Horn-ICE Learning for Synthesizing Invariants and Contracts

P. EZUDHEEN, Indian Institute of Science, India
DANIEL NEIDER, Max Planck Institute for Software Systems, Germany
DEEPAK D'SOUZA, Indian Institute of Science, India
PRANAV GARG, Amazon India, India
P. MADHUSUDAN, University of Illinois at Urbana-Champaign, USA

We design learning algorithms for synthesizing invariants using Horn implication counterexamples (Horn-ICE), extending the ICE learning model. In particular, we describe a decision tree learning algorithm that learns from *non-linear* Horn-ICE samples, works in polynomial time, and uses statistical heuristics to learn small trees that satisfy the samples. Since most verification proofs can be modeled using non-linear Horn clauses, Horn-ICE learning is a more robust technique to learn inductive annotations that prove programs correct. Our experiments show that an implementation of our algorithm is able to learn adequate inductive invariants and contracts efficiently for a variety of sequential and concurrent programs.

CCS Concepts: • **Theory of computation** → **Program verification**; • **Computing methodologies** → **Machine learning**; **Classification and regression trees**;

Additional Key Words and Phrases: Software Verification, Constrained Horn Clauses, Decision Trees, ICE Learning

ACM Reference Format:
P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. 2018. Horn-ICE Learning for Synthesizing Invariants and Contracts. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 131 (November 2018), 25 pages. https://doi.org/10.1145/3276501

1 INTRODUCTION

Synthesizing inductive invariants, including loop invariants, pre/post contracts for functions, and rely-guarantee contracts for concurrent programs, is one of the most important problems in program verification. In deductive verification, this is often done manually by the verification engineer, and automating invariant synthesis can significantly reduce the burden of building verified software, allowing the engineer to focus on the more complex specification and design aspects of the code.

There are several techniques for finding inductive invariants, including abstract interpretation [Cousot and Cousot 1977], predicate abstraction [Ball et al. 2001], interpolation [Jhala and McMillan 2006; McMillan 2003], and IC3 [Bradley 2011]. These techniques are typically *white-box* techniques that carefully examine the program, evaluating it symbolically or extracting unsatisfiable

Authors' addresses: P. Ezudheen, CSA Department, Indian Institute of Science, C V Raman Avenue, Bangalore, 560012, India, ezudheen@gmail.com; Daniel Neider, Max Planck Institute for Software Systems, Paul-Ehrlich-Str. 26, Kaiserslautern, 67663, Germany, neider@mpi-sws.org; Deepak D'Souza, CSA Department, Indian Institute of Science, C V Raman Avenue, Bangalore, 560012, India, deepakd@iisc.ac.in; Pranav Garg, Amazon India, Bangalore, 560012, India, pranav.garg2107@gmail.com; P. Madhusudan, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA, madhu@cs.uiuc.edu.

2475-1421/2018/11-ART131
https://doi.org/10.1145/3276501

cores from proofs of unreachability of error states in bounded executions in order to synthesize an inductive invariant that can prove the program correct.

A new class of *black-box* techniques based on learning has emerged in recent years to synthesize inductive invariants [Garg et al. 2014, 2016]. In this technique, there are two distinct agents, the Learner and the Teacher. In each round the Learner proposes an invariant for the program, and the Teacher, with access to a verification engine, checks whether the invariant proves the program correct. If not, it synthesizes *concrete* counterexamples that witness why the invariant is inadequate and sends it back to the learner. The learner takes all such *samples* the teacher has given in all the rounds to synthesize the next proposal for the invariant. The salient difference in the black-box approach is that the Learner synthesizes invariants from concrete sample configurations of the program, and is otherwise oblivious to the program or its semantics.

It is tempting to think that the learner can learn invariants using positively and negatively labeled configurations, similar to machine learning. However, Garg et al. [2014] argued that we need a richer notion of samples for robust learning of inductive invariants. Let us recall this simple argument.

Consider a system with variables $\vec{x}$, with initial states captured by a predicate $Init(\vec{x})$, and a transition relation captured by a predicate $Trans(\vec{x}, \vec{x}')$, and assume we want to prove that the system does not reach a set of bad/unsafe states captured by the predicate $Bad(\vec{x})$. An inductive invariant $I(\vec{s})$ that proves this property needs to satisfy the following three constraints:

(1) $\forall \vec{x}. Init(\vec{x}) \Rightarrow I(\vec{x})$;
(2) $\forall \vec{x}.\ \neg(I(\vec{x}) \wedge Bad(\vec{x}))$; and
(3) $\forall \vec{x}, \vec{x}'. I(\vec{x}) \wedge Trans(\vec{x}, \vec{x}') \Rightarrow I(\vec{x}')$.

When a proposed invariant fails to satisfy the first two conditions, the verification engine can indeed come up with configurations labeled positive and negative to indicate ways to correct the invariant. However, when the third property above fails, it cannot come up with a single configuration labeled positive/negative; and the most natural counterexample is a *pair* of configurations c and c', with the instruction to the learner that if $I(c)$ holds, then $I(c')$ must also hold. These are called *implication counterexamples* and the ICE (Implication Counter-Example) learning framework developed by Garg et al. [2014] is a robust learning framework for synthesizing invariants. Garg et al. [2014, 2016] devised several learning algorithms for learning invariants, in particular learning algorithms based on decision trees that can learn Boolean combinations of Boolean predicates and inequalities that compare numerical terms to arbitrary thresholds.

Despite the argument above, it turns out that implication counterexamples are *not sufficient* for learning invariants in program verification settings. This is because reasoning in program verification is more stylized, to deal *compositionally* with the program. In particular, programs with function calls and/or concurrency are not amenable to the above form of reasoning. In fact, it turns out that most reasoning in program verification can be expressed in terms of *Horn clauses*, where the Horn clauses contain some formulas that need to be synthesized.

For example, consider the imperative program snippet

$$I_{pre}(\vec{x}, y)\ S(\text{mod}\ \vec{x});\ \ y := foo(\vec{x});\ \ I_{post}(\vec{x}, y),$$

which we want to show correct, where S is some straight-line program that modifies $\vec{x}$, I_{pre} and I_{post} are some annotation (like the contract of a function we are synthesizing). Assume that we are synthesizing the contract for *foo* as well, and assume the post-condition for *foo* is $PostFoo(res, \vec{x})$, where *res* denotes the result it returns. Then the verification condition that we want to be valid is

$$\big(I_{pre}(\vec{x}, y) \wedge Trans_S(\vec{x}, \vec{x}') \wedge PostFoo(y', \vec{x}')\big) \Rightarrow I_{post}(\vec{x}', y'),$$

where $Trans_S$ captures logically the semantics of the snippet S in terms of how it affects the post-state of $\vec{x}$.

In the above, all three of the predicates I_{pre}, I_{post} and *PostFoo* need to be synthesized. When a learner proposes concrete formulas for these, the verifier checking the above logical formula may find it to be invalid, and find concrete valuations $v_{\vec{x}}, v_y, v_{\vec{x}'}, v_{y'}$ for $\vec{x}, y, \vec{x}', y'$ that makes the above implication false. However, notice that the above cannot be formulated as a simple implication constraint. The most natural constraint to return to the learner is

$$\left(I_{pre}(v_{\vec{x}}, v_y) \wedge PostFoo(v_{y'}, v_{\vec{x}'})\right) \Rightarrow I_{post}(v_{\vec{x}'}, v_{y'}),$$

asking the learner to meet this requirement when coming up with predicates in the future. The above is best seen as a (non-linear) *Horn* Implication CounterExample (Horn-ICE). (Plain implication counterexamples are *linear* Horn samples.)

The primary goal of this paper is to build Horn-ICE (Horn implication counterexample) learners for learning predicates that facilitate inductive invariant and contract synthesis that prove safety properties of programs. It has been observed in the literature that most program verification mechanisms can be stated in terms of proof rules that resemble Horn clauses [Grebenshchikov et al. 2012]; in fact, the formalism of *constrained Horn clauses* (CHC) has emerged as a robust general mechanism for capturing program verification problems in logic [Gurfinkel et al. 2015]. Consequently, whenever a Horn clause fails, it results in a Horn-ICE sample that can be communicated to the learner, making Horn-ICE learners a much more general mechanism than ICE for synthesizing invariants and contracts.

Our main technical contribution is to devise a decision tree-based Horn-ICE algorithm. Given a set of (Boolean) predicates over configurations of programs and numerical functions that map configurations to integers, the goal of the learning algorithm is to synthesize predicates that are *arbitrary* Boolean combinations of the Boolean predicates and atomic predicates of the form $n \leq c$, where n denotes a numerical function, and where c is arbitrary. The classical decision tree learning algorithm by Quinlan [1986] learns such predicates from samples labeled +/− only, and the work by Garg et al. [2016] extends decision tree learning to learning from ICE samples. In this work, we extend the latter algorithm to one that learns from Horn-ICE samples.

Extending decision tree learning to handle Horn samples turns out to be non-trivial. When a decision tree algorithm reaches a node that it decides to make a leaf and label it *true*, in the ICE learning setting it can simply *propagate* the labels across the implication constraints. However, it turns out that for Horn constraints, this is much harder. Assume there is a single invariant we are synthesizing and we have a Horn sample $(s_1 \wedge s_2) \Rightarrow s'$ and we decide to label s' false when building the decision tree. Then we must later turn *at least one of* s_1 *and* s_2 *to false*. This choice makes the algorithms and propagation much more complex, and ensuring that the decision tree algorithm will always construct a correct decision tree (if one exists) and works in polynomial time becomes much harder. Furthermore, statistical measures based on entropy for choosing attributes (to split each node) get more complicated as we have to decide on a more complex logical space of Horn constraints between samples.

The contributions of this paper are the following:

(1) A robust decision tree learning algorithm that learns using Horn implication counterexamples, runs in polynomial time (in the number of samples) and has a bias towards learning small trees (expressions) using statistical measures for choosing attributes.
(2) We show that algorithm guarantees that a decision tree consistent with all samples is constructed, provided there exists one. An incremental maintenance of Horn constraints during tree growth followed by an *amortized analysis* over the construction of the tree gives us an efficient algorithm.

