LEARNING FRAMEWORKS FOR PROGRAM SYNTHESIS

BY

SHAMBWADITYA SAHA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

ABSTRACT

The field of synthesis is seeing a renaissance in recent years, where the task is to automatically synthesize small expressions or programs. One of the most prominent techniques counterexample guided inductive synthesis (CEGIS), uses a teacher(verification oracle) and a learner(learning algorithm) to learn such expressions across multiple rounds. A learning framework is a sub-framework of CEGIS where the learner is entirely agnostic of the specification and learns only from input-output examples provided by the teacher as natural notions of counterexamples. Thus, learning frameworks for synthesis have three components: the verification oracle, the notion of a natural counterexample, and the learning algorithm.

The goals of this thesis are to study learning frameworks for synthesis, developing new and more efficient algorithms for learning, exploring new classes of counterexamples, and finding applications of synthesis to new domains. Specifically, by co-designing the notion of counterexamples, the learning algorithms, the verification oracle, and taking into account the aspects of the application domain, we achieved more effective program synthesis. We discuss learning frameworks for four different applications, illustrating the co-design of oracle-counterexample-learner for each of them.

For the first application, we developed a general purpose SyGuS solver for piece-wise functions, using multiple learners to learn parts of the expression modularly and then compose them together to get the final expression. Second, we considered the application of automatic verification, where we synthesized inductive invariants using incomplete verification oracles. We also propose a novel property driven ICE learning algorithm to learn conjunctive inductive invariants. We considered specification mining for the next two applications, where we learned preconditions and postconditions of a method. Instead of using a verification engine as the oracle, which is not efficient, does not scale, and needs loop invariants, we bypassed all these limitations by using a test generator as the oracle.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The field of synthesis is a classical discipline in formal methods that is seeing a renaissance, mainly due to a variety of new techniques [1, 2, 3, 4, 5] to automatically synthesize small expressions or programs, given input-output examples [6, 7] or correctness specifications [1, 8]. Such expressions are useful in niche application domains, including end-user programming for data manipulation such as the FlashFill feature of Microsoft Excel [7, 9, 10, 11, 12, 13, 14, 15], filling holes in program sketches [2, 3], program transformations [16, 17], computer-aided education [18, 19, 20, 21, 22, 23], synthesizing network configurations and migrations [24, 25, 26], optimizing code generated by compilers using Superoptimization [26, 27, 28, 29], concurrent programming [30, 31, 32], program repair [19, 22, 23, 33, 34, 35, 36], code suggestion [37, 38, 39, 40, 41], distributed transition systems [42, 43], probabilistic modelling [44], as well as synthesizing annotations such as invariants [45, 46, 47], and pre/post conditions [48]. The field of program synthesis has emerged as a thriving area in programming languages and formal methods (see the articles [4, 5, 49]).

In order to introduce the reader to a simple model of program synthesis, let us consider the following problem. Let us start with a *specification* of how an output relates to a tuple of inputs, and set the goal of synthesis to automatically synthesize an expression or a program implementation that realizes the given specification. More formally, let the synthesis problem be stated using a specification $\forall \vec{x}.\ \psi(f,\ \vec{x}\,)$, where ψ is a quantifier-free first order logic (that could be over a combination of interpreted theories) formula that uses a special uninterpreted function symbol f. The goal of synthesis is then to find a concrete expression e for the function f, that satisfies the specification, i.e., $\forall \vec{x}.\ \psi(e/f,\ \vec{x}\,)$ is valid. We hence want the concrete definition e to satisfy the specification for *all* inputs. Note that the specification can, of course, describe input-output examples as well, but in this case, we would also want the expression to *generalize* well (one way to formalize generalization would be to insist that e be a simple expression, in the style of an Occam's razor).

In recent synthesis settings, in addition to the (correctness) specification, a syntactic template for the desired expression is also provided. This syntactic constraint makes the problem more tractable by limiting the search space of the solution, and also giving the user fine-grained control over the potential solutions using a combination of syntactic and semantic constraints. The *syntax guided synthesis* (SyGuS) [1, 50, 51] problem, presents a standard input format to describe such problems, very similar to SMT-LIB [52], the common interchange format used in SMT solvers. The SyGuS format thus provides a way to formalize the syntactic guidance under a general framework based on logics and grammars.

Many algorithmic approaches have been proposed over the years to solve the synthesis problem. Classically, *deductive synthesis* [53, 54] derives the program from the constructive proof of a theorem using logical inferences and constraint solving. Another technique is to fix a template structure syntax for the expression f, and formulate the synthesis problem as a formula in first order logic of the form $\exists \vec{t}\, \forall \vec{x}.\ \psi(\vec{t}, \vec{x})$, where $\vec{t}$ encodes the instantiation of the template and ψ interprets the instantiated template using its semantics. Quantifier elimination methods can then be used to reduce this problem to quantifier free satisfiability [5] (also see recent work by us on such elimination algorithms [55]).

Alternatively, *inductive synthesis* [56, 57, 58] techniques learn from input-output examples, generalizing from them to synthesize program expressions. An input-output pair consist of concrete values that characterize the behavior of the function to synthesize, i.e., when the function is called with each input, its return value is the corresponding output. Learning from input-output example closely resembles problems in the field of machine learning [59], especially the subfield of inductive programming, which has a long tradition in solving this problem using inductive methods [60, 61]. Machine learning, which is the field of learning algorithms that builds models from training data, is a rich field that encompasses algorithms for several problems, including classification, regression, and clustering [59]. The template-based approach discussed earlier, when used with input-output examples, has the added benefit that we no longer need the universal quantification, and the formula becomes quantifier-free. This is so because instead of quantifying over all inputs, the specification needs to hold only on the provided input-output examples, and hence the universal quantification can be replaced by a conjunctive formula.

Counter-example guided inductive synthesis (CEGIS): Turning to synthesis from a more general specification than input-output examples, a new technique called counter-example guided inductive synthesis (CEGIS) has emerged [1, 2, 3, 62]. The CEGIS approach [3] for program/expression synthesis advocates pairing inductive learning algorithms with a verification oracle (see Figure 1.1). The idea is to have the learning algorithm propose hypotheses expression to samples given by the verification oracle, in rounds of interaction, until it finds an expression that the oracle can verify to satisfy the specification. In each round, the verification oracle, when it finds the hypothesized expression to not satisfy the specification, produces *counterexamples* that have concrete values and show why the proposed hypothesis is wrong. A majority of the current synthesis approaches rely on counterexample guided inductive synthesis [3, 45, 46, 63].

The CEGIS framework [1] for the FO specifications of the form $\forall \vec{x}.\ \psi(f, \vec{x})$ that we fixed, works as follows. The framework maintains a global set of counterexamples, which is

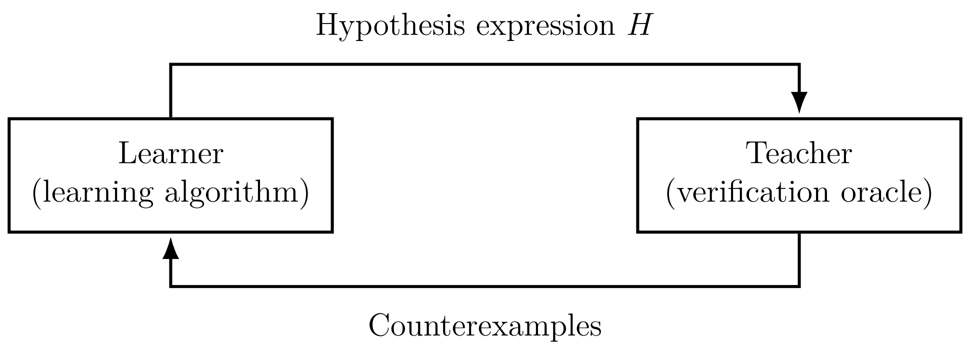


Figure 1.1: Counter-Example Guided Inductive Synthesis (CEGIS) framework structure.

initially set to be empty. The learning takes place across multiple rounds, where, in each round, the learner learns inductively from this global set of counterexamples, and proposes a hypothesis H. The verification oracle then checks the validity of the hypothesis H against the specification $\forall\vec{x}.\ \psi(f,\vec{x})$. It does so by checking if the formula obtained by substituting the hypothesis H for f, i.e. $\forall\vec{x}.\ \psi(H/f,\vec{x})$, is valid. If this is the case, then learning stops and the hypothesis H is declared as the solution. Otherwise, the verification oracle proposes as counterexample a concrete valuation of $\vec{x}$ (using a satisfying model of the negation of the formula), and adds it to the global set of counterexamples. The learner then proceeds to the next round on the new set of counterexamples. Notice that though the valuation of $\vec{x}$ is concrete, the learner needs to know the specification in order to understand what this counterexample means.

Learning Frameworks for Synthesis: Though all the synthesis techniques mentioned in this thesis is based on CEGIS, we would like to differentiate it from the paradigm of "learning". A learning framework can be seen as a sub-framework of CEGIS where the learner is entirely agnostic of the specification and learns *only* from input-output examples. Note that the general CEGIS approach for FO specifications described is not of this form, as the learner needs to know the specification to understand what the counterexample means.

In learning frameworks, we would like the learner to be agnostic of the specification and also learn from a *natural* notion of samples. For example, when learning a predicate over $\vec{i}$, a natural notion of a sample would be valuation of $\vec{i}$ with a +/- label. When learning functions, a natural notion of a sample is input-output examples for the function being synthesized.

In this thesis, we will explore learning frameworks for synthesis, and in particular design several frameworks where the oracle can be designed to give natural samples to a specification-agnostic learner.

1.1 THESIS WORK

The synthesis problem, viewed through the learning lens, naturally poses synthesis as a learning problem from concrete samples. However, the counterexamples that the verification oracles provide as witnesses for showing a hypothesis is incorrect, cannot always be seen as natural samples that can be interpreted by a learner that does not know the specification. Furthermore, the samples/counterexamples that the oracle can provide greatly varies according to the synthesis application.

The goals of this thesis are to study learning frameworks for synthesis, developing new and more efficient algorithms for learning, exploring new classes of counterexamples, and finding applications of synthesis to new domains.

The thesis statement is:

In learning for synthesis, co-designing the notion of counterexamples, the learning algorithms, the verification oracle, and taking into account the aspects of the application domain leads to more effective program synthesis.

In learning frameworks for synthesis, frameworks have three components: (1) the verification oracle, (2) the notion of a natural counterexample, and (3) the learning algorithm. All these three components required to be co-designed with the particular aspects of the application domain for effective synthesis.

In the rest of this chapter, we discuss several learning frameworks for different applications, illustrating the co-design of oracle-counterexample-learner for each of them. Table 1.1 summarizes these results. The later chapters describe each such framework in detail.

1.1.1 A SyGuS Solver for Synthesizing Piece-wise Functions and an Arithmetic Instantiation

In Chapter 2, we look into the application of building a general purpose SyGuS solver or synthesis engine to synthesize piece-wise functions (functions that split the domain into regions and apply simpler functions to each region) from both logical specifications as well as input-output examples. We propose a learning framework to build such an engine, where we can use multiple learners to learn simple functions for regions, use learners to synthesize predicates defining regions, and then compose them using a classifier.

The most natural class of counterexamples that can facilitate such learning are those that

Applications	**Framework Components**		
	Oracle	**Counterexamples**	**Learner**
SyGuS: Piece-wise Functions	SMT Solvers	Input-Output Examples	Compositional Expr. Synth. & Multi-label Classification
Verification: Inductive Invariants	Incomplete Verification Oracle	Non-Provability Information for Validity of Invariants	ICE Learners including SORCAR
Specification Mining: Preconditions	Test Generator	Valid and Invalid Abstractions of Input States	Conflict Resolver + Classification Algorithms
Specification Mining: Postconditions	Test Generator	Positively Labeled Abstractions of Input and Output States	Learner for Conjunctive Functional Concepts

Table 1.1: The different framework components of the applications we explored in this thesis.

precisely identify inputs for the synthesized function on which a given hypothesis is wrong. Thus the requirement on the learners fixes the class of counterexamples we consider. However, it turns out that *not all* synthesis specifications are such that such kinds of counterexamples can be found.

We develop a theory of *single-point definable specifications*, a semantic property, whose definition ensures such counterexamples always exist, and a subclass of *single-point refutable specifications*, a syntactic property, that reduces finding such counterexample to satisfiability problems over the underlying quantifier-free logic (which is often decidable). Our framework works robustly for the class of single-point refutable specifications and we thus use SMT solvers as the verification oracle.

We instantiate our framework to build a SyGuS solver for the class of conditional linear integer arithmetic (CLIA) expressions. Our implementation uses an SMT solver as the verification oracle, and learns the expression by combining leaf expression synthesis using constraint-solving with predicate synthesis using enumeration, and tying them together using a custom decision tree algorithm as the multi-label classifier. We demonstrate that this compositional approach is competitive compared to traditional synthesis engines on a set of CLIA specifications from the 2015 and 2016 SyGuS competitions [1, 51, 64]. Our compositional learning technique inspired the SyGuS solver EUSolver [65], which was able to synthesize piece-wise functions from the ICFP benchmarks [66] (which are input-output specifications on bit-vectors) for the first time in the SyGuS competitions.

To summarize, in this framework: Specifications are either input-output examples or single-point refutable specifications. Counterexamples are inputs on which a hypothesis is incorrect. The learner is a compositional learner that combines a synthesis engine that synthesizes expressions for subsets of inputs in various regions, an enumerative synthesis algorithm for synthesizing boundaries between regions, and a decision-tree based multi-labeled classifier for learning the final formula.

1.1.2 Synthesizing Inductive Invariants for Program Verification with Incomplete Logic Engines

In Chapter 3, we look into the application of deductive program verification, where we facilitate automatic verification by synthesizing inductive invariants. Once invariants are found, program verification reduces to checking validity of verification conditions (expressed in pure logic). Though prior learning-based counterexample guided inductive synthesis (CEGIS) methods have been proposed [45, 46, 47, 67], we look into the situation where the validity of the verification conditions is undecidable. In a learning-based synthesis framework, the verification oracle is hence incomplete i.e., the oracle resorts to sound but incomplete heuristics to check the validity of the verification conditions and hence cannot generate a concrete model when verification conditions are not provable.

The framework we propose in this chapter thus assumes that we have an incomplete verification oracle. In this setting, we extract certain non-provability information from the verification oracle as counterexamples when the conjectured invariant results in verification conditions that cannot be proven. The non-provability information is a Boolean formula on a fixed set of predicates that generalizes the reason for non-provability, hence pruning the space of future conjectured predicates.

The notion of the counterexamples being non-provability information dictates the learning algorithm we need to design to learn from such a sample of formulas. We reduce the formula-driven problem of learning expressions from non-provability information to the data-driven ICE model [45]. This reduction thus allows us to use a host of existing ICE learning algorithms and results in a robust invariant synthesis framework that guarantees to synthesize a provable invariant if one exists.

We evaluate our invariant synthesis framework to automatically verify two classes of programs. First, programs that dynamically manipulate the heap (singly and doubly linked lists, sorted lists, balanced and sorted trees). We use the *natural proof verification engines* [68] as the oracle against a undecidable separation logic called DRYAD [69], that combine properties of structure, separation, arithmetic, and data. Second, verification of programs against specifications with universal quantification, which renders verification undecidable in general. In both the cases we are able to automatically verify the programs efficiently.

To summarize, in this framework: The application is inductive invariant synthesis. We consider two verification oracles: one oracle for heap verification using natural proofs and the other a quantifier-instantiation based logic solver for annotations that involve universally quantified formulas. In both cases, the verification oracle is sound but not complete. The class of counterexamples encode *non-provability* information of particular predicates in the postcondition of Hoare-triples. The learning algorithm needs to propose hypotheses that include only provable concepts. We implement this using a reduction to ICE-learning of Boolean formulas.

1.1.3 Property-driven synthesis of conjunctive inductive invariants: The SORCAR ICE-Learning Algorithm

In Chapter 4, we present a novel ICE learning algorithm to learn conjunctive inductive invariants over a fixed finite set of predicates P. Houdini [70] is an existing learner to learn conjunctive inductive invariants, and synthesizes the *tightest* inductive invariant. Consequently, it can ignore the property to be proven about the program. However, the tightest invariant can be quite complex (have many conjuncts) and hard to synthesize.

We present SORCAR, a property driven learning algorithm for conjunctive inductive invariants and performs better than existing Houdini based tools on certain classes of benchmarks. Intuitively, SORCAR grows slowly a set of relevant predicates $R \subseteq P$ in each round and proposes the tightest conjunctive invariant over R. It guarantees convergence to a conjunctive invariant (if one exists over P) in $2|P|$ rounds of communication with any verification oracle.

We implement SORCAR on top of the BOOGIE program verifier [71]. We evaluate its efficiency on two domains of benchmarks. The first is the class of programs consisting of GPU programs handled by the tool GPUVerify [72, 73] to prove data race freedom. The second class of programs dynamically update heaps against specifications in separation logics [74]. We compared SORCAR to the current state-of-the-art tools for these programs, which use the HOUDINI algorithm. Though SORCAR did not work more efficiently on every program, however, our empirical evaluation shows that it is overall more competitive than HOUDINI. In general, SORCAR produces much smaller invariants, worked more efficiently overall in verifying these programs, and verified a larger number of programs than HOUDINI.

To summarize: The Sorcar learning algorithm is a general conjunctive ICE learning algorithm that can be used in many program verification algorithms. However, we use it in two settings: one for GPUVerify programs and the other where validation of verification conditions is incomplete (heap verification). The notion of counterexamples are either concrete states or non-provability information, respectively.

1.1.4 Specification Mining: Preconditions and Postconditions for Methods

In Chapter 5, we consider the problem of synthesizing contracts, where we propose frameworks to synthesize precondition and postcondition of a method in a class of a program in an object-oriented language. In this context, the natural oracle to use would be a verification engine that verifies programs (with loops/recursion). However, verification engines that can do completely automated program verification are not often effective or scalable. We hence consider using test generators as teaching oracles.

Given a program annotated with preconditions and assertions, the test generator creates a valid object state (using object modifying methods) and concrete input parameters of the method that satisfy the precondition. Furthermore, several test generators are guided by the assertions, and try to generate inputs that violate them.

Synthesizing Preconditions:

In the learning framework we develop for synthesizing preconditions, we use as counterexamples abstractions of the input states, which consists of the primitive type inputs, and the valuations of a certain set of observer methods of the non-primitive-type input objects.

Each input state (and similarly for counterexample) created by the test generator, can be either valid: execute successfully and terminate, or invalid: encounter an uncaught exception, or result in an assertion violation. Note that the predicates used in the logic for expressing preconditions and the observer methods for deriving properties of objects, create abstractions of input states. Abstractions of invalid input states must be excluded by the precondition. However, abstractions of valid input states need not necessarily be included in the precondition, as there could be input states with the same abstraction but are invalid.

We define the problem of precondition synthesis using a notion of *ideal preconditions*. An ideal precondition for a method with respect to a test generator is a precondition which satisfies two properties. First, *safety*: the test generator should not be able to find any invalid input state allowed by the precondition. Second is *maximality*: the precondition should include as many valid input states as possible. More precisely, it can exclude an abstraction of input states only if there exists some invalid input state that has this abstraction. The maximality requirement intuitively captures the desire to synthesize weakest (most liberal) preconditions.

In our learning framework, counterexamples are positively and negatively labeled abstraction of input states. The meaning of the counterexample is as follows. The learner needs to find a formula in a fixed logic L that (a) excludes all negatively labeled inputs, and (b) includes all positively labeled inputs i unless there is another sample i' that is negatively labeled and is indistinguishable from i by any formula in L.

The learning algorithms we propose for ideal preconditions first resolve all conflicts (conflicts are pairs of samples labeled positive and negative that are indistinguishable by the logic L) by reclassifying them as negative. A classification algorithm is then used to learn a hypothesis precondition from the conflict resolved samples.

We implement a prototype of our framework in a tool called PROVISO using a learner based on the ID3 classification algorithm [75], and PEX [76] as the test generator. We evaluate PROVISO on two important tasks in specification inference: runtime-failure prevention and conditional-commutativity inference [77]. The former problem asks to synthesize preconditions that avoid runtime exceptions of a single method. The latter problem asks, given two methods, find a precondition that ensures that the two methods commute, when called in succession. PROVISO takes on average ∼740 seconds per method/method pair to synthesize preconditions. Moreover, 91% of the preconditions synthesized by PROVISO are safe, while 77% of the preconditions are both safe and maximal.
To summarize, our framework for precondition synthesis has the following features. The application is synthesizing precondition of a method of a class in an object oriented program. The verification oracle is a test generator, which is inherently incomplete hence introduces conflicting counterexamples. The counterexamples are valid and invalid abstractions of input states. We formulate the synthesis problem using the notion of an ideal precondition with respect to the oracle and a particular logic L for stating preconditions; this implicitly gives the meaning of what counterexamples mean. The learning algorithm to find an ideal precondition first resolves all conflicts in the sample and then uses a classification algorithm that does not make any mistakes.

Synthesizing Conjunctive Postconditions

We finally propose a framework to synthesize conjunctive postconditions of a method using a test generator as the oracle. We assume that the method is already annotated with a precondition, which prevents all exception failures (this can be done using the precondition synthesis mentioned above). Synthesized postconditions need to be *strong*, and ideally the strongest postcondition expressible in a given logic.

In a learning framework, the test generator can only provide pairs of feasible input-output states (where input states satisfy the precondition). Consequently, given a hypothesis postcondition, a test generator can refute that the postcondition is correct by giving executions that end in states that are not satisfied by the postcondition. However, it cannot refute the assertion that the postcondition is the strongest one. In terms of counterexamples, we can think of the test generator as being able to only provide positively labeled pairs of abstractions of input and output states.

We propose to learn postconditions in a logic that consists of conjunctions of (a) predicates over a fixed set P and (b) an equality expression that defines an output as a function of the input. The predicates P and the parameters for the functions synthesized are based on input parameters and abstractions of objects using observer methods.

The above logic facilitates learning *tight* concepts (as the above logic is closed under conjunction). We synthesize functional relationships between input and output using a SyGuS solver. We then seed this as equality predicates and add them to P, and then use the elimination algorithm [78] to learn the semantically smallest conjunctive formula over the predicates that includes all the positive counterexamples.

We implement a prototype of our framework in a tool called Precis with Pex [76] as the test generator and the elimination algorithm as the learner. To synthesize equality predicates, we used the enumerative solver [65] from the SyGuS competition [1]. We evaluated our framework on datastructure methods from two open-source projects QuickGraph and the .NET Core. Precis was able to synthesize postconditions of reasonable size (average of 4.6 conjuncts per method), very efficiently taking an average time of 200s per method.

In summary, our learning framework for postcondition synthesis has the following features. The application is to learn a strong conjunctive postcondition of a method that is already annotated with a precondition. The counterexamples in this framework are only positive. The oracle is a test generator, and counterexamples are abstractions of pairs of input and output states that are always classified positively. The learning algorithm first synthesizes new predicates using a SyGuS solver and then uses the elimination algorithm to learn the tightest conjunctive postcondition.

These learning frameworks and the experimental results argue my thesis statement, that co-designing the notion of counterexamples, the learning algorithms, the verification oracle, and taking into account the aspects of the application domain leads to more effective program synthesis.

CHAPTER 2: COMPOSITIONAL SYNTHESIS OF PIECE-WISE FUNCTIONS BY LEARNING CLASSIFIERS

In this chapter, we propose a framework to build a general-purpose synthesis engine to synthesize piece-wise functions (functions that split the domain into regions and apply simpler functions to each region) from logical specifications or input-output examples.

In this framework, instead of learning the whole expression at once using one learner, we use multiple learners to learn different parts of the expression modularly. We use a learner for simple functions for fixed concrete inputs and another learner for predicates that can be used to define regions. We then join these expressions into one expression, using a multi-label classifier that does not make any mistakes and is biased towards learning smaller expressions, thus achieving generalization.

For logical specifications, the most natural class of counterexamples that can facilitate learning are those that precisely identify inputs on which a given hypothesis is wrong. However, it turns out that *not all* synthesis specifications are such that such kinds of counterexamples can be found.

We develop a theory of *single-point definable specifications*, a semantic property, whose definition ensures such counterexamples always exist, and a subclass of *single-point refutable specifications*, a syntactic property, that reduce finding such counterexample to satisfiability problems over the underlying quantifier-free logic (which is decidable). Our framework works robustly for the class of single-point refutable specifications and we thus use SMT solvers as the verification oracle.

In particular, in this chapter:

- Specifications are either input-output examples or single-point refutable specifications.
- Counterexamples are inputs on which a hypothesis is incorrect.
- The learner is a compositional learner that combines a synthesis engine that synthesizes expressions for a subset of inputs in a region, an enumerative synthesis algorithm for synthesizing boundaries between regions, and and a decision-tree based multi-labeled classifier for learning the final overall formula.

2.1 INTRODUCTION

We present a general technique that uses the CEGIS framework for synthesizing expressions, that can learn piece-wise functions. A piece-wise function is a function that partitions the input domain into a finite set of regions, and then maps each region using a simpler class of functions. In this setup, we synthesize *modularly* with the help of two other synthesis engines, one for synthesizing expressions for single inputs and another for synthesizing predicates that separate concrete inputs from each other. The technique is general in the sense that it is independent of the logic used to write specifications and the logic used to express the synthesized expressions. The counterexample guided synthesis proceeds in the following fashion:

- In every round, the learner proposes a piece-wise function H for f, and the verification oracle checks whether it satisfies the specification. If not, it returns one input $\vec{p}$ on which H is incorrect (Returning such a counterexample is nontrivial).
- We show that we can now use an *expression synthesizer* for the single input $\vec{p}$ which synthesizes an expression that maps $\vec{p}$ to a correct value. This expression synthesizer will depend on the underlying theory of basic expressions, and we can use any synthesis algorithm that performs this task.
- Once we have the new expression, we compute for every counterexample input obtained thus far the set of basic expressions synthesized so far that work correctly for these inputs. This results in a set of *samples*, where each sample is of the form $(\vec{p}, Z)$, where $\vec{p}$ is a concrete input and Z is the set of basic expressions that are correct for $\vec{p}$. The problem we need to solve now can be seen as a multi-label classification problem— that of finding a mapping from *every* input to an expression that is consistent with the set of samples.
- Since we want a classification that is a piece-wise function that divides the input domains into regions, and since the predicates needed to define regions can be arbitrarily complex and depend on the semantics of the underlying logical theory, we require a *predicate synthesizer* that synthesizes predicates that can separate concrete inputs with disjoint sets of labels. Once we have such a set of predicates, we are equipped with an adequate number of regions to find a piece-wise function.
- The final phase uses *classification learning* to generalize the samples to a function from all inputs to basic expressions. The learning should be biased towards finding *simple* functions, finding few regions, or minimizing the Boolean expression that describes the piece-wise function.

The framework above requires many components, in addition to the expression synthesizer and predicate synthesizer. First, given a hypothesis function H and a specification $\forall \vec{x}.\ \psi(f, \vec{x})$, we need to find a concrete counter-example input on which H is wrong.

It turns out that there may be *no* such input point for some specifications and even if there was, finding one may be hard. In current standard CEGIS approaches [1, 3], when H and $\forall \vec{x}.\ \psi(f, \vec{x})$ are presented, the teacher simply returns a concrete value of $\vec{x}$ for which $\neg\psi(H/f, \vec{x})$ is satisfied. We emphasize that such valuations for the universally quantified variables cannot be interpreted as inputs on which H is incorrect, and hence cannot be used with any learner that learns from input-output examples (including machine learning algorithms).

We develop a theory of *single-point definable specifications*, a semantic property, whose definition ensures such counterexample inputs always exist, and a subclass of *single-point refutable specifications*, a syntactic property, that reduce finding such counterexample inputs to satisfiability problems over the underlying quantifier-free logic (which is decidable). The framework of single-point refutable specifications and the counterexample input generation procedures we build for them is crucial in order to be able to use classifiers to synthesize expressions.

Our framework works robustly for the class of single-point refutable specifications, and we show how to extract concrete counterexamples, how to automatically synthesize a new specification tailored for any input $\vec{p}$ to be given to the expression synthesizer, and how to evaluate whether particular expressions work for particular inputs.

The classifier learning algorithm can be any learning algorithm for multi-label classification, which is the problem of learning a predictive model (i.e., a classifier) from samples that are associated with multiple labels (preferably with the learning bias as to learn small trees). However, the classifier learning algorithm must ensure that the learned classifier is *consistent* with the given samples (i.e., it is not allowed to misclassify datapoints in the training set). Machine-learning algorithms more often than not make mistakes and are not consistent with the sample, often because they want to generalize assuming that the sample is noisy. We proposed an adaptation of decision-tree learning to multi-label learning that produces classifiers that are consistent with the sample. We also explore a variety of statistical measures used within the decision-tree learning algorithm to bias the learning towards smaller trees in the presence of multi-labeled samples. The resulting decision-tree learning algorithms form one class of classifier learning algorithms that can be used to synthesize piece-wise functions over any theory that works using our framework.

We instantiated the framework to build an efficient synthesizer of piece-wise linear integer arithmetic functions for specifications given in the theory of linear integer arithmetic. We implement the components of the framework for single-point refutable functions: to synthesize input counterexamples, to reformulate the synthesis problem for a single input, and to evaluate whether an expression works correctly for any input. These problems are reduced to the satisfiability of the underlying quantifier-free theory of linear integer arithmetic, which is decidable using SMT solvers. The expression-synthesizer for single inputs is performed using an inner CEGIS-based engine using a constraint solver.

We also looked into the problem of synthesizing the expressions for the restricted settings of a unique specification where the specification permits only one solution. Assuming the function to synthesize f is of arity n, then the problem of finding the expressions that satisfy all the counterexamples can be viewed as the problem finding planes in a $(n+1)$-dimensional space where the space describes the input-output behavior of the function f. The synthesizer learns from all previously found counterexamples, where each counterexample is an input-output pair and can be viewed as a *point* in this $(n+1)$-dimensional space. We would essentially like to find a small set of planes, that include all the given points in this space. We solve this problem using a greedy algorithm that uses geometric techniques to determine coplanarity between points in this $(n+1)$-dimensional space.

The predicate synthesizer is instantiated using an enumerative synthesis algorithm. We use a straightforward modification of Quinlan's C 5.0 algorithm [75, 79] to solve the disjoint multi-label learning problem, and experimented with the different statistical measures to bias the learning towards smaller trees. The resulting solver works extremely well on a large class of conditional linear integer arithmetic benchmarks drawn from the SyGuS 2015 synthesis competition [51] and fared significantly better than all the traditional SyGuS solvers (enumerative, stochastic, and symbolic constraint-based solvers).

2.2 THE SYNTHESIS PROBLEM AND SINGLE-POINT REFUTABLE SPECIFICATIONS

The synthesis problem we tackle in this chapter is that of finding a function f that satisfies a logical specification of the form $\forall \vec{x}.\ \psi(f, \vec{x})$, where ψ is a *quantifier-free* first-order formula over a logic with fixed interpretations of constants, functions, and relations (except for f). Further, we will assume that the quantifier-free fragment of this logic admits a *decidable* satisfiability problem and furthermore, effective procedures for producing a model that maps the variables to the domain of the logic are available. These effective procedures are required in order to generate counterexamples while performing synthesis.

For the rest of the chapter, let f be a function symbol with arity n representing the target function that is to be synthesized. The specification logic is a formula in first-order logic, over an arbitrary set of function symbols $\mathcal{F}$, (including a special symbol f), constants $\mathcal{C}$, and relations/predicates $\mathcal{P}$, all of which with fixed interpretations, except for f. We will assume that the logic is interpreted over a countable universe D and, further, and that there is a constant symbol for every element in D. For technical reasons, we assume that negation is pushed into atomic predicates.

The specification for synthesis is a formula of the form $\forall \vec{x}.\ \psi(f, \vec{x})$ where ψ is a formula expressed in the following grammar (where $g \in \mathcal{F}$, $c \in \mathcal{C}$, and $P \in \mathcal{P}$):

$$\begin{aligned} \mathit{Term} \quad t \ &::-\ x \mid c \mid f(t_1, \ldots, t_n) \mid g(\vec{t}\,) && (2.1) \\ \mathit{Formula} \quad \varphi \ &::-\ P(\vec{t}\,) \mid \neg P(\vec{t}\,) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi && (2.2) \end{aligned}$$

We will assume that equality is a relation in the logic, with the standard model-theoretic interpretation.

The synthesis problem is to find, given a specification $\forall \vec{x}.\ \psi(f, \vec{x})$, a definition for the function f in a particular syntax that satisfies the specification. More formally, given a subset of function symbols $\widehat{\mathcal{F}} \subseteq \mathcal{F}$ (excluding f) and a subset of constants $\widehat{\mathcal{C}}$ and a subset of relation/predicate symbols $\widehat{\mathcal{P}} \subseteq \mathcal{P}$, the task is to find an *expression* e for f that is a term with free variables $y_1, \ldots, y_n$ adhering to the following syntax (where $\widehat{g} \in \widehat{\mathcal{F}}$, $\widehat{c} \in \widehat{\mathcal{C}}$, $\widehat{P} \in \widehat{\mathcal{P}}$)

$$\mathit{Expr} \quad e \ ::-\ \widehat{c} \mid y_i \mid \widehat{g}(\vec{t}\,) \mid \mathit{ite}(\widehat{P}(\vec{t}\,),\ e,\ e), \qquad (2.3)$$

such that e satisfies the specification (i.e., $\forall \vec{x}.\ \psi(e/f,\ \vec{x}\,)$ is valid).

2.2.1 Single-Point Definable Specifications

In order to be able to define a general CEGIS algorithm for synthesizing expressions for f based on learning classifiers, as described in Section 2.1, we need to be able to refute any hypothesis H that does not satisfy the specification with a concrete input on which H is wrong. We will now define sufficient conditions that guarantee this property. The first is a semantic property, called *single-point definable specifications*, that guarantees the existence of such concrete input counterexamples and the second is a syntactic fragment of the former, called *single-point refutable specifications*, that allows such concrete counterexamples to be found effectively using a constraint solver.

