Carl von Ossietzky
Universität
Oldenburg

II. School of Computing Science,
Business Administration,
Economics, and Law
Department of Computing Science

Bachelor's Thesis

Submitted to the Formal Methods Research Group
in Partial Fullfilment of the Requirements for the Degree of
Bachelor of Science

Using ICE Learning to generate Interpolants for Predicate Abstraction

by
Joshua Falk

Thesis Supervisor:
Prof. Dr. Heike Wehrheim,
M.Sc. Jan Haltermann

Oldenburg, January 1, 2024

Erklärung

Hiermit versichere ich an Eides statt, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Ort, Datum	Unterschrift

Abstract. Predicate abstraction is a common model checking method which relies on interpolation. This thesis tests, whether interpolants can be generated with Machine Learning by adapting the ICE-Learning method from [GLMN14] to interpolation. When adapting ICE-Learning the question of how to used the implication data introduced in this method can be used. This thesis argues that for the context of interpolation, the implication data from ICE-Learning is not useful. The ICE-Learning based method is implemented and tested, revealing that ML and specifically ICE-Learning can be used for interpolation.

Contents

1 Introduction

Verifying a property of a program is a complex and difficult task, which lead to the introduction of automatic verification tools, called *model checker*. One such property is the safety of a program, which can be verified by determining the reachable states of that program. Due to the structure of some programs, their set of reachable states might be infinitely large. To avoid this case, the reachable states can be over-approximated with a method called *predicate abstraction*. In this method, sets of multiple concrete states are represented by a combinations of predicates. Finding the right predicates for a useful abstraction is important and not trivial, but can be automated in an iterative manner using *Counterexample-Guided Abstraction Refinement* (CEGAR). Thereby, the abstraction is searched for indefeasible paths to an error state, which can be used to generate new predicates by finding the *interpolant* of the first-order formula representing this path.

As shown, *interpolation* is an important part of verifying the safety of a program, therefor, finding an efficient method for finding interpolants is important. This thesis explores the feasibility of a Machine Learning (ML) based interpolation by applying the learning method presented in [GLMN14] to interpolation.

Chapter 3 explains the concept of this ICE-Learning based interpolation. For this, Section 3.2 defines the structure and generation of data points for the learning process inspired by the data points used in [SNA12] and argues that the implication data points introduced in [GLMN14] are not useful for the context of interpolation. This ICE-Learning based interpolation is implemented and tested with benchmarks in Chapter 4. Goal of these benchmarks is to validate whether the implementation can learn valid interpolants and how the chosen Learner or size of the data set impact the performance of the implementation.

2 Fundamentals

This thesis relies on some underlying concepts. This chapter will summarize these concepts and introduce the notations used in the following chapters. Additionally, the running code example is presented in Figure 2.1.

2.1 Program Representation

Based on [JPR18, Chapter 15] programs will be considered in the form of $P = (V, pc, init, T, err)$. V is a set of numerical and boolean variables. The primed variables $v' \in V'$ represent for each variable $v \in V$ the state of the variable after an execution step. The transitions in T are formulas over V and V' representing execution steps in the program, $pc \in V$ is the program counter, *init* are the initial states and *err* are the error states, both represented by formulas over the program variables V. With this representation the *post condition* of a formula over the variables, called state s, and a transition t is defined as $post(s,t) = \exists V'' : s[V''/V] \wedge t[V''/V][V/V']$ [JPR18, Section 15.4.1]. The post condition of a state and a transition represent the result of executing the transition.
Another representation for program is given by its *control flow automaton* (CFA). A CFA $C = (L, \ell_0, G)$ consists of a set of program locations L, modeling the program counter, an initial location $\ell_0 \in L$ and a set of control-flow edges $G \subseteq (loc \times Op \times loc)$.

```
1 function(int x, y, z) {
2   assume(y > z);
3   while (x > z) {
4     x = x - z;
5   }
6   assert(y > x);
7 }
```

Figure 2.1: Example program

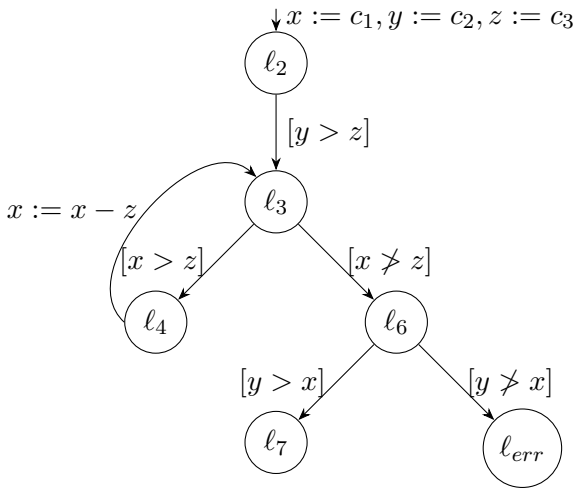


Figure 2.2: control flow automaton (CFA) of example program

These are directed edges between program locations *loc* and are annotated with assignment and assumption operations *Op*, modeling the program operations [BGS18, Section 16.3.1].
The code example in Figure 2.1 would be represented as $P = (V, pc, init, T, err)$ with
$V = \{pc, x, y, z\}$
$T = \{(pc = \ell_2 \wedge pc' = \ell_3 \wedge y > z \wedge x' = x \wedge y' = y \wedge z' = z),$
$(pc = \ell_3 \wedge pc' = \ell_4 \wedge x > z \wedge x' = x \wedge y' = y \wedge z' = z),$
$(pc = \ell_4 \wedge pc' = \ell_3 \wedge x' = x - z \wedge y' = y \wedge z' = z),$
$(pc = \ell_3 \wedge pc' = \ell_6 \wedge x \leqq z \wedge x' = x \wedge y' = y \wedge z' = z),$
$(pc = \ell_6 \wedge pc' = \ell_7 \wedge y > x \wedge x' = x \wedge y' = y \wedge z' = z),$
$(pc = \ell_6 \wedge pc' = \ell_{err} \wedge y \leqq x \wedge x' = x \wedge y' = y \wedge z' = z)\}$
$init = (pc = \ell_2 \wedge x = c_1 \wedge y = c_2 \wedge z = c_3)$ with $c_{1,2,3} \in \mathbb{R}$
$err = (pc = \ell_{err})$

The CFA of the code example can be seen in Figure 2.2, where the line numbers are used as program locations.