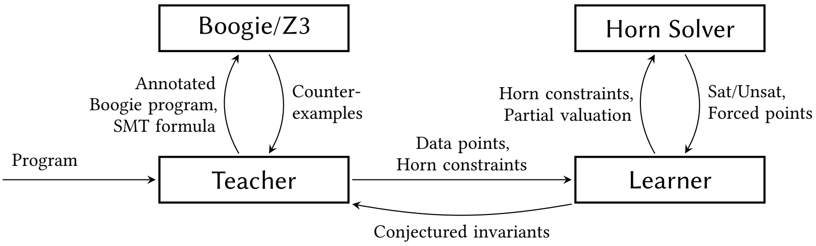


Fig. 1. Architecture of the Horn-ICE invariant synthesis tool

(3) We show that we can use our learning algorithm to learn over an *infinite* countable set of predicates $\mathcal{P}$, and we can ensure learning is complete (i.e., that will find an invariant if one is expressible using the predicates $\mathcal{P}$).
(4) An implementation of our algorithm and an automated verification tool built with our algorithm over the Boogie framework for synthesizing invariants. We evaluate our algorithm for finding loop invariants and summaries for sequential programs and also Rely-Guarantee contracts in concurrent programs.

The paper is structured as follows. In Sec. 2 we present an overview of Horn-ICE invariant synthesis; in Sec. 3 we describe the decision tree based algorithm for learning invariant formulas from Horn-ICE samples; in Sec. 4, we describe the algorithm that propagates the data point classifications across Horn constraints; we describe the node/attribute selection strategies used in the decision tree based learning algorithm in Sec. 5 and the experimental evaluation in Sec. 6. We discuss related work in Sec. 7, and conclude with directions for future work in Sec. 8.

2 OVERVIEW

In this section, we first argue the need for building learners that work with non-linear Horn-ICE examples and then give an example of how our Horn-ICE invariant synthesis framework works on a particular example. Fig. 1 shows the main components of our Horn-ICE invariant synthesis framework. The Teacher has a program specification she would like to verify. Based on the style of proof, she determines the kind of invariants needed (a name for each invariant, and the set of terms it may mention) and the corresponding verification conditions (VCs) they must satisfy. The Learner conjectures a concrete invariant for each invariant name, and communicates these to the Teacher. The Teacher plugs in these conjectured invariants and asks a verification engine (in this case Boogie, but we could use any suitable program verifier) to check if the conjectured invariants suffice to prove the specification. If not, Boogie returns a counterexample showing why the conjectured invariants do not constitute a valid proof. The Teacher passes these counterexamples to the Learner. The Learner now learns new invariants that are consistent with the set of counterexamples given by the Teacher so far. The Learner frequently invokes the Horn Solver to guide it in the process of building concrete invariants that are consistent with the set of counterexamples given by the Teacher. The Teacher and Learner go through a number of such rounds, until the Teacher finds that the invariants supplied by the Learner constitute a valid proof.

2.1 The Need for Nonlinear Horn-ICE Learning

Several authors of this paper were also authors of the original ICE learning framework [Garg et al. 2014]. The present work came about in our realization that implication counterexamples are just not sufficient. It is now commonly accepted that constrained Horn clauses are the right formulation

```
main() {
  result = fib(5);
  assert (result > 2);
}
```

```
int fib(int x) {
  if (x < 2) return 1;
  else return fib(x-1) + fib(x-2);
}
```

Fig. 2. A sequential program with a recursive function

for most verification tasks. This includes programs with function calls (that cannot be inlined, say due to recursion, or due to deep nesting) and concurrent programs. Learning to solve these clauses gives rise to Horn-implication counterexamples naturally.

It may be tempting to think that vanilla ICE learning (i.e., learning using linear Horn-ICE samples) is sufficient for learning invariants and contracts for sequential and concurrent programs. Let us now see why non-linear Horn-ICE learning is necessary by considering the synthesis of pre/post contracts and inductive invariants for loops in a sequential program with several functions $f_1, f_2, \ldots, f_k$.

If we were given (say by the user) inductive pre/post contracts for all functions, we can use ICE learning to synthesize the required loop invariants, and would not need to deal with non-linear Horn-ICE samples. However, we assume a completely automated verification setting where such contracts are not given and need to be synthesized as well.

First, notice that a completely bottom-up modular approach for synthesizing contracts is hard. Consider one of the functions, say *foo*. Let us even assume that *foo* is a leaf function that does not call other functions. It is still hard to figure out what contract to generate for *foo* without looking at its clients. There are many trivial contracts (e.g., $\{true\}$ *foo* $\{true\}$) and some that may be too trivial for use (e.g., $\{x > 0\}$ *foo* $\{result > 0\}$ may be useless for the client to prove its assertions). There may be no "most useful" contract or, even if it exists, it may be inexpressible in the logic or too expensive and unnecessary for the program's verification. The correctness properties being proved (i.e., the assertions stated across the program) should somehow dictate the granularity of the contracts.

In our setup, invariants are synthesized simultaneously but verification is modular (i.e., the VCs are local to a process/function) and the configurations learned from them are local. One can in fact think of our solution as a simultaneous synthesis of contracts for all functions (in the sequential program setting), where the synthesis engines communicate and are simultaneously constrained through Horn clauses.

For example, consider the following program in Fig 2 adapted from one of the SVComp benchmarks and found in our benchmark suite.

For proving the assertion in `main`, we do not need the most precise contract for `fib`. However, when looking only at `fib`, it is hard to predict what the client would need of its behavior. The contract that says *fib*(x) computes a value greater than or equal to x is sufficient, but is not inductive. Our tool generates an invariant equivalent to

```
@pre: x >= 0 ; @post: result >= x && result > 0
```

which is inductive and proves the program correct. The communication between the client `main` and the function `fib`, and between the contracts in `fib` itself is what Horn implication constraints facilitate, allowing `fib`'s contract to adapt to the client's needs.

In fact, Horn implications are needed in many other scenarios too, even such as finding inductive pre/post contracts for a single function that calls itself recursively. For example, take the `fib` function in Fig. 2 with the candidate contract `@pre: x >= 0` and `@post: result >= x`. This contract is not inductive, and an honest teacher (one that does not make arbitrary choices for the learner) would have to return a counterexample of the form:

```
IF pre of fib contains a configuration with x=2 AND post contains a
configuration with (x=0, result=0) and another with (x=1, result=1),
THEN post must contain the configuration (x=2, result=1).
```

This counterexample takes the form of a non-linear Horn implication counterexample, and cannot be represented as an ICE counterexample. It relates one configuration in the precondition and two configurations in the postcondtion, all satisfying the current conjectured contract, to a configuration in the postcondition that does not satisfy the current contract.

2.2 An Illustrative Example

We now illustrate how our Horn-ICE learning framework works on a concrete example, in this case a concurrent program for which we want to synthesize Rely-Guarantee contracts.

Fig. 3a shows a concurrent program, adapted from Miné [2014], with two threads T_1 and T_2 that access two shared variables x and y. The precondition expresses that initially $x = y = 0$ holds. The postcondition asserts that when the two threads terminate, the state satisfies $x \leq y$. For the Teacher to obtain a Rely-Guarantee proof for this program [Jones 1983; Xu et al. 1997], she needs to come up with invariants associated with the program points $P0$–$P4$ in T_1 and $Q0$–$Q4$ in T_2 (as in Floyd-Hoare proofs of sequential programs), as well as two "two-state" invariants $G1$ and $G2$, which act as the "guarantee" on the interferences caused by each thread respectively. Fig. 3b shows a partial list of the VCs that the invariants need to satisfy, in order to constitute a valid Rely-Guarantee proof of the program. The VCs are grouped into four categories: "Adequacy" and "Inductiveness" are similar to the requirements for sequential programs, while "Guarantee" requires that the atomic statements of each thread satisfy its promised guarantee, and "Stability" requires that the invariants at each point are stable under interferences from the other thread. We use the notation "$[\![$x := x+1$]\!]$" to denote the two-state (or "before-after") predicate describing the semantics of the statement "x := x+1", which in this case is the predicate $x' = x + 1 \wedge y' = y$. The guarantee invariants $G1$ and $G2$ are predicates over the variables x, y, x', y', describing the possible state changes that an atomic statement in a thread can effect, while $P0$–$P4$ and $Q0$–$Q4$ are predicates over the variables x, y. By $P0'$ we mean the predicate $P0$ applied to the variables x', y'.

```
                  Pre: x = y = 0

         T1           ||            T2

P0  while (*) {             Q0  while (*) {
P1    if (x < y)            Q1    if (y < 10)
P2      x := x + 1;         Q2      y := y + 3
P3  }                       Q3  }
P4                          Q4

                  Post: x <= y
```

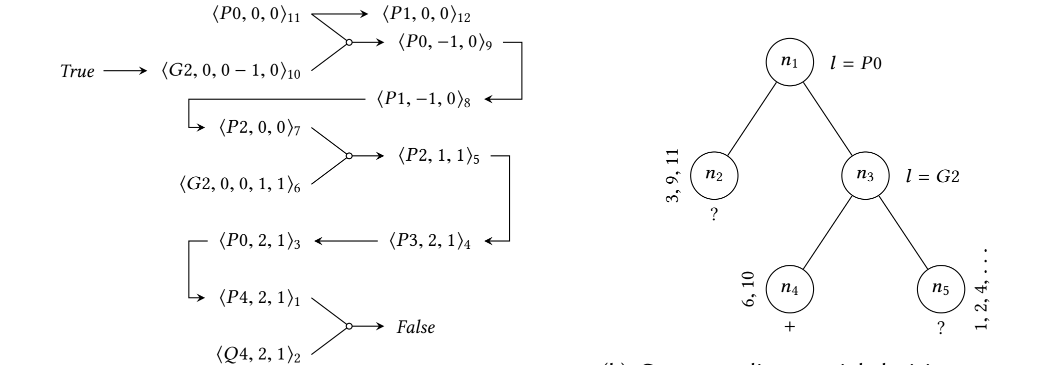
(a) The program

Adequacy	Inductiveness
1. $(x = 0 \wedge y = 0) \rightarrow P0$	1. $P0 \rightarrow P1 \wedge P4$
2. $P4 \wedge Q4 \rightarrow (x \leq y)$	2. $P1 \wedge (x < y) \rightarrow P2$
	3. $P2 \wedge [\![\texttt{x := x + 1}]\!] \rightarrow P3'$
	4. $P3 \rightarrow P0$
	...
Stability	**Guarantee**
1. $P0 \wedge G2 \rightarrow P0'$	1. $P2 \wedge [\![\texttt{x := x + 1}]\!] \rightarrow G1$
2. $P1 \wedge G2 \rightarrow P1'$	2. $Q2 \wedge [\![\texttt{y := y + 3}]\!] \rightarrow G2$
...	

(b) The verification conditions

Fig. 3. A concurrent program and corresponding Rely-Guarantee VCs.