A single-point definable specification is, intuitively, a specification that restricts how each input is mapped to the output, *independent* of how other inputs are mapped to outputs. More precisely, a single-point definable specification restricts each input $\vec{p} \in D^n$ to a *set of outputs* $X_{\vec{p}} \subseteq D$ and allows any function that respects this restriction for each input. It cannot, however, restrict the output on $\vec{p}$ based on how the function behaves on other inputs. Many synthesis problems fall into this category (see Section 2.6 for several examples taken from a recent synthesis competition).

Formally, we define this concept as follows. Let $I = D^n$ be the set of inputs and $O = D$ be the set of outputs of the function being synthesized.

Definition 2.1 (Single-Point Definable (SPD) Specifications)**.** *A specification α is said to be* single-point definable *if the following holds. Let $\mathcal{F}$ be the class of all functions that satisfy the specification α. Let $g : I \to O$ be a function such that for every $\vec{p} \in I$, there exists some $h \in \mathcal{F}$ such that $g(\vec{p}) = h(\vec{p})$. Then, $g \in \mathcal{F}$ (i.e., g satisfies the specification α).*

Intuitively, a specification is single-point definable if whenever we construct a function that maps each input independently according to *some* arbitrary function that satisfies the specification, the resulting function satisfies the specification as well. For each input $\vec{p}$, if $X_{\vec{p}}$ is the set of all outputs that functions that meet the specification map $\vec{p}$ to, then any function g that maps every input $\vec{p}$ to some element in $X_{\vec{p}}$ will also satisfy the specification. This captures the requirement, semantically, that the specification constrains the outputs of each input independent of other inputs.

Let us illustrate this definition with the following examples.

Example 2.1. Consider the following specifications in the first-order theory of arithmetic:

- The specification

  $$\forall x, y.\ f(15, 23) = 19 \wedge f(90, 20) = 91 \wedge \ldots \wedge f(28, 24) = 35 \tag{2.4}$$

  is single-point definable. More generally, any set of input-output samples can be written as a conjunction of constraints that forms a single-point definable specification.

- Any specification that is not realizable (i.e., that has no function that satisfies it) is single-point definable.

- The specification

  $$\forall x.\ f(0) = 0 \wedge f(x + 1) = f(x) + 1 \tag{2.5}$$

  is single-point definable as the identity function is the only function that satisfies this specification. More generally, any specification that has a unique solution is single-point definable.

While single-point definable specifications are quite common, there are prominent specifications that are not single-point definable. For example, *inductive loop invariant synthesis* specifications for programs are not single-point definable, as counterexamples to the inductiveness constraint involve *two counterexample inputs* (the ICE learning model [45] formalizes this). Similarly, ranking function synthesis is also not single-point definable.

Note that for any single-point definable specification, if H is some expression conjectured for f that does not satisfy the specification, there will always be *one* input $\vec{p} \in D^n$ on which H is *definitely wrong* in that no correct solution agrees with H on $\vec{p}$. More precisely, we obtain the following directly from the definition.

Proposition 2.1. *Let $\forall \vec{x}.\ \psi(f, \vec{x})$ be a single-point definable specification and let $h\colon D^n \to D$ be an interpretation for f such that $\forall \vec{x}.\ \psi(f, \vec{x})$ does not hold. Then, there exists an input $\vec{p} \in D^n$ such that for* every *function $h'\colon D^n \to D$ that satisfies the specification, $h(\vec{p}) \neq h'(\vec{p})$.*

2.2.2 Single-Point Refutable Specifications

While the above proposition ensures that there is a counterexample input for any hypothesized function that does not satisfy a single-point definable function, it does not ensure that finding such an input is tractable. We now define single-point refutable specifications, which we show to be a subclass of single-point definable specifications, and for which we can reduce the problem of finding counterexample inputs to logical satisfiability of the underlying quantifier-free logic.

Intuitively, a specification $\forall \vec{x}.\ \psi(f, \vec{x})$ is single-point refutable if for any given hypothetical interpretation H to the function f that does not satisfy the specification, we can find a particular input $\vec{p} \in D^n$ such that the formula $\exists \vec{x}.\ \neg\psi(f, \vec{x})$ evaluates to true, and where the truth-hood is caused *solely* by the interpretation of H on $\vec{p}$. The definition of single-point refutable specifications is involved as we have to define what it means for H on $\vec{p}$ to solely contribute to falsifying the specification.

We first define an alternate semantics for a formula $\psi(f, \vec{x})$ that is parameterized by a set of n variables $\vec{u}$ denoting an input, a variable v denoting an output, and a Boolean variable b. The idea is that this alternate semantics evaluates the function by interpreting f on $\vec{u}$ to be v, but "ignores" the interpretation of f on all other inputs, and reports whether the formula would evaluate to b. We do this by expanding the domain to $D \cup \{\bot\}$, where $\bot$ is a new element, and have f map all inputs other than $\vec{u}$ to $\bot$. Furthermore, when evaluating formulas, we let them evaluate to b only when we are sure that the evaluation of the formula to b depended only on the definition of f on $\vec{u}$. We define this alternate semantics by *transforming* a formula $\psi(f, \vec{x})$ to a formula with the usual semantics but over an extended domain $D^+ = D \cup \{\bot\}$. In this transformation, we use if-then-else (*ite*) terms for simplicity. Moreover, given a vector $\vec{z} = (z_1, \ldots, z_\ell)$ (e.g., of variables), we use $\vec{z}[i]$ as a shorthand for the i-th entry z_i of $\vec{z}$ (i.e., $z[i] = z_i$) throughout the rest of this chapter.

Definition 2.2 (Isolate Transformer)**.** *Let $\vec{u}$ be a vector of n first-order variables (where n is the arity of the function to be synthesized), v a first-order variable (different from ones in $\vec{u}$), and $b \in \{T, F\}$ a Boolean value. Moreover, let $D^+ = D \cup \{\bot\}$, where $\bot \notin D$, be the extended domain, and let the functions and predicates be extended to this domain (the precise extension does not matter).*

For a formula $\psi(f, \vec{x})$, we define the formula $Isolate_{\vec{u},v,b}(\psi(f, \vec{x}))$ over D^+ by

$$Isolate_{\vec{u},v,b}(\psi(f,\vec{x})) \coloneqq ite\Bigg(\bigvee_{x_i} x_i = \bot, \neg b, Isol_{\vec{u},v,b}(\psi(f,\vec{x}))\Bigg), \tag{2.6}$$

where $Isol_{\vec{u},v,b}$ is defined recursively as follows:

$$Isol_{\vec{u},v,b}(x) \coloneqq x \tag{2.7}$$

$$Isol_{\vec{u},v,b}(c) \coloneqq c \tag{2.8}$$

$$Isol_{\vec{u},v,b}(g(t_1,\ldots,t_k)) \coloneqq ite\left(\bigvee_{i=1}^{k} Isol_{\vec{u},v,b}(t_i) = \bot, \bot, g\Big(Isol_{\vec{u},v,b}(t_1), \ldots, Isol_{\vec{u},v,b}(t_k)\Big)\right) \tag{2.9}$$

$$Isol_{\vec{u},v,b}(f(t_1,\ldots,t_n)) \coloneqq ite\left(\bigwedge_{i=1}^{n} Isol_{\vec{u},v,b}(t_i) = \vec{u}[i], v, \bot\right) \tag{2.10}$$

$$Isol_{\vec{u},v,b}(P(t_1,\ldots,t_k)) \coloneqq ite\left(\bigvee_{i=1}^{k} Isol_{\vec{u},v,b}(t_i) = \bot, \neg b, P\Big(Isol_{\vec{u},v,b}(t_1), \ldots, Isol_{\vec{u},v,b}(t_k)\Big)\right) \tag{2.11}$$

$$Isol_{\vec{u},v,b}(\neg P(t_1,\ldots,t_k)) \coloneqq ite\left(\bigvee_{i=1}^{k} Isol_{\vec{u},v,b}(t_i) = \bot, \neg b, \neg P\Big(Isol_{\vec{u},v,b}(t_1), \ldots, Isol_{\vec{u},v,b}(t_k)\Big)\right) \tag{2.12}$$

$$Isol_{\vec{u},v,b}(\varphi_1 \vee \varphi_2) \coloneqq Isol_{\vec{u},v,b}(\varphi_1) \vee Isol_{\vec{u},v,b}(\varphi_2) \tag{2.13}$$

$$Isol_{\vec{u},v,b}(\varphi_1 \wedge \varphi_2) \coloneqq Isol_{\vec{u},v,b}(\varphi_1) \wedge Isol_{\vec{u},v,b}(\varphi_2) \tag{2.14}$$

Intuitively, the function $Isolate_{\vec{u},v,b}(\psi)$ captures whether ψ will evaluate to b if f maps $\vec{u}$ to v and independent of how f is interpreted on other inputs. A function of the form $f(t_1, \ldots t_n)$ is interpreted to be v if the input matches $\vec{u}$ and otherwise evaluated to $\bot$. Functions on terms that involve $\bot$ are sent to $\bot$ as well. Predicates are evaluated to b only if the predicate is evaluated on terms none of which is $\bot$— otherwise, they get mapped to $\neg b$, to reflect that it will not help to make the final formula ψ evaluate to b. Note that when $\mathit{Isolate}_{\vec{u},v,b}(\psi)$ evaluates to $\neg b$, there is no property of ψ that we claim. Also, note that $\mathit{Isolate}_{\vec{u},v,b}(\psi(f,\vec{x}))$ has no occurrence of f in it, but has free variables $\vec{x}$, $\vec{u}$ and v. The following examples illustrates the isolate transformer.

Example 2.2. Consider the (single-point refutable) specification

$$\psi(f,x) = f(x) > x + 1 \tag{2.15}$$

in linear integer arithmetic over a single variable x. The formula $\mathit{Isol}_{\vec{u},v,b}$ will have free variables x, u, and v (note that x and u are variables not vectors of variables in this example).

In the first step, we adapt the semantics of the operator $+$ and the predicate $>$ to account for the new value $\bot$ by introducing a new operator $+_\bot$ and a new predicate $>_\bot$. Given two terms t_1 and t_2, the operator $+_\bot$ is defined by

$$t_1 +_\bot t_2 \coloneqq \mathit{ite}\Big(t_1 = \bot \lor t_2 = \bot, \bot, t_1 + t_2\Big), \tag{2.16}$$

while the predicate $>_\bot$ is defined by

$$t_1 >_\bot t_2 \coloneqq \mathit{ite}\Big(t_1 = \bot \lor t_2 = \bot, \bot, t_1 > t_2\Big). \tag{2.17}$$

In both cases, the result is $\bot$ if one one of the terms evaluates to $\bot$, whereas the original semantics is retained otherwise.

In the second step, we can now apply the isolate transformer to ψ:

$$\begin{aligned}
\mathit{Isol}_{\vec{u},v,b}(\psi(f,x)) &= \mathit{Isol}_{\vec{u},v,b}(f(x) > x + 1) && (2.18)\\
&= \mathit{Isol}_{\vec{u},v,b}(f(x)) >_\bot \Big(\mathit{Isol}_{\vec{u},v,b}(x) +_\bot \mathit{Isol}_{\vec{u},v,b}(1)\Big) && (2.19)\\
&= \mathit{ite}(x = u, v, \bot) >_\bot (x +_\bot 1). && (2.20)
\end{aligned}$$

In total, we obtain

$$\mathit{Isolate}_{\vec{u},v,b}(\psi(f,x)) = \mathit{ite}\Big(x = \bot, \neg b, \mathit{ite}(x = u, v, \bot) >_\bot (x +_\bot 1)\Big) \tag{2.21}$$

which captures whether ψ will evaluate to b if f maps u to v (and independent of how f is interpreted on other inputs).

We can show (using a induction over the structure of the specification) that the isolation of a specification to a particular input with $b = F$, when instantiated according to a function that satisfies a specification, cannot evaluate to false. This is formalized below.

Lemma 2.1. *Let $\forall \vec{x}.\ \psi(f,\ \vec{x}\,)$ be a specification and $h \colon D^n \to D$ a function satisfying the specification. Then, there is no interpretation of the variables in $\vec{u}$ and $\vec{x}$ (over D) such that if v is interpreted as $h(\vec{u})$, the formula $Isolate_{\vec{u},v,F}(\psi(f,\ \vec{x}\,))$ evaluates to false.*

Proof. Let $\forall \vec{x}.\ \psi(f, \vec{x})$ be a specification and $h \colon D^n \to D$ a function satisfying the specification. Moreover, let $\vec{u}$ a vector of variables over the domain D, v a variable over D, and $b \in \{T, F\}$ a Boolean value. Finally, fix a valuation $d_z \in D$ for each free variable z in $Isolate_{\vec{u},v,b}(\psi(f, \vec{x}))$ such that $d_v = h(d_{u[1]}, \ldots, d_{u[n]})$.

We split the proof into two parts:

1. We show that if $Isol_{\vec{u},v,b}(t)$ evaluates to a non-$\perp$ value (i.e., to a value in D) for a term t, then t evaluates to the same value.

2. Using Part 1, we show that if $Isol_{\vec{u},v,F}(\varphi)$ evaluates to false for a formula φ, the formula φ evaluates to false as well.

The claim of Lemma 2.1 then follow immediately from Part 2 and the definition of $Isolate_{\vec{u},v,b}$ since h satisfies the specification and the variable v is interpreted as $h(\vec{u})$.

We prove the first part using an induction over the structure of a term t.

Base case Let $t = x$ or $t = c$. Then, the claim holds immediately by definition of $Isol_{\vec{u},v,b}$.

Induction step In the induction step, we distinguish between $t = g(t_1, \ldots, t_k)$ and $t = f(t_1, \ldots, t_n)$.

- Let $t = g(t_1, \ldots, t_k)$ and assume that $Isol_{\vec{u},v,b}(t)$ evaluates to a non-$\perp$ value, say $d \in D$. By definition of $Isol_{\vec{u},v,b}$, this means that $Isol_{\vec{u},v,b}(t_i)$ evaluates to a non-$\perp$ value, say $d_i \in D$, for each $i \in \{1, \ldots, k\}$. Moreover, $Isol_{\vec{u},v,b}(t)$ evaluates to $g(Isol_{\vec{u},v,b}(t_1), \ldots, Isol_{\vec{u},v,b}(t_k))$ and, hence, $d = g(d_1, \ldots, d_k)$. Applying the induction hypothesis now yields that t_i also evaluates to d_i. Since $t = g(t_1, \ldots, t_k)$, this means that t evaluates to d, as claimed.

- Let $t = f(t_1, \ldots, t_n)$ and assume that $Isol_{\vec{u},v,b}(t)$ evaluates to a non-$\perp$ value. By definition of $Isol_{\vec{u},v,b}$, this means that $Isol_{\vec{u},v,b}(t_i) = \vec{u}[i]$ for $i \in \{1, \ldots, n\}$, Moreover, $Isol_{\vec{u},v,b}(t)$ evaluates to d_v. Applying the induction hypothesis now yields that t_i evaluates to $d_{\vec{u}[i]} \in D$ for each $i \in \{1, \ldots, n\}$. Since $t = f(t_1, \ldots, t_n) = f(\vec{u}[1], \ldots, \vec{u}[n])$ and v is interpreted as $h(\vec{v})$, this means that t evaluates to $h(d_{\vec{u}[1]}, \ldots, d_{\vec{u}[n]}) = d_v$, as claimed.

We prove the second part using an induction over the structure of a formula φ. Recalls that we fix $b = F$ for this part of the proof.

Base case In the base case, we distinguish between the two cases $\varphi = P(t_1, \dots, t_k)$ and $\varphi = \neg P(t_1, \dots, t_k)$.

- Let $\varphi = P(t_1, \dots, t_k)$ and assume that $\mathit{Isol}_{\vec{u},v,F}(\varphi)$ evaluates to false. By definition of $\mathit{Isol}_{\vec{u},v,b}$, this means that $\mathit{Isol}_{\vec{u},v,F}(t_i)$ evaluates to a non-$\bot$ value, say $d_i \in D$, for each $i \in \{1, \dots, k\}$. Moreover, $\mathit{Isol}_{\vec{u},v,F}(\varphi)$ evaluates to $P(\mathit{Isol}_{\vec{u},v,F}(t_1), \dots, \mathit{Isol}_{\vec{u},v,F}(t_k))$ and, hence, $P(d_1, \dots, d_k)$ evaluates to false. The first part of the proof now yields that t_i evaluates to d_i. Since $\varphi = P(t_1, \dots, t_k)$, this means that φ evaluates to false, as claimed.
- The case $\varphi = \neg P(t_1, \dots, t_k)$ is analogous to the case $\varphi = P(t_1, \dots, t_k)$ and therefore skipped.

Induction step In the induction step, we distinguish between the two cases $\varphi = \varphi_1 \vee \varphi_2$ and $\varphi = \varphi_1 \wedge \varphi_2$.

- Let $\varphi = \varphi_1 \vee \varphi_2$ and assume that $\mathit{Isol}_{\vec{u},v,F}(\varphi)$ evaluates to false. Thus, both $\mathit{Isol}_{\vec{u},v,F}(\varphi_1)$ and $\mathit{Isol}_{\vec{u},v,F}(\varphi_2)$ evaluate to false. Applying the induction hypothesis yields that both φ_1 and φ_2 evaluate to false. Thus, $\varphi = \varphi_1 \vee \varphi_2$ evaluates to false, as claimed.
- The case $\varphi = \varphi_1 \wedge \varphi_2$ is analogous to the case $\varphi = \varphi_1 \vee \varphi_2$ and therefore skipped. Q.E.D.

We can also show (again using structural induction) that when the isolation of the specification with respect to $b = F$ evaluates to false, then v is definitely not a correct output on $\vec{u}$.

Lemma 2.2. *Let $\forall \vec{x}.\ \psi(f, \vec{x})$ be a specification, $\vec{p} \in D^n$ an interpretation for $\vec{u}$, and $q \in D$ an interpretation for v such that there is some interpretation for $\vec{x}$ that makes the formula $\mathit{Isolate}_{\vec{u},v,F}(\psi(f, \vec{x}))$ evaluate to false. Then, there exists no function h satisfying the specification that maps $\vec{p}$ to q.*

Proof. Let h be a function that satisfies the specification and maps $\vec{p}$ to q. Then, $\psi(f, \vec{x})$ evaluates to true for every interpretation of $\vec{x}$. By Lemma 2.1, this means that $\mathit{Isolate}_{\vec{u},v,F}(\psi(f, \vec{x}))$ always evaluates to true or $\bot$ (it cannot evaluate to false because then φ would evaluate to false as well). However, this is a contradiction to the assumption that there exists an interpretation for $\vec{x}$ on which the formula $\mathit{Isolate}_{\vec{u},v,F}(\psi(f, \vec{x}))$ evaluates to false. Q.E.D.

We can now define single-point refutable specifications.

Definition 2.3 (Single-Point Refutable Specifications (SPR))**.** *A specification* $\forall \vec{x}.\ \psi(f,\ \vec{x}\ )$ *is said to be* single-point refutable *if the following holds. Let* $H : D^n \rightarrow D$ *be any interpretation for the function* f *that does not satisfy the specification (i.e., the specification does not hold under this interpretation for* f*). Then, there exists some input* $\vec{p}$ *that is an interpretation for* $\vec{u}$ *and an interpretation for* $\vec{x}$ *such that when* v *is interpreted to be* $H(\vec{u})$*, the isolated formula* $Isolate_{\vec{u},v,F}(\psi(f,\vec{x}))$ *evaluates to false.*

Intuitively, the above says that a specification is single-point refutable if whenever a hypothesis function H does not satisfy a specification, there is a single input $\vec{p}$ such that the specification evaluates to false independent of how the function maps inputs other than $\vec{p}$. More precisely, ψ evaluates to *false* for some interpretation of $\vec{x}$ only assuming that $f(\vec{p}) = H(\vec{p})$.

In fact, single-point refutable specifications are single-point definable, which we formalize below.

Lemma 2.3. *If a specification* $\forall \vec{x}.\ \psi(f,\vec{x})$ *is single-point refutable, then it is single-point definable.*

Proof. Let $\forall \vec{x}.\ \psi(f,\vec{x})$ be a single-point refutable specification, and assume that it is not single-point definable. Moreover, let $\mathcal{F}$ be the class of all functions that satisfy this specification. Then, there exists a function $h' : D^n \rightarrow D$ such that for every input $\vec{p} \in D^n$, there exists some function $h \in \mathcal{F}$ such that $h'(\vec{p}) = h(\vec{p})$, and yet h' does not satisfy the specification. By single-point refutability of the specification, there must be some input $\vec{p}$ such that when we interpret $v = h'(\vec{p})$, there is an interpretation of $\vec{x}$ such that $Isolate_{\vec{u},v,F}(\psi(f,\ \vec{x}\ ))$ evaluates to false. Let $h \in \mathcal{F}$ be some function that agrees with h' on $\vec{p}$. By Lemma 2.2, there is no function that satisfies the specification and that maps $\vec{u}$ to v, which contradicts the fact that h satisfies the specification. Q.E.D.

Let us illustrate the definition of single-point refutable specifications through an example and a non-example.

Example 2.3. Consider the following specifications in the first-order theory of arithmetic:

- The specification

$$\forall x, y.\ f(15,23) = 19 \wedge f(90,20) = 91 \wedge \ldots \wedge f(28,24) = 35 \tag{2.22}$$

  is single-point refutable. More generally, any set of input-output samples can be written as a conjunction of constraints that forms a single-point refutable specification.

- The specification

$$\forall x.\ f(0) = 0 \wedge f(x+1) = f(x) + 1 \tag{2.23}$$

  is *not* a single-point refutable specification, though it is single-point definable. Given a hypothesis function (e.g., $H(i) = 0$ for all i), the formula $f(x+1) = f(x) + 1$ evaluates to false, but this involves the definition of f on *two* inputs, and hence we cannot isolate a single input on which the function H is incorrect. (In evaluating the isolated transformation of the specification parameterized with $b = F$, at least one of $f(x+1)$ and $f(x)$ will evaluate to $\perp$ and, hence, the whole formula will never evaluate to false.)

When a specification $\forall \vec{x}.\ \psi(f, \vec{x})$ is single-point refutable, given an *expression* H for f that does not satisfy the specification, we can check satisfiability of the formula

$$\exists \vec{u}\, \exists v \exists \vec{x}.\ \Big(v = H(\vec{u}) \wedge \neg \mathit{Isolate}_{\vec{u},v,F}(\psi(H/f, \vec{x}))\Big). \tag{2.24}$$

Assuming the underlying quantifier-free theory has a decidable satisfiability problem and one can construct models, the valuation of $\vec{u}$ gives a *concrete* input $\vec{p}$, and Lemma 2.2 shows that H is definitely wrong on this input. This will form the basis of generating counterexample inputs in the synthesis framework that we present next.

2.3 A GENERAL SYNTHESIS FRAMEWORK BY LEARNING CLASSIFIERS

We now present our general framework for synthesizing functions over a first-order theory that uses machine-learning of classifiers. Our technique, as outlined in the introduction, is a *counterexample-guided inductive synthesis approach (CEGIS)*, and works most robustly for single-point refutable specifications.

Given a single-point refutable specification $\forall \vec{x}.\ \psi(f, \vec{x})$, the framework combines several simpler synthesizers and calls to SMT solvers to synthesize a function, as depicted in Figure 2.1. The solver globally maintains a finite set of expressions E, a finite set of predicates A (also called attributes), and a finite set S of multi-labeled samples, where each sample is of the form $(\vec{p}, Z)$ consisting of an input $\vec{p} \in D^n$ and a set $Z \subseteq E$ of expressions that are correct for $\vec{p}$ (such a sample means that the specification allows mapping $\vec{p}$ to $e(\vec{p})$, for any $e \in Z$, but not to $e'(\vec{p})$, for any $e' \in E \setminus Z$).

Phase 2.1. In every round, the classifier produces a hypothesis expression H for f. The process starts with a simple expression H, such as one that maps all inputs to a constant. We feed H in every round to a *counterexample input finder* module, which essentially is a

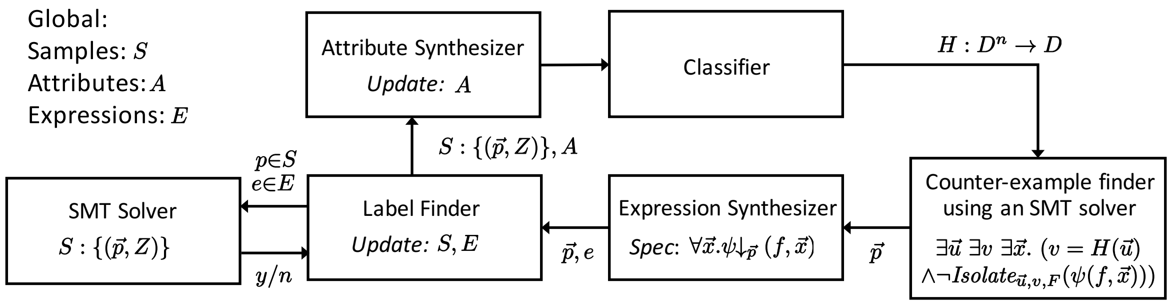

Global:
Samples: S
Attributes: A
Expressions: E
Attribute Synthesizer
Update: A
Classifier
$H : D^n \to D$
SMT Solver
$S : \{(\vec{p}, Z)\}$
$p \in S$
$e \in E$
y/n
Label Finder
Update: S, E
$S : \{(\vec{p}, Z)\}, A$
Expression Synthesizer
Spec: $\forall \vec{x}. \psi{\downarrow_{\vec{p}}} (f, \vec{x})$
$\vec{p}, e$
$\vec{p}$
Counter-example finder using an SMT solver
$\exists \vec{u}\ \exists v\ \exists \vec{x}.\ (v = H(\vec{u}) \wedge \neg Isolate_{\vec{u},v,F}(\psi(f, \vec{x})))$

Figure 2.1: The general synthesis framework based on learning classifiers

call to an SMT solver to check whether the formula

$$\exists \vec{u}\ \exists v\ \exists \vec{x}.\ (v = H(\vec{u}) \wedge \neg \mathit{Isolate}_{\vec{u},v,F}(\psi(f, \vec{x}))) \tag{2.25}$$

is satisfiable. Note that from the definition of the single-point refutable functions (see Definition 2.3), whenever H does not satisfy the specification, we are guaranteed that this formula is satisfiable, and the valuation of $\vec{u}$ in the satisfying model gives us an input $\vec{p}$ on which H is definitely wrong (see Lemma 2.2). If H satisfies the specification, the formula would be unsatisfiable (by Lemma 2.1) and we can terminate, reporting H as the synthesized expression.

Phase 2.2. The counterexample input $\vec{p}$ is then fed to an expression synthesizer whose goal is to find *some* correct expression that works for $\vec{p}$. We facilitate this by generating a *new specification for synthesis* that tailors the original specification to the particular input $\vec{p}$. This new specification is the formula

$$\psi{\downarrow_{\vec{p}}} (\widehat{f}, \vec{x}) := \mathit{Isolate}_{\vec{u},v,T}(\psi(f, \vec{x}))[\vec{p}/\vec{u}, \widehat{f}(\vec{p})/v]. \tag{2.26}$$

Intuitively, the above specification asks for a function $\widehat{f}$ that *"works"* for the input $\vec{p}$ (i.e., there exists a function g satisfying the specification such that $g(\vec{p}) = \widehat{f}(\vec{p})$). We do this by first constructing the formula that isolates the specification to $\vec{u}$ with output v and demand that the specification evaluates to true; then, we substitute $\vec{p}$ for $\vec{u}$ and a new function symbol $\widehat{f}$ evaluated on $\vec{p}$ for v. Any expression synthesized for $\widehat{f}$ in this synthesis problem maps $\vec{p}$ to a value that is consistent with the original specification, which we formalize next.

Lemma 2.4. *Let $\psi(f, \vec{x})$ be a single-point refutable specification and $\mathcal{F}$ the class of all functions satisfying $\psi(f, \vec{x})$. Moreover, let $\vec{p} \in D^n$ be an input to f, and let e be a solution to* *the synthesis problem with specification $\psi{\downarrow_{\vec{p}}}\,(\widehat{f}, \vec{x})$. Then, there exists a function $g \in \mathcal{F}$ such that $g(\vec{p}) = e(\vec{p})$.*

Proof. Using similar arguments as in the proof of Lemma 2.1, we can show that if $\mathit{Isol}_{\vec{u},v,T}(\varphi)$ (and, hence, $\mathit{Isolate}_{\vec{u},v,T}(\varphi)$) evaluates to true on some valuation of the free variables $\vec{x}$, $\vec{u}$, and v, then the formula φ also evaluates to true on the valuation for $\vec{x}$. Thus, by substituting $\vec{p}$ for $\vec{u}$ in $\mathit{Isolate}_{\vec{u},v,T}(\psi(f, \vec{x}))$, we know that if $\mathit{Isolate}_{\vec{u},v,T}(\psi(f, \vec{x}))[\vec{p}/\vec{u}]$ evaluates to true, then ψ evaluates to true if f maps $\vec{p}$ to v. Moreover, by substituting $\widehat{f}(\vec{p})$ for v, we obtain the specification $\psi{\downarrow_{\vec{p}}}\,(\widehat{f}, \vec{x})$, which constraints $\widehat{f}$ such that $\psi(f, \vec{x})$ to evaluates to true if $f(\vec{p}) = \widehat{f}(\vec{p})$. Thus, any solution e to the synthesis problem with specification $\psi{\downarrow_{\vec{p}}}\,(\widehat{f}, \vec{x})$ guarantees that $\psi(f, \vec{x})$ evaluates to true if $f(\vec{p}) = e(\vec{p})$.

Now, let $h \in \mathcal{F}$ some function satisfying ψ. Since ψ is single-point refutable (and, hence, also single-point definable), the function

$$g(\vec{x}) = \begin{cases} e(\vec{p}) & \text{if } \vec{x} = \vec{p}\text{; and} \\ h(\vec{x}) & \text{otherwise} \end{cases} \tag{2.27}$$

also satisfies the specification. Thus, g is a function satisfying $g \in \mathcal{F}$ and $g(\vec{p}) = e(\vec{p})$. Q.E.D.

We emphasize that this new synthesis problem is simpler than the original problem (since it only requires to synthesize an expression for a single input) and that we can use any (existing) expression synthesizer to solve it. One important challenge in this context clearly is to synthesize an expression that is “good” (or general) in the sense that it works for all (or at least many) inputs in the region $\vec{p}$ belongs to. One possible way to achieve this is to apply Occam’s razor principle and synthesize an expression that is as simple as possible with respect to some total order of the expressions (e.g., the length of the expression or the maximal nesting of sub-expressions). Another way is to define a distance metric on inputs and synthesize an expression that does not only work for the input $\vec{p}$ but also for other known inputs in the sample S whose distance to $\vec{p}$ is small. We describe these two approaches and various further heuristics in more detail in Section 2.5.1, where we present our implementation of a synthesizer for linear integer arithmetic expressions.

Phase 2.3. Once we synthesize an expression e that works for $\vec{p}$, we feed it to the next phase, which adds e to the set of all expressions E (if e is new) and adds $\vec{p}$ to the set of samples. It then proceeds to find the set of *all* expressions in E that work for all the inputs in the samples, and computes the new set of samples. In order to do this, we take every input $\vec{r}$ that previously existed, and ask whether e works for $\vec{r}$, and if it does, add e to the set of labels for $\vec{r}$. Also, we take the new input $\vec{p}$ and every expression $e' \in E$, and check whether e' works for $\vec{p}$.

To compute this labeling information, we need to be able to check, in general, whether an expression e' works for an input $\vec{r}$. We can do this using a call to an SMT solver that checks whether the formula $\forall \vec{x}.\ \psi{\downarrow_{\vec{r}}} (e'(\vec{r})/\widehat{f}(\vec{r}),\ \vec{x})$ is valid.

Phase 2.4. We now have a set of samples, where each sample consists of an input and a set of expressions that work for that input. This is when we look upon the synthesis problem as a *classification* problem— that of mapping every input in the domain to an expression that generalizes the sample (i.e., that maps every input in the sample to *some* expression that it is associated with it). In order to do this, we need to split the input domain into *regions* defined by a set of predicates A. We hence need an adequate set of predicates that can define enough regions that can separate the inputs that need to be separated.

Let S be a set of samples and let A be a set of predicates. Two samples $(\vec{j}, E_1)$ and $(\vec{j'}, E_2)$ are said to be *inseparable* if for every predicate $p \in A$, $p(\vec{j}) \equiv p(\vec{j'})$. The set of predicates A is said to be *adequate* for a sample S if any set of inseparable inputs in the sample has a common label as a classification. In other words, if every subset $T \subseteq S$, say $T = \{(\vec{i_1}, E_1), (\vec{i_2}, E_2), \ldots (\vec{i_t}, E_t)\}$, where every pair of inputs in T is inseparable, then $\bigcap_{i=1}^{t} E_i \neq \emptyset$. We require the *attribute synthesizer* to synthesize an adequate set of predicates A, given the set of samples.

Intuitively, if T is a set of pairwise inseparable points with respect to a set of predicates P, then no classifier based on these predicates can separate them, and hence they all need to be classified using the same label; this is possible only if the set of points have a common expression label.

Phase 2.5. Finally, we give the samples and the predicates to a classification learner, which divides the set of inputs into regions, and maps each region to a single expression such that the mapping is consistent with the sample. A region is a *conjunction* of predicates and the set of points in the region is the set of all inputs that satisfy all these predicates. The classification is consistent with the set of samples if for every sample $(\vec{r}, Z) \in S$, the classifier maps $\vec{r}$ to a label in Z. (In Section 2.4, we present a general learning algorithm based on decision trees that learns such a classifier from a set of multi-labeled samples, and which biases the classifier towards small trees.)

The classification synthesized is then converted to an expression in the logic (this will involve nested *ite* expressions using predicates to define the regions and expressions at leaves to define the function). The synthesized function is fed back to the counterexample input finder, as in Phase 1, and the process continues until we manage to synthesize a function that meets the specification.

2.4 MULTI-LABEL DECISION TREE CLASSIFIERS

In this section, we sketch a decision tree learning algorithm for a special case of the so-called multi-label learning problem, which is the problem of learning a predictive model (i.e., a classifier) from samples that are associated with multiple labels. (We refer to standard textbooks on machine learning, e.g., [59], for more information on decision tree learning.) For the purpose of learning the classifier, we assume samples to be vectors of the Boolean values $\mathbb{B} = \{F, T\}$ (these encode the values of the various attributes on the counterexample input returned). The more general case that datapoints also contain rational numbers can be handled in a straightforward manner as in Quinlan's C 5.0 algorithm [79, 79].

