LEARNING INDUCTIVE INVARIANTS USING WINNOW ALGORITHM

BY

ANJANA SURESH KUMAR

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Adviser:

Professor Madhusudan Parthasarathy

ABSTRACT

Invariant synthesis is crucial for program verification and is a challenging task. We present a new concrete learning algorithm, Winnow-ICE, to synthesize inductive invariants for proving that a program is correct by validating its assertions. Winnow is an online learning algorithm that can be used to learn Boolean formulae from positive, negative and implication counterexamples.

We implemented the Winnow algorithm as a plug-in for the Horn-ICE framework which is built on the Boogie program verifier. We compare our learning algorithm against Houdini and Sorcar by evaluating the algorithm on a subset of two different classes of benchmarks. The first class of benchmark is obtained from the GPUVerify tool, and the second class of benchmark is derived from Neider et al. (2018).

On the GPUVerify benchmark suite, it is noted that the Winnow algorithm takes considerably fewer rounds as compared to Sorcar for similar total performance in time, whereas, as compared to the Houdini, the total time taken is notably large for a similar performance in total number of rounds.

On the Dryad benchmark suite, it is observed that the Winnow algorithm takes fewer rounds as compared to Houdini and more rounds as compared to Sorcar; however, the Winnow algorithm is slower in comparison to both Sorcar and Houdini.

To my parents and family, for their unconditional love and support.

ACKNOWLEDGMENTS

I wish to express my deepest gratitude to my adviser, Professor Madhusudhan Parthasarathy, for his invaluable guidance and encouragement. He has always been supportive of my efforts and compassionate with me. His wisdom and work ethic will continue to inspire me in both my professional and personal life.

My gratitude is also extended to my previous adviser, Professor Shobha Vasudevan, who taught me many important lessons, as well as all of my colleagues and friends in the Department of Electrical and Computer Engineering and the Department of Computer Science, for always motivating and inspiring me.

Special thanks to my friend Vibhor Dodeja for always believing in the best of me.

Finally, I would like to thank my parents and my family for their unconditional love and support. Thank you everyone.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

The typical flow of program verification begins by expressing the program in terms of verification conditions which are mathematical logic formulas inferred from the program's source code. Verification conditions are based on the theory of Hoare triples [1] and involve unknown predicates that correspond to unknown inductive invariants [2]. An invariant is understood as a set of program states that includes all the initial states of the program (initialization), excludes all the set of states that fail to meet the program specification (on termination), and is inductive; that is, if a program starts its execution from a state that belongs to the invariant, it results in a state that is also the invariant (maintenance). The inductive invariant synthesis is fundamental to prove the correctness of imperative programs, and generating inductive invariants is the most difficult aspect of program verification [3, 4, 5].

With sufficient invariants, the process of program verification can be reduced to determining the logical validity of verification conditions [2, 6]. The process of checking the validity of verification conditions has been automated over the past several years [7, 8, 9]. Constrained Horn clauses (CHCs) provide a convenient and elegant way to formally represent many encoding styles of verification conditions [10, 11, 12]. Given a set of CHCs, finding an invariant that satisfies each constraint will prove the program correct.

However, finding an invariant was traditionally the responsibility of a human, either the person performing the verification, or the programmer writing the program, but over the years, different ways to generate invariants automatically have been discovered [6, 3, 13, 14, 15, 16, 17, 18, 19, 20, 21]. These approaches can be broadly classified into white-box techniques and black-box techniques.

In the white-box techniques such as abstract interpretation [3], predicate abstraction [13], and interpolation [20], the invariant synthesizer has a pro-

found insight about the program and the property being proven. These techniques evaluate the program symbolically or use proofs of unreachability of error states to synthesize an inductive invariant [22].

On the other hand, in the black-box techniques [2, 6, 15, 22, 21], the invariant synthesizer is largely agnostic to the program and works only with a partial view of the requirements of the invariant. Black-box techniques are data-driven and primarily have two components, a teacher (program verifier) and a learner (invariant synthesizer), which interact with each other in rounds. In each round, the learner proposes an invariant hypothesis and the teacher evaluates the CHCs for the proposed invariant and returns a counterexample or proves that the program is correct. Recently, machine learning [23] techniques have been explored and applied to the field of invariant generation.

In this work, we focus on a learning sum of product (SOP) form of Boolean functions as invariants using a traditional machine learning algorithm Winnow [24]. Given a fixed set of predicates and CHCs, the goal is to learn an invariant that satisfies the CHCs. Winnow is an online learning algorithm that is adapted in the ICE learning setting [6] and implemented as a plug-in for the Horn-ICE framework [22].

The details of the Winnow-ICE framework are described in depth in the following sections. The performance of Winnow is evaluated on two benchmarks: GPUVerify [25] and Neider et al.'s tool (Dryad) [26]. Winnow is compared against existing tools Sorcar [21] and Houdini [4].

The remainder of this thesis is organized as follows. Chapter 2 provides the required background needed for this thesis. Chapter 3 describes the related work in this domain. Chapter 4 covers the specifics of implementation. Chapter 5 evaluates the performance of the implementation. Chapter 6 concludes the thesis with future directions.

CHAPTER 2

BACKGROUND

In this chapter, Section 2.1 introduces the different notations and definitions used in this thesis. Section 2.2 briefly explains black-box learning techniques, their advantages and challenges. Sections 2.4 and 2.5 provide an overview of algorithms implemented in this thesis.

2.1 Notations and Definitions

A *predicate* on X is a Boolean-valued function P: X $\rightarrow$ {True, False}. Predicate P(x) is True if x is in the set and False, if x is not in the set.

A *loop invariant (LI)* is a property of the loop that is true for every execution of the loop. Every loop can be expressed as a loop invariant. Loop invariants always hold immediately before the loop and after each iteration of the loop.

A *constrained Horn clause (CHC)* is a robust verification condition formalism that encodes a program. Table 2.1 demonstrates CHCs for the given C program.

- Constraint (1) proves the trivial case when the loop body is never executed.
- Constraint (2) guarantees that LI is True when the loop head is initially reached.
- Constraint (3) ensures that LI is inductive with the loop body.
- Constraint(4) says that the LI does not violate the post condition.

The goal is to find a formula for LI such that all the constraints from (1) through (4) are valid. The LI for the above program is $(y \geq 1) \bigwedge (y \geq x)$.

