A Data-Driven CHC Solver

He Zhu
Galois, Inc., USA
hezhu@galois.com

Stephen Magill
Galois, Inc., USA
stephen@galois.com

Suresh Jagannathan
Purdue University, USA
suresh@cs.purdue.edu

Abstract

We present a data-driven technique to solve Constrained Horn Clauses (CHCs) that encode verification conditions of programs containing unconstrained loops and recursions. Our CHC solver neither constrains the search space from which a predicate's components are inferred (e.g., by constraining the number of variables or the values of coefficients used to specify an invariant), nor fixes the shape of the predicate itself (e.g., by bounding the number and kind of logical connectives). Instead, our approach is based on a novel machine learning-inspired tool chain that synthesizes CHC solutions in terms of arbitrary Boolean combinations of unrestricted atomic predicates. A CEGAR-based verification loop inside the solver progressively samples representative positive and negative data from recursive CHCs, which is fed to the machine learning tool chain. Our solver is implemented as an LLVM pass in the SeaHorn verification framework and has been used to successfully verify a large number of nontrivial and challenging C programs from the literature and well-known benchmark suites (e.g., SV-COMP).

CCS Concepts • **Software and its engineering → Formal software verification**; **Automated static analysis**;

Keywords Constrained Horn Clauses (CHCs), Invariant Inference, Program Verification, Data-Driven Analysis

ACM Reference Format:
He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A Data-Driven CHC Solver. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3192366.3192416

1 Introduction

Automated program verification typically encodes program control- and data-flow using a number of first-order verification conditions (VCs) with unknown predicates, which correspond to unknown inductive loop invariants and inductive pre- and post-conditions of recursive functions. If adequate inductive invariants are given to interpret each unknown predicate, the problem of checking whether a program satisfies its specification can be efficiently reduced to determining the logical validity of the VCs, and is decidable with modern automated decision procedures for some fragments of first-order logic. However inductive invariant inference is still very challenging, and is even more so in the presence of multiple nested loops and arbitrary recursion; these challenges pose a major impediment towards the use of fully automated verification methods.

Constrained Horn clauses (CHC) are a very popular VC formalism for specifying and verifying safety properties of programs written in a variety of programming languages and styles [6, 13, 15, 18]. Given a set of CHC constraints, with unknown predicate symbols, the goal is to produce an interpretation to solve each unknown predicate symbol as a formula such that each constraint is logically valid. Many powerful CHC solvers have been proposed to automatically and efficiently solve CHC constraints [1, 13, 16, 17, 19, 23–25, 32]; these systems typically rely on sophisticated first-order methods such as interpolation [22] and property-directed reachability [17].

Consider the program[1] shown in Fig. 1. The CHCs for this program, generated by the SeaHorn C program verifier [15], can be expressed as:

$$x = 1 \land y = 0 \rightarrow p(x, y) \tag{1}$$

$$p(x, y) \land x' = x + y \land y' = y + 1 \rightarrow p(x', y') \tag{2}$$

$$p(x, y) \land x' = x + y \land y' = y + 1 \rightarrow x' >= y' \tag{3}$$

$$x = 1 \land y = 0 \rightarrow x >= y \tag{4}$$

Here $p(x, y)$ encodes the unknown loop invariant over the program variables x and y. Constraint (1) ensures that $p(x, y)$ is established when the loop head is initially reached; constraint (2) guarantees that it is inductive with the loop body. Constraint (3) uses the loop invariant to show that the execution of the loop body does not violate the post condition. Constraint (4) corresponds to the trivial case when the loop body is never executed. Notably, a state-of-the-art CHC solver, Spacer [19], times-out when attempting to infer an interpretation of p because it diverges and fails to generalize an inductive invariant from the iteratively generated counterexamples produced by the solver, despite the simplicity of the formulas.

[1]In our examples, we use $*$ to denote a nondeterministically chosen value.

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5698-5/18/06...$15.00
https://doi.org/10.1145/3192366.3192416

```
main(){
 int x,y;
 x=1; y=0;
 while(*){
  x=x+y;
  y++;}
 assert(x>=y);}
```

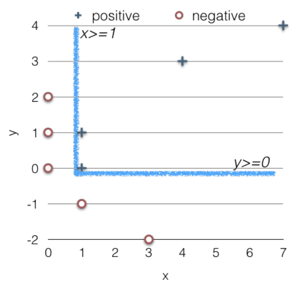


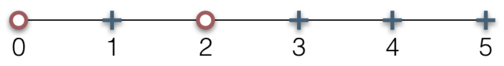
Figure 1. Data-driven Invariant Inference

In recent years, data-driven techniques have gained popularity to complement logic-based VC solvers. These approaches are constrained to find a solution that is consistent with a finite number of concrete samples agnostic of the underlying logics used to encode VCs [10, 11, 29, 35, 37, 38, 40]. Using these methods, one can reason about $p(x, y)$ by sampling x and y, as depicted on the right-side of Fig. 1. Intuitively, the data labeled with + are *positive* samples on the loop head that validate the assertion; data labeled with $\circ$ are *negative* loop head samples that cause the program to trigger an assertion violation. An interpretation of $p(x, y)$ can be learned as a classifier of + and $\circ$ data with off-the-shelf machine learning algorithms. For example, suppose we constrain the interpretation of $p(x, y)$ that we sample to be drawn from the Box abstract interpretation domain (bounds on at most one variable $x \geq d$ where d is a constant). We can consider this domain as playing the role of *features* in machine learning parlance. In this case, standard learning algorithms incorporated within data-driven verification frameworks [29, 35, 37] can be used to interpret the unknown loop invariant as $x \geq 1 \land y \geq 0$. Plugging this invariant as the concrete interpretation of $p(x, y)$ in CHC constraints (1)-(4) proves the program safe.

Despite the promising prospect of learning invariants purely from data as informally captured by this example, there are several key limitations of existing frameworks that make realizing this promise challenging; these challenges are addressed by the techniques described below.

Learning Features. A notable challenge in applying existing learning techniques to program verification is determining the set of atomic predicates (as *features* in machine learning) that should comprise an invariant. In some cases, simply choosing a suitably constrained domain (e.g., the *Box* domain as used in Fig. 1) is sufficient. However, for many program verification tasks, richer domains (e.g., *Polyhedra*) are required; the feature space of these domains have high dimensionality, requiring techniques that go beyond exhaustive enumeration of feature candidates [29]. Additionally, realistic programs will have invariants that are expressed in terms of arbitrary Boolean combinations of these features. We propose a new data-driven framework that allows us to learn invariants as arbitrary Boolean combinations of features learned via linear classification, the shape of whose components are not pre-determined.

Generality vs. Safety. Machine learning algorithms strive to avoid over-fitting concepts to training data by relaxing the requirement that a proposed concept must be fully consistent with given samples. Verification tasks, on the other hand, are centered on safety - their primary goal is to ensure that a hypothesis is correct, even if this comes at the expense of generality. In other words, verification expects any classifier to be perfect, even if that entails over-fitting, while machine learning algorithms expect classifiers to generalize, even if that compromises precision. This tension between the desire for producing general hypotheses, central to machine learning, and the need to ensure these hypotheses produce a perfect classification, central to verification, is a primary obstacle to seamlessly adapting existing learning frameworks to solve general verification problems. To illustrate, consider the example given below:



Here, it might not be possible to draw a linear classifier that can separate all positive samples ($\{1, 3, 4, 5\}$) from all negative samples ($\{0, 2\}$), or even all positives ($\{1, 3, 4, 5\}$) from a single negative ($\{2\}$).

Rather than strictly requiring a data-driven verification framework to always produce a perfect classifier as a learned feature [38], we instead propose a technique that *accepts* precision loss. Our approach exploits a linear classification algorithm that may return a classifier which misclassifies either positive data or negative data or both in order to facilitate generalization. To recover the loss of precision, we develop a new data-driven algorithm, which we call LinearArbitrary that applies a linear classification algorithm *iteratively* on unseparated (misclassified) samples, allowing us to learn a family of linear classifiers that separate all samples correctly in the aggregate; such classifiers are conceptually logically connected with appropriate boolean operators $\land$ and $\lor$.

Combating Over- and Under-fitting. Data-driven analysis techniques that suffer from over- and under-fitting are likely to produce large, often unintuitive, invariants. When input samples are not easily separable because of outliers and corner cases, inference may yield invariants with complex structure that arise because of the need to compensate for the inability to discover a clean and general separator.

Our approach is based on the simple observation that each learned linear classifier defines a hyperplane that separates a *subset* of positive samples from a *subset* of negative samples. Thus, each learned feature has a different *information gain* in splitting the current collection of samples, given in terms of Shannon Entropy [34], a well-established information theoretic concept, which can measure how homogeneous the

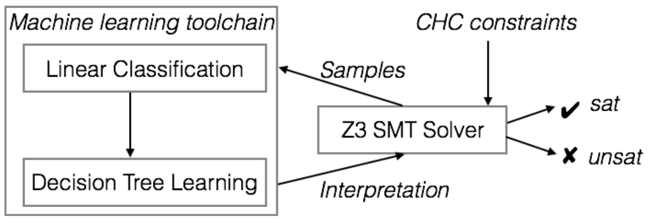


Figure 2. The overall framework of our approach.

samples are after choosing a specific feature as a classifier. This observation motivates the idea of using standard *Decision Trees* to further generalize LinearArbitrary. Decision trees allow us to select high-quality features while dropping unnecessarily complex ones, based on the same data from which feature predicates are learned, by heuristically selecting features that lead to the greatest information gain. A learned decision tree defines a Boolean function over selected feature predicates and can be converted to a first-order logic formula for verification.