To make the learning problem precise, let us fix a finite set $L = \{\lambda_1, \dots, \lambda_k\}$ of labels with $k \geq 2$, and let $\vec{x}_1, \dots, \vec{x}_m \in \mathbb{B}^n$ be m individual inputs (in the following also called *datapoints*). The task we are going to solve, which we call *disjoint multi-label learning problem* (cf. [80]), is defined as follows.

Definition 2.4 (Disjoint Multi-Label Learning Problem)**.** *Given a finite training set $S = \{(\vec{x}_1, Y_1), \dots, (\vec{x}_m, Y_m)\}$ where $Y_i \subseteq L$ and $Y_i \neq \emptyset$ for every $i \in \{1, \dots, m\}$, the* disjoint multi-label learning problem *is to find a decision tree classifier $h \colon \mathbb{B}^n \to L$ such that $h(\vec{x}) \in Y$ for all $(\vec{x}, Y) \in S$.*

Note that this learning problem is a special case of the multi-label learning problem studied in machine learning literature, which asks for a classifier that predicts all labels that are associated with a datapoint. Moreover, it is important to emphasize that we require our decision tree classifier to be consistent with the training set (i.e., it is not allowed to misclassify datapoints in the training set), in contrast to classical machine learning settings where classifier are allowed to make (small) errors.

We use a straightforward modification of Quinlan's C 5.0 algorithm [79, 79] to solve the disjoint multi-label learning problem. This modification, sketched in pseudo code as Algorithm 2.1, is a recursive algorithm that constructs a decision tree top-down. More precisely, given a training set S, the algorithm heuristically selects an attribute $i \in \{1, \dots, n\}$ and splits the set into two disjoint, nonempty subsets $S_i = \{(\vec{x}, Y) \in S \mid \vec{x}[i] = T\}$ and $S_{\neg i} = \{(\vec{x}, Y) \in S \mid \vec{x}[i] = F\}$ (we explain shortly how the attribute i is chosen). Then the algorithm recurses on the two subsets, whereby it no longer considers the attribute i. Once the algorithm arrives at a set S' in which all datapoints share at least one common label (i.e., there exists a $\lambda \in L$ such that $\lambda \in Y$ for all $(\vec{x}, Y) \in S'$), it selects a common label

Algorithm 2.1: Multi-label decision tree learning algorithm

```
Input: A finite set $S \subseteq \mathbb{B}^n \times 2^L$ of datapoints.
1  return DecTree ($S$, $\{1, \dots, n\}$).
2  Procedure DecTree (Set of datapoints $S$, Attributes $A$):
3      Create a root node $r$.
4      if if all datapoints in $S$ have a label in common (i.e., there exists a label $\lambda$ such that
         $\lambda \in Y$ for each $(\vec{x}, Y) \in S$) then
5          Select a common label $\lambda$ and return the single-node tree $r$ with label $\lambda$.
6      else
7          Select an attribute $i \in A$ that (heuristically) best splits the sample $S$.
8          Split $S$ into $S_i = \{(\vec{x}, Y) \in S \mid \vec{x}[i] = T\}$ and $S_{\neg i} = \{(\vec{x}, Y) \in S \mid \vec{x}[i] = F\}$.
9          Label $r$ with attribute $i$ and return the tree with root node $r$, left subtree DecTree
             ($S_i$, $A \setminus \{i\}$), and right subtree DecTree ($S_{\neg i}$, $A \setminus \{i\}$).
10     end
11 end
```

λ (arbitrarily), constructs a single-node tree that is labeled with λ, and returns from the recursion. However, it might happen during construction that a set of datapoints does not have a common label and cannot be split by any (available) attribute. In this case, it returns an error, as the set of attributes is not adequate (which we make sure does not happen in our framework by synthesizing new attributes whenever necessary).

The following theorem states the correctness of Algorithm 2.1 (i.e., that the algorithm indeed produces a solution to the disjoint multi-label learning problem), which is a result independent of the exact way an attribute is chosen.

Theorem 2.1. *Let $L = \{\lambda_1, \dots, \lambda_k\}$ be a set of labels with $k \geq 2$ and $S = \{(\vec{x}_1, Y_1), \dots, (\vec{x}_m, Y_m)\} \subseteq \mathbb{B}^n \times 2^L$ a finite training set where $Y_i \neq \emptyset$ for every $i \in \{1, \dots, n\}$. Moreover, assume that each two distinct datapoints $(\vec{x}, Y), (\vec{x}', Y') \in S$ can be separated by some attribute $i \in \{1, \dots, n\}$ (i.e., $\vec{x}[i] \neq \vec{x}'[i]$). Then, Algorithm 2.1 terminates and returns a decision tree classifier $h \colon \mathbb{B}^n \to L$ that satisfies $h(\vec{x}) \in Y$ for each $(\vec{x}, Y) \in S$.*

Proof. We show Theorem 2.1 by induction over the construction of the tree.

Base case Assume that the function DecTree is called with a set S of datapoint that share a common label (i.e., that there exists a label λ such that $\lambda \in Y$ for each $(\vec{x}, Y) \in S$). Then, the condition in Line 4 evaluates to true and the algorithm returns a decision tree h consisting of a single node that is labeled with a label shared by all datapoints of S (see Line 5). Thus, h satisfies $h(\vec{x}) \in Y$ for each $(\vec{x}, Y) \in S$.

Induction step Assume that the function DecTree is called with a set S of datapoints that do not share a common label and a set A of available attributes. Then, the condition in Line 4 is false, and the algorithm proceeds with Lines 7 to 9.

First, we observe that $A \neq \emptyset$: since we assume that all datapoints can be separated by an attribute, $A = \emptyset$ implies that S is a singleton and, hence, the condition in Line 4 would be true. Thus, the algorithm can pick an attribute $i \in A$ (Line 7), partition S into two subsamples $S_i, S_{\neg i}$ (Line 8), and recursively constructs the decision trees $h_i = \texttt{DecTree}(S_i, A \setminus \{i\})$ and $h_{\neg i} = \texttt{DecTree}(S_{\neg i}, A \setminus \{i\})$ (Line 9). Finally, it returns the decision tree h with root node r whose subtrees are h_i and $h_{\neg i}$, respectively.

Since the root of h is labeled with attribute i, one can write h as

$$h(\vec{x}) = \begin{cases} h_i(\vec{x}) & \text{if } \vec{x}[i] = T\text{; and} \\ h_{\neg i}(\vec{x}) & \text{if } \vec{x}[i] = F. \end{cases} \tag{2.28}$$

Moreover, applying the induction hypothesis yields that h_i satisfies $h_i(\vec{x}) \in Y$ for each $(\vec{x}, Y) \in S_i$ and $h_{\neg i}(\vec{x}) \in Y$ for each $(\vec{x}, Y) \in S_{\neg i}$. Thus, if $\vec{x}[i] = T$ for some $(\vec{x}, Y) \in S$, then $(\vec{x}, Y) \in S_i$ and, hence, $h(\vec{x}) = h_i(\vec{x}) \in Y$; on the other hand, if $\vec{x}[i] = F$ for some $(\vec{x}, Y) \in S_{\neg i}$, then $(\vec{x}, Y) \in S_{\neg i}$ and, hence, $h(\vec{x}) = h_{\neg i}(\vec{x}) \in Y$. Q.E.D.

The selection of a “good” attribute to split a a set of datapoints lies at the heart of the decision tree learner as it determines the size of the resulting tree and, hence, how well the tree generalizes the training data. This problem is best understood in the simpler single-label setting in which datapoints are labeled with one out of two possible labels, say 0 or 1. To obtain a small decision tree, the learning algorithm should split samples such that the resulting subsamples are as pure as possible (i.e., one subsample contains as many datapoints as possible labeled with 0, whereas the other subsample contains as many datapoints as possible labeled with 1). This way, the learner will quickly arrive at samples that contain a single label and, hence, produce small a tree.

The quality of a split can be formalized by the notion of a *measure*, which, roughly, is a measure μ mapping pairs of sets of datapoints to a set R that is equipped with a total order $\preceq$ over elements of R (usually, $R = \mathbb{R}_{\geq 0}$ and $\preceq$ is the natural order over $\mathbb{R}$). Given a set S to split, the learning algorithm first constructs subsets S_i and $S_{\neg i}$ for each available attribute i and evaluates each such candidate split by computing $\mu(S_i, S_{\neg i})$. It then chooses a split that has the least value.

In the single-label setting, information theoretic measures, such as *information gain* (based on *Shannon entropy*) and *Gini*, have proven to produce successful classifiers [81]. In the case of multi-label classifiers, however, finding a good measure is still a matter of ongoing research (e.g., see [82] for an overview). Both the classical entropy and Gini measures can be adapted to the multi-label case in a straightforward way by treating datapoints with multiple labels as multiple identical datapoints with a single label. More precisely, the main idea is to replace each multi-labeled datapoint $(\vec{x}, \{\lambda_1, \ldots, \lambda_k\})$ with the datapoints $(\vec{x}, \lambda_1), \ldots, (\vec{x}, \lambda_k)$ and proceeds as in classical decision tree learning.

We now briefly sketch these modifications, including a modification described by [83]. In all cases, we fix $R = \mathbb{R}$ and let $\preceq$ be the natural order over $\mathbb{R}$.

Entropy Intuitively, entropy is a measure for the amount of "information" contained in a sample; the higher the entropy, the higher the randomness of the sample. Formally, one defines the entropy of a sample S with multiple labels by

$$e(S) = -\sum_{\lambda \in L} p_\lambda \cdot \log_2 p_\lambda, \tag{2.29}$$

where p_λ is the relative frequency of the label λ defined by

$$p_\lambda = \frac{|\{(\vec{x}, Y) \in S \mid \lambda \in Y\}|}{\sum_{(\vec{x}, Y) \in S} |Y|}. \tag{2.30}$$

The corresponding measure $\mu_e(S_1, S_2)$ is the weighted average of $e(S_1)$ and $e(S_2)$.

Gini One can think of Gini as the probability of making a classification error if the whole sample is uniformly labeled with a randomly chosen label. Formally, for a sample S with multiple labels, one defines Gini by

$$g(S) = \sum_{\lambda \neq \lambda' \in L} p_\lambda \cdot p_{\lambda'}, \tag{2.31}$$

where p_λ is again the relative frequency of the label λ (see above). The Gini measure $\mu_g(S_1, S_2)$ is the weighted average of $g(S_1)$ and $g(S_2)$.

pq-entropy The modification of entropy by [83] accounts for multiple labels by considering for each label the probability of being labeled with λ (i.e., the relative frequency of λ) as well as probability of not being labeled with λ. More precisely, for a sample S with multiple labels, Clare and King define

$$pq\text{-}e = -\sum_{\lambda \in L} p_\lambda \cdot \log_2 p_\lambda + q_\lambda \cdot \log_2 q_\lambda, \tag{2.32}$$

where p_λ is the relative frequency of the label λ and $q_\lambda = 1 - p_\lambda$. As measure $\mu_{pq\text{-}e}(S_1, S_2)$, Clare and King use the weighted average of $pq\text{-}e(S_1)$ and $pq\text{-}e(S_2)$.

However, all of these approaches share the disadvantage that the association of datapoints to sets of labels is lost. As a consequence, measures can be high even if all datapoints share a common label; for instance, such a situation occurs for $S = \{(\vec{x}_1, Y_1), \ldots, (\vec{x}_m, Y_m)\}$ with $\{\lambda_1, \ldots, \lambda_\ell\} \subseteq Y_i$ for every $i \in \{1, \ldots, m\}$. Therefore, one would ideally like to have a measure that maps to 0 if all datapoints in a set share a common label and to a value strictly greater than 0 if this is not the case. We now present a measure, based on the combinatorial problem of finding minimal hitting sets, that possesses this property. To the best of our knowledge, this measure is a novel contribution and has not been studied in the literature.

For a set S of datapoints, a set $H \subseteq L$ is a *hitting set* if $H \cap Y \neq \emptyset$ for each $(\vec{x}, Y) \in S$. Moreover, we define the measure $hs(S) = \min_{\text{hitting set } H} |H| - 1$ (i.e., the cardinality of a smallest hitting set reduced by 1). As desired, we obtain $hs(S) = 0$ if all datapoints in S share a common label and $hs(S) > 0$ if this is not the case. When evaluating candidate splits, we would prefer to minimize the number of labels needed to label the datapoints in the subsets; however, if two splits agree on this number, we would like to minimize the total number of labels required. Consequently, we propose $R = \mathbb{N} \times \mathbb{N}$ with $(n, m) \preceq (n', m')$ if and only if $n < n'$ or $n = n' \wedge m \leq m'$, and as measures

$$\mu_{hs}(S_1, S_2) = \Big(\max\{hs(S_1), hs(S_2)\}, hs(S_1) + hs(S_2)\Big). \tag{2.33}$$

Unfortunately, computing $hs(S)$ is computationally hard. Therefore, we implemented a standard greedy algorithm (the dual of the standard greedy set cover algorithm [84]), which runs in time polynomial in the size of the sample and whose solution is at most logarithmically larger than the optimal solution.

2.5 A SYNTHESIS ENGINE FOR LINEAR INTEGER ARITHMETIC

We now describe an instantiation of our framework (described in Section 2.3) for synthesizing functions expressible in linear integer arithmetic against quantified linear integer arithmetic specifications.

The counterexample input finder (Phase 1) and the computing of labels for counterexample inputs (Phase 3) are implemented straightforwardly using an SMT solver (note that the respective formulas will be in quantifier-free linear integer arithmetic). The *Isolate*() function works over a domain $D \cup \{\bot\}$; we can implement this by choosing a particular element $\widehat{c}$ in the domain and modeling every term using a *pair* of elements, one that denotes the original term and the second that denotes whether the term is $\perp$ or not, depending on whether it is equal to $\widehat{c}$. It is easy to transform the formula now to one that is on the original domain D (which in our case integers) itself.

2.5.1 Expression Synthesizer

Given an input $\vec{p}$, the expression synthesizer has to find an expression that works for $\vec{p}$. Our implementation deviates slightly from the general framework.

In the first phase, it checks whether one of the existing expressions in the global set E already works for $\vec{p}$. This is done by calling the label finder (as in Phase 3). If none of the expressions in E work for $\vec{p}$, the expression synthesizer proceeds to the second phase, where it generates a new synthesis problem with specification $\forall \vec{x}.\ \psi{\downarrow_{\vec{p}}}(\hat{f}, \vec{x})$ according to Phase 2 of Section 2.3, whose solutions are expressions that work for $\vec{p}$. It solves this synthesis problem using a simple CEGIS-style algorithm, which we sketch next.

Let $\forall \vec{x}.\ \psi(f, \vec{x})$ be a specification with a function symbol $f \colon \mathbb{Z}^n \to Z$, which is to be synthesized, and universally quantified variables $\vec{x} = (x_1, \dots, x_m)$. Our algorithm synthesizes affine expressions of the form $(\sum_{i=1}^{n} a_i \cdot y_i) + b$ where $y_1, \dots, y_n$ are integer variables, $a_i \in \mathbb{Z}$ for $i \in \{1, \dots, n\}$, and $b \in \mathbb{Z}$. The algorithm consists of two components, a *synthesizer* and a *verifier*, which implement the CEGIS principle in a similar but simpler manner as our general framework. Roughly speaking, the synthesizer maintains an (initially empty) set $V \subseteq \mathbb{Z}^m$ of valuations of the variables $\vec{x}$ and constructs an expression H for the function f that satisfies ψ at least for each valuation in V (as opposed to all possible valuations). Then, it hands this expression over to the verifier. The task of the verifier is to check whether H satisfies the specification. If this is the case, the algorithm has identified a correct expression, returns it, and terminates. If this not the case, the verifier extracts a particular valuation of the variables $\vec{x}$ for which the specification is violated and hands it over to the synthesizer. The synthesizer adds this valuation to V, and the algorithm iterates. The synthesizer and verifier are implemented as follows.

Synthesizer The synthesizer maintains a finite set $V \subseteq \mathbb{Z}^m$ of valuations of the universally quantified variables $\vec{x}$ and constructs expressions for the synthesis function f that satisfies ψ at least on all valuations in V. To this end, the synthesizer first constructs a template expression $t(\vec{a}, b, \vec{y})$ of the form described above, but where $\vec{a} = (a_1, \dots, a_n)$, and b are now variables (note that this expression is not linear due to the terms $a_i \cdot y_i$). Then, it constructs the formula

$$\varphi(\vec{a}, b) \coloneqq \bigwedge_{\vec{v} \in V} \psi(t/f, \vec{v}/\vec{x}). \tag{2.34}$$

Note that φ is a formula in linear integer arithmetic since all occurrences of variables y_i have been replaced with integers values. Finally, the algorithm uses an SMT solver to obtain valuations of $\vec{a}$ and b that satisfy φ; note that φ is guaranteed to be satisfiable since we use the synthesizer in a special setting, namely to synthesize expressions for a single-point definable specification. The synthesizer substitutes the satisfying assignment for $\vec{a}$ and b in the template t and returns the resulting expression H.

Verifier Given an expression H conjectured by the synthesizer, the verifier has to check whether

$$\varphi \coloneqq \psi(H/f, \vec{x}) \tag{2.35}$$

is valid. To this end, the verifier turns this validation problem into a satisfiability problem by querying an SMT solver whether $\neg\varphi$ is satisfiable. If $\neg\varphi$ is satisfiable, then the verifier extracts a satisfying assignment $\vec{v}$ for the universal quantified variables and returns $\vec{v}$ to the synthesizer. If $\neg\varphi$ is unsatisfiable, an expression satisfying the specification has been found and the synthesizing algorithm returns it.

2.5.1.1 Further Heuristics

Our implementation for synthesizing expressions for linear arithmetic constraints has several other heuristics that we briefly describe.

First, some technical aspects of the counterexample finder and the label finder are implemented a bit differently than explained above. We use array theories and uninterpreted functions to extract the counterexample point from the specification and the hypothesis (instead of using the ISOLATE transformer), and to check if a point works for an expression.

The counterexample finder prioritizes data-points that have a single classification, and returns multiple counterexamples in each round, to facilitate better learning.

We also maintain an initial set of enumerated expressions, and dovetail through them lazily before invoking the expression synthesizer. These initial expressions has coefficients between -1 and 1 for all the variables. If none of these expression work for the counterexample point, the expression synthesizer is invoked.

There are also phases in our algorithm where, when examining an enumerated expression, we would ask a constraint solver whether this expression would work for any input (not necessarily the counterexample input) and if possible, add the new point and the expression to the sample. Further, when we do find a unique expression that works for the counterexample, we ask the constraint solver for *more* points for which this expression would work, and add them as extra samples. When multiple expressions work for a point, we strive to find another point for which only one of them works (in order to avoid considering spurious expressions) and add them to the sample.

The expression synthesizer also uses a heuristic motivated by a geometric intuition. We would expect the correct expression for a point to work also on points neighboring it (unless it lies close to the boundary of a piece-wise region). In order to synthesize a n-dimension expression, we need at least $(n+1)$ points that lie on that plane. If $(y_1, y_2, \ldots y_n)$ is a point, then it is highly likely that the "correct" expression for the point would also work for its immediate neighboring points in each dimension, namely $(y_1 + 1, y_2 \ldots y_n)$, $(y_1, y_2 + 1 \ldots y_n)$, $\ldots$ $(y_1, y_2, \ldots, y_n + 1)$. We constrain the SMT solver to synthesize a n-dimensional expression with integer coefficients that works for all these $(n+1)$ points. If no such expression exists, then we resort to synthesizing an expression only for the counterexample.

2.5.2 Expression Synthesizer for Unique Specifications

In an earlier work [85], we designed an expression synthesizer module for the restricted case of a unique specification where the specification has only one unique solution. As the function to synthesize is unique, it is not hard to see that each counterexample can only be labeled with one expression, instead of multiple expressions in the general case. In addition to synthesizing expressions, the synthesizer module also labels the counterexamples with the expression that works for it. The framework globally maintains a set P consisting of all the counterexamples found across rounds, where each counterexample is an input-output pair $(\vec{p}, v)$ such that $f(\vec{p}) = v$ according to the specification. Once completed the synthesizer updates the set E of expressions, and the set S of samples, where each sample is of the form $(\vec{p}, \{e\})$, such that $\vec{p} \in P$, and the expression e is correct for $\vec{p}$ according to the specification.

To solve this problem we used computational geometry. Assuming the function to synthesize f is of arity n, then the problem of finding the expressions that covers all the counterexamples can be viewed as the problem finding planes in a $(n+1)$-dimensional space where the space describes the input-output behavior of the function f. The synthesizer learns from all previously found counterexamples, where each counterexample is an input-output pair $(\vec{p}, v)$, and can be viewed as a *point* in this $(n+1)$-dimensional space. We would essentially like to find a small set of planes, that include all the given points in this space. We solve this problem using a greedy algorithm that uses geometric techniques to determine coplanarity between points in this $(n+1)$-dimensional space.

Algorithm 2.2: Algorithm for constructing planes that cover the input points

```
Function Construct-Plane(P, k):
    Select a random point $pt = (\vec{p}, v) \in P$ ;
    $C$ := set of $2n$ points closest to $pt$ ;
    $Y$ := collection of all subsets of $(n+1)$ points in $C$ ;
    foreach subset $Z$ in $Y$ do
        if the set of points in $(Z \cup pt)$ are coplanar then
            $pln$ := find_plane $(Z \cup pt)$ ;
            $Sel$ := set of points in $P$ that lie on plane $pln$ ;
            if $|Sel| > \lceil |P|/k \rceil$ then
                label the points in $Sel$ as $pln$ ;
                Return $\{ (pt, \{pln\}) \mid pt \in Sel \}$ ;
            end
        end
    end
end
```

Let us assume that the number of points in P is large, and that we want to find k planes, where k is a small constant, that cover all the points. We try to find a greedy strategy to find a small number of k planes such that every point is covered by one plane. We start with a small budget of k and increase k when it does not suffice. Note that if k planes are sufficient to cover *all* points, then there must be at least $\lceil |P|/k \rceil$ points that are covered by a single plane. Hence our strategy is to find a plane in a greedy manner that covers at least these many points. Once we find such a plane, we can remove all the points that are covered by the plane, and recurse decrementing k.

Note that in a $(n+1)$-dimensional space, one can always construct a plane that passes through any $(n+1)$ points. Hence, our strategy is to choose sets of $(n+2)$ points and check if they are coplanar (and then check if they cover enough points). Since we are synthesizing a piece-wise functions, it is likely that the expressions are defined over a local region, and hence we would like to choose the $(n+2)$ points such that they are *close* to each other. Our algorithm `Construct-Plane`, searches for a plane by (a) choosing a random point pt and taking the closest $2n$ points next to pt, by computing the distance of all points to pt, sorting them, and picking the closest $2n$ points and (b) choosing *every* combination of $(n+2)$ points from this set and checking it for coplanarity.

Coplanarity of such $(n+2)$ points in the $(n+1)$-dimensional space, can be verified by checking the value of the following determinant:

$$\begin{vmatrix} p_1^1 & p_1^2 & \dots & p_1^n & v_1 & 1 \\ p_2^1 & p_2^2 & \dots & p_2^n & v_2 & 1 \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ p_{n+2}^1 & p_{n+2}^2 & \dots & p_{n+2}^n & v_{n+2} & 1 \end{vmatrix} = 0 \tag{2.36}$$

This is a standard result in computational geometry [86]. The plane defined by these $(n+2)$ points can be constructed by solving for the coefficients c_i in the set of equations, where the j-th equation is $\Sigma_{i=1}^{n} c_i \, p_j^i + c_{n+1} \, v_j = c_{n+2}$, and we substitute the p_j's and v_j with the j-th point. The above two require numerical solvers and can be achieved using software like MatLab or Octave.

If the above process discovers k planes that cover all points in the sample, then we are done. If not, we are either left with too few points ($< n+2$) or too many points and have run out of the budget of planes. In the former case, we ignore these points and proceed to the later phases. Once the classifier has proposed a hypothesis, we then add these points back as special points on which the answers are fixed using appropriate constants. In the latter case, we increase our budget k, and continue.

There are several parameters that can be tuned for performance, including (a) how many points the teacher returns in each round, (b) the number of points in the window from which we select points to find planes, (c) the threshold of coverage for selecting a plane, etc. These parameters can be tweaked for better performance on the application at hand.

2.5.3 Predicate Synthesizer

Since the decision tree learning algorithm (which is our classifier) copes extremely well with a large number of attributes, we do not spend time in generating a small set of predicates. We build an *enumerative* predicate synthesizer that simply enumerates and adds predicates until it obtains an adequate set.

More precisely, the predicate synthesizer constructs a set A_q of attributes for increasing values of $q \in \mathbb{N}$. The set A_q contains all predicates of the form $\sum_{i=1}^{n} a_i \cdot y_i \leq b$, where y_i are variables corresponding to the function arguments of the function f that is to be synthesized, $a_i \in \mathbb{Z}$ such that each $\Sigma_{i=1}^{n} |a_i| \leq q$, and $|b| \leq q^2$. If A_q is already adequate for S (which can be checked by recursively splitting the sample with respect to each predicate in A_q and checking if all samples at each leaf has a common label), we stop, else we increase the parameter q by one and iterate. Note that the predicate synthesizer is guaranteed to find an adequate set for any sample. The reason for this is that one can separate each input $\vec{p}$ into its own subsample (assuming each individual variable is also an attribute) provided q is large enough.

2.5.4 Classifier Learner

We use the decision tree learner described in Section 2.4 to learn a decision tree classifier over the samples S and the predicates A. In a preparatory step, the classification learner transforms each input $\vec{p} \in S$ to a Boolean vector $\vec{b}_{\vec{p}}$: given $\vec{p}$ and predicates $A = \{p_1, \dots, p_m\}$, it constructs the Boolean vector $\vec{b}_{\vec{p}} = (p_1(\vec{p}), \dots, p_m(\vec{p})) \in \mathbb{B}^m$. It collects all transformed inputs in a new sample S', where the label of $\vec{b}_{\vec{p}}$ is the classification of $\vec{p}$. (Also here, one would clearly construct S' incrementally, growing it in each iteration.)

Once the set S' has been created, we run the decision tree learner on S'. The result is a tree τ, say with root node v_r, whose inner nodes are labeled with predicates from A and whose leafs are labeled with expression from E. The formula in linear integer arithmetic corresponding to τ is the nested if-then-else expression $\textit{to-ite}(v_r)$, where $\textit{to-ite}(v)$ for a tree node v is recursively defined by

- if v is a leaf node labeled with expression e, then $\textit{to-ite}(v) := e$; and
- if v is an inner node labeled with predicate p and children v_1 and v_2, then $\textit{to-ite}(v) := \textit{ite}(p, \textit{to-ite}(v_1), \textit{to-ite}(v_2))$.

The classification learner finally returns $\textit{to-ite}(v_r)$.

2.6 EVALUATION

We implemented the framework described in Section 2.5 for specifications written in the SyGuS format [1, 87]. The implementation is about 5K lines in C++ with API calls to the Z3 SMT solver [88].

We evaluated our tool parameterized using the different measures in Section 2.4 against 44 benchmarks. These benchmarks are predominantly from the 2014–2016 SyGuS competitions [1, 51, 64]. Additionally, there is an example from [63] for deobfuscating C code using bitwise operations on integers (we query this code 30 times on random inputs, record its output and create an input-output specification, `Jha_Obs`, from it). The synthesis specification `max3Univ` reformulates the specification for `max3` using universal quantification, as

$$\begin{aligned}&\forall x, r_1, r_2, y_1, y_2, y_3.\ (r_1 < 0 \land r_2 < 0) \Rightarrow \\ &((y_1{=}x \land y_2{=}x{+}r_1 \land y_3{=}x{+}r_2) \lor (y_1{=}x{+}r_1 \land y_2{=}x \land y_3{=}x{+}r_2) \lor (y_1{=}x{+}r_1 \land y_2{=}x{+}r_2 \land y_3{=}x)) \\ &\qquad \Rightarrow \mathit{max3}(y_1, y_2, y_3) = x. \quad (2.37)\end{aligned}$$

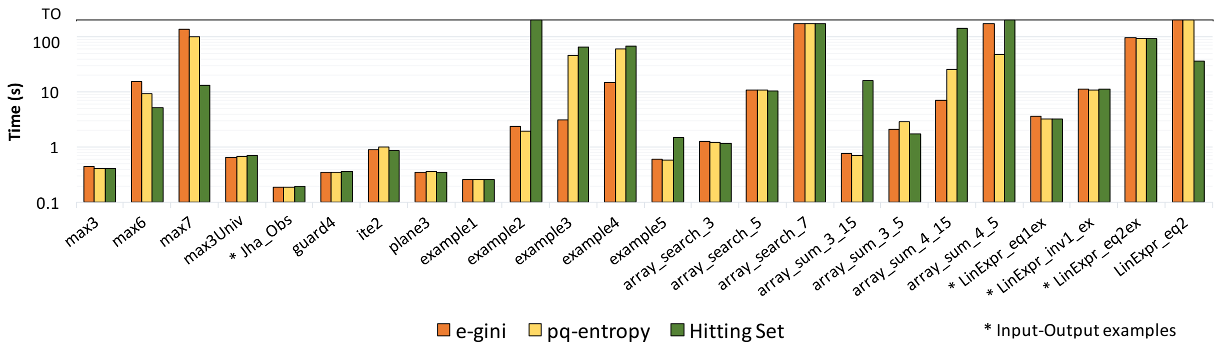


Figure 2.2: Experimental results

All experiments were performed on a system with an Intel Core i7-4770HQ 2.20 GHz CPU and 4 GB RAM running 64-bit Ubuntu 14.04 with a 200 seconds timeout.

Table 2.1 and Figure 2.2 compare the three measures: *e-gini*, *pq-entropy* and *hitting set*. None of the algorithms dominates. All solvers time-out on two benchmarks each. The hitting-set measure is the only one to solve `LinExpr_eq2`. E-gini and pq-entropy can solve the same set of benchmarks but their performance differs on the `example*` specs, where e-gini performs better, and `max*` where pq-entropy performs better.

Table 2.1 also compares the compositional approach to synthesis presented in this chapter (using three learners, one for leaf expressions, one for predicates, and one for the Boolean expression combining them) with a *monolithic* learner based on a CEGIS algorithm that synthesizes the entire function using a constraint solver. As is evident from this table, our algorithm is more often than not faster than the monolithic constraint-based solver. The latter times out on a large number of specifications.

The CVC4 SMT-solver based synthesis tool [89] (which won the conditional linear integer arithmetic track in the SyGuS 2015 and 2016 competitions [51, 64]) worked very fast on these benchmarks, in general, but does not *generalize* from underspecifications. On specifications that list a set of input-output examples (marked with $*$ in Figure 2.2), CVC4 simply returns the precise map that the specification contains, without generalizing it. CVC4 allows restricting the syntax of target functions, but using this feature to force generalization (by disallowing large constants) renders them unsolvable. CVC4 was also not able to solve, surprisingly, the fairly simple specification `max3Univ` (although it has the single-invocation property [89]).

The general track SyGuS solvers (enumerative, stochastic, constraint-solver, and Sketch) [1] do not work well for these benchmarks (and did not fare well in the competitions either); for example, the enumerative solver, which was the winner in 2014 can solve only 16 of the 44 benchmarks.

Benchmarks	e-gini		pq-entropy		Hitting set		Constr. solver
	Rounds	Time	Rounds	Time	Rounds	Time	Time
Jha_Obs	1	0.2	1	0.2	1	0.2	0.5
LinExpr_eq1	-	TO	-	TO	-	TO	TO
LinExpr_eq1ex	9	3.6	10	3.3	10	3.2	122.9
LinExpr_eq2	-	TO	-	TO	31	36.8	2.3
LinExpr_eq2ex	48	94.5	49	93.6	50	93.1	1.0
LinExpr_inv1_ex	39	11.5	39	10.9	38	11.1	0.1
array_search_2	7	0.6	7	0.6	6	0.6	2.6
array_search_3	7	1.3	7	1.2	7	1.2	30.2
array_search_4	12	4.5	11	4.4	16	4.6	TO
array_search_5	19	10.7	20	10.7	16	10.6	TO
array_search_6	22	50.2	22	40.0	22	49.3	TO
array_search_7	27	174.6	22	174.0	20	170.1	TO
array_sum_2_15	5	0.3	5	0.3	5	0.3	0.7
array_sum_2_5	7	0.4	7	0.4	12	0.6	0.2
array_sum_3_15	9	0.8	9	0.7	40	15.8	43.0
array_sum_3_5	31	2.1	39	2.9	20	1.7	12.7
array_sum_4_15	28	7.0	76	26.0	44	139.5	TO
array_sum_4_5	187	169.8	105	47.2	-	TO	TO
max2	3	0.2	3	0.2	3	0.2	0.3
max3	8	0.4	8	0.4	8	0.4	4.7
max3Univ	9	0.7	9	0.7	8	0.7	2.5
max4	18	1.0	16	0.9	12	0.7	TO
max5	44	2.8	51	3.4	32	2.2	TO
max6	130	15.4	94	9.3	47	5.2	TO
max7	327	136.1	271	98.6	65	13.2	TO
max8	-	TO	-	TO	140	92.0	TO
example1	3	0.3	3	0.3	3	0.3	1.2
example2	31	2.4	30	2.0	-	TO	TO
example3	10	3.2	12	46.8	14	65.3	2.6
example4	29	15.0	46	61.1	52	66.7	TO
example5	9	0.6	9	0.6	20	1.5	TO
guard1	3	0.3	3	0.3	3	0.3	0.2
guard2	2	0.3	2	0.3	2	0.3	0.2
guard3	4	0.4	4	0.4	4	0.4	0.3
guard4	4	0.4	4	0.4	4	0.4	0.3
ite1	4	0.7	4	0.7	4	0.7	2.8
ite2	9	0.9	11	1.0	7	0.9	1.9
plane1	1	0.2	1	0.2	1	0.2	0.1
plane2	1	0.4	1	0.4	1	0.4	0.2
plane3	1	0.4	1	0.4	1	0.4	0.2
s1	5	0.3	5	0.3	5	0.3	0.1
s2	2	0.2	2	0.2	2	0.2	0.1
s3	1	0.2	1	0.2	1	0.2	0.1

Table 2.1: Experimental performance of the measures e-gini, pq-entropy, hitting set and the constraint solver. Times are given in seconds. “TO” indicates a timeout of $200s$.

