Learning Invariants using Decision Trees and Implication Counterexamples

Pranav Garg P. Madhusudan Daniel Neider Dan Roth
University of Illinois at Urbana-Champaign, USA
{garg11, madhu, neider2, danr}@illinois.edu

Abstract

Inductive invariants can be robustly synthesized using a learning model where the teacher is a program verifier who instructs the learner through concrete program configurations, classified as positive, negative, and implications. We propose the first learning algorithms in this model with implication counter-examples that are based on scalable machine learning techniques. In particular, we extend decision tree learning algorithms, building new scalable and heuristic ways to construct small decision trees using statistical measures that account for implication counterexamples. We implement the learners and an appropriate teacher, and show that they are scalable, efficient and convergent in synthesizing adequate inductive invariants in a suite of more than 50 programs, and superior in performance and convergence than existing constraint-solver based learners.

1. Introduction

Automatically synthesizing invariants, in the form of inductive pre / post conditions and loop invariants, is a challenging problem that lies at the heart of automated program verification. If an adequate inductive invariant is found or given by the user, the problem of checking whether the program satisfies the specification can be reduced to logical validity of the verification conditions, which is increasingly tractable with the advances in automated logic solvers.

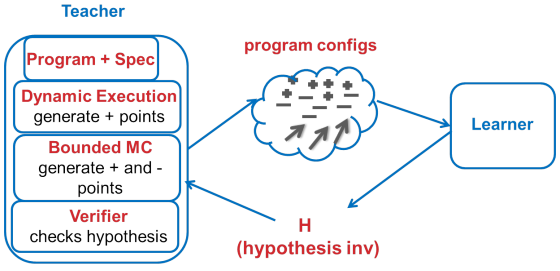


Figure 1: Black-box learning of invariants

In recent years, the *black-box* or *learning* approach to finding invariants has gained popularity [Sharma et al. 2013a,b; Sharma and Aiken 2014; Garg et al. 2013, 2014], in contrast to white-box approaches such as interpolants, methods using Farkas' lemma, IC3, etc. [McMillan 2003; Jhala and McMillan 2006; Colón et al. 2003; Gulwani et al. 2008; Gupta and Rybalchenko 2009; Bradley 2011]. In this data-driven approach, we split the synthesizer of invariants into two parts (see Figure 1). One component is a *teacher*, which is essentially a program verifier that can verify the program using a conjectured invariant and generates counter-examples; it may also have other ways of generating configurations that must or must not be in the invariant (e.g., dynamic execution engines, bounded model-checking engines, etc.). The other component is a *learner*, which learns from counter-examples given by the teacher to synthesize the invariant. In each round, the learner proposes an invariant hypothesis H, and the teacher checks if the hypothesis is adequate to verify the program against the specification; if not, it returns concrete program configurations that are used in the next round by the learner to refine the conjecture. The most important feature of this framework is that the learner is completely agnostic of the program and the specification (and hence the semantics of the programming language, memory model, etc.). The learner is simply constrained to learn some formula (or predicate) that satisfies the sample configurations given by the teacher.

ICE Learning Model The simplest way for the teacher to refute an invariant is to give positive and negative program configurations, S^+ and S^-, constraining the learner to find a predicate that includes S^+ and excludes S^-. However, this is not always possible. In a recent paper, Garg et al. [Garg et al. 2014] note that if the learner gives a hypothesis that covers all states known to be positive by the teacher and excludes all states known to be negative by the teacher, but yet is not *inductive*, then the teacher is stuck and cannot give any positive or negative counter-example to refute the hypothesis.

Garg et al. [Garg et al. 2014] define a new learning model, which they call *ICE* (for implication counter-examples) that allows the teacher to give counterexamples of the form (x, y), where both x and y are program configurations, with the constraint that the learner must propose a predicate such that if it includes x, then it includes y as well. These implication counter-examples can be used to refute non-inductive

invariants— if H is not inductive, then the teacher can find a configuration x satisfying H and where x evolves to y in the program, but y is not satisfied by H. This learning model forms a robust paradigm for learning invariants, including loop invariants, multiple loop invariants, and nested loop invariants in programs [Garg et al. 2014]— the teacher can be both *honest* (never give an example classification that precludes an invariant) and make *progress* (always be able to refute an invariant that is not inductive or adequate). This is in sharp contrast to learning only from positive and negative examples, where the teacher is forced to be dishonest (as it does not know an invariant) in order to make progress.

Machine Learning for Finding Invariants One of the biggest advantages of the black-box learning paradigm has been the possible usage of scalable *machine learning techniques* to synthesize invariants. The learner, being completely agnostic to the program (its programming language, semantics, etc.), can be seen as a machine learning algorithm that learns a Boolean classifier of configurations. Macine Learning algorithms can be trained to learn functions that belong to various Boolean functions, such as k-CNF/k-SNF, Linear Threshold Functions, Decisions Trees, etc. and some algorithms have already been used in invariant generation [Sharma et al. 2012]. However standard machine learning algorithms for classification are trained on given sets of positive and negative examples, and do not handle implications, and hence do not help in building robust learning platforms for invariant generation.

In this paper, we build the first true machine learning algorithms for robust invariant generation. We adapt and extend classical scalable machine learning algorithms for constructing decision trees (which can express any Boolean function) to scalable ICE learning algorithms. We show that this results in scalable and fast algorithms for synthesizing invariants.

Decision trees over a set of attributes provide a universal representation of a Boolean function defined over this set of numerical and categorical attributes. Internal nodes in decision trees are decision variables that split over a value of a single attribute, and the leaves are labeled with classification labels (positive or negative in our setting).

Our starting point is the well-known decision tree learning algorithms of Quinlan [Quinlan 1986, 1993; Mitchell 1997] that work by constructing the tree top-down from positive and negative samples. These are extremely scalable algorithms as they choose heuristically the best attribute that classifies the sample at each stage of the tree based on statistical measures, and do not backtrack nor look ahead. One of the well-known ways to pick these attributes is based on a notion called *information gain*, which is in turn based on a statistical measure using Shannon's entropy function [Shannon 1948; Quinlan 1993; Mitchell 1997]. The inductive bias in these learning algorithms is roughly to compute the *smallest* decision tree that is consistent with the sample— a bias that again suits our setting well, as we would like to construct the smallest invariant formula amongst the large number of invariants that may exist. Machine learning algorithms, including the decision tree learning algorithm, often do not produce concepts that are fully consistent with the given sample— this is done on purpose to avoid over-fitting to the training set, under the assumption that, once learned, the hypothesis will be evaluated on new, previously unseen data. We remove these aspects from the decision tree algorithm (which includes, for example, pruning to produce smaller trees at the cost of inconsistency) as we aim at identifying a hypothesis that is *correct* rather than one that only *approximates* the target hypothesis.

Our first technical contribution is a generic top-down decision tree algorithm that works on samples with implications. This algorithm constructs a tree top-down, classifying end-points of implications in a way that reduces the sizes of trees and manages always to create a tree that is consistent with the sample. Our second technical contribution is a study of various natural measures for learning decisions tree in the presence of positive, negative examples, and implications. We do this by developing several novel "information gain" measures that are used to determine the best attribute to split on given the current collection of examples and implications. The first naive metric is to simply ignore implications when choosing attributes— however, as we find in experiments, this metric does not scale or converge well. The second metric that we propose is a natural extension of the entropy measure to account for implications, where we consider the unclassified examples as probabilistically classified, under the constraints imposed by the implications. A third proposal stems from the observation that in most of the generated invariants, the end-points of implications are mostly classified either both positively or both negatively. Consequently, the third measure we propose is one that uses the classical entropy for positive and negative points, but imposes a *weighted penalty* based on the number of implications that are *cut* by the considered attribute (i.e., where one end-point satisfies the attribute and the other does not), and which do not connect a predominantly negative set to a predominantly positive set. We also tried several other intuitive statistical ways to define heuristic measures to find the best attributes, but the two described above gave the best results.

We implement our ICE decision tree algorithms, implementing the generic ICE learning algorithm with the various statistical measures for choosing attributes, and also build teachers to work with these learners to guide them towards learning invariants. We perform extensive experiments that show that all the decision tree learners we build are competitive, and superior in performance and convergence to the constraint-solving based ICE learner presented in [Garg et al. 2014]. The penalty based method for choosing the attributed performs the best overall. We use our tool to find invariants in around 50 programs, and the new learning techniques work extremely well; surprisingly, all of them converge on

all examples, and the penalty-based tool takes less than 1s on average on each program. We also perform several other experiments— experiments that show, in general, how scalable the decision tree learning is in learning from large sets of points compared with techniques based on constraint solving, and experiments showing that stochastic search based techniques perform erratically, and do not converge on more several examples, and diverge when the space to be learned from is increased.

