0 \implies f(x) \geq 0$ $x < 0 \implies f(x) \geq -0.75$ $x > 0 \implies f(x) \geq f(-x)$	**Benchmark:** pagie1 **Solution:** $f(x, y) = \frac{1}{1 + x^{-4}} + \frac{1}{1 + y^{-4}}$ **Precondition:** $x, y \neq 0$ **Constraints:** $f(x, y) \geq 0$ $f(x, y) \leq 2$ $f(x, y) = f(y, x)$
Benchmark: res2 **Solution:** $f(r_1, r_2) = \frac{r_1 r_2}{r_1 + r_2}$ **Precondition:** $r_1, r_2 > 0$ **Constraints:** $f(r_1, r_2) = f(r_2, r_1)$ $f(r_1, r_2) \leq r_1 \ \wedge \ f(r_1, r_2) \leq r_2$ $f(r_1, r_2) > 0$	**Benchmark:** res3 **Solution:** $f(r_1, r_2, r_3) = \frac{r_1 r_2 r_3}{r_1 r_2 + r_1 r_3 + r_2 r_3}$ **Precondition:** $r_1, r_2, r_3 > 0$ **Constraints:** $f(r_1, r_2, r_3) = f(r_2, r_1, r_3)$, $f(r_1, r_2, r_3) = f(r_3, r_2, r_1)$, $f(r_1, r_2, r_3) = f(r_1, r_3, r_2)$ $f(r_1, r_2, r_3) \leq r_1 \ \wedge \ f(r_1, r_2, r_3) \leq r_2 \ \wedge \ f(r_1, r_2, r_3) \leq r_3$ $f(r_1, r_2, r_3) > 0$	

R ::= R + R | R - R | R * R | R / R | x_1 | x_2 | ... | x_n | $U(-1, 1)$

Fig. 2. The grammar of programs generated by CDSR. x_i is the ith input variable, and $U(-1, 1)$ is an random constant sampled from $[-1, 1]$.

VI. BENCHMARKS

Since there are no well-established benchmarks for SRFC, we adapted several well-known regression benchmarks from [24] (page 8, table 3) and defined formal constraints for them; see Tables II and I. We included also three benchmarks (*gravity*, *res2*, *res3*) from our previous work [1] based on the well-known laws of physics – Newton's law of universal gravitation, and the equivalent resistance of two and three resistors connected in parallel, respectively. Additionally, we included a real-world benchmark *hardware* and its modified version *hardware2*, both with unknown correct model, described in more detail in Section VIII.

Listing 1: The *res2* benchmark in SyGuS format (noise-free version). Each `constraint` defines one property. Constraints representing tests are omitted.

```
(set-logic NRA)
(synth-fun res2 ((r1 Real)(r2 Real)) Real)
(declare-var r1 Real)
(declare-var r2 Real)
(precondition (and (> r1 0.0) (> r2 0.0)))
(constraint (= (res2 r1 r2) (res2 r2 r1)))
(constraint (and (<= (res2 r1 r2) r1)
                 (<= (res2 r1 r2) r2)))
(constraint (> (res2 r1 r2) 0.0))
(check-synth)
```

Each benchmark consists of:

- Preconditions specifying which function's arguments are

TABLE II
CHARACTERISTICS OF THE SRFC BENCHMARKS. $U(a, b)$ STANDS FOR A UNIFORM DISTRIBUTION IN RANGE $[a, b]$, INCLUSIVE FOR a AND b.

Benchmark	Arity	Training set	# constraints
gravity	3	$U(0.0001, 21)$	4
keijzer5	3	$U(-10, 11)$	3
keijzer12	2	$U(-10, 11)$	6
keijzer14	2	$U(-10, 11)$	4
keijzer15	2	$U(-10, 11)$	3
nguyen1	1	$U(-10, 11)$	3
nguyen3	1	$U(-10, 11)$	3
nguyen4	1	$U(-10, 11)$	3
pagie1	2	$U(-10, 11)$	3
res2	2	$U(0.0001, 21)$	3
res3	3	$U(0.0001, 21)$	5
hardware	6	real-world dataset	3
hardware2	6	real-world dataset	3

valid. For example, for $gravity$ these are $m_1, m_2, r > 0$.

- A set of formal constraints we devised based on the properties of solutions. For example, for $gravity$ we selected symmetry with respect to masses $g(m_1, m_2, r) = g(m_2, m_1, r)$, non-negative codomain $g(m_1, m_2, r) \geq 0$, and increasing monotonicity with respect to masses.
- 500 examples generated from the benchmark-dependent uniform distribution specified in Table II — for problems with a known target model (all except *hardware* benchmarks). All examples are required to meet the precondition. Examples are generated once per each benchmark, and in every algorithm's run they are randomly partitioned into a training set (300 examples), a validation set (75), and a test set (125).

There are two variants of each benchmark (with the exception of the *hardware* benchmarks): in the noise-free benchmarks, examples are generated directly from the ground truth formulas; in the noisy ones ('N' appended to name), inputs and outputs generated for the noise-free benchmark are distorted by a multiplicative Gaussian noise with $\mu = 1$ and $\sigma = 0.01$ (i.e., $x'_j = x_j \cdot \mathcal{N}(1, 0.01)$).

The benchmarks are represented in the SyGuS format [25], which we slightly extended to explicitly delineate preconditions; see example in Listing 1.

VII. EXPERIMENTS

We examine the efficiency and generalization power of CDSR variants in different configurations. CDSR inherits most of its hyperparameters and components from GP, and adds several of its own. The hyperparameters of CDSR that remain constant throughout experiments are shown in Table III. All setups use the most common GP search operators, i.e., subtree-swapping crossover and a mutation operator that replaces a randomly selected subtree with a new randomly generated subtree.

The instruction set of CDSR contains standard arithmetic operators (+, -, *, /), and the formal grammar is presented in Figure 2. Division by 0 is not tolerated and is penalized with the worst possible fitness ($+\infty$).

TABLE III
THE SETTINGS OF HYPERPARAMETERS OF CDSR.

Parameter	*Value*
Number of runs	50
Population size	1000
Maximum number of generations	∞
Maximum runtime in seconds	1800
Verification threshold α	$\{0.75, 1\}$
Solver timeout in seconds	3
Probability of mutation	0.5
Probability of crossover	0.5
Test passing threshold (relative MAE)	0.05
Tournament size	7
Maximum height of initial programs	4
Maximum height of trees inserted by mutation	4
Maximum height of programs in population	12
Validation set improvement window (generations)	25

TABLE IV
THE SETTINGS OF HYPERPARAMETERS OF CONVENTIONAL REGRESSION ALGORITHMS USED IN THE EXPERIMENT.