The Teacher asks the Learner to synthesize the invariants $P0$–$P4$ and $Q0$–$Q4$ over the variables x, y, and $G1$ and $G2$ over the variables x, y, x', y'. As a first cut the Learner conjectures "*true*" for all these invariants. The Teacher encodes the VCs in Fig. 3b as annotated procedures in Boogie's programming language, plugs in *true* for each invariant, and asks Boogie if the annotated program verifies. Boogie comes back saying that the ensures clause corresponding to VC Adequacy-2 may fail, and gives a counterexample, say $\langle x \mapsto 2, y \mapsto 1\rangle$, which satisfies $P4$ and $Q4$, but not $x \leq y$. The Teacher returns this counterexample as a Horn sample $d_1 \wedge d_2 \rightarrow \mathit{false}$, where d_1 is the data



(a) Horn constraints given by the Teacher

(b) Corresponding partial decision tree produced by the learner

Fig. 4. Intermediate results of our framework on the introductory example.

point $\langle P4, 2, 1\rangle$ and d_2 is the data point $\langle Q4, 2, 1\rangle$. We use the convention that the data points are vectors in which the first component is the value of a "location" variable "l" which takes one of the values "$P0$", "$P1$", etc, while the second and third components are values of x and y respectively. This Horn constraint is shown in the bottom of Fig. 4a.

To focus on the technique used by the Learner let us pan to several rounds ahead, where the Learner has accumulated a set of counterexamples given by the Teacher, shown in Fig. 4a. The Learner's goal is simply to find a small (in expression size) invariant φ that is *consistent* with the given set of Horn constraints, in that for each Horn constraint of the form $d_1 \wedge \cdots \wedge d_k \rightarrow d$, whenever each of $d_1, \ldots, d_k$ satisfy φ, it is the case that d also satisfies φ.

Our Learner uses a decision tree based learning technique. Here the internal nodes of the decision tree are labeled by the base predicates (or "attributes") and leaf-nodes are classified as "True", "False", or "?" (for "Unclassified", which happens during construction). Each leaf node in a decision tree represents a logical formula which is the conjunction of the node labels along the path from the root to the leaf node, and the whole tree represents the disjunction of the formulas corresponding to the leaf nodes labeled "True". The Learner builds a decision tree for a given set of Horn constraints incrementally, starting from the root node. Each leaf node in the tree has a corresponding subset of the data-points associated with it, namely the set of points that satisfy the formula associated with that node. In each step the Learner can choose to mark a node as "True", or "False", or to split a node with a chosen attribute and create two child nodes associated with it.

Before marking a node as "True" or "False" the Learner would like to ensure that the consistency of the decision tree with respect to the Horn constraints is preserved. For this he calls the Horn Solver, which reports whether the proposed extension of the partial valuation is indeed consistent with the given set of Horn constraints, and if so which are the data-points that are "forced" to be true or false. For example, let us say the Learner has constructed the partial decision tree shown in Fig. 4b, where node n_4 has already been set to "True" and nodes n_2 and n_5 are unclassified. He now asks the Horn Solver if it is okay for him to turn node n_2 "True", to which the Horn Solver replies "Yes" since this extended valuation would still be consistent with the set of Horn constraints in Fig. 4a. The Horn Solver also tells him that the extension would force the data-points d_{12}, d_8, d_7, d_5, d_4, d_3, d_1 to be *true*, and the point d_2 to *false*.

The Learner uses this information to go ahead and set n_2 to "True", and also to make note of the fact that n_5 is now a "mixed" node with some points that are forced to be *true* (like d_1) and some *false* (like d_2). Based on this information, the Learner may choose to split node n_5 next. After completing the decision tree, the Learner may send the conjecture in which $P0$–$P4$, $G1$ and $G2$ are set to *true*, and $Q1$–$Q4$ are set to *false*. The Teacher sends back another counterexample to this conjecture, and the exchanges continue. Finally, our Learner would make a successful conjecture like: $x \leq y$ for $P0$, $P1$, $P3$, $P4$, and $Q0$–$Q4$; $x < y$ for $P2$; $y = y' \wedge x' \leq y'$ for $G1$; and $x = x' \wedge y \leq y'$ for $G2$.

3 DECISION TREE LEARNING WITH HORN CONSTRAINTS

In this section describe our decision-tree based learning algorithm that works in the presence of non-linear Horn samples. We begin with some preliminary definitions.

3.1 Preliminary Definitions

Valuations and Horn Constraints. We will consider propositional formulas over a fixed set of propositional variables X, using the usual Boolean connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, etc. The *data points* we introduce later will also play the role of propositional variables. A *valuation* for X is a map $v : X \rightarrow \{true, false\}$. A given formula over X evaluates, in the standard way, to either *true* or *false* under a given valuation. We say a valuation v *satisfies* a formula φ over X, written $v \models \varphi$, if φ evaluates to *true* under v.

A *partial* valuation for X is a partial map $u : X \rightharpoonup \{true, false\}$. We denote by $dom_{true}(u)$ the set $\{x \in X \mid u(x) = true\}$ and $dom_{false}(u)$ the set $\{x \in X \mid u(x) = false\}$. We say a partial valuation u is *consistent* with a formula φ over X, if there exists a full valuation v for X, which extends u (in that for each $x \in X$, $u(x) = v(x)$ whenever u is defined on x), and $v \models \varphi$.

Let φ be a formula over X, and u a partial valuation over X which is consistent with φ. We say a variable $x \in X$ is *forced to be true* by u in φ, if for all valuations v which extend u, whenever $v \models \varphi$ we have $v(x) = true$. Similarly we say x is *forced to be false* by u in φ, if for all valuations v which extend u, whenever $v \models \varphi$ we have $v(x) = false$. We denote the set of variables forced true (by u in φ) by *forced-true*(φ, u), and those forced false by *forced-false*(φ, u). For a partial valuation u over X, and subsets T and F of X, which are disjoint from each other and from the domain of u, we denote by u_F^T the partial valuation extending u by mapping all variables in T to *true* and all variables in F to *false*.

A *Horn clause* (or a *Horn constraint*) over X is disjunction of literals over X with *at most* one positive literal. Without loss of generality, we will write Horn clauses in one of the three forms: (1) $true \rightarrow x$, (2) $(x_1 \wedge \cdots \wedge x_k) \rightarrow false$, or (3) $(x_1 \wedge \cdots \wedge x_l) \rightarrow y$, where $l \geq 1$ and each of the x_i's and y belong to X. A *Horn formula* is a conjunction of Horn constraints.

Data Points. Our decision tree learning algorithm is paired with a teacher that refutes incorrect conjectures with positive data points, negative data points, and, more generally, with Horn constraints over data points. Roughly speaking, a data point corresponds to a program configuration and contains the values of each program variable and potentially values that are derived from the program variables, such as $x + y$, x^2, or $is\_list(z)$. For the sake of a simpler presentation, however, we assume that a data point is an element $d \in \mathbb{D}$ of some (potentially infinite) abstract domain of data points $\mathbb{D}$ (we encourage the reader to think of programs over integers where data points correspond to vectors of integers).

Base Predicates and Decision Trees. The aim of our learning algorithm is to construct a decision tree representing a Boolean combination of some base predicates. We assume a set of base predicates, each of which evaluates to *true* or *false* on a data point $d \in D$. More precisely, a *decision tree* is a finite binary tree $\mathcal{T}$ whose nodes either have two children (*internal nodes*) or no children (*leaf nodes*), whose internal nodes are labeled with base predicates, and whose leaf nodes are labeled with *true* or *false*. The formula $\psi_{\mathcal{T}}$ corresponding to a decision tree $\mathcal{T}$ is defined to be $\bigvee_{\pi \in \Pi_{true}} (\bigwedge_{\rho \in \pi} \rho)$ where Π_{true} is the set of all paths from the root of $\mathcal{T}$ to a leaf node labeled *true*, and $\rho \in \pi$ denotes that the base predicate ρ occurs as a label of a node on the path π. Given a set of data points $X \subseteq D$, a decision tree $\mathcal{T}$ induces a valuation $v_{\mathcal{T}}$ for X given by $v_{\mathcal{T}}(d) = true$ iff $d \models \psi_{\mathcal{T}}$. Finally, given a finite set of Horn constraints C over a set of data points X, we say a decision tree $\mathcal{T}$ is *consistent* with C if $v_{\mathcal{T}} \models \bigwedge C$. We will also deal with "partial" decision trees, where some of the leaf nodes are yet unlabeled, and we define a partial valuation $u_{\mathcal{T}}$ corresponding to such a tree $\mathcal{T}$, and the notion of consistency, in the expected way.

Horn Samples. In the traditional setting, the learning algorithm collects the information returned by the teacher as a set of *samples*, comprising "positive" and "negative" data points. In our case, the set of samples will take the form of a finite set of Horn constraints C over a finite set of data points X. We note that a positive point d (resp. a negative point e) can be represented as a Horn constraint $true \rightarrow d$ (resp. $e \rightarrow false$). We call such a pair (X, C) a *Horn sample*.

In each iteration of the learning process, we require the learning algorithm to construct a decision tree $\mathcal{T}$ that agrees with the information in the current Horn sample $\mathcal{S} = (X, C)$, in that $\mathcal{T}$ is consistent with C. The learning task we address then is *"given a Horn sample $\mathcal{S}$, construct a decision tree consistent with $\mathcal{S}$"*.

3.2 The Learning Algorithm

Our learning algorithm, shown in Algorithm 1, is an extension of the algorithm by Garg et al. [2016], which in turn is based on the classical decision tree learning algorithm of Quinlan [1986]. Given a Horn sample (X, C), the algorithm creates an initial (partial) decision tree $\mathcal{T}$ which has a single unlabeled node, whose associated set of data points is X. As an auxiliary data structure, the learning algorithm maintains a partial valuation u, which is always an extension of the partial valuation induced by the decision tree. In each step, the algorithm picks an unlabeled leaf node n (using the Select-Node routine which we describe in Sec. 5), and checks if it is "pure" in that all points in the node are either positive (i.e., *true*) or unsigned, or similarly neg/unsigned. If so, it calls the procedure Label which tries to label it positive if all points are positive or unsigned, or negative if all points are negative or unsigned. To label it positive, the procedure first checks whether extending the current partial valuation u by making all the unsigned data points in n true results in a valuation that is consistent with the given set of Horn constraints C. It does this by calling the classical Horn-Sat procedure of Dowling and Gallier [1984], which checks whether the proposed extension is consistent with C. If so, we call our procedure Horn-Forced (described in the next section), which returns the set of points forced to be true and false, respectively. The node n is labeled *true* and the partial valuation u is extended with the forced values. If the attempt to label positive fails, it tries to label the node negative, in a similar way. If both these fail, it "splits" the node using a suitably chosen base predicate a. The corresponding method Select-Attribute, which (heuristically) aims to obtain a small tree (i.e., a concise formula), is described in Sec. 5.