We believe that this work breaks new ground in adapting machine learning techniques to invariant synthesis, giving the first scalable ICE machine-learning algorithm for synthesizing invariants.

Related Work Algorithms for invariant synthesis can be broadly categorized as white-box techniques and black-box techniques. Prominent examples for white-box techniques include abstract interpretation [Cousot and Cousot 1977], interpolation [McMillan 2003; Jhala and McMillan 2006], and IC3 [Bradley 2011]. Abstract interpretation is predominately used for synthesizing invariants over convex domains [Miné 2001; Cousot and Halbwachs 1978; Ball et al. 2001; Henzinger et al. 2002], but there also exist applications to non-convex domains [Filé and Ranzato 1999; Sankaranarayanan et al. 2006] and even non-lattice domains [Gange et al. 2013]. Template-based approaches for synthesizing invariants using constraint solvers have been also explored in a white-box setting in [Colón et al. 2003; Gulwani et al. 2008; Gupta and Rybalchenko 2009]. On the other hand, a prominent early example of black-box techniques for synthesizing invariants is Daikon [Ernst et al. 2000], which uses conjunctive Boolean learning to find *likely* invariants from test runs. Actual learning in the context of verification has been introduced by Cobleight et al. [Cobleigh et al. 2003], followed by applications of Angluin's L* [Angluin 1987] to verification problems such as finding rely-guarantee contracts [Alur et al. 2005b] and stateful interface specifications for programs [Alur et al. 2005a], verifying CTL properties [Vardhan and Viswanathan 2007], and Regular Model Checking [Neider and Jansen 2013]. Houdini [Flanagan and Leino 2001], which learns conjunctive invariants using conjunctive Boolean learning, is another prominent algorithm. [Kawaguchi et al. 2009] uses a similar algorithm for inferring refinement types for program expressions and use them to statically verify memory safety.

Recently, learning has gained renewed interest in the context of program verification, particularly for synthesizing loop invariants [Sharma et al. 2012, 2013a,b, 2014; Garg et al. 2013]. In a recent paper, Garg et al. [Garg et al. 2014] argue that merely learning from positive and negative examples for synthesizing invariants is inherently non-robust as it lacks a way to communicate why a hypothesis is not inductive. To overcome this problem, they introduce ICE-learning, which extends the classical learning setting with implications. Examples of algorithms using ICE-learning have been proposed for learning invariants over octogonal constrains and universally quantified invariants over linear data structures [Garg et al. 2013], Regular Model Checking [Neider 2014], and a general framework for generating invariants based on randomized search [Sharma and Aiken 2014]. Some generalizations of Houdini [Thakur et al. 2013; Garoche et al. 2013] can also be seen as ICE-learning algorithms.

Machine learning algorithms (see [Mitchell 1997] for an overview) are often used in practical learning scenarios due to their high scalability. Amongst the most well-known machine learning algorithms are the winnow algorithm [Littlestone 1987], perceptron [Rosenblatt 1958], and support vector machines [Cortes and Vapnik 1995] for learning linear classifiers, and decision tree algorithms, such as ID3 [Quinlan 1986] and C4.5 [Quinlan 1993], respectively C5.0 (on which we build in this work) for learning arbitrary Boolean functions. Several learning models like PAC [Kearns and Vazirani 1994] and exact learning by Angluin [Angluin 1988] are well known in machine learning literature. However, the learning setting closest to ICE-learning is perhaps the mistake bound learning model [Littlestone 1987], in which a learner iteratively learns from incorrectly classified data and needs to converge to a correct classifier within a bounded number of wrong conjectures. However, we are not aware of any machine learning algorithm designed to learn from positive and negative data as well as implications.

More broadly, in *program* or *expression* synthesis, a popular approach to synthesis is using counter-example guided inductive synthesis (CEGIS), which is also a black-box learning paradigms [Alur et al. 2013; Solar Lezama 2008], is gaining traction, aided by explicit enumeration, symbolic constraint-solving and stochastic search algorithms.

2. Background: Learning Decision Trees from Positive and Negative Examples

Our algorithm for learning invariants builds on the classical recursive algorithms to build *decision trees*. (We refer the interested reader to standard texts on learning for more information on decision tree learning [Mitchell 1997].) In particular, our learning algorithms are based on learning decision trees using the algorithms of Quinlan, where the tree is built top-down, choosing the best attribute at each stage using an information theoretic measure that computes the information gain in applying the attribute (quantified as the decrease in entropy on the resulting divided samples), and as implemented by the ID3, C4.0, and C5.0 algorithms [Quinlan 1986, 1993; Mitchell 1997].

We now describe the classical decision tree learning algorithm in the highest detail level possible that allows us to explain how we extend it to samples that include implications.

Before we delve into this, let us give the context in which these algorithms will be used for invariant synthesis. The reader is encouraged to think of decision trees as learning Boolean combinations of formulae of the form $a_i \leq c$, where a_i is drawn from a fixed set of *numerical attributes*

A (which assign a numerical value to each sample and c is a constant of the form b_i, where b_i is drawn from a fixed set of *Boolean attributes* (which assign a truth value to each sample). When performing invariant learning, we will fix a set of attributes typically as certain arithmetic combinations of integer variables (for example, octagonal combinations of variables or certain linear combinations of variables with bounded co-efficients). Boolean attributes are useful for other non-numerical constraints (are x and y aliased, does x point to *nil*, etc.). Consequently, the learner would learn the thresholds (the values for c in $a_i \leq c$) and the Boolean combination of the resulting predicates, including arbitrary conjunctions of disjunctions as a proposal for the invariant.

The Basic ID3 Algorithm

Given a set of positive and negative samples, the classical approach to building a decision tree works by constructing the tree top-down (without backtracking), and is sketched in pseudo code as Algorithm 1. In each step, the algorithm chooses an attribute a from the given set of attributes (and possibly a threshold c if the attribute is numeric), and then applies the predicate defined by the attribute to the sample; this splits the sample into two sets, distinguished by the predicate— those satisfying the predicate and those that do not satisfy it. The algorithm then recurses on these two sets of samples, building decision trees for them, and then returns a tree where the root is labeled with the predicate and the left and right subtrees are those returned by the recursive calls. The base cases are when the sample is entirely positive or entirely negative—in this case the algorithm returns a tree with a single node that assigns the classification of these points appropriately as true or false, respectively.

In the settings where we use learning, we will assume that all integer variables mentioned in the program occur explicitly as numerical attributes; hence it turns out that *any* sample of mixed positive and negative points can be split, and eventually separated into purely positive and purely negative points (in the worst case, each point may be described *precisely* by reaching a leaf of the tree). Consequently, we are guaranteed to always obtain some decision tree that is consistent with any input sample.

The crucial aspect of the extremely scalable decision tree learning algorithms is that they choose the attribute for the current sample in some heuristic manner, and never back-track (or look forward) to optimize the size of the decision tree. Typically, we spend linear time in analyzing the current sample against the various attributes, in order to choose the attribute we think is best in classifying the sample. For numerical attributes, the thresholds also need to be synthesized; in most of the classifications based on information theory (see below), it turns out that the best thresholds will be always at the constants defined by the points in the sample. Figuring out the precise attribute and threshold in the case of numerical attributes will typically take more than linear time on the current sample.

input : A sample $S = \langle S^+, S^- \rangle$ and $Attributes$

Return `ID3` $(\langle S^+, S^- \rangle, Attributes)$.