2.2 Reachability and Program Safety

A program's execution can be represented as a sequence of states derived from the transitions of the program, with each pair of adjacent states representing a transition of the program. We consider a state *reachable*, if a program execution exists that contains this state in the sequence of states [JPR18, Section 15.2.2-15.3.1].
For a programs safety, a set of error states is defined. A Program is then considered *safe* if these error states can not be reached in any execution. To prove that a state is not reachable, the set of all reachable states can be calculated. This, for example, can be achieved by calculating the post condition of all transitions or exploring the control flow graph (CFG) of a program.

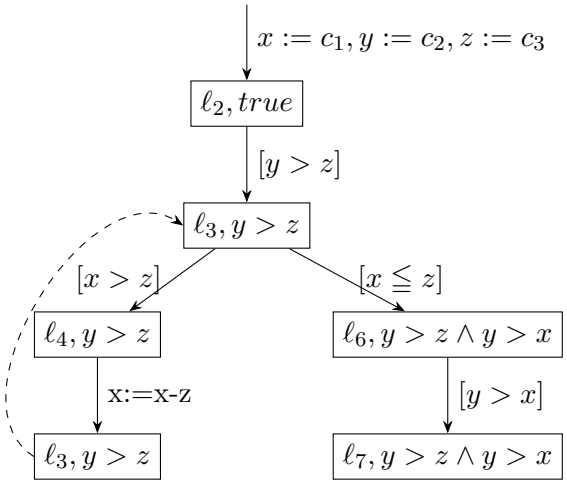


Figure 2.3: abstract reachability graph (ARG) of example program

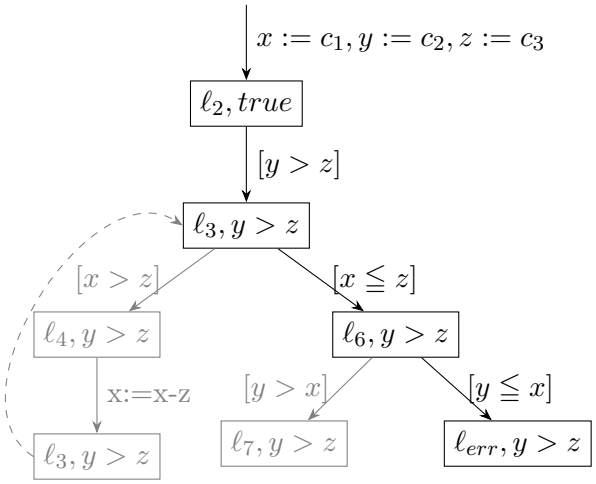
2.3 Predicate Abstraction

A program's set of reachable states can be infinitely large, introducing the need to abstract the state space. One such abstraction can be achieved with *Predicate Abstraction* [GS97]. It uses a predefined set of predicates called *precision* to abstract the states of the concrete program. An abstract state can represent multiple concrete states and is constructed by the conjunction of predicates entailed by these concrete states.
In the example in Figure 2.1 the concrete state $s_{con} = (pc = \ell_6 \wedge y > z \wedge x \leqq z \wedge x = c_1 - nc_3 \wedge y = c_2 \wedge z = c_3), n \in \mathbb{N}$ can be reached. Using the predicates $\{y > z, y > x\}$, the state can be abstracted in the following way:

$$y > z \Rightarrow y > z \vDash s_{con} \Rightarrow y > z$$

$$y > z \wedge x \leqq z \Rightarrow y > z \vDash s_{con} \Rightarrow y > z$$

The concrete state s_{con} implies the predicates $y > z$ and $y > x$ so the corresponding abstract state is $s_{abs} = (y > z \wedge y > x)$.
Different concrete states can result in the same abstract state, therefor allowing the abstract states to represent multiple concrete states. As the states are reduced to a combination of finite predicates, the resulting states space is finite as well.
By using this smaller abstract state space the reachable states can be over-approximated. If the possibly larger set of reachable states represented by the abstraction does not contain a specific state, this state is also not reachable in the concrete state space. This allows Predicate Abstraction to be used for checking the reachability of error states.
From this abstraction an *abstract reachability graph* (ARG) [BHJM07] can be generated. This directed graph $ARG = (n, succ, root, W, \phi)$ contains a set of abstract states



predicates: $true, y > z$

Figure 2.4: ARG of example program with error path

N, a successor relation $succ \subseteq N \times G \times N$, detailing for each reachable abstract state the previous state with G being the control-flow edges of a CFG, an abstract state $root \in N$ with no predecessor, a set of frontier nodes $W \subseteq N$ of nodes that have not been fully examined for successors and a precision ϕ containing information about the abstraction, in this case the used predicates [BJ21, Section 2.4]. Each node of the ARG holds information about the program location and the corresponding abstract state from the predicate abstraction.
The ARG of the example program is shown in Figure 2.3. In this graphical representation each node contains the program location and the formula representing the abstract state. The edges are annotated with the corresponding conditions and operations from the CFA. Using the predicates $(true)$, $(y > z)$ and $(y > x)$, the concrete state space can be abstracted by only five abstract states. Additionally, the path through the loop computes a state that is already covered by the predecessor $(\ell_3, y > z)$, reducing the state space of the loop from possibly infinite to only two states.
Leaving out the predicate $(y > x)$ produces an abstraction with a path to an error, shown in Figure 2.4. It follows that the quality of the predicate analysis is highly dependent on the precision. However, finding the right predicates is not trivial. To minimize manual effort, the predicates can be generated automatically using *Counterexample-Guided Abstraction Refinement* (CEGAR).

2.4 CEGAR

Counterexample-Guided Abstraction Refinement (CEGAR) [CGJ+00] is a method for automatically refining abstractions. It works in the following manner:

1. An initial abstraction is generated.

2. The model checking algorithm is run on the abstraction. If an error is found, this *counterexample* is checked on the concrete program, if no error is found the program is safe

   - If the counterexample can be reproduced in the concrete program, it is a real counterexample, the process stops and the error is reported to the user.
   - If the counterexample can not be reproduced in the concrete program, it is a *spurious* counterexample and the process continues to the next step.

3. The abstraction gets refined. For this some information is obtained from the spurious counterexample and used to change the abstraction. This ensures the counterexample will not be present in the next abstraction. The process is repeated from step 2 until no spurious counterexample is found.

In the context of predicate abstraction the counterexamples are in the form of paths from an initial to an error state. If these *counterexample paths* are spurious, additional predicates can be obtained from the path. Adding these predicates to the abstraction insures the spurious counterexample path does not exist in the next iteration.