Algorithm	**Parameter: values**
AdaBoost Regressor	`n_estimators`: $10, 100, 1000$ `learning_rate`: $0.01, 0.1, 1, 10$
GradientBoosting Regressor	`n_estimators`: $10, 100, 1000$ `min_weight_fraction_leaf`: $0.0, 0.25, 0.5$ `max_features`: sqrt, log2, None
KernelRidge	`kernel`: linear, poly, rbf, sigmoid `alpha`: $1e{-}4, 1e{-}2, 0.1, 1$ `gamma`: $0.01, 0.1, 1, 10$
LassoLARS	`alpha`: $1e{-}04, 0.001, 0.01, 0.1, 1$
LinearRegression	defaults
MLPRegressor	`activation`:logistic, tanh, relu `solver`: lbfgs, adam, sgd `learning_rate`: constant, invscaling, adaptive
RandomForest Regressor	`n_estimators`:$10, 100, 1000$ `min_weight_fraction_leaf`: $0.0, 0.25, 0.5$ `max_features`: sqrt, log2, None
SGDRegressor	`alpha`:$1e{-}06, 1e{-}04, 0.01, 1$ `penalty`: l2, l1, elasticnet
LinearSVR	`C`: $1e{-}06, 1e{-}04, 0.1, 1$ `loss`: epsilon_insensitive, squared_epsilon_insensitive
XGBoost	`n_estimators`:$10, 50, 100, 250, 500, 1000$ `learning_rate`: $1e{-}4, 0.01, 0.05, 0.1, 0.2$ `gamma`: $0, 0.1, 0.2, 0.3, 0.4$ `max_depth`: 6 `subsample`: $0.5, 0.75, 1$

Our implementation[1] uses the Z3 [26], [27] SMT solver.

A. Impact of verification threshold α

In the following, we compare CDSR and CDSR$_\text{p}$ in combination with two considered selection methods: tournament and ϵ-lexicase, with various settings of hyperparameters. For CDSR$_\text{p}$, we use $w_p = 1$ and $w_p = 5$. We also employ regular GP as a baseline, which is configured in the same way as CDSR but no formal verification is conducted during runtime and constraints are effectively ignored.

We assess first the impact of α, the ratio of tests that a program must pass in order to be submitted to verification.

[1] https://github.com/iwob/CDGP

TABLE V
SUCCESS RATES FOR ALL BENCHMARKS (N=NOISE).

	GP		CDSR		CDSR$_p$			
	Tour	*Lex*	*Tour*	*Lex*	*Tour*		*Lex*	
w_p					1	5	1	5
gravity	0.00	**0.14**	0.02	0.08	0.00	0.00	0.02	0.06
keijzer12	0.02	0.04	0.04	0.10	0.00	0.00	0.04	**0.12**
keijzer14	0.66	0.02	0.58	0.00	0.64	0.70	0.20	**0.72**
keijzer15	0.00	0.00	0.00	0.06	0.00	0.02	0.14	**0.28**
keijzer5	0.02	0.04	0.02	0.04	0.02	0.00	0.02	**0.08**
nguyen1	0.80	0.12	**0.90**	0.42	**0.90**	0.86	0.62	0.64
nguyen3	0.18	0.10	**0.40**	0.12	0.36	0.26	0.22	0.38
nguyen4	0.10	0.10	0.12	0.12	0.24	0.24	0.34	**0.50**
pagie1	0.28	0.12	0.32	0.10	0.42	0.24	0.40	**0.58**
res2	0.78	0.70	**0.90**	0.84	0.86	0.80	0.66	0.76
res3	0.06	**0.40**	0.10	**0.40**	0.00	0.00	0.28	0.22
gravityN	0.00	**0.06**	0.00	0.00	0.00	0.00	0.00	0.04
keijzer12N	**0.02**	**0.02**	**0.02**	0.00	0.00	0.00	0.00	**0.02**
keijzer14N	0.66	0.00	0.50	0.00	0.62	0.64	0.26	**0.74**
keijzer15N	0.00	0.06	0.02	0.02	0.00	0.02	0.14	**0.22**
keijzer5N	0.00	0.06	0.00	0.06	0.02	0.02	**0.10**	0.06
nguyen1N	0.26	0.40	0.36	0.38	0.40	0.50	**0.70**	0.52
nguyen3N	0.12	0.20	0.22	0.20	0.24	0.28	0.24	**0.36**
nguyen4N	0.22	0.22	0.28	0.18	0.28	0.24	**0.42**	0.40
pagie1N	0.32	0.00	0.32	0.08	0.44	0.18	0.44	**0.74**
res2N	0.26	0.16	0.54	0.66	0.70	0.42	**0.72**	0.58
res3N	0.00	0.18	0.02	**0.30**	0.00	0.00	0.18	0.24
hardware	0.34	0.66	0.34	0.42	0.74	0.76	0.76	**0.82**
hardware2	0.46	0.66	0.48	0.82	0.66	0.48	0.84	**0.92**
Mean	0.23	0.19	0.27	0.23	0.31	0.28	0.32	0.42
Rank	5.96	5.15	4.71	4.90	4.60	5.00	3.62	2.06

TABLE VII
THE AVERAGE RATIO OF SATISFIED CONSTRAINTS, AGGREGATED ACROSS ALL BENCHMARKS.

	CDSR		CDSR$_p$			
	Tour	*Lex*	*Tour*		*Lex*	
w_p			1	5	1	5
α=0.75	0.40	0.35	0.46	0.44	0.49	**0.61**
α=1.0	0.38	0.34	0.46	0.44	0.49	0.60

Table VII presents the average ratios of satisfied constraints, aggregated across all benchmarks, for $\alpha = 0.75$ and $\alpha = 1.0$. Note that this metric is different from the success rate, where 'success' is identified with *all* constraints being satisfied. We can observe that α did not make a big difference in the number of constraints satisfied by the algorithms. However, the MSE on test set was overall much better for $\alpha = 1.0$, and thus we continue our analysis only for that setting, which requires a program to pass all tests in order to be submitted to verification.

B. Comparison of CDSR variants

We evaluate the variants of CDSR on the success rate (Table V), the ratio of satisfied constraints to the total number of constraints of a given benchmark ('qualitative' generalization, Table VI), and on the median MSE on test set ('quantitative' generalization, Table VIII). Overall, the highest success rate and satisfiability ratio, both on benchmarks with and without noise, was obtained by CDSR$_p$/Lex/w_p=5. This proves that the additional focus on constraints was effective. This was, however, achieved at the cost of a significantly worse MSE, especially for CDSR$_p$ with lexicase selection.

TABLE VI
THE AVERAGE RATIO OF SATISFIED CONSTRAINTS FOR ALL BENCHMARKS (N=NOISE).