Proc `ID3` *(Examples* $= \langle Pos, Neg \rangle$*, Attributes)*

- Create a root node of the tree.
- **if** *all examples are positive or all are negative* **then**
  - Return the single node tree Root with label $+$ or $-$, respectively.
- **else**
  - Select an attribute a (and a threshold c for a if a is numerical) that (heuristically) *best* classifies *Examples*.
  - Label the root of the tree with this attribute (and threshold).
  - Divide *Examples* into two sets: $Examples_a$ that satisfy the predicate defined by attribute (and threshold), and $Examples_{\neg a}$ that do not.
  - Return tree with root and left tree `ID3` $(Examples_a, Attributes \setminus \{a\})$ and right subtree `ID3` $(Examples_{\neg a}, Attributes \setminus \{a\})$
- **end**

Algorithm 1: The basic inductive decision tree construction algorithm underlying ID3, C4.0, and C5.0

The prominent technique for learning decision trees uses a statistical property, called *information gain*, to measure how well each attribute classifies the examples at any stage of the algorithm. This measure is often defined using a notion called *Shannon entropy* [Shannon 1948], which, intuitively, captures the impurity of a sample, and can be seen as the number of bits required to encode the classification of the sample. The entropy of a sample S with p positive samples and n negative samples is a value between 0 and 1, defined to be

$$Entropy(S) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}.$$

Intuitively, if the sample contains only positive (or only negative) points (i.e., if $p = 0$ or $n = 0$), then its entropy is 0, while if the sample had roughly an equal number of positive and negative points, then its entropy is close to 1. Samples of lower entropy indicate that the classification is more complete, and hence are preferred.

When evaluating an attribute a (and threshold) on a sample S, splitting it into S_a and $S_{\neg a}$ (those satisfying the attribute and those not), we compute the information gain of that attribute: the information gain is the difference between the entropy of S and the sum of the entropies of S_a and $S_{\neg a}$ weighted by the number of points in the respective samples. The algorithm chooses the attribute that results in the largest information gain. So, for example, an attribute that results in a split that causes two samples each with roughly half positive and half negative points will be less preferred than an attribute that results in a split that causes skewed samples (more positive than negative or vice versa).

The above heuristic for greedily picking the attribute that works best at each level has been shown to work very well

in large and a wide variety of machine learning applications [Quinlan 1993; Mitchell 1997]. When decision tree learning is used in machine learning contexts, there are other important aspects: (a) the learning is achieved using a small portion of the available sample, so that the tree learned can be evaluated for accuracy against the rest of the sample, and (b) there is a *pruning* procedure that tries to generalize and reduce the size of the tree so that the tree does not overfit the sample. In the setting of using decision tree learning for synthesizing invariants, we prefer to use all the samples as we anyway place the passive learning algorithm in a larger context by building a teacher which is a verification oracle. Also, we completely avoid the pruning phase since pruning often produces trees that are (mildly) inconsistent with the sample; since we cannot tolerate any inconsistent trees, we prefer to avoid this (incorporating pruning in a meaningful and useful way in our setting is an interesting future direction).

3. A Generic Decision Tree Learning Algorithm in the Presence of Implications

In this section, we present the skeleton of our new decision tree learning algorithm for samples containing implication examples in addition to positive and negative examples. We present this algorithm at the level of Algorithm 1, excluding the details of *how* the best attribute at each stage is chosen. In Section 4, we articulate several different natural ways of choosing the best attribute, and evaluate them in experiments.

Our goal in this section is to build a top-down construction of a decision tree for an *ICE sample*, such that the tree is guaranteed to be consistent with respect to the sample; an ICE sample is a tuple $S = (S^+, S^-, S^{\Rightarrow})$ consisting of a finite set S^+ of positive points, a finite set S^- of negative points, and a finite set $S^{\Rightarrow}$ of pairs of points corresponding to the implications. The algorithm is an extension of the classical decision tree algorithm presented in Section 2 and, hence, will automatically be consistent with positive and negative samples. The main hurdle we need to cross is to construct a tree consistent with the implications. Note that the pairs of points corresponding to implications do not have a classification, and it is the learner's task to come up with a classification in a manner consistent with the implication constraints. As part of the design, we would like the learner to not classify the points a priori in any way, but classify these points in a way that leads to a smaller concept (or tree).

Our algorithm, shown in pseudo code as Algorithm 2, works as follows. First, given an ICE sample $\langle S^+, S^-, S^{\Rightarrow}\rangle$ and a set of attributes, we store $S^{\Rightarrow}$ in a global variable *Impl* and create a set *Unclass* of unclassified points as the end-points of the implication samples. We also create a global table G that holds the partial classification of all the unclassified points (initially empty). We then call our recursive decision tree constructor with the sample $\langle S^+, S^-, \mathit{Unclass}\rangle$.

Receiving a sample $\langle \mathit{Pos}, \mathit{Neg}, \mathit{Unclass}\rangle$ of positive, negative, and unclassified examples, our algorithm chooses the best attribute that divides this sample, say a, and recurses on the two resulting samples $\mathit{Examples}_a$ and $\mathit{Examples}_{\neg a}$. Unlike the classical learning algorithm, we do not recurse *independently* on the two sets—rather we recurse first on $\mathit{Examples}_a$, which will, while constructing the left subtree, make classification decisions on some of the unclassified points, which in turn will affect the construction of the right subtree for $\mathit{Examples}_{\neg a}$ (see the `else` clause in Algorithm 2). The new classifications that are decided by the algorithm are stored and communicated using the global variable G.

input : An ICE sample $S = \langle S^+, S^-, S^{\Rightarrow}\rangle$ and *Attributes*

1 Initialize global *Impl* to $S^{\Rightarrow}$.
2 Initialize G, a partial valuation of end-points of implications in $S^{\Rightarrow}$, to be empty.
3 Let *Unclass* be the set of all end-points of implications in $S^{\Rightarrow}$.
4 Take implication closure of G with respect to the positive and negative classifications in S^+ and S^-.
5 Return `DecTreeICE` $(\langle S^+, S^-, \mathit{Unclass}\rangle, \mathit{Attributes})$.

Proc `DecTreeICE` *(Examples* $= \langle \mathit{Pos}, \mathit{Neg}, \mathit{Unclass}\rangle$, *Attributes)*

6 Move all points from *Unclass* that are positively, respectively negatively, classified in G to *Pos*, respectively *Neg*.
7 Create a root node of the tree.
8 **if** $\mathit{Neg} = \emptyset$ **then**
9 Mark all elements in *Unclass* as positive in G.
10 Take the implication closure of G w.r.t. *Impl*.
11 Return the single node tree Root, with label $+$.
12 **else if** $\mathit{Pos} = \emptyset$ **then**
13 Mark all elements in *Unclass* as negative in G.
14 Take the implication closure of G wrt *Impl*.
15 Return the single node tree Root, with label $-$.
16 **else**
17 Select an attribute a (and a threshold c for a if a is numerical) that (heuristically) *best* classifies *Examples* and *Impl*.
18 Label the root of the tree with this attribute a (and threshold c).
19 Divide *Examples* into two sets: $\mathit{Examples}_a$ that satisfy the predicate defined by the attribute (and threshold) and $\mathit{Examples}_{\neg a}$ that do not.
20 $T_1 = \mathit{DecTreeICE}(\mathit{Examples}_a, \mathit{Attributes})$ (may update G).
21 $T_2 = \mathit{DecTreeICE}(\mathit{Examples}_{\neg a}, \mathit{Attributes})$ (may update G).
22 Return tree with root, left tree T_1, and right tree T_2.
end

Algorithm 2: The basic inductive decision tree construction algorithm for ICE samples

Whenever Algorithm 2 reaches a node where the current sample has only positive points and implication end-points that are either classified positively or unclassified yet, then the algorithm will, naturally, decide to mark *all* remaining

unclassified points positive, and declare the current node to be a leaf of the tree (see first conditional in the algorithm). Moreover, it marks in G that all the unclassified end-points of implications in *Unclass* as positive and propagates this constraint across implications (taking the implication closure of G with respect to the global set $Impl$ of implications). For instance, if (x, y) is an implication pair, both x and y are yet unclassified, and the algorithm decides to classify x as positive, it propagates this constraint by making y also positive in G; similarly, if the algorithm decided to classify y as negative, then it would mark x also as negative in G. Deciding to classify x as negative or y as positive places no restrictions on the other point, of course.

We need to argue why Algorithm 2 always results in a terminating procedure that constructs a decision tree consistent with the sample. First, we ensure at every stage that the valuations forced by the implications are always propagated across the partial valuation G we have constructed thus far. More precisely, we always maintain that the original sample S and the partial valuation for the implication end-points G computed so far has the following property, called a *valid sample*. In the description below, we refer to $S \cup G$ as the combined valuation defined on the points by S^+, S^-, and G.