The above results show that the synthesis framework developed in this chapter that uses theory-specific solvers for basic expressions and predicates, and combines them using a classification learner yields a competitive solver for the linear integer arithmetic domain. We believe more extensive benchmarks are needed to fine-tune our algorithm, and especially in choosing the right statistical measures for decision-tree learning.

2.7 RELATED WORK

Our learning task is closely related to the syntax-guided synthesis framework (SyGuS) [1], which provides a language, similar to SMTLib [90], to describe synthesis problems. Several solvers following the counterexample-guided inductive synthesis approach (CEGIS) [3] for SyGuS have been developed [1], including an enumerative solver, a solver based on constraint solving, one based on stochastic search, and one based on the program synthesizer Sketch [91]. Recently, a solver based on CVC4 [89] has also been presented.

There has been several works on synthesizing piece-wise affine models of hybrid dynamical systems from input-output examples [92, 93, 94, 95] (we refer the reader to [96] for a comprehensive survey). The setting there is to learn an affine model passively (i.e., without feedback whether the synthesized model satisfies some specification) and, consequently, only approximates the actual system. The learning framework we develop in this chapter, as well as the synthesis algorithms we use for linear-arithmetic (the outer learner, the expression synthesizer and the predicate synthesizer) can be seen as an abstract learning framework [97].

2.8 CONCLUSIONS AND FUTURE WORK

We have presented a novel compositional framework for synthesizing piece-wise functions over any theory that combines three engines— a synthesizer for the simpler leaf expressions used in a region, the predicates that can be used to define the boundaries of regions, and the Boolean expression that defines the regions themselves and chooses the leaf expressions to apply to each region. We have shown how to formulate automatically the specifications for synthesizing leaf expressions and predicate expressions from the synthesis specification, and developed generic classification algorithms for learning regions and mapping them to expressions using decision-tree based machine-learning algorithms.

One future direction worth pursuing is to build both specific learning algorithms for synthesis problems based on our framework, as well as build general solutions to synthesis (say for all SyGuS specifications). One piece of work that has emerged since the publication of our result is the EUSolver [65], which can be seen as an instantiation of our framework, using enumerative techniques to synthesize both leaf expressions and predicates, and using a decision-tree classifier similar to ours for finding and mapping regions to expressions. The EUSolver has performed particularly well in the SyGuS 2016 [64] competition, winning several tracks, and in particular working well for the class of ICFP benchmark synthesis challenge problems, solving a large proportion of them for the first time. We also note that the original winner for the ICFP benchmarks (in the competition held in 2013 [66]) also used a compositional approach to synthesis that discovered leaf expressions individually for points and then combined them. This experimental evidence suggests that the compositional framework outlined in this chapter is likely a more efficient approach to synthesis of piece-wise functions.

Another interesting future direction is to extend our framework beyond single-point definable/refutable specifications, in particular to bounded point definable/refutable functions. When synthesizing *inductive invariants* for programs, the counterexamples to hypothesized invariants are not single counterexamples, but usually involve *two* counterexamples connected with an implication (see the model of ICE learning [45]). Extending our framework to synthesize for such specifications would be interesting. (Note that in invariants, the leaf expressions are fixed (T/F), and only the predicates separating regions need to be synthesized.)

In summary, the synthesis approach developed in this chapter brings a new technique, namely machine-learning, to solving synthesis problems, in addition to existing techniques such as enumeration, constraint-solving, and stochastic search. Leaf expressions and predicates belong to particular theories that have complex semantics, and are hence best synthesized using dedicated synthesis procedures. However, combining the predicates to form regions and mapping regions to particular expressions can be seen as a generic classification problem that can be realized by learning Boolean formulas, and is independent of the underlying theories. By using a learner of Boolean formulas (decision trees in our setting), we can combine theory-specific synthesizers for leaf expressions and predicates to build efficient learners of piece-wise functions.

CHAPTER 3: INVARIANT SYNTHESIS FOR INCOMPLETE VERIFICATION ENGINES

In this chapter we consider the application of deductive program verification where programmers typically annotate their code with contracts and inductive invariants, and use high-level directives to validate verification conditions using a mostly-automated logic engine. Specifically, we propose a framework for synthesizing inductive invariants to make the task of program verification automatic.

In prior work, learning-based counterexample guided inductive synthesis (CEGIS) methods have been proposed [45, 46, 47, 67], which are data-driven and learn from concrete program configurations returned by the verification oracle as counterexamples to incorrect invariants. However, these techniques cannot be used in settings where the underlying verification problem falls in an undecidable theory, and the verification oracle is incomplete i.e., the oracle resorts to sound but incomplete heuristics to check validity of verification conditions and hence cannot generate a concrete model when verification conditions are not provable.

The framework we propose in this chapter assumes that we have an incomplete verification oracle. In this setting, we extract certain non-provability information from the verification oracle as counterexamples when the conjectured invariant results in verification conditions that cannot be proven. The non-provability information is a Boolean formula on a fixed set of predicates, that generalizes the reason for non-provability, hence pruning the space of future conjectured predicates. Finally, we reduce the formula-driven problem of learning expressions from non-provability information to the data-driven ICE model [45]. This reduction allows us to use a host of existing ICE learning algorithms and results in a robust invariant synthesis framework that guarantees to synthesize a provable invariant if one exists.

In particular, in this chapter:

- The application is inductive invariant synthesis.
- The verification oracle is an oracle for heap verification using natural proofs, and a quantifier-instantiation based logic solver for contracts that have universally quantified formulas. In both cases, the verification oracle is sound but not complete.
- The verification oracle, being incomplete, cannot return concrete counterexamples for proposed invariants. Consequently, the class of counterexamples encode *non-provability* information of particular predicates in the postcondition of Hoare-triples.
- The learning algorithm needs to propose hypotheses that include only provable concepts. We implement this using a reduction to ICE-learning of Boolean formulas.

3.1 INTRODUCTION

The paradigm of *deductive verification* [98, 99] combines manual annotations and semi-automated theorem proving to prove programs correct. Programmers annotate code they develop with contracts and inductive invariants, and use high-level directives to an underlying, mostly-automated logic engine to verify their programs correct. Several mature logic engines have emerged that support such verification, in particular tools based on the intermediate verification language Boogie [71] and the SMT solver Z3 [88] (e.g., Vcc [100] and Dafny [101]).

Viewed through the lens of deductive verification, the primary challenges in automating verification are two-fold. First, even when strong annotations in terms of contracts and inductive invariants are given, the validity problem for the resulting verification conditions is often undecidable (e.g., in reasoning about the heap, reasoning with quantified logics, and reasoning with non-linear arithmetic). Second, the synthesis of loop invariants and strengthenings of contracts that prove a program correct needs to be automated to lift this burden currently borne by the programmer.

A standard technique to solve the first problem (i.e., intractability of validity checking of verification conditions) is to build automated, sound but incomplete verification engines for validating verification conditions, thus skirting the undecidability barrier. Several such techniques exist; for instance, for reasoning with quantified formulas, tactics such as E-matching [102, 103], pattern-based quantifier instantiation [103], model-based quantifier instantiation [104] are effective in practice, and they are known to be complete in certain settings [105]. In the realm of heap verification, the so-called *natural proof method* explicitly aims to provide automated and sound but incomplete methods for checking validity of verification conditions with specifications in separation logic [68, 69, 105, 106].

Turning to the second problem of invariant generation, several techniques have emerged that can synthesize invariants automatically when validation of verification conditions fall in decidable classes. Prominent among these are interpolation [107] and IC3/PDR [108, 109]. Moreover, a class of learning-based counter-example guided inductive synthesis (CEGIS) methods have emerged recently, including the ICE learning model [45] for which various instantiations exist [45, 46, 47, 67]. The key feature of the latter methods is a program-agnostic, data-driven learner that learns invariants in tandem with a verification engine that provides concrete program configurations as counterexamples to incorrect invariants.

Although classical invariant synthesis techniques, such as Houdini [70], are sometimes used with incomplete verification engines, to the best of our knowledge, there is no fundamental argument as to why this should work in general. In fact, we are not aware of any systematic technique for synthesizing invariants when the underlying verification problem falls in an undecidable theory. When verification is undecidable and the engine resorts to sound but incomplete heuristics to check validity of verification conditions, it is unclear how to extend interpolation/IC3/PDR techniques to this setting. Data-driven learning of invariants is also hard to extend since the verification engine typically cannot generate a concrete model for the negation of verification conditions when verification fails. Hence, it cannot produce the concrete configurations that the learner needs.

In this work we propose a learning-based invariant synthesis framework that learns invariants using non-provability information provided by verification engines. Intuitively, when a conjectured invariant results in verification conditions that cannot be proven, the idea is that the verification engine must return information that generalizes the reason for non-provability, hence pruning the space of future conjectured invariants.

Our framework assumes a verification engine for an undecidable theory $\mathcal{U}$ that reduces verification conditions to a decidable theory $\mathcal{D}$ (e.g., using heuristics such as bounded quantifier instantiation to remove universal quantifiers, function unfolding to remove recursive definitions, and so on) that permits producing models for satisfiable formulas. The translation is assumed to be conservative in the sense that if the translated formula in $\mathcal{D}$ is valid, then we are assured that the original verification condition is $\mathcal{U}$-valid. If the verification condition is found to be not $\mathcal{D}$-valid (i.e., its negation is satisfiable), on the other hand, our framework describes how to extract non-provability information from the $\mathcal{D}$-model. This information is encoded as conjunctions and disjunctions in a Boolean theory $\mathcal{B}$, called *conjunctive/disjunctive non-provability information (CD-NPI)*, and communicated back to the learner.

To complete our framework, we show how the formula-driven problem of learning expressions from CD-NPI constraints can be reduced to the data-driven ICE model. This reduction allows us to use a host of existing ICE learning algorithms and results in a robust invariant synthesis framework that guarantees to synthesize a provable invariant if one exists.

However, our CD-NPI learning framework has non-trivial requirements on the verification engine, and building or adapting appropriate engines is not straightforward. To show that our framework is indeed applicable and effective in practice, our second contribution is an application of our technique to two verification domains where the underlying verification is undecidable:

- Our first setting is the verification of dynamically manipulated data-structures against rich logics that combine properties of structure, separation, arithmetic, and data. We show how *natural proof verification engines* [68, 105], which are sound but incomplete verification engines that translate a powerful undecidable separation logic called DRYAD to decidable logics, can be fit into our framework. Moreover, we implement a prototype of such a verification engine on top of the program verifier Boogie [71] and demonstrate that this prototype is able to fully automatically verify a large suite of benchmarks, containing standard algorithms for manipulating singly and doubly linked lists, sorted lists, as well as balanced and sorted trees.

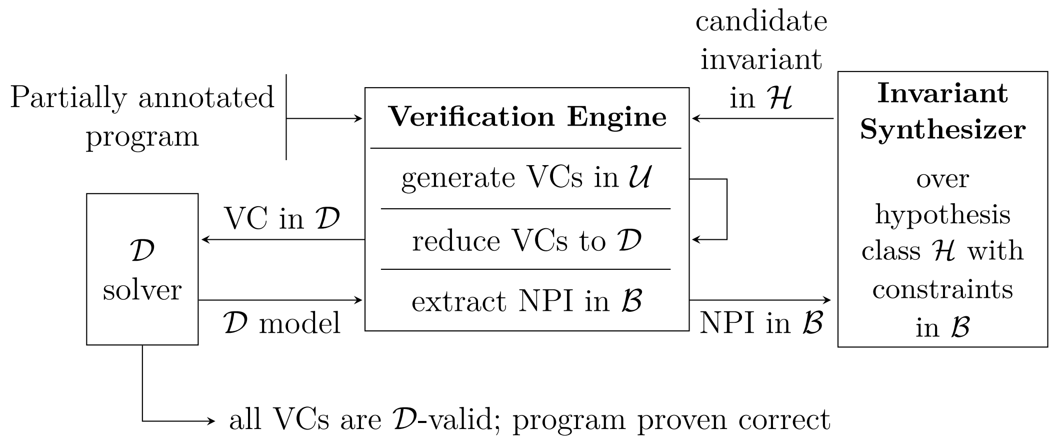
- The second setting addresses the verification of programs against specifications with universal quantification, which renders verification undecidable in general. In this situation, automated verification engines commonly use a variety of bounded quantifier instantiation techniques (such as E-matching, triggers, and model-based quantifier instantiation) to replace universal quantification by conjunctions over a specific set of terms. This soundly reduces satisfiability checking of the negated verification conditions to a decidable theory. Based on such techniques, we implement our framework and we show that it is able to effectively generate invariants that prove a challenging suite of programs correct against universally quantified specifications.

3.2 AN INVARIANT SYNTHESIS FRAMEWORK FOR INCOMPLETE VERIFICATION ENGINES

In this section, we develop our framework for synthesizing inductive invariants for incomplete verification engines, using a counter-example guided inductive synthesis approach. We do this in a setting where the hypothesis space consists of formulas that are Boolean combinations of a fixed set of predicates $\mathcal{P}$, which need not be finite for the general framework (although we assume $\mathcal{P}$ to be a finite set of predicates when developing concrete learning algorithms later). For the rest of this section, let us fix a program P that is annotated with assertions (and possibly with some partial annotations describing pre-conditions, post-conditions, and assertions). Moreover, we say that a formula α is *weaker* (*stronger*) than a formula β in a logic $\mathcal{L}$ if $\vdash_{\mathcal{L}} \beta \Rightarrow \alpha$ ($\vdash_{\mathcal{L}} \alpha \Rightarrow \beta$) where $\vdash_{\mathcal{L}} \varphi$ means that φ is valid in $\mathcal{L}$.

Figure 3.1 depicts our general framework of invariant synthesis when verification is undecidable. We fix several parameters for our verification effort. First, let us assume a uniform signature for logics in terms of constant symbols, relation symbols, functions, and types. For simplicity of exposition, we use the same syntactic logic for the various logics $\mathcal{U}$, $\mathcal{D}$, $\mathcal{B}$ in our framework as well as for the logic $\mathcal{H}$ used to express invariants.

Let us fix $\mathcal{U}$ as the underlying theory that is ideally needed for validating the verification conditions that arise for the program; we presume validity of formulas in $\mathcal{U}$ is undecidable. Since $\mathcal{U}$ is an undecidable theory, the engine will resort to sound approximations (e.g., using


candidate invariant in $\mathcal{H}$
Partially annotated program
Verification Engine
generate VCs in $\mathcal{U}$
reduce VCs to $\mathcal{D}$
extract NPI in $\mathcal{B}$
Invariant Synthesizer
over hypothesis class $\mathcal{H}$ with constraints in $\mathcal{B}$
$\mathcal{D}$ solver
VC in $\mathcal{D}$
$\mathcal{D}$ model
NPI in $\mathcal{B}$
all VCs are $\mathcal{D}$-valid; program proven correct

$\mathcal{H}$	–	the hypothesis class of invariants
$\mathcal{U}$	–	the underlying theory of the program, assumed to be undecidable
$\mathcal{D}$	–	the theory that the verification engine soundly reduces verification conditions to; decidable and can produce models
$\mathcal{B}$	–	the theory of propositional logic that the verification engine uses to communicate to the invariant synthesis engine

Figure 3.1: A non-provability information (NPI) framework for invariant synthesis

bounded quantifier instantiations using mechanisms such as triggers [102], bounded unfolding of recursive functions, or natural proofs [68, 105]) to reduce this logical task to a *decidable* theory $\mathcal{D}$. This reduction is assumed to be sound in the sense that if the resulting formulas in $\mathcal{D}$ are valid, then the verification conditions are valid in $\mathcal{U}$ as well. If a formula is found *not valid* in $\mathcal{D}$, then we require that the logic solver for $\mathcal{D}$ returns a model for the negation of the formula.[1] Note that this model may not be a model for the negation of the formula in $\mathcal{U}$.

Moreover, we fix a hypothesis class $\mathcal{H}$ for invariants consisting of *positive* Boolean combination of predicates over a fixed set of predicates $\mathcal{P}$. Note that considering only *positive* formulas over $\mathcal{P}$ is not a restriction in general because one can always add negations of predicates to $\mathcal{P}$, thus effectively synthesizing any Boolean combination of predicates. The restriction to positive Boolean formulas is in fact desirable as it allows restricting invariants to *not* negate certain predicates, which is useful when predicates have intuitionistic definitions (as several recursive definitions of heap properties do).

The invariant synthesis proceeds in rounds, where in each round the synthesizer proposes invariants in $\mathcal{H}$. The verification engine generates verification conditions in accordance to these invariants in the underlying theory $\mathcal{U}$. It then proceeds to translate them into the

[1]Note that our framework requires model construction in the theory $\mathcal{D}$. Hence, incomplete logic solvers for $\mathcal{U}$ that simply time out after some time threshold or search for a proof of a particular kind and give up otherwise are not suitable candidates.

decidable theory $\mathcal{D}$, and gives them to a solver that decides their validity in the theory $\mathcal{D}$. If the verification conditions are found to be $\mathcal{D}$-valid, we have successfully proven the program correct by virtue of the fact that the verification engine reduced verification conditions in a sound fashion to $\mathcal{D}$.

However, if the formula is found not to be $\mathcal{D}$-valid, the solver returns a $\mathcal{D}$-model for its negation. The verification engine then extracts from this model certain *non-provability information (NPI)*, expressed as Boolean formulas in a Boolean theory $\mathcal{B}$, which captures more general reasons why the verification failed (the rest of this section is devoted to developing this notion of non-provability information). This non-provability information is communicated to the synthesizer, which then proceeds to synthesize a new conjecture invariant that satisfies the non-provability constraints provided in all previous rounds.

In order for the verification engine to extract meaningful non-provability information, we make the following natural assumption, called *normality*, which essentially states that the engine can do at least some minimal Boolean reasoning (if a Hoare triple is not provable, then Boolean weakenings of the precondition and Boolean strengthening of the post-condition must also be unprovable):

Definition 3.1. *A verification engine is* normal *if it satisfies two properties:*

1. *If the engine cannot prove the validity of the Hoare triple* $\{\alpha\}s\{\gamma\}$ *and* $\vdash_{\mathcal{B}} \delta \Rightarrow \gamma$*, then it cannot prove the validity of the Hoare triple* $\{\alpha\}s\{\delta\}$*.*
2. *If the engine cannot prove the validity of the Hoare triple* $\{\gamma\}s\{\beta\}$ *and* $\vdash_{\mathcal{B}} \gamma \Rightarrow \delta$*, then it cannot prove the validity of the Hoare triple* $\{\delta\}s\{\beta\}$*.*

In Section 3.2.1, we now develop an appropriate language to communicate non-provability constraints, which allow the learner to appropriately weaken or strengthen a future hypothesis. It turns out that *pure conjunctions* and *pure disjunctions* over $\mathcal{P}$, which we term *CD-NPI constraints* (conjunctive/disjunctive non-provability information constraints), are sufficient for this purpose. We also describe concretely how the verification engine can extract this non-provability information from $\mathcal{D}$-models that witness that negations of VCs are satisfiable. Then, in Section 3.2.2, we show how to build learners for CD-NPI constraints by reducing this learning problem to another, well-studied learning framework for invariants called ICE learning. Section 3.2.3 illustrates our definitions on an example, and Section 3.2.4 argues the soundness of our framework and guarantees of convergence.

3.2.1 Conjunctive/Disjunctive Non-provability Information

We assume that the underlying decidable theory $\mathcal{D}$ is stronger than propositional theory $\mathcal{B}$, meaning that every valid statement in $\mathcal{B}$ is valid in $\mathcal{D}$ as well. The reader may want to keep the following as a running example where $\mathcal{D}$ is the decidable theory of uninterpreted functions and linear arithmetic, say. In this setting, a formula is $\mathcal{B}$-valid if, when treating atomic formulas as Boolean variables, the formula is propositionally valid. For instance, $f(x) = y \Rightarrow f(f(x)) = f(y)$ will not be $\mathcal{B}$-valid though it is $\mathcal{D}$-valid, while $f(x) = y \vee \neg(f(x) = y)$ is $\mathcal{B}$-valid.

To formally define CD-NPI constraints and their extraction from a failed verification attempt, let us first introduce the following notation. For any $\mathcal{U}$-formula φ, let $approx(\varphi)$ denote the $\mathcal{D}$-formula that the verification engine generates such that the $\mathcal{D}$-validity of $approx(\varphi)$ implies the $\mathcal{U}$-validity of φ. Moreover, for any Hoare triple $\{\alpha\}s\{\beta\}$, let $VC(\{\alpha\}s\{\beta\})$ denote the verification condition in $\mathcal{U}$ corresponding to the Hoare triple that the verification engine generates.

Let us now assume, for the sake of a simpler exposition, that the program has a single annotation hole A where we need to synthesize an inductive invariant to prove the program correct. Further, suppose the learner conjectures an annotation γ as an inductive invariant for the annotation hole A, and the verification engine fails to prove the verification condition corresponding to a Hoare triple $\{\alpha\}s\{\beta\}$, where either α, β, or both could involve the synthesized annotation. This means that the negation of $approx(VC(\{\alpha\}s\{\gamma\}))$ is $\mathcal{D}$-satisfiable and the verification engine needs to extract non-provability information from a model of it. To this end, we assume that every program snippet s has been augmented with a set of ghost variables $g_1, \ldots, g_n$ that track the predicates $p_1, \ldots, p_n$ mentioned in the invariant (i.e., these ghost variables are assigned the values of the predicates). The valuation $\vec{v} = \langle v_1, \ldots, v_n \rangle$ of the ghost variables in the model before the execution of s and the valuation $\vec{v}' = \langle v'_1, \ldots, v'_n \rangle$ after the execution of s can then be used to derive non-provability information, as we describe shortly.

The type of non-provability information the verification engine extracts depends on where the annotation appears in a Hoare triple $\{\alpha\}s\{\beta\}$. More specifically, the synthesized annotation might appear in α, in β, or in both. We now handle all three cases individually.

- Assume the verification of a Hoare triple of the form $\{\alpha\}s\{\gamma\}$ fails (i.e., the verification engine cannot prove a verification condition where the pre-condition α is a user-supplied annotation and the post-condition is the synthesized annotation γ). Then, $approx(VC(\{\alpha\}s\{\gamma\}))$ is not $\mathcal{D}$-valid, and the decision procedure for $\mathcal{D}$ would generate a model for its negation.

Since γ is a positive Boolean combination, the reason why $\vec{v}'$ does not satisfy γ is due to the variables mapped to *false* by $\vec{v}'$, as any valuation extending this will not satisfy γ. Intuitively, this means that the $\mathcal{D}$-solver is not able to prove the predicates in $P_{\mathit{false}} = \{p_i \mid v'_i = \mathit{false}\}$. In other words, $\{\alpha\}s\{\bigvee P_{\mathit{false}}\}$ is unprovable (a witness to this fact is the model of the negation of $\mathit{approx}(VC(\{\alpha\}s\{\gamma\}))$ from which the values $\vec{v}'$ are derived). Note that any invariant γ' that is stronger than $\bigvee P_{\mathit{false}}$ will result in an unprovable verification condition due to the verification engine being normal. Consequently we can choose $\chi = \bigvee P_{\mathit{false}}$ as the weakening constraint, demanding that future invariants should not be stronger than χ.

The verification engine now communicates χ to the synthesizer, asking it never to conjecture in future rounds invariants γ'' that are stronger than χ (i.e., such that $\not\models_{\mathcal{B}} \gamma'' \Rightarrow \chi$).

- The next case is when a Hoare triple of the form $\{\gamma\}s\{\beta\}$ fails to be proven (i.e., the verification engine cannot prove a verification condition where the post-condition β is a user-supplied annotation and the pre-condition is the synthesized annotation γ). Using similar arguments as above, the *conjunction* $\eta = \bigwedge\{p_i \mid v_i = \mathit{true}\}$ of the predicates mapped to *true* by $\vec{v}$ in the corresponding $\mathcal{D}$-model gives a stronger precondition η such that $\{\eta\}s\{\alpha\}$ is not provable. Hence, η is a valid *strengthening* constraint. The verification engine now communicates η to the synthesizer, asking it never to conjecture in future rounds invariants γ'' that are weaker than η (i.e., such that $\not\models_{\mathcal{B}} \eta \Rightarrow \gamma''$).

- Finally, consider the case when the Hoare triple is of the form $\{\gamma\}s\{\gamma\}$ and fails to be proven (i.e., the verification engine cannot prove a verification condition where the pre- and post-condition is the synthesized annotation γ). In this case, the verification engine can offer advice on how γ can be strengthened *or* weakened to avoid this model. Analogous to the two cases above, the verification engine extracts a pair of formulas (η, χ), called an *inductivity constraint*, based on the variables mapped to *true* by $\vec{v}$ and to *false* by $\vec{v}'$. The meaning of such a constraint is that the invariant synthesizer must conjecture in future rounds invariants γ'' such that either $\not\models_{\mathcal{B}} \eta \Rightarrow \gamma''$ or $\not\models_{\mathcal{B}} \gamma'' \Rightarrow \chi$ holds.

This leads to the following scheme, where γ denotes the conjectured invariant:

- When a Hoare triple of the form $\{\alpha\}s\{\gamma\}$ fails, the verification engine returns the $\mathcal{B}$-formula $\bigvee_{i \mid v'_i = \mathit{false}} p_i$ as a weakening constraint.
- When a Hoare triple of the form $\{\gamma\}s\{\beta\}$ fails, the verification engine returns the $\mathcal{B}$-formula $\bigwedge_{i \mid v_i = \mathit{true}} p_i$ as a strengthening constraint.

- When a Hoare triple of the form $\{\gamma\}s\{\gamma\}$ fails, the verification engine returns the pair $(\bigwedge_{i|v_i=true} p_i, \bigvee_{i|v_i'=false} p_i)$ of $\mathcal{B}$-formulas as an inductivity constraint.

It is not hard to verify that the above formulas are proper strengthening and weakening constraints in the sense that *any* inductive invariant must satisfy these constraints. This motivates the following form of non-provability information.

Definition 3.2 (CD-NPI Samples). *Let $\mathcal{P}$ be a set of predicates. A* CD-NPI sample *(short for* conjunction-disjunction-NPI sample*) is a triple $\mathfrak{S} = (W, S, I)$ consisting of*

- *a finite set W of disjunctions over $\mathcal{P}$ (weakening constraints);*
- *a finite set S of conjunctions over $\mathcal{P}$ (strengthening constraints); and*
- *a finite set I of pairs, where the first element is a conjunction and the second is a disjunction over $\mathcal{P}$ (inductivity constraints).*

An annotation γ is consistent *with a CD-NPI sample $\mathfrak{S} = (W, S, I)$ if $\not\vdash_{\mathcal{B}} \gamma \Rightarrow \chi$ for each $\chi \in W$, $\not\vdash_{\mathcal{B}} \eta \Rightarrow \gamma$ for each $\eta \in S$, and $\not\vdash_{\mathcal{B}} \eta \Rightarrow \gamma$ or $\not\vdash_{\mathcal{B}} \gamma \Rightarrow \chi$ for each $(\eta, \chi) \in I$.*

A CD-NPI learner is an effective procedure that synthesizes, given an CD-NPI sample, an annotation γ consistent with the sample. In our framework, the process of proposing candidate annotations and checking them repeats until the learner proposes a valid annotation or it detects that no valid annotation exists (e.g., if the class of candidate annotations is finite and all annotations are exhausted). We comment on using an CD-NPI learner in this iterative fashion in the next section.

3.2.2 Building CD-NPI Learners

Let us now turn to the problem of building efficient learning algorithms for CD-NPI constraints. To this end, we assume that the set of predicates $\mathcal{P}$ is finite.

Roughly speaking, the CD-NPI learning problem is to synthesize annotations that are positive Boolean combinations of predicates in $\mathcal{P}$ and that are consistent with a given CD-NPI sample. Though this is a learning problem where samples are *formulas*, in this section we reduce CD-NPI learning to a learning problem from *data*. In particular, we show that CD-NPI learning reduces to the ICE learning framework for learning positive Boolean formulas. The latter is a well-studied framework, and the reduction allows us to use efficient learning algorithms developed for ICE learning in order to build CD-NPI learners.

We now first recap the ICE-learning framework and then reduce CD-NPI learning to ICE learning. Finally, we briefly sketch how the popular Houdini algorithm can be seen as an

ICE learning algorithm, which, in turn, allows us to use Houdini as an CD-NPI learning algorithm.

3.2.2.1 The ICE learning framework

Although the ICE learning framework [45] is a general framework for learning inductive invariants, we here consider the case of learning Boolean formulas. To this end, let us fix a set B of Boolean variables. Moreover, let $\mathcal{H}$ be a subclass of positive Boolean formulas over B (i.e., Boolean combinations of variables from B without negation). This class, called the hypothesis class, specifies the admissible solutions to the learning task.

The objective of the (passive) ICE learning algorithm is to learn a formula in $\mathcal{H}$ from positive examples, negative examples, and implication examples. More formally, if $\mathcal{V}$ is the set of valuations $v\colon B \to \{\mathit{true}, \mathit{false}\}$ (mapping variables in B to true or false), then an *ICE sample* is a triple $\mathcal{S} = (S_+, S_-, S_\Rightarrow)$ where $S_+ \subseteq \mathcal{V}$ is a set of *positive examples*, $S_- \subseteq \mathcal{V}$ is a set of *negative examples*, and $S_\Rightarrow \subseteq \mathcal{V} \times \mathcal{V}$ is a set of *implications*. Note that positive and negative examples are *concrete* valuations of the variables in B, and the implication examples are pairs of such concrete valuations.

A formula φ is said to be *consistent with an ICE sample* $\mathcal{S}$ if it satisfies the following three conditions:[2] $v \models \varphi$ for each $v \in S_+$, $v \not\models \varphi$ for each $v \in S_-$, and $v_1 \models \varphi$ implies $v_2 \models \varphi$, for each $(v_1, v_2) \in S_\Rightarrow$.

In algorithmic learning theory, one distinguishes between *passive learning* and *iterative learning*. The former refers to a learning setting in which a learning algorithm is confronted with a finite set of data and has to learn a concept that is consistent with this data. Using our terminology, the *passive ICE learning problem* for a hypothesis class $\mathcal{H}$ is then: *"given an ICE sample $\mathcal{S}$, find a formula in $\mathcal{H}$ that is consistent with $\mathcal{S}$"*. Recall that we here require the learning algorithm to learn positive Boolean formulas, which is stricter than the original ICE framework [45].

Iterative learning, on the other hand, is the iteration of passive learning where new data is added to the sample from one iteration to the next. In a verification context, this new data is generated by the verification engine in response to incorrect annotations and used to guide the learning algorithm towards an annotation that is adequate to prove the program. To reduce our learning framework to ICE learning, it is therefore sufficient to reduce the (passive) CD-NPI learning problem described above to the passive ICE learning problem. We do this next.

[2]In the following, $\models$ denotes the usual satisfaction relation.

3.2.2.2 Reduction of Passive CD-NPI Learning to Passive ICE Learning

Let $\mathcal{H}$ be a subclass of positive Boolean formulas. We reduce the CD-NPI learning problem for $\mathcal{H}$ to the ICE learning problem for $\mathcal{H}$. The main idea is to (a) treat each predicate $p \in \mathcal{P}$ as a Boolean variable for the purpose of ICE learning and (b) to translate a CD-NPI sample $\mathfrak{S}$ into an *equi-consistent* ICE sample $\mathcal{S}_{\mathfrak{S}}$, meaning that a positive Boolean formula is consistent with $\mathfrak{S}$ if and only if it is consistent with $\mathcal{S}_{\mathfrak{S}}$. Then, learning a consistent formula in the CD-NPI framework reduces to learning a consistent formula in the ICE learning framework.

The following lemma will us help translate between the two frameworks. Its proof is straightforward and follows from the following observation about any *positive* formula α: if a valuation v sets a larger subset of variables to true than v' does and $v' \models \alpha$, then $v \models \alpha$ holds as well.

Lemma 3.1. *Let v be a valuation of $\mathcal{P}$ and α be a positive Boolean formula over $\mathcal{P}$. Then, the following holds:*

- *$v \models \alpha$ if and only if $\vdash_{\mathcal{B}} (\bigwedge_{p|v(p)=true} p) \Rightarrow \alpha$ (and, therefore, $v \not\models \alpha$ if and only if $\not\vdash_{\mathcal{B}} (\bigwedge_{p|v(p)=true} p) \Rightarrow \alpha$).*
- *$v \models \alpha$ if and only if $\not\vdash_{\mathcal{B}} \alpha \Rightarrow (\bigvee_{p|v(p)=false} p)$.*

This motivates our translation, which relies on two functions, c and d. The function c translates a conjunction $\bigwedge J$, where $J \subseteq \mathcal{P}$, into the valuation

$$c\Big(\bigwedge J\Big) = v \text{ with } v(p) = \textit{true} \text{ if and only if } p \in J. \tag{3.1}$$

The function d, on the other hand, translates a disjunction $\bigvee J$, where $J \subseteq \mathcal{P}$ is a subset of propositions, into the valuation

$$d\Big(\bigvee J\Big) = v \text{ with } v(p) = \textit{false} \text{ if and only if } p \in J. \tag{3.2}$$

By substituting v in Lemma 3.1 with $c(\bigwedge J)$ and $d(\bigvee J)$, respectively, one immediately obtains the following result.

Lemma 3.2. *Let $J \subseteq \mathcal{P}$ and α be a positive Boolean formula over $\mathcal{P}$. Then, the following holds:*

- *$c\big(\bigwedge J\big) \not\models \alpha$ if and only if $\not\vdash_{\mathcal{B}} \bigwedge J \Rightarrow \alpha$, and*
- *$d\big(\bigvee J\big) \models \alpha$ if and only if $\not\vdash_{\mathcal{B}} \alpha \Rightarrow \bigvee J$.*

Based on the functions c and d, the translation of a CD-NPI sample into an equi-consistent ICE sample is as follows.