Recursive CHC Structure. In the presence of loops and recursive functions, a CHC constraint (such as constraint (2) in the example above) may take the form:

$$\phi \wedge p_1(\overline{T_1}) \wedge p_2(\overline{T_2}) \cdots \wedge p_k(\overline{T_k}) \rightarrow p_{k+1}(\overline{T_{k+1}}) \qquad (*)$$

where each p_i $(1 \leq i \leq k+1)$ is an unknown predicate symbol over a vector of first-order terms $\overline{T_i}$ and ϕ is an unknown-predicate-free formula with respect to some background theory. In this paper, we assume ϕ is expressible using linear arithmetic.

A formula like $(*)$ is a recursive CHC constraint if, for some p_i $(1 \leq i \leq k)$, p_{k+1} is identical to p_i or p_i is recursively implied by p_{k+1} in some other CHCs.

Dealing with recursive CHCs is challenging because there are occurrences of mutually dependent unknown predicate symbols on both sides of a CHC. Using a novel counterexample driven sampling approach, we iteratively solve a CHC system with recursive CHCs like $(*)$ by exploring the interplay of either strengthening the solutions of some p_i $(1 \leq i \leq k)$ with new negative data or weakening the solution of p_{k+1} with new positive data. This process continues until either a valid interpretation of each unknown predicate or a counterexample to validity is derived.

1.1 Main Contributions

We depict our overall framework in Fig. 2. Our machine learning toolchain efficiently learns CHC interpretations, which are discharged by an SMT solver, in a fully automated counterexample-guided manner. This paper makes the following contributions:

- We propose to learn arbitrarily shaped invariants with feature predicates learned from linear classification algorithms. We show how to extract such predicates even when samples are not linearly separable.
- We propose a layered machine learning toolchain to combat over- and under-fitting of linear classification. We use decision tree learning on top of linear classification to improve the quality of learned hypotheses.
- We propose counter-example guided CHC sampling to verify CHC-modeled programs with complex loop and recursive structure.

We have implemented this framework inside the SeaHorn verification framework [15] as an LLVM pass. Evaluation results over a large number of challenging and nontrivial C programs collected from the literature and SV-COMP benchmarks [39] demonstrate that our solver is both efficient in proving that a CHC system is satisfiable (a program is safe) and effective in generating concrete counterexamples when a CHC system is unsatisfiable (a program is unsafe).

2 Overview

In this section, we present the main technical contributions of the paper using a number of simple programs that nonetheless have intricate invariants that confound existing inference techniques.

2.1 Learning Arbitrarily-Shaped Invariants

Our predicate search space includes the infinite Polyhedra domain: $\mathbf{w}^T \cdot \mathbf{v} + b \geq 0$ where $\mathbf{v}$ is a vector of variables and the weight values of $\mathbf{w}$ and the bias value of b are unrestricted. The search space defines the concept class from which invariants are generated. We discuss the basic intuition of how Polyhedral predicates are discovered from data using the program shown in Fig. 3.

We draw positive and negative samples collected in the loop head of this program, as depicted in Fig. 6(*i*). Intuitively, the positive samples (+) are obtained by running the loop with

$$\{(x, y) \mid (0, -2), (0, -1), (0, 0), (0, 1)\}$$

which do not raise a violation of the assertion in Fig. 3; the negative samples (∘) are derived from running the loop with

$$\{(x, y) \mid (3, -3), (-3, 3)\}$$

which do throw an assertion violation.[2]

As opposed to previous approaches [38] that tune linear classification techniques to obtain a perfect classifier that must classify all positive samples correctly, our algorithm LinearArbitrary simply applies standard linear classification techniques (e.g. SVM [30] and Perceptron [9]) to the samples in Fig. 6(*i*), to yield the linear classifier:

$$-x - y - 1 \geq 0$$

Notably, this classifier only partially classifies the positive data, as depicted in Fig. 6(*ii*). In particular, the three positive

[2]We describe how to obtain positive (+) and negative (∘) data in Sec. 2.3.

```
// Program needs ⋁ ⋀-invariant
main(){
 int x,y;
 x=0; y=*;
 while(y!=0){
  if (y<0) {x--; y++;}
  else {x++; y--;}
  assert(x!=0);}}
```

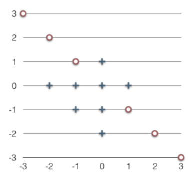
Figure 3. Program (a)

```
// Program needs Polyhedral invariant
main(){
  int x,y,i,n;
  x=y=i=0; n=*;
  while(i<n) {
    i++; x++;
    if(i%2==0) y++;}
  assert(i%2!=0||x==2*y);}
```

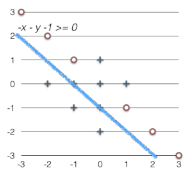
Figure 4. Program (b)

```
// Program with recursive function
fibo(int x) {
 if (x < 1) return 0;
 else if (x==1) return 1;
 else
  return fibo(x-1)+fibo(x-2);}
main(int x) {
 assert(fibo(x)>=x-1);}
```

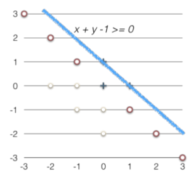
Figure 5. Program (c)



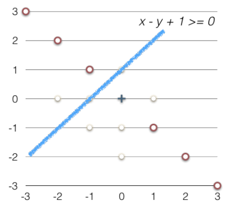
(i) Samples of Program (a) in Fig. 3



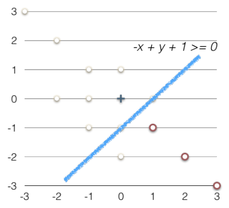
(ii) $\varphi : -x - y - 1 \geq 0$



(iii) $\varphi : -x - y - 1 \geq 0 \vee x + y - 1 \geq 0$



(iv) $\varphi : -x - y - 1 \ \geq 0 \ \vee \ x + y - 1 \geq 0 \vee$
$x - y + 1 \geq 0$



(v) $\varphi : -x - y - 1 \geq 0 \ \vee \ x + y - 1 \geq 0 \vee$
$(x - y + 1 \geq 0 \ \wedge \ -x + y + 1 \geq 0)$

Figure 6. Learning *arbitrarily shaped* invariants with *linear* classification on Program (a) in Fig. 3.

samples that are above the classification line $-x - y - 1 = 0$ (in blue) are clearly misclassified.

Because we are solving a verification problem, we cannot tolerate any misclassification of a positive sample if we wish to have our presumed invariant pass the verification oracle (i.e., an SMT solver). We therefore apply the linear classification technique again on the misclassified positive samples and all the negative samples; this results in the classification shown in Fig. 6(iii), represented by the formula $x+y-1 >= 0$. The combined classifier is thus:

$$-x - y - 1 \geq 0 \vee x + y - 1 \geq 0$$

that separates all positive samples except $\{x = 0, y = 0\}$ from all the negative data.

Our only concern for the moment is how to deal with the remaining unseparated positive data $\{x = 0, y = 0\}$. We apply our learning algorithm again on this sample and all the negative data. As shown in Fig. 6(iv), this yields another linear classifier $x - y + 1 \geq 0$, which unfortunately now misclassifies a number of negative samples below the classification line $x - y + 1 \geq 0$ (in blue). Based on our goal that every positive sample must be separated from any negative one, we again apply linear classification on the positive sample $\{x = 0, y = 0\}$ and the misclassified negative samples in Fig. 6(iv) to yield the classifier shown in Fig. 6(v). From Figs. 6(iv) and Fig. 6(v), we realize the conjunctive classifier

$$x - y + 1 \geq 0 \wedge -x + y + 1 \geq 0$$

that fully separates $\{x = 0, y = 0\}$ from all negative samples.

Finally, combining the classifiers learnt from Figs. 6(ii), 6(iii), 6(iv) and 6(v), we obtain the classifier

$$\begin{gathered} -x - y - 1 \ \geq \ 0 \ \vee \ x + y - 1 \geq \ 0 \vee \\ (x - y + 1 \geq \ 0 \ \wedge \ -x + y + 1 \geq \ 0) \end{gathered}$$

that separates *all* positive samples from *all* negative data.

In summary, our machine learning algorithm LinearArbitrary uses off-the-shelf linear classification algorithms to separate samples even when they are not linearly separable, and accepts classification candidates in arbitrary Boolean combinations. As a result, LinearArbitrary inherits all the benefits of linear classification, explored over the years in the machine learning community, with well-understood trade-offs between precision and generalization. LinearArbitrary can efficiently search from the Polyhedra abstract domain, which among the many numerical domains introduced over the years, is one of the most expressive (and expensive). We reduce searching Polyhedral invariants to well-optimized linear classification tasks to gain efficiency; but, because we do not bound the Boolean form of invariants generated, we do not lose expressivity in the process. Observe that the learned classifier shown above requires both conjunctions and disjunctions, a capability which is out of the reach of existing linear classification-based verification approaches [38].