	GP		CDSR		CDSR$_p$			
	Tour	*Lex*	*Tour*	*Lex*	*Tour*		*Lex*	
w_p					1	5	1	5
gravity	0.38	0.48	0.40	0.46	0.41	0.39	0.48	**0.53**
keijzer12	0.07	0.06	0.10	0.12	0.46	0.40	0.43	**0.56**
keijzer14	0.68	0.05	0.62	0.03	0.67	**0.73**	0.23	**0.73**
keijzer15	0.00	0.00	0.03	0.07	0.03	0.03	0.17	**0.36**
keijzer5	0.27	0.19	0.23	0.19	0.23	0.16	0.20	**0.32**
nguyen1	0.80	0.12	**0.90**	0.42	**0.90**	0.86	0.66	0.69
nguyen3	0.18	0.13	0.40	0.12	0.37	0.26	0.33	**0.50**
nguyen4	0.10	0.11	0.12	0.12	0.24	0.25	0.35	**0.56**
pagie1	0.47	0.46	0.48	0.40	0.67	0.56	0.73	**0.84**
res2	0.81	0.79	**0.91**	0.88	0.87	0.86	0.77	0.83
res3	0.12	**0.48**	0.10	0.45	0.10	0.14	0.46	0.42
gravityN	0.38	0.36	0.38	0.38	0.41	0.42	0.47	**0.49**
keijzer12N	0.19	0.19	0.15	0.13	0.38	0.36	0.35	**0.49**
keijzer14N	0.73	0.06	0.56	0.01	0.66	0.70	0.33	**0.77**
keijzer15N	0.05	0.07	0.03	0.03	0.05	0.05	0.17	**0.29**
keijzer5N	0.21	0.16	0.21	0.23	0.24	0.19	0.33	**0.35**
nguyen1N	0.38	0.48	0.45	0.42	0.49	0.59	**0.77**	0.68
nguyen3N	0.23	0.32	0.39	0.33	0.35	0.45	0.37	**0.52**
nguyen4N	0.32	0.24	0.31	0.22	0.35	0.30	**0.50**	0.49
pagie1N	0.51	0.35	0.52	0.37	0.74	0.61	0.74	**0.88**
res2N	0.47	0.43	0.69	0.77	**0.81**	0.62	**0.81**	0.78
res3N	0.02	0.30	0.05	0.38	0.10	0.10	0.33	**0.50**
hardware	0.55	0.79	0.56	0.69	0.83	0.85	0.87	**0.91**
hardware2	0.58	0.81	0.59	0.86	0.76	0.61	0.89	**0.96**
Mean	0.35	0.31	0.38	0.34	0.46	0.44	0.49	0.60
Rank	6.02	6.17	5.17	5.69	3.75	4.42	3.15	1.65

TABLE VIII
THE MEDIAN MSE ON TEST SET FOR ALL BENCHMARKS (N=NOISE).

	GP		CDSR		CDSR$_p$			
	Tour	*Lex*	*Tour*	*Lex*	*Tour*		*Lex*	
w_p					1	5	1	5
gravity	2.7E-14	**1.3E-16**	7.3E-15	6.7E-16	3.4E-15	1.1E-14	7.2E-15	7.4E-15
keijzer12	**4.6E-3**	2.4E-2	3.7E-2	3.0E-1	6.5E2	7.0E2	6.0E2	4.7E2
keijzer14	1.2E-1	**2.2E-5**	1.3E-1	7.7E-5	1.3E-1	1.2E-1	9.6E-2	1.3E-1
keijzer15	5.1E2	**2.5E-1**	1.1E2	5.1E-1	1.2E3	1.3E3	2.9E2	1.5E2
keijzer5	1.1E8	2.2E6	4.9E6	1.7E6	1.3E7	**1.4E6**	1.3E8	1.2E8
nguyen1	**2.6E-27**	3.1E-27	2.7E-27	3.0E-27	3.0E-27	2.8E-27	2.4E-2	2.8E-2
nguyen3	**5.2E-23**	2.5E0	6.0E-23	1.8E1	6.2E-23	5.4E-23	6.0E2	1.5E3
nguyen4	7.8E-21	5.1E2	**7.4E-21**	3.5E3	4.4E-2	9.3E-21	1.8E5	4.0E5
pagie1	6.2E-2	**4.4E-3**	1.1E-1	**4.4E-3**	1.1E-1	1.0E-1	1.0E-1	1.2E-1
res2	**2.0E-31**	2.6E-31	2.5E-31	2.6E-31	2.4E-31	2.5E-31	3.1E-31	2.6E-31
res3	1.4E-1	**1.0E-5**	1.4E-1	2.4E-4	6.3E-1	6.9E-1	7.8E-2	2.1E-1
gravityN	1.1E-14	3.3E-15	3.3E-15	**1.2E-15**	2.1E-14	1.3E-14	2.1E-14	1.1E-14
keijzer12N	3.1E4	**2.8E4**	**2.8E4**	2.9E4	3.4E4	3.7E4	3.3E4	2.9E4
keijzer14N	1.2E-1	**8.1E-5**	1.2E-1	1.1E-4	1.1E-1	1.1E-1	1.1E-1	1.2E-1
keijzer15N	3.4E2	**6.6E1**	8.5E2	7.0E1	1.3E3	1.4E3	1.5E2	1.5E2
keijzer5N	2.0E7	**2.4E6**	3.4E6	2.7E6	2.8E6	6.8E6	7.5E7	8.2E6
nguyen1N	**2.4E2**	2.7E2	2.9E2	**2.4E2**	**2.4E2**	2.5E2	2.7E2	2.8E2
nguyen3N	4.9E6	5.3E6	4.9E6	**4.3E6**	4.9E6	5.0E6	5.2E6	5.5E6
nguyen4N	**5.8E8**	6.4E8	6.4E8	7.4E8	6.1E8	6.3E8	7.0E8	6.1E8
pagie1N	1.1E-1	**5.0E-3**	1.1E-1	6.7E-3	1.1E-1	8.6E-2	8.6E-2	1.4E-1
res2N	4.1E-3	**3.9E-3**	**3.9E-3**	4.1E-3	4.1E-3	**3.9E-3**	4.1E-3	4.2E-3
res3N	2.2E-1	2.2E-3	1.3E-1	**1.5E-3**	6.5E-1	5.9E-1	2.7E-1	1.5E-1
hardware	4.7E3	3.8E3	**2.8E3**	3.9E3	4.4E3	4.6E3	4.6E3	5.6E3
hardware2	3.4E3	**2.8E3**	3.7E3	3.1E3	5.4E3	5.1E3	4.3E3	3.5E3
Rank	4.12	2.73	4.04	3.04	5.19	5.06	5.69	6.12

For CDSR$_p$, lexicase achieves better constraints satisfiability than the corresponding tournament selection variants. However, for test-set MSE, tournament variants of CDSR$_p$ are better than the lexicase ones. Interestingly, the situation is reversed for 'vanilla' CDSR, where lexicase leads to lower MSE on the test set, while the fraction of satisfied constraints is higher for tournament selection. Unsurprisingly, GP fares the worst in terms of satisfaction of constraints, although it

This article has been accepted for publication in IEEE Transactions on Evolutionary Computation. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TEVC.2022.3205286

must be noted that its test-set MSE is competitive and, in the case of lexicase selection, outperforms the CDSR variants.