The crucial property of our learning algorithm is that if the given set of constraints C is satisfiable and if the data points in X are "separable" (as defined below), it will always construct a decision tree consistent with C. We say that the points in X are *separable* if for every pair of points d_1 and d_2 in X we have a base predicate ρ which distinguishes them (i.e., $d_1 \models \rho$ iff $d_2 \not\models \rho$). This result, together with its time complexity, is formalized in Theorem 3.1. Below, by the size of a Horn formula we mean the total number of occurrences of literals in it.

Theorem 3.1. *Consider an input Horn sample (X, C) to Algorithm 1. Let n denote $|X|$, and h the size of C. If the set of points X is separable and the Horn constraints C are satisfiable, then Algorithm 1 runs in time $O(h \cdot n)$ and returns a decision tree that is consistent with the Horn sample (X, C).*

Algorithm 1 Decision Tree Learner for Horn Samples

```
 1: procedure DECISION-TREE-HORN
Input: A Horn sample $(X, C)$
Output: A decision tree $\mathcal{T}$ consistent with $C$, if one exists.
 2:     Initialize tree $\mathcal{T}$ with root node $r$, with $r.dat \leftarrow X$
 3:     Initialize partial valuation $u$ for $X$, with $u \leftarrow \emptyset$
 4:     while ($\exists$ an unlabeled leaf in $\mathcal{T}$) do
 5:         $n \leftarrow$ SELECT-NODE( )
 6:         $result \leftarrow false$
 7:         if (pure($n$)) then   // $n.dat \cap dom_{true}(u) = \emptyset$ or $n.dat \cap dom_{false}(u) = \emptyset$
 8:             $result \leftarrow$ LABEL($n$)
 9:         if ($\neg$pure($n$) $\vee$ $\neg result$) then
10:             if ($n.dat$ is singleton) then
11:                 print "Unable to construct decision tree"; return
12:             $result \leftarrow$ SPLIT-NODE($n$)
13:             if ($\neg result$) then
14:                 print "Unable to construct decision tree"; return
15:     return $\mathcal{T}$    // decision tree constructed successfully
```

```
 1: procedure LABEL(node $n$)
 2:     $Y \leftarrow n.dat \setminus dom(u)$;
 3:     if ($n.dat \cap dom_{false}(u) = \emptyset$) then
          // $n$ contains only pos/unsigned pts
 4:         if (HORN-SAT($C, u^{Y}_{\emptyset}$)) then
 5:             $n.label \leftarrow true$
 6:             $(T, F) \leftarrow$ HORN-FORCED($C, u^{Y}_{\emptyset}$)
 7:             $u \leftarrow u^{Y \cup T}_{F}$
 8:             return $true$
 9:     if ($n.dat \cap dom_{true}(u) = \emptyset$) then
          // $n$ contains only neg/unsigned pts
10:         ... // try to label neg
11:     return $false$
```

```
 1: procedure SPLIT-NODE(node $n$)
 2:     $(res, a) \leftarrow$ SELECT-ATTRIBUTE($n$)
 3:     if ($res$) then
 4:         Create new nodes $l$ and $r$
 5:         $l.dat \leftarrow \{d \in n.dat \mid d \models a\}$
 6:         $r.dat \leftarrow \{d \in n.dat \mid d \not\models a\}$
 7:         $n.left \leftarrow l$, $n.right \leftarrow r$
 8:         return $true$
 9:     else
10:         return $false$
```

PROOF. At each iteration step, the algorithm maintains the invariant that the partial valuation u is an extension of the partial valuation $u_{\mathcal{T}}$ induced by the current (partial) decision tree $\mathcal{T}$, and is consistent with C. This is because each labelling step is first checked by a call to the Horn-Sat algorithm, and subsequently the HORN-FORCED procedure correctly identifies the set of forced variables, which are then used to update u. It follows that if the algorithm terminates successfully in Line 15, then $u_{\mathcal{T}}$ is a full valuation which coincides with u, and hence satisfies C. The only way the algorithm can terminate unsuccessfully is in Line 11 or Line 14. The first case is ruled out since if $n.dat$ is singleton, and by assumption $u_{\mathcal{T}}$ is consistent with C, we must be able to label the single data point with either *true* or *false* in a way that is consistent with C. The second case is ruled out, since under the assumption of separability the SELECT-ATTRIBUTE procedure will always return a non-trivial attribute (see Sec. 5).

The learning algorithm (Algorithm 1) can be seen to run in time $O(h \cdot n)$. To see this, observe that in each iteration of the loop the algorithm produces a tree that is either the same as the previous

Algorithm 2 Finding variables forced to True and False

```
procedure HORN-FORCED
Input: Horn constraints $C$ over $X$, partial valuation $u$ over $X$ consistent with $C$.
Output: $forced\text{-}true(C, u)$, $forced\text{-}false(C, u)$.
    Add two new variables $True$ and $False$ to $X$. Let $X' = X \cup \{True, False\}$.
    $C' \leftarrow C$ + clauses "$true \rightarrow x$" for each $x$ such that $u(x) = true$ + clauses "$x \rightarrow false$" for each $x$ such that $u(x) = false$.
    Mark variable $True$ with "$*$", and each variable $x \in X$ with "$*x$".
    repeat
        For each constraint $x_1 \wedge \cdots \wedge x_l \rightarrow y$ in $C'$:
        if $x_1, \ldots, x_l$ are all marked "$*$" then mark $y$ with "$*$"
        if $\exists z \in X$ s.t. $x_1, \ldots, x_l$ are all marked "$*z$" or "$*$" then mark $y$ with "$*z$"
    until no new marks can be added
    $P \leftarrow \{x \in X \mid x \text{ is marked "} * \text{"}\}$, $N \leftarrow \{x \in X \mid False \text{ is marked "} *x\text{"}\}$
    $P' \leftarrow P - dom_{true}(u)$, $N' \leftarrow N - dom_{false}(u)$
    return $P'$, $N'$
```

step (but with a leaf node labeled), or splits a leaf to extend the previous tree. At each step we maintain an invariant that the collection of data points in the leaf nodes forms a partition of the input set X. Thus the number of leaf nodes is bounded by n, and hence each tree has a total of at most $2n$ nodes. When the algorithm returns (successfully or unsuccessfully) each node in the final tree has been processed at most once by calling the labeling/splitting subroutines on it. Furthermore, the main work in the subroutines is the calls to the HORN-SAT and HORN-FORCED procedures. Each call to HORN-SAT takes $O(h)$ time, and hence the calls to HORN-SAT totally take $O(h \cdot n)$ time. The calls to HORN-FORCED (which is called only for the leaf-nodes), can be seen to take a *total* of $O(h \cdot n)$ time (see Sec. 4). It follows that Algorithm 1 runs in $O(h \cdot n)$ time. □

If the points in X are not separable, we can follow a similar route to that described by Garg et al. [2016]. For every pair of inseparable points d_1 and d_2 in X, we add the constraints $d_1 \rightarrow d_2$ and $d_2 \rightarrow d_1$ to C, to obtain a new set C' of Horn constraints. Our decision tree algorithm with input (X, C') is now guaranteed to construct a decision tree consistent with C if and only if there exists a valuation describable as a Boolean combination of the base predicates and satisfying C.

Furthermore, we can extend our algorithm to work on an infinite enumerable set of predicates $\mathcal{P}$, and assure that the algorithm will find an invariant if one exists, as done by Garg et al. [2016]. To this end, we start with a finite set $Q \subset \mathcal{P}$, ask whether there is some invariant over Q, and if not grow Q by taking finitely more predicates from $\mathcal{P} \setminus Q$. It is not hard to verify that this strategy is guaranteed to converge to an invariant if one is expressible over $\mathcal{P}$.

4 ALGORITHM FOR FINDING FORCED VARIABLES

A crucial step in our decision tree algorithm when we decide to label a node as true or false is to compute the set of variables forced to be true or false respectively. In this section we describe an efficient algorithm to carry out this step. Our algorithm is an adaptation of the "pebbling" algorithm of Dowling and Gallier [1984] for checking satisfiability of Horn formulas, to additionally find the variables that are forced to be true or false. We begin with a conceptual extension of the pebbling algorithm in Section 4.1 and describe in Section 4.2 how this algorithm can be implemented more efficiently.

4.1 An Improved Pebbling Algorithm for Checking Satisfiability of Horn Formulas

Procedure Horn-Forced in Algorithm 2 shows our procedure for identifying the subset of variables forced to true or false, by a partial valuation for a set of Horn constraints. Intuitively, the standard linear-time algorithm for Horn satisfiability by Dowling and Gallier [1984] in fact already identifies the *minimal* set M of variables that are forced to be true in any satisfying valuation, and assures us that the others can be set to false. However, the other variables are not *forced* to be false. Our algorithm essentially runs another set of SAT problems where each of the other variables are set to *true*; the instance returns SAT if and only if the variable is not forced to be *false*. In our algorithm, variable x being set to *true* is modeled by x being marked "$*x$", and this instance being SAT corresponds to the fact that *False* is *not* marked "$*x$" after all marks have been propagated. Fig. 5 illustrates Algorithm 2. The final marking computed by the procedure is shown above or below each variable. Variables set to *true* in the partial assignment are shown with a "+" above/below them. Variables forced to true (respectively false) are shown with a "(+)" (respectively "(-)") above/below them.

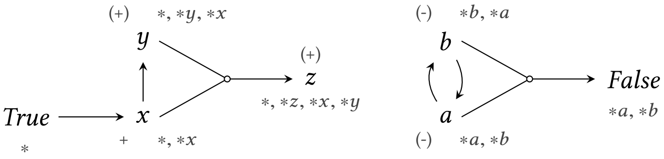


Fig. 5. Example illustrating Algorithm 2. The given set of Horn constraints is $C = \{x \rightarrow y, x \wedge y \rightarrow z, a \rightarrow b, b \rightarrow a, a \wedge b \rightarrow False\}$, and the partial valuation u sets x to true and is undefined elsewhere. The algorithm outputs the forced-true set $P' = \{y, z\}$ and forced-false set $N' = \{a, b\}$.

Theorem 4.1. *Let (X, C) be a Horn sample, u be a partial valuation consistent with C. Then procedure Horn-Forced correctly outputs the set of variables forced true and false respectively by u in C.*

Proof. Let us fix X, C, and u to be the inputs to the procedure, and let X', C', P, N, P' and N' be as described in the algorithm. It is clear that there exists an extension of u satisfying C if and only if C' is satisfiable. Furthermore, the set of variables forced true by C with respect to u coincides with those forced true in C', less the variables in $dom_{true}(u)$. A similar claim holds for the variables forced to false.