2.2 A Layered Machine-Learning Toolchain

An important design consideration of data-driven methods like LinearArbitrary's is how to best combat over-fitting and under-fitting of data, allowing learned invariants to be as general as possible. If the linear classifier imposes a high penalty for misclassified points it may only produce over-fitted classifiers; otherwise it may under-fit. In either case, a classification-based verification algorithm may never reach a correct program invariant or may depend on a sufficiently long cycle of sampling, learning, and validation to converge. The problem is exacerbated as the complexity of the concept class (here Polyhedra) increases.

Ideally, we would like to learn a classifier within the sweet spot between under-fitting and over-fitting. Consider the example program (b) in Fig. 4.[3] Using LinearArbitrary, we obtain the following candidate invariant for the unknown loop invariant $\rho(i, x, y, n)$ of this program:

$$\rho \equiv \left\{ \begin{array}{c} (-10i - x + 5y + 6n + 7 \geq 0 \wedge \\ -i + x \geq 0 \wedge i - x \geq 0 \wedge -i + 2x - 2y \geq 0) \vee \\ 2i + 3x + 4y + 2n - 34 \geq 0 \end{array} \right\}$$

This candidate does not generalize and is not a loop invariant of the program. The first and last atomic predicates shown above are unnecessarily complex and restrictive. We can certainly ask for more samples from the verification oracle to refine ρ. However, a more fruitful approach is to explore ways to produce simpler and more generalizable invariants from the same amount of data.

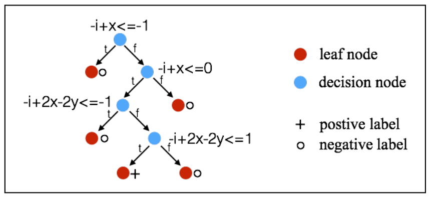
Observe that each learned Polyhedral classifier implies a hyperplane in the form of $f(\mathbf{v}) = 0$ where $f(\mathbf{v}) = \mathbf{w}^T \cdot \mathbf{v} + b$. We call $f(\mathbf{v})$ a *feature attribute* selected for classification. For example, the first classifier of the invariant ρ above is based on the feature attribute $-10i - x + 5y + 6n + 7$. Using it as a separator leads to binary partitions of data each of which contains a *subset* of positive and negative samples, causing ρ to be a disjunctive conjunctive formula. Clearly, the learning algorithm has made a trade-off, misclassifying some sample instances to avoid over-fitting based on its built-in generalization measure.

Is the choice always reasonable? To generalize the invariant ρ, we first need to quantify the *goodness* of a feature attribute. In machine learning theory, *information gain* is often used to evaluate the quality of an attribute $f(\mathbf{v})$ for a particular classification task. Informally, the information gain of an attribute evaluates how homogeneous the samples are after choosing the attribute as a classifier. We prefer high information gain attributes that lead to a split that causes two partitions, one with more positive samples and the other with more negative. However, there are no guarantees on learning a high information gain split at every internal classification task. Leveraging information gain as a measure, we aim to use Decision Tree learning [31] to generalize the classification result produced by LinearArbitrary to heuristically choose attributes to build classifiers with higher information gains. When used for classification, such attributes usually generalize better and can yield simpler classifiers than low information gain ones. Empirically a simple invariant is more likely to generalize than a complex one [29].

[3]We do not draw the samples of this program because they are complex to visualize, involving constraints over four dimensions.

Decision Trees. A decision tree (DT) is a binary tree that represents a Boolean function. Each inner node of the tree is labeled by a decision of the form $f(\mathbf{v}) \leq c$ where $f(\mathbf{v})$ is a feature attribute over a vector of variables $\mathbf{v}$ and c is a threshold. In our context, $f(\mathbf{v})$ is learned from LinearArbitrary. Each leaf of the tree is labeled either positive $+$ or negative $\circ$. To evaluate an input $\mathbf{x}$, we trace a path down from the root node of the tree, going to a true branch (t) or a false branch (f) at each inner node depending on whether its feature $f(\mathbf{x})$ is less or equal to its threshold c. The output of the tree on $\mathbf{x}$ is the label of the leaf reached by this process.

For example, applying a DT learning algorithm [31] with feature attributes drawn from atomic predicates in ρ, $\{-10i - x + 5y + 6n + 7, -i + x, i - x, i + x - 4y, 2i + 3x + 4y + 2n - 34\}$, which are all learned from LinearArbitrary, yields the following decision tree:



At the root node, the DT does not choose the complex attribute $-10i - x + 5y + 6n + 7$ as in LinearArbitrary. Instead, DT learning chooses the simpler attribute $-i + x$ and learns a threshold -1 to bound the attribute because such a decision can lead to a higher information gain split as the left child node of the root now contains only negative samples. Eventually, DT learning is able to pick two concise attributes $-i + x$ and $-i + 2x - 2y$ to separate all the positive and negative data and equip them with properly adjusted thresholds. As there is a single decision path that leads to positive samples in the above DT, combining all decisions along that path yields a new loop invariant for program (b) in Fig. 4:

$$\neg(-i + x \leq -1) \wedge -i + x \leq 0 \wedge$$
$$\neg(-i + 2x - 2y \leq -1) \wedge -i + 2x - 2y \leq 1$$

which suffices to verify the program. Importantly, the use of DT learning generalizes ρ on the same data from which ρ was learned before by LinearArbitrary, without having to ask for more samples.

2.3 Counterexample Guided CHC Sampling

The prior sections assume the existence of positive and negative data sampled from the program to bootstrap learning. However, sampling from a program is challenging when the code base is large. To make data-driven methods of the kind we propose practical for scalable verification, we need to efficiently sample positive and negative data directly from CHCs in an automatic manner, that nonetheless can scale to programs with complex loops and recursive functions.

The program (c) in Fig. 5 shows why recursion might confound existing learning based tools, and complicate sampling data directly from CHCs. We show the CHCs of the program:

$$x < 1 \land y = 0 \rightarrow p(x, y) \tag{5}$$

$$x \geq 1 \land x = 1 \land y = 1 \rightarrow p(x, y) \tag{6}$$

$$x \geq 1 \land x \neq 1 \land p(x-1, y_1) \land p(x-2, y_2) \land y = y_1 + y_2 \rightarrow p(x, y) \tag{7}$$

$$p(x, y) \rightarrow y >= x - 1 \tag{8}$$

To prove this program correct, it is sufficient to find an interpretation of $p(x, y)$ which encodes the input and output behavior of the function `fibo`. CHC (5) and (6) correspond to the initial cases of the function while constraint (8) encodes the safety property to be satisfied by $p(x, y)$. CHC (7) corresponds to the inductive case. Given an interpretation of the unknown predicate p, it is straightforward to sample positive data from CHC (5) and (6) and negative data from constraint (8), but is less clear how to deal with the recursive CHC (7) because there are occurrences of the unknown predicate symbol p on both sides of the constraint.

For example, suppose that we interpreted $p(x, y)$ to be:

$$-23x + 25y + 22 >= 0 \land -y + 1 >= 0$$

and we used an SMT solver to check the validity of constraint (7) under this interpretation. The following counterexample might be returned:

$$p(x-1, y_1) = p(2, 1),\ p(x-2, y_2) = p(1, 1),$$
$$p(x, y) = p(3, 2)$$

This is a counterexample because the second conjunct in the above interpretation of $p(x, y)$ is false when $y = 2$. At this point, however, it is not clear whether we should add $(3, 2)$ as a new positive sample, given that $p(2, 1)$ and $p(1, 1)$ are true, thus implying $p(3, 2)$ is true because of CHC (7); or, whether we should add $(2, 1)$ and/or $(1, 1)$ as negative samples, given that $p(3, 2)$ is a counterexample.[4]

Our approach is inspired by modern CHC solvers [1, 25, 32]. They solve a recursive CHC system $\mathcal{H}$ by iteratively constructing a series of recursion-free unwindings of $\mathcal{H}$, which can be considered as derivation trees of $\mathcal{H}$ that are essentially logic program executions [25], whose solutions are then generalized as candidate solutions of $\mathcal{H}$.

[4]Notably the ICE framework [10, 11] is not suitable to deal with this counterexample as p occurs more than once on the left-hand side of CHC (7).

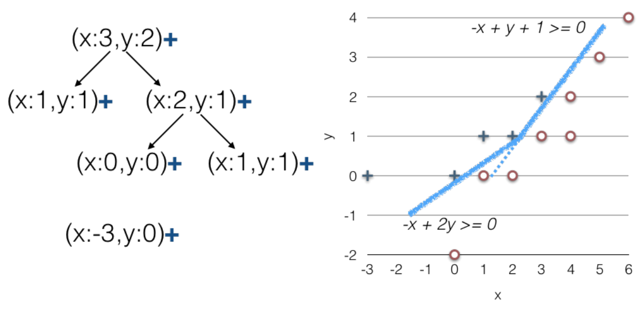


Figure 7. Learning the invariant for Program (c) in Fig. 5 with bounded positive samples that form derivation trees.