Table 2.1: Example of a CHC

C Code	CHCs

`main()`
`{`
`  int x, y;`
`  x = 0;`
`  y = 1;`
`  while(*)`
`  {`
`  x = x + 1;`
`  y = x + y;`
`  }`
`  assert (y >= x);`
`}`
`(* indicates unknown number of loop iterations)`
	1. $(x = 0) \bigwedge (y = 1) \implies (y \geq x)$
	2. $(x = 0) \bigwedge (y = 1) \implies LI(x, y)$
	3. $LI(x, y) \bigwedge (x' = x + 1) \bigwedge (y' = x + y) \implies LI(x', y')$
	4. $LI(x, y) \implies (y \geq x)$

2.2 Data-driven Learning

To verify a program using Hoare logic, a program is annotated with assertions and each assertion corresponds to a Hoare triple. To check the validity of Hoare triples, predicate transformer semantics as proposed by Dijkstra in [27] is used. Predicate transformers such as weakest precondition and strongest postcondition provide complete strategies to verify a Hoare triple by transforming the problem equivalently to that of proving a first-order logic formula [28]. Formulas obtained from predicate transformers are called *verification conditions (VC)*. CHCs are a form of VC. CHCs involve unknown inductive invariants. Machine learning algorithms can be used for invariant synthesis and these are black-box techniques of invariant generation.

2.2.1 Advantages

When the programs get large and complex with convoluted data structures, complex code semantics, pointers, heaps, non-linear arithmetic operations, etc., generating a *simple* inductive invariant that is sufficient to prove a prop-

erty of the program is hard. This may be because even user-written annotations are complex. Black-box techniques ignore this complexity by learning only over a restricted observation of the state of the program. Additionally, black-box techniques provide a way to apply machine learning techniques to verification problems.

2.2.2 Disadvantages

One of the challenges of this approach is the choice of atomic predicates (*features* in machine learning). As the program complexity increases, there arises a need for domains of higher dimensionality to express the feature vectors (richer domains like *polyhedra* domain). Machine learning algorithms tend to propose a generalized concept and avoid overfitting, but for verification tasks we need the algorithm to learn a concept that is fully consistent with given samples even if it is at the expense of overfitting. In other words, we need a perfect classifier for verification tasks. Another limitation of this approach is that when the samples are not easily separable, the machine learning model may underfit the data and may yield incorrect and complex invariants.

2.3 Framework

There are primarily two components in a black-box technique. The first component is a teacher which is the verification engine that returns appropriately labeled counterexamples in every round, and the second is the learner that learns using the examples and proposes an invariant hypothesis. This hypothesis can be any arbitrary combination of Boolean functions or a linear threshold function (LTF) or a decision tree depending on the machine learning algorithm used. We develop a learner using the Winnow algorithm in this thesis.

2.3.1 Teacher

Given a program, its VC (CHCs) can be automatically generated by tools such as Boogie [9] and Seahorn [12]. Given the VC, their validity is checked by constraint solvers such as Z3 [7], SMT solvers, etc.

There are primarily three kinds of counterexamples that can be generated. They are positive, negative and implication counterexamples.

- *Positive* counterexamples are generated when the given state s satisfies the precondition α but does not satisfy the loop invariant ϕ, that is, $s \models \alpha$ and $s \not\models \phi$.
  Equations are of the form $\alpha \implies \phi$.

- *Negative* counterexamples are generated when the given state s satisfies the loop invariant ϕ but does not satisfy the post condition β, that is, $s \models \phi$ and $s \not\models \beta$.
  Equations are of the form $\phi \implies \beta$.

- *Horn/Implication* counterexamples are generated when the invariant is not inductive and is of the form $(\{s_1, s_2, ..s_n\}, s)$, where $s_i \models \phi$ for each $i \in \{1, 2, .., n\}$ but $s \not\models \phi$. These equations encode the inductive property of the loop invariant.

Traditionally, the teacher generated only positive and negative counterexamples. Then came along ICE-learning [6], which included implication counterexamples, and later came Horn-ICE learning [22] for implication counterexamples arising from Horn implication constraints.

The teacher takes in the invariant hypothesis from the learner, CHCs, and returns Horn-ICE samples which include positive (+), negative (-), and Implication (?) counterexamples as shown in Figure 2.1.

2.3.2 Learner

In this thesis, we develop the learner using the Winnow algorithm [24] and Houdini algorithm (similar to classical elimination algorithm) [4] and we call it Winnow-ICE. The details of the implementation are mentioned in Chapter 4. Section 2.4 provides a brief outline of the Winnow algorithm. The learner takes Horn-ICE samples that include examples, counterexamples, and implication examples and proposes an invariant as a hypothesis as shown in Figure 2.1.

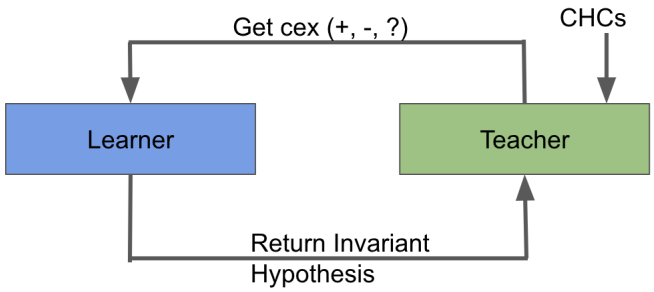

Figure 2.1: Teacher-Learner Framework

2.4 Winnow Algorithm

The Winnow algorithm is an online machine learning algorithm that learns a linear classifier from labeled examples [24]. An online learning algorithm has no separate set of training examples and the learner learns from each example given. The learner is told if the prediction was accurate after making it, and then uses this knowledge to strengthen its hypothesis. As long as it receives examples, the learner continues to learn and the algorithm returns a linear threshold function (LTF). These kinds of algorithms are useful where we need to compute the hypothesis incrementally.

The goal of the Winnow algorithm is to learn a concept described by a Boolean function by making fewer mistakes. The Winnow algorithm is observed to perform well when there is a large number of irrelevant attributes and the correct-response function depends on a small set of attributes. This behavior is attributable to Winnow's use of multiplicative weight update. Since we want to minimize the number of rounds (interactions between teacher and learner) and minimize the time taken to prove, Winnow is our choice of algorithm in this work.