Definition 1 (Valid sample)**.** A sample S is *valid* if for every implication $(x, y) \in S^{\Rightarrow}$

- it is not the case that x is classified positively and y is classified negatively in $S \cup G$;
- it is not the case that x is classified positively and y is unclassified in $S \cup G$; and
- it is not the case that y is classified negatively and x is unclassified in $S \cup G$.

A valid sample has the following interesting property.

Lemma 1. *For any valid sample (with partially classified end-points of implications), extending it by classifying all unclassified points as positive will result in a consistent classification, and extending it by classifying all unclassified points as negative will also result in a consistent classification.*

Proof. The above lemma is easy to prove: consider the extension of a valid sample by classifying all unclassified points as positive. Assume, by way of contradiction, that this valuation inconsistent. Then, there is some implication pair (x, y) such that x is classified as positive and y is classified as negative. Since such an implication pair could not have already existed in the valid sample (by definition), it must have been caused by the extension. Since we introduced only positive classifications, it must have been that x (and not y) is the only new classification. Hence the valid sample must have had the implication pair (x, y) with y classified as negative and x being unclassified, which contradicts the definition of a valid sample. The proof of the extension using negative classifications can be shown using a similar argument. □

Note that Algorithm 2 always maintains a valid sample. When we have a valid sample and are at a leaf where the algorithm is working on a subsample that has only positive points and unclassified points, it would classify all these unclassified points to be positive. Since we have a valid set and the extension only causes positive classifications, the sample cannot be inconsistent (classifying all points as positive, after all, will satisfy all implications, by Lemma 1). Moreover, at this point we will take the implication closure of the new positively classified points (if (x, y) is an implication pair, and x is classified positive, we will also make y positive), and hence maintain a valid sample. A similar argument holds when the algorithm creates a negative leaf node.

The above argument shows that our algorithm will classify unclassified points as it proceeds, and will never end up with an inconsistent sample; also given that there are attributes that can always split any set with more than one element, this proves termination as well. Under this assumption, we obtain the following result.

Theorem 2. *Algorithm 2, independent of how the attributes are chosen to split a sample, always terminates and produces a decision tree that is consistent with the input ICE sample.*

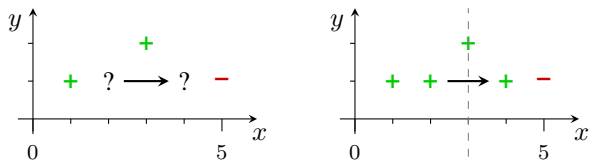
The running time of Algorithm 2 depends, of course, on the time taken to choose the best attribute in each recursive call. However, notice that apart from this, the algorithm is linear in the size of the sample— in each round, the sample set is divided into two parts and recursed on, and the total updates to G and the sum of the implication closures on G can be done in time linear in the number of implications. We will explore several different ways to choose the best attribute at each stage in the next section, but all these will be linear or quadratic on the sample.

4. Choosing Attributes in the Presence of Implications

Algorithm 2 ensures that the resulting decision tree is consistent with the given sample, irrespective of the exact mechanism used to determine the attributes to split and their thresholds. As a consequence, the original split heuristic based on information gain (see Section 2), which is unaware of implications, might simply be employed. However, since our algorithm propagates the classification of data points once a leaf node has been reached, just ignoring implications can easily lead to seemingly good splits that later turn into bad ones. The following example illustrates such a situation.

Example 1. Suppose that Algorithm 2 processes the sample shown in Figure 2a, which also depicts the (only) implication in the global set $Impl$.

When using the original split procedure (i.e., using information gain while ignoring the implication and its corresponding unclassified data points), the learner splits the sample with respect to attribute x at threshold $c = 3$ since this split yields the highest information gain—the information gain is 1 since the entropy of the resulting two subsamples is 0. Using this split, the learner partitions the sample into $Examples_a$ and



(a) Sample before splitting (b) Samples after splitting

Figure 2: The samples discussed in Example 1.

$Examples_{\neg a}$ and recurses on $Examples_a$. Since $Examples_a$ contains only unclassified and positively classified points, it turns all unclassified points in this sample positive and propagates this information along the implication. This results in the situation depicted in Figure 2b. Note that the learner now needs to split $Examples_{\neg a}$ since the unclassified data points in it are now classified positively.

On the other hand, considering the implication and its corresponding data points allows splitting the sample such that the antecedent and the consequent of the implication both belong to either $Examples_a$ or $Examples_{\neg a}$ (e.g., with respect to x and threshold $c = 2$). Such a split has the advantage that no further splits are required and, hence, results in a smaller tree. ⌟

As our experiments (and the example above) showed, ignoring implications when choosing an attribute and a threshold (i.e., using the original information gain) often results in large decision trees. In order to construct smaller trees, we next propose two methods that take implications into account. We also observed that these two methods yielded faster convergence when used in an invariant synthesis setting.

4.1 Computing Information Gain for an ICE Sample

The entropy of a set of examples is a function of the discrete probability distribution of the classification of a point drawn randomly from the examples. In a classical sample that only has points labeled positive or negative, one could count the fraction of positive (or negative) points in the set to compute these probabilities. However, an estimation of these probabilities becomes non-trivial in the presence of unclassified points that can be probabilistically classified as either positive or negative. Moreover, the classification of these points is not independent anymore as the classification for the points need to satisfy the implication constraints. Given a set of examples with implications and unclassified points, we will first estimate the probability distribution of the classification of a random point drawn from these examples, taking into account the implication constraints, and then use it for computing the entropy. We will use this new entropy to compute the information gain while deciding on the best attribute.

Given $Examples = \langle Pos, Neg, Unclass \rangle$, and a set of implications $Impl$, let $Impl_{Examples}$ be the set of implications projected onto the $Examples$ such that both the antecedent and consequent points in the implication are unclassified (i.e., $Impl_{Examples} = \{(x_1, x_2) \in Impl \mid x_1, x_2 \in Unclass\}$). For the purpose of entropy computation, we will assume that there is no point in the examples that is common to more than one implication. This is a safe assumption if the space enclosing all the points is much larger than the number of points and was validated by our experiments. Let $Unclass' \subseteq Unclass$ be the set of unclassified points in the sample that are not part of any implication in $Impl_{Examples}$ (as one example, $x_1 \in Unclass'$ if $(x_1, x_2) \in Impl$ and $x_2 \in Pos$). Note that points in $Unclass'$ can be classified as positive or negative, independent of the classification of other points.

Let $Pr(x = c)$ be the probability of a point $x \in Examples$ being classified as c for c being positive $(+)$ or negative $(-)$. Note that $Pr(x = +)$ is one for $x \in Pos$ and zero for $x \in N$. Also, let $Pr(Examples, c)$ be the probability of a point drawn randomly from $Examples$ being classified as c. Then,

$$Pr(Examples, +) = \frac{1}{|Examples|} \sum_{x \in Examples} Pr(x = +)$$

$$= \frac{1}{|Examples|} \Big(\sum_{x \in Pos \cup Neg \cup Unclass'} Pr(x = +) \; + \sum_{(x_1, x_2) \in Impl_{Examples}} Pr(x_1 = +) \; + \; Pr(x_2 = +) \Big) \quad (1)$$

We assume that unclassified points consisting of $x_U \in Unclass'$ and $x_1 \in Examples$ where $(x_1, x_2) \in Impl_{Examples}$ are classified independently and probabilistically in accordance with the distribution of points in the total set $Examples$. In other words, we recursively assign $Pr(x_U = +) = Pr(x_1 = +) = Pr(Examples, +)$. However, the classification of x_2 is dependent on the classification of x_1 as the implication constraint between them needs to be satisfied by any classification. Using conditional probabilities we obtain,

$$Pr(x_2 = +) = Pr(x_2 = + \mid x_1 = +)\, Pr(x_1 = +) + Pr(x_2 = + \mid x_1 = -)\, Pr(x_1 = -)$$

Now, if x_1 is classified positively then x_2 should be positive as well; so $Pr(x_2 = + \mid x_1 = +) = 1$. If x_1 is however classified negatively then x_2 can take any classification and its classification is determined by the distribution of points in the set of examples (i.e., $Pr(x_2 = + \mid x_1 = -) = Pr(Examples, +)$). Plugging in these values for probabilities in Equation 1 and using $p = |Pos|, n = |Neg|, i = |Impl|, u' = |Unclass'|$ and $|Examples| = p + n + 2i + u'$, we can compute $Pr(Examples, +)$ as a positive solution of the following quadratic equation:

$$ix^2 + (p + n - i)x - p = 0$$

As a sanity check, note that $Pr(Examples, +)$ obtained by solving the above equation defaults to the fraction of positive points, $\frac{p}{p+n}$, if there are no implications in the sample set (i.e., $i = 0$). Also, $Pr(Examples, +) = 1$ if $n = 0$ and the set of examples consists exclusively of positive and unclassified points. Similarly, $Pr(Examples, +) = 0$ if $p = 0$ and the set of examples consists exclusively of negative and unclassified points. Once we have computed

$Pr(Examples, +)$, the entropy of *Examples* can be computed in the standard way as

$$\begin{aligned} &Entropy(Examples) = \\ &\quad - Pr(Examples, +)\ \log_2\ Pr(Examples, +) \\ &\quad\quad - Pr(Examples, -)\ \log_2\ Pr(Examples, -), \end{aligned}$$

where $Pr(Examples, -) = \big(1 - Pr(Examples, +)\big)$. Now, we plug this new entropy in the information gain and obtain a gain measure that explicitly takes implications into account.