Using the example in Figure 2.4, a counterexample path can be found, represented by the sequence of reached program location along the path: $(\ell_2, \ell_3, \ell_6, \ell_{err})$. Validating this path on the concrete program yields the following: The assume in line 2 gives $(y > z)$ and since the loop is not entered $(x \leqq z)$ is true. It follows $(y > z \wedge z \geqq x) \Rightarrow (y > z \geqq x) \Rightarrow (y > x)$. Therefor the assert on line 6 holds and the error state is not reached, concluding that the counterexample is spurious.
From this spurious counterexample the predicates $(true)$, $(y > z)$ and $(y > x)$ are generated at the different program locations in the path. This is achieved by finding the *interpolant* of the post condition of the previous subpath and the post condition of the remaining subpath. Adding the new predicate $(y > x)$ allows the abstract states to store more information, making the error path unreachable in the next abstraction.

2.5 Interpolation

Craig's Interpolation Theorem [Cra57], also known as *Craig Interpolation* or just *interpolation*, applied to first-order Logic proves the following: Take two formulas A and B with $(A \wedge B \equiv \bot)$. Since the conjunction these formulas can not be fulfilled, they are considered *mutually exclusive*. For these mutually exclusive formulas A and B a third formula I, called *interpolant* of A and B, exists with the following properties:

$$vars(I) \subseteq vars(A) \cap vars(B) \tag{2.1}$$

$$A \Rightarrow I \tag{2.2}$$

$$B \wedge I \equiv \bot \tag{2.3}$$

The interpolant I contains only variables present in both A and B (2.1), A implies I (2.2) and I and B are mutually exclusive (2.3) [Lyn59].

In our example the new predicate $(y > x)$ comes from the interpolant of

$$A = postCondition(path(2, 3, 6)) = (y > z \wedge z \geqq x)$$

$$B = postCondition(path(6, err) = (x \geqq y)$$

It can be shown, that A an B are mutually exclusive, therefor an interpolant can be generated. Assume that $I = (y > x)$ is an interpolant of A and B. It must fulfill the equations (2.1), (2.2) and (2.3):

$$(2.1) : \{x, y\} \subseteq x, y$$

$$(2.2) : (y > z \wedge z \geqq x) \Rightarrow (y > z \geqq x) \Rightarrow (y > x)$$

$$(2.3) : (x \geqq y \wedge y > z) \Rightarrow (x \geqq y > x) \Rightarrow (x > x \equiv \bot)$$

Therefor $I = (y > x)$ is an interpolant of $A = (y > z \wedge z \geqq x)$ and $B = (x \geqq y)$.

2.6 ICE-Learning

Learning using Examples, Counter-examples, and Implications (ICE-Learning) [GLMN14] is a Machine Learning (ML) framework originally proposed for learning loop invariants. It divides the learning process into two components, a *Teacher* and a *Learner*, as depicted in Figure 2.5. The Teacher is fully aware of the program and the property to be validated. It has two tasks, first generate samples from the program for the Learner to process and second validate the result from the Learner for the desired property. The Learner is unaware of any context and simply finds a classifier for the given samples.
The Learning process then follows an iterative loop. The Teacher generates an initial sample and the Learner proposes a classifier. The Teacher checks the classifier for the desired properties and generates additional samples if the properties are not met, repeating until a classifier satisfying the properties is learned.
When faced with validating the learned classifier to be a valid invariant, a situation might occur where the invariant is not inductive but succeeds the other to checks. In this case, two points are relevant, one inside the classification that implies another one outside the classification. When using only positive and negative data, the Teacher has to arbitrarily choose whether to label both of these points positive or negative. As this introduces some non-determinism on the side of the Teacher, the learning process is not guaranteed to converge. To combat this issue, ICE-Learning introduces a new type of data: *implication data*. Where positive and negative data is a singular point

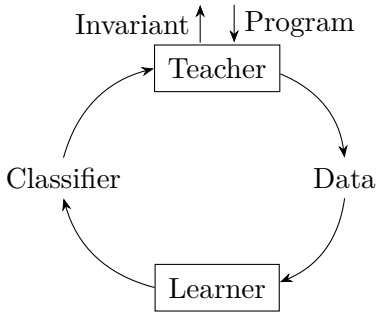


Figure 2.5: ICE-Learning structure

that should be true or false, implication data consist of two points. The classification of these points can not be determined a-priori but is instead defined by a logic implication. Given the points p and p' their implication sample is of the form $p \Rightarrow p'$. This means, if p is part of the classification, p' has to be part of the classification as well or p need to be left out.

2.7 Support Vector Machine

A *support vector machine* (SVM) is a supervised ML method for classifying input data into two classes. This is achieved by considering the input data as points in an n-dimensional space, with n being the amount of variables in the input, and finding a hyperplane that separates the input data. These hyperplanes can then be described by a formula of the form $wx + b = 0$ with w and b being constant vectors. To improve the generalization the hyperplane with the maximal distance from the input points is chosen from the possible options. As this process only works with linear separable inputs, SVMs allow the misclassification of some input points [Moh17].
For the context of model checking, finding a correct classifier is absolutely necessary, therefor, input data that is not linear separable can not be classified with a single SVM. For these cases a combination of multiple hyperplanes can be used by running multiple different SVM instances. An example for this approach can be found in [SNA12], where an intersection of half spaces, defined by the hyperplanes, is used to classify not linear separable data sets.

2.8 Decision Tree

A *decision tree* (DT) is a supervised ML method that can be used to classify input data into multiple classes. A DT is build step by step, where a new subtree is generated on a leaf by finding the attribute that has the best *information gain* for the subset represented by the leaf. The information gain is determined by the reduction of the *entropy* when the value for the attribute is known, whereby the entropy represents the distribution of the classes in the subset. A higher entropy points to a mixture