2.4.1 Algorithm Description

The input to the Winnow learner is a set of feature vectors that are a list of Boolean attributes, and the label is an output of some Boolean function of the attributes. The Winnow algorithm is similar to the Perceptron algorithm [29] except that it has a multiplicative update scheme instead of an additive weight update scheme.

The algorithm takes an input vector $X = \{0,1\}^N$ and maintains non-negative real-valued weights W. The weights W are of size (N, 1). There are two hyperparameters: real numbers θ and α. θ is the threshold and α is the amount by which the weights are changed.

On receiving an input vector $(x_1, ..., x_N)$, the learner responds as follows:

- Prediction is 1.0 if $\sum_{i=1}^{N} w_i * x_i >= \theta$
- Prediction is 0.0 if $\sum_{i=1}^{N} w_i * x_i < \theta$

The weight update scheme of the algorithm is described in Algorithm 1. The weights are changed only for the non-zero data (x_i) and only if the learner makes a mistake. The amount by which the weights are changed depends on a fixed parameter α and it can be set to any value greater than 1. Each mistake is either a promotion step or a demotion step.

2.4.2 Mistake Bounds

Littlestone [24] gives the following theorem. We give the proof of the theorem from [24] here since it is directly related to our work.

Theorem 1. *Suppose that the target function is a k-literal monotone disjunction given by* $f(x_1, ..., x_n) = x_{i1} \vee ... \vee x_{ik}$. *If the Winnow algorithm is run with* $\alpha > 1$ *and* $\theta > 1/\alpha$, *then for any sequence of instances the total number of mistakes will be bounded by* $\alpha * k(log_\alpha \theta + 1) + \frac{N}{\theta}$.

To prove this theorem, Littlestone uses three lemmas (enumerated here as Lemma 1, Lemma 2, Lemma 3) that are related to finding bounds on the number of promotion and demotion steps that occur.

Let u be the number of promotion steps that have occurred by the end of some sequence of trials and let v be the number of demotion steps that have occurred by the end of the same sequence of trials.

Lemma 1. $v < N/\theta + (\alpha - 1)u$.

Proof. Initially, $\sum_{i=1}^{N} w_i = N$ because w_i is initialized to 1.

- Each promotion steps increases this sum by at most $(\alpha - 1)\theta$, since when a promotion step occurs we have $\sum_{i|x_i=1} w_i \leq \theta$.

- Each elimination step decreases $\sum_{i=1}^{N} w_i$ by at least θ.

Since the sum of weights is never negative, we have

$$0 \leq \sum_{i=1}^{N} w_i \leq N + u(\alpha - 1)\theta - v\theta$$

□

Lemma 2. *For all i, $w_i < \alpha\theta$.*

Proof. Since $\theta > 1/\alpha$, the weights are initially less than or equal to $\alpha\theta$. For any j, the value of w_j is only increased for $x_j = 1$ and $\sum_{i=1}^{N} w_i x_i \leq \theta$.

These conditions can only occur together if $w_i < 0$ immediately prior to the promotion. Thus $w_i < \alpha\theta$ after the promotion. □

Lemma 3. *After u promotion steps and an arbitrary number of elimination steps, there exists some i for which $log_\alpha w_i > u/k$.*

Proof. Let $R = i_1, ..., i_k$.

$f(x_1, ..., x_n) = 0$ if and only if $x_i = 0$ for all $i \in R$. Elimination steps occur only when $f(xi, ..., xn) = 0$, and promotion steps occur only when $f(x_1, ..., x_n) = 1$.

Thus $\prod_{i \in R} w_i$ is unchanged by elimination steps and is increased by a factor of at least α by each promotion step. Initially $\prod_{i \in R} w_i = 1$.

Thus after u promotion steps $\prod_{i \in R} w_i \geq \alpha^u$ giving $\sum_{i \in R} log_\alpha w_i \geq u$. Since $|R| = k$, for some $i \in R$ we have $log_\alpha w_i \geq u/k$, as desired. □

We demonstrate the proof of Theorem 1 using Lemmas 1, 2, and 3 as follows.

Proof. The total number of mistakes made during a run of the algorithm is equal to the number of promotion steps, u, plus the number of elimination steps, v.

Combining Lemmas 2 and 3 we get

$$u/k \leq log_\alpha w_i \leq log_\alpha \theta + 1$$
$$u \leq k(log_\alpha \theta + 1)$$

Lemma 1 gives

$$v \leq \frac{N}{\theta} + (\alpha - 1)k(log_{\alpha}\theta + 1)$$

Adding the bounds on u and v leads to the desired bound on the total number of mistakes. □

It is observed that a good bound is achieved when the weights are initialized to 1.0, α is 2, and threshold θ is initialized to $N/2$. With these parameters, for a k-literal monotone disjunction of the form, from the Theorem 1, in the worst case, the Winnow algorithm makes O($k$log(N)) mistakes.

The Perceptron algorithm is noted to make $(N + k - 1)/2$ mistakes in learning a k-literal monotone disjunction. The number of mistakes is linear in N whereas, for the Winnow algorithm, the number of mistakes is logarithmic in N.

Algorithm 1: Winnow Algorithm Weight Update

Input: A single data point x, true label y_{true}, prediction y_{pred}
Output: $wgts$

```
for i ← 0 to len(x) do
    Demotion Step;
    if x[i] and y_true == 0 and y_pred == 1 then
        wgts[i] = wgts[i]/α;
    end
    Promotion Step;
    if x[i] and y_true == 1 and y_pred == 0 then
        wgts[i] = wgts[i] * α;
    end
end
```

2.5 Houdini Algorithm

Houdini is based on the elimination algorithm [23, 30] and learns a conjunctive invariant over a given set of predicates. The pseudo-code of the algorithm is depicted in Algorithm 2. There are N Boolean predicates P and each predicate is denoted by P_i for $i \in \{1, 2, ..N\}$. Initially, the conjunctive concept X is proposed over all predicates P. With each positive counterexample s_+,

all predicates that violate this example are removed from the conjunctive concept. Consequently, we get the largest conjunctive formula that is consistent with S_+. For each failed inductive constraint with a counterexample of the form (s, s'), s' is always added as a positive example. The elimination algorithm does not learn from any negative examples. If a negative example s_- does satisfy X, that means that there is no conjunctive invariant.

Remark. *Houdini algorithm is guaranteed to converge to a conjunctive invariant, if one exists, in $O(N)$ rounds.*

In Chapter 3, we review the previous works in this domain.