Definition 3.3. *Given a CD-NPI sample $\mathfrak{S} = (W, S, I)$, the ICE sample $\mathcal{S}_{\mathfrak{S}} = (S_+, S_-, S_\Rightarrow)$ is defined by*

- $S_+ = \Big\{d(\bigvee J) \mid \bigvee J \in W\Big\}$*;*
- $S_- = \Big\{c(\bigwedge J) \mid \bigwedge J \in S\Big\}$*; and*
- $S_\Rightarrow = \Big\{\Big(c(\bigwedge J_1), d(\bigvee J_2)\Big) \mid (\bigwedge J_1, \bigvee J_2) \in I\Big\}$.

By virtue of the lemma above, we can now establish the correctness of the reduction from the CD-NPI learning problem to the ICE learning problem as follows.

Theorem 3.1. *Let $\mathfrak{S} = (W, S, I)$ be a CD-NPI sample, $\mathcal{S}_{\mathfrak{S}} = (S_+, S_-, S_\Rightarrow)$ the ICE sample as in Definition 3.3, γ a positive Boolean formula over $\mathcal{P}$. Then, γ is consistent with $\mathfrak{S}$ if and only if γ is consistent with $\mathcal{S}_{\mathfrak{S}}$.*

Proof. Let $\mathfrak{S} = (W, S, I)$ be an CD-NPI sample, and let $\mathcal{S}_{\mathfrak{S}} = (S_+, S_-, S_\Rightarrow)$ the ICE sample as in Definition 3.3. Moreover, let γ be a positive Boolean formula. We prove Theorem 3.1 by considering each weakening, strengthening, and inductivity constraint together with their corresponding positive, negative, and implication examples individually.

- Pick a weakening constraint $\bigvee J \in W$, and let $v \in S_+$ with $v = d(\bigvee J)$ be the corresponding positive sample. Moreover, assume that γ is consistent with $\mathfrak{S}$ and, thus, $\not\vdash_{\mathcal{B}} \gamma \Rightarrow \bigvee J$. By Lemma 3.2, this is true if and only if $d\Big(\bigvee J\Big) \models \gamma$. Hence, $v \models \gamma$.

  Conversely, assume that γ is consistent with $\mathcal{S}$. Thus, $v \models \gamma$, which means $d\Big(\bigvee J\Big) \models \gamma$. By Lemma 3.2, this is true if and only if $\not\vdash_{\mathcal{B}} \gamma \Rightarrow \bigvee J$.

- Pick a strengthening constraint $\bigwedge J \in S$, and let $v \in S_-$ with $v = c(\bigwedge J)$ be the corresponding negative sample. Moreover, assume that γ is consistent with $\mathfrak{S}$ and, thus, $\not\vdash_{\mathcal{B}} \bigwedge J \Rightarrow \gamma$. By Lemma 3.2, this is true if and only if $c\Big(\bigwedge J\Big) \not\models \gamma$. Hence, $v \not\models \gamma$.

  Conversely, assume that γ is consistent with $\mathcal{S}$. Thus, $v \not\models \gamma$, which means $c\Big(\bigwedge J\Big) \not\models \gamma$. By Lemma 3.2, this is true if and only if $\not\vdash_{\mathcal{B}} \bigwedge J \Rightarrow \gamma$.

- Following the definition of implication, we split the proof into two cases, depending on whether $\not\vdash_{\mathcal{B}} \bigwedge J \Rightarrow \gamma$ or $\not\vdash_{\mathcal{B}} \gamma \Rightarrow \bigvee J$ (and $v_1 \not\models \gamma$ or $v_2 \models \gamma$ for the reserve direction). However, the proof of the former case uses the same arguments as the proof for strengthening constraints, while the proof of the latter case uses the same arguments as the proof for weakening constraints. Hence, combining both proofs immediately yields the claim. Q.E.D.

3.2.2.3 ICE learners for Boolean formulas

The reduction above allows us to use any ICE learning algorithm in the literature that synthesizes positive Boolean formulas. As we have mentioned earlier, we can add negations of predicates as first-class predicates and, hence, synthesize invariants over the more general class of all Boolean combinations as well.

The problem of passive ICE learning for one round, synthesizing a formula that satisfies the ICE sample, can usually be achieved efficiently and in a variety of ways. However, the crucial aspect is not the complexity of learning in one round but the *number* of rounds it takes to converge to an adequate invariant that proves the program correct. When the set $\mathcal{P}$ of candidate predicates is large (hundreds in our experiments), since the number of Boolean formulas over $\mathcal{P}$ is doubly exponential in $n = |\mathcal{P}|$, building an effective learner is not easy. However, there is one class of formulas that are particularly amenable to efficient ICE learning: *conjunctions of predicates over* $\mathcal{P}$. For this specific case, ICE learning algorithms exist that promise learning the invariant (provided one exists expressible as a conjunct over $\mathcal{P}$) in $(n+1)$ rounds. Note that this learning is essentially finding an invariant in a hypothesis class $\mathcal{H}$ of size 2^n in $(n+1)$ rounds.

Houdini [70] is such a learning algorithm for conjunctive formulas. Though it is typically seen as a particular way to synthesize invariants, it is a prime example of an ICE learner for conjuncts, as described in the work by Garg et al. [45]. In fact, Houdini is similar to the classical PAC learning algorithm for conjunctions [78], but extended to the ICE model. The time Houdini spends in each round is *polynomial* in the size of the sample, and it is guaranteed to converge to an invariant in at most $(n+1)$ rounds (or reports that no conjunctive invariant over $\mathcal{P}$ exists). In our applications, we use this ICE learner to build a CD-NPI learner for conjunctions.

3.2.3 An Illustrative Example

Figure 3.2 illustrates an example program of the Software Verification Competition [110] that given an injective, surjective function A returns the inverse B of the function A. The post-condition of this program expresses that the function B is injective. To prove this program correct, one needs to specify adequate invariants at the loop header and before the

```
int A[ ], B[ ];
int N; axiom ($N > 0$);
bool inImage(int i) { return true; }

procedure inverse()
requires ($\forall x, y.\ 0 \leq x < y < N \implies A[x] \neq A[y]$); // A is injective
requires ($\forall x.\ 0 \leq x < N \wedge inImage(x) \implies (\exists y.\ 0 \leq y < N \wedge A[y] = x)$); // A is surjective
ensures ($\forall x, y.\ 0 \leq x < y < N \implies B[x] \neq B[y]$); // B is injective
{
    int i = 0;
    while (i < N)
    SynthesizeInv($\forall x.\ 0 \leq x < i \implies B[A[x]] = x$); // $b_1$
    {
        B[A[i]] = i;
        i = i + 1;
    }
    SynthesizeInv($\forall x.\ 0 \leq x < N \implies A[B[x]] = x$, // $b_2$
                  $\forall x.\ 0 \leq x < N \wedge inImage(x) \implies A[B[x]] = x$); // $b_3$
    return;
}
```

Figure 3.2: Synthesizing invariants for the program that constructs an inverse B of an injective, surjective function A [110].

return statement in function *inverse* in the program. We wish to synthesize these invariants. For simplicity, let us assume we are provided with a small set of predicates that serve as the basic building blocks for the invariants to be synthesized: b_1 at the loop header and b_2, b_3 before the return statement. Our task, therefore, is to synthesize adequate invariants for this program over these predicates.[3]

Clearly, the verification conditions of this program are undecidable. In fact, the constant Boolean function *inImage* is crucially required to validate certain verification conditions by triggering appropriate quantifier instantiations in the surjectivity condition. Let us now assume that the learner conjectures the loop invariant $\gamma_L = b_1$ and the invariant at the return statement $\gamma_R = b_2 \wedge b_3$. Moreover, suppose that that the verification condition ϕ along the path from the loop exit to the return statement, though valid in the undecidable theory $\mathcal{U}$ (cf. Figure 3.1), is not provable in the decidable theory $\mathcal{D}$ (one that has instantiated quantifiers with ground-terms). Thus, the $\mathcal{D}$-solver returns a model that captures this non-provability information.

The verification engine now gleans this model—it looks for the values assigned to the

[3]In general, one starts with a much larger set of candidate predicates that are automatically generated using program/specification-dependent heuristics.

predicate variables in the model and constructs a CD-NPI constraint for the learner to learn from. For this particular verification condition, the verification engine extracts a pair of formulas (η, χ) with $\eta = b_1$ and $\chi = b_2$ and communicates this as an inductivity constraint to the learner. Intuitively, this constraint means that the verification condition obtained by substituting γ_L with η and γ_R with χ is itself not provable. In subsequent rounds, the learner thus needs to conjecture only those invariants where γ_L is not weaker than η (i.e., $\not\vdash_{\mathcal{B}} b_1 \Rightarrow \gamma_L$) or γ_R is not stronger than χ (i.e., $\not\vdash_{\mathcal{B}} \gamma_R \Rightarrow b_2$).

The learner works by reducing the CD-NPI passive learning problem to ICE learning over a sample over the given set of predicates. Concretely, the inductivity constraint specified above that captures the non-provability of the verification condition ϕ in the theory $\mathcal{D}$ is reduced to an implication constraint $((1,0,0),(1,0,1))$ in the ICE setting, where each datapoint in the ICE sample has values for the predicates b_1, b_2, and b_3, respectively. In the next round, let us assume the learner conjectures the invariants $\gamma_L = b_1$ and $\gamma_R = b_3$. Note that these conjectures satisfy both the ICE constraints and the CD-NPI constraints. In this case, it turns out that the verification conditions along all program paths using these invariants can be proved valid in the theory $\mathcal{D}$. As a result, our invariant synthesis procedure terminates with these as adequate inductive invariants.

3.2.4 Correctness and Convergence of the Invariant Learning Framework

To state the main result of this chapter, let us first assume that the set $\mathcal{P}$ of predicates is finite. We comment on the case of infinitely many predicates at the end of this section.

Theorem 3.2. *Assume a normal verification engine for a program P to be given. Moreover, let $\mathcal{P}$ be a finite set of predicates over the variables in P and $\mathcal{H}$ a hypothesis class consisting of positive Boolean combinations of predicates in $\mathcal{P}$. If there exists an annotation in $\mathcal{H}$ that the verification engine can use to prove P correct, then the CD-NPI framework described in Section 3.2.1 is guaranteed to converge to such an annotation in finite time.*

Proof. The proof proceeds in two steps. First, we show that a normal verification engine is *honest*, meaning that the non-provability information returned by such an engine does not rule out any adequate and provable annotation. Second, we show that any consistent learner (i.e., a learner that only produces consistent hypotheses), when paired with an honest verification engine, makes *progress* from one round to another. Finally, we combine both results to show that the framework eventually converges to an adequate and provable annotation.

Honesty of the verification engine We show honesty of the verification engine by contradiction.

- Suppose that the verification replies to a candidate invariant γ proposed by the learner with a weakening constraint χ because it could not prove the validity of the Hoare triple $\{\alpha\}s\{\gamma\}$. This effectively forces any future conjecture γ' to satisfy $\nvdash_{\mathcal{B}} \gamma' \Rightarrow \chi$.

  Now, suppose that there exists an invariant δ such that $\vdash_{\mathcal{B}} \delta \Rightarrow \chi$ and the verification engine can prove the validity of $\{\alpha\}s\{\delta\}$ (in other words, the adequate invariant δ is ruled out by the weakening constraint χ). Due to the fact that the verification engine is normal (in particular, by contraposition of Part 1 of Definition 3.1), this implies that the verification engine can also prove the validity of $\{\alpha\}s\{\chi\}$. However, this is a contradiction to χ being a weakening constraint.

- Suppose that the verification engine replies to a candidate invariant γ proposed by the learner with a strengthening constraint η because it could not prove the validity of the Hoare triple $\{\gamma\}s\{\beta\}$. This effectively forces any future conjecture γ to satisfy $\nvdash_{\mathcal{B}} \eta \Rightarrow \gamma'$.

  Now, suppose that there exists an invariant δ such that $\vdash_{\mathcal{B}} \eta \Rightarrow \delta$ and the verification engine can prove the validity of $\{\delta\}s\{\beta\}$ (in other words, the adequate invariant δ is ruled out by the weakening constraint η). Due to the fact that the verification engine is normal (in particular, by contraposition of Part 2 of Definition 3.1), this implies that the verification engine can also prove the validity of $\{\eta\}s\{\beta\}$. However, this is a contradiction to η being a strengthening constraint.

- Combining the arguments for weakening and strengthening constraints immediately results in a contradiction for the case of inductivity constraints as well.

Progress of the learner Now suppose that the learning algorithm is consistent, meaning that it always produces an annotation that is consistent with the current sample. Moreover, assume that the sample in iteration $i \in \mathbb{N}$ is $\mathfrak{S}_i$ and the learner produces the annotation γ_i. If γ_i is inadequate to prove the program correct, the verification engine returns a constraint c. The learner adds this constraint to the sample, obtaining the sample $\mathfrak{S}_{i+1}$ of the next iteration.

Since verification with γ_i failed, which is witnessed by c, we know that γ_i is not consistent with c. The next conjecture γ_{i+1}, however, is guaranteed to be consistent with $\mathfrak{S}_{i+1}$ (which contains c) because the learner is consistent. Hence, γ_i and γ_{i+1} are semantically different.

Using this argument repeatedly shows that each annotation γ_i that a consistent learner has produced is semantically different from any previous annotation γ_j for $j < i$.

Convergence We first make two observations.

1. The number of semantically different hypotheses in the hypothesis space $\mathcal{H}$ is finite because the set $\mathcal{P}$ is finite. Recall that $\mathcal{H}$ is the class of all positive Boolean combinations of predicates in $\mathcal{P}$.

2. Due to the honesty of the verification engine, every annotation that the verification engine can use to prove the program correct is guaranteed to be consistent with any sample produced during the learning process.

Now, suppose that there exists an annotation that the verification engine can use to prove the program correct. Since the learner is consistent (implying that it is honest), all conjectures produced during the learning process are semantically different. Thus, the learner will at some point have exhausted all incorrect annotations in $\mathcal{H}$ (due to Observation 1). However, there exists at least one annotation that the verification engine can use to prove the program correct. Moreover, any such annotation is guaranteed to be consistent with the current sample (due to Observation 2). Thus, the annotation conjectured next is necessarily one that the verification engine can use to prove the program correct. Q.E.D.

Under certain realistic assumptions on the CD-NPI learning algorithm, Theorem 3.2 remains true even if the number of predicates is infinite. An example of such an assumption is that the learning algorithm always conjectures a smallest consistent annotation with respect to some fixed total order on $\mathcal{H}$. In this case, one can show that such a learner will at some point have proposed all inadequate annotation up to the smallest annotation the verification engine can use to prove the program correct. It will then conjecture this annotation in the next iteration. A correctness proof of this informal argument in a more general setting, called *abstract learning frameworks for synthesis*, has been given by Löding, Madhusudan, and Neider [97].

3.3 LEARNING INVARIANTS THAT AID NATURAL PROOFS FOR HEAP REASONING

We now develop an instantiation of our learning framework for verification engines based on natural proofs for heap reasoning [68, 69].

3.3.1 Background: Natural Proofs and Dryad

Dryad [68, 69] is a dialect of separation logic that allows expressing second order properties using recursive functions and predicates. Dryad has a few restrictions, such as disallowing negations inside recursive definitions and in sub-formulas connected by spatial conjunctions (see Pek, Qiu, and Madhusudan [68]). However, it is expressive enough to define a variety of data-structures (singly and doubly linked lists, sorted lists, binary search trees, AVL trees, maxheaps, treaps) and recursive definitions over them that map to numbers (length, height, etc.). Dryad also allows expressing properties about the data stored within the heap (the multiset of keys stored in lists, trees, etc.).

Natural proofs [68, 69] is a sound but incomplete strategy for deciding satisfiability of Dryad formulas. The first step the natural proof verifier performs is to convert all predicates and functions in a Dryad-annotated program to classical logic. This translation introduces *heaplets* (modeled as sets of locations) explicitly in the logic. Furthermore, it introduces assertions demanding that the access of each method is contained to the heaplet implicitly defined by its pre-condition (taking into account newly allocated or freed nodes) and the modified heaplet at the end of the program precisely matches the heaplet implicitly defined by the post-condition.

In the second step, the natural proof verifier applies the following three transformations to the program: (a) it abstracts all recursive definitions on the heap using uninterpreted functions and introduces finite-depth unfoldings of these definitions at every place in the code where locations are dereferenced, (b) it model heaplets and other sets using the decidable theory of maps, and (c) it insert frame reasoning explicitly in the code, which allows the verifier to derive that certain properties continue to hold across a heap update (or function call) using the heaplet that is being modified. Subsequently, the natural proof verifier translates the transformed program to Boogie [71], an intermediate verification language that handles proof obligations using automatic theorem provers (typically SMT solvers).

To perform both steps automatically, we used the tool VCDryad [68], which extends VCC [100] and operates on heap-manipulating C programs. The result is a Boogie program with no recursive definitions, where all verification conditions are in decidable logics, and where a standard SMT solver, such as Z3 [88], can return models when formulas are satisfiable. The program in question can then be verified if supplied with correct inductive loop-invariants and adequate pre/post-conditions. We refer the reader to the work on Dryad [69] and VCDryad [68] for more details.

3.3.2 Learning Heap Invariants

We have implemented a prototype[4] that consists of the entire VCDryad pipeline, which takes C programs annotated in Dryad and converts them to Boogie programs via the natural proof transformations described above. We then apply our transformation to the ICE learning framework and pair Boogie with the ICE learning algorithm Houdini [70] that learns conjunctive invariants over a finite set of predicates (we describe shortly how these predicates are generated). After these transformations, Boogie satisfies the requirements on verification engines of our framework.

Given the Dryad definitions of data structures, we automatically generate a set $\mathcal{P}$ of *predicates*, which serve as the basic building blocks of our invariants. The predicates are generated from generic templates, which are instantiated using all combinations of program variables that occur in the program being verified. Figure 3.3 presents these templates in detail.

Our templates cover a fairly exhaustive set of predicates. This includes properties of the store (equality of pointer variables, equality and inequalities between integer variables, etc.), shape properties (singly and doubly linked lists and list segments, sorted lists, trees, binary search trees, AVL trees, treaps, etc.), and recursive definitions that map data structures to numbers, involving arithmetic relationships and set relationships (keys/data stored in a structure, lengths of lists and list segments, height of trees). In addition, our templates include predicates describing heaplets of various structures, which involve set operations, disjointness, and equalities. The structures and predicates are extensible, of course, to any recursive definition expressible in Dryad.

The templates are grouped into three categories, roughly in increasing complexity. Predicates of Category 1 involve shape-related properties, predicates of Category 2 involve properties related to the keys stored in the data-structure, and predicates of Category 3 involve size-related properties (lengths of lists and heights of trees). Given a program to verify and its annotations, we choose the category of templates depending on whether the specification refers to shape only, shapes and keys, or shapes, keys, and sizes (choosing a category includes the predicates of lower category as well). Then, predicates are automatically generated by instantiating the templates with all (combinations of) program variables. This approach gives us a fairly fine-grained control over the set of predicates used in the verification process.

[4]Our prototype as well as the benchmarks used to reproduce the results presented below are publicly available on figshare [111].

$x, y \in \mathit{PointerVars} \quad \vec{x}, \vec{y}, \vec{z} \in \mathit{PointerVars}^* \quad \mathit{pf} \in \mathit{PointerFields} \quad \mathit{df} \in \mathit{DataFields}$
$i, j \in \mathit{IntegerVars} \cup \{0, \mathsf{IntMax}, \mathsf{IntMin}\}$

$$\begin{aligned}
\mathit{listshape}(\vec{x}) &\coloneqq \mathsf{LinkedList}(x_1) \mid \mathsf{DoublyLinkedList}(x_1) \mid \mathsf{SortedLinkedList}(x_1) \\
&\quad \mid \mathsf{LinkedListSeg}(x_1, x_2) \mid \mathsf{DoublyLinkedListSeg}(x_1, x_2) \\
\mathit{treeshape}(x) &\coloneqq \mathit{BST}(x) \mid \mathit{AVLtree}(x) \mid \mathit{Treap}(x) \\
\mathit{shape}(\vec{x}) &\coloneqq \mathit{listshape}(\vec{x}) \mid \mathit{treeshape}(\vec{x}) \\
\mathit{size}(\vec{x}) &\coloneqq \mathit{listshape}\_\mathsf{length}(\vec{x}) \mid \mathit{treeshape}\_\mathsf{height}(\vec{x})
\end{aligned}$$

Category 1

$x = \mathsf{nil}$	$x = y$
$x \neq \mathsf{nil}$	$x \neq y$
$\mathit{shape}(\vec{x})$	$x.\mathit{pf} = \mathsf{nil}$
$x \in \mathit{shape}\_\mathsf{heaplet}(\vec{y})$	$x.\mathit{pf} \neq \mathsf{nil}$
$x \notin \mathit{shape}\_\mathsf{heaplet}(\vec{y})$	$x.\mathit{pf} = y$
$\mathit{shape}\_\mathsf{heaplet}(\vec{x}) \cap \mathit{shape}\_\mathsf{heaplet}(\vec{y}) = \emptyset$	$x.\mathit{pf} \neq y$

Category 2

$i \in \mathit{shape\_key}\_\mathsf{set}(\vec{x})$	$x.\mathit{df} = i$
$i \notin \mathit{shape\_key}\_\mathsf{set}(\vec{x})$	$x.\mathit{df} \neq i$
$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) \leq_{\mathsf{set}} \{i\}$	$x.\mathit{df} \leq i$
$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) \geq_{\mathsf{set}} \{i\}$	$x.\mathit{df} \geq i$
$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) \leq_{\mathsf{set}} \{y.\mathit{df}\}$	$x.\mathit{df} = y.\mathit{df}$
$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) \geq_{\mathsf{set}} \{y.\mathit{df}\}$	$x.\mathit{df} \neq y.\mathit{df}$
$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) = \mathit{shape\_key}\_\mathsf{set}(\vec{y})$	$x.\mathit{df} \leq y.\mathit{df}$
$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) \leq_{\mathsf{set}} \mathit{shape\_key}\_\mathsf{set}(\vec{y})$	$x.\mathit{df} \geq y.\mathit{df}$
$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) \geq_{\mathsf{set}} \mathit{shape\_key}\_\mathsf{set}(\vec{y})$	
$\mathit{shape\_key}\_\mathsf{set}(\vec{x}) = \mathit{shape\_key}\_\mathsf{set}(\vec{y}) \cup \mathit{shape\_key}\_\mathsf{set}(\vec{z})$	

Category 3

$\mathit{size}(\vec{x}) = i - j$	$\mathit{size}(\vec{x}) = i$
$\mathit{size}(\vec{x}) - \mathit{size}(\vec{y}) = i$	$\mathit{size}(\vec{x}) \leq i$
$\mathit{size}(\vec{x}) - \mathit{size}(\vec{y}) = i - j$	$\mathit{size}(\vec{x}) \geq i$

Figure 3.3: Templates of Dryad predicates. The operator $\leq_{\mathsf{set}}$ denotes comparison between integer sets, where $A \leq_{\mathsf{set}} B$ if and only if $x \leq y$ holds for all $x \in A$ and $y \in B$. The operator $\geq_{\mathsf{set}}$ is similarly defined. Shape properties such as LinkedList, AVLtree, and so on are recursively defined in Dryad (not shown here) and are extensible to any class of Dryad-definable shapes. Similarly, the definitions related to keys stored in a data structure and the sizes of data structures also stem from recursive definitions in Dryad.

3.3.3 Benchmarks

We have evaluated our prototype on ten benchmark suits that contain standard algorithms on dynamic data structures, such as searching, inserting, or deleting items in lists and trees. These benchmarks were taken from the following sources:

1. GNU C Library(glibc) singly/sorted linked lists;
2. GNU C Library(glibc) doubly linked lists;
3. OpenBSD SysQueue;
4. GRASSHopper [112] singly linked lists;
5. GRASSHopper [112] doubly linked lists;
6. GRASSHopper [112] sorted linked lists;
7. VCDryad [68] sorted linked lists;
8. VCDryad [68] binary search trees, AVL trees, and treaps;
9. AFWP [113] singly/sorted linked lists; and
10. ExpressOS [114] MemoryRegion.

The specifications for these programs generally checks for their full functional correctness, such as preserving or altering shapes of data structures, inserting or deleting keys, filtering or finding elements, as well as sortedness of elements. The specifications, hence, involve separation logic with arithmetic as well as recursive definitions that compute numbers (such as lengths and heights), data-aggregating recursive functions (such as multisets of keys stored in the data structures), and complex combinations of these properties (e.g., to specify binary search trees, AVL trees, and treaps). All programs are annotated in Dryad, and checking validity of the resulting verification conditions is undecidable.

From these benchmark suites, we first picked all programs that contained iterative loops and erased the user-provided loop invariants. We also selected some programs that were purely recursive and where the contract for the function had manually been strengthened to make the verification succeed. We weakened these contracts to only state the specification (typically by removing formulas in the post-conditions of recursively called functions) and introduced annotation holes instead. The goal was to synthesize strengthenings of these contracts that allow proving the program correct. We also chose five straight-line programs, deleted their post-conditions, and evaluated whether our prototype was able to learn post-conditions for them (since our conjunctive learner learns the strongest invariant expressible as a conjunct, we can use our framework to synthesize post-conditions as well). In total, we obtained 82 routines.

After removing annotations from the benchmarks, we automatically inserted appropriate predicates over which to build invariants and contracts as described above. For all benchmark suits, conjunctions of these predicates were sufficient to prove the programs correct.

3.3.4 Experimental Results

We performed all experiments in a virtual machine running Ubuntu 16.04.1 on a single core of an Intel Core i7-7820 HK 2.9 GHz CPU with 2 GB memory. The box plots in Figure 3.4 summarize the results of this empirical evaluation aggregated by benchmark suite, specifically the time required to verify the programs, the number of automatically inserted base predicates (i.e., $|\mathcal{P}|$), and the number of iterations in the learning process. Each box in the diagrams shows the lower and upper quartile (left and right border of the box, respectively), the median (line within the box), as well as the minimum and maximum (left and right whisker, respectively).

Our prototype was successful in learning invariants and contracts for all 82 programs. The fact that the median time for a great majority of benchmark suits is less than $10\,s$ shows that our technique is extremely effective in finding inductive DRYAD invariants. We observe that despite many examples starting with hundreds of base predicates, which suggests a worst-case complexity of hundreds of iterations, the learner was able to learn with much fewer iterations and the number of predicates in the final invariant is small. This shows that non-provability information of our framework provide much more information than the worst-case suggests.

To the best of our knowledge, our prototype is currently the only tool able of fully automatically verifying these challenging benchmark suits. We must emphasize, however, that there are subsets of these benchmarks that can be verified by reformulating verification problem in decidable fragments of separation logic—we refer the reader to the related work in Section 3.5 for a survey of such work. Our goal in this evaluation, however, is not to compete with other, mature tools on a subset of benchmarks, but to measure the efficacy of our proposed CD-NPI-based invariant synthesis framework on the complete benchmark set.

3.4 LEARNING INVARIANTS IN THE PRESENCE OF BOUNDED QUANTIFIER INSTANTIATION

Software verification must deal with quantification in numerous application domains. For instance, quantifiers are often needed for axiomatizing theories that are not already equipped with decision procedures, for specifying properties of unbounded data structures and dynamically allocated memory, as well as for defining recursive properties of programs. For instance, the power of two function can be axiomatized using quantifiers:

Benchmark suite	1	2	3	4	5	6	7	8	9	10
Number of programs	16	9	3	8	8	11	3	11	9	4
Max. category of templates	3	3	1	1	1	2	2	3	2	1

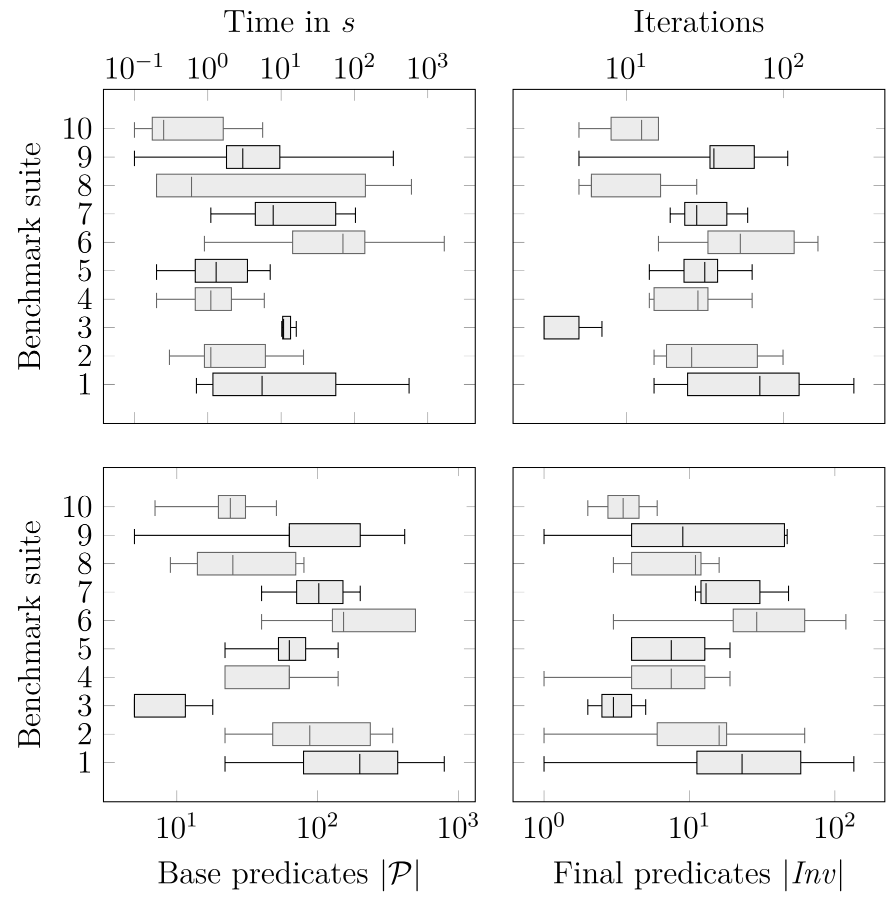


Figure 3.4: Aggregated experimental results of the Dryad benchmarks.

$$pow_2(0) = 1 \quad \wedge \quad \forall n \in \mathbb{N} \colon n > 0 \Rightarrow pow_2(n) = 2 \cdot pow_2(n-1). \tag{3.3}$$

Despite the fact that various important first-order theories are undecidable (e.g., the first-order theory of arithmetic with uninterpreted functions), modern SMT solvers implement a host of heuristics to cope with quantifier reasoning. Quantifier instantiation, including pattern-based quantifier instantiation (e.g., E-matching [103]) and model-based quantifier instantiation [104], are particularly effective heuristics in this context. The key idea of instantiation-based heuristics is to instantiate universally quantified formulas with a finite number of ground terms and then check for validity of the resulting quantifier-free formulas (whose theory needs to be decidable). The exact instantiation of ground terms varies from method to method, but most instantiation-based heuristics are necessarily incomplete in general due to the undecidability of the underlying decision problems.

We can apply invariant synthesis framework for verification engines that employ quantifier instantiation in the following way. Assume that $\mathcal{U}$ is an undecidable first-order theory allowing uninterpreted functions and that $\mathcal{D}$ is its decidable quantifier-free fragment. Then, quantifier instantiation can be seen as a transformation of a $\mathcal{U}$-formula φ (potentially containing quantifiers) into a $\mathcal{D}$-formula $\mathit{approx}(\varphi)$ in which all existential quantifiers have been eliminated (e.g., using skolemization) and all universal quantifiers have been replaced by finite conjunctions over ground terms.[5] This means that if the $\mathcal{D}$-formula $\mathit{approx}(\varphi)$ is valid, then the $\mathcal{U}$-formula φ is valid as well. On the other hand, if $\mathit{approx}(\varphi)$ is not valid, one cannot deduce any information about the validity of φ. However, a $\mathcal{D}$-model of $\mathit{approx}(\varphi)$ can be used to derive non-provability information as described in Section 3.2.1.

3.4.1 Benchmarks

Our benchmark suite consists of twelve slightly simplified programs from IronFleet [115] (provably correct distributed systems), the Verified Software Competition [110], ExpressOS [114] (a secure operating system for mobile devices), and tools for sparse matrix multiplication [116]. In these programs, quantifiers are used to specify recursively defined predicates, such as $\mathit{power}(n, m)$ and $\mathit{sum}(n)$, as well various array properties, such as no duplicate elements, periodic properties of array elements, and bijective (injective and surjective) maps. All these specifications are undecidable in general. In particular, the array specifications fall outside of the decidable array property fragment [117] because they involve strict comparison between universally quantified index variables, array accesses in the index guard, nested array accesses (e.g., $a_1[a_2[i]]$), arithmetic expressions over universally quantified index variables, and alternation of universal and existential quantifiers.

From this benchmark suite, we erased the user-defined loop invariants and generated a set of predicates that serve as the building blocks of our invariants. To this end, we used the pre/post-conditions of the program being verified as templates from which the actual predicates were generated—as in the case of Dryad benchmarks, the templates were instantiated using all combinations of program variables that occur in the program. Additionally, we generated predicates for octagonal constraints over the integer variables in the programs (i.e., relations between two integer variables of the form $\pm x \pm y \leq c$). For programs involving arrays, we also generated octagonal constraints over array access expressions that appear in the program.

[5] Quantifier instantiation is usually performed iteratively, but we here abstract away from this fact.

Program	$\lvert\mathcal{P}\rvert$	# Iterations	$\lvert Inv\rvert$	Time in s
inverse	414	126	73	9.04
power2	109	55	34	2.10
powerN	160	60	31	13.52
recordArraySplit	1264	49	51	57.46
recordArrayUnzip	222	17	25	0.84
removeDuplicates	280	67	86	4.43
setFind	492	74	136	2.76
setInsert	556	73	188	6.70
sparseMatrixGen	816	278	90	22.07
sparseMatrixMul	768	313	91	14.49
sum	128	40	22	1.02
sumMax	192	61	45	4.31

Table 3.1: Experimental results of the quantifier instantiation benchmarks. The column $\lvert\mathcal{P}\rvert$ refer to the number of automatically inserted base predicates, the column *# Iterations* to the number of iterations of the teacher and learner, and the column $\lvert Inv\rvert$ to the number of predicates in the inferred invariant.