4.2 Penalizing Cutting Implications

In order to better understand how to deal with implications, we analyzed classifiers learned by other ICE-learning algorithms for invariant synthesis, such as the randomized search of [Sharma and Aiken 2014] and the constraint solver-based ICE learner of [Garg et al. 2014]. This analysis showed that the classifiers finally learned (and also those conjectured during the learning) almost always classify the antecedent and consequents of implications equally (either both positive or both negative).

The fact that successful ICE learners almost always classify antecedents and consequents of implications equally suggests that our decision tree learner should avoid to "cut" implications. Formally, we say that an implication $(x, y) \in Impl$ is *cut* by the samples S_a and $S_{\neg a}$ if $p \in S_a$ and $q \in S_{\neg a}$, or $p \in S_{\neg a}$ and $q \in S_a$;[1] in this case, we also say that the split of S into S_a and $S_{\neg a}$ cuts the implication.

A straight-forward approach to discourage cutting implications builds on top of the original information gain and imposes a penalty for every implication that is cut. This idea gives rise to the *penalized information gain* that we define by

$$\begin{aligned} &Gain_{pen}(S, S_a, S_{\neg a}) = \\ &\quad Gain(S, S_a, S_{\neg a}) - Penalty(S_a, S_{\neg a}, Impl), \end{aligned}$$

where $S_a, S_{\neg a}$ is the split of the sample S, $Gain(S, S_a, S_{\neg a})$ is the original information gain based on Shannon's entropy, and $Penalty(S_a, S_{\neg a}, Impl)$ is a total penalty function that we assume to be monotonically increasing with the number of implications cut (we make this precise shortly). Note that this new information gain does not prevent the cutting of implications (if this is necessary) but favors not to cut them.

However, not every cut implication poses a problem: implications whose antecedents are classified positively and whose consequents are classified negatively are safe to cut (as this helps creating more pure samples), and we do not want to penalize cutting those. However, since we do not know the classifications of unclassified points when choosing an attribute, we penalize a cut implication depending on how "likely" its an implication is of this type (i.e., we assign no penalty if the sample containing the antecedent is predominantly negative and the one containing the consequent is predominantly positive). More precisely, given the samples S_a and $S_{\neg a}$, we define the penalty function $Penalty(S_a, S_{\neg a}, Impl)$ by

$$\Big(\sum_{\substack{(x,y)\in Impl \\ x\in S_a, y\in S_{\neg a}}} 1 - f(S_a, S_{\neg a})\Big) + \Big(\sum_{\substack{(x,y)\in Impl \\ x\in S_{\neg a}, y\in S_a}} 1 - f(S_{\neg a}, S_a)\Big),$$

where for two samples $S_1 = \langle Pos_1, Neg_2, Unclass_3\rangle$ and $S_2 = \langle Pos_2, Neg_2, Unclass_2\rangle$

$$f(S_1, S_2) = \frac{|Neg_1|}{|Pos_1| + |Neg_1|} \cdot \frac{|Pos_2|}{|Pos_2| + |Neg_2|}$$

is the relative frequency of the negative points in S_1 and the positive points in S_2 (which can be interpreted as likelihood of an implication pointing from S_1 to S_2 being safe).

5. Experiments and Evaluation

To assess the performance of our decision tree ICE learner, we implemented a prototype of Algorithm 2 as an invariant synthesis tool in Boogie [Barnett et al. 2005] and benchmarked it to other ICE learning algorithms. Our experimental setup is as follows.

Learner We implemented Algorithm 2 (with the attribute selection methods described in Section 4) on top of the freely available version of the C5.0 algorithm (Release 2.10). Since we rely on learning without classification errors, we disabled all of C5.0's optimizations, such as pruning, boosting, etc.

Teacher We implemented a teacher in Boogie, which translates an input program into verification conditions and prototypes of functions that need to be synthesized. Before the learning starts, the teacher and learner agree on the number and names of the variables (which depend on the program to verify). Since we are interested in octagonal constraints, we add auxiliary attributes of the form $x + y$, $x - y$, and $-x + y$ for all combinations of variables x, y of the program (totaling in $3n^2$ additional attributes if n is the number of variables). Note that the learner figures out the thresholds (i.e., c in the constraint $x + y \leq c$) as well as the Boolean combination of them. These auxiliary attributes are natively supported by C5.0 and can be arbitrary Boolean or arithmetic expressions.

The actual learning starts with an empty ICE sample. Whenever the learner proposes a conjecture, the learner checks whether it satisfies the verification conditions. If this is not the case, he derives a counterexample—either a spurious pos/neg data point or an implication—and returns it to the learner. The teacher favors to return pos/neg data points, and returns implications only if no spurious pos/neg data points can be found (we found that this performed best for our learners). Since loop invariants usually do not involve large constants, we employed the following search strategy: when searching for counterexamples, we successively bound the absolute values of the variables to 2, 5, and 10 and successively proceed to the next larger bound if no counterexample within the current bound exists; if no counterexample within

[1] Given a sample $S = \langle Pos, Neg, Unclass\rangle$, we write $x \in S$ as a shorthand notation for $x \in Pos \cup Neg \cup Unclass$.

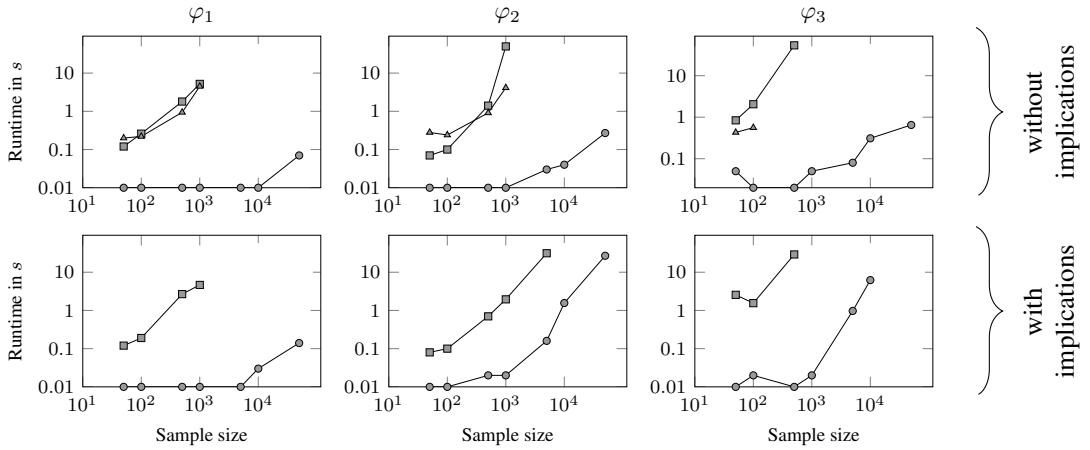


Figure 3: Results of the scalability benchmark

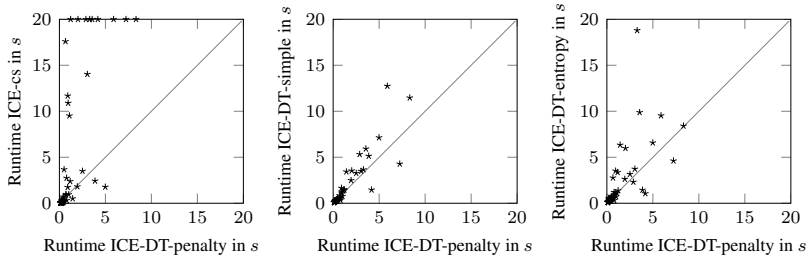


Figure 4: Runtime comparison of ICE-DT-penalty to ICE-CS, ICE-DT-simple and ICE-DT-entropy. Each point in the diagrams correspond to a program in Table 1. Points above the diagonal indicate that ICE-DT-penalty is faster than the other learner.

any of these bounds exists, we fall back to the most general case and do not impose any bound.