Algorithm 2: Houdini Algorithm

Input: Positive counterexamples S_+, Horn ICE samples S_H,
Negative counterexamples S_-, predicates P
Output: Conjunctive concept X

```
1  X = ⋀_{i=0}^{N} P_i;
2  foreach s_+ in S_+ do
3  |  X : X \ {P_i | s_+[P_i] = False}
4  end
5  repeat
6  |  foreach Horn-ICE sample (s, s') in S_H do
7  |  |  if s is satisfied by X then
8  |  |  |  Add s' as positive
9  |  |  |  Take every P_i False in (s, s'), and remove it from X
10 |  |  end
11 |  end
12 until X converges;
   /* Check negative examples violate Conjunctive concept */
13 foreach s_- in S_- do
14 |  if s_- is satisfied by X then
15 |  |  Halt; No conjunctive invariant
16 |  end
17 end
```

CHAPTER 3

RELATED WORK

In this chapter, we discuss the evolution of several black-box techniques for a conjunctive invariant generation. A conjunctive invariant is a conjunctive formula over a set of predicates. An early example is the *elimination* algorithm [30] that proposes a conjectured invariant over a set of predicates and iteratively eliminates every predicate that is False for a positive counterexample until a fixed point is reached. This algorithm semantically proposes the smallest conjunctive formula over the predicates that includes all the positive counterexamples.

Another early prominent example is Daikon [31], which is based on an elimination algorithm and generates *likely* invariants of a program using dynamic analysis. The tool observes the values that the program computes, and tracks properties that were True during the observed executions to propose an invariant that is statistically justified by the observations. Daikon proposes a conjunctive invariant; however, it is not guaranteed to be the True inductive invariant of the program.

Houdini [4] circumvents the drawback of Daikon by generating conjunctive invariants over a set of predicates and uses a constraint-solver as the teacher to propose counterexamples and refine its conjectured invariant. This happens iteratively until the concept learned is inductive and adequate. Inferred invariant depends on the initial set of predicates chosen and does not take into account negative counterexamples or assertions in the program. If the number of predicates is too large, it takes a considerable amount of time to remove predicates, leading to an increase in the running time of the algorithm. Houdini also does not take into account the assertions in the program, and thus the tightest invariant can be quite complex (have many conjuncts) and hard to synthesize.

Recently, machine learning techniques have been applied for an invariant generation. Earlier, the goal was to find a linear classifier that separates

positive and negative counterexamples [23, 30]; however, it was found that this method was not robust since for implication counterexamples of the form (s, s'), either the teacher would lie and arbitrarily pick s as negative or s' as positive or the learner would make a guess. Either way, the teacher would cheat and there was no way of knowing the precise invariant. Then came ICE learning [6] where the teacher communicated the implication pair to the learner and the learner made the choice, based on considerations of generalization, simplicity etc., whether it would include both s and s' in its set (add them as positive) or leave s out (add s as negative sample).

Sorcar [21] is a Horn-ICE learning framework that learns conjunctive inductive invariants that are sufficient to prove the assertions in the program, making it a property-driven algorithm as opposed to Houdini. Additionally, Sorcar proposes a weaker invariant, unlike Houdini which proposes the *tightest* conjunctive invariant. Given a set of predicates P, Sorcar only considers a subset of relevant predicates $R \subseteq P$, enough to prove the assertions. The authors propose different variants of Sorcar depending on how relevant predicates are chosen.

Data-driven CHC solvers [2] use machine algorithms like support vector machines (SVMs) and decision trees to generate inductive invariants. In this paper, the authors develop a new data-driven algorithm, that does apply a linear classification algorithm iteratively on unseparated (misclassified) samples, thereby learning a family of linear classifiers that separate all samples correctly in the aggregate. The problem of overfitting is dealt with by constructing a layered machine learning toolchain, which allows generalization in the first layer by learning a combination of linear classifiers and uses a decision tree to pick the most attributes with high information gain.

Chapter 4 provides the necessary implementation details of this work.

CHAPTER 4

IMPLEMENTATION

In this chapter, we discuss the motivation for choosing the Winnow algorithm and describe the overall Winnow-ICE framework and its components. We describe the challenges for integrating Winnow in the Horn-ICE framework and the different methodologies used to implement the Winnow-ICE framework. We also present several heuristics to speed up the implementation. When we say *teacher*, we mean the Boogie program verifier and the *learner* is the Winnow algorithm and Houdini.

4.1 Motivation

Our primary motivation for adapting Winnow is its guaranteed mistake bound of $O(klogN)$ where the target function is a k-literal monotone disjunction and N is the total number of predicates. This bound is achieved assuming the threshold θ is $N/2$. The expectation is that Winnow will make fewer mistakes and will consequently reduce the number of rounds to the teacher. For black-box techniques, passive learning is not enough and online machine learning models like Winnow are the closest to iterative models. The data samples are constantly arriving and the learner proposes a new hypothesis each time. The Winnow algorithm essentially performs binary classification, over N-dimensional real vectors when we have N predicates; thus, it can handle higher dimensionality.

4.2 Winnow-ICE Framework

One of the major impediments to integrating Winnow in the current Horn-ICE setting is the incompatibility with the existing interface between the teacher and learner. At present, the learner in the Horn-ICE setting returns

a Boolean formula, whereas Winnow returns a linear threshold function. We resolve this in two ways. The first method is converting the linear threshold function into a Boolean formula, and the second method is extending the interface to support linear threshold function.

Another consideration is the choice of algorithm to label the unlabeled implication counterexamples. In this work, we choose the Houdini algorithm to extend the Winnow algorithm to implication counterexamples since the Houdini algorithm is known to take a polynomial number of rounds (related to the number of predicates) to converge to a conjunctive formula. Given an implication counterexample of the form (s, s'), either s can be labeled as negative or s' can be labeled as positive. Using Houdini, we compute the strongest conjunctive formula ϕ over predicates that are consistent with positive counterexamples and implication counterexamples. If s satisfies ϕ, we label s' as positive and if s does not satisfy ϕ, we label s as negative.

A point to note is that Winnow primarily learns a disjunctive concept over X for a function of the form $w^T * X >= \theta$. Therefore, to make Winnow learn conjunctive concepts over X, we need to modify the linear threshold function to the form $w^T * X' < \theta$. This would mean inverting the atomic predicates and inverting the label before training Winnow.