3.4.2 Experimental Results

We have implemented a prototype based on Boogie [71] and Z3 [88] as the verification engine and Houdini [70] as a conjunctive ICE learning algorithm. As in the case of the Dryad benchmarks, all experiments were conducted in a virtual machine running Ubuntu 16.04.1 on a single core of an Intel Core i7-7820 HK 2.9 GHz CPU with 2 GB memory. The results of these experiments are listed in Table 3.1.

As can be seen from this table, our prototype was effective in finding inductive invariants and was able to prove each program correct in less than one minute (in 75 % of the programs in less than 10 s). Despite having hundreds of base predicates in many examples, which in turn suggests a worst-case complexity of hundreds of rounds, the learner was able to learn an inductive invariant with much fewer rounds. As in the case of the Dryad benchmarks, the non-provability information provided by the verification engine provided much more information than the worst-case suggests.

3.5 RELATED WORK

Techniques for invariant synthesis include abstract interpretation [118], interpolation [107], IC3 [108], predicate abstraction [119], abductive inference [120], as well as synthesis algorithms that rely on constraint solving [121, 122, 123]. Complementing them are data-driven invariant synthesis approaches that are based on machine learning. Examples include techniques that learn likely invariants, such as Daikon [124], and techniques that learn inductive invariants, such as HOUDINI [70], ICE [45], and Horn-ICE [125, 126]. The latter typically requires a teacher that can generate counter-examples if the conjectured invariant is not adequate or inductive. Classically, this is possible only when the verification conditions of the program fall in decidable logics. In this chapter, we investigate data-driven invariant synthesis for incomplete verification engines and show that the problem can be reduced to ICE learning if the learning algorithm learns from non-provability information and produces hypotheses in a class that is restricted to positive Boolean formulas over a fixed set of predicates. Data-driven synthesis of invariants has regained recent interest [45, 47, 67, 127, 128, 129, 130, 131, 132, 133], and our work addresses an important problem of synthesizing invariants for programs whose verification conditions fall in undecidable fragments.

Our application to learning invariants for heap-manipulating programs builds upon DRYAD [68, 69], and the natural proof technique line of work for heap verification developed by Qiu et al. Techniques, similar to DRYAD, for automated reasoning of dynamically manipulated data structure programs have also been proposed in [106, 134]. However, unlike our current work, none of these works synthesize heap invariants. Given invariant annotations in their respective logics, they provide procedures to validate if the verification conditions are valid. There has also been a lot of work on synthesizing invariants for separation logic using shape analysis [135, 136, 137]. However, most of these techniques are tailored for memory safety and shallow properties rather than rich properties that check full functional correctness of data structures. Interpolation has also been suggested recently to synthesize invariants involving a combination of data and shape properties [138]. It is, however, not clear how the technique can be applied to a more complicated heap structure, such as an AVL tree, where shape and data properties are not cleanly separated but are intricately connected. Recent work also includes synthesizing heap invariants in the logic from [113] by extending IC3 [139, 140].

In this work, our learning algorithm synthesizes invariants over a fixed set of predicates. When all programs belong to a specific class, such as the class of programs manipulating data structures, these predicates can be uniformly chosen using templates. Investigating automated ways for discovering candidate predicates is a very interesting future direction. Related work in this direction includes recent works [132, 133].

3.6 CONCLUSIONS AND FUTURE WORK

We have presented a learning-based framework for invariant synthesis in the presence of sound but incomplete verification engines. To prove that our technique is effective in practice, we have successfully applied it two important and challenging verification setting: verifying heap-manipulating programs against specifications expressed in an expressive and undecidable dialect of separation logic and verifying programs against specifications with universal quantification. In particular for the former setting, we are not aware of any other technique that can handle our extremely challenging benchmark suite.

Several future research directions are interesting. First, the framework we have developed is based on the principle of counterexample-guided inductive synthesis, where the invariant synthesizer synthesizes invariants using non-provability information but does not directly work on the program's structure. It would be interesting to extend white-box invariant generation techniques such as interpolation/IC3/PDR, working using $\mathcal{D}$ (or $\mathcal{B}$) abstractions of the program directly in order to synthesize invariants for them. Second, in the CD-NPI learning framework we have put forth, it would be interesting to change the underlying logic of communication $\mathcal{B}$ to a richer logic, say the theory of arithmetic and uninterpreted functions. The challenge here would be to extract non-provability information from the models to the richer theory, and pairing them with synthesis engines that synthesize expressions against constraints in $\mathcal{B}$. Finally, we think invariant learning should also include *experience* gained in verifying other programs in the past, both manually and automatically. A learning algorithm that combines logic-based synthesis with experience and priors gained from repositories of verified programs can be more effective.

CHAPTER 4: SORCAR: PROPERTY-DRIVEN ALGORITHMS FOR LEARNING CONJUNCTIVE INVARIANTS

In this chapter, we present a novel ICE learning algorithm to learn conjunctive inductive invariants over a fixed finite set of predicates $\mathcal{P}$. Houdini [70] is an existing learner to learn conjunctive inductive invariants, and synthesizes the *tightest* inductive invariant. Consequently, it can ignore the property to be proven about the program. However, the tightest invariant can be quite complex (have many conjuncts) and hard to synthesize.

We present Sorcar, a property driven learning algorithm for conjunctive inductive invariants and performs better than existing Houdini based tools on certain classes of benchmarks. Intuitively, Sorcar grows slowly a set of relevant predicates $R \subseteq \mathcal{P}$ in each round and proposes the tightest conjunctive invariant over R. It guarantees convergence to a conjunctive invariant (if one exists over $\mathcal{P}$) in $2|\mathcal{P}|$ rounds of communication with any verification oracle.

In particular, in this chapter:

- The Sorcar learning algorithm is a general conjunctive ICE learning algorithm that can be used in many program verification algorithms.
- We use it in two settings: one for GPUVerify programs and the others where validation of verification conditions is incomplete (heap verification).
- The notion of counterexamples are either concrete states or non-provability information, respectively.

4.1 INTRODUCTION

We present a new class of learning algorithms, Sorcar, to synthesize conjunctive inductive invariants, for proving that a program satisfies its assertions, and hence is property-driven.

While one can potentially learn/synthesize invariants in complex logics, one technique that has been particularly effective and scalable is to fix a finite set of predicates $\mathcal{P}$ over the program configurations and only learn inductive invariants that can be expressed as a conjunction of predicates over $\mathcal{P}$. For particular domains of programs and types of specifications, it is possible to identify classes of candidate predicates that are typically involved in invariants (e.g., based on the code of the programs and/or the specification), and learning invariants over such a class of predicates has proven very effective. A prominent example is device

Learning algorithm	**Property driven?**	**Complexity per round**	**Maximum # rounds**	**Final conjunct**
HOUDINI	No	Polynomial	$\lvert\mathcal{P}\rvert$	Largest set
SORCAR	Yes	Polynomial	$2 \cdot \lvert\mathcal{P}\rvert$	Bias towards weaker invariants (smaller sets of conjunctions) involving only relevant predicates

Table 4.1: Comparison of HOUDINI [70] and SORCAR

drivers, and Microsoft's Static Driver Verifier [141, 142] (specifically the underlying tool CORRAL [143]) is an industry-strength tool that leverages exactly this approach.

The classical algorithm for learning conjunctive invariants over a finite class of predicates is the HOUDINI algorithm [70], which mimics the *elimination algorithm* for learning conjuncts in classical machine learning [78]. HOUDINI starts with a conjectured invariant that contains *all* predicates in $\mathcal{P}$ and, in each round, uses information from a failed verification attempt to remove predicates. The most salient aspect of the algorithm is that it is guaranteed to converge to a conjunctive invariant, if one exists, in $n = |\mathcal{P}|$ rounds (which is logarithmic in the number of invariants, as there are 2^n of them). However, the HOUDINI algorithm has disadvantages as well. Most notably, it is not property-driven as it does not consider the assertions that occur in the program (which is a consequence of the fact that it was originally designed to infer invariants of unannotated programs). Secondly, HOUDINI synthesizes invariants that have the *largest* number of conjuncts (i.e., the semantically smallest sets of program configurations expressible as a conjunctive formula). In fact, one can view the HOUDINI algorithm as a way of computing the least fixed point in the abstract interpretation framework, where the abstract domain consists of conjunctions over the candidate predicates.

The primary motivation to build a property-driven learning algorithm is to explore invariant generation techniques that can be potentially more efficient in proving programs correct. The SORCAR algorithm has the following design features (also see Table 4.1). First, it is property-driven, in other words, the algorithm tries to find conjunctive inductive invariants that are sufficient to prove the assertions in the program. By contrast, HOUDINI computes the *tightest* inductive invariant. Since SORCAR is property-driven, it can find weaker inductive invariants (i.e., invariants with fewer conjuncts). Our intuition is that by synthesizing weaker, property-driven invariants, we can verify programs more efficiently. Second, SORCAR guarantees that the number of rounds of interaction with the teacher is still linear ($2n$ rounds compared to HOUDINI's promise of n rounds). Third, SORCAR promises to do only polynomial amount of work in each round (i.e., polynomial in n and in the number of current counterexamples), similar to HOUDINI.

The SORCAR algorithm works, intuitively, by finding conjunctive invariants over a set of *relevant predicates* $R \subseteq \mathcal{P}$. This set is grown slowly (but monotonically, as monotonic growth is crucial to ensure that the number of rounds of learning is linear) by adding predicates only when they were found to be relevant to prove assertions. More specifically, predicates are considered relevant based on information gained from counterexamples of failed verification conditions that involve assertions in the program. The precise mechanism of growing the set of relevant predicates can vary, and we define four variants of SORCAR (e.g., choosing all predicates that show promise of relevance or greedily choosing a minimal number of relevant predicates). The SORCAR suite of algorithms is hence a new class of property-driven learning algorithms for conjunctive invariants with different design principles.

The SORCAR algorithm is certainly well suited for applications where property-driven invariants are expected to be small and one wishes to learn small invariants. Learning small invariants can be particularly useful in applications where these algorithms are used to *mine specifications* that a user may read. For example, if we use SORCAR to mine provable contracts for methods, then contracts would be easier to read if they are smaller (i.e., contain fewer predicates). However, even when invariants are not required to be small, SORCAR, being property-driven, can be more effective.

We evaluated the efficiency of SORCAR on two domains of benchmarks, where learning conjunctive invariants is very effective. The first is the class of programs handled by GPUVerify [72, 73], which considers GPU programs, reduces the problem to a sequential verification problem (by simulating two threads at each parallel fork), and proceeds to find conjunctive invariants over a fixed set $\mathcal{P}$ of predicates to prove the resulting sequential program correct. The second class of programs dynamically update heaps against specifications in separation logic and requires synthesizing invariants to prove their correctness. The verification oracle in the former is an SMT solver that returns concrete Horn-ICE counterexamples. In the latter, predicates involve *inductively defined relations* (such as a list-segment, the heaplet associated with it, or the set of keys stored in it), and validating verification conditions is undecidable in general. Hence, the verification oracle is sound but incomplete (based on "natural proofs") and returns non-provability information as counterexamples that can be transformed into ICE counterexamples. In both domains, the set $\mathcal{P}$ consists of hundreds of candidate predicates, which makes invariant synthesis challenging (as there are $2^{|\mathcal{P}|}$ possible conjunctive invariants).

We have implemented SORCAR on top of the BOOGIE program verifier [71] and have applied it to verify both GPU programs for data races [72, 73] and heap manipulating programs against separation logic specifications [74]. To assess the performance of Sorcar, we have compared it to the current state-of-the-art tools for these programs, which use the Houdini algorithm. Though Sorcar did not work more efficiently on every program, our empirical evaluation shows that it is overall more competitive than Houdini. In summary, we found that (a) Sorcar produces much smaller invariants, (b) Sorcar worked more efficiently overall in verifying these programs, and (c) Sorcar verified a larger number of programs than Houdini did (for a suitably large timeout).

4.2 BACKGROUND

In this section, we provide the background on learning-based invariant synthesis. In particular, we briefly recapitulate the Horn-ICE learning framework (in Section 4.2.1) and discuss the Houdini algorithm (in Section 4.2.2), specifically in the context of the Horn-ICE framework. Note that in this chapter, we use the term ICE and Horn-ICE interchangeably as the Horn-ICE framework can be seen as an extension of the ICE framework [126].

To make the Horn-ICE framework mathematically precise, let P be the program (with assertions) under consideration and $\mathcal{C}$ the set of all program configurations of P. Furthermore, let us fix a finite set $\mathcal{P}$ of *predicates* $p\colon \mathcal{C} \to \mathbb{B}$ over the program configurations, where $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ is the set of Boolean values. These predicates capture interesting properties of the program and serve as the basic building blocks for constructing invariants. We assume that the values of these predicates can either be obtained directly from the program configurations or that the program is instrumented with ghost variables that track the values of the predicates at important places in the program (e.g., at the loop header and immediately after the loop). As notational convention, we write $c \models p$ if $p(c) = \mathit{true}$ and $c \not\models p$ if $p(c) = \mathit{false}$. Moreover, we lift this notation to formulas φ over $\mathcal{P}$ (i.e., arbitrary Boolean combinations of predicates from $\mathcal{P}$) and use $c \models \varphi$ ($c \not\models \varphi$) to denote that c satisfies φ (c does not satisfy φ).

To simplify the presentation in the remainder of this chapter, we use conjunctions $p_1 \wedge \cdots \wedge p_n$ of predicates over $\mathcal{P}$ and the corresponding sets $\{p_1, \ldots, p_n\} \subseteq \mathcal{P}$ interchangeably. In particular, for a (sub-)set $X = \{p_1, \ldots, p_n\} \subseteq \mathcal{P}$ of predicates and a program configuration $c \in \mathcal{C}$, we write $c \models X$ if and only if $c \models p_1 \wedge \cdots \wedge p_n$.

4.2.1 The Horn-ICE Learning Framework

The Horn-ICE learning framework [125, 126] is a general framework for learning inductive invariants in a black-box setting. We here assume without loss of generality that the task is

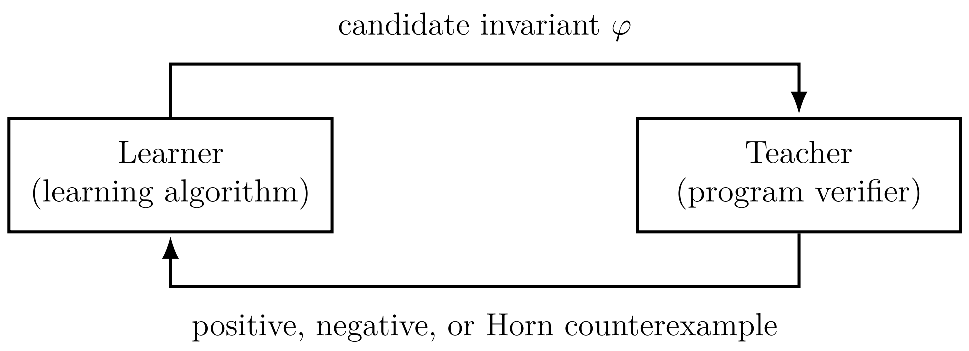

candidate invariant φ
Learner
(learning algorithm)
Teacher
(program verifier)
positive, negative, or Horn counterexample

Figure 4.1: The Horn-ICE learning framework [125, 126]

to synthesize a single invariant. In the case of learning multiple invariants, say at different program locations, one can easily expand the given predicates to predicates of the form $(pc = l) \rightarrow p$ where pc refers to the program counter, l is the location of an invariant in the program, and $p \in \mathcal{P}$. Learning a conjunctive invariant over this extended set of predicates then corresponds to learning multiple conjunctive invariants at the various locations.

As sketched in Figure 4.1, the Horn-ICE framework consists of two distinct entities—the *learner* and the *teacher*—and proceeds in rounds. In each round, the teacher receives a candidate invariant φ from the learner and checks whether φ proves the program correct. Should φ not be adequate to prove the program correct, the learner replies with a counterexample, which serves as a means to correct inadequate invariants and guide the learner towards a correct one. More precisely, a counterexample takes one of three forms:[1]

- If the pre-condition α of the program does not imply φ, then the teacher returns a *positive counterexample* $c \in \mathcal{C}$ such that $c \models \alpha$ but $c \not\models \varphi$.
- If φ does not imply the post-condition β of the program, then the teacher returns a *negative counterexample* $c \in \mathcal{C}$ such that $c \models \varphi$ but $c \not\models \beta$.
- If φ is not inductive, then the teacher returns a *Horn counterexample* $(\{c_1, \ldots, c_n\}, c) \in 2^{\mathcal{C}} \times \mathcal{C}$ such that $c_i \models \varphi$ for each $i \in \{1, \ldots, n\}$ but $c \not\models \varphi$. (We encourage the reader to think of Horn counterexamples as constraints of the form $(c_1 \wedge \cdots \wedge c_n) \rightarrow c$.)

A teacher who returns counterexamples as described above always enables the learner to make *progress* in the sense that every counterexample it returns is inconsistent with the current conjecture (i.e., it violates the current conjecture). Moreover, the Horn-ICE framework requires the teacher to be *honest*, meaning that each counterexample needs to be

[1]By abuse of notation, we write $c \models \alpha$ ($c \not\models \alpha$) to denote that c satisfies (violates) the formula α even if α contains predicates that do not belong to $\mathcal{P}$.

consistent with *all* inductive invariants that prove the program correct (i.e., the teacher does not rule out possible solutions). Finally, note that such a teacher can indeed be built since program verification can be stated by means of *constrained Horn clauses* [144]. When the candidate invariant does not make such clauses true, some Horn clause failed, and the teacher can find a Horn counterexample using a logic solver (positive counterexamples arise when the left-hand-side of the Horn counterexample is empty, while negative counterexamples arise when the left-hand-side has one element and the-right-hand side is *false*).

The objective of the learner, on the other hand, is to construct a formula φ over $\mathcal{P}$ from the counterexamples received thus far. For the sake of simplicity, we assume that the learner collects all counterexamples in a data structure $\mathcal{S} = (S_+, S_-, S_H)$, called *Horn-ICE sample*, where

1. $S_+ \subseteq \mathcal{C}$ is a finite set of positive counterexamples;
2. $S_- \subseteq \mathcal{C}$ is a finite set of negative counterexamples; and
3. $S_H \subseteq 2^{\mathcal{C}} \times \mathcal{C}$ is a finite set of Horn counterexamples.

To measure the complexity of a sample, we define its *size*, denoted by $|\mathcal{S}|$, to be $|S_+| + |S_-| + \sum_{(L,c) \in S_H} (|L| + 1)$.

Given a Horn-ICE sample $\mathcal{S} = (S_+, S_-, S_H)$, the learner's task is then to construct a formula φ over $\mathcal{P}$ that is *consistent* with $\mathcal{S}$ in that

1. $c \models \varphi$ for each $c \in S_+$;
2. $c \not\models \varphi$ for each $c \in S_-$; and
3. for each $(\{c_1, \ldots, c_n\}, c) \in S_H$, if $c_i \models \varphi$ for all $i \in \{1, \ldots, n\}$, then $c \models \varphi$.

This task is called *passive Horn-ICE learning*, while the overall learning setup can be thought of as *iterative (or online) Horn-ICE learning*. In the special case that the learner produces conjunctive formulas, we say that a set $X \subseteq \mathcal{P}$ is consistent with $\mathcal{S}$ if and only if the corresponding conjunction $\bigwedge_{p \in X} p$ is consistent with $\mathcal{S}$.

In general, the Horn-ICE learning framework permits arbitrary formulas over the predicates as candidate invariants. In this chapter, however, we exclusively focus on conjunctive formulas (i.e., conjunctions of predicates from $\mathcal{P}$). In fact, conjunctive invariants form an important subclass in practice as they are sufficient to prove many programs correct [70, 74] (also see our experimental evaluation in Section 4.4). Moreover, one can design efficient learning algorithms for conjunctive Boolean formulas, as we show next.

4.2.2 Houdini as a Horn-ICE Learning Algorithm

Houdini [70] is a popular algorithm to synthesize conjunctive invariants in interaction with a theorem prover. For our purposes, however, it is helpful to think of Houdini as an adaptation of the classical elimination algorithm [78] to the Horn-ICE learning framework that is modified to account for Horn counterexamples. To avoid confusion, we refer to algorithmic component that the Houdini learning algorithm as the *"elimination algorithm"* and the implementation of the elimination algorithm as a learner in the context of the Horn-ICE framework as Houdini-ICE.

Let us now describe the elimination algorithm as it is used in the design of Sorcar as well. Given a Horn-ICE sample $\mathcal{S} = (S_+, S_-, S_H)$, the elimination algorithm computes the largest conjunctive formula $X \subseteq \mathcal{P}$ in terms of the number of predicates in X (i.e., the semantically smallest set of program configurations expressible by a conjunctive formula) that is consistent with $\mathcal{S}$. Starting with the set $X = \mathcal{P}$ of all predicates, the elimination algorithm proceeds as follows:

1. The elimination algorithm removes all predicates $p \in X$ from X that violate a positive counterexample (i.e., there exists a positive counterexample $c \in S_+$ such that $c \not\models p$). The result is the unique largest set X of predicates—alternatively the largest conjunctive formula—that is consistent with S_+ (i.e., $c \models X$ for all $c \in S_+$).

2. The elimination algorithm checks whether all Horn counterexamples are satisfied. If a Horn counterexample $(\{c_1, \ldots, c_n\}, c) \in S_H$ is not satisfied, it means that each program configuration c_i of the left-hand-side satisfies X, but the configuration c on the right-hand-side does not. However, X corresponds to the semantically smallest set of program configurations expressible by a conjunctive formula that is consistent with S_+. Moreover, all program configurations c_i on the left-hand-side of the Horn counterexample also satisfy X. Thus, the right-hand-side c necessarily has to satisfy X as well (otherwise X would not satisfy the Horn counterexample). To account for this, the elimination algorithm adds c as a new positive counterexample to S_+.

3. The elimination algorithm repeats Steps 1 and 2 until a fixed point is reached. Once this happens, X is the unique largest set of predicates that is consistent with S_+ and S_H.

Finally, the elimination algorithm checks whether each negative counterexample violates X (i.e., $c \not\models X$ for each $c \in S_-$). If this is the case, X is the largest set of predicates that is consistent with $\mathcal{S}$; otherwise, no consistent conjunctive formula exists. Note that the elimination algorithm does not learn from negative counterexamples.

It is not hard to verify that the time the elimination algorithm spends in each round is *polynomial* in the number of predicates and the size of the Horn-ICE sample (provided predicates can be evaluated in constant time). If the elimination algorithm is employed in the iterative Horn-ICE setting (as Houdini-ICE), it is guaranteed to converge in at most $|\mathcal{P}|$ rounds, or it reports that no conjunctive invariant over $\mathcal{P}$ exists.

The property that Houdini-ICE converges in at most $|\mathcal{P}|$ rounds is of great importance in practice. One can, for instance, in every round learn the *smallest* set of conjuncts satisfying the sample, say using a SAT solver. Doing so would not significantly increase the time taken for learning in each round (thanks to highly-optimized SAT solvers), but the worst-case number of iterations to converge to an invariant becomes exponential. An exponential number of rounds, however, makes learning invariants often intractable in practice (we implemented such a SAT-based learner, but it performed poorly on our set of benchmarks). Hence, it is important to keep the number of iterations small when learning invariants. Note that Houdini-ICE does not use *negative examples* to learn formulas and, hence, is *not property-driven* (negative examples come from configurations that lead to violating assertions). The Sorcar algorithm, which we describe in the next section, has this feature and aims for potentially weaker invariants that are sufficient to prove the assertions in the program. Note, however, that Houdini-ICE is complete in the sense that it is guaranteed to find an inductive invariant that proves the program correct against its assertions, if one exists that can be expressed as a conjunction over the given predicates.

4.3 THE SORCAR HORN-ICE LEARNING ALGORITHM

One disadvantage of Houdini-ICE is that it learns in each round the largest set of conjuncts, *independent* of negative counterexamples, and, hence, independent of the assertions and specifications in the program—in fact, it learns the semantically smallest inductive invariant expressible as a set of conjuncts over $\mathcal{P}$. As a consequence, Houdini-ICE may spend a lot of time finding the tightest invariant (involving many predicates) although a simpler and weaker invariant suffices to prove the program correct. This motivates the development of our novel Sorcar Horn-ICE learning algorithm for conjuncts, which is property-driven (i.e., it also considers the assertions in the program) and has a bias towards learning conjunctions with a smaller number of predicates.

The salient feature of Sorcar is that it always learns invariants involving what we call *relevant* predicates, which are predicates that have shown some evidence to affect the assertions in the program. More precisely, we say that a predicate is *relevant* if it evaluates to *false* on some negative counterexample or on a program configuration appearing on the left-hand-side of a Horn counterexample. This indicates that *not* assuming this predicate leads to an assertion violation or the invariant not being inductive, and is hence deemed important as a candidate predicate in the synthesized invariant. However, naively choosing relevant predicates does, in general, lead to an exponential number of rounds. Thus, Sorcar is designed to select relevant predicates carefully and requires at most $2|\mathcal{P}|$ rounds to converge to an invariant (which is twice the number that Houdini-ICE guarantees). Moreover, the set of predicates learned by Sorcar is always a subset of those learned by Houdini-ICE.

Algorithm 4.1 presents the Sorcar Horn-ICE learner in pseudo code. In contrast to Houdini-ICE, it is not a purely passive learning algorithm but is divided into a passive part (`Sorcar-Passive`) and an iterative part (`Sorcar-Iterative`), the latter being invoked in every round of the Horn-ICE framework. More precisely, `Sorcar-Iterative` maintains a state in form of a set $R \subseteq \mathcal{P}$ in the course of the iterative learning, which is empty in the beginning and used to accumulate *relevant predicates* (Line 19). The exact choice of relevant predicates, however, is delegated to an external function `Relevant-Predicates`. We treat this function as a parameter for the Sorcar algorithm and discuss four possible implementations at the end of this section. Let us now present Sorcar in detail.

4.3.1 The Passive Sorcar Algorithm

Given a Horn-ICE sample $\mathcal{S}$ and a set $R \subseteq \mathcal{P}$, `Sorcar-Passive` first constructs the largest conjunction $X \subseteq \mathcal{P}$ that is consistent with $\mathcal{S}$ (Line 5). This construction follows the elimination algorithm described in Section 4.2.2 and ensures that X is consistent with all counterexamples in $\mathcal{S}$. Since X is the largest set of predicates consistent with $\mathcal{S}$, it represents the smallest consistent set of program configurations expressible as a conjunction over $\mathcal{P}$. As a consequence, it follows that $X \cap R$—in fact, any subset of X—is consistent with S_+. However, $X \cap R$ might not be consistent with S_- or S_H. To fix this problem, `Sorcar-Passive` collects all inconsistent negative counterexamples in a set N and all inconsistent Horn counterexamples in a set H (Lines 7 to 14). Based on these two sets, `Sorcar-Passive` then computes a set of relevant predicates, which it adds to R (Line 15). As mentioned above, the exact computation of relevant predicates is delegated to a function `Relevant-Predicates`, which we treat as a parameter. The result of this function is a set $R' \subseteq \mathcal{P}$ of predicates that needs to contain at least one new predicate that is not yet present in R. Once such a set has been computed and added to R, the process repeats (R grows monotonically larger) until a consistent conjunctive formula is found. Then, `Sorcar-Passive` returns both the conjunction $X \cap R$ as well as the new set R of relevant predicates. Note that the resulting conjunction is always a subset of the relevant predicates.

Algorithm 4.1: The SORCAR Horn-ICE learning algorithm

```
Function Relevant-Predicates(N, H, X, R):
    return a set of $R' \subseteq \mathcal{P}$ of relevant predicates such that $R' \setminus R \neq \emptyset$;
end

Procedure Sorcar-Passive($\mathcal{S} = (S_+, S_-, S_H)$, R):
    Run the elimination algorithm to compute the set $X = \{p_1, \ldots, p_n\}$, corresponding to
      the largest conjunctive formula $\bigwedge_{i=1}^{n} p_i$ over $\mathcal{P}$ that is consistent with $\mathcal{S}$ (abort if no
      such formula exists);
    while $X \cap R$ is not consistent with $\mathcal{S}$ do
        $N \leftarrow \emptyset$;                    // Stores inconsistent negative counterexamples
        $H \leftarrow \emptyset$;                       // Stores inconsistent Horn counterexamples
        foreach negative counterexample $c \in S_-$ not consistent with $X \cap R$ do
            $N \leftarrow N \cup \{c\}$;
        end
        foreach Horn counterexample $(L, c) \in S_H$ not consistent with $X \cap R$ do
            $H \leftarrow H \cup \{(L, c)\}$;
        end
        $R \leftarrow R \cup$ Relevant-Predicates(N, H, X, R);
    end
    return $(X \cap R, R)$;
end

static $R \leftarrow \emptyset$;                       // Stores relevant predicates across rounds
Procedure Sorcar-Iterative($\mathcal{S}$):
    $(Y, R) \leftarrow$ Sorcar-Passive($\mathcal{S}, R$);
    return $Y$;
end
```

The condition of the loop in Line 6 immediately shows that the set $X \cap R$ is consistent with the Horn-ICE sample $\mathcal{S}$ once `Sorcar-Passive` terminates. The termination argument, however, is less obvious. To argue termination, we first observe that X is consistent with each positive counterexample in S_+ and, hence, $X \cap R$ remains consistent with all positive counterexamples during the run of `Sorcar-Passive`. Next, we observe that the termination argument is independent of the exact choice of predicates added to R—in fact, the predicates need not even be relevant in order to prove termination of `Sorcar-Passive`. More precisely, since the function `Relevant-Predicates` is required to return a set $R' \subseteq \mathcal{P}$ that contains at least one new (relevant) predicate not currently present in R, we know that R grows strictly monotonically. In the worst case, the loop in Lines 6 to 16 repeats $|\mathcal{P}|$ times until $R = \mathcal{P}$; then, $X \cap R = X$, which is guaranteed to be consistent with $\mathcal{S}$ by construction of X (see Line 5). Depending on the implementation of `Relevant-Predicates`, however, `Sorcar-Passive` can terminate earlier with a much smaller consistent set $X \cap R \subsetneq X$. Since the time spent in each iteration of the loop in Lines 6 to 16 is proportional to $|\mathcal{P}| \cdot |\mathcal{S}| + f(|\mathcal{S}|)$, where f is a function capturing the complexity of `Relevant-Predicates`, the overall runtime of `Sorcar-Passive` is in $\mathcal{O}\Big(|\mathcal{P}|^2 \cdot |\mathcal{S}| + |\mathcal{P}| \cdot f(|\mathcal{S}|)\Big)$. This is summarized in the following theorem.

Theorem 4.1 (Passive Sorcar algorithm)**.** *Given a Horn-ICE sample $\mathcal{S}$ and a set $R \subseteq \mathcal{P}$ of relevant predicates, the passive* Sorcar *algorithm learns a consistent set of predicates (i.e., a consistent conjunction over $\mathcal{P}$) in time $\mathcal{O}\Big(|\mathcal{P}|^2 \cdot |\mathcal{S}| + |\mathcal{P}| \cdot f(|\mathcal{S}|)\Big)$ where f is a function capturing the complexity of the function* `Relevant-Predicates`*.*

Before we continue, let us briefly mention that the set of predicates returned by Sorcar is always a subset of those returned by Houdini-ICE.

4.3.2 The Iterative Sorcar Algorithm

`Sorcar-Iterative` maintains a state in form of a set $R \subseteq \mathcal{P}$ of relevant predicates in the course of the learning process (Line 19). In each round of the Horn-ICE learning framework, the learner invokes `Sorcar-Iterative` with the current Horn-ICE sample $\mathcal{S}$ as input, which contains all counterexamples that the learner has received thus far. Internally, `Sorcar-Iterative` calls `Sorcar-Passive`, updates the set R, and returns a new conjunctive formula, which the learner then proposes as new candidate invariant to the teacher. If `Sorcar-Passive` aborts (because no conjunctive formula over $\mathcal{P}$ that is consistent with $\mathcal{S}$ exists), so does `Sorcar-Iterative`.

To ease the presentation in the remainder of this section, let us assume that the program under consideration can be proven correct using an inductive invariant expressible as a conjunction over $\mathcal{P}$. Under this assumption, the iterative Sorcar algorithm identifies such an inductive invariant in at most $2|\mathcal{P}|$ rounds, as stated in the following theorem.

Theorem 4.2 (Iterative Sorcar algorithm)**.** *Let P be a program and $\mathcal{P}$ a finite set of predicates over the configurations of P. When paired with an honest teacher that enables progress, the iterative* Sorcar *algorithm learns an inductive invariant (in the form of a conjunctive formula over $\mathcal{P}$) that proves the program correct in at most $2|\mathcal{P}|$ rounds, provided that such an invariant exists.*

Proof. We first observe that the computation of the set X in Line 5 of `Sorcar-Passive` always succeeds. This is a direct consequence of the honesty of the teacher (see Section 4.2.1) and the assumption that at least one inductive invariant exists that is expressible as a conjunction over $\mathcal{P}$. This observation is essential as it shows that `Sorcar-Iterative` does not abort.

Next, recall that the teacher enables progress in the sense that every counterexample is inconsistent with the current conjecture (see Section 4.2.1). We use this property to argue that the number of iterations of `Sorcar-Iterative` has an upper bound of at most $2|\mathcal{P}|$, which can be verified by carefully examining the updates of X and R as counterexamples are added to the Horn-ICE sample $\mathcal{S}$:

- If a positive counterexample c is added to $\mathcal{S}$, then it is added because $c \not\models X \cap R$ (as the teacher enforces progress). This implies $c \not\models X$, which in turn means that there exists a predicate $p \in X$ with $c \not\models p$. In the subsequent round of the passive Sorcar algorithm, p is no longer present in X (see Line 5) and $|X|$ decreases by at least one as a result.

- If a negative counterexample c is added to $\mathcal{S}$, then it is added because $c \models X \cap R$ (as the teacher enforces progress). This means that the set X remains unchanged in the next iteration but at least one relevant predicate is added to R in order to account for the new negative counterexample (Line 15). This increases $|R|$ by at least one.