We first introduce the notion of *pebblings* (adapted from Dowling and Gallier [1984]) and state several straight-forward propositions, from which the theorem will follow. Let x be a variable in X'. A *C'-pebbling* of (x, m) from *True* is a sequence of markings $(x_0, m_0), \ldots, (x_k, m_k)$, such that $x_k = x$ and $m_k = m$, and each x_i and m_i satisfy:

- $x_i \in X'$ and $m_i \in \{*\} \cup \{*x \mid x \in X\}$, and
- one of the following:
  - $x_i = True$ and $m_i =$ " $*$ ", or
  - $m_i =$ " $* x_i$", or
  - $\exists i_1, \ldots, i_l < i$ such that each $m_{i_k} =$ " $*$ ", and $m_i =$ " $*$ ", and $x_{i_1} \wedge \cdots \wedge x_{i_l} \rightarrow x_i \in C'$, or
  - $\exists z \in X$ and $\exists i_1, \ldots, i_l < i$ such that each $m_{i_k} =$ " $*$ " or "$*z$", and $m_i =$ " $*z$", and $x_{i_1} \wedge \cdots \wedge x_{i_l} \rightarrow x_i \in C'$.

A C'-pebbling is *complete* if the sequence cannot be extended to add a new mark.

It is easy to see that each time the procedure Horn-Forced marks a variable x with a mark m, the sequence of markings till this point forms a valid C'-pebbling of (x, m) from *True*, and the final sequence of markings produced does indeed constitute a complete pebbling.

Proposition 4.2. *Consider a C'-pebbling of $(x, \text{“} * \text{”})$ from True. Let v be a valuation such that $v \models C'$. Then $v(x) = true$.*

Proposition 4.3. *Consider a C'-pebbling of $(y, \text{“} * x\text{”})$ from True. Let v be a valuation such that $v(x) = true$ and $v \models C'$. Then $v(y) = true$.*

Proposition 4.4. *Consider a complete C'-pebbling from True, in which False is not marked “∗”. Let v be the valuation given by*

$$v(x) = \begin{cases} true & \text{if } x \text{ is marked “}*\text{”; and} \\ false & \text{otherwise.} \end{cases}$$

Then $v \models C'$.

Proposition 4.5. *Let $y \in X$. Consider a complete C'-pebbling from True, in which False is not marked “$*y$”. Let v be the valuation given by*

$$v(x) = \begin{cases} true & \text{if } x \text{ is marked “}*\text{” or “}*y\text{”; and} \\ false & \text{otherwise.} \end{cases}$$

Then $v \models C'$.

Given Propositions 4.2 to 4.5, the proof of Theorem 4.1 follows immediately. □

4.2 An Efficient Version of Algorithm Horn-Forced

Algorithm 2 can be made to run in time $O(h \cdot n)$ where h and n are the sizes of C and X respectively, by using an efficient data structure for the propagation of marks. Algorithm 3 shows this efficient version Horn-Forced-2 of Algorithm 2. We essentially use similar data structures (like *marked*, *clauses*, and *count*) as the algorithm of Dowling and Gallier [1984], except that we need one such set for *each* mark (“$*$” or “$*x$” where $x \in X$). The propagation of marks proceeds independently, except for the propagation of the $*$-mark which also impacts the other marks (Line 25).

The algorithm runs in time $O(h \cdot n)$. To see this, observe that each variable x in X is put in the workset for each mark m, at most once. This is because we first check whether $marked[m][x]$ is true before adding x to $WorkSet[m]$, and once added it is marked. While processing each x in $WorkSet[m]$ (at Line 17), we do at most $O(h)$ work overall, corresponding to distinct occurrences of literals in C. While processing an x in $WorkSet[*]$, we additionally (in Line 25) do $O(n)$ work to scan the marked entries for $*y$ for each $y \in X$. Thus for processing entries in the $*$ workset, we do $O(h) + O(n^2)$ work totally. Thus the total time taken by Alg. 3 is bounded by $O(h \cdot n)$.

A crucial advantage of our algorithm over naive invocations of the Horn-Sat algorithm is that if we make a series of calls to Horn-Forced in which the set of Horn constraints C is fixed and the partial valuations u are successive extensions, then the *total* time taken across this series of calls is bounded by $O(h \cdot n)$. To see this, observe that if we have two successive calls to Horn-Forced-2 with arguments X, C, u and X, C, u' respectively, with u' an extension of u (i.e., $u' = u_F^T$ for some $T, F \subseteq X$), then for the second call to the algorithm we can continue from where the first call finished. That is, in the second call to the procedure, there is no need to initialize the data structures in Line 9, and instead in Line 11 we use T instead of $dom_{true}(u')$ and F instead of $dom_{false}(u')$. Thus the total time taken by the algorithm *across* the two calls is still bounded by $O(h \cdot n)$. It follows that

Algorithm 3 An efficient version of Algorithm Horn-Forced

```
procedure Horn-Forced-2
Input: Set $C$ of Horn constraints over $X$, partial valuation $u$ over $X$ consistent with $C$. Let $|X| = n$
    and $|C| = m$.
Output: $forced\text{-}true(C, u)$, $forced\text{-}false(C, u)$.
    Add two new variables $True$ and $False$ to $X$. Let $X' = X \cup \{True, False\}$.
    $C' \leftarrow C$ + clauses $true \rightarrow x$ for each $x$ such that $u(x) = true$, and $x \rightarrow false$ for each $x$ such
    that $u(x) = false$.
    Bool $marked[n+1][n]$ // marked arrays
    Integer $count[n+1][m]$ // LHS count for each mark for each clause
    List of Int $clauses[n]$ // $clauses[i]$ is the list of clauses with $x_i$ in LHS
    Int $head[m]$ // $head[c]$ is variable at the head of clause $c$.
    Set of Var $WorkSet[n+1]$ // Workset of variables to process for each mark
    Initialize data structures;
    $P \leftarrow \emptyset, N \leftarrow \emptyset$ // Forced sets
    for all $x \in dom_{true}(u)$ do
        if $(\neg marked[*][x])$ then
            $marked[*][x] \leftarrow true$
            $WorkSet[*] \leftarrow WorkSet[*] \cup \{x\}$.
    while ($\exists$ some $x$ in $WorkSet[mk]$ for some mark $mk$) do
        $WorkSet[mk] \leftarrow WorkSet[mk] \setminus \{x\}$
        for all $(c \in clauses[x])$ do
            $z \leftarrow head[c]$
            $count[mk][c] \leftarrow count[mk][c] - 1$
            if $(count[mk][c] = 0)$ then
                if $(z = False)$ and ($mk$ is of the form $*y$) then
                    $N \leftarrow N \cup \{y\}$
                else if $(\neg marked[mk][z])$ then
                    $WorkSet[mk] \leftarrow WorkSet[mk] \cup \{z\}$
                if $(m = *)$ then
                    $P \leftarrow P \cup \{z\}$
                    for all $(y \in X)$ do
                        if $(\neg marked[*y][x])$ then
                            $marked[*y][x] \leftarrow true$
                            $WorkSet[*y] \leftarrow WorkSet[*y] \cup \{x\}$
    $P' \leftarrow P - dom_{true}(u)$, $N' \leftarrow N - dom_{false}(u)$
    return $P', N'$
```

for a *series* of such calls to procedure Horn-Forced-2, in which the constraints are fixed and the partial valuations are successive extensions, the total time taken will be bounded by $O(h \cdot n)$.

In particular the calls made by the decision tree algorithm (Algorithm 1) of Sec. 3 are of this type, and hence the total time across those calls is bounded by $O(h \cdot n)$.

5 NODE AND ATTRIBUTE SELECTION

The decision tree algorithm in Section 3 returns a consistent tree irrespective of the order in which nodes of the tree are processed or the heuristic used to choose the best attribute to split nodes in the tree. If one is not careful while selecting the next node to process or one ignores the Horn constraints while choosing the attribute to split the node, seemingly good splits can turn into bad ones as data points involved in the Horn constraints get classified during the construction of the tree. We experimented with the following strategies for node and attribute selection:

Node selection: breadth-first-search, depth-first-search, random selection, and selecting nodes with the maximum/minimum entropy.

Attribute selection: based on a new information gain metric that penalizes node splits that cut Horn constraints; based on entropy for Horn samples obtained by assigning probabilistic likelihood values to unlabeled datapoints using model counting.

So as to clutter the paper not too much, we here only describe the best performing combination of strategies in detail. The experiments reported in Section 6 have been conducted with this combination.

5.1 Choosing the Next Node to Expand the Decision Tree

We select nodes in a breadth-first search (BFS) order for building the decision tree. BFS ordering ensures that while learning multiple invariant annotations, the subtree for all invariants gets constructed simultaneously. In comparison, in depth-first ordering of the nodes, subtrees for the multiple invariants are constructed one after the other. In this case, learning a simple invariant for an annotation (e.g., *true*) usually forces the invariant for a different annotation to become very complex.

5.2 Choosing Attributes for Splitting Nodes

Similar to Garg et al. [2016], we observed that if one chooses attribute splits based on the entropy of the node that ignores Horn constraints, the invariant learning algorithm tends to produce large trees. In the same spirit as Garg et al. [2016], we penalize the information gain for attribute splits that cut Horn constraints, and choose the attribute with the highest corresponding information gain. For a sample $S = (X, C)$ that is split with respect to attribute a into subsamples S_a and $S_{\neg a}$, we say that the corresponding attribute split cuts a Horn constraint $\psi \in C$ if and only if (a) $x \in \mathit{premise}(\psi)$, $x \in S_a$, and $\mathit{conclusion}(\psi) \in S_{\neg a}$, or (b) $x \in \mathit{premise}(\psi)$, $x \in S_{\neg a}$, and $\mathit{conclusion}(\psi) \in S_a$. The *penalized* information gain is defined as

$$\mathit{Gain}_{\mathit{pen}}(S, S_a, S_{\neg a}) = \mathit{Gain}(S, S_a, S_{\neg a}) - \mathit{Penalty}(S, S_a, S_{\neg a}, C),$$

where the penalty is proportional to the number of Horn constraints cut by the attribute split. However, we do not penalize a split when it cuts a Horn constraint such that the premise of the constraint is labeled negative and the conclusion is labeled positive. We incorporate this in the penalty function by formally defining $\mathit{Penalty}(S, S_a, S_{\neg a}, H)$ as

$$\sum_{\substack{\psi \in H, x \in S_a \\ x \in \mathit{premise}(\psi) \\ \mathit{conclusion}(\psi) \in S_{\neg a}}} \big(1 - f(S_a, S_{\neg a})\big) + \sum_{\substack{\psi \in H, x \in S_{\neg a} \\ x \in \mathit{premise}(\psi) \\ \mathit{conclusion}(\psi) \in S_a}} \big(1 - f(S_{\neg a}, S_a)\big),$$

where, for subsamples S_1 and S_2, $f(S_1, S_2)$ is the likelihood of S_1 being labeled negative and S_2 being labeled positive (i.e., $f(S_1, S_2) = \frac{N_1}{P_1+N_1} \cdot \frac{P_2}{P_2+N_2}$). Here, P_i and N_i is the number of positive and negative datapoints respectively in the sample S_i.