To speed up the overall implementation, we take advantage of another aspect of Houdini. As the number of predicates increases, we can drop the irrelevant predicates in the proposed loop invariant by setting their corresponding weights to zero. This dramatically reduces the size of the loop invariant, which results in faster convergence on large programs. Any predicate is considered irrelevant if it is not in the conjunctive formula proposed by Houdini. The intuition is that, if a predicate does not satisfy the positive counterexamples, it is not in the proposed invariant, so we can "knock" them off.

Like Sorcar, Winnow-ICE is a plug-in to the Horn-ICE framework. All the communication between teacher and learner happens through files. The invariant(s) are communicated to Boogie in JSON format. The Boolean formula (in form of a tree) is parsed and then converted to (multiple) Boogie formulas. The conversion to a Boogie formula happens implicitly by the JSON framework since JSON format makes it easy to serialize and deserialize trees.

4.3 Core Algorithm

The general working of the Winnow-ICE algorithm is described in Algorithm 3. The input to the algorithm is the set of N predicates P. In Line 1, the learner proposes a loop invariant hypothesis ϕ with default initialization. In Line 3, the teacher returns the Horn-ICE samples if the proposed loop invariant does not satisfy the CHCs. In Line 4, the Houdini algorithm is used to propose the tightest conjunct X that satisfies all positive counterexamples S_+. In Line 5, all implication counterexamples are labeled either positive or negative depending on whether they satisfy the tightest conjunct. In Line 6, all irrelevant predicates are eliminated from the loop invariant ϕ. Line 7 demonstrates the inversion of data samples and their labels to learn the conjunctive formula. In Line 8, the Winnow algorithm is trained on all positive and negative counterexamples until all samples are classified correctly. After Winnow converges, a loop invariant is proposed. The entire process from Line 3 to Line 8 repeats until the loop invariant satisfies all CHCs.

Algorithm 3: Winnow-ICE Algorithm

Input: Predicates P
Output: Invariant ϕ

```
   /* Get initial loop invariant φ                          */
1  φ = InitializeWinnow(P);
2  repeat
3  |  S+, SH, S− = Teacher(φ);
   |  /* Obtain the tightest conjunct                       */
4  |  X = Houdini(S+, P);
   |  /* Label the Implication counterexamples              */
5  |  S+, S− = Satisfies(X, SH);
   |  /* Knock-off predicates                               */
6  |  P = P \ {Pi | Pi ∉ X};
   |  /* Invert the data and label                         */
7  |  S+, S− = Invert(S+, S−);
8  |  TrainWinnow(S+, S−);
   |  /* Return φ                                          */
9  until Program verified;
```

In Subsections 4.3.1 and 4.3.2, the two different methods of implementation are discussed.

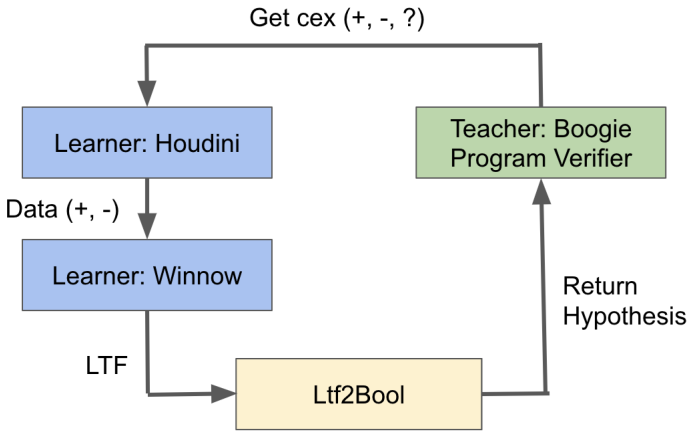

Figure 4.1: Method1: Winnow-ICE Framework

4.3.1 Method 1

In this method, we use a component called Ltf2Bool to assist in the conversion of linear threshold function to a Boolean formula. The overall Winnow-ICE framework with this additional component is shown in Figure 4.1. This component takes the linear threshold function given by Winnow as input and converts it into an equivalent Boolean expression and is necessary for increased ease of integration into the Horn-ICE framework.

The pseudo-code for this method is shown in Algorithm 4. The only change as compared to Algorithm 3 is the addition of Lines 2 and 10. Lines 2 and 10 call the Ltf2Bool function to convert the linear threshold function LTF to a Boolean formula ϕ. The pseudo-code of the Ltf2Bool algorithm is described in Algorithm 5. The algorithm takes the converged weights w, predicates P, and set of predicate indices J and a threshold θ as input and returns a Boolean formula ϕ. Initially, J is the set of all predicate indices and θ is the difference of the sum of weights and $N/2$, where N is the number of predicates. Ltf2Bool is a recursive function that chooses the predicate corresponding to the maximum weight in each iteration until either the threshold is less than zero or the sum of weights is less than the threshold.

Algorithm 4: Method1: Winnow-ICE Algorithm

Input: Predicates P
Output: Invariant ϕ
/* Get initial LTF */
1 LTF = InitializeWinnow(P);
/* Get initial loop invariant ϕ */
2 ϕ = Ltf2Bool(LTF);
3 **repeat**
4 S_+, S_H, S_- = Teacher(ϕ);
/* Obtain the tightest conjunct */
5 X = Houdini(S_+, P);
/* Label the Implication counterexamples */
6 S_+, S_- = Satisfies(X, S_H);
/* Knock-off predicates */
7 $P = P \setminus \{P_i \mid P_i \notin X\}$;
/* Invert the data and label */
8 S_+, S_- = Invert(S_+, S_-);
9 LTF = TrainWinnow(S_+, S_-);
/* Convert LTF to a Boolean formula */
10 ϕ = Ltf2Bool(LTF);
/* Return ϕ */
11 **until** *Program verified*;

4.3.2 Method 2

In this method, we modify the teacher (Boogie program verifier) to process the linear threshold function directly. The modified Winnow-ICE framework is shown in Figure 4.2. Winnow proposes an LTF as a loop invariant and the teacher returns Horn-ICE samples if the LTF does not satisfy the CHCs. The pseudo-code for this method is the same as the Winnow-ICE algorithm except the invariant ϕ is now the LTF.

4.4 Parameter Tuning

It is important to choose the right values of θ, initial weights, and α for an increase in the performance of the algorithm. We measure performance both in terms of the number of rounds taken to verify a program and the subsequent time for the same.