Instead of explicitly unwinding recursive CHCs in $\mathcal{H}$, we study the problem from a data perspective. Our approach only implicitly unwinds $\mathcal{H}$ by considering positive data sampled from a finite unwinding. A positive sample s is obtained from a bounded unwinding of $\mathcal{H}$ if we are able to recursively construct a derivation tree using other positive samples, explaining how s is derived. For example, we draw the positive samples collected for p using this approach in Fig. 7. For the counterexample generated above, we can add $(x = 3, y = 2)$ as a positive example of $p(x, y)$ because both $(x = 2, y = 1)$ and $(x = 1, y = 1)$ are already labeled as positive, which can be used to explain how $(x = 3, y = 2)$ is derived. Note that in Fig. 7 every positive sample can be explained by how it is derived from other positive samples except data points generated from initial CHC (5) and (6). Samples that do not satisfy this condition are labeled negative to allow strengthening CHC (7) until it becomes inductive. If p gets strengthened too much, it can be weakened later with more positive samples from CHC (5) and (6).

Similar to how solutions of recursive CHC systems are derived from unwound recursion-free CHC systems in modern CHC solvers, our verification framework uses a machine learning algorithm to explain *why* p has a valid interpretation by separating positive samples of p which are derived from implicit unwindings of the CHCs (5)-(8), from sampled negative data, relying on the underlying machine learning algorithm to generalize our choice of the solution to p. For the positive data and negative data depicted in Fig. 7, applying LinearArbitrary yields a classifier,

$$-x + y + 1 \geq 0 \land -x + 2y \geq 0$$

that correctly interprets $p(x, y)$ in CHCs (5)-(8) and hence proves that the program (c) in Fig. 5 is safe.

Of course, we may have to iteratively increase the implicit unwinding bound based on newly discovered positive samples until we can prove the satisfiability of the CHCs. Empirically, we find that our counterexample guided sampling approach is efficient. For example, consider changing the annotated assertion in Fig. 5 to *assert* $(x < 9 \,||\, \texttt{fibo(x)} \geq 34)$, a difficult verification task in the SV-COMP benchmarks [39].

To verify the program, our tool needs to sample positive data of the `fibo` function using at least inputs from 0 to 10. It proves the program in less than one minute with a complicated disjunctive and conjunctive invariant learned while the best two tools in the recursive category of SV-COMP'17 either timeout or run out of memory [39] .

3 Learning Procedure

We now formalize our learning algorithm discussed in Sec. 2. Given a number of positive samples S^+ and negative samples S^- over a vector of variables $\mathbf{v}$, the learning procedure aims to produce a classifier over $\mathbf{v}$ that can separate positive instances in S^+ from negative ones in S^-.

3.1 Background: Linear Classification

We briefly survey background in the context of linear binary classification. A linear model considers training examples $S^+ \cup S^-$ as points in a d-dimensional space where d is the dimension of $\mathbf{v}$, and treats each dimension as one feature. A linear binary classifier defines a hyperplane in the space classifying the examples, which is a generalization of a straight line in 2-dimensional space, in the form $f(\mathbf{v}) = 0$ where $f(\mathbf{v}) = \mathbf{w}^T \cdot \mathbf{v} + b$.

Each coefficient in the weight vector $\mathbf{w}$ can be thought of as a weight on the corresponding feature. Geometrically, $\mathbf{w}$ is also called a *normal vector* of the separating hyperplane (which is perpendicular to the hyperplane). The bias b is the intercept of the hyperplane (which can also be included in the weight vector by adding a dummy feature to $\mathbf{v}$ which is always set to 1). A separating hyperplane can be used as a classifier $\varphi(\mathbf{v}) \equiv f(\mathbf{v}) \geq 0$ to predict the label of a new point x, by simply computing $\varphi(x)$. In other words, if $\varphi(x)$ is valid (*true*) then we predict the label to be +1 (positive) and −1 (negative) otherwise.

The goal of the learning process is to come up with a "good" weight vector $\mathbf{w}$ (including b) estimated from training examples. If the samples in S^+ and S^- are indeed linearly separable, then we expect $\varphi(s)$ to be valid for all positive samples, and $\neg\varphi(s)$ to be valid for all negative samples. However, different notions of "goodness" exist, which yield different linear classification learning algorithms. If the samples are not linearly separable, there are also different strategies that can be adopted to balance the trade-off between precision and generalization.

We consider two linear classification algorithms in our implementation: Perceptron [9] and SVM [30]. They consider the quality of a candidate classifier by measuring the *margin* of a classifier, which is determined by the distance from the classifier decision surface to the closet data points, often referred to as the support vectors in a vector space. For example, the SVM algorithm maximizes the margin of its produced classifier because a decision boundary drawn in the middle of the void between data items of the two classes is deemed to be better than one which closely approaches examples of one or both classes.

In practice, data is complex and may not be separated perfectly with a hyperplane. In these cases, we must relax our goal of maximizing the margin of the hyperplane that separates the classes . To generalize, some training data should be allowed to violate the hyperplane. To constrain the amount of margin violation permitted, existing SVM algorithms use a so-called C parameter to control the precision of a classifier. Thus, to balance the trade-off between generalization and precision, we must adjust C. For large values of C, the optimization chooses a smaller-margin hyperplane if that hyperplane can classify *all* the training points correctly. Conversely, a very small value of C will cause the optimizer to search a larger-margin separating hyperplane, even if that hyperplane misclassifies *more* points. We prefer a smaller value of C to obtain classifiers with larger margins that are more likely to generalize.

3.2 LinearArbitrary

We now formalize our classification algorithm LinearArbitrary, which can find classifiers that can be expressed within the Polyhedra domain (i.e., the theory of linear arithmetic). In cases that the samples are not linearly separable, our algorithm can find a classifier expressed as an arbitrary Boolean combination of linear inequalities.

The pseudo-code of LinearArbitrary is given in Algorithm 1. In line 1 of the algorithm, we ask a linear classification algorithm to produce a classifier φ that tries to separate given positive samples S^+ and negative samples S^-. To this end, we exploit well-developed heuristics in linear classification to balance the trade-off between generalization and precision. For example, if we use SVM classification, we predefine the C parameter to be reasonably small based on the margin constraint, so that larger margin separating hyperplanes are produced, introducing the possibility of misclassified samples. For the remaining misclassified samples, our key insight is that we can apply the linear classification algorithm *iteratively* to learn a family of classifiers that together separate all positive and negative samples. These classifiers can be thought of as logically connected with appropriate boolean operators $\wedge$ or $\vee$.

Line 2, 3, 4 of Algorithm 1 collect positive samples that are correctly classified $S^+_{\checkmark}$ by φ, positive samples misclassified $S^+_{\times}$ by φ, and negative samples misclassified $S^-_{\times}$ by φ.

If positive samples in $S^+_{\checkmark}$ are not fully separated from all negative samples (line 5), we recursively call Algorithm 1 on $S^+_{\checkmark}$ and $S^-_{\times}$ to learn a new classifier $\varphi \wedge \text{LinearArbitrary}(S^+_{\checkmark}, S^-_{\times})$ that should together make $S^+_{\checkmark}$ and S^- separable. Dually, as any positive samples must be included in an invariant to ensure generalization so as to eventually pass the verification oracle but φ still misclassifies $S^+_{\times}$, in line 7 and 8, we recursively call Algorithm 1 on $S^+_{\times}$ and S^- to learn a new

Algorithm 1: LinearArbitrary (S^+, S^-)

```
1 $\varphi$ = LinearClassify ($S^+, S^-$);
2 $S^+_{✓} = \{ s \in S^+ \mid \varphi(s) \}$;
3 $S^+_{✗} = \{ s \in S^+ \mid \neg\varphi(s) \}$;
4 $S^-_{✗} = \{ s \in S^- \mid \varphi(s) \}$;
5 if $S^-_{✗} \neq \emptyset$ then
6     $\varphi = \varphi \wedge$ LinearArbitrary($S^+_{✓}, S^-_{✗}$);
7 if $S^+_{✗} \neq \emptyset$ then
8     $\varphi = \varphi \vee$ LinearArbitrary($S^+_{✗}, S^-$);
9 return $\varphi$
```

invariant $\varphi \vee$ LinearArbitrary ($S^+_{✗}, S^-$) that should separate all positive samples from all negative samples. We then return the final classifier φ in line 9. Observe that the classifier returned by Algorithm 1 can be an arbitrary Boolean combination of discovered predicates.

Complexity. The number of LinearClassify calls made by Algorithm 1 is $O(|S^+||S^-|)$ in the worst case scenario when the most unbalanced partition of S^+ occurs at every step, dividing the positive samples into $S^+_{✓} = S^+, S^+_{✗} = \emptyset$ (assuming $S^-_{✗} \neq \emptyset$). In the most balanced case, the algorithm divides S^+ into two nearly equal partitions. This means each recursive call processes a positive sample set of half the size. The result is that the algorithm uses only $O(|S^+|)$ LinearClassify calls.

3.3 Machine Learning Tool Chain

Despite the reuse of well-developed heuristics that aim to balance precision and generalizability found in machine learning tools exploited by LinearArbitrary, we still lack a formal guarantee that a learned classifier does not over-fit or under-fit, especially when training samples are not linearly separable. If over- and under-fitting indeed affects verification results, we can ask the verification oracle to provide more examples to refine a learned invariant. In this section, however, we consider the problem from a different perspective: can we generalize the outcome of LinearArbitrary without additional data? Doing so may not only increase the *quality* of the classifier produced by LinearArbitrary, but would also enable a more efficient verification technique with improved convergence time.