6 IMPLEMENTATION AND EXPERIMENTAL EVALUATION

We have implemented two different prototypes to demonstrate the benefits of the proposed learning framework.[1] The first prototype, named *Horn-DT-Boogie*, builds on top of Microsoft's program verifier Boogie [Barnett et al. 2005] and reuses much of the code originally developed by Garg et al. [2016] for their ICE learning tool. The decision tree learning algorithm and the Horn solver are our own implementations, consisting of roughly 6000 lines of C++ code. We use *Horn-DT-Boogie* to demonstrate the effectiveness of learning correlated invariants that are required for the verification of recursive and concurrent programs.

The second prototype is named *Horn-DT-CHC*. Its teacher component is a fresh implementation consisting of roughly 2500 lines of C++ code, while the decision tree learning algorithm and the Horn solver are taken from *Horn-DT-Boogie*. *Horn-DT-CHC* takes CHCs in the *SMTLib2* format as input and learns a predicate for each uninterpreted function declared. We use *Horn-DT-CHC* to evaluate the performance of our learning algorithm as a CHC solver.

Both these prototypes use a predicate template of the form $x \pm y \leq c$, called *octagonal constraints*, where x, y are numeric program variables or non-linear expressions over numeric program variables and c is a constant determined by the decision tree learner. Moreover, our prototypes use the split and node selection strategies described in Sec. 5. Finally, both *Horn-DT-Boogie* and *Horn-DT-CHC* employ two additional heuristics: first, both tools initially search for conjunctive invariants (using Houdini [Flanagan and Leino 2001]) and if this fails, proceed with the decision tree learning algorithm; second, since we are working over an infinite set of potential predicates (i.e., all predicates of the form $x \pm y \leq c$), we slowly increase the number of predicates as sketched in Section 3 to guarantee convergence to an invariant (if one exists).

We evaluated our prototypes on three benchmark suites:

(1) The first suite consists of 52 recursive programs from the Software Verification Competition (SV-COMP 2018) [Beyer 2017]. We compared *Horn-DT-Boogie* on this benchmark suite with *Ultimate Automizer* [Heizmann et al. 2013], the winner of the SV-COMP 2018 recursive programs track.

(2) The second benchmark suite consists of 12 concurrent programs, which includes popular concurrent protocols such as Peterson's algorithm and producer-consumer problems. However, we are not aware of any automated tool for generating Rely-Guarantee proofs [Xu et al. 1997] or Owicki-Gries proofs [Owicki and Gries 1976] with which we could compare with on these programs.

(3) The third benchmark suite consists of 45 sequential programs without recursion taken from Dillig et al. [2013]. Our aim here is to evaluate the performance of our technique as a solver for constraint Horn clauses (CHCs). To this end, we first generated CHCs of the programs in Dillig et al.'s benchmarks suite using SeaHorn [Gurfinkel et al. 2015]. Then, we compared *Horn-DT-CHC* with *Z3/PDR* [Hoder and Bjørner 2012], a state-of-the-art CHC solver. As SeaHorn does currently not handle recursive or concurrent programs, we limited our comparison to Dillig et al.'s suite of non-recursive sequential programs.

Note that we did not compare our tool with Garg et al. [2016]'s original ICE learning algorithm as both algorithms can be integrated seemlessly: as long as the Teacher generates linear Horn clauses, one would use Garg et al.'s simpler algorithm and only switch to our Learner once the first non-linear Horn clauses is returned by the Teacher. Moreover, note that our tool learns invariants in a specific template class consisting of arbitrary Boolean combinations of user-specified atomic inequalities. The other tools do not have such a strict template, though they do of course search for

[1] The sources are publicly available at https://github.com/horn-ice as well as in the ACM digital library.

invariants in a specific logic. The experimental comparisons, especially when we outperform other tools, should be considered with this in mind.

The rest of this section presents the three benchmarks suits in detail and discusses our empirical evaluation. All experiments were conducted on a Intel Core i3-4005U 4x1.70GHz CPU with 4 GB of RAM running Ubuntu 16.04 LTS 64 bit. We used a timeout of 600 s for each benchmark.

6.1 First Benchmark Suite (Recursive Programs)

The first benchmark suite consists of the entire set of "`true-unreach`" (error is unreachable) recursive programs of the Software Verification Competition (SV-COMP 2018) [Beyer 2017]. This benchmark suite contains 52 programs, including both terminating and non-terminating programs. For recursive programs we used a modular verification technique, in the form of function contracts for each procedure. We run *Horn-DT-Boogie* on these programs by manually converting them into Boogie programs. For three of the 52 programs we used non-linear expressions over numerical variables as ground terms, for rest of the programs we used numerical variables as ground terms.

To assess the performance of *Horn-DT-Boogie*, we compared it to *Ultimate Automizer* [Heizmann et al. 2013], the winner of the SV-COMP 2018 verification competition in the "ReachSafety-Recursive" track (recursive programs with satisfy assertions).

Figure 6 summarizes the results of our experimental evaluation on the recursive benchmark suite, while Table 1 gives more detailed results. As the left-hand-side of Figure 6 shows, *Horn-DT-Boogie* was able to synthesize function contracts and verified 39 programs in the benchmark suite while it timed out on 13 remaining programs. Out of the 39 programs verified, 16 (41 %) required disjunctive invariants. *Ultimate Automizer*, on the other hand, was able to verify 38 programs and timed out on 14 programs. On eleven programs, both tools timed out.

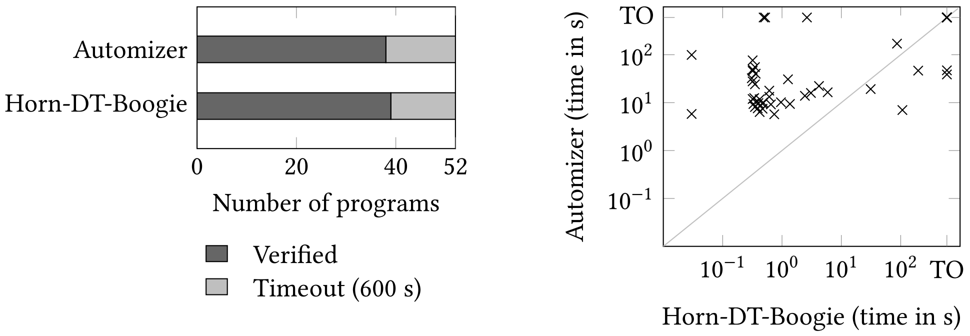


Fig. 6. Experimental comparison of *Horn-DT-Boogie* with *Ultimate Automizer* on the recursive programs benchmark suite. TO indicates a time out after 600 s.

Note that larger timeouts (i.e., 1200 s) did not lead to additional programs being verified. On at least ten programs, the reason for this is that the invariants to synthesize require large constants; in fact, SV-COMP has several such examples, including proving $\mathit{fib}(15) = 610$ for a recursive implementation of *fib*. However, black-box techniques in general—and our learning-based techniques in particular—are less effective in synthesizing formulas involving large constants. A lack of expressiveness (in terms of richer set of ground terms) does not seem to be the reason for timeouts on these benchmarks.

The right-hand-side of Figure 6 compares the runtimes of *Horn-DT-Boogie* and *Ultimate Automizer*. *Horn-DT-Boogie* was able to verify 28 programs very quickly, requiring less than one second each.

Table 1. Experimental results of *Horn-DT-Boogie* and *Ultimate Automizer* on the recursive programs benchmark suite. "Rounds" corresponds to the number of rounds of the learning process. "Pos", "Neg", and "Horn" refers to the number of positive, negative, and Horn examples produced during the learning, respectively. "TO" indicates a timeout after 600 s. All times are given in seconds.

Benchmark	Horn-DT-Boogie						Automizer
	Rounds	Pos	Neg	Horn	Learner time	Total time	Time
Ackermann01.c	13	2	1	11	0	0.61	17.8
Ackermann03.c						TO	38.75
Ackermann04.c	262	2	5	275	24	31.18	19.21
Addition01.c	7	3	1	3	0	0.47	7.47
Addition03.c	7	4	1	2	0	0.49	TO
afterrec_2calls.c	0	0	0	0	0	0.03	99.24
afterrec.c	0	0	0	0	0	0.03	5.81
EvenOdd01.c						TO	TO
fibo_10.c						TO	TO
fibo_15.c						TO	TO
fibo_20.c						TO	TO
fibo_25.c						TO	TO
fibo_2calls_10.c						TO	TO
fibo_2calls_15.c						TO	TO
fibo_2calls_20.c						TO	TO
fibo_2calls_25.c						TO	TO
fibo_2calls_2.c	53	1	4	51	0	1.26	30.7
fibo_2calls_4.c	92	1	3	93	0	2.43	13.88
fibo_2calls_5.c	118	9	14	97	3	3.07	15.93
fibo_2calls_6.c	144	8	17	121	1	4.2	22.27
fibo_2calls_8.c	569	9	22	540	177	196.97	46.61
fibo_5.c	13	1	2	10	0	0.42	11.77
fibo_7.c	147	1	3	143	4	5.88	16.45
Fibonacci01.c	11	2	2	9	0	0.4	9.79
Fibonacci02.c						TO	47.26
Fibonacci03.c	372	7	3	363	70	86.63	170.62
gcd01.c	11	3	1	8	0	0.61	13.24
gcd02.c						TO	TO
id2_b2_o3.c	20	4	2	14	1	0.65	9.19
id2_b3_o5.c	44	5	2	38	0	0.96	10.18
id2_b5_o10.c	71	4	2	66	0	1.36	9.39
id2_i5_o5.c	10	1	1	10	0	0.48	10.09
id_b2_o3.c	11	4	2	5	0	0.39	7.61
id_b3_o5.c	14	4	2	8	0	0.42	6.36
id_b5_o10.c	42	7	2	33	0	0.74	5.72
id_i10_o10.c	6	1	1	4	0	0.32	11.93
id_i15_o15.c	6	1	1	4	0	0.32	27.83
id_i20_o20.c	6	1	1	4	0	0.32	45.92
id_i25_o25.c	6	1	1	4	0	0.32	49.22
id_i5_o5.c	6	1	1	4	0	0.33	9.36
McCarthy91.c	877	14	840	23	79	106.77	7.01
MultCommutative.c	7	4	0	3	0	0.52	TO
Primes.c	74	2	0	83	1	2.63	TO
recHanoi01.c						TO	TO
recHanoi02.c	5	1	1	3	1	0.32	76.49
recHanoi03.c	5	1	1	3	0	0.31	32.32
sum_10x0.c	4	1	1	3	0	0.34	12.45
sum_15x0.c	4	1	1	3	0	0.35	24.12
sum_20x0.c	4	1	1	3	0	0.36	40.21
sum_25x0.c	4	1	1	3	0	0.35	55.25
sum_2x3.c	4	1	1	2	0	0.35	8.11
sum_non_eq.c	9	4	1	4	0	0.47	9.47

By contrast, on half of the programs (i.e., 26), *Ultimate Automizer* required more than 10 s to finish. On benchmarks that both tools successfully verified, *Horn-DT-Boogie* is about two times faster than *Ultimate Automizer* in terms of the total time taken.