- If a Horn counterexample $(\{c_1, \ldots, c_n\}, c)$ is added to $\mathcal{S}$, then it is added because $c_i \models X \cap R$ for each $i \in \{1, \ldots, n\}$ but $c \not\models X \cap R$ (as the teacher enforces progress). In this situation, two distinct cases can arise:

  1. If $(\{c_1, \ldots, c_n\}, c)$ is not consistent with X (i.e., $c_i \models X$ for each $i \in \{1, \ldots, n\}$ but $c \not\models X$), the computation in Line 5 identifies and removes a predicate $p \in X$ with $c \not\models X$ in order to make X consistent with $\mathcal{S}$. This means that $|X|$ decreases by at least one.
  2. If $(\{c_1, \ldots, c_n\}, c)$ is consistent with X but not with $X \cap R$, then X remains unchanged. However, at least one new relevant predicate is added to R in order to account for the new Horn counterexample (Line 15). This means that $|R|$ increases by at least one.

  Thus, either $|X|$ decreases or $|R|$ increases by at least one.

In the worst case, `Sorcar-Iterative` arrives at a state with $X = \emptyset$ and $R = \mathcal{P}$ (if it does not find an inductive invariant earlier). Since the algorithm starts with $X = \mathcal{P}$ and $R = \emptyset$, this worst-case situation occurs after at most $2|\mathcal{P}|$ iterations.

Let us now assume that `Sorcar-Iterative` indeed arrives at a state with $X = \emptyset$ and $R = \mathcal{P}$. Then, we claim that the result of `Sorcar-Iterative`, namely $X \cap R = \emptyset$, is an inductive invariant. To prove this claim, first recall that Theorem 4.1 shows that `Sorcar-Passive` always learns a set of predicates that is consistent with the given Horn-ICE sample $\mathcal{S}$. In particular, Line 5 of `Sorcar-Passive` computes the (unique) largest set $X \subseteq \mathcal{P}$ that is consistent with $\mathcal{S}$. Second, we know that every inductive invariant $X^\star$ is consistent with $\mathcal{S}$ because the teacher is honest. Thus, we obtain $X^\star \subseteq X = \emptyset$ and, hence, $X^\star = X$ because both X and $X^\star$ are consistent with $\mathcal{S}$ and X is the largest consistent set. This means that X is an inductive invariant because $X^\star$ is one.

Note, however, that `Sorcar-Iterative` might terminate earlier, in which case the current conjecture is an inductive invariant by definition of the Horn-ICE framework. In summary, we have shown that `Sorcar-Iterative` terminates in at most $2|\mathcal{P}|$ iterations with an inductive invariant (if one is expressible as an conjunctive formula over $\mathcal{P}$). Q.E.D.

Finally, let us note that `Sorcar-Iterative` can also detect if no inductive invariant exists that is expressible as a conjunction over $\mathcal{P}$. In this case, the computation of X in Line 5 of `Sorcar-Passive` fails and the algorithm aborts.

4.3.3 Computing Relevant Predicates

We develop *four* different implementations of the function `Relevant-Predicates`. All of these functions share the property that the search for relevant predicates is limited to the set $X \setminus R$ because only predicates in this set can help making $X \cap R$ consistent with negative and Horn counterexamples (cf. Line 6 of Algorithm 4.1). Moreover, recall that we define a predicate to be relevant if it evaluates to *false* on some negative counterexample or on a program configuration appearing on the left-hand-side of a Horn counterexample. Intuitively, these are predicates in $\mathcal{P}$ that have shown some relevancy in the sense that they can be used to establish consistency with the Horn-ICE sample.

`Relevant-Predicates-Max:`
The function `Relevant-Predicates-Max`, shown as Algorithm 4.2, computes the maximal set of relevant predicates from $X \setminus R$ with respect to the negative counterexamples in N and the Horn counterexamples in H. To this end, it accumulates all predicates that evaluate to *false* on a negative counterexample in N or on a program configuration appearing on the left-hand-side of a Horn counterexample in H. The resulting set R' can be large, but $X \cap R'$ is guaranteed to be consistent with N and H (because each negative counterexample and each program configuration on the left-hand-side of a Horn counterexample violates at least one predicates in R', the latter causing each Horn counterexample to be violated). Since

Algorithm 4.2: Computing the maximal set of relevant predicates

```
1  Function Relevant-Predicates-Max($N$, $H$, $X$, $R$):
2      $R' \leftarrow \emptyset$;
3      foreach negative counterexample $c \in N$ do
4          $R' \leftarrow R' \cup \{p \in X \setminus R \mid c \not\models p\}$;
5      end
6      foreach Horn counterexample $(\{c_1, \ldots, c_n\}, c) \in H$ do
7          $R' \leftarrow R' \cup \bigcup_{i=1}^{n} \{p \in X \setminus R \mid c_i \not\models p\}$;
8      end
9      return $R'$;
10 end
```

$X \cap R$ was neither consistent with N nor with H, and since $R' \subseteq X \setminus R$, it follows that R' must contain at least one relevant predicate not in R, thus satisfying the requirement of `Relevant-Predicates`. Finally, the runtime of `Relevant-Predicates-Max` is in $\mathcal{O}(|\mathcal{P}| \cdot |\mathcal{S}|)$ since $X \setminus R \subseteq \mathcal{P}$, $N \subseteq S_-$, and $H \subseteq S_H$.

`Relevant-Predicates-First:`

The function `Relevant-Predicates-First` is shown as Algorithm 4.3. Its goal is to select a smaller set of relevant predicates than `Relevant-Predicates-Max`, while giving the user some control over which predicates to choose. More precisely, `Relevant-Predicates-First` selects for each negative counterexample and each Horn counterexample only one relevant predicate $p \in X \setminus R$. The exact choice is determined by a total ordering $<_{\mathcal{P}}$ over the predicates, which reflects a preference among predicates and which we assume to be a priori given by the user. Using the same arguments as for the function `Relevant-Predicates-Max`, it is not hard to verify that the resulting set R' contains at least one additional relevant predicate not in R and that $X \cap R'$ is consistent with N and H. Moreover, R' clearly contains only a subset of the predicates returned by `Relevant-Predicates-Max`. Again, the runtime is in $\mathcal{O}(|\mathcal{P}| \cdot |\mathcal{S}|)$.

`Relevant-Predicates-Min:`

The function `Relevant-Predicates-Min`, shown as Algorithm 4.4, takes the idea of `Relevant-Predicates-First` one step further and computes a (not necessarily unique) *minimum* set of relevant predicates with respect to N and H. It does so by means of a reduction to a well-known optimization problem called *minimum hitting set* [145].[2] For a collection $\{A_1, \ldots, A_\ell\}$ of finite sets, a set B is a *hitting set* if $B \cap A_i \neq \emptyset$ for all $i \in \{1, \ldots, \ell\}$, and the

[2]Note that the corresponding decision problem is NP-complete.

Algorithm 4.3: Computing relevant predicates based on a preference ordering

```
1  Function Relevant-Predicates-First($N$, $H$, $X$, $R$):
2      Define a total order $<_\mathcal{P}$ over $\mathcal{P}$;
3      $R' \leftarrow \emptyset$;
4      foreach negative counterexample $c \in N$ do
5          $R' \leftarrow R' \cup \{p\}$ where $p$ is the $<_\mathcal{P}$-smallest predicate with $p \in X \setminus R$ and $c \not\models p$;
6      end
7      foreach Horn counterexample $(\{c_1, \ldots, c_n\}, c) \in H$ do
8          $R' \leftarrow R' \cup \{p\}$ where $p$ is the $<_\mathcal{P}$-smallest predicate from the set
             $\bigcup_{i=1}^{n} \{p \in X \setminus R \mid c_i \not\models p\}$;
9      end
10     return $R'$;
11 end
```

Algorithm 4.4: Computing a minimal set of relevant predicates

```
1  Function Relevant-Predicates-Min($N$, $H$, $X$, $R$):
2      For each $c \in N$, construct $A_c \coloneqq \{p \in X \setminus R \mid c \not\models p\}$;
3      For each $(L, c) \in H$, construct $A_{(L,c)} \coloneqq \{p \in X \setminus R \mid \exists c' \in L \colon c' \not\models p\}$;
4      Compute a minimal hitting set $R'$ for the instance
         $Q \coloneqq \{A_c \mid c \in N\} \cup \{A_{(L,c)} \mid (L, c) \in H\}$ (e.g., using a SAT solver);
5      return $R'$;
6  end
```

minimum hitting set problem asks to compute a hitting set of minimum cardinality. In the first step of the reduction, the function `Relevant-Predicates-Min` constructs for each negative counterexample $c \in N$ the set A_c of all predicates $p \in X \setminus R$ violating c and for each Horn counterexample $(L, c) \in H$ the set $A_{(L,c)}$ of all predicates $p \in X \setminus R$ violating some program configuration $c' \in L$. In a second step, it uses an exact algorithm (e.g., a SAT solver) to find a minimum hitting set R' for the problem instance $Q \coloneqq \{A_c \mid c \in N\} \cup \{A_{(L,c)} \mid (L, c) \in H\}$. By construction of the sets A_c and $A_{(L,c)}$, the resulting minimum hitting set R' then is a minimum set of relevant predicates guaranteeing that $X \cap R'$ is consistent with N and H. Moreover, R' contains at least one relevant predicate not in R. However, the downside of approach is that it is not a polynomial time algorithm as the underlying decision problem is NP-complete.

`Relevant-Predicates-Greedy:`

The key idea underlying the function `Relevant- Predicates-Greedy`, which is shown as Algorithm 4.5, is to replace the exact computation of a minimum hitting set with a polynomial-time approximation algorithm. More precisely, `Relevant-Predicates-Greedy` implements

Algorithm 4.5: Greedily computing a “small” set of relevant predicates

```
Function Relevant-Predicates-Greedy($N$, $H$, $X$, $R$):
    For each $c \in N$, construct $A_c \coloneqq \{p \in X \setminus R \mid c \not\models p\}$;
    For each $(L, c) \in H$, construct $A_{(L,c)} \coloneqq \{p \in X \setminus R \mid \exists c' \in L \colon c' \not\models p\}$;
    $R' \leftarrow \emptyset$;
    $Q \leftarrow \{A_c \mid c \in N\} \cup \{A_{(L,c)} \mid (L, c) \in H\}$;
    while $Q \neq \emptyset$ do
        Pick $p \in X \setminus (R \cup R')$ such that $|\{A \in Q \mid p \in A\}|$ is maximal;
        $R' \leftarrow R' \cup \{p\}$;
        $Q \leftarrow Q \setminus \{A \in Q \mid p \in A\}$;
    end
    return $R'$;
end
```

a straightforward greedy heuristic that successively chooses predicates $p \in X \setminus R$ that have the largest number of a non-empty intersections with sets in Q. This heuristic is essentially the dual of the well-known greedy algorithm for the minimum set cover problem [84] and guarantees to find a solution that is at most logarithmically larger than the optimal one. Apart from being an approximation of the minimal set, choosing relevant predicates greedily based on the *number* of sets it hits also has a statistical bias (choosing predicates more commonly occurring in the sets). Otherwise, except for a runtime in $\mathcal{O}(|\mathcal{P}| \cdot |\mathcal{S}|^2)$ and an approximation factor of $\log |\mathcal{S}|$, `Relevant-Predicates-Greedy` shares the same properties as the function `Relevant-Predicates-Min`.

4.4 EXPERIMENTAL EVALUATION

To evaluate the performance of Sorcar, we implement a prototype, featuring all four variants of Sorcar (as well as more heuristics, which we do not discuss here). This prototype is built on top of the program verifier Boogie [71], which natively supports Houdini and provides a so-called “Abstract-Houdini framework” [146] on top of which we have implemented ICE/Horn-ICE algorithms, including Sorcar. Consequently, Sorcar can easily be integrated into existing, Boogie-based verification tool chains.

We compared Sorcar with two Houdini-based tools: GPUVerify [72, 73], a tool for checking data race freedom in GPU kernels, and a tool by Neider et al. [74] for verifying programs that dynamically manipulate heaps against specifications in separation logic. Since separation logic is undecidable in general, the latter tool is designed to work in tandem with a sound-but-incomplete verification engine rather than a complete decision procedure. To the best of our knowledge, both tools are the best ones available for their respective domains.

We have evaluated our implementation on two benchmarks suites: the first suite is shipped with GPUVerify, while the second is included in Neider at al.'s tool. As both of these tools use Houdini, all benchmarks were already equipped with a large number of predicates (often several hundred). We describe each benchmark suite in more detail shortly.

The goal of our experimental evaluation was twofold: (a) to determine whether Sorcar can prove programs correct that the Houdini-based tools cannot (and vice versa) as well as (b) to assess the performance of Sorcar in comparison to these two tools. Since one of the key design principles of Sorcar is to improve verification by constructing weaker invariant (smaller sets of conjuncts), we also report on the size of the invariants (number of conjuncts) inferred by Sorcar and compare to the other tools.

4.4.1 Benchmarks and Compared Tools

The first benchmark suite originates from GPUVerify [72, 73] and was obtained from GPU kernels written in OpenCL and CUDA. GPUVerify processes such programs automatically by means of a complex process, involving sequentialization and compilation to the Boogie programming language. After removing all programs that did not have loops or recursion, this benchmark suite contained 287 programs.

GPUVerify proceeds in three stages. The first stage compiles an OpenCL or CUDA program into a Boogie program. The second stage uses Houdini in a custom version of Boogie to infer an inductive invariant; in this phase, the assertions are in fact removed as Houdini is anyway agnostic to the property being verified. Finally, the third phase substitutes the synthesized invariants, inserts the assertions back into the Boogie program, and verifies it.

The second benchmark suite is taken from Neider et al. [74]. It consists of 62 heap manipulating programs, written in C and are equipped with specifications in Dryad, a dialect of separation logic that allows expressing second order properties using recursive functions and predicates.

Neider et al.'s tool uses the following verification tool chain. First, an extension of VCC [100], called VCDryad [68], compiles the C code into a Boogie program by unfolding recursive definitions, modeling heaplets as sets, and applying frame reasoning using a technique called natural proofs [68, 69, 105]. The tool then poses the verification problem as an invariant synthesis problem over a class of predicates that express complex properties of the heap (such as whether the heaplets of two data structures are disjoint, whether a list is sorted, and so on). Finally, Neider et al.'s tool uses Houdini to infer a loop invariant.

Note that the final phase of both tools is to synthesize a conjunctive invariant over a fixed set of predicates using HOUDINI. In our experiments, we have replaced HOUDINI with SORCAR.

4.4.2 Evaluation

All experiments were conducted on an Intel Xeon E7-8857 v2 CPU at 3, 6 GHz, running Debian GNU/Linux 9.5. The timeout limit was 1200 s. So as to not clutter the following presentation too much, we only report on the version of SORCAR that performed best: SORCAR-MAX (using `Relevant-Predicates-Max`). Additionally, we briefly compare SORCAR-MAX to SORCAR-GREEDY, the latter using `Relevant-Predicates-Greedy`.

Figures 4.2a and 4.2b compare SORCAR-MAX and GPUVerify on the first benchmark suite consisting of GPU kernels. Figure 4.2a compares the time taken to verify a program, while Figure 4.2b compares the number of predicates in the final invariant (there is only one loop invariant in these programs). As can be seen from the figures, SORCAR-MAX compares highly favorably in efficiency. Specifically, SORCAR-MAX was able to verify 15 programs that GPUVerify could not verify, whereas GPUVerify verified only 2 programs that SORCAR-MAX could not verify. SORCAR-MAX was also able to show 9 programs to not have a conjunctive invariant that GPUVerify could not (GPUVerify was not able to show this for any program that SORCAR-MAX could not). On programs that both tools were able to verify (216 programs in total), SORCAR-MAX took on average 34 s per program (and synthesized invariants with an average number of 12 predicates). GPUVerify, on the other hand, took on average 89 s per program (and synthesized invariants with an average number of 23 predicates).

Additionally (not depicted in the scatter plots), we increased the time limit for programs that only one tool could verify from 1200 s to 3600 s. GPUVerify was able to verify 8 additional programs within this time limit. SORCAR, on the other hand, verified both programs that it had timed out on previously. Thus, with this larger timeout, SORCAR was able to verify a proper superset of programs that GPUVerify verified.

Figures 4.2c and 4.2d compare SORCAR-MAX to the tool of Neider et al. [74] on the second benchmark suite of programs with DRYAD specifications. Again, SORCAR-MAX outperformed the HOUDINI-based tool. Specifically, SORCAR-MAX was able to verify 3 programs that Neider et al.'s tool could not verify, whereas Neider et al.'s tool verified 2 programs that SORCAR-MAX could not verify. On programs that both tools were able to verify (57 programs in total), SORCAR-MAX took on average 20 s per program (and synthesized invariants with an average number of 19 predicates). On the other hand, Neider et al.'s tool took on average

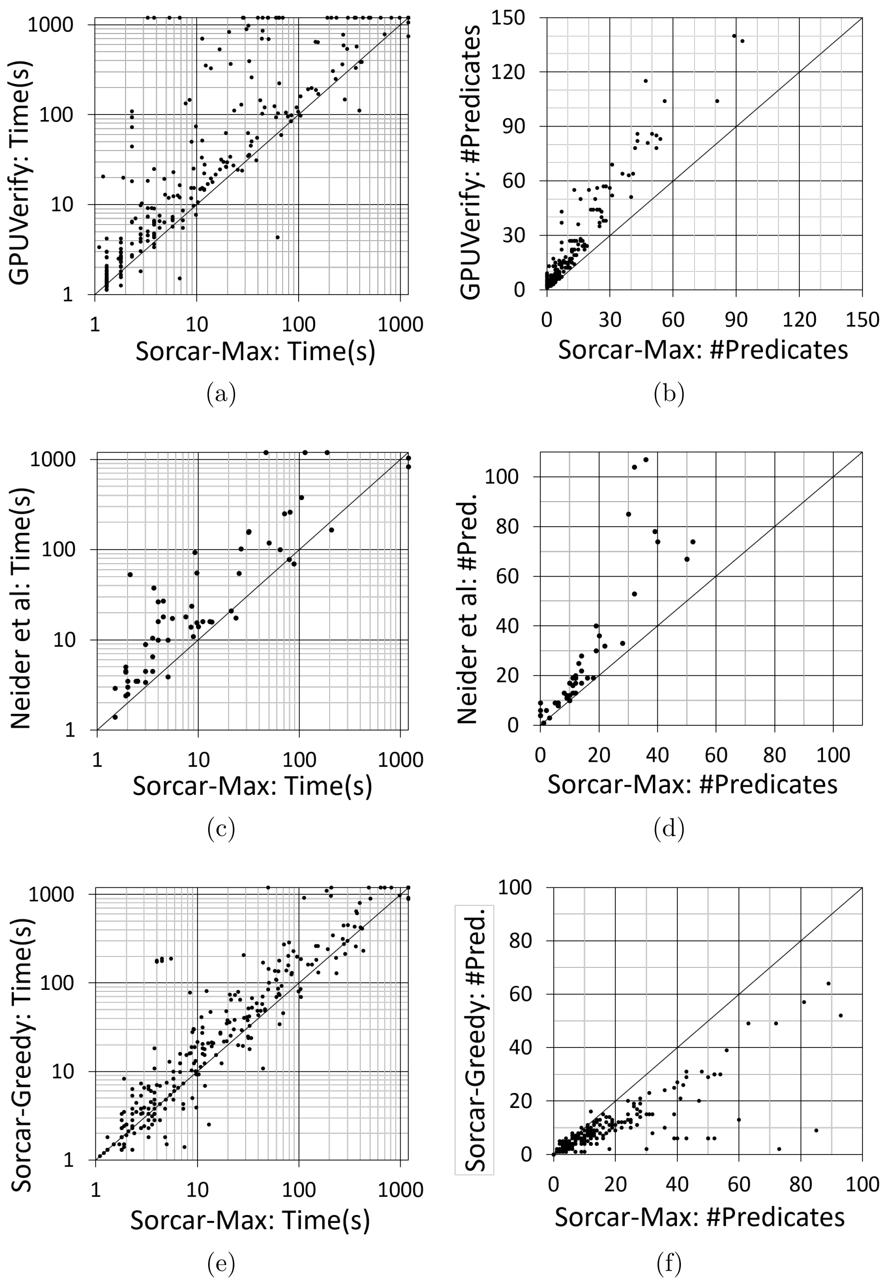


Figure 4.2: Comparison of the time taken to verify a benchmark and the number of predicates in the final invariant. Subfigures (a) and (b) compare SORCAR-MAX and GPUVerify on the first benchmark suite. Subfigures (c) and (d) compare SORCAR-MAX and Neider et al.'s tool on the second benchmark suite. Subfigures (e) and (f) compare SORCAR-MAX and SORCAR-GREEDY on both benchmark suites.

45 s per program (and synthesized invariants with an average number of 37 predicates).

Figures 4.2e and 4.2f compare Sorcar-Max and Sorcar-Greedy on both benchmark suits. The latter was slightly slower overall, but synthesized invariants with fewer predicates.

4.4.3 Comparison of Sorcar and Houdini-ICE

We performed additional experiments with Houdini-ICE (i.e., an implementation of Houdini as a Horn-ICE learning algorithm based on the elimination algorithm). This allowed us to force the number of counterexamples returned by Boogie in each round to be the same for Sorcar and Houdini-ICE (a parameter over which we do not have control in Boogie's implementation of Houdini).

On the GPUVerify benchmark suite, Sorcar-Max verified 5 programs that Houdini-ICE could not, whereas Houdini-ICE was able to verify 1 program that Sorcar-Max could not. Houdini-ICE was also able to show 2 programs to not have a conjunctive invariant, which Sorcar-Max could not. On programs that both were able to verify (233 programs in total), both algorithms performed with similar times.

On the Dryad benchmark suite, Sorcar-Max was able to solve 1 more program than Houdini-ICE (and verified all programs that Houdini-ICE verified). On the 59 programs that both could verify, Sorcar-Max was roughly twice as fast (averaging 24 s per program for Sorcar-MAX vs. 51 s per program for Houdini-ICE).

While Sorcar-Max still emerges better overall than Houdini-ICE, we are not entirely sure why implementing Houdini as an external Horn-ICE learning algorithm makes it perform much better than the internal implementation of Houdini in Boogie (the internal Houdini algorithm within Boogie is embedded deep and is very hard to configure or control). For the GPUVerify benchmarks, the tool GPUVerify does invariant synthesis without assertions and then inserts assertions to verify the program, and this could be one difference. We leave answering this question for future work.

4.5 RELATED WORK

Invariant synthesis lies at the heart of automated program verification. Over the years, various techniques have been proposed, including abstract interpretation [118], interpolation [107], IC3 [108], predicate abstraction [119], abductive inference [120], as well as synthesis algorithms that rely on constraint solving [121, 122, 123, 147]. Complementing these techniques are data-driven approaches that are based on machine learning. Examples include Daikon [124] and Houdini [70], the ICE learning framework [45] and its successor

Horn-ICE learning [125, 126], as well as numerous other techniques that employ machine learning to synthesize inductive invariants [47, 67, 127, 128, 129, 130, 148].

One potentially interesting question is whether ICE/Horn-ICE algorithms (and in particular, Houdini and Sorcar) are qualitatively related to algorithms such as IC3 for synthesizing invariants. For programs with Boolean domains, Vizel et al. [149] study this question and find that the algorithms are quite different. In fact, the authors propose a new framework that generalizes both. In the setting of this chapter, however, there are too many differences to reconcile with: (a) IC3 finds invariants by bounded symbolic exploration, forward from initial configurations and backward from bad configurations (hence inherently unfolding loops), while ICE/Horn-ICE algorithms do not do that, (b) ICE/Horn-ICE algorithms instead use implication/Horn counterexamples, which can relate configurations arbitrarily far away from initial or bad configurations, and there seems to be no analog to this in IC3, (c) it is not clear how to restrict IC3 to finding invariants in a particular hypothesis class, such as conjunctions over a particular set of predicates, (d) IC3 works very closely with a SAT solver, whereas ICE/Horn-ICE algorithms are essentially independent, communicating with the SAT/SMT engine only indirectly, and (e) we are not aware of any guarantees that IC3 can give in terms of the number of rounds/conjectures, whereas the ICE/Horn-ICE algorithms Houdini and Sorcar give guarantees that are linear in the number of predicates. We believe that the algorithms are in fact very different, though more general algorithms that unify them would be interesting to study.

Learning of conjunctive formulas has a long history. An early example is the so-called elimination algorithm [78], which operates in the Probably Approximately Correct Learning model (PAC). Daikon [124] was the first technique to apply the elimination algorithm in a software setting, learning likely invariants from dynamic traces. Later, the popular Houdini [70] algorithm built on top of the elimination algorithm to compute inductive invariants in a fully automated manner. In fact, as Garg et al. [46] and later Ezudheen et al. [126] argued, Houdini can be seen as a learning algorithm for conjunctive formulas in both the ICE and the Horn-ICE learning framework.

Using Houdini to compute conjunctive invariants over a finite set of candidate predicates is extremely scalable and has been used with great success in several practical settings. For example, Corral [143], which uses Houdini internally, has replaced Slam [150] and Yogi [151], and is currently shipped as part of Microsoft's industrial-strength Static Driver Verifier (SDV) [141, 142]. GPUVerify [72, 73] is another example that uses Houdini with great success to prove race freedom of GPU programs.

4.6 CONCLUSIONS AND FUTURE WORK

In this chapter, we have developed a new class of learning algorithms for conjunctions, named SORCAR, which are biased towards the simplest conjunctive invariant that can prove the assertions correct. SORCAR is parameterized by functions to identify relevant predicates and guarantees to learn an invariant in a linear number of rounds (if one exists). We have shown that SORCAR proves programs correct significantly faster than state-of-the-art HOUDINI-based tools.

There are several future directions to pursue. First, we believe that further algorithms for learning conjunctions need to be explored. For instance, the Winnow algorithm [152] learns from positive and negative samples in time $\mathcal{O}(r \log n)$, where r is the size of the final formula and n is the number of predicates. Finding Horn-ICE learning algorithms that have such sublinear round guarantees can be very interesting as r is often much smaller than n in verification examples. Second, we would like to use the new SORCAR algorithms in specification mining settings where smaller invariants are valuable as they are read by humans. Third, there are several types of inference algorithms similar to HOUDINI (see [153]), and it would be interesting to explore how well SORCAR performs in such settings.

CHAPTER 5: LEARNING PRECONDITIONS AND POSTCONDITIONS USING TEST GENERATORS AS ORACLES

In this chapter, we consider the problem of synthesizing contracts, where we propose frameworks to synthesize precondition and postcondition of a method in a class of a program in an object-oriented language. In this context, the natural oracle to use would be a verification engine that verifies programs (with loops/recursion). However, verification engines that can do completely automated program verification are not often effective or scalable. We hence consider using test generators as teaching oracles.

Given a program annotated with preconditions and assertions, the test generator creates a valid object state (using object modifying methods) and concrete input parameters of the method that satisfy the precondition. Furthermore, several test generators are guided by the assertions, and try to generate inputs that violate them.

Synthesizing Preconditions:

In the learning framework we develop for synthesizing preconditions, we use as counterexamples abstractions of the input states, which consists of the primitive type inputs, and the valuations of a certain set of observer methods of the non-primitive-type input objects.

Each input state (and similarly for counterexample) created by the test generator, can be either valid: execute successfully and terminate, or invalid: encounter an uncaught exception, or result in an assertion violation. Note that the predicates used in the logic for expressing preconditions and the observer methods for deriving properties of objects, create abstractions of input states. Abstractions of invalid input states must be excluded by the precondition. However, abstractions of valid input states need not necessarily be included in the precondition, as there could be input states with the same abstraction but are invalid.

We define the problem of precondition synthesis using a notion of *ideal preconditions*. An ideal precondition for a method with respect to a test generator is a precondition which satisfies two properties. First, *safety*: the test generator should not be able to find any invalid input state allowed by the precondition. Second is *maximality*: the precondition should include as many valid input states as possible. More precisely, it can exclude an abstraction of input states only if there exists some invalid input state that has this abstraction. The maximality requirement intuitively captures the desire to synthesize weakest (most liberal) preconditions.

In our learning framework, counterexamples are positively and negatively labeled abstraction of input states. The meaning of the counterexample is as follows. The learner needs to find a formula in a fixed logic L that (a) excludes all negatively labeled inputs, and (b) includes all positively labeled inputs i unless there is another sample i' that is negatively labeled and is indistinguishable from i by any formula in L.

The learning algorithms we propose for ideal preconditions first resolve all conflicts (conflicts are pairs of samples labeled positive and negative that are indistinguishable by the logic L) by reclassifying them as negative. A classification algorithm is then used to learn a hypothesis precondition from the conflict resolved samples.

To summarize, our framework for precondition synthesis has the following features.

- The application is synthesizing precondition of a method of a class in an object oriented program.
- The verification oracle is a test generator, which is inherently incomplete hence introduces conflicting counterexamples.
- The counterexamples are valid and invalid abstractions of input states.
- We formulate the synthesis problem using the notion of an ideal precondition with respect to the testing oracle and a particular logic L for stating preconditions.
- The learning algorithm to find an ideal precondition first resolves all conflicts in the sample and then uses a classification algorithm that does not make any mistakes.

Synthesizing Conjunctive Postconditions

We propose a framework to synthesize conjunctive postconditions of a method using a test generator as the oracle. We assume that the method is already annotated with a precondition, which prevents all exception failures (this can be done using the precondition synthesis mentioned above). Synthesized postconditions need to be *strong*, and ideally the strongest postcondition expressible in a given logic.

In a learning framework, the test generator can only provide pairs of feasible input-output states (where input states satisfy the precondition). Consequently, given a hypothesis postcondition, a test generator can refute that the postcondition is correct by giving executions that end in states that are not satisfied by the postcondition. However, it cannot refute the assertion that the postcondition is the strongest one. In terms of counterexamples, we can think of the test generator as being able to only provide positively labeled pairs of abstractions of input and output states.

We propose to learn postconditions in a logic that consists of conjunctions of (a) predicates over a fixed set P and (b) an equality expression that defines an output as a function of the input. The predicates P and the parameters for the functions synthesized are based on input parameters and abstractions of objects using observer methods.

The above logic facilitates learning *tight* concepts (as the above logic is closed under conjunction). We synthesize functional relationships between input and output using a SyGuS solver. We then seed this as equality predicates and add them to P, and then use the elimination algorithm [78] to learn the semantically smallest conjunctive formula over the predicates that includes all the positive counterexamples.

In summary, our learning framework for postcondition synthesis has the following features:

- The application is to learn a strong conjunctive postcondition of a method that is already annotated with a precondition.
- The counterexamples in this framework are only positive. The oracle is a test generator, and counterexamples are abstractions of pairs of input and output states that are always classified positively.
- The learning algorithm first synthesizes new predicates using a SyGuS solver and then uses the elimination algorithm to learn the tightest conjunctive postcondition.

5.1 INTRODUCTION

We present a novel counterexample guided inductive synthesis (CEGIS) framework, to synthesize stateful preconditions, with the help of a test generator as the verification oracle with the precondition being guaranteed to be safe and maximal with respect to the given test generator.

We assume that we have a method $m(\vec{p})$ with formal parameters $\vec{p}$ and assertions in it for which we want to synthesize a precondition, such that the synthesized precondition satisfy two requirements: (a) be *safe*, the test generator cannot find a precondition-satisfying input whose execution leads to an exception (either a runtime exception such as division by zero or an assertion-violating exception), and (b) be *maximal*, the test generator cannot find an input disallowed by the precondition whose execution does not lead to any exception, Since we do not know a priori the precise set of inputs on which the method throws an exception and does not throw exceptions, respectively, we resort to obtaining this information from a test generator.

Defining the precondition synthesis formally modulo a test generator is complicated by three main aspects of the problem:

1. *Incomplete information of object state:* One fundamental aspect of our problem is that the client does not know precisely the internal states of objects (the receiver object state and state of other objects given as parameters $\vec{p}$) on which the precondition depends on. We propose a set of *observer methods* that give properties of these objects, allowing the client to have incomplete information about them gleaned from the return values of observer methods. We define a *feature vector* $\vec{f}$ as a vector of values of the primitive parameters in $\vec{p}$ and the return values of the observer methods on the object states, hence enabling the precondition to state restrictions using these features (typically, the features are of primitive types). This abstraction is not all bad as the learned precondition can express abstract properties of the non-primitive-type objects while avoiding revealing implementation details (e.g., primitive-type object fields recursively reachable from an input object along with the heap structure of an input object). However, using observer methods and feature vectors intrinsically introduces *incomplete information* about the object state as several different input states can have the same feature vector.

2. *Incomplete test generator:* Given a method and a precondition for it, the test generator can find input states that the precondition should *disallow* as the method can throw an exception on these input states and find input states that the precondition should *allow* as the method does not throw any exception on these input states. A feature vector is valid (or invalid) if the method can throw an exception on none (or one) of all input states conforming to the feature vector. Inherently the test generator cannot guarantee feature vectors to be valid (but can certify invalid feature vectors). First, the test generator can tentatively label certain vectors as valid, and later change its mind and label these vectors invalid. Learning of preconditions hence needs to accommodate such fluctuations.

3. *Expressiveness of the logic:* The logic used for expressing the precondition may not be expressive enough to distinguish two feature vectors, one being valid and the other being invalid. In other words, there is another level of abstraction caused by the logic, in addition to the abstraction induced by the use of observer methods, and the precondition must be permitted to disallow certain positive feature vectors.

Notice that if we can find an input state that conforms to a precondition φ on which the program throws an exception, we can deem the precondition to be unsafe, and declare the feature vector corresponding to that input state as invalid. However, execution of the method on a single input state cannot show φ to be non-maximal. If the testing tool finds a valid input state s_1 disallowed by φ, we still cannot say that the feature vector corresponding to the input state is valid. The reason is that there may be *another* invalid input state s_2 that conforms to the same feature vector.

Intuitively, witnessing non-maximality boils down to finding a valid feature vector disallowed by φ such that *all* input states conforming to the feature vector are valid. The $\exists\forall$ nature of the question is what makes finding counterexamples for maximality hard using test generation. (Even logic-based tools, such as Pex, that use SMT solvers are typically effective/decidable for only $\exists^*$ properties, i.e., quantifier-free formulas.) On the other hand, finding an invalid feature vector (included by φ) asks whether there *exists* a feature vector allowed by φ such that there does *exist* an invalid input state conforming to the feature vector; this question is an $\exists\exists$ question that can be found using tools such as Pex.