Littlestone [24] states that a good bound for Winnow is achieved when

Algorithm 5: Ltf2Bool Algorithm

Input: Converged Weights w, θ, J, predicates P
Output: Invariant ϕ

```
if θ ≤ 0 then
    return True
end
if Σ_{i=1}^{J} w_j > θ then
    weight_max = max(w);
    idx_max = index corresponding to weight_max;
    J.erase(idx_max);
    return ((P[idx_max] && Ltf2Bool(W, θ - weight_max, J, P)
                || Ltf2Bool(W, θ, J, P));
end
else
    return False
end
```

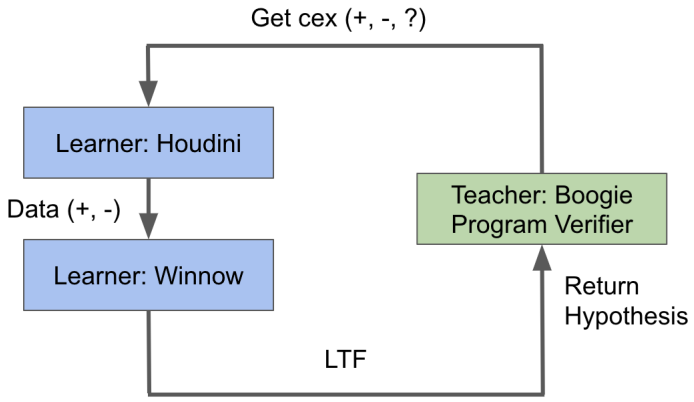

Figure 4.2: Method2: Winnow-ICE Framework

$\theta = N/2$, α is 2 and initial weights are set to 1. However, Kivinen et al. [32] argue that for $k < N/e$, where k is the number of literals in the target concept and N is the total number of predicates, the best values are α of 0.875, θ of 0.441 and initial weights $2N/5$. For $k \geq N/e$, α of 1, θ of 0.425 and initial weights 0.368 achieve good bounds.

4.4.1 Choosing Initial Weights

Following is a mathematical explanation for choosing initial weights given $\theta = N/2$ and α of 2. Let X be the vector of predicates and assume there

are N predicates, and let W be the vector of weights. For Winnow to learn conjunctions, we invert the atomic predicates and labels, thereby learning

$$W * (1 - X) < \theta$$
$$\sum_{i=1}^{N}(w_i) - W * X < \theta$$
$$W * X > \sum_{i=1}^{N}(w_i) - \theta$$

Say, initially, all weights are equal to value "k" and we set $\theta = N/2$.

$$k * \sum_{i=1}^{N}(x_i) > N * k - (N/2)$$
$$\sum_{i=1}^{N}(x_i) > N - (N/2k)$$

Say we have initial weights of $k = 1$, we then get

$$\sum_{i=1}^{N}(x_i) > N - (N/2)$$

$$\sum_{i=1}^{N}(x_i) > N/2 \tag{4.1}$$

Equation 4.1 is True when at least $N/2$ predicates are 1 (equivalently True). This can happen in $\binom{N}{N/2} + \binom{N}{(N/2)+1} + \binom{N}{(N/2)+2} ... + \binom{N}{N}$ ways. From this, we know that this is exponential in N. When Winnow proposes an LTF and that gets translated to a Boolean function, the number of terms in the SOP is exponential in N and thus choosing initial weights as 1 is not scalable.

Say we have initial weights of $k = N$, we then get

$$\sum_{i=1}^{N}(x_i) > N - (N/2) * (1/N)$$

$$\sum_{i=1}^{N}(x_i) > N - (1/2) \tag{4.2}$$

Equation 4.2 is True only when all the predicates are 1 (equivalently True). This can happen in only one way. The number of terms in the SOP after Ltf2Bool is a conjunction of all predicates, which is the same as that proposed by the Houdini algorithm standalone.

Say we have initial weights of $k = N/2$, we then get

$$\sum_{i=1}^{N}(x_i) > N - (N/2) * (2/N)$$

$$\sum_{i=1}^{N}(x_i) > N - 1 \tag{4.3}$$

Equation 4.3 is True only when all the predicates are 1 (equivalently True). This can happen in only one way. The number of terms in the SOP after Ltf2Bool is a conjunction of all predicates, which is the same as what we observe when $k = N$.

Remark. *If initial weights are between [N/2, N], we notice and theoretically expect the same behavior as Houdini.*

Say we have initial weights of $k = N/4$, we then get

$$\sum_{i=1}^{N}(x_i) > N - (N/2) * (4/N)$$

$$\sum_{i=1}^{N}(x_i) > N - 2 \tag{4.4}$$

Equation 4.4 is True when at least one predicate of N is 1 (equivalently True). This can happen in $\binom{N}{1} + \binom{N}{N} = N + 1$ ways. From this, we know that choosing initial weights as $N/4$ makes the number of terms in the SOP returned by Ltf2Bool linear in N. Thus, $N/4$ will be our initial weights.

Chapter 5 discusses the benchmarks and provides a thorough analysis of the results.

CHAPTER 5

EVALUATION

In this chapter, we give a brief overview of the benchmarks used. Following that are results corresponding to different implementation strategies.

5.1 Benchmarks

There are two benchmarks we target in this thesis. The first is obtained from the GPUVerify tool [25]. This tool is used for verifying that GPU kernels are race and divergence-free. The tool automatically preprocesses the programs written in CUDA and OpenCL into BOOGIE programs during compilation.

The second suite of benchmarks is from Neider et al.'s tool, VCCDyrad [26], which compiles C programs into BOOGIE programs.

We select programs from both these benchmarks and compare the performance of the Winnow-ICE algorithm to Houdini and Sorcar. When we use performance, we measure the number of rounds and the total time taken to verify the program. The number of rounds means the number of times the teacher (prover or verification engine) is called.

5.2 Results and Analysis

We evaluate Winnow on the two benchmarks described in the preceding section. We report the average numbers over the entire benchmark in Figures 5.1, 5.2, 5.3, and 5.4.