We observe that each linear classifier $\varphi(\mathbf{v})$ found by LinearArbitrary defines a *feature attribute*

$$f(\mathbf{v}) = \mathbf{w}^T \cdot \mathbf{v} + b$$

where $\mathbf{w}$ and b are the weights and bias of the classifier *resp.*, that may only separate a *subset* of positive and negative samples. In machine learning parlance, the capability of a feature attribute $f(\mathbf{v})$ to serve as a classifier is given in terms of a threshold c - $f(\mathbf{v}) \leq c$ (*resp.* $f(\mathbf{v}) > c$). We leverage c to characterize the information gain of an attribute over a set

Algorithm 2: Learn (S^+, S^- /*, *additional_features* */)

```
1 $\varphi$ = LinearArbitrary($S^+, S^-$);
2 features = atomics ($\varphi$) /*$\cup$ additional_features */;
3 return DT-Learn($S^+, S^-$, features)
```

of samples $S = S^+ \cup S^-$ based on Shannon Entropy ϵ:

$$\epsilon(S) = -\frac{|S^+|}{|S|}\log_2\frac{|S^+|}{|S|} - \frac{|S^-|}{|S|}\log_2\frac{|S^-|}{|S|}$$

which yields a value that rates the ratio of positive and negative samples. A small entropy value indicates that S contains significantly more of one than the other. The *information gain* γ of f on S with respect to a chosen threshold c is specified as:

$$\gamma(S, f : c) = \epsilon(S) - \Big(\frac{|S_{f:c}|\epsilon(S_{f:c})}{|S|} + \frac{|S_{\neg f:c}|\epsilon(S_{\neg f:c})}{|S|}\Big)$$

where $S_{f:c}$ and $S_{\neg f:c}$ are instances satisfying $f(\mathbf{v}) \leq c$ and instances that do not. Informally, information gain evaluates to how homogeneous the samples are after choosing f and threshold c. An attribute with less information gain that results in two partitions each with roughly half positive and half negative instances is less preferred than an attribute with high information gain that results in partitions which have a dominant fraction of positive or negative samples. Choosing attributes with large information gains naturally leads to a simpler classifier which is more likely to generalize than a complex one.

To generalize the outcome of LinearArbitrary, we apply decision tree (DT) learning, another well-developed machine learning algorithm for this generalization task, as explained in Section 2.2. The hypothesis set corresponding to all DTs consists of arbitrary Boolean combinations of linear inequalities of the form $f(\mathbf{v}) \leq c$, between features and thresholds. Appropriate threshold values are expected to be learned to bound selected attributes. Standard DT learning algorithms [31] start from an empty tree, and greedily pick at each node the best feature attribute and threshold that separates the remaining training samples. This procedure continues until all leaves of the tree have samples labeled by a same class.

Algorithm 2 describes our machine learning toolchain. For the moment, ignore the pseudo-code that is commented. The algorithm takes positive and negative samples as inputs. It automatically learns feature attributes as a number of atomic predicates that compose the classifier learned by LinearArbitrary in line 1 and line 2. In line 3, we run a standard DT learning algorithm on the samples to obtain a decision tree using the feature attributes. In the tool, we tune the parameter of the used DT learning implementation to ensure that the decision tree must classify all samples correctly.

In the algorithm, the use of DT learning can be thought as a posterior process of LinearArbitrary, which selects learned Polyhedral attributes with high information gains at a separation that is more likely to generalize and simultaneously adjusts the thresholds for selected feature attributes in order to fit the data. Algorithm 2 essentially forms a machine learning tool chain to combat over- and under-fitting.

To convert the DT into a formula, we note that the set of states that reach a particular leaf is given by the conjunction of all predicates on the path from the root to that leaf. Thus, the set of all states classified as positive by the DT is the disjunction of the sets of states that reach all the positive leaves. A simple conversion is then to take the disjunction over all paths to good leaves of the conjunction of all predicates on such paths. We can compute this formula recursively by traversing the learned DT.

Lemma 3.1. *If* φ = Learn (S^+, S^-), *then* $\forall s \in S^+$, $\varphi(s)$ *is valid and* $\forall s \in S^-$, $\varphi(s)$ *is invalid.*

Beyond Polyhedra. The use of DT learning has additional benefits. Although the Polyhedra domain is sufficient in capturing numerical relationships among numeric variables, it does not include predicates over enumeration and mod operations. Incorporating such predicates is straightforward, and is shown in Algorithm 2 as part of the commented code. The algorithm additionally takes a number of predefined feature attributes as inputs. In our experiments, the predefined features are simply Boolean variables and mod operations of a numeric variable against a constant, which can be parameterized *a priori*. DT learning can then jointly learn a unified classifier that is a combination of learned Polyhedral features and these predetermined ones.

4 Verification Procedure

This section formalizes the sampling and verification algorithms of our approach. Several verification frameworks [6, 13, 15, 18] provide customized verification semantics with different degrees of precision for CHC encoding of a functional or imperative program. Checking if a program satisfies a safety property amounts to establishing the satisfiability of a program's CHCs.

4.1 Constrained Horn Clauses

Given sets of function symbols $\mathcal{F}$ (e.g. + or −), predicate symbols $\mathcal{P}$ (unknown predicates), and variables $\mathcal{V}$, a CHC constraint, which we denote as C, is a formula

$$\forall \mathcal{V}.\ (\phi \wedge p_1[\overline{T_1}] \wedge p_2[\overline{T_2}] \wedge \cdots \wedge p_k[\overline{T_k}] \rightarrow h[\overline{T}])$$

where: k is non-negative; ϕ is a constraint over $\mathcal{F}$ and $\mathcal{V}$ with respect to some background theory (e.g., linear arithmetic in this paper); $\mathcal{V}$ are universally quantified;[5] $p_i[\overline{T_i}]$ is an application $p(t_1, \cdots, t_n)$ of an n-ary predicate symbol $p \in \mathcal{P}$ ranging over n free variables with first-order terms t_i constructed from $\mathcal{F}$ and $\mathcal{V}$; and $h[T]$ is either defined analogously to p_i or is a known predicate without $\mathcal{P}$ symbols. Here, h is called the head of the constraint and $\phi \wedge p_1[\overline{T_1}] \wedge p_2[\overline{T_2}] \wedge \cdots \wedge p_k[\overline{T_k}]$ is called the body of the constraint. A CHC system $\mathcal{H}$ consists of a set of CHC constraints. We say $\mathcal{H}$ is a recursive CHC system, if in one of its constraints C, one of the predicate symbols appearing in the body of C is identical to the head of C, or is recursively implied by the head in some other constraints other than C within $\mathcal{H}$. We use $\mathcal{P}(\mathcal{H})$ to denote all the unknown predicate symbols in $\mathcal{H}$.

An interpretation $\mathcal{A}$ of a CHC C associates each predicate symbol p_i of arity n appearing in C as a formula over its free variables; we use $C[\mathcal{A}]$ to denote the interpreted constraint. We say a CHC system $\mathcal{H}$ is satisfiable if there exists an interpretation $\mathcal{A}$ of each predicate symbol in $\mathcal{P}$ such that for each constraint $C \in \mathcal{H}$, $C[\mathcal{A}]$ is valid, i.e., the conjunction of interpretations of all predicates in the body of C and the constraint ϕ entail the interpretation of the head of C.

4.2 Data-Driven CHC Solving

We now present our CEGAR (counterexample guided abstraction refinement) based verification procedure. The basic idea is that for each unproved CHC C under a current interpretation $\mathcal{A}$, we leverage the counterexample to improve $\mathcal{A}$ until C becomes valid.

Bounded Positive Samples. As argued in Sec. 2.3, handling a counterexample obtained from discharging a recursive CHC constraint in the form $p_1(\overline{T_1}) \wedge p_2(\overline{T_2}) \cdots \wedge p_k(\overline{T_k}) \wedge \phi \rightarrow p_{k+1}(\overline{T_{k+1}})$ is challenging because the counterexample is of the form $(\{s_1, s_2, \cdots, s_k\}, s_{k+1})$ where each s_i is a sample of predicate p_i such that $1 \le i \le k+1$. We do not know in general whether we should add s_{k+1} as a new positive sample of p_{k+1} or add some s_i as a negative sample of p_i for some i where $i \le k$. Our solution is inspired by modern CHC solvers for recursive CHC constraints [1, 25, 32]. Given a recursive CHC system $\mathcal{H}$, they attempt to solve $\mathcal{H}$ by constructing a series of recursion-free systems from bounded unwindings of $\mathcal{H}$, solving each of the recursion-free systems, and combining the solutions to construct a solution of $\mathcal{H}$. In our approach, we do not attempt to unwind $\mathcal{H}$ explicitly. Instead, we implicitly unroll $\mathcal{H}$ by considering positive samples bounded by a finite number of $\mathcal{H}$ unwindings. We add a positive sample s_{k+1} of p_{k+1} *iff* $s_1, \cdots, s_k$ are all labeled as positive for $p_1, \cdots, p_k$ respectively. The new positive sample s_{k+1} is bounded by a finite unwinding of $\mathcal{H}$ because we are able to recursively construct a derivation tree of positive samples, explaining how s_{k+1} is obtained (from $s_1, \cdots s_k$). Samples that do not satisfy such a condition are considered negative at first.