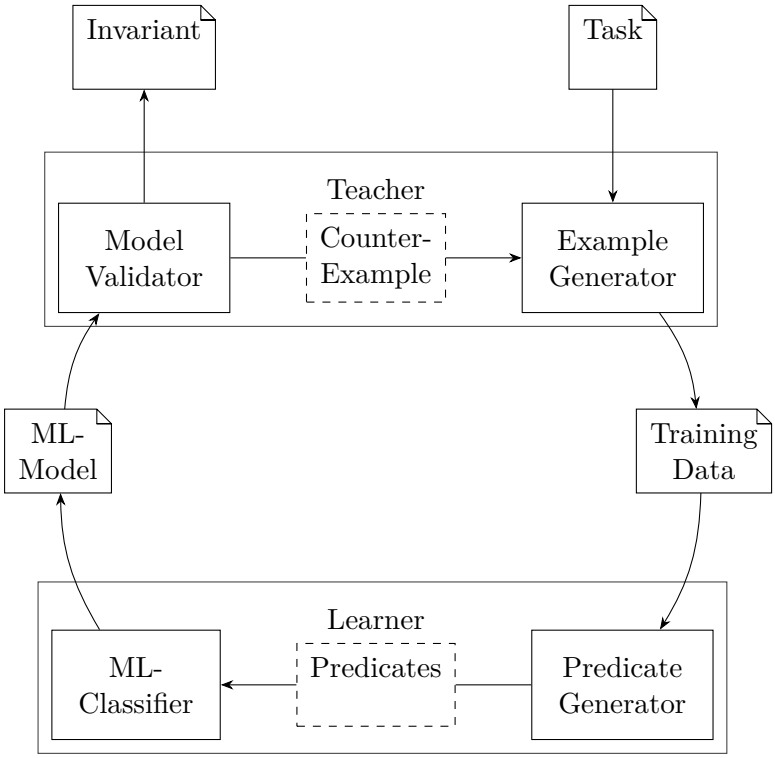


Figure 2.6: MIGML structure

of multiple classes, while a low entropy points to a containing mostly a single class. New subtrees are generated until the entropy in the leafs is small enough. To better generalize and prevent overfitting the DT gets pruned, removing subtrees that overfit the input data [Moh17].
In the context of model checking, the DT only uses the two classes $true$ and $false$. In this case, each node in the DT represents an atomic formula, which enables the DT to be transformed into a first-order formula by a conjunction of these formulas. Since model checking requires an exact classification of the input data, subtrees are generated until the entropy in the leafs equals 0 and the pruning of subtrees is discarded.

2.9 MIGML

The *Modular framework for Invariant Generation using Machine Learning* (MIGML) [HW22] is a framework for generating invariants with ML. It was first introduced to compare different ML based invariant generators in an equal environment. This is achieved by splitting the process into multiple modular components, depicted in Figure 2.6. The overall structure of Teacher and Learner from [GLMN14] is used, but the Teacher and Learner are further split into components, enabling the implementation of concepts not following this structure.
A MIGML Teacher consists of a *Example Generator* and a *Model Validator*. The Teacher's Example Generator is used to generate data points for the Learner, as an

initial data set or from a counterexample given by the Model Validator. The Teacher's Model Validator checks whether the learned ML-Model is a valid invariant for the program. If it is, the process finishes and returns the invariant, otherwise a counterexample is passed to the Example Generator. The counterexample consists of a formula for the violated condition and satisfying assignments for this formula.
A MIGml Learner consists of a *Predicate Generator* and a *ML-Classifier*. The Predicate Generator is used to find some predicates over the variables in the data set. These predicates, expressing logic connections over the variables, can be used as additional inputs for the ML-Classifier. The ML-Classifier learns a classification consistent with the data set and the predicates.
Communication between the Teacher and Learner is file-based, where the Learner expects a set of data points that are positive, negative or implications. The Learner encodes its classifier as a boolean combination of predicates over the program variables and returns this so called ML-Model.

3 Concept for ICE based Interpolation

This chapter describes the concept of this thesis' ICE-Learning based interpolation method. The process explained in this chapter generates an interpolant for either two mutually exclusive formulas or one unsatisfiable formula. Section 3.1 describes the process of the iterative learning of interpolants. Section 3.2 details the form and generation of the data used in the learning process and argues, whether implication data can be used in context of interpolation. Section 3.3 describes the functions of the interpolation Teacher and Section 3.4 describes the Learner used in this implementation.

3.1 Overall Learning Process

The overall learning process orientates itself on the structure of ICE-Learning [GLMN14] and the MIGml [HW22] framework. As depicted in Figure 3.1 the implementation is divided into multiple components, a Teacher, a Learner, a controller and an interface handling communication between Teacher and Learner.
The Learner is an independent component unaware of the context and goal of the learning process. Its sole function is to find a classifier that splits a given set of data into two classes. This classifier is then translated int a formula in first-order logic. To further decouple the Learner from the process, the communication interface is used. The interface converts between the internal data structures of the controller and the external files used for communication. The Teacher is aware of the interpolation task and uses this information to generate data points for the Learner. Additionally, the Teacher validates whether a classifier form the Learner is a valid interpolant. The controller is used to manage the input and output of the program, the stored data and the data exchange between the other components.
The process starts by the controller receiving an unsatisfiable formula in the SMT-LIBv2 [SMT] format. This formula is transformed into negative normal form and split along a conjunction into two formulas called A and B. These formulas are mutually