Statistical analysis with the Friedman test and Nemenyi post-hoc test [28] showed that CDSR$_p$/Lex/w_p=5 satisfies significantly more constraints (p-values $<$ 0.0023) than all other configurations except CDSR$_p$/Tour/w_p=1 and CDSR$_p$/Lex/w_p=1, the latter of which satisfies significantly more constraints (p-value $<$ 0.0078) than the worst CDSR configuration, i.e., CDSR/Lex, and all GP configurations (p-values $<$ 0.002). For test-set MSE, CDSR/Lex was better (p-values $<$ 0.002) than both CDSR$_p$/Lex variants, and GP/Lex was better than all CDSR$_p$/Lex (p-values $=$ 0.001) and CDSR$_p$/Tour variants (p-values $<$ 0.022).

C. Comparison with conventional regression algorithms

In this section, we compare CDSR with several popular constraint-agnostic regression algorithms. For such algorithms, the only source of information about a regression problem is the training set of examples. Therefore, if a model they produce meets the constraints, this can be only due to some inherent biases of a given algorithm, its (usually implicit) capability of "inferring" a constraint from the training sample, or simply pure chance.

The regression algorithms, together with a grid of hyperparameters they were tested on, are presented in Table IV. We followed [29] in the selection of algorithms and their hyperparameters and used the open-source framework proposed there[2]. We assess each setting of hyperparameters with the average prediction error obtained from 5-fold cross-validation. To measure the performance of the best parametrization of a given algorithm (i.e., that with the lowest cross-validation error), we test it on the test set. This entire procedure is repeated 10 times for different partitioning of data into training set and test set, and the average test error is the final measure of quality of the regression algorithm.

Because some of the regression algorithms considered here produce models that involve operations not supported by the NRA logic in contemporary SMT-solvers (e.g., logarithms or trigonometric functions), we cannot apply to them formal verification to determine whether a model satisfies a given constraint or not. Therefore, we apply *approximate verification*, in which for each constraint we check whether it is satisfied for a number of points in the grid of benchmark's inputs domain (Table II). We use the following heuristic to determine the number of points for each benchmark: a grid of 41 equally spaced points per dimension for arity 1 (41 points in total), 11 points for arity 2 (121 points in total), and 7 points for arity 3 (343 points in total). For the *hardware* benchmarks with arity 6 there are 1024 points in total. This evaluation is only approximate, and false positive errors (i.e., constraints incorrectly claimed as satisfied) can occur; for CDSR runs, we compared the results of the approximate verifier with those of the formal verification, and the discrepancy was low.

TABLE IX
THE AVERAGE RATIO OF SATISFIED CONSTRAINTS. THE DEEPER THE SHADING, THE BETTER; BEST PER-BENCHMARK VALUE IN BOLD.

	AdaBoost	GradientBoosting	KernelRidge	LassoLARS	LinearRegression	LinearSVR	MLPRegressor	RandomForest	SGDRegressor	XGBoost
gravity	**0.50**	**0.50**	0.00	**0.50**	**0.50**	**0.50**	0.00	**0.50**	**0.50**	**0.50**
keijzer12	0.17	0.00	**0.33**	0.17	0.17	0.17	0.00	0.00	0.17	0.17
keijzer14	0.75	0.50	**1.00**	**1.00**	0.25	0.50	0.25	0.75	0.25	0.75
keijzer15	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**
keijzer5	0.33	**0.67**	**0.67**	**0.67**	0.33	0.33	0.33	**0.67**	**0.67**	0.33
nguyen1	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
nguyen3	**1.00**	**1.00**	0.00	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
nguyen4	**1.00**	**1.00**	0.67	0.67	0.67	0.67	**1.00**	**1.00**	0.67	**1.00**
pagie1	**0.67**	0.33	**0.67**	0.00	0.00	0.33	0.00	**0.67**	0.33	**0.67**
res2	**0.67**	**0.67**	**0.67**	0.33	0.33	0.33	0.33	**0.67**	0.33	**0.67**
res3	**0.80**	**0.80**	**0.80**	0.20	0.20	0.20	0.00	**0.80**	0.20	**0.80**
gravityN	**0.50**	**0.50**	**0.50**	**0.50**	0.25	**0.50**	0.00	**0.50**	**0.50**	**0.50**
keijzer12N	**0.17**	0.00	**0.17**	**0.17**	**0.17**	**0.17**	0.00	0.00	**0.17**	**0.17**
keijzer14N	0.75	0.50	**1.00**	**1.00**	0.25	0.50	0.25	0.75	0.25	0.75
keijzer15N	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**	**0.33**
keijzer5N	0.33	**0.67**	**0.67**	**0.67**	0.33	0.33	0.33	**0.67**	**0.67**	0.33
nguyen1N	**1.00**	**1.00**	0.67	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
nguyen3N	**1.00**	**1.00**	0.33	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
nguyen4N	**1.00**	**1.00**	0.67	0.67	0.67	0.67	**1.00**	**1.00**	0.67	**1.00**
pagie1N	0.67	0.33	**1.00**	0.33	0.00	0.67	0.33	0.33	0.00	0.33
res2N	**0.67**	**0.67**	**0.67**	0.33	0.33	0.33	0.33	**0.67**	0.33	**0.67**
res3N	**0.80**	**0.80**	**0.80**	0.20	0.20	0.20	0.00	**0.80**	0.20	**0.80**
hardware	**1.00**	**1.00**	0.00	**1.00**	0.67	0.67	0.33	**1.00**	0.67	**1.00**
hardware2	**1.00**	**1.00**	0.33	**1.00**	0.67	0.67	0.67	**1.00**	**1.00**	**1.00**
Mean	0.69	0.65	0.55	0.59	0.44	0.52	0.41	0.69	0.51	0.67
Rank	4.27	4.75	5.25	5.27	7.00	6.19	7.38	4.40	6.08	4.42

We evaluate the algorithms on the ratio of satisfied constraints (Table IX) and the MSE on the test set (Table X). As expected, the benchmarks vary in difficulty. The best MSE is achieved by KernelRidge, which has also a decent rate of satisfied constraints. In contrast, the second algorithm with the best MSE, i.e., MLP (multilayer perceptron), achieved the lowest ratio of satisfied constraints. The presence of noise, at least at the assumed magnitude (normal distribution with $\sigma = 1\%$ of the value which is being distorted), does not have much effect on the number of satisfied constraints. The success rate (not presented here for brevity) was either 0% or 100%, and the latter case was always co-occurring with 100% of satisfied properties.