Experimental Setup We conducted all experiments on a Core i5 CPU with 6 GiB of RAM running Windows 7. We used a 60 s timeout limit for all experiments.

5.1 Scalability Benchmark

We begin with a benchmark demonstrating the scalability of the decision tree ICE learner compared to other learning techniques, namely the constraint solver-based ICE learner of [Garg et al. 2014] and the learner based on computational geometry from [Sharma et al. 2013b] (which can learn from positive and negative data only). This benchmark comprises six sample suits, each consisting of randomly drawn samples of increasing size (containing between 50 and 50,000 data points each). Two suits are classified by $\varphi_1 = x_1 \leq -1 \vee y \geq 1$, two by $\varphi_2 = x_1 - x_2 \geq 2$, and two by $\varphi_3 = 0 \leq x_0 \wedge 0 \leq x_1 \wedge 0 \leq x_2 \wedge x_3 \neq 1 \wedge x_4 \neq 1 \wedge x_5 \neq 1$, each such that half of the data points are positive and half are negative. For each two suits classified by the same formula, the first contains no implications (i.e., all points are positive or negative), whereas $\frac{1}{4}$ of the data points in the second suite occur in an implication and, hence, are classified as unknown.

The results of this benchmark are shown in Figure 3. The upper row shows the results on samples without implications, wheres the lower row shows results on samples with implications; note that the times taken by the learner from [Sharma et al. 2013b] is only shown in the upper row as it does not support implications. Except for one sample suite, the decision tree-based learner can handle samples up to 50 000 data points, whereas the constraint solver-based learner and the learner from [Sharma et al. 2013b] time out or run out of memory on samples with 1000 data points (with one exception). As Figure 3 shows, the decision tree-learner is on average one order of magnitude faster then the other algorithms while learning formulas that are roughly of the same "size" (which is not visualized in the figure). *In total, the benchmark shows that the decision tree-based ICE learner scales much better then other ICE-learners described in literature*, which motivates the use of machine-learning based tools for invariant synthesis. Note, however, that these are simply *scalability* microbenchmarks to compare the learners, and do not measure their efficacy in actually learning invariants; we consider this latter question next.

Program	ICE-CS			ICE-DT-simple			ICE-DT-entropy			ICE-DT-penalty		
	P,N,I	#R	T(s)	P,N,I	#R	T(s)	P,N,I	#R	T(s)	P,N,I	#R	T(s)
afnp	1,19,15	29	3.6	1,3,7	11	0.5	1,3,10	14	0.7	1,2,8	11	0.5
array	4,7,11	14	0.5	2,15,45	61	3.4	15,16,67	95	6.3	4,6,16	25	1.4
arrayn	4,7,5	7	0.3	2,1,2	5	0.2	2,1,2	5	0.2	2,1,2	5	0.2
arrayn.v	6,8,6	8	0.3	2,4,1	7	0.3	3,3,1	6	0.3	3,3,1	6	0.7
cegar1	1,1,1	3	0.0	3,1,1	5	0.2	3,1,1	5	0.2	3,1,1	5	0.2
cegar1.v	1,1,1	3	0.0	7,2,1	9	0.4	7,2,1	9	0.4	7,2,1	9	0.4
cegar2	**4,20,14**	**28**	**9.5**	**5,10,15**	**30**	**1.3**	**5,10,8**	**23**	**1.0**	**5,9,9**	**22**	**1.0**
cegar2.v	**Timeout**			**11,40,19**	**68**	**3.5**	**11,50,33**	**89**	**5.9**	**12,36,12**	**55**	**2.0**
cggmp05	**1,36,50**	**71**	**51.1**	**1,12,53**	**66**	**4.2**	**1,12,55**	**68**	**4.6**	**1,17,50**	**68**	**7.2**
dec	1,1,1	3	0.0	1,2,0	3	0.2	1,2,0	3	0.4	1,2,0	3	0.2
decn.v	2,4,3	7	0.1	5,4,1	9	0.3	5,4,1	9	0.3	5,4,1	9	0.3
ex14	2,5,1	7	0.0	1,1,0	2	0.1	1,1,0	2	0.1	1,1,0	2	0.1
ex14n	2,2,1	4	0.0	1,1,0	2	0.1	1,1,0	2	0.0	1,1,0	2	0.1
ex14n.v	8,8,7	10	0.1	3,4,0	5	0.2	3,4,0	5	0.5	3,4,0	5	0.2
ex23	**5,32,40**	**69**	**17.5**	**5,8,1**	**11**	**0.5**	**5,21,12**	**34**	**2.7**	**5,8,1**	**11**	**0.6**
ex23.v	**14,39,53**	**78**	**48.7**	**9,25,5**	**31**	**1.4**	**9,14,4**	**20**	**1.0**	**11,16,49**	**68**	**4.1**
ex7	1,2,1	2	0.0	1,1,0	2	0.1	1,1,0	2	0.1	1,1,0	2	0.1
ex7.v	1,2,1	2	0.0	1,1,0	2	0.1	1,1,0	2	0.1	1,1,0	2	0.1
fig1	2,5,1	6	0.1	2,4,1	6	0.2	2,4,1	6	0.2	2,4,1	6	0.4
fig1.v	6,11,5	14	0.5	3,5,1	8	0.3	4,5,1	8	0.4	4,6,1	9	0.3
fig3	2,4,2	6	0.1	4,5,0	6	0.2	4,5,0	6	0.3	4,5,0	6	0.2
fig3.v	5,8,5	8	0.2	5,7,0	8	0.3	5,7,0	8	0.4	5,7,0	8	0.4
fig9	0,2,0	2	0.0	1,1,0	2	0.1	1,1,0	2	0.1	1,1,0	2	0.1
fig9.v	0,2,0	2	0.0	1,1,0	2	0.1	1,1,0	2	0.0	1,1,0	2	0.1
inc	**3,12,101**	**112**	**1.7**	**4,5,106**	**114**	**7.1**	**9,6,104**	**117**	**6.5**	**5,3,100**	**106**	**4.9**
form22	1,18,11	22	1.8	1,10,43	54	2.4	1,13,34	48	2.6	1,8,32	41	1.9
form25	**1,46,30**	**49**	**14.0**	**1,65,7**	**73**	**3.4**	**1,71,6**	**78**	**3.7**	**1,62,4**	**67**	**3.0**
form27	**Timeout**			**1,194,25**	**220**	**12.7**	**1,142,13**	**156**	**9.5**	**1,104,15**	**120**	**5.8**
incn	3,4,3	8	0.1	3,3,1	7	0.3	3,3,1	7	0.3	4,3,3	10	0.4
incn.v	9,10,6	11	0.2	3,4,1	7	0.3	4,3,1	7	0.2	5,4,1	8	0.3
matrix1	2,9,3	8	0.3	5,5,2	8	0.4	5,5,2	8	0.5	5,5,2	8	0.5
matrix1n	4,12,4	8	0.9	2,7,2	7	0.4	4,9,2	8	0.5	5,11,2	9	0.7
matrix1n.v	2,13,4	7	0.9	6,13,1	10	0.7	6,13,1	10	0.7	8,19,2	14	1.0
matrix2	**8,19,13**	**27**	**22.9**	**9,10,7**	**24**	**1.4**	**9,10,7**	**24**	**1.3**	**9,10,5**	**22**	**1.2**
matrix2n	**Timeout**			**20,37,29**	**82**	**5.9**	**18,52,38**	**107**	**9.8**	**17,36,16**	**66**	**3.5**
matrix2n.v	**Timeout**			**20,88,39**	**143**	**11.4**	**32,60,11**	**101**	**8.4**	**17,73,19**	**101**	**8.3**
nc11	5,15,7	18	0.7	2,6,5	12	0.4	3,6,5	13	0.6	2,4,4	9	0.4
nc11n	4,6,3	10	0.4	3,3,3	8	0.3	3,3,3	8	0.4	3,3,3	8	0.3
nc11n.v	7,11,7	13	0.9	8,10,3	14	0.8	10,13,3	17	0.7	8,9,3	13	0.7
sum1	2,15,10	17	2.3	3,16,7	25	1.5	4,17,3	21	3.3	3,13,3	17	1.1
sum1.v	**Timeout**			**7,45,18**	**65**	**3.6**	**5,93,75**	**169**	**18.7**	**6,20,5**	**26**	**3.2**
sum3	1,3,1	4	0.1	1,4,1	6	0.2	1,4,1	6	0.3	1,4,1	6	0.3
sum3.v	5,8,6	8	0.2	5,8,0	9	0.4	5,8,0	9	0.4	5,8,0	9	0.4
sum4	1,23,31	44	3.5	1,10,48	59	3.2	1,9,44	54	3.1	1,7,38	46	2.5
sum4n	**6,29,21**	**34**	**11.6**	**5,14,6**	**21**	**1.2**	**5,14,5**	**20**	**1.1**	**5,13,3**	**17**	**0.9**
sum4n.v	**12,33,24**	**38**	**31.8**	**10,60,24**	**87**	**5.3**	**6,27,12**	**40**	**2.3**	**8,40,10**	**51**	**2.8**
tacas	7,8,5	14	1.7	11,9,18	36	1.6	10,7,4	19	0.9	10,7,4	20	0.9
tacas.v	10,11,9	17	2.4	40,20,28	78	5.1	18,14,5	28	1.4	16,16,5	29	3.8
trex01	2,3,0	3	0.0	2,3,0	5	0.2	2,3,0	5	0.2	2,3,0	5	0.2
trex01.v	2,3,0	3	0.0	3,3,0	4	0.3	3,3,0	4	0.3	3,3,0	4	0.2
trex3	6,19,6	19	2.7	4,5,2	10	0.5	7,12,4	20	1.2	4,8,3	12	0.7
trex3.v	**12,25,12**	**25**	**10.8**	**8,10,4**	**18**	**1.0**	**12,8,2**	**19**	**3.5**	**10,8,2**	**16**	**0.9**
vsend	1,1,0	2	0.0	1,1,0	2	0.1	1,1,0	2	0.1	1,1,0	2	0.1
w1	1,3,3	5	0.0	2,2,2	6	0.2	2,1,1	4	0.4	2,2,1	5	0.2
w2	2,4,1	4	0.0	1,2,1	4	0.2	2,2,1	5	0.3	1,2,1	4	0.2
Aggregate	**Timeouts = 5 Total time = 243s + TO**			**Timeouts = 0 Total time = 93s**			**Timeouts = 0 Total time = 112s**			**Timeouts = 0 Total time = 74s**		