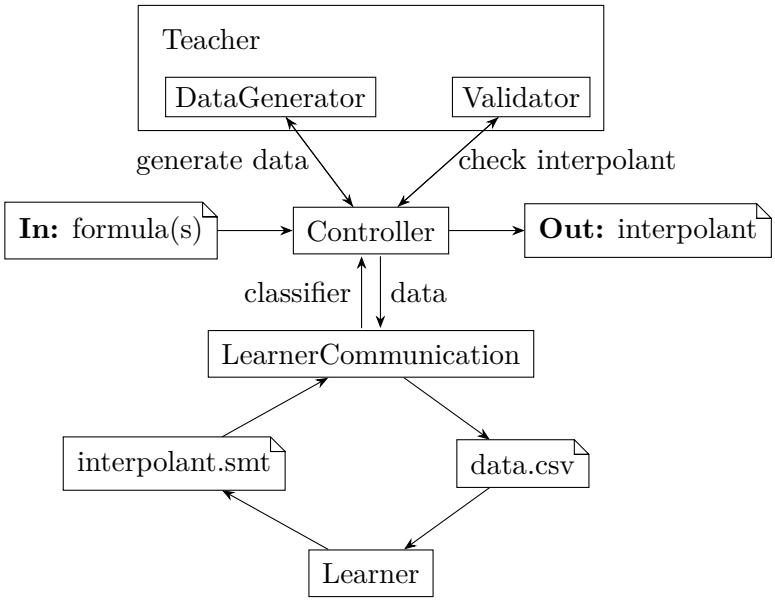


Figure 3.1: Data exchange between components

exclusive since $A \wedge B$ is equivalent to the original unsatisfiable formula, therefor an interpolant of A and B can be generated. Alternatively the controller can receive two individual formulas, eliminating the need for a split.
Using these input formulas A and B the iterative learning loop in Figure 3.2 can be executed. The controller first determines and saves the common variables in A and B. An initial data set is generated with the Teacher's DataGenerator by finding assignments for the input formulas A and B as explained in Section 3.2 and reducing them to the common variables.
An interpolant candidate called *hypothesis* H is generated by transforming the classifier from the Learner into a formula. The current data set gets written to a file and the Learner is executed. After a successful execution, the Learner writes the hypothesis to a file, which gets parsed to the internal format and returned to the controller.
Afterwards, the controller passes the hypothesis to the Teacher's Validator to check the conditions $(A \Rightarrow H)$(eq. (2.2)) and $(B \wedge H \equiv \bot)$(eq. (2.3)). If both checks hold, the process terminates and returns the valid interpolant, otherwise assignments for the failed checks are generated as described in Section 3.2. These new assignments are then used in the Teacher's DataGenerator to expand the data set and the process is repeated until a valid interpolant is found.

3.2 Generating the Learning Data

The data for learning logic formulas consist of assignments to all their free variables, referred to as *points* from here on. Points that are supposed to satisfy the learned formula are labeled *positive* and points that are supposed to not satisfy the learned formula are labeled *negative*.

```
1  interpolate(Formula a,Formula b, int dataSetSize){
2    //input: formula A and B to be interpolated
3    //determine common variables of a and b
4    commonVars = calculateCommonVars(a,b);
5    //generate initial data set
6    data=DataGenerator.getInitialData(a,b,commonVars,dataSetSize);
7    while(true){
8      //run until interpolant found
9      //get interpolant hypothesis from Learner
10     Formula h = learnerCall(data);
11     //check if true interpolant
12     //generate data from failed checks
13     validationResult,modelA,modelB =
14       Validator.validateHypothesis(a,b,h);
15     if (validationResult == true) {
16       //hypothesis is true interpolant, terminate
17       return h;
18     } else {
19       //hypothesis is not interpolant of A and B
20       //expand data set and try again
21       data = DataGenerator.updateData(
22         data,modelA,modelB,commonVars);
23     }
24   }
25 }
```

Figure 3.2: Interpolation loop

In the context of interpolation these points can be generated in multiple ways. From the property $(A \Rightarrow I)$(eq. (2.2)) follows that all satisfying assignments of A are also satisfying assignments of the interpolant I. Therefor, positive points can be obtained by finding satisfying assignments of A. Furthermore, when testing a hypothesis H generated by the Learner for violation of this property, an assignment for $(A \wedge \neg H)$ might be found. As this assignment satisfies A it should also satisfy the interpolant and can be added as a positive point [SNA12].
The second interpolation property $(B \wedge I \equiv \bot)$(eq. (2.3)) implies that all assignments that satisfy B do not satisfy the interpolant I. Therefor, satisfying assignments of B can be used as negative points. Similar to the first property, when testing the hypothesis H from the Learner for violation of the second property, an assignment for $(B \wedge H)$ might be found. As all satisfying assignment of B should not satisfy the interpolant this assignment can be added as a negative point [SNA12].
Since the interpolant only contains variables common in A and B, the generated points can be reduced to assignments for the common variables only.
For the use of ICE-Learning's implication data two points are needed where one implies the other in some way. In the work from Garg et al. [GLMN14] implications are first

introduced for learning invariants. Invariants have a property that the conjunction of the invariant with the loop condition implies the invariant after execution of the loop body. Therefor, the values of the invariant variables before the loop execution imply the values of the invariant variables after the execution. The two points of the implication naturally follow from the properties of invariants and are important for the learning process.

For the case of interpolation the properties represented by eq. (2.1), eq. (2.2) and eq. (2.3) do not contain the same sub formula on both sides of an implication, therefor not constructing a situation where one point is dependent on another. As a consequence implication data can not be generated and is not necessary in the context of interpolation.

Additionally, implication data is used when the label of the dependent points can not be determined before the ML classification. The use of implication data solves this problem by propagating labels along the implications whenever one side of the implication becomes labeled during the classification. In the case of interpolation any arbitrary point can safely be labeled before the classification. It only needs to be checked, whether the point (partially) satisfies A or B. Points satisfying exclusively A or B can be labeled as discussed above and points satisfying both A and B can not exist due to the condition of A and B being mutually exclusive. Points satisfying neither A nor B will automatically satisfy the conditions $(A \Rightarrow I)$ and $(B \wedge I \equiv \bot)$ making them irrelevant to the validity of the interpolant. While these points can be labeled as either positive or negative, opening the possibility for future optimization, arbitrarily choosing one label for all of them does not effect whether a valid interpolant can be learned.

In conclusion, implication data can not be used in a meaningful way when learning interpolants.

```
1  validateHypothesis(Formula a,Formula b,Formula h{
2    if (a∧¬h is satisfiable) {
3      //A ⇒ I violated
4      modelA = getAssignments(a∧¬h);
5      return validationResult=false,modelA=modelA,modelB=null;
6    } else if (b∧h is satisfiable) {
7      //B ∧ I ≡ ⊥ violated
8      modelB = getAssignments(b∧h);
9      return validationResult=false,modelA=null,modelB=modelB;
10   } else {
11     //is correct interpolant
12     return validationResult=true,modelA=null,modelB=null;
13   }
14 }
```

Figure 3.3: Validator: validate hypothesis

```
1  getInitialData(Formula a,Formula b,commonVars, size){
2    //get assignments satisfying A or B
3    modeslA = Validator.getAssignments(a, size/2);
4    modelsB = Validator.getAssignments(b, size/2);
5    //reduce to assignments for common variables and add label
6    pointsA = generatePointsFromModels(modelsA,commonVars, true);
7    pointsB = generatePointsFromModels(modelsB,commonVars, false);
8    //return as single data set
9    return {pointsA + pointsB};
10 }
11 updateData(data,modelA,modelB,commonVars){
12   if (modelA exists) {
13     pointsA = generatePointsFromModels(modelsA,commonVars, true);
14     data.add(pointsA)
15   }
16   if (modelB exists) {
17     pointsB = generatePointsFromModels(modelsB,commonVars, false);
18     data.add(pointsB)
19   }
20 }
```