Friedman's test for multiple achievements of multiple subjects with the Nemenyi post-hoc test [28] indicates that AdaBoost, RandomForest, and XGBoost satisfy significantly more constraints (p-value $<$ 0.025) than MLP. As for the MSE on the test set, there were several statistically significant differences – LassoLars, LinearRegression, LinearSVR, and SGD were dominated (p-values $<$ 0.034) by all other approaches with the exception of AdaBoost, which was significantly better (p-values $<$ 0.04) only than LassoLars and LinearSVR. Only KernelRidge managed to be significantly better than AdaBoost (p-value 0.03).

In Table XI, we juxtapose the above best constraint-agnostic algorithms, RandomForest (the best ratio of satisfied constraints and success rate) and KernelRidge (the best MSE on the test set), with the two best-performing CDSR configurations: CDSR$_p$/Lex/α=1.0/w_p=5 (the best ratio; CDSR$_{sat}$ in the following) and CDSR/Lex/α=1.0 (the best MSE; CDSR$_{MSE}$ in the following). CDSR$_{sat}$ boasts the best average success rate of 42% and ranks first on this metric calculated per benchmark.

[2] https://github.com/EpistasisLab/srbench

TABLE X
THE MEDIAN MSE ON TEST SET (N=NOISE). DARKER SHADING MARKS BETTER VALUES; BEST VALUE IN BOLD.

	AdaBoost	GradientBoosting	KernelRidge	LassoLars	LinearRegression	LinearSVR	MLPRegressor	RandomForest	SGDRegressor	XGBoost
gravity	**2.6E-14**	5.5E-14	5.5E-14	5.6E-14	5.5E-14	5.6E-14	5.5E-14	5.6E-14	5.6E-14	4.2E-14
keijzer12	4.6E5	4.2E3	**3.9E2**	9.0E6	9.1E6	9.1E6	1.9E3	3.6E3	9.1E6	1.6E3
keijzer14	1.1E-2	2.5E-2	**2.6E-4**	6.4E-2	6.4E-2	6.3E-2	3.9E-3	1.3E-3	6.4E-2	1.2E-3
keijzer15	1.3E3	2.1E2	**1.4E-10**	1.0E4	1.0E4	1.0E4	8.2E0	5.2E2	1.0E4	1.3E2
keijzer5	7.5E4	1.1E7	2.8E9	1.1E7	2.3E7	**4.3E4**	1.2E7	1.4E7	8.3E6	2.7E8
nguyen1	6.5E2	5.2E1	**2.1E-9**	5.0E4	5.0E4	5.1E4	5.9E1	2.6E1	5.0E4	1.5E1
nguyen3	2.8E7	1.3E6	**2.1E3**	6.0E8	6.0E8	6.0E8	7.7E4	1.3E6	6.0E8	7.4E5
nguyen4	5.1E9	1.6E8	3.2E9	7.1E10	7.1E10	7.1E10	**1.3E7**	3.1E7	7.2E10	3.6E7
pagie1	1.3E-2	**1.2E-3**	3.0E-3	1.4E-1	1.4E-1	1.4E-1	1.3E-3	1.1E-2	1.4E-1	2.7E-3
res2	1.0E-1	1.8E-2	**3.1E-5**	1.4E0	1.4E0	1.4E0	7.2E-4	1.6E-2	1.4E0	8.2E-3
res3	1.5E-1	2.4E-2	**3.1E-3**	6.4E-1	6.4E-1	6.4E-1	4.1E-3	1.7E-2	6.4E-1	8.9E-3
gravityN	2.1E-14	1.4E-14	1.2E-14	1.4E-14	1.3E-14	1.4E-14	1.4E-14	1.4E-14	1.4E-14	**9.9E-15**
keijzer12N	4.8E5	3.8E4	2.4E4	8.9E6	8.8E6	8.8E6	**2.3E4**	3.2E4	8.9E6	3.5E4
keijzer14N	1.1E-1	7.1E-2	**5.9E-4**	2.1E-1	2.1E-1	2.1E-1	6.0E-3	2.5E-2	2.1E-1	1.2E-2
keijzer15N	1.4E3	2.8E2	**6.7E1**	1.1E4	1.1E4	1.1E4	8.5E1	7.0E2	1.1E4	2.2E2
keijzer5N	**4.5E6**	1.4E7	4.3E7	1.4E7	2.6E7	**4.5E6**	9.8E6	1.9E7	1.6E7	1.1E8
nguyen1N	9.8E2	3.9E2	2.6E2	4.0E4	4.0E4	4.0E4	**2.5E2**	3.2E2	4.0E4	3.3E2
nguyen3N	1.4E7	1.0E7	**7.5E6**	5.9E8	5.9E8	5.9E8	7.7E6	1.1E7	5.9E8	1.0E7
nguyen4N	4.9E9	8.9E8	4.4E8	1.1E11	1.1E11	1.1E11	**3.7E8**	7.8E8	1.2E11	1.0E9
pagie1N	2.5E-2	7.6E-3	3.0E-3	1.8E-1	1.8E-1	1.9E-1	**1.9E-3**	2.2E-2	1.8E-1	7.7E-3
res2N	1.3E-1	2.0E-2	5.5E-3	1.0E0	9.9E-1	1.0E0	**4.0E-3**	2.1E-2	1.0E0	1.1E-2
res3N	1.7E-1	2.0E-2	**4.8E-3**	7.8E-1	7.8E-1	7.8E-1	1.3E-2	3.4E-2	7.8E-1	1.3E-2
hardware	6.8E3	**2.3E3**	3.3E3	1.0E4	7.8E3	1.3E4	4.2E3	5.1E3	8.2E3	3.7E3
hardware2	**2.4E3**	1.3E4	1.0E4	1.8E4	1.5E4	1.6E4	1.2E4	1.5E4	1.4E4	1.4E4
Rank	5.35	4.21	2.48	8.10	7.81	7.92	2.65	4.75	8.12	3.60

TABLE XI
COMPARISON OF THE BEST CONSTRAINT-AGNOSTIC REGRESSION ALGORITHMS WITH THE BEST CONFIGURATIONS OF CDSR.