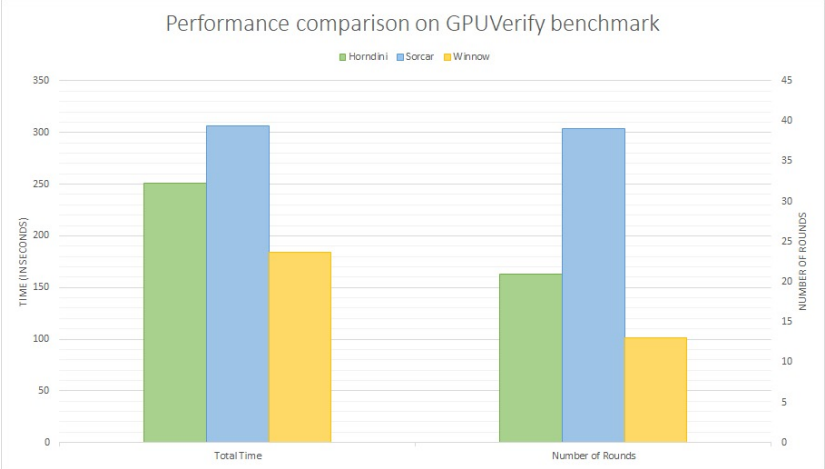

Figure 5.1: Method1: Evaluation of Winnow on GPUVerify Benchmarks

5.2.1 Method 1

Results of evaluation for Method 1 are summarized in Figures 5.1 and 5.2. Figure 5.1 shows the performance of Winnow as compared to Sorcar and Houdini on a subset of GPUVerify benchmarks. The files are chosen based on the number of rounds taken by the Sorcar algorithm. All the files where the Sorcar tool takes more than 20 rounds to verify the program are selected. From Figure 5.1 we note that Winnow takes significantly fewer rounds on programs as compared to both Sorcar and Houdini. Additionally, Winnow performs better in terms of the total time taken. However, Winnow is noted to perform better than Sorcar and Houdini only on this small subset of benchmarks.

Figure 5.2 shows the performance of Winnow as compared to Sorcar and Houdini on Dryad benchmarks. We see that Winnow's performance is worse on these files both in terms of the number of rounds and total time taken. The time taken by Winnow is the longest due to the fact that the Boolean formula generated from Ltf2Bool is large and complex. The time taken by Winnow is more than Houdini because Winnow internally uses Houdini and that is adding to the extra total time taken.

5.2.2 Method 2

In this method, we eliminated the dependency on the Ltf2Bool function and directly returned the LTF returned by Winnow to the teacher. This made a considerable difference in Winnow's performance since we could harness

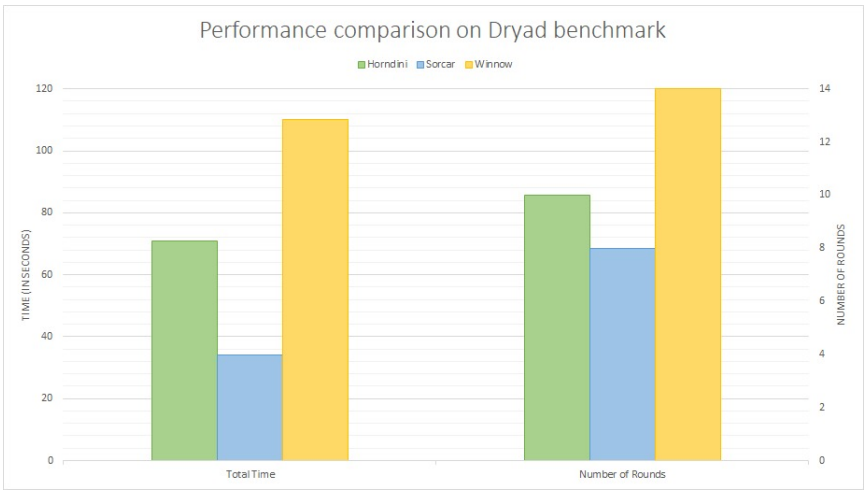

Figure 5.2: Method1: Evaluation of Winnow on Dryad Benchmarks

Winnow's potential on logarithmic mistake bounds.

On the GPUVerify benchmark suite, for the 222 programs verified by all three algorithms (Winnow, Houdini, Sorcar), the Winnow algorithm took 22% fewer rounds as compared to Sorcar and 2.7% more rounds as compared to Houdini. It was observed that Sorcar and the Winnow algorithm performed with similar times, whereas the Winnow algorithm took 21% more time as compared to Houdini. Overall, the Winnow algorithm takes considerably fewer rounds than Sorcar for similar total performance in time, whereas, as compared to Houdini, the total time taken is notably large. The summary of results on this benchmark is shown in Figure 5.3.

On the Dryad benchmark suite, a total of 59 programs were verified for all three learning-based algorithms. The winnow algorithm took 13.8% more rounds as compared to Sorcar and 9.6% fewer rounds as compared to Houdini. As for the total time, the Winnow algorithm took 139% more time as compared to Sorcar and 10.5% more time as compared to Houdini. For all the programs in this class of benchmarks, Winnow performed worse as compared to Sorcar and Houdini both in terms of the number of rounds and total time taken. The summary of results on this benchmark is shown in Figure 5.4.