Figure 3.4: DataGenerator

3.3 Teacher Concept

The main purpose of the Teacher is to generate data for the Learner and check interpolant candidates. Inspired by MIGml [HW22] these tasks are divided into two sub-components, the *DataGenerator* and the *Validator*.
The DataGenerator has two functions as depicted in Figure 3.4: generate an initial data set of specified size by finding assignments for the input formulas and update the existing data set with assignments for the failed validation checks as described in Section 3.2. In both cases the assignments are reduced to the common variables of the input formulas.
The Validator is tasked with finding assignments for formulas and checking hypotheses from the Learner on the conditions defining an interpolant. For finding the assignments a *SMT Solver* is used. SMT is short for *Satisfiability Modulo Theories* and describes the process of determining the satisfiability of a formula in first-order logic given a specific background theory [BSST09]. To find multiple assignments for a formula, each generated model is negated and added to the formula before generating the next model. This is repeated until either the specified number of models is generated or the formula is no longer satisfiable.
The validation of the hypothesis is shown in Figure 3.3. Assuming the input formulas are called A and B and the hypothesis H, the condition $(A \Rightarrow H)$ can be validated by checking if $(A \wedge \neg H)$ is satisfiable. If the formula is not satisfiable the condition holds.

Assuming that the Learner perfectly classifies all points, all assignments satisfying A do not satisfy B and all points in the data set that satisfy A also satisfy H by definition. Therefor, these assignments satisfying A but not H can not be part of the existing data set and the assignments generated from this check can be used to generate additional data for the next iteration. The same holds for the condition $(B \wedge H \equiv \bot)$ by simply checking whether $(B \wedge H)$ is satisfiable [SNA12]. The advantage of this approach over checking the conditions itself is that the satisfying assignments can be directly used as new data points in the next iteration.

3.4 Learner Concept

The task of the Learner is to generate an interpolant candidate from a set of data points. This *hypothesis* is generated by using ML to find a classifier of the data set. For this task specifically classifiers that can be transformed into first-order logic formulas are required. Considering this criteria, SVMs, combinations of multiple SVMs and DTs are good candidates for ML classifiers. The linear formulas learned by a SVM are already formulas in first-order logic eliminating the need for a conversion all together. The same holds true for conjunctions or disjunctions of SVMs. While DT Learner do not produce formulas directly, a DT can be transformed into a formula very easily. Additionally, SVMs and DTs are widely used techniques, resulting in access highly performant implementations.
To allow the reuse of existing Learners, the Learner uses predefined files to communicate with the rest of the system. These files follow the format used in MIGml [HW22].

4 Implementation and Evaluation

4.1 Implementation

To evaluate the ICE-Learning based interpolation, the concept in Chapter 3 has been implemented in Java 17. It expects either a single unsatisfiable formula or two mutually exclusive formulas as input. These formulas have to be in SMT-Libv2 [SMT] format with a singular assume containing the formula. When given a singular unsatisfiable formula, it is split recursively into two sub formulas until both sub formulas are satisfiable. Thereby the formula is divided into multiple smaller formulas such that their conjunction is equivalent to the original formula. These smaller formula are divided into two sets of roughly equals size and the conjunctions of these sets are the two sub formulas. If one of the sub formulas is not yet satisfiable, it is split again using one part of the split as the new sub formula and adding the second part of the split to the other sub formula.
In order to find assignments to generate the data points, the implementation uses the Z3 [dMB08] SMT solver. The output of the implementation consists of the files used for communication with the Learner, the final interpolant in SMT-LIB format and the last used data set as a csv table.

4.2 ML Configurations

Multiple optimization possibilities can be derived from the concept for the Teacher in section 3.3. Naturally the choice of SMT solver for finding assignments has great impact on the performance, but since it is easily interchangeable this will not be further explored in this work.
Regarding the learning process itself, the first optimization possibility presents itself in the size of the initial data set. A small initial data set reduces the needed calls of the theorem solver, whereby a larger data set might reduce the needed iteration for

finding a true interpolant. Another optimization possibility exists by choosing when to start the next learning attempt after checking the hypothesis. The possible options are whether to run the next iteration as soon as the first check finds an assignment or always checking both conditions for assignments. Additionally multiple assignments could be generated for the failed checks before a new learning attempt is started. Finally, the used Learner can effect the rate at which the learning converges.
Based on this the implementation will be adapted and tested with the combinations of following options

init20 ten assignments of A and B each in the initial data set

init200 one hundred assignments of A and B each in the initial data set

corr1 generate one assignment per failed check

corr5 generate five assignments per failed check

SVM use a SVM as the Learner

DT use a DT as the Learner

4.3 Evaluation

The evaluation focuses on the following questions:

RQ1 Can the ICE-Learning based interpolation learn valid interpolants?

RQ2 How do the configurations affect the performance of the learning process?

In order to evaluate these questions the different configurations are tested in a benchmark. The splitting of the input formulas is evaluated in a separate benchmark. This allows the other benchmarks to focus on the performance of the learning alone, since the time needed to calculate the split is constant and unaffected by the configurations.

4.3.1 Setup

These Question are evaluated using benchmarks. The benchmarks are run using the benchmarking tool BENCHEXEC [BLW19] inside a virtual machine (VM). The VM is configured to run Ubuntu 20.04.3 LTS in 64-bit with four cores and 8096 MB of memory. The host machine runs Windows 10 Pro 22H2 in 64-bit on an AMD Ryzen 5 2600X with six cores, a frequenzy of 3.6 GHz and 16 GB of Memory. The use of resources is not limited beyond the limits of the VM, apart from a timeout of five minutes.
The benchmarking tasks are taken from the SMT-LIB benchmarks[1], which are also used in the SMT-COMP [WCD+19]. From the category of quantifier-free linear integer algebra (QF_LIA), two set of benchmark tasks are chosen, limiting the choice to

[1] https://smtlib.cs.uiowa.edu/benchmarks.shtml Accessed: 2023-11-29

tasks labeled unsatisfiable and containing a single assume in the SMT-LIB file. The first set contains 20 tasks mostly from the CAV_2009_benchmarks set with increasing variable count or increasing coefficient size. The second set contains 9 task from the CAV_2009_benchmarks, check, cut_lemmas and wisa sets, specifically chosen to complete within the timeout.
The benchmarks test these eight configurations: (**SVM, init20, corr1**), (**DT, init20, corr1**), (**SVM, init20, corr5**), (**DT, init20, corr5**), (**SVM, init200, corr1**), (**DT, init200, corr1**), (**SVM, init200, corr5**), (**DT, init200, corr5**).