	Kernel-Ridge	Random-Forest	CDSR_{MSE}	CDSR_{sat}
(avg. rank) median tests set MSE	2.25	3.54	**1.38**	2.83
(avg. rank) satisfiability ratio	2.33	**2.02**	3.44	2.21
(avg. rank) success rate	3.21	2.67	2.40	**1.73**
(avg.) satisfiability ratio	0.55	**0.69**	0.34	0.60
(avg.) success rate	0.17	0.33	0.23	**0.42**
(avg.) runtime (s)	**38.36**	179.25	987.75	1733.75

Its success rate per benchmark is also much more evenly distributed, in contrast to the constraint-agnostic algorithms, which either have a success rate of 0% or 100%. To ensure fair comparison, all methods are verified stochastically, i.e., by querying models on inputs sampled uniformly from the domains listed in Table II and checking whether the constraints are met. Notice that this does not *guarantee* meeting the constraints, but only indicates that no constraint violation has been observed in the process.

Surprisingly, both constraint-agnostic methods managed to satisfy on average a greater fraction of constraints than the best variants of CDSR. This may, however, originate in the specific choice of constraints used in our benchmark suite. To investigate that aspect, in Table XII we present the satisfiability ratio for each constraint in each benchmark, and label them by type (for brevity, we present these only for the noise-free benchmarks; the results for the noisy benchmarks are very similar). Clearly, some constraints are easy to satisfy for CDSR_{sat} and hard for constraint-agnostic algorithms, for example the equality constraint for *keijzer12* ($x = y = 0 \implies f(x, y) = 0$), monotonicity constraints for *gravity* ($\Delta f(m_i) > 0$), and the output bound for *res2* ($f(r_1, r_2) \leq r_1 \ \wedge \ f(r_1, r_2) \leq r_2$). On the other hand, CDSR is worse on many other constraints. A clear pattern can be observed in terms of types of constraints: while the conventional algorithms outperform CDSR on symmetry constraints and bound constraints, CDSR is unmatched on monotonicity and equality constraints. The prevalence of the former two types of constraints in our suite (34 vs. 12 constraints) is indeed the cause of better average satisfiability of constraint-agnostic methods, reported in Table XI.

In Table XI, CDSR_{MSE} achieves the best average rank on MSE, and in general most of the lowest MSE scores were obtained by either of the CDSR variants. This result is surprising: it is natural to expect CDSR, and especially CDSR_{sat} (which uses CDSR_{p}), to trade its MSE in favor of meeting the constraints. A possible explanation is that the search space for CDSR, restricted by the constraints, is effectively smaller than for the constraint-agnostic methods, which reduces the risk of overfitting and improves generalization. However, the properties of representations used by particular methods (regression trees and kernels vs. explicit mathematical formulas used by CDSR) can play a role here too.

Statistical analysis with the Friedman test and Nemenyi post-hoc test [28] reveals that all algorithms satisfy significantly more constraints than CDSR_{MSE} (p-values < 0.017), CDSR_{MSE} achieves significantly lower MSE than RandomForest and CDSR_{sat} (p-values 0.001), and KernelRidge had significantly lower MSE than RandomForest (p-value < 0.003).

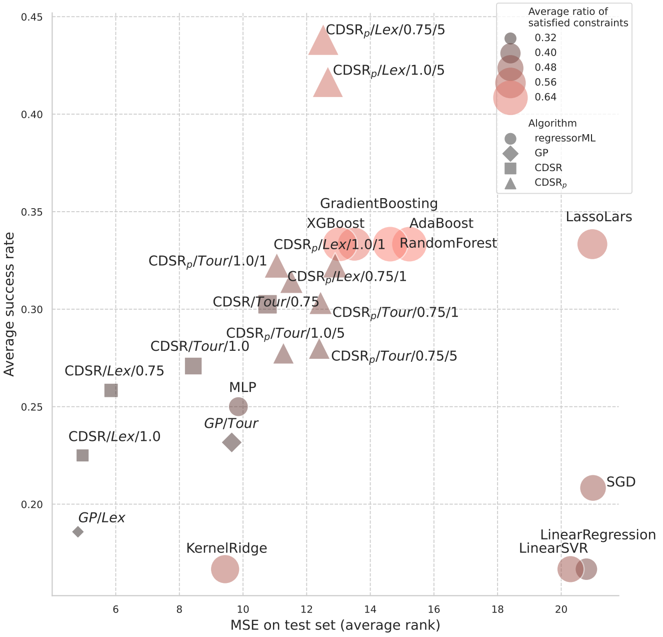

Fig. 3. Comparison in terms of rank on test-set MSE (minimized), success rate (maximized), and satisfaction of constraints (marker size and hue).

The conventional regression algorithms are much faster than CDSR, which is slowed down by the burden of formal verification. The fraction of the runtime spent in the SMT solver to the total runtime of CDSR is on average 0.1 for $\alpha = 1$ and 0.3 for $\alpha = 0.75$, while for CDSR$_\text{p}$ it is about 0.9 for tournament selection and 0.3 for lexicase selection, regardless of the value of α – lexicase selection is more expensive computationally, and thus the calls to the solver take a smaller fraction of total time than for tournament selection. However, optimizing CDSR for maximum speed was not our priority, and relatively easy improvements (e.g., approximate verification of constraints in CDSR$_\text{p}$ instead of full verification with the SMT solver) could significantly reduce its runtime.

Figure 3 summarizes the results visually in terms of generalization (test-set MSE, horizontal axis, minimized rank) and the average success rate (vertical axis, maximized). The Pareto front spanning these two metrics is formed mostly by configurations of CDSR, stretching from CDSR/Lex/α=1.0 on one end to CDSR$_\text{p}$/Lex/α=0.75/w_p=5 on the other. The only nondominated reference method is GP/Lex, but it is located at the very end of the front and achieves a very low success rate. Notably, many of the reference methods align in two horizontal bands, which is due to the fact that they meet the constraints by mere chance and do that on a systematic basis. The percentage of satisfied constraints (reflected by marker size) significantly correlates with the average success rate, though less so for the constraint-agnostic methods.

VIII. Real-world case study

Even though some of the benchmarks used here can be viewed as realistic in concerning the known laws of physics and involving noise, most of them do not involve real data. To corroborate our claims in a real-world scenario, our suite of benchmarks includes also the Computer Hardware Data Set problem from the UCI ML repository [30], for which the underlying true model of the sought dependency is not known. The task is to estimate the relative performance score of a CPU based on six integer-valued parameters (the largest number in our suite): machine cycle length, the size of cache memory, the minimum and maximum admissible size of memory, and the minimum and maximum number of channels. The dataset contains 209 examples, which we split randomly into a training set (157 examples) and a test set (52 examples).