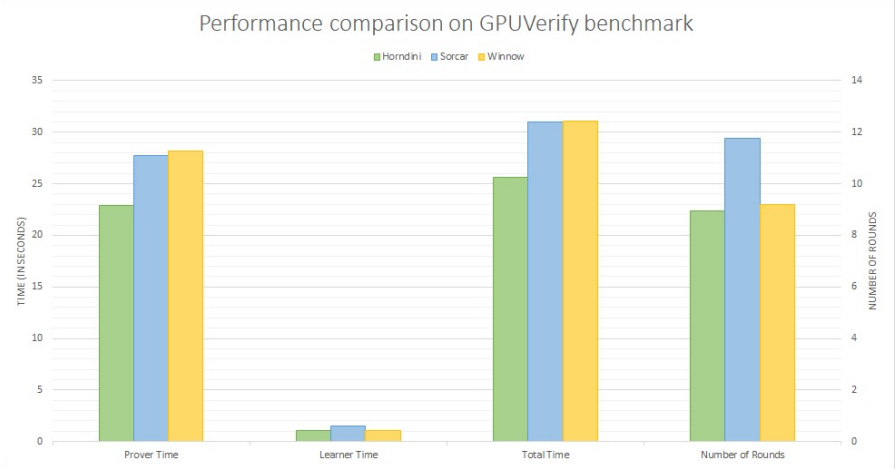

Figure 5.3: Method2: Evaluation of Winnow on GPUVerify Benchmarks

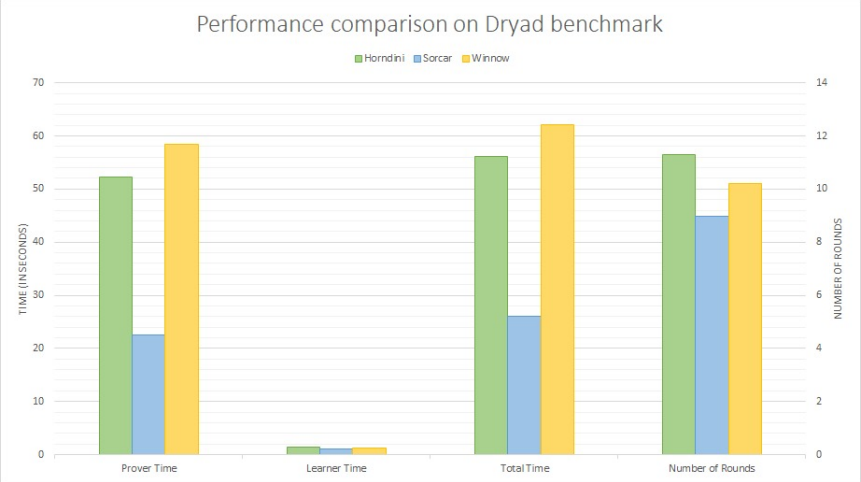

Figure 5.4: Method2: Evaluation of Winnow on Dryad Benchmarks

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, we propose a learner-based invariant generation called Winnow-ICE. We implement Winnow-ICE as a plug-in to the Horn ICE framework and evaluate the performance of Winnow against tools like Sorcar and Houdini for different methodologies. Factors such as the choice of initial weights, initial θ, and learning rate are crucial for the convergence of Winnow. We observe that Winnow improves the number of rounds drastically on the GPU-Verify benchmark; however, the total run time is similar to that of Sorcar. We also observe that Winnow improves neither the number of rounds nor the run time on the Dryad benchmark.

In the future, we plan to evaluate Winnow's performance on other state-of-the-art tools. We also want to implement and evaluate many different heuristics of labeling implication counterexamples that might help in decreasing the number of rounds and total time. Currently, Winnow is a plug-in framework inside the Sorcar code base and we would like to make Winnow a standalone tool.

REFERENCES

[1] E. Denney and B. Fischer, "Explaining verification conditions," in *International Conference on Algebraic Methodology and Software Technology*. Springer, 2008, pp. 145–159.

[2] H. Zhu, S. Magill, and S. Jagannathan, "A data-driven CHC solver," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3192366.3192416 p. 707–721.

[3] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: Association for Computing Machinery, 1977. [Online]. Available: https://doi.org/10.1145/512950.512973 p. 238–252.

[4] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, ser. FME '01. Berlin, Heidelberg: Springer-Verlag, 2001, p. 500–517.

[5] A. Miné, "The octagon abstract domain," *Higher-order and Symbolic Computation*, vol. 19, no. 1, pp. 31–100, 2006.

[6] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 69–87.

[7] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177.

[9] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387.

[10] B. Cosman and R. Jhala, "Local refinement typing," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, pp. 1–27, 2017.

[11] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2254064.2254112 p. 405–416.

[12] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 343–361.

[13] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," *SIGPLAN Not.*, vol. 36, no. 5, p. 203–213, May 2001. [Online]. Available: https://doi.org/10.1145/381694.378846

[14] A. R. Bradley, "Sat-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation*, R. Jhala and D. Schmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87.

[15] A. Champion, T. Chiba, N. Kobayashi, and R. Sato, "ICE-based refinement type discovery for higher-order functional programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 365–384.

[16] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, "Linear invariant generation using non-linear constraint solving," in *Computer Aided Verification*, W. A. Hunt and F. Somenzi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 420–432.

[17] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," *SIGPLAN Not.*, vol. 51, no. 1, p. 499–512, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2914770.2837664

[18] S. Krishna, C. Puhrsch, and T. Wies, "Learning invariants using decision trees," *CoRR*, vol. abs/1501.04725, 2015. [Online]. Available: http://arxiv.org/abs/1501.04725

[19] A. Gupta and A. Rybalchenko, "InvGen: An efficient invariant generator," in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 634–640.

[20] K. L. McMillan, "Interpolation and sat-based model checking," in *Computer Aided Verification*, W. A. Hunt and F. Somenzi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1–13.

[21] D. Neider, S. Saha, P. Garg, and P. Madhusudan, "Sorcar: Property-driven algorithms for learning conjunctive invariants," in *Static Analysis*, B.-Y. E. Chang, Ed. Cham: Springer International Publishing, 2019, pp. 323–346.

[22] P. Ezudheen, D. Neider, D. D'Souza, P. Garg, and P. Madhusudan, "Horn-ICE learning for synthesizing invariants and contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276501

[23] T. Mitchell, *Machine Learning*, ser. McGraw-Hill International Editions. McGraw-Hill, 1997. [Online]. Available: https://books.google.com/books?id=EoYBngEACAAJ

[24] N. Littlestone, "Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm," *Machine Learning*, vol. 2, no. 4, pp. 285–318, Apr 1988. [Online]. Available: https://doi.org/10.1023/A:1022869011914

[25] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: A verifier for GPU kernels," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2384616.2384625 p. 113–132.

[26] D. Neider, P. Garg, P. Madhusudan, S. Saha, and D. Park, "Invariant synthesis for incomplete verification engines," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds. Cham: Springer International Publishing, 2018, pp. 232–250.

[27] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, p. 453–457, Aug. 1975. [Online]. Available: https://doi.org/10.1145/360933.360975

[28] L. Lamport, "Predicate transformers for concurrency," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, p. 396–428, July 1990. [Online]. Available: https://doi.org/10.1145/78969.78970

[29] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, pp. 65–386, 1958.

[30] M. J. Kearns and U. V. Vazirani, *An Introduction to Computational Learning Theory.* Cambridge, MA, USA: MIT Press, 1994.

[31] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1–3, p. 35–45, Dec. 2007. [Online]. Available: https://doi.org/10.1016/j.scico.2007.01.015

[32] J. Kivinen, M. K. Warmuth, and P. Auer, "The Perceptron algorithm versus Winnow: Linear versus logarithmic mistake bounds when few input variables are relevant," *Artificial Intelligence*, vol. 97, no. 1-2, pp. 325–343, 1997.