Z3 Support. Our algorithm is built on top of the Z3 SMT solver [7]. We assume that a number of Z3 functions are available to us to build the CHC solver: *Z3Check* $(C[\mathcal{A}])$ which

[5]We ignore universal quantifiers in our presentation for simplicity.

Algorithm 3: CHCSolve ($\mathcal{H}$)

```
1  $\mathcal{A} = \lambda p : true$;
2  $\forall p \in \mathcal{P}(\mathcal{H}).\ s^+(p) = s^-(p) = \emptyset$;
3  while
     $\exists\ (C \equiv \phi \wedge p_1[\overline{T_1}] \wedge p_2[\overline{T_2}] \wedge \cdots \wedge p_k[\overline{T_k}] \rightarrow h[\overline{T}]) \in \mathcal{H}$
     s.t. not (Z3Check ($C[\mathcal{A}]$)) do
4      do
5          $s$ = Z3Model ($C[\mathcal{A}]$);
6          $\forall\ i.\ 1 \leq i \leq k.\ s_i = \{Z3Eval\ (t_{ij}, s) \mid t_{ij} \in$
7                $\overline{T_i} \equiv \{t_{i1}, \cdots, t_{in}\}\}$;
8          $s_h = \{Z3Eval\ (t_j, s) \mid t_j \in \overline{T} \equiv \{t_1, \cdots, t_n\}\}$;
9          if $(\forall\ i.\ 1 \leq i \leq k.\ s_i \in s^+(p_i))$ then
10             if $h \in \mathcal{P}$ then
11                 $s^+(h) = s^+(h) \cup \{s_h\}$;
12                 $s^-(h) = \emptyset$;
13                 $\mathcal{A}(h) = true$;
14             else
15                 return $\mathcal{H}$ is unsat with counterex $s_h$
16         else
17             for $i \leftarrow 1$ to $k$ do
18                 if $s_i \notin s^+(p_i)$ then
19                     $s^-(p_i) = s^-(p_i) \cup \{s_i\}$;
20                     $\mathcal{A}(p_i)$ = Learn ($s^+(p_i)$, $s^-(p_i)$);
21             end
22     while not (Z3Check ($C[\mathcal{A}]$));
23 end
24 return $\mathcal{H}$ is sat with interpretation $\mathcal{A}$
```

validates an interpreted formula $C[\mathcal{A}]$ by checking whether the negation of the formula is unsat; *Z3Model* ($C[\mathcal{A}]$) that returns a model s explaining why *Z3Check* finds that $C[\mathcal{A}]$ is invalid where a model in Z3 is a satisfiable assignment to its input formula; *Z3Eval* (t, s) which evaluates a first-order logic term t with a model s.

Algorithm. We present the CHC solver in Algorithm 3. It takes a (recursive) CHC system $\mathcal{H}$ as input and outputs either an interpretation $\mathcal{A}$ of $\mathcal{H}$ if it is satisfiable or a counterexample showing why $\mathcal{H}$ is unsatisfiable.

In line 1, the initial interpretation $\mathcal{A}$ maps each of the unknown predicate symbols to the logic predicate *true*. Line 2 initializes the sample set of all the unknown predicate symbols in $\mathcal{H}$ to the empty set. From line 3 to line 23, the algorithm iteratively picks a CHC C which cannot be proven with the current interpretation $\mathcal{A}$, and then resolves it via the loop from line 4 to line 22 until $C[\mathcal{A}]$ becomes valid.

In line 5, for the invalid constraint C, we ask Z3 to return a model s as a counterexample witnessing why C is unprovable with the current interpretation $\mathcal{A}$, by calling *Z3Model*. To refine the solution for each involved unknown predicate symbol p_i, we are, however, more interested in counterexamples for each p_i (and h). To this end, in line 6 to 8, the model s is converted to samples of each predicate symbol occurring in C: $p_1, \cdots, p_k$ and h if $h \in \mathcal{P}$. Note that $p_i[\overline{T_i}]$ is an application $p_i(t_{i1}, \cdots, t_{in})$ of an n-ary predicate symbol p_i. The sample s_i of p_i can be obtained by calling *Z3Eval* with the model s on each first-order term t_{ij} ($1 \leq j \leq n$). The sample s_h of h is derived from s analogously.

In line 9, if each sample s_i of p_i can be observed in $s^+(p_i)$, we fix that s_h should be a positive sample for h as discussed above. An example of such a case is depicted in Fig. 7 where the head h needs to be weakened to include the new sample s_h (in this example h corresponds to $p(x, y)$ in CHC (7) of the program in Fig. 5). In the algorithm, we do so in line 11 if $h \in \mathcal{P}$. In line 12 we clear the negative samples of s_h so that we can set $\mathcal{A}(h)$ to the weakest solution *true* and break the refinement loop for C from line 4 to line 22. This is intentional - since h should be updated to accommodate the new positive sample, we prefer to solve a CHC constraint that uses h in its body *before* solving C that consumes h in the head. Thus, Algorithm 3 can be thought as propagating *premises* of a CHC constraint as *conclusions* for another CHC constraint on which it topologically depends.

However, if $h \notin \mathcal{P}$ and is a known predicate (e.g. assertion predicate), h cannot be weakened to accommodate s_h. In fact, this indicates $\mathcal{H}$ is not satisfiable. In line 15, Algorithm 3 terminates with the counterexample s_h. Because positive samples are derived from an unwinding of $\mathcal{H}$, we can construct a derivation tree of positive samples from s_h, showing why $\mathcal{H}$ is unsatisfiable.

If s_h cannot be included as a new positive sample for h, in line 17-21, we in turn add s_i to $s^-(p_i)$ as a tentative negative sample if s_i does not show up in $s^+(p_i)$. In line 20, the solution for such a p_i is strengthened in $\mathcal{A}$ by observing its positive and new negative samples, using the learning algorithm presented in Algorithm 2, i.e., the body of C is strengthened. Algorithm 3 iteratively strengthens the body of C until $C[\mathcal{A}]$ is valid. It terminates when all CHCs are solved in line 24.

Note that in Algorithm 3, negative samples of a CHC system $\mathcal{H}$ are not bounded by an unwinding of $\mathcal{H}$. Instead, they are key to the interplay of interpretation strengthening and weakening in the algorithm. The solutions of unknown predicates in the body of an unproved CHC C are strengthened until C is inductive in the inner loop. However, across the outer loop iterations, the algorithm may include a larger set of positive samples for these predicates, weakening candidate invariants. Alternatively one could sample negative instances bounded to an unwinding of $\mathcal{H}$. However, in this case, as both positive and negative samples need to be bounded, one must explicitly construct a series of unwindings of $\mathcal{H}$ as in existing CHC solvers [25]. We use the former approach because we want to avoid explicit unwindings and we have found that the interplay between strengthening and weakening is effective in our experience.

Lemma 4.1. *Algorithm 3 is sound.*

The loop invariant that Algorithm 3 maintains is that, at any time, for any unknown predicate symbol p: (i) $\mathcal{A}(p)$ is a classifier that separates positive and negative samples of p collected so far; (*ii*) positive examples of p form a forest of derivation trees. Thus the algorithm is sound as any counterexample to unsatisfiability is guaranteed to be valid. Due to Lemma 3.1, Algorithm 3 can make progress in the sense that it does not repeat a positive sample of p.

Although Algorithm 3 is sound, it may not terminate as derivation trees of positive samples can grow unboundedly and there is also no restriction on the size of potential negative samples. Nonetheless, our experimental results indicate that the algorithm performs well in practice.

5 Implementation

We have implemented our CHC solver, LinearArbitrary, in C++ as an LLVM pass in the SeaHorn verification framework [15]. Our tool is available in [26]. Our implementation supports C programs with multiple nested loops and recursive functions. Notably, as discussed above, it does not require seeded tests as input, automatically generating samples purely from counterexamples. Our tool is customized to use any linear classification algorithms specified by the user. The built-in linear classification algorithms include Perceptron [9] and SVM [30]. We use the decision tree implementation described in [11].

In certain cases the SVM implementation [4] we use only outputs a "dummy classifier" $1 \geq 0$, i.e., all weight values from $\mathbf{w}$ are reduced to zero. For example, consider a case in which positive samples only include $(x : 0, y : 0)$ and all its neighbors are negative. The SVM implementation produces a dummy classifier because, from all possible directions, it cannot find a decision surface that stands out. To prevent considering such classifiers, we intercept the result of the call to LinearClassify in line 1 of Algorithm 1. If the result is a dummy classifier, we call the SVM implementation again taking either S^+ and a random negative sample or a random positive sample and S^- as inputs depending on which case such a simpler call can yield a non-dummy classifier.

6 Experiments

We conduct a thorough evaluation of our CHC solver and compare it with several state-of-the-art CHC solvers [16, 17, 19, 25] and learning-based verifiers [11, 27, 29] using a large set of benchmarks including test suites used by such previous approaches [11, 29] and several categories of non-trivial programs in the SV-COMP benchmarks [39]. In this paper, we report results that only use the SVM library given in [4] as the implementation of LinearClassify called by Algorithm 1.