We devise three constraints for this task that seem plausible according to the domain knowledge: (1) the performance score must be non-negative (range constraint), (2) the score cannot deteriorate when decreasing the machine cycle length and (3)

TABLE XII
Satisfiability ratio of individual constraints for the noise-free benchmarks. Legend: (E) equality , (C) constant output bound , (V) variable output bound , (S) symmetry w.r.t. arguments , (M) monotonicity . The order of constraints is the same as in Table I.

	KernelRidge	RandomForest	CDSR$_\text{MSE}$	CDSR$_\text{sat}$
(E) keijzer12-2			0.10	**0.62**
(E) keijzer15-0			0.06	**0.34**
(E) keijzer5-0			**0.14**	**0.14**
(C) gravity-1		**1.00**	0.64	0.96
(C) keijzer14-0	**1.00**	**1.00**	0.06	0.74
(C) keijzer14-1	**1.00**	**1.00**		0.74
(C) keijzer15-1		**1.00**	0.06	0.38
(C) keijzer15-2	**1.00**		0.08	0.36
(C) keijzer5-1	**1.00**	**1.00**	0.32	0.58
(C) keijzer5-2	**1.00**	**1.00**	0.10	0.24
(C) nguyen1-0	**1.00**	**1.00**	0.42	0.72
(C) nguyen1-1	**1.00**	**1.00**	0.42	0.64
(C) nguyen3-0		**1.00**	0.12	0.56
(C) nguyen3-1		**1.00**	0.12	0.40
(C) nguyen4-0	**1.00**	**1.00**	0.12	0.58
(C) nguyen4-1	**1.00**	**1.00**	0.12	0.52
(C) pagie1-0	**1.00**	**1.00**	0.70	0.98
(C) pagie1-1		**1.00**	0.40	0.96
(C) res2-2	**1.00**	**1.00**	0.94	0.92
(C) res3-4	**1.00**	**1.00**	0.62	0.74
(C) hardware-0		**1.00**	0.64	0.92
(C) hardware2-0		**1.00**	0.82	0.96
(V) keijzer12-0	**1.00**		0.12	0.78
(V) keijzer12-1	**1.00**		0.12	0.74
(V) keijzer14-2	**1.00**	**1.00**	0.06	0.74
(V) nguyen1-2	**1.00**	**1.00**	0.42	0.72
(V) nguyen3-2		**1.00**	0.12	0.54
(V) nguyen4-2		**1.00**	0.12	0.58
(V) res2-1			**0.86**	0.76
(V) res3-3			**0.42**	0.24
(S) gravity-0		**1.00**	0.40	0.94
(S) keijzer14-3	**1.00**			0.72
(S) pagie1-2	**1.00**		0.10	0.58
(S) res2-0	**1.00**	**1.00**	0.84	0.82
(S) res3-0	**1.00**	**1.00**	0.40	0.38
(S) res3-1	**1.00**	**1.00**	0.40	0.36
(S) res3-2	**1.00**	**1.00**	0.40	0.36
(M) gravity-2			**0.40**	0.10
(M) gravity-3			**0.38**	0.10
(M) keijzer12-3			0.12	**0.78**
(M) keijzer12-4			0.12	**0.22**
(M) keijzer12-5			0.12	**0.20**
(M) hardware-1		**1.00**	0.64	0.86
(M) hardware-2		**1.00**	0.80	0.94
(M) hardware2-1	**1.00**	**1.00**	0.84	0.96
(M) hardware2-2		**1.00**	0.92	0.96
Rank	2.59	2.05	2.98	2.38

when increasing the cache size (monotonicity).

Applying all CDSR variants and the reference regression methods to this problem reveals the superiority of GradientBoosting in terms of the training-set MSE (median of 9.6×10^1). However, the test-set MSE of this model is much worse (2.3×10^3), suggesting heavy overfitting, and many configurations of CDSR catch up with it: the median MSE ranges from 2.8 (CDSR/Tour/α=1.0) to 5.6×10^3 (CDSR$_\mathrm{p}$/Lex/α=1.0/w_p=1). In terms of constraints, GradientBoosting and a few other reference models (AdaBoost, LassoLars, RandomForrest and XGBoost) achieve 100% ratio of satisfied constraints on average, while CDSR configurations manage to meet between 25% of constraints (CDSR/Tour/α=0.75) and 89% of constraints (CDSR/Lex/α=1.0/w_p=5); nevertheless, many CDSR runs produced models that met all constraints.

We hypothesize that the main reason why CDSR yields to the reference methods is the relative simplicity of the *hardware* problem – the authors of the benchmark admit that even a simple linear regression model achieves almost perfect correlation with the target score. To illustrate validity of this claim, we consider a transformed variant of this problem, in which we replace the machine cycle variable with its reciprocity, i.e., the frequency of the CPU clock. While both CDSR and reference methods sustain roughly the same ratio of satisfied properties, the test-set MSE of the latter deteriorates substantially, and ranges between 2.4×10^3 and 1.8×10^4. CDSR, to the contrary, maintains roughly the same MSE on the test set as on the training set, ranging from 3.1×10^3 to 5.4×10^3. This suggests that the reference methods meet the required constraints by mere chance, and fail to generalize well when a problem becomes more complex. CDSR, to the contrary, provides both good generalization and good ratio of fulfilled constraints.

The detailed results for the original benchmark (*hardware*) and its modified version (*hardware2*) are included at the bottom of the previously presented tables.

IX. Conclusions

We demonstrated that CDSR achieves better success rate, better MSE on the test set, can synthesize models that satisfy constraints which prove impossible to achieve for the constraint-agnostic approaches, and performs well on real-world problems. This clearly merits involving formal verification, which allows virtually unlimited expressiveness. In this study, we used fairly general constraints; there are arguably many applications where domain knowledge implies constraints that are more complex and precise, which may further limit the search space and improve generalization.

We consider the satisfiability ratio of formal constraints to be an interesting measure of generalization. At a more detailed level, individual properties can be considered a form of multiobjective characterization of generalization. Contrary to the quantitative evaluation of generalization that reflects the point-wise errors, constraints describe behavior that a function exhibits over multiple data points, and thus can be thought of as a 'higher-order generalization', or, as we call it in this study, *qualitative generalization*. Exploring a conceptual framework built on these observations can be an interesting avenue of future research.

Acknowledgments

I.B. acknowledges support from grant 2018/29/N/ST6/01646 funded by the National Science Centre, Poland. K.K. acknowledges support from TAILOR project funded by EU Horizon 2020 research and innovation program under GA no. 952215, and Polish Ministry of Education and Science grant no. 0311/SBAD/0709. Both authors acknowledge support from the statutory funds of Poznan University of Technology.