Table 1: Results comparing different learners for synthesizing invariants. P, N, I are the number of positive, negative examples and implications in the final sample; *#R* is the number of rounds and T is the time in seconds (Timeout was *60 s*).

5.2 Invariant Generation

We evaluate various configurations of the decision-tree based learner in the context of invariant synthesis and compare them to the constraint based learning algorithm proposed in [Garg et al. 2014]. The experimental results of our experiments are tabulated in Table 1. The first set of results correspond to the constraint-based learner from [Garg et al. 2014] (called ICE-CS); then we tabulate results for the simple decision-tree learner that completely ignores implications while deciding the *best* attribute to split the sample (called ICE-DT-simple); next we tabulate results for the decision tree learner that computes the implication gnostic information gain presented in Section 4.1 to decide on the attribute to split (called ICE-DT-entropy); finally we tabulate results for the decision-tree learner that penalizes splits that cut implications (called ICE-DT-penalty). For each learner we provide details about the composition, in terms of the number of positive and negative examples and implications, of the final sample that was required to learn an adequate invariant, the number of rounds to converge and the final time in seconds that was taken to learn an adequate invariant.

We evaluate the learners on several programs taken from previous literature [Garg et al. 2014; Gupta and Rybalchenko 2009]. To check if the decision tree learners we propose in this paper are robust and do not overfit a small set of programs, we generated program variations for twenty program by adding *three* extra variables that are havoc-ed inside the loop. These programs have the suffix ".v". From the table, we make the following key observations:

- The most important observation is that *all the decision tree learners converge and successfully find an adequate invariant for all programs*. We were surprised at this convergence; we expected heuristic machine learning to not necessarily converge; however, the outer ICE learning guidance from the teacher seems to make these learning algorithms fairly robust.
- The constraint solver-based learner ICE-CS times out on five programs. There are several programs where ICE-CS converges, but takes much longer than the decision-tree learners (e.g., for programs *cggmp05, ex23v, matrix2*).
- *The learner ICE-DT-penalty performs the best when compared to the other three learners*. It is easy to observe this from Figure 4 where we plot the run time of ICE-DT-penalty for all programs in Table 1 and compare them to the time taken by ICE-CS, ICE-DT-simple and ICE-DT-entropy. Moreover, we observe ICE-DT-penalty takes less than $10\,s$ to prove every program.
- *Implication counterexamples are required for progress*. Our teacher returns an implication counterexample only it cannot return positive or negative counterexamples. The final sample for most programs (see Table 1) contains implication counterexamples, indicating that we needed true ICE learning to learn invariants in these programs.

From Table 1 we further observe that decision tree learners usually perform very well even for programs in which we added a few extra variables (those with suffix ".v"). On the other hand, ICE-CS times outs on several programs (*cegar2, sum1, matrix2n*) when these extra variables were added. This is on expected lines—the search space for a constraint-based learner increases as one increases the number of variables and therefore it might start taking more time. On the other hand, machine learning algorithms are typically good at filtering out irrelevant information, and is witnessed by our decision tree learners as well.

We further observe from the table that ICE-DT-entropy is slower than ICE-DT-simple, even though it takes smaller number of rounds to converge on an adequate invariant averaged over all programs. This is so because computing the gain in ICE-DT-entropy involves slightly more computation as compared to ICE-DT-default and the slowness of this learner (that has not been optimized much) offsets the gain it might have achieved by converging in fewer rounds.

Comparison With Stochastic Search Recently, Sharma and Aiken [Sharma and Aiken 2014] have built an ICE learner based on randomized search for synthesizing program invariants. From our experience with this tool, we found it currently not robust enough for comparison. The tool is randomized, and the time to learn an invariant for some programs like *cegar2* and *fig1* spans a very large range from less than a second to more than a minute. Moreover, the technique constrains the random search towards only those numerical invariants that have constant thresholds that belong to a small bag of values, usually of size three to five, and that are mined from the program code. In comparison, the decision tree learners learn invariants that are arbitrary Boolean combinations of atomic numerical predicates where these predicates could have arbitrary constant thresholds in them.

6. Conclusions

We have presented a promising machine learning technique of decision trees from positive, negative, and implication counter-examples that can be used to efficiently synthesize invariants expressed as Boolean combinations of Boolean predicates and numerical predicates. We believe that our work presents opens up the gateway for adapting more machine learning algorithms for various other learning domains for invariant generation, by adapting them to ICE.

The impressive performance of our learner in weeding out the irrelevant attributes and quickly choosing the ones that matter suggest that machine learning techniques may even be useful in the simpler Houdini-based invariant synthesis of purely *conjunctive* formulae (which is also an ICE learner). In applications such as verification of GPU programs [Betts et al. 2012], Houdini has been used to synthesize from a large set of attributes, and we believe the learner we have presented here, or other adaptations of machine learning algorithms for conjunctions (such as linear classifiers) to ICE can help

solve these problems faster. Our decision tree learning also works for arbitrary predicates, and hence we would like to use them to synthesize other kinds of invariants, such as data-structure invariants using separation logic. Finally, we would like to use machine learning to learn not just predicates but also general program expressions/functions, building effective solvers using the learning/CEGIS technique in program synthesis [Solar Lezama 2008; Alur et al. 2013], though this seems to require a more powerful learning model.

References

R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 98–109, 2005a. doi: 10.1145/1040305.1040314. URL http://doi.acm.org/10.1145/1040305.1040314.

R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 548–562. Springer, 2005b. ISBN 3-540-27231-3. doi: 10.1007/11513988_52. URL http://dx.doi.org/10.1007/11513988_52.