Table 2. Results of *Horn-DT-Boogie* on concurrent programs. The columns show the number of invariants to be synthesized ("Inv"), the total number of terms used ("Dim"), the number of iterations ("Rounds"), the number of counterexamples generated ("Pos", "Neg", and "Horn"), and the time taken ("Time"). Benchmarks with the suffix *-RG* and *-OG* indicate Rely-Guarantee-style proofs and Owicki-Gries-style proofs, respectively.

Benchmark	Inv	Dim	Rounds	Pos	Neg	Horn	Time (s)
12_hour_clock_serialization	1	3	6	1	1	4	1.05
18_read_write_lock-OG	8	16	18	2	1	15	2.21
fib_bench-OG	6	18	33	2	0	37	1.61
Mine_Fig_1-OG [Miné 2014]	10	20	48	2	0	53	2.85
Mine_Fig_1-RG [Miné 2014]	12	28	49	3	0	62	5.16
Mine_Fig_4-OG [Miné 2014]	13	28	116	3	0	146	28.26
Mine_Fig_4-RG [Miné 2014]	15	44	7	2	0	5	1.11
peterson-OG	8	32	297	3	74	221	23.81
pro_cons_atomic-OG	8	16	32	2	0	30	1.75
pro_cons_queue-OG	8	16	32	2	0	30	1.77
qw2004-OG	13	20	24	2	1	23	2.84
stateful01-OG	6	12	287	2	0	285	13.03

In fact, we find it surprising that our prototype without optimizations is competitive to the best tool for this track of SV-COMP. We believe that this demonstrates lucidly that template-based black-box invariant synthesis is a promising and competitive technique for program verification.

6.2 Second Benchmark Suite (Concurrent Programs)

The second benchmark suite consists of 12 concurrent programs obtained from the literature on concurrent verification, including the work of Miné [2014]. Note that some of these programs use non-linear expressions over numerical variables as ground terms. Consequently, the annotations our tool generated were also non-linear.

For concurrent programs, we have used both Rely-Guarantee proof techniques [Xu et al. 1997] and Owicki-Gries proof techniques [Owicki and Gries 1976] to verify the assertions. All benchmarks were manually converted into Boogie programs, essentially by encoding the verification conditions for Rely-Guarantee- and Owicki-Gries-style proof requirements, respectively.

Table 2 shows the results of running *Horn-DT-Boogie* on these programs. The column "Inv" reports the number of invariants that need to be synthesized in parallel for a particular benchmark. The column "Dim" refers to the learning dimension (i.e., total number of predicates over which invariants are synthesized).

Verification using Owicki-Gries proof rules requires adequate invariants at each program point in each thread. In comparison, Rely-Guarantee additionally requires two-state invariants for each thread for the Rely/Guarantee conditions. These additional invariants make learning for Rely-Guarantee proofs more difficult. Nonetheless, our tool successfully learned invariants for all of these programs in reasonable time, with most verification tasks finishing in less than 10 s. Two of the 12 programs (16.6 %) required disjunctive invariants.

6.3 Third Benchmark Suite (Sequential Programs)

The third benchmark suite consists of 45 sequential programs from Dillig et al. [2013] (we omitted one program, named 39.c, as the translation to CHCs trivially solves it). These programs vary in complexity and range from simple integer manipulating programs to programs involving non-linear computations. Each program contains at least one loop and at least one assertion. Moreover, some benchmarks contain nested loops or multiple sequential loops. All of these programs can be proven correct using invariants that fall into our class of templates.

To evaluate our CHC-based verifier *Horn-DT-CHC*, we used *SeaHorn* [Gurfinkel et al. 2015] to convert the programs of Dillig et al. [2013] into verification conditions in the form of CHCs (which are output in the *SMTLib2* format). This allowed us to compare our technique to the popular *Z3/PDR* engine [Hoder and Bjørner 2012], which is implemented in the Z3 SMT solver [de Moura and Bjørner 2008]. However, as *SeaHorn* is currently unable process recursive and concurrent programs, our comparison is necessarily limited to sequential programs.

Figure 7 summarizes the results of our experimental evaluation on the sequential programs benchmark suite, while Table 3 lists the results in more detail. As shown on the left of Figure 7, *Horn-DT-CHC* was able to verify 29 programs of the benchmark suite, while it timed out on 16 programs. Out of the 29 programs verified, five (17.24 %) required disjunctive invariants. *Z3/PDR*, on the other hand, was able to verify 22 programs and timed out on 23 programs. There were 11 programs on which both tools timed out.

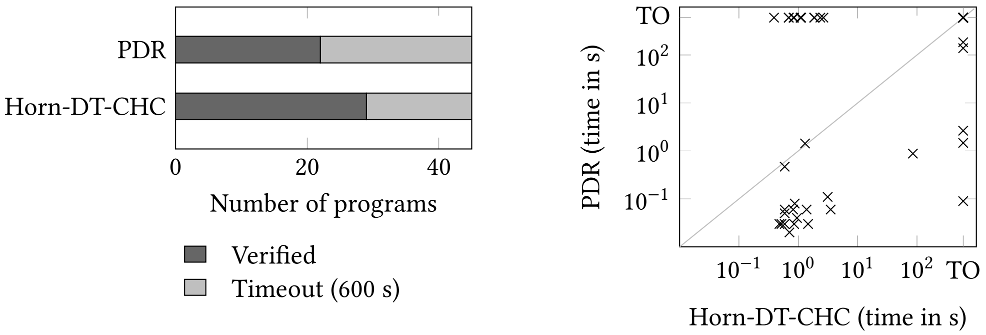


Fig. 7. Experimental comparison of *Horn-DT-CHC* with *PDR* on the sequential programs benchmark suite. TO indicates a time out after 600 s.

The right-hand-side of Figure 7 compares the runtimes of *Horn-DT-CHC* with those of *Z3/PDR*. *Z3/PDR* requires less time overall to verify programs. *Horn-DT-CHC*, on the other hand verifies more programs.

6.4 Summary of the Experimental Evaluation

We believe our results show that our extension of decision tree-based ICE learners to Horn-ICE learners is quite efficient, and favorably compares with state-of-the-art tools for solving non-linear Horn constraints. Our technique is able to prove a large class of programs correct drawn from a variety of styles (sequential programs with and without recursion, concurrent programs) that result in non-linear Horn clauses.

7 RELATED WORK

Invariant synthesis is the central problem in automated program verification and, over the years, several techniques have been proposed for synthesizing invariants, including abstract interpretation [Cousot and Cousot 1977], interpolation [Jhala and McMillan 2006; McMillan 2003], IC3 and PDR [Bradley 2011; Karbyshev et al. 2015], predicate abstraction [Ball et al. 2001], abductive inference [Dillig et al. 2013], as well as synthesis algorithms that rely on constraint solving [Colón et al.

Table 3. Experimental results of *Z3/PDR* and *Horn-DT-CHC* on the sequential programs benchmark suite. “TO” indicates a timeout after 600 s. All times are given in seconds.

Benchmark	Horn-DT-CHC	Z3/PDR	Benchmark	Horn-DT-CHC	Z3/PDR
01.c	0.59	0.47	24.c	3.51	0.06
02.c	TO	2.64	25.c	1.85	TO
03.c	1.46	0.03	26.c	2.35	TO
04.c	0.48	0.03	27.c	0.71	0.02
05.c	0.69	TO	28.c	0.69	TO
06.c	1.09	TO	29.c	2.65	TO
07.c	TO	TO	30.c	0.39	TO
08.c	1.12	TO	31.c	3.13	0.11
09.c	TO	TO	32.c	TO	TO
10.c	0.59	0.05	33.c	TO	TO
11.c	TO	138.65	34.c	85.21	0.88
12.c	TO	TO	35.c	0.52	0.03
13.c	TO	TO	36.c	TO	TO
14.c	0.85	TO	37.c	0.82	0.06
15.c	TO	TO	38.c	TO	TO
16.c	0.83	TO	40.c	TO	TO
17.c	1.38	0.06	41.c	0.87	0.08
18.c	TO	184.39	42.c	TO	1.47
19.c	1.3	1.43	43.c	0.95	0.04
20.c	TO	0.09	44.c	1.1	TO
21.c	0.85	0.03	45.c	1.87	TO
22.c	TO	TO	46.c	0.59	0.06
23.c	0.59	0.03			

2003; Gulwani et al. 2008; Gupta and Rybalchenko 2009]. Subsequent to Grebenshchikov et al. [2012], there has been a lot of work towards Horn-clause solving [Beyene et al. 2013; Bjørner et al. 2013], using a combination of these techniques. For instance, SeaHorn [Gurfinkel et al. 2015] is a verification framework that translates verification conditions of a program to constraint Horn clauses that can be solved using several backend solvers.

Complementing these techniques are data-driven invariant synthesis techniques, the first ones to be proposed being Daikon [Ernst et al. 2000], which learns likely program invariants, and the popular Houdini algorithm [Flanagan and Leino 2001], which learns conjunctive inductive invariants. Although Houdini can be seen as an algorithm that learns from Horn samples, it is fundamentally different from our approach in that it requires a finite set of predicates to be given by the user. Thus, as all of our benchmarks programs (and their invariants and contracts) involve numeric variables, we cannot directly apply Houdini, even if an conjunctive invariant exists, without manually providing a sufficient set of predicates. By contrast, our decision tree learning algorithm learns Boolean combinations over an infinite set of predicates. These predicates of the form $e \leq c$, where e is an arithmetic expression over the program variables provided by the user (or an octagoal constraint of two such expressions) and the value c is automatically inferred by our learning algorithm.