References

[1] I. Błądek and K. Krawiec, "Solving symbolic regression problems with formal constraints," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '19. New York, NY, USA: ACM, 2019, pp. 977–984. [Online]. Available: http://doi.acm.org/10.1145/3321707.3321743

[2] M. I. Jordan, "Constrained supervised learning," *Journal of Mathematical Psychology*, vol. 36, no. 3, pp. 396–425, 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022249692900297

[3] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design (FMCAD), 2013*. IEEE, Oct. 2013, pp. 1–8. [Online]. Available: http://dx.doi.org/10.1109/fmcad.2013.6679385

[4] S. Barocas, A. D. Selbst, and M. Raghavan, "The hidden assumptions behind counterfactual explanations and principal reasons," in *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, ser. FAT* '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 80–89. [Online]. Available: https://doi.org/10.1145/3351095.3372830

[5] H. Gatignon, "Marketing-mix models," in *Marketing*, ser. Handbooks in Operations Research and Management Science. Elsevier, 1993, vol. 5, pp. 697–732. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0927050705800386

[6] H. Chen, M. Zhang, L. Han, and A. Lim, "Hierarchical marketing mix models with sign constraints," *Journal of Applied Statistics*, vol. 48, no. 13-15, pp. 2944–2960, 2021. [Online]. Available: https://doi.org/10.1080/02664763.2021.1946020

[7] E. Daya, D. Roth, and S. Wintner, "Learning hebrew roots: Machine learning with linguistic constraints," in *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*. ACL, 2004, pp. 357–364. [Online]. Available: https://aclanthology.org/W04-3246/

[8] D. Roth and W. tau Yih, "A linear programming formulation for global inference in natural language tasks," in *HLT-NAACL 2004 Workshop: Eighth Conference on Computational Natural Language Learning (CoNLL-2004)*, H. T. Ng and E. Riloff, Eds. Boston, Massachusetts, USA: Association for Computational Linguistics, 2004, pp. 1–8. [Online]. Available: http://rutcor.rutgers.edu/~amai/aimath04/SpecialSessions/Roth-aimath04.pdf

[9] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," https://smtlib.cs.uiowa.edu/, 2016, accessed: 2021-08-27.

[10] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening, Eds., 2010.

[11] K. Krawiec, I. Błądek, and J. Swan, "Counterexample-driven genetic programming," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '17. New York, NY, USA: ACM, 2017, pp. 953–960. [Online]. Available: http://doi.acm.org/10.1145/3071178.3071224

[12] I. Błądek, K. Krawiec, and J. Swan, "Counterexample-driven genetic programming: Heuristic program synthesis from formal specifications," *Evolutionary Computation*, vol. 26, no. 3, pp. 441–469, Fall 2018.

[13] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications*, C. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, ch. 12, pp. 825–885.

[14] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Communications of the ACM*, vol. 54, no. 9, p. 69–77, Sep. 2011. [Online]. Available: https://doi.org/10.1145/1995376.1995394

[15] L. Spector, "Assessment of problem modality by differential performance of lexicase selection in genetic programming: A preliminary report," in *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, ser. GECCO '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 401–408. [Online]. Available: https://doi.org/10.1145/2330784.2330846

[16] W. La Cava, L. Spector, and K. Danai, "Epsilon-lexicase selection for regression," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, ser. GECCO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 741–748. [Online]. Available: https://doi.org/10.1145/2908812.2908898

[17] C. Johnson, "Genetic programming with fitness based on model checking," in *Proceedings of the 10th European Conference on Genetic Programming*, ser. Lecture Notes in Computer Science, M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, and A. I. Esparcia-Alcázar, Eds., vol. 4445. Valencia, Spain: Springer, 11-13 Apr. 2007, pp. 114–124. [Online]. Available: https://kar.kent.ac.uk/14594/1/Genetic.pdf

[18] P. He, L. Kang, C. G. Johnson, and S. Ying, "Hoare logic-based genetic programming," *SCIENCE CHINA Information Sciences*, vol. 54, no. 3, pp. 623–637, Mar. 2011.

[19] G. Katz and D. Peled, "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, ser. Electronic Proceedings in Theoretical Computer Science, L. Holik and L. Clemente, Eds., vol. 140. Open Publishing Association, 2014, pp. 70–84.

[20] G. Katz and D. A. Peled, "Synthesizing, correcting and improving code, using model checking-based genetic programming," *International Journal on Software Tools for Technology Transfer*, vol. 19, pp. 449–464, 2016.

[21] A. Arcuri and X. Yao, "Co-evolutionary automatic programming for software development," *Information Sciences*, vol. 259, pp. 412–432, Feb. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.ins.2009.12.019

[22] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, 2006, pp. 404–415. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168907

[23] A. Solar-Lezama, C. G. Jones, and R. Bodík, "Sketching concurrent data structures," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, Tucson, AZ, USA, 2008, pp. 136–148. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375599

[24] J. McDermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, and U.-M. O'Reilly, "Genetic programming needs better benchmarks," in *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 791–798. [Online]. Available: https://doi.org/10.1145/2330163.2330273

[25] M. Raghothaman and A. Udupa, "Language to specify syntax-guided synthesis problems," 2014.

[26] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2008, vol. 4963, ch. 24, pp. 337–340. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78800-3_24

[27] ——, "Z3 Theorem Prover: github repository," https://github.com/Z3Prover/z3, 2020.

[28] D. G. Pereira, A. Afonso, and F. M. Medeiros, "Overview of friedman's test and post-hoc analysis," *Communications in Statistics - Simulation and Computation*, vol. 44, no. 10, pp. 2636–2653, 2015. [Online]. Available: https://doi.org/10.1080/03610918.2014.931971

[29] P. Orzechowski, W. La Cava, and J. H. Moore, "Where are we now? a large benchmark study of recent symbolic regression methods," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1183–1190. [Online]. Available: https://doi.org/10.1145/3205455.3205539

[30] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

Iwo Błądek received Ph.D. from Poznan University of Technology, Poland, under the supervision of Krzysztof Krawiec. His scientific interests include: program synthesis, computational and artificial intelligence, genetic programming. More details at www.cs.put.poznan.pl/ibladek.

Krzysztof Krawiec received Ph.D. and Habilitation degrees from Poznan University of Technology, Poland. His work includes semantics in genetic programming, coevolutionary algorithms and test-based problems, and EC for learning game strategies and synthesis of pattern recognition systems. He is an associate editor of Genetic Programming and Evolvable Machines and the author of *Behavioral Program Synthesis with Genetic Programming* (Springer, 2016). More details at www.cs.put.poznan.pl/kkrawiec.