4.3.2 Benchmark Results

In order to answer **RQ1**, the number of successful executions are considered, since the process only terminates when a valid interpolant is learned. The results for the first test set in Table 4.2 show, that most but not all of the tasks did not finish within the time limit of five minutes. Therefor, the implementation is theoretically able to find valid interpolants, but for arbitrary formulas a fast execution can not be guaranteed. Specifically for the used tasks from the SMT-LIB benchmark, finding assignments to generate the data set is difficult as the formulas are designed to be difficult for SMT solvers.
To test the impact of splitting the formulas on the performance, the splitting has been run separately before the actual benchmark. The results of this benchmark, summarized in Table 4.1, show a good performance of the splitting. Additionally, removing the splitting from the interpolation benchmark did not reduce the number of timeout. It follows, that the splitting of the input formula does not significantly impact the performance of the process.
Since the first test set mostly results in timeouts, the impact of the different configurations can not be evaluated effectively. In order to properly compare the configurations a second test set is used, with tasks chosen specifically to terminate without timeout in at least one of the configurations. The result of this benchmark is summarized in Table 4.3. Apart from the performance, the chosen Learner also impacts which tasks terminate before timeout, as two of the task only terminate with the SVM and another one only with the DT. The following observations can be made from the results in Table 4.3: the SVM finishes with fewer iterations than the DT, increasing the size of the data set reduces the iterations needed with SVM by a small amount, whereas for the DT the iterations are reduced significantly. The SVM profits from larger initial data set and larger data set per iteration in similar amounts, while the DT profits much more from a larger data set per iteration than from a larger initial data set. In all cases increasing the data set reduced the runtime. Finally, while the runtime of the DT Learner is worse than the runtime of the SVM Learner in most configurations, for the configuration (init200, corr5), where the DT has roughly twice as many iterations as the SVM, the runtime is lower. This suggests that the DT finishes an iteration much

Table 4.1: Results of splitting benchmark

min time (s)	max time (s)	average time (s)	mean time (s)
0.576	77.7	6.41	0.94

Table 4.2: Results of first test set

	Timeouts	Successful	Total runtime (s) of successful	Total iterations in all configs
SVM, init20, corr1	17	3	27.45	9
DT, init20, corr1	17	3	26.19	6
SVM, init20, corr5	17	3	29.17	5
DT, init20, corr5	16	4	29.71	6
SVM, init200, corr1	16	4	40.45	11
DT, init200, corr1	17	3	34.76	6
SVM, init200, corr5	17	3	38.49	5
DT, init200, corr5	16	4	33.48	4

faster than the SVM.

In conclusion, the ICE-Learning based interpolation is able to generate valid interpolants. A larger data set reduces the needed iterations and therefor significantly improves the performance of the learning process. The SVM Learner has better performance with small data sets, as the number of iterations is affected less by the size of the data set, whereas for larger data sets the DT Learner has better performance, as the needed iterations strongly reduce with larger data sets and the time per iteration is much shorter.

Table 4.3: Results of second test set

	Timeouts	Successful	Total runtime (s) of successful	Total iterations in all configs
SVM, init20, corr1	1	8	103.87	32
DT, init20, corr1	2	7	144.64	138
SVM, init20, corr5	1	8	75.84	25
DT, init20, corr5	2	7	79.05	64
SVM, init200, corr1	1	8	74.72	23
DT, init200, corr1	2	7	119.52	92
SVM, init200, corr5	1	8	72.21	20
DT, init200, corr5	2	7	63.88	44

5 Related Work

This thesis concerns itself with generating interpolants by the use of ICE-Learning. These interpolants are then used in the context of predicate abstraction. There have been other works about defining interpolation methods or using ICE-Learning, as well as other use cases for interpolants in the context of model checking. These are shortly summarized in this chapter.

Examples for other interpolation methods are presented by McMillan [McM05, McM03], Kupferschmid and Becker [KB11] and Rybalchenko and Sofronie-Stokkermans [RS07]. In contrast to these logic based interpolations, this thesis follows a data driven approach. Another data driven interpolation method is given by Sharma et al. in [SNA12]. They present an algorithm that interpolates two formulas in linear arithmetic using Machine Learning. This is achieved by viewing the interpolation as a classification problem. Satisfying assignments for the formulas A and B are generated, whereby assignments for A represent positive dat points and assignments for B represent negative data points in the data set. To account for the case that the set of positive and the set of negative data points are not linear separable, the examples are classified by an intersection of half-spaces. These half-spaces are generated using off-the-shelf SVMs.
While this hypothesis H correctly separates the positive and negative data, it is not necessarily an interpolant of A and B. Therefor, the formulas $(\neg A \wedge H)$ and $(B \wedge H)$ are checked for satisfiability and any satisfying assignments are added to the data set. If these formulas are not satisfiable the interpolation conditions $(A \Rightarrow H)$ and $(B \wedge H)$ hold and the learned hypothesis H is a true interpolant of A and B.
The author state that due to the ML process only relying on examples, regardless of how they are generated, the method can handle formulas with superficial non-linearity. The problem then lies in verifying the learned classifier as an interpolant, which might not be possible for arbitrary non-linear functions. The authors state that other interpolation methods fail on their non-linear example while their method succeeds, concluding

that this method has a larger pool of solvable problems.
While this thesis builds on the ideas of Sharma et al., it focuses more on building a modular system for learning interpolants where Teacher and Learner are separate components. Since the components communicate with files conforming to the formats in MIGml [HW22], any learner implemented in that framework can be used. In contrast Sharma et al. build the Teacher and Learner into a single algorithm. Furthermore Sharma et al. only considered positive and negative examples, while this thesis checks if the implications from ICE-Learning [GLMN14] can be used for learning interpolants. Finally the algorithm of Sharma et al. is implemented standalone expecting two formulas as inputs, whereas the implementation in this thesis is also able to generate interpolants by splitting a single unsatisfiable formula.