In fact, data-driven invariant synthesis has seen renewed interest lately [Brockschmidt et al. 2017; Fedyukovich et al. 2017; Garg et al. 2013, 2014, 2016; Nguyen et al. 2012; Padhi et al. 2016; Pavlinovic et al. 2016; Sharma and Aiken 2014; Sharma et al. 2013b,a, 2012; Zhu et al. 2015, 2016].

When the program manipulates complex data-structures, arrays, pointers, etc., or when one needs to reason over a complicated memory model and its semantics, the invariant for the correctness of the program might still be simple. In such a scenario, a black-box, data-driven *guess-and-check* approach, guided by a finite set of program configurations, has been shown to be advantageous. Recent work by Vizel et al. [2017] shows how IC3 can be viewed as a form of ICE learning.

However, implication counterexamples proposed by Garg et al. [2014] are not sufficient for learning invariants in general program verification settings. Grebenshchikov et al. [2012] have shown that most reasoning in program verification is expressed in terms of Horn clauses which are in general non-linear. In the context of data-driven invariant synthesis, our work generalizes the ICE learning model by Garg et al. [2014] to non-linear Horn counterexamples, expanding the class of programs that can be handled by learning from Horn-ICE counterexamples.

We are aware of a concurrently developed work by Champion et al. [2018] that builds a decision tree learner for Horn samples in the context of finding refinement types for higher-order functional programs. However, this algorithm is quite different from ours in that it builds different trees for different annotations, processing one annotation at a time, while our approach builds all the annotations simultaneously. Moreover, Champion et al.'s algorithm does neither guarantee that a decision tree will always be constructed if one exists, nor that the learner will eventually converge to a valid solution when the hypothesis class is infinite, as in our case.

Recent work by Zhu et al. [2018] presents another data-driven solver for constrained Horn clauses that is also based on learning decision trees. This algorithm has two main differences to our approach. First, Zhu et al.'s algorithm generates decision predicates automatically, whereas we operate within a fixed template class (octagonal constraints over the program variables). Note, however, that this is not a restriction but a design choice that balances effectiveness with performance: in fact, our algorithm can easily handle any other user-specified set of decidable predicate. Second, Zhu et al.'s algorithm operates in the classical machine learning setup with only positively and negatively labeled data. If a conjucture is found to be non-inductive, additional positive and negative examples are generated by unwinding the constrained Horn clauses a finite number of times (which increases during the learning process). However, this approach of generating counterexamples cannot guarantee to converge to a valid solution (if one exists). This is in stark contrast to our algorithm, which provides this guarantee.

An experimental comparison of our algorithm with those of Champion et al. [2018] as well as Zhu et al. [2018] is part of future work.

8 CONCLUSIONS AND FUTURE WORK

We have developed learning-based black-box algorithms for synthesizing invariants for programs that generate Horn-style proof constraints. We have overcome several challenges in this process that non-linear Horn constraints bring, giving new and efficient decision tree algorithms that build small decision trees consistent with samples. Our algorithms come with robustness guarantees that they will always succeed in building a tree when one exists while working in polynomial time, and with convergence guarantees that they will find an inductive invariant if one expressible in the logic exists. We have implemented and evaluated our technique, and shown that our tool favorably compares with state-of-the-art tools on a large class of benchmarks for several styles of programs that compile to non-linear Horn constraints.

One avenue where black-box learning engines for invariants have been particularly useful is in the context of programs where checking validity of verification conditions is itself undecidable [Neider et al. 2018]. It would be interesting to extend our technique to such domains.

Second, we believe that our experimental results suggest that a template-based learning engine is quite competitive. Building a more competitive tool based on our techniques for the SVComp competition is an interesting future direction. We believe choosing a template from a class of templates based on extracting feaures from a simple static analysis of the program and using priors gained from the experience of verifying similar programs in the past would make our approach efficient.

Finally, we would like to apply our invariant synthesis technique for particular practical domains, where domain-specific templates involving complex atomic predicates can be used to synthesize invariants (much like GPUVerify [Betts et al. 2012; Chong et al. 2013] does for learning conjunctive invariants to prove GPU programs race-free).

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Nos. 1527395 and 1138994.

REFERENCES

Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. 2001. Automatic Predicate Abstraction of C Programs. In *PLDI, 2001*. 203–213.

Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO 2005 (LNCS)*, Vol. 4111. Springer, 364–387.

Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *OOPSLA 2012*. ACM, 113–132.

Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. 2013. Solving Existentially Quantified Horn Clauses. In *CAV 2013 (LNCS)*. Springer, 869–882.

Dirk Beyer. 2017. Software Verification with Validation of Results - (Report on SV-COMP 2017). In *TACAS 2017 (LNCS)*, Vol. 10206. 331–349.

Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. 2013. On Solving Universally Quantified Horn Clauses. In *SAS 2013*. 105–125.

Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *VMCAI 2011 (LNCS)*, Vol. 6538. Springer, 70–87.

Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. 2017. Learning Shape Analysis. In *SAS 2017*. 66–87.

Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. 2018. ICE-Based Refinement Type Discovery for Higher-Order Functional Programs. In *Proc. TACAS 2018, Thessaloniki, Greece, April 14-20, 2018 (LNCS)*, Vol. 10805. Springer, 365–384. https://doi.org/10.1007/978-3-319-89960-2_20

Nathan Chong, Alastair F. Donaldson, Paul H. J. Kelly, Jeroen Ketema, and Shaz Qadeer. 2013. Barrier invariants: a shared state abstraction for the analysis of data-dependent GPU kernels. In *Proc. OOPSLA 2013, Indianapolis, USA, October 26-31, 2013*. ACM, 605–622. https://doi.org/10.1145/2509136.2509517

Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *CAV 2003 (LNCS)*, Vol. 2725. Springer, 420–432.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL 1977*. ACM Press, 238–252.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS 2008 (LNCS)*, Vol. 4963. Springer, 337–340.

Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *Object Oriented Programming Systems Languages & Applications, OOPSLA 2013*. 443–456.

William F. Dowling and Jean H. Gallier. 1984. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *J. Log. Program.* 1, 3 (1984), 267–284.

Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. 2000. Quickly detecting relevant program invariants. In *ICSE 2000*. ACM Press, 449–458.

Grigory Fedyukovich, Samuel J Kaufman, and Rastislav Bodik. 2017. Sampling Invariants from Frequency Distributions. (2017).

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001 (LNCS)*, Vol. 2021. Springer, 500–517.

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2013. Learning Universally Quantified Invariants of Linear Data Structures. In *CAV 2013 (LNCS)*, Vol. 8044. Springer, 813–829.

Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *CAV 2014 (LNCS)*, Vol. 8559. Springer, 69–87.

Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *POPL 2016*. ACM, 499–512.

Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing software verifiers from proof rules. In *PLDI 2012*. ACM, 405–416.

Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program analysis as constraint solving. In *PLDI 2008*. ACM, 281–292.

Ashutosh Gupta and Andrey Rybalchenko. 2009. InvGen: An Efficient Invariant Generator. In *CAV 2009 (LNCS)*, Vol. 5643. Springer, 634–640.

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *CAV 2015 (LNCS)*, Vol. 9206. Springer, 343–361.

Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. 2013. Ultimate Automizer with SMTInterpol. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*. Springer-Verlag, Berlin, Heidelberg, 641–643. https://doi.org/10.1007/978-3-642-36742-7_53

Krystof Hoder and Nikolaj Bjørner. 2012. Generalized Property Directed Reachability. In *SAT 2012 (LNCS)*, Vol. 7317. Springer, 157–171.

Ranjit Jhala and Kenneth L. McMillan. 2006. A Practical and Complete Approach to Predicate Refinement. In *TACAS 2006 (LNCS)*, Vol. 3920. Springer, 459–473.

C. B. Jones. 1983. Tentative Steps Toward a Development Method for Interfering Programs. *Transactions on Programming Languages and System* 5, 4 (1983), 596–619.

Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. 2015. Property-Directed Inference of Universal Invariants or Proving Their Absence. In *CAV 2015 (LNCS)*, Vol. 9206. Springer, 583–602.

Kenneth L. McMillan. 2003. Interpolation and SAT-Based Model Checking. In *CAV 2003 (LNCS)*, Vol. 2725. Springer, 1–13.

Antoine Miné. 2014. Relational Thread-Modular Static Value Analysis by Abstract Interpretation. In *VMCAI 2014 (LNCS)*, Vol. 8318. Springer, 39–58.

Antoine Miné. 2014. Relational thread-modular static value analysis by abstract interpretation. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 39–58.

Daniel Neider, Pranav Garg, P Madhusudan, Shambwaditya Saha, and Daejun Park. 2018. Invariant Synthesis for Incomplete Verification Engines. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 232–250.

ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *ICSE 2012*. IEEE, 683–693.

Susan Owicki and David Gries. 1976. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM* 19, 5 (1976), 279–285.

Saswat Padhi, Rahul Sharma, and Todd D. Millstein. 2016. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*. 42–56.

Zvonimir Pavlinovic, Akash Lal, and Rahul Sharma. 2016. Inferring annotations for device drivers from verification histories. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*. 450–460.

J. Ross Quinlan. 1986. Induction of Decision Trees. *Machine Learning* 1, 1 (1986), 81–106.

Rahul Sharma and Alex Aiken. 2014. From Invariant Checking to Invariant Inference Using Randomized Search. In *CAV 2014 (LNCS)*, Vol. 8559. Springer, 88–105.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. 2013b. A Data Driven Approach for Algebraic Loop Invariants. In *ESOP 2013 (LNCS)*, Vol. 7792. Springer, 574–592.

Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. 2013a. Verification as Learning Geometric Concepts. In *SAS 2013 (LNCS)*, Vol. 7935. 388–411.

Rahul Sharma, Aditya V. Nori, and Alex Aiken. 2012. Interpolants as Classifiers. In *CAV 2012 (LNCS)*, Vol. 7358. Springer, 71–87.

Yakir Vizel, Arie Gurfinkel, Sharon Shoham, and Sharad Malik. 2017. IC3 - Flipping the E in ICE. In *VMCAI 2017 (LNCS)*, Vol. 10145. Springer, 521–538. https://doi.org/10.1007/978-3-319-52234-0_28

Qiwen Xu, Willem P. de Roever, and Jifeng He. 1997. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Asp. Comput.* 9, 2 (1997), 149–174.

He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A data-driven CHC solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 707–721. https://doi.org/10.1145/3192366.3192416

He Zhu, Aditya V. Nori, and Suresh Jagannathan. 2015. Learning Refinement Types. In *ICFP 2015*. ACM, 400–411.

He Zhu, Gustavo Petri, and Suresh Jagannathan. 2016. Automatically Learning Shape Specifications. In *PLDI (2016)*. ACM, New York, NY, USA, 491–507. https://doi.org/10.1145/2908080.2908125