Learning Feature Predicates. Our CHC solver uses machine learning algorithms to infer feature predicates as opposed to PIE [29] that enumerates a hypothesis space in a syntax-guided manner. Fig. 8(a) compares the performance of the two approaches on a test suite of 82 programs used to evaluate [29],[6] in terms of total inference (learning) and verification time. The points under the diagonal line $y = x$ are benchmarks for which our tool discovers a verified solution more quickly than PIE. The points on the line $y = \text{TO}$ or $x = \text{TO}$ indicate a timeout.[7] Note that solution time is roughly an order of magnitude faster using LinearArbitrary.

We characterize two of the programs on which PIE times-out below. $\#\mathcal{C}$, $\#\mathcal{P}$, $\#\mathcal{V}$ and $\#\mathcal{S}$ denote the number of CHCs, unknown predicates, variables, and samples, resp. As each interpretation is in DNF format (recall that a learned invariant is a disjunction of all paths to positive leaves in a decision tree), $\mathcal{A}$ shows the number of conjunctions of each disjunction in the most complex invariant, separated by commas.

	$\#\mathcal{C}$	$\#\mathcal{P}$	$\#\mathcal{V}$	$\#\mathcal{S}$	$\#\mathcal{A}$	T
31.c	11	5	49	281	8, 7	14s
33.c	18	6	101	662	5	13s

These programs contain multiple nested loops with nondeterministic behavior leading to non-trivial CHCs. For example, the CHCs in 31.c contain 11 constraints over 5 unknown predicates, and the solution requires a disjunctive structure, necessitating a non-trivial search space navigation to discover such invariants using a syntax-guided approach.

Low-Dimensional Learning. When samples are not linearly separable, LinearArbitrary learns a Boolean combination of several linear classifiers to separate them. On the other end of the spectrum, DIG [27] projects samples into a high-dimensional space and learns *conjunctive* nonlinear classifiers driven by predefined *template* equations, but with limited support for disjunctive invariants. We compare the two approaches in Fig. 8(b) using benchmarks adapted from [29] for which linear invariants suffice. We characterize two of the programs on which DIG times-out:

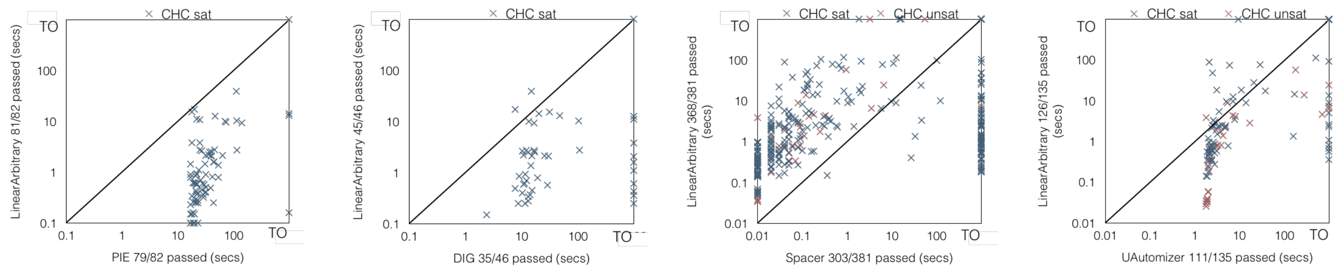
	$\#\mathcal{C}$	$\#\mathcal{P}$	$\#\mathcal{V}$	$\#\mathcal{S}$	$\#\mathcal{A}$	T
04.c	8	4	19	27	1, 1	0.4s
10.c	9	4	42	22	7, 8	0.4s

Although these programs require a relatively small number of constraints and predicates, they require disjunctive invariants to satisfy the verification oracle, which cannot be found by DIG from polynomial equation templates. We note, however, that our tool currently cannot find nonlinear polynomial invariants discoverable by DIG, an extension we leave for our future work.

Comparison with Existing CHC Solvers. In Fig. 8(c), we present a comparison of our CHC solver with Spacer [19], a state-of-the-art CHC solver that extends GPDR [17] with under-approximate summaries of unknown predicates. Our test suite includes 381 C programs obtained and adapted from the *loop-$*$* and *recursive-$*$* categories of SV-COMP [39], additional complicated loop programs from our related work, e.g. [8, 14, 29]. All the benchmarks are available in [26].

[6]Note that all graphs in the section are in log-scale.

[7]The timeout parameter was set to 180 seconds.



(a) Learning *vs* Enumeration (b) Learning *vs* Template (c) Learning *vs* PDR (d) Learning *vs* Interpolation

Figure 8. Verification time - evaluation and comparison.

In general, Spacer is able to generate a solution faster than our technique, when it terminates. However, on this benchmark suite, it was only able to verify 303 of the 381 programs, as opposed to the 368 programs we were able to generate a solution for.

We also compare our tool with another two CHC solvers, GPDR [17] and Duality [24, 25], using the same timeout parameter. Similar to Spacer, these two solvers also ran faster on the benchmarks that they terminated on but verified less CHC systems than LinearArbitrary. The result, in terms of the number of verified benchmarks, is summarized in the table below:

#Total	#GPDR	#Spacer	#Duality	#LinearArbitrary
381	300	303	309	368

Finally, to quantify the significance of DT learning in the verification pipeline, we ran all of the above experiments again, but disabled the use of DT learning in the learning procedure. The convergence rate of this version decreased significantly because it is possible that Algorithm 1 produces a low-quality classifier as the example in Sec. 2.2 shows. In this setting, most of the benchmarks could not be verified within the timeout range.

SV-COMP Programs. Since a large subset of our benchmarks come from SV-COMP [39], we compare our CHC solver with UAutomizer [16], an interpolation-based program verifier that won the SV-COMP'17 competition. Fig. 8(d) depicts the comparison using 135 benchmarks in the *loop-lit*, *loop-invgen* and *recursive-∗* categories of SV-COMP [39]. Our solver was able to verify 126 of the total 135 benchmarks, compared to UAutomizer's 111. In the table below we characterize some of the programs that UAutomizer times-out on that were solvable using LinearArbitrary.

	$\#\mathcal{C}$	$\#\mathcal{P}$	$\#\mathcal{V}$	$\#\mathcal{S}$	$\#\mathcal{A}$	T
Prime	21	10	99	261	11,13,15,12,14,15	18s
EvenOdd	8	4	31	541	4,6,6,6,6	105s
recHanoi3	12	6	22	9	4	0.4s
Fib2calls	12	6	53	630	2,8,8,12,9,10,7,4	168s

For example, program Prime verifies that $\forall f_1, f_2, n.\ (f_1 > 1 \land f_2 > 1 \land mult(f_1, f_2) = n) \Rightarrow \neg isPrime(n)$, i.e., n is not prime in the case. The generated CHCs contain 21 constraints over 10 unknown predicate symbols, and 99 variables, requiring 261 samples (from SMT calls) that could nonetheless be verified in 18 seconds using our toolchain. The complex structure of the program, however, makes the interpolation queries generated by UAutomizer costly, resulting in a timeout. EvenOdd and Fib2calls are complex because they have nested recursions, with EvenOdd requiring reasoning over mod operations not expressible in the Polyhedra domain.

We study the scalability of our CHC solver using several large SV-COMP benchmarks taken from the *NTDriver*, *Product-lines*, *Psyco* and *Systemc* categories.[8] Results in terms of the number of verified benchmarks are given below for the 644 programs we were able to verify within the time bound, out of the 679 total programs considered. As a comparison, UAutomizer was able to solve 403 of these programs.

Total	NTDriver (#10)	Product (#597)	Psyco (#10)	Systemc (#62)
UAutomizer	7	357	8	31
LinearArbitrary	9	589	6	40

We characterize some of these sample programs below (#L denotes the number of lines of a program). Many of these programs, although large, have disjunctive invariants that are easy to learn; for example "parport" although sizable at 10KLOC, required only 65 samples, and was able to be verified in 13 seconds.

	#L	$\#\mathcal{C}$	$\#\mathcal{P}$	$\#\mathcal{V}$	$\#\mathcal{S}$	$\#\mathcal{A}$	T
sfifo	309	32	10	292	926	12,12,13	350s
acclrm	842	8	4	8266	26	2, 7, 7	15s
elevator	3405	57	16	880	817	18	18s
parport	10012	275	59	4201	65	1,2	13s

7 Related Work

Machine Learning Based Invariant Generation. Some machine learning-based approaches learn over a fixed space of invariants chosen in advance either by bounding the structure of discovered formulae, or restricting the search space to some finite sub-lattice of an abstract domain. For example,

[8] We used these benchmarks because they can be verified without encoding heap properties, functionality our tool currently does not support. The timeout parameter was relaxed to 1000s for these large programs.

given invariant templates, randomized searches such as random walks are used in [35] to find a valid configuration of template parameters to fit samples; constraint and equation solving are applied in [10, 27, 36, 40] to iterate over Boolean structures and/or coefficient ranges used within templates; and max-plus algebra is used in [28] to find a restricted form of disjunctive invariants. As the search space is constrained, such approaches can find expressive invariants beyond the Polyhedra domain such as polynomial invariants.