R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–17. IEEE, 2013. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679385.

D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987. doi: 10.1016/0890-5401(87)90052-6. URL http://dx.doi.org/10.1016/0890-5401(87)90052-6.

D. Angluin. Queries and concept learning. *Mach. Learn.*, 2 (4):319–342, Apr. 1988. ISSN 0885-6125. doi: 10.1023/A:1022821128753. URL http://dx.doi.org/10.1023/A:1022821128753.

T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 203–213, 2001. doi: 10.1145/378795.378846. URL http://doi.acm.org/10.1145/378795.378846.

M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005. ISBN 3-540-36749-7. doi: 10.1007/11804192_17. URL http://dx.doi.org/10.1007/11804192_17.

A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. Gpuverify: A verifier for gpu kernels. *SIGPLAN Not.*, 47(10): 113–132, Oct. 2012. ISSN 0362-1340. doi: 10.1145/2398857.2384625. URL http://doi.acm.org/10.1145/2398857.2384625.

A. Biere and R. Bloem, editors. *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, 2014. Springer. ISBN 978-3-319-08866-2. doi: 10.1007/978-3-319-08867-9. URL http://dx.doi.org/10.1007/978-3-319-08867-9.

A. R. Bradley. Sat-based model checking without unrolling. In R. Jhala and D. A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011. ISBN 978-3-642-18274-7. doi: 10.1007/978-3-642-18275-4_7. URL http://dx.doi.org/10.1007/978-3-642-18275-4_7.

G. Brat, N. Rungta, and A. Venet, editors. *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 of *Lecture Notes in Computer Science*, 2013. Springer. ISBN 978-3-642-38087-7. doi: 10.1007/978-3-642-38088-4. URL http://dx.doi.org/10.1007/978-3-642-38088-4.

J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In H. Garavel and J. Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003. ISBN 3-540-00898-5. doi: 10.1007/3-540-36577-X_24. URL http://dx.doi.org/10.1007/3-540-36577-X_24.

M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In Jr. and Somenzi [2003], pages 420–432. ISBN 3-540-40524-0. doi: 10.1007/978-3-540-45069-6_39. URL http://dx.doi.org/10.1007/978-3-540-45069-6_39.

C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. doi: 10.1007/BF00994018. URL http://dx.doi.org/10.1007/BF00994018.

P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi: 10.1145/512950.512973. URL http://doi.acm.org/10.1145/512950.512973.

P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In A. V. Aho, S. N. Zilles, and T. G. Szymanski, editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978. doi: 10.1145/512760.512770. URL http://doi.acm.org/10.1145/512760.512770.

M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In C. Ghezzi, M. Jazayeri, and A. L. Wolf, editors, *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, pages 449–458. ACM, 2000. ISBN 1-58113-206-9. doi: 10.1145/337180.337240. URL http://doi.acm.org/10.1145/337180.337240.

G. Filé and F. Ranzato. The powerset operator on abstract interpretations. *Theor. Comput. Sci.*, 222(1-2):77–111, 1999. doi: 10.1016/S0304-3975(98)00007-3. URL http://dx.

doi.org/10.1016/S0304-3975(98)00007-3.

C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001. ISBN 3-540-41791-5. doi: 10.1007/3-540-45251-6_29. URL http://dx.doi.org/10.1007/3-540-45251-6_29.

G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Abstract interpretation over non-lattice abstract domains. In Logozzo and Fähndrich [2013], pages 6–24. ISBN 978-3-642-38855-2. doi: 10.1007/978-3-642-38856-9_3. URL http://dx.doi.org/10.1007/978-3-642-38856-9_3.

P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning universally quantified invariants of linear data structures. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 813–829, 2013. doi: 10.1007/978-3-642-39799-8_57. URL http://dx.doi.org/10.1007/978-3-642-39799-8_57.

P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In Biere and Bloem [2014], pages 69–87. ISBN 978-3-319-08866-2. doi: 10.1007/978-3-319-08867-9_5. URL http://dx.doi.org/10.1007/978-3-319-08867-9_5.

P. Garoche, T. Kahsai, and C. Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In Brat et al. [2013], pages 139–154. ISBN 978-3-642-38087-7. doi: 10.1007/978-3-642-38088-4_10. URL http://dx.doi.org/10.1007/978-3-642-38088-4_10.

S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In R. Gupta and S. P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 281–292. ACM, 2008. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375616. URL http://doi.acm.org/10.1145/1375581.1375616.

A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 634–640, 2009. doi: 10.1007/978-3-642-02658-4_48. URL http://dx.doi.org/10.1007/978-3-642-02658-4_48.

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 58–70, 2002. doi: 10.1145/503272.503279. URL http://doi.acm.org/10.1145/503272.503279.

R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006. ISBN 3-540-33056-9. doi: 10.1007/11691372_33. URL http://dx.doi.org/10.1007/11691372_33.

W. A. H. Jr. and F. Somenzi, editors. *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-40524-0.

M. Kawaguchi, P. M. Rondon, and R. Jhala. Type-based data structure verification. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 304–315, 2009. doi: 10.1145/1542476.1542510. URL http://doi.acm.org/10.1145/1542476.1542510.

M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11193-4.

N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1987. doi: 10.1007/BF00116827. URL http://dx.doi.org/10.1007/BF00116827.

F. Logozzo and M. Fähndrich, editors. *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, 2013. Springer. ISBN 978-3-642-38855-2. doi: 10.1007/978-3-642-38856-9. URL http://dx.doi.org/10.1007/978-3-642-38856-9.

K. L. McMillan. Interpolation and sat-based model checking. In Jr. and Somenzi [2003], pages 1–13. ISBN 3-540-40524-0. doi: 10.1007/978-3-540-45069-6_1. URL http://dx.doi.org/10.1007/978-3-540-45069-6_1.

A. Miné. The octagon abstract domain. In *WCRE*, page 310, 2001. URL http://computer.org/proceedings/wcre/1303/13030310abs.htm.

T. M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997. ISBN 978-0-07-042807-2.

D. Neider. *Applications of Automata Learning in Versification and Synthesis*. PhD thesis, RWTH Aachen University, April 2014.

D. Neider and N. Jansen. Regular model checking using solver technologies and automata learning. In Brat et al. [2013], pages 16–31. ISBN 978-3-642-38087-7. doi: 10.1007/978-3-642-38088-4_2. URL http://dx.doi.org/10.1007/978-3-642-38088-4_2.

J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. doi: 10.1023/A:1022643204877. URL http://dx.doi.org/10.1023/A:1022643204877.

J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. ISBN 1-55860-238-0.

F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65 (6):386–408, 1958.

S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In K. Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2006. ISBN 3-540-37756-5. doi: 10.1007/11823230_2. URL http://dx.doi.org/10.1007/11823230_2.

C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, 1948. URL http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf.

R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In Biere and Bloem [2014], pages 88–105. ISBN 978-3-319-08866-2. doi: 10.1007/978-3-319-08867-9_6. URL http://dx.doi.org/10.1007/978-3-319-08867-9_6.

R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 2012. ISBN 978-3-642-31423-0. doi: 10.1007/978-3-642-31424-7_11. URL http://dx.doi.org/10.1007/978-3-642-31424-7_11.

R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 574–592, 2013a. doi: 10.1007/978-3-642-37036-6_31. URL http://dx.doi.org/10.1007/978-3-642-37036-6_31.

R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In Logozzo and Fähndrich [2013], pages 388–411. ISBN 978-3-642-38855-2. doi: 10.1007/978-3-642-38856-9_21. URL http://dx.doi.org/10.1007/978-3-642-38856-9_21.

R. Sharma, A. V. Nori, and A. Aiken. Bias-variance tradeoffs in program analysis. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 127–138, 2014. doi: 10.1145/2535838.2535853. URL http://doi.acm.org/10.1145/2535838.2535853.

A. Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-177.html.

A. Thakur, A. Lal, J. Lim, and T. Reps. Posthat and all that: Attaining most-precise inductive invariants. Technical Report TR1790, University of Wisconsin, Madison, WI, Apr 2013.

A. Vardhan and M. Viswanathan. Learning to verify branching time properties. *Formal Methods in System Design*, 31(1):35–61, 2007. doi: 10.1007/s10703-006-0026-x. URL http://dx.doi.org/10.1007/s10703-006-0026-x.