An implementation of ICE-Learning can be found in [GNMR16]. Garg et al. develop a DT Learner capable of handling ICE-Learning implications based on the algorithms of Quinlan [Qui86, Qui93]. In contrast to Quinlan's algorithm the construction of each subtree can not be handled independently, hence there was a need to introduce an order into the construction. This allows to propagate classifications of implication ends to the other end. Whenever the algorithm reaches a leaf with a sample of unclassified and only positive or negative points, all unclassified points in this sample will be classified positive or negative respectively. These classifications are then propagated through their respective implications, classifying their other ends accordingly. This insures consistency with the implications as each following construction of a subtree has to adhere to these classifications, handling it the same as any other positive or negative point.
Additionally, two methods are presented for deciding the attribute for splitting the sample when dealing with implications. For the first, the information gain of an attribute is penalized when splitting the sides of an implication into different samples. This is based on the observation that ICE Learners tend to classify both sides of an implication the same way. The other method changes the entropy of an attribute based on the probability of implication points being classified as positive or negative in the split.
Finally, to ensure convergence of the iterative learning of an invariant, a bounded version of the Learner is introduced.
While Garg et al. focus on learning of invariants and adapting a learner to ICE-Learning, this thesis' aim is to learn interpolants an consider the usability of implication for this purpose.
Further works adapting ICE-Learning are presented by Ezudheen et al. [END+18] and Champion et al. [CCKS20]. These works extend the scope of ICE-Learning implications to handle more complex logic systems while remaining in the context of finding invariants. In contrast, this thesis does not improve ICE-Learning but instead tries to apply it to a different problem.
Some other model checking methods using interpolation are given by McMillan in [McM03,

McM06] and by Vizel and Grumberg in [VG09]. These approaches use interpolation directly for the abstraction instead of predicate abstraction.
Building upon the fact that predicate abstraction might converge with arbitrary predicates, Jhala and McMillan propose an method for finding predicates that restricts the language of generated interpolants [JM06].

6 Conclusion

This thesis explored the use of ICE-Learning for generating interpolants. A learning process based on the structure of ICE-Learning has been designed and implemented, showing that ICE-Learning can be used for interpolation. However, the implication data points special to ICE-Learning have been found to not be useful in the context of interpolation. A set of benchmarks has been used to test the validity of the learning process and the performance of different configurations. From these benchmarks has been concluded, that a larger initial data set improves the performance of the process by reducing the needed iterations. An even greater effect has the number of data points generated per iteration, where increasing them greatly improved the performance. For the tested SVM and DT Learner has been found, that the SVM usually needs less iterations, while the DT finishes each iteration significantly faster. Additionally, the SVM Learner benefits less from the larger data set, therefor, given a large enough data set, the DT is better suited for interpolation.

For future works, the generation of data points still has some optimization possibilities that need to be tested. In this thesis, data points are generated by generating models for satisfiable formulas. Since each arbitrary point can be checked, whether it fulfills one of the input formulas, randomly generated points could be used for data generation as well. On the same note, points fulfilling neither of the input formulas do not impact the validity of the learner interpolant, allowing the them be labeled either positive or negative. Since choosing either label still affects the learner interpolant, they might be useful for reducing the number of iterations.

Further tests still present themselves in comparing the performance of this thesis' learning based interpolation method to existing logic based interpolation methods. Furthermore, to compare with the learning based interpolation of [SNA12], the effect of when to start the next iteration can be evaluated. In [SNA12] the next iteration is started as soon as the first of the two validity checks is false, whereas the benchmarks in this thesis always run both checks before running the next iteration.

After the evaluation has shown, that the interpolation method of this thesis might not be useful for all contexts, testing the method for the specific context of predicate abstraction is still open. By integrating this method into a model checker using predicate abstraction, not only can the performance for a specific task be evaluated but also, whether the learned interpolants improve the abstraction refinement by reducing the needed refinement steps. When testing in the context of predicate abstraction, the most suited Learner should be evaluated again, checking whether one Learner is better suited for this specific set of task instead of general tasks. Being in a specific context, checking which SMT solver generates the data points fastest for this context can be evaluated as well.

Bibliography

[BGS18] Dirk Beyer, Sumit Gulwani, and David A. Schmidt. Combining model checking and data-flow analysis. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 493–540. Springer, 2018.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *Int. J. Softw. Tools Technol. Transf.*, 9(5-6):505–525, 2007.

[BJ21] Dirk Beyer and Marie-Christine Jakobs. Cooperative verifier-based testing with coveritest. *Int. J. Softw. Tools Technol. Transf.*, 23(3):313–333, 2021.

[BLW19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transf.*, 21(1):1–29, 2019.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[CCKS20] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. Ice-based refinement type discovery for higher-order functional programs. *J. Autom. Reason.*, 64(7):1393–1418, 2020.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[Cra57] William Craig. Linear reasoning. A new form of the herbrand-gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.

[dMB08] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[END+18] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. Horn-ice learning for synthesizing invariants and contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA):131:1–131:25, 2018.

[GLMN14] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In Armin Biere and Roderick 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*, pages 69–87. Springer, 2014.

[GNMR16] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 499–512. ACM, 2016.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[HW22] Jan Haltermann and Heike Wehrheim. Machine learning based invariant generation: A framework and reproducibility study. In *15th IEEE Conference on Software Testing, Verification and Validation, ICST 2022, Valencia, Spain, April 4-14, 2022*, pages 12–23. IEEE, 2022.

[JM06] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In Holger Hermanns and Jens 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.

[JPR18] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. Predicate abstraction for program verification. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 447–491. Springer, 2018.

[KB11] Stefan Kupferschmid and Bernd Becker. Craig interpolation in the presence of non-linear constraints. In Uli Fahrenberg and Stavros Tripakis, editors, *Formal Modeling and Analysis of Timed Systems - 9th International Conference, FORMATS 2011, Aalborg, Denmark, September 21-23, 2011. Proceedings*, volume 6919 of *Lecture Notes in Computer Science*, pages 240–255. Springer, 2011.

[Lyn59] Roger C. Lyndon. An interpolation theorem in the predicate calculus. *Pacific Journal of Mathematics*, 9(1):129 – 142, 1959.

[McM03] Kenneth L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt Jr. and Fabio 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*, pages 1–13. Springer, 2003.

[McM05] Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[Moh17] Amr E Mohamed. Comparative study of four supervised machine learning techniques for classification. *International Journal of Applied*, 7(2):1–15, 2017.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, 1986.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[RS07] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*, pages 346–362. Springer, 2007.

[SMT]

[SNA12] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants as classifiers. In P. Madhusudan and Sanjit 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.

[VG09] Yakir Vizel and Orna Grumberg. Interpolation-sequence based model checking. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 1–8. IEEE, 2009.

[WCD+19] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT competition 2015-2018. *J. Satisf. Boolean Model. Comput.*, 11(1):221–259, 2019.