Other approaches can learn invariants in the form of arbitrary Boolean combinations of linear inequalities but have to restrict themselves to a limited abstract domain such as the Octagon domain. For example, a greedy set cover algorithm is used in [37] and a decision tree learner is applied in [3, 11, 20, 33]. The ICE learning framework [3, 11], although strongly convergent, needs a set of implication counterexamples [11], which require a nontrivial effort to redevelop existing machine learning algorithms. Other than one recent extension [3], the ICE framework cannot deal with *non-linear* Horn clauses such as CHC (7) in Sec. 2.3, which are important to model recursive programs. Such a CHC has more than one unknown predicate that are related in its body and its counterexample does not follow the form of implication samples required by [11].

To search from a more generous abstract domain, syntax-guided invariant synthesis is applied in [12, 29] and has the potential to cover invariants over various domains, such as the domain of the Z3 string theory, which are currently not implemented in LinearArbitrary. However, the search procedure in [29] is based on enumeration and is less effective than our machine-learning-driven approach in the infinite Polyhedra domain. SVM classification is used in [21, 38] but the techniques are not suitable to find invariants that have arbitrary Boolean structure even if they exist in the theory of linear arithmetic. Consequently, we found that these algorithms produced overfitted feature predicates on our benchmarks and thus often failed verification.

In contrast to these efforts, our approach uses machine learning to discover arbitrarily shaped invariants from the Polyhedra domain, as well as in related and somewhat richer domains (e.g., mod operations).

CHC Solvers. Many advanced CHC solvers for different classes of Constrained Horn Clauses (e.g. for loop programs only or for recursive programs in general) have been developed in recent years. These techniques are invariably designed to satisfy a *bounded safety* criterion - given a safety property φ and a bound, determine whether all unwindings of a CHC system under the bound satisfy φ. The bound is iteratively increased until the proof of bounded safety becomes inductive independent of the bound.

Some CHC solvers [1, 13, 16, 23–25, 32] explicitly unwind a CHC system $\mathcal{H}$. These techniques are based on a combination of Bounded Model Checking [5] and Craig Interpolation [22], attempting to solve $\mathcal{H}$ by generating and solving a series of bounded unwindings of $\mathcal{H}$. Such acyclic unwindings can be solved by applying an interpolating SMT solver to counterexamples to over-approximate unknown predicates. Other solvers implicitly unwind a CHC system. For example, GPDR [17] follows the approach of IC3 [2] by solving Bounded Model Checking incrementally without unrolling a CHC system. It is a bidirectional search that composes the forward image calculation of the solution of an unknown predicate with guidance from suspected counterexamples. Spacer [19] simultaneously maintains the overapproximation and underapproximation of an unknown predicate symbol for a bounded safety proof. The overapproximation can block spurious counterexample while the use of underapproximation effectively avoids inlining the analysis.

Our solver is in line with the aforementioned approaches by generating positive samples from implicit unwindings of a CHC system. It relies on lightweight machine learning algorithms to generalize bounded safety as opposed to the use of interpolating SMT solvers, thereby shifting the burden of invariant discovery to a generic machine learning toolchain.

8 Future Work and Conclusions

Our tool LinearArbitrary currently takes CHCs encoded using linear arithmetic but our approach is sufficiently general so that it can be extended to support richer domains as long as Z3 can provide sound counterexamples (supplying samples for Algorithm 3). For example, to search nonlinear invariants, like DIG [27], we could add monomials over program variables up to a fixed degree as additional features to Algorithm 1. We could also include uninterpreted functions (*e.g.,* tree height) as features, provided that such functions are encoded in CHCs. By additionally supplying reachability predicates quantified over data structure nodes [40], we should be able to verify universally quantified data structure properties. Such extensions are topics for future work.

In this paper, we present a new learning-based algorithm that can verify recursive programs within an unbounded search space of invariants. The key idea is to apply a machine learning tool chain that can discover invariants with arbitrary Boolean combinations drawn from the Polyhedra domain. Efficiency and accuracy are achieved by incorporating techniques to combat over- and under-fitting, and leveraging a CEGAR approach to automatically sample CHC systems. Experimental results demonstrate that our solver complements existing CHC solvers and outperforms state-of-the-art learning based invariant inference techniques.

Acknowledgments

We thank our shepherd Rahul Sharma and the anonymous reviewers for their comments and suggestions. The third author was funded in part by the National Science Foundation under Grant No. CCF-SHF 1717741 and the Air Force Research Lab under Grant No. FA8750-17-1-0006.

References

[1] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. 2012. Whale: An Interpolation-based Algorithm for Inter-procedural Verification. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'12)*. Springer-Verlag, Berlin, Heidelberg, 39–55.

[2] Aaron R. Bradley. 2011. SAT-based Model Checking Without Unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*. Springer-Verlag, Berlin, Heidelberg, 70–87.

[3] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2018. ICE-based Refinement Type Discovery for Higher-Order Functional Programs. In *Proceedings of the Theory and Practice of Software, 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'18)*. Springer-Verlag New York, Inc., New York, NY, USA.

[4] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A Library for Support Vector Machines. *ACM Trans. Intell. Syst. Technol.* 2, 3, Article 27 (May 2011), 27 pages.

[5] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded Model Checking Using Satisfiability Solving. *Form. Methods Syst. Des.* 19, 1 (July 2001), 7–34.

[6] Benjamin Cosman and Ranjit Jhala. 2017. Local Refinement Typing. *Proc. ACM Program. Lang.* 1, ICFP, Article 26 (Aug. 2017), 27 pages.

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[8] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications (OOPSLA '13)*. ACM, New York, NY, USA, 443–456.

[9] Yoav Freund and Robert E. Schapire. 1999. Large Margin Classification Using the Perceptron Algorithm. *Mach. Learn.* 37, 3 (Dec. 1999), 277–296.

[10] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Learning Framework for learning Invariants. In *Proceedings of the 26th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag New York, Inc., New York, NY, USA, 69–87.

[11] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 499–512.

[12] Timon Gehr, Dimitar Dimitrov, and Martin T. Vechev. 2015. Learning Commutativity Specifications. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, Proceedings, Part I*. Springer-Verlag New York, Inc., New York, NY, USA, 307–323.

[13] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing Software Verifiers from Proof Rules. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 405–416.

[14] Ashutosh Gupta and Andrey Rybalchenko. 2009. InvGen: An Efficient Invariant Generator. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 634–640.

[15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, Proceedings, Part I*. Springer-Verlag New York, Inc., New York, NY, USA, 343–361.

[16] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2010. Nested Interpolants. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 471–482.

[17] Kryštof Hoder and Nikolaj Bjørner. 2012. Generalized Property Directed Reachability. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*. Springer-Verlag, Berlin, Heidelberg, 157–171.

[18] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. 2016. JayHorn: A Framework for Verifying Java programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, Proceedings, Part I*. Springer-Verlag New York, Inc., New York, NY, USA, 352–358.

[19] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *Proceedings of the 26th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag New York, Inc., New York, NY, USA, 17–34.

[20] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. 2015. Learning Invariants using Decision Trees. http://cs.nyu.edu/~siddharth/invariants_dt.pdf.

[21] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. 2017. Automatic Loop-invariant Generation and Refinement Through Selective Sampling. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 782–792.

[22] Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 1–13.

[23] Kenneth L. McMillan. 2006. Lazy Abstraction with Interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*. Springer-Verlag, Berlin, Heidelberg, 123–136.

[24] Kenneth L. Mcmillan. 2014. Lazy Annotation Revisited. In *Proceedings of the 26th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag New York, Inc., New York, NY, USA, 243–259.

[25] K. L. McMillan and A. Rybalchenko. 2013. Computing Relational Fixed Points Using Interpolation. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/MSR-TR-2013-6.pdf.

[26] LinearArbitrary. 2018. https://github.com/GaloisInc/LinearArbitrary-SeaHorn/.

[27] ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-guided Approach to Finding Numerical Invariants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 605–615.

[28] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. Using Dynamic Analysis to Generate Disjunctive Invariants. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 608–619.

[29] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 42–56.

[30] John C. Platt. 1999. Advances in Kernel Methods. MIT Press, Cambridge, MA, USA, Chapter Fast Training of Support Vector Machines Using Sequential Minimal Optimization, 185–208.

[31] J. Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[32] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. 2013. Disjunctive Interpolants for Horn-clause Verification. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV'13)*. Springer-Verlag, Berlin, Heidelberg, 347–363.

[33] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. 2008. Dynamic Inference of Likely Data Preconditions over Predicates by Tree Learning. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 295–306.

[34] C. E. Shannon. 2001. A Mathematical Theory of Communication. *SIGMOBILE Mob. Comput. Commun. Rev.* 5, 1 (Jan. 2001), 3–55.

[35] Rahul Sharma and Alex Aiken. 2014. From Invariant Checking to Invariant Inference Using Randomized Search. In *Proceedings of the 26th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag New York, Inc., New York, NY, USA, 88–105.

[36] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013. A Data Driven Approach for Algebraic Loop Invariants. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems (ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 574–592.

[37] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. 2013. Verification as Learning Geometric Concepts. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 388–411.

[38] Rahul Sharma, Aditya V. Nori, and Alex Aiken. 2012. Interpolants As Classifiers. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*. Springer-Verlag, Berlin, Heidelberg, 71–87.

[39] SV-COMP. 2017. http://sv-comp.sosy-lab.org/2017/.

[40] He Zhu, Gustavo Petri, and Suresh Jagannathan. 2016. Automatically Learning Shape Specifications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 491–507.