We shape the definition of our problem with respect to a test generator, which we call a *testing-based teacher* (TBT). A TBT can be seen as a function that takes a method m, a precondition φ for m, and generates a finite set of input states for m (that may or may not be allowed by φ) and whether they are valid or not.

Note that we denote the TBT as a *function*; hence, for any method and precondition, we expect the TBT to be deterministic, i.e., it produces the same set of test inputs across rounds for a given precondition. This assumption is not a limitation of our framework, but a way to formalize a TBT. Any testing-based tool can be made *deterministic* by fixing its random seeds, and by fixing configurable bounds such as the number of branches explored, etc.

We introduce the notion of an ideal precondition that captures both safety and maximality modulo the incomplete information that the client has of the object state and modulo the expressiveness of the logic, with respect to the TBT. The safety property demands that the TBT is not able to find any invalid input state allowed by the precondition (i.e., one on which m throws an exception). The maximality property requires that for any valid input state found by the TBT but disallowed by the precondition, there must be *some* invalid input state (returned by the TBT allowed by *some* precondition, not necessarily φ), such that these two states cannot be distinguished by the logic $\mathcal{L}$. We aim to synthesize an ideal precondition for $m(\vec{p})$ with respect to the TBT.

We propose a general learning framework for synthesizing ideal preconditions with respect to a testing-based teacher (TBT). It consists of five distinct components: (1) a passive learner (precondition synthesizer) that synthesizes preconditions from positive and negative feature vectors, (2) a TBT, interacting in rounds of communication with the learner, that returns valid/invalid input states, (3) a featurizer that converts valid/invalid input states to positive/negative feature vectors, and (4) a conflict resolver ($CR_{\mathcal{L}}$), that resolves conflicts (created by incomplete information) by changing positive feature vectors to negative ones when necessary. We emphasize that one can use any standard passive learner in this framework as long as it finds formulas that are consistent with the set X of labeled feature vectors.

The framework maintains a set X, which contains the accumulated set of (conflict-resolved) positive/negative labeled feature vectors that the TBT has returned. In each round, the learner proposes a precondition that is *consistent* with the set, and the TBT returns a set of valid and invalid input states. The featurizer, with the help of observer-method calls, converts the input states to positive/negative labeled feature vectors. We add the counterexample input states to X and call the conflict resolver for the logic $\mathcal{L}$, and update X. We then check whether the current conjecture is consistent with the updated X—namely whether φ is *true* on every positive feature vector and $false$ on every negative feature vector. If it is, then we exit having found an ideal precondition, and if not, we iterate with the precondition synthesizer for the new set X.

We show that our framework is convergent (i.e., guaranteed to terminate). When the following conditions are met: (1) the hypothesis space $\mathcal{H}$ of preconditions is finite, (2) the logic is closed under Boolean operations, and (3) the learner is able to always produce formulas consistent with samples when they exist, we are guaranteed to converge to an ideal precondition.

We implement a prototype of our framework in a tool called Proviso using a learner based on the ID3 classification algorithm [75], a powerful classification algorithm in the machine learning community, and Pex [76], an industrial test generator based on dynamic symbolic execution [154, 155], shipped as IntelliTest in the Microsoft Visual Studio Enterprise Edition since 2015.

We also instantiate the framework for two important tasks in specification inference: runtime-failure prevention and conditional-commutativity inference [77]. The former problem asks to synthesize preconditions that avoid runtime exceptions of a single method. The latter problem asks, given two methods, a precondition that ensures that the two methods commute, when called in succession.

5.2 SYNTHESIZING PRECONDITIONS

5.2.1 An Illustrative Example

We next show how our framework is instantiated for the task of conditional-property inference and then illustrate through an example how our approach addresses the precondition

```
[PexMethod]
public void PUT-CommutativityAddContains(ArrayList s1, int x, int y)
{
   ArrayList s2 = new ArrayList(s1); //clone s1
   int a1, a2; bool ad1, ad2;

   //First Interleaving
   a1 = s1.Add(x);
   ad1 = s1.Contains(y);

   //Second Interleaving
   ad2 = s2.Contains(y);
   a2 = s2.Add(x);

   PexAssert.IsTrue(a1 == a2 && ad1 == ad2 && Equals(s1, s2));
}
```

Figure 5.1: Encoding conditional property: Commutativity conditions for methods `Contains` and `Add` from the ArrayList class in the .NET Library.

synthesis problem.

Let us first model the problem of conditional-commutativity inference (finding conditions under which two methods commute) as a problem of precondition synthesis. Consider the parameterized unit test [156] in Figure 5.1. The method `PUT_CommutativityAddContains` checks whether the methods of an arraylist, `Add` and `Contains`, commute when called with an arraylist $s1$, and for particular parameter inputs. The method `Add(x)` returns the index at which x has been added, and `Contains(y)` returns *true* if y is in $s1$ and *false* otherwise. To check for commutativity, the test method first clones the input arraylist $s1$ into $s2$. It then calls the method sequence `Add(x)` and `Contains(y)` on $s1$, and `Contains(y)` and `Add(x)` on $s2$. Finally, it checks whether the return values of the methods and resulting objects $s1$, $s2$ are equal. If they are not, the methods do not commute and hence it raises an exception; it follows that the precondition for the method `PUT_CommutativityAddContains` to prevent exceptions (e.g., assertion failure) is precisely the condition under which the two methods `Add(x)` and `Contains(y)` commute.

To synthesize stateful preconditions, we instantiate our framework by fixing a logic $\mathcal{L}$ of octagonal constraints, by fixing a conflict resolver, a component that effectively relabels positive feature vectors to negative ones when necessary (see Section 5.2.3.1 for details), by fixing an exact learning engine, decision-tree learning, and by fixing a test generator, PEX. As inputs, our approach takes a method m (e.g., Figure 5.1) for precondition synthesis and a set of Boolean and integer observer methods in the `ArrayList` class, $Obs_{\mathbb{B}} = \{$`Contains(int)`$\}$

and $Obs_{\mathbb{Z}} = \{$`Count`, `IndexOf(int)`,`LastIndexOf(int)`$\}$, respectively. Our approach uses these observer methods and primitive parameters of m to generate a feature vector $\vec{f}$ by applying those methods using various combinations of parameters of m:
[`s1.Count()`, `x`, `y`, `s1.IndexOf(x)`, `s1.IndexOf(y)`, `s1.LastIndexOf(x)`, `s1.LastIndexOf(y)`, `s1.Contains(x)`, `s1.Contains(y)`].

Next we demonstrate how our algorithm proceeds. A set X (initially empty) of cumulative positive and negative feature vectors is maintained. Our algorithm proceeds in rounds: the learner begins by proposing a conjectured precondition, the testing-based teacher generates counterexamples. To generate negative counterexamples, the teacher generates inputs that are allowed by the conjectured precondition but cause the method to fail. To generate positive counterexamples, the teacher generates inputs that are disallowed by the conjectured precondition and do not cause the method to fail. These counterexamples are given to a conflict resolver, which then relabels a positive counterexample c to negative if in X there is a negative counterexample c' that is $\mathcal{L}$-indistinguishable from c. The algorithm then checks whether the current conjectured precondition is consistent with the updated set X (i.e., the conjectured precondition allows the positive feature vectors in X and disallow the negative feature vectors in X): if yes, we stop and output the precondition; otherwise, we proceed to the next round. We elaborate the role of the conflict resolver and the soundness of the preceding technique in the rest of the chapter.

To illustrate the conflict resolver on this example, we assume that no observer methods are given, and the feature vector is $\vec{f'} = [\mathtt{x}, \mathtt{y}]$. The learner begins by proposing *true*, and the testing-based teacher produces negative counterexamples $([0,0],-)$, $([10,10],-)$, being added to X (which is initially empty). The precondition *true* is not consistent with X and so we proceed with the next round. The learner next proposes *false* (as it is consistent with X). The teacher then generates two positive feature vectors $([0,0],+)$ and $([8,9],+)$. At this point, we have encountered conflict. X has a negative feature vector $([0,0],-)$ and an $\mathcal{L}$-indistinguishable positive vector $([0,0],+)$. The conflict resolver relabels $([0,0],+)$ to $([0,0],-)$. Again, the current conjecture *false* is not consistent with the updated X and so we proceed. This process (in our tool) continues for 4 rounds when the learner ultimately proposes $(x \neq y)$, which is consistent with all vectors that the test generator returns, and we stop and return $(x \neq y)$ as the precondition. When the feature vector $(\vec{f})$ mentioned earlier includes all the observer methods, the preceding conflict does not occur, and the learner synthesizes the precondition $(x = y \wedge s1.\mathit{Contains}(x)) \;\vee\; (x \neq y)$.

A crucial aspect here is that the testing-based teacher helps the learner by generating counterexamples that show the conjectured precondition to be unsafe or non-maximal. We terminate only when the learner is able to convince the test generator that the precondition is safe and maximal (modulo the power of the test generator).

5.2.2 Problem Formalization of Precondition Synthesis Modulo a Test Generator

In this section, we formalize the problem of synthesizing preconditions with the aid of a test generator.

We assume that we have a method $m(\vec{p})$ with formal parameters $\vec{p}$ and assertions in it for which we want to synthesize a precondition. Intuitively, we want the precondition to satisfy two requirements: (a) be *safe*, in the sense that the method when called with any state allowed by the precondition does not throw an exception (either a runtime exception such as division by zero or an assertion-violating exception), and (b) be *maximal*, in the sense that it allows as many inputs as possible on which the method does not throw an exception. Since we do not know a priori the precise set of inputs on which the method throws an exception and does not throw exceptions, respectively, we resort to obtaining this information from a test generator.

Challenges in defining the problem and framework. Defining the precondition synthesis formally modulo a test generator is complicated by three main aspects of the problem:
– *Incomplete information of object state:* Preconditions can depend on the receiver object state of the method $m()$ for which we are synthesizing the precondition for, and the state of objects that are passed as parameters to $m()$. We propose a set of *observer methods* that give properties of these objects, and allow the precondition to state restrictions using these properties. We hence work with *feature vectors*, which capture the return values of observer methods on objects. However, using observer methods intrinsically introduces *incomplete information* about the object state: several different input states can have the same feature vector.
– *Incomplete test generator:* Given a method and a precondition for it, the test generator can find input states that the precondition should *disallow* as the method can throw an exception on these input states and find input states that the precondition should *allow* as the method does not throw any exception on these input states. A feature vector is valid (or invalid) if the method can throw an exception on none (or one) of all input states conforming to the feature vector. However, since we work with an abstraction of input states using feature vectors, we need a test generator to find valid feature vectors and invalid ones. It turns out that given a precondition, a test generator can readily be adapted to find invalid feature vectors, but not valid ones. Consequently, we need to work with a test generator that may mark a feature vector tentatively valid, and then later change its mind and find it invalid.

Learning of preconditions hence needs to accommodate such fluctuations.
– *Expressiveness of the logic:* The logic used for expressing the precondition may not be expressive enough to distinguish two feature vectors, one being valid and the other being invalid. In other words, there is another level of abstraction caused by the logic, in addition to the abstraction induced by the use of observer methods, and the precondition must be permitted to disallow certain positive feature vectors.

Our solution to the preceding challenges involves (1) defining the precondition synthesis problem as synthesizing an ideal precondition (Definition 5.2), where the notion of an ideal precondition accommodates the fluctuations of a test generator, and (2) a framework that synthesizes ideal preconditions using a *conflict resolver* (Section 5.2.3 and Figure 5.2) that manipulates counterexamples returned by the test generator in each round. We emphasize that the component for synthesizing formulas from the (conflict-resolved) counterexamples is standard, and we can use a variety of learning algorithms from the literature. However, arguing convergence of such learning algorithms in learning ideal preconditions in the presence of the conflict resolver has to be argued anew (Section 5.2.3.3).

We next formalize the notions of programs, valid and invalid input states, and testing-based teachers (Section 5.2.2.1), and then formalize the problem of precondition synthesis modulo a testing-based teacher using the notion of an ideal precondition (Section 5.2.2.2).

5.2.2.1 Observer Methods, Logic for Preconditions, and Testing-Based Teachers

Methods. Let us fix a set of *types* $\mathcal{T}$, including primitive types and classes. Each type $t \in \mathcal{T}$ is associated with a data domain $\mathcal{D}(t)$ that denotes the set of values that variables of type t range over. In the following, we assume that each variable v has an implicit type t associated with it. In addition, we denote $\mathcal{D}(t)$ by simply using $\mathcal{D}(v)$.

We assume that we have a target method $m(\vec{p})$ with formal parameters $\vec{p}$ that we want to synthesize a precondition for.

Let us also fix a set of *pure* (i.e., side-effect free) *observer methods* $F = \{f_1(\vec{p_1}), \ldots, f_n(\vec{p_n})\}$ that return a primitive type. These methods help query properties of the state of the objects whose class defines these methods. For a method m with input parameters $\vec{p}$ that we aim to find a precondition for, we allow the precondition to express properties of $\vec{p}$ using constraints on variables of primitive types in $\vec{p}$ as well as the return values of observer methods that return a value of primitive type when called with tuples of parameters drawn from $\vec{p}$.

We have, apart from the above, other methods for classes (including constructors and mutating methods, i.e., those that mutate the object). The test generator can use these methods to create valid object states, by using method sequences composed of constructors and mutating methods.

Let us now define the semantics of the methods abstractly. For any class c, let $\mathcal{S}_c$ denote the set of *valid states of the object* of the class c ($\mathcal{S}_c$ can be infinite, of course, and denotes the set of valuations and heaps maintained by the public/private fields in the class). Note that we assume that the set $\mathcal{S}_c$ contains *valid* object states, i.e., reachable states from initial object construction. For each parameter p of type class c, let us denote by $\mathcal{D}(p)$ the valid states $\mathcal{S}_c$.

The semantics of the observer method $f_i(\vec{p_i})$ is given by a (complete) function $[\![f_i]\!] : \mathcal{D}(\vec{p_i}) \longrightarrow D_i$, where D_i is the data domain for the return primitive type of method f_i. Note that the observer methods return properties of the state of the object but do not change the state. Note also that we require these observer methods not to throw exceptions, and hence model their semantics using *complete* functions.

The semantics of the method $m(\vec{p})$ is given by a *partial function* $[\![m]\!] : \mathcal{S}_c \times \mathcal{D}(\vec{p}) \rightharpoonup \mathcal{S}_c \times D$, where c is the class that m belongs to and D is the data domain for the return type of m (whether it be of primitive type or a class).

Valid and invalid input states. An input state for $m(\vec{p})$ is pair $(s, v) \in \mathcal{S}_c \times \mathcal{D}(\vec{p})$ where c is the class that method m belongs to and v is a valuation of the parameters in $\vec{p}$ of method m. Note that the input state contains the receiver object state namely s of m, and the values of the parameters in $\vec{p}$ (some of which can be object states as well of their respective classes).

We say that an input state (s, v) is an *invalid input state for* m if m throws an exception[1] on that input state i.e., $[\![m]\!]$ is undefined on (s, v). We say that an input state (s, v) is a *valid input state for* m if (s, v) is not an invalid input state.

Feature vectors. One fundamental aspect of our problem is that the client does not know precisely the internal states of objects (the receiver object state and state of other objects given as parameters), but has incomplete information about them gleaned from the return values of observer methods.

We define a *feature vector* $\vec{f}$ as a vector of values of the primitive parameters in $\vec{p}$ and the values of observer methods on the object states (called with various combinations of parameters from $\vec{p}$). Typically, the features are of primitive types (integer and Boolean in our tool).

[1]Note that in this chapter, when we say an exception, we refer to an uncaught exception as unexpected program behaviors such as DivideByZeroException. Assertion violation can also cause an uncaught exception to be thrown.

Logic for expressing preconditions. The *logic* $\mathcal{L}$, for expressing preconditions for a method $m(\vec{p})$ in this chapter is quantifier-free first-order logic formulas. Recall that classical first-order logic is defined by a class of *functions*, *relations*, and *constants*. We choose this vocabulary to include the following: (a) the usual vocabulary over the various primitive data domains that the program operates on (Booleans, integers, strings, arrays of integers, etc.), and (b) observer methods as functions. The logic then allows quantifier-free formulas with free variables $\vec{p}$. Note that such a formula φ, when interpreted at a particular program state (which gives meaning to various objects and hence to corresponding observer methods), defines a set of input states—the input states (s, v) such that when observer methods are interpreted using the state s, and input parameters $\vec{p}$ are interpreted using v, the formula holds. Hence, a logical formula represents a precondition—the set of states that satisfy the formula being interpreted as the precondition.

Note that the logic cannot distinguish between two input states that have the same feature vector. We can in fact view logical formulas as defining sets of feature vectors. The logic hence introduces a coarser abstraction of feature vectors (which themselves are abstractions of input states).

For the tool and evaluation in this chapter, the logic $\mathcal{L}$ is a combination of Boolean logic and octagonal constraints on integers; the observer methods work on more complex datatypes/heaps (e.g., stacks, sets), returning Booleans or integers as output (e.g., whether a stack is empty, the size of a set container).

Testing-based teachers and counterexamples. The general problem of precondition synthesis is to find a precondition expression φ (in logic $\mathcal{L}$) that captures a maximal set of valid feature vectors (where a *valid feature vector* is one whose conforming input states are all valid) for the method m. This synthesis problem is clearly undecidable. In fact, checking whether m throws an exception on even a single input state is undecidable. Proving a precondition to be safe requires verification, a hard problem in practice, and current automatic verification techniques do not scale to large code bases.

We hence shape the definition of our problem with respect to a test generator, which we call a *testing-based teacher* (TBT). (We call it a teacher as it teaches a learner the precondition.)

A TBT is just a test generator that generates test input states for m. Ideally, we would like the TBT to be guided to find test input states for showing that a given precondition φ is not safe or maximal, i.e., input states allowed by φ on which m throws exceptions and input states disallowed by φ where m does not throw an exception (hence property-driven testing tools such as PEX are effective, but not testing tools such as RANDOOP that generate random inputs).

Formally,

Definition 5.1 (Testing-based teacher)**.** *A testing-based teacher (TBT) is a function that takes a method m, a precondition φ for m, and generates a finite set of input states for m (that may or may not be allowed by φ) and whether they are valid or not.*

Note that in our formulation, the TBT is a *function*; hence, for any method and precondition, we expect the TBT to be deterministic, i.e., it produces the same set of test inputs across rounds for a given precondition. This assumption is not a limitation of our framework, but a way to formalize a TBT. Any testing-based tool can be made *deterministic* by fixing its random seeds, and by fixing configurable bounds such as the number of branches explored, etc. We do not require a TBT to report all or any input states. The TBT is incomplete and may not be able to find a counterexample (for safety or maximality), even if one exists.

Given a method m and a precondition φ for it, we can examine the test inputs generated by the TBT to check whether they contain counterexamples. An input state that is allowed by φ but leads m to throw an exception shows that φ is not safe, and is a negative counterexample. An input state that is disallowed by φ and on which m executes without throwing any exception indicates *potentially* that φ may not be maximal, and we call this input state a positive counterexample. (As we shall see, such counterexample does not necessarily indicate that φ is not maximal.)

We are now ready to define the goal of precondition generation parameterized over such a TBT. Roughly speaking, we want to find maximal and safe preconditions expressible in our logic; however, the precise definition is more subtle as we describe next.

5.2.2.2 Precondition Synthesis Modulo a Testing-Based Teacher

Incomplete information. Since the learner learns only with respect to an observer abstraction in terms of feature vectors, we assume that input states returned by the testing-based teacher are immediately converted to feature vectors, where the feature values are obtained by calling the respective observer methods. We also refer to feature vectors as positive or negative counterexamples, if the conforming input states are positive or negative, respectively.

For any feature vector $\vec{f}$, there are, in general, *several input states* that are conforming to $\vec{f}$ (i.e., those input states whose features are precisely $\vec{f}$). Recall that a feature vector $\vec{f}$ is *valid* if *all* input states conforming to it are valid input states; a feature vector $\vec{f}$ is invalid if it is not valid—i.e., there is *some* invalid input state whose feature vector is $\vec{f}$.

It turns out that incomplete information the client has about the object state creates many complications. In particular, a testing tool can find *invalid* feature vectors but cannot find *valid* feature vectors using test generation.

Consider a method m and a precondition φ for it. The precondition defines a set of feature vectors, which in turn define a set of input states. Notice that if we can find an input state that conforms to φ on which the program throws an exception, we can deem the precondition to be unsafe, and declare the feature vector corresponding to that input state as invalid. We name such feature vectors *negative counterexamples*, and a testing tool can find these invalid vectors.

However, notice that an execution of the method on a single input state cannot show φ to be non-maximal. If the testing tool finds a valid input state (s, v) disallowed by φ, we still cannot say that the feature vector corresponding to the input state is valid. The reason is that there may be *another* invalid input state (s', v') that conforms to the same feature vector. Intuitively, witnessing non-maximality boils down to finding a valid feature vector. This situation is the same as asking whether there *exists* a feature vector disallowed by φ such that *all* input states conforming to the feature vector are valid. The $\exists\forall$ nature of the question is what makes finding counterexamples for maximality hard using test generation. (Even logic-based tools, such as Pex, that use SMT solvers are typically effective/decidable for only $\exists^*$ properties, i.e., quantifier-free formulas.) On the other hand, finding an invalid feature vector (included by by φ) asks whether there *exists* a feature vector allowed by φ such that there does *exist* an invalid input state conforming to the feature vector; this question is an $\exists\exists$ question that can be found using tools such as Pex.

Formalizing precondition generation modulo a testing-based teacher. As explained earlier, for a precondition φ, an invalid input state (allowed by φ) found by a testing-based teacher (TBT) is a witness to the fact that φ is unsafe, i.e., no safe precondition should allow this input state.

Valid input states $((s, v), +)$ found by the TBT but disallowed by the current precondition indicate that the precondition may potentially not be maximal, as it disallows an input state where m does not throw an exception. However, *we do not want to demand that we find a precondition that definitely allows* (s, v). The reason is that such a requirement is too strong as there may be another input state of the form $((s', v'), -)$ that conforms to the same feature vector as (s, v). Another reason is that even if the feature vectors are not the same, the logic may be unable to distinguish between the two vectors. In other words, it may be the case that *no* precondition expressible in our logic is both safe and allows this positive example (s, v).

We next define the notion of an ideal precondition that captures both safety and maximality modulo the incomplete information that the client has of the object state and modulo the expressiveness of the logic, with respect to the TBT. First, let us define some terminology: for any two input states (s, v) and (s', v'), we say (s, v) is $\mathcal{L}$-indistinguishable from (s', v') if there is no formula (in the logic $\mathcal{L}$) that evaluates to true on one of them and false on the other (note that if the two input states conform to the same feature vector, then they are indistinguishable no matter the logic). In a similar way, we define $\mathcal{L}$-indinguishability for feature vectors.

Definition 5.2. *An* ideal precondition *for $m(\vec{p})$ with respect to a TBT is a precondition φ in the logic $\mathcal{L}$ such that φ satisfies the following two conditions:*

- **Safety wrt TBT:** *the TBT returns a set that has no invalid input state allowed by φ.*
- **Maximality wrt TBT:** *for every valid input state $((s, v), +)$ returned by the TBT but disallowed by φ, there is* some *invalid input state $((s', v'), -)$ (returned by the TBT allowed by* some *precondition) that is $\mathcal{L}$-indistinguishable from (s, v).*

Intuitively, the first condition of safety demands that the TBT is not able to find any invalid input state allowed by the precondition (i.e., one on which m throws an exception). The second condition states that for any valid input state (s, v) found by the TBT but disallowed by the precondition, there must be *some* invalid input state (returned by the TBT allowed by *some* precondition, not necessarily φ) that is $\mathcal{L}$-indistinguishable from (s, v).

A precondition on which the TBT returns the empty set is hence also an ideal precondition. Note that in general, there may be no *unique* safe and maximal precondition.
We can now state the precise problem of precondition generation modulo a TBT:

Problem Statement 5.1. Given a program with the method $m(\vec{p})$ and observer methods and a logic $\mathcal{L}$ for expressing preconditions for m, and given a testing-based teacher TBT, find an ideal precondition for $m(\vec{p})$ with respect to the TBT.

5.2.3 The Learning Framework for Synthesizing Preconditions Modulo a Test Generator

In this section, we describe our general learning framework for synthesizing ideal preconditions with respect to a testing-based teacher (TBT).

We first describe this framework (Section 5.2.3.1) and then discuss multiple ways to instantiate the framework (Section 5.2.3.2). Adapting a TBT to realize this framework is discussed in Section 5.2.4. Finally, in Section 5.2.3.3, we discuss general conditions under

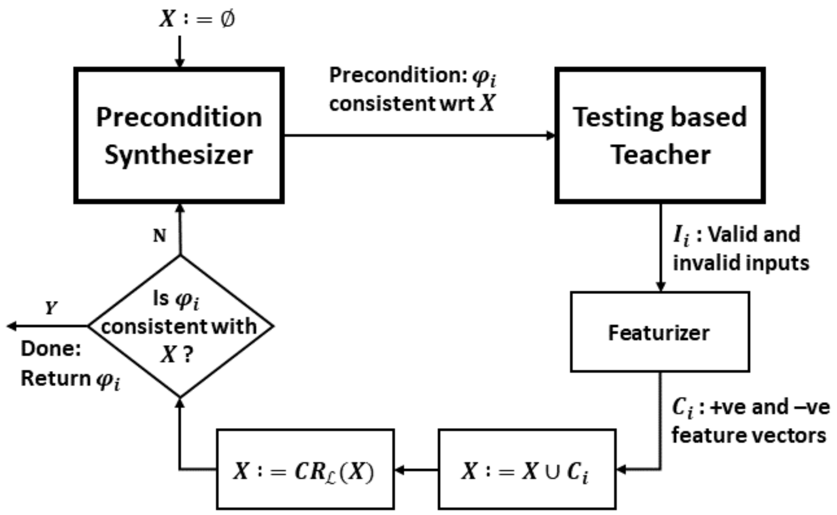

$X := \emptyset$
Precondition Synthesizer
Precondition: φ_i consistent wrt X
Testing based Teacher
I_i : Valid and invalid inputs
Featurizer
C_i : +ve and –ve feature vectors
$X := X \cup C_i$
$X := CR_{\mathcal{L}}(X)$
Is φ_i consistent with X ?
N
Y
Done: Return φ_i

Figure 5.2: The learning framework for synthesizing ideal preconditions with respect to a TBT.

which we can show that our learners and learning framework converge to an ideal precondition with respect to any TBT.

5.2.3.1 Framework Overview

Our learning framework, depicted in Figure 5.2, consists of five distinct components: (1) a passive learner (precondition synthesizer) that synthesizes preconditions from positive and negative feature vectors, (2) a TBT, interacting in rounds of communication with the learner, that returns valid/invalid input states, (3) a featurizer that converts valid/invalid input states to positive/negative feature vectors, and (4) a conflict resolver ($CR_{\mathcal{L}}$), which is the main novel component, that resolves conflicts (created by incomplete information) by changing positive feature vectors to negative ones when necessary. We emphasize that one can use any standard passive learner in this framework as long as it finds formulas that are consistent with the set X of labeled feature vectors.

The framework maintains a set X, which contains the accumulated set of (conflict-resolved) positive/negative labeled feature vectors that the TBT has returned. In each round i, the learner proposes a precondition φ_i that is *consistent* with the set, and the TBT returns a set of valid and invalid input states. The featurizer, with the help of observer-method calls, converts the input states to positive/negative labeled feature vectors C_i. We add the counterexample input states to X and call the conflict resolver for the logic $\mathcal{L}$, and update X. We then check whether the current conjecture φ_i is consistent with the updated X—namely whether φ is true on every positive feature vector and false on every negative feature vector.

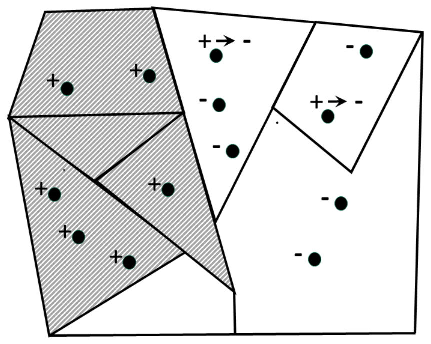


Figure 5.3: An example of conflict resolution where positive vectors are made negative. Partitions denote equivalence classes of indistinguishable vectors; points denote positive and negative feature vectors. The shaded region denotes a consistent precondition.

If it is, then we exit having found an ideal precondition, and if not, we iterate with the precondition synthesizer for the new set X.

Conflict resolver. Formally, the conflict resolver, given a set X of positive and negative feature vectors, returns the set of positive and negative feature vectors such that

- the returned set contains every feature vector (in X) that is negative;
- for any positive feature vector $(\vec{f}, +)$ in X, if there is a negative feature vector $(\vec{f'}, -)$ in X such that $\vec{f}$ and $\vec{f'}$ are $\mathcal{L}$-indistinguishable, then the returned set contains the negative feature vector $(\vec{f}, -)$; otherwise, the set contains the positive feature vector $(\vec{f}, +)$.

To understand why the conflict resolver working as above is a sound way to obtain ideal preconditions, recall the two properties of ideal preconditions in Definition 5.2: safety wrt TBT and maximality wrt TBT. The conflict resolver keeps negative feature vectors as they are (since safety wrt TBT requires that the precondition exclude them). However, when a positive feature vector has a corresponding indistinguishable negative feature vector (returned by the TBT in this round or a previous round), it is clear that *no* precondition expressible in the logic can include the positive feature vector. Hence the conflict resolver turns it negative, which is allowed by the definition of maximality wrt TBT in the definition of ideal preconditions.

Figure 5.3 shows an example of the effect of a conflict resolver— it converts two positive feature vectors to negative ones since they have corresponding negative feature vectors (in X) that are not distinguishable from them. A consistent precondition (shown as the shaded region) consists of some equivalence classes of indistinguishable feature vectors that include the positive vectors and exclude the negative ones, *after conflict resolution*.

Notice that in any set X of counterexamples accumulated during the rounds, X is a subset of the set of all counterexamples that TBT returns on all possible preconditions. Hence it is easy to see that if φ_i is consistent with the conflict-resolved set obtained from $X \cup C_i$, then it is in fact ideal (for every positive counterexample disallowed by φ_i, in X there is a negative counterexample (returned by the TBT) being indistinguishable). Consequently, φ is ideal when the learning framework terminates.

5.2.3.2 Instantiations of the Framework

Our framework can be instantiated by choosing a logic $\mathcal{L}$, choosing any synthesis/learning engine for exactly learning logical expressions in $\mathcal{L}$, and building conflict resolvers for $\mathcal{L}$. We next list multiple such possibilities.

Logic for preconditions. We can instantiate our framework to the logic $\mathcal{L}_{\mathbb{B},\mathbb{Z}}$ described below for expressing preconditions. Let us assume that feature vectors consist of a set of Boolean features $P = \{\alpha_1(\vec{p}), \ldots, \alpha_n(\vec{p})\}$ and a set of integer features $N = \{r_1(\vec{p}), \ldots, r_t(\vec{p})\}$. Note that these features all depend on the parameters $\vec{p}$, and can be either Boolean or integer parameters in $\vec{p}$ or calls to observer methods (using parameters in $\vec{p}$) that return Booleans or integers. The grammar for the logic $\mathcal{L}_{\mathbb{B},\mathbb{Z}}$ of preconditions that we consider is

$$\varphi \ ::- \ \alpha(\vec{p}) \ \mid \ r(\vec{p}) \leq c \ \mid \ \varphi \vee \varphi \ \mid \ \varphi \wedge \varphi \ \mid \ \neg\varphi \tag{5.1}$$

where $\alpha \in P$, $r \in N$, and $c \in \mathbb{Z}$.

We also consider certain sublogics of the preceding logic; one being of particular interest is discussed in Section 5.2.3.3 on convergence, where we require the threshold constants c to be from a finite set of integers B.

Learners. By treating the Boolean and integer features as Boolean and integer variables, we can use exact learning variants of the ID3 algorithm for learning decision trees [59, 75] in order to synthesize preconditions for the logic $\mathcal{L}_{\mathbb{B},\mathbb{Z}}$. It is easy to adapt Quinlan's decision tree learning algorithm (which synthesizes small trees using a greedy algorithm guided by statistical measures based on entropy) to an exact learning algorithm [46]. In our evaluation (Section 5.2.5), we mainly use such a learner.

A second and more expressive choice is to use passive learners expressed in the syntax-guided synthesis framework (SyGuS [1]). This framework allows specifying a logic syntactically (using standard logic theories) and allows a specification expressing properties of the formula to be synthesized. By making this specification express that the formula is consistent with the set of samples, we can obtain a passive learner that synthesizes expressions. The salient feature here is that instead of having a fixed set of predicates (like in the preceding decision-tree algorithm), predicates are dynamically synthesized based on the samples. There are multiple solvers available for the SyGuS format, as it also is part of a synthesis competition, and learners based on stochastic search, constraint solving, and combinations with machine learning are known [65, 157, 158]. In fact, one recent tool named PIE [133] is similar to a SyGuS solver and can be used as a passive learner too. We have, in our evaluation, tried multiple SyGuS solvers and also the PIE passive learner.

Conflict resolvers. Conflict resolver algorithms crucially depend on the logic. For the preceding logic $\mathcal{L}_{\mathbb{B},\mathbb{Z}}$ with Boolean and integer features, it is easy to see that any two feature vectors that are different are in fact separable using the logic, as each vector can be isolated from the rest. Consequently, the conflict resolver simply changes a positive feature vector to negative iff the same feature vector also occurs negatively in the set X.

Consider now the same preceding logic $\mathcal{L}_{\mathbb{B},\mathbb{Z}}$ but where we require the threshold constants c to be *bounded*—i.e., $|c| < b$, where b is a fixed natural number. It is easy to see that a conflict resolver for this logic needs to turn a positive feature vector $\vec{f}$ to negative iff there is a negative feature vector $\vec{g}$ that agrees with $\vec{f}$ on all Boolean features and, for each integer feature, either $\vec{f}$ and $\vec{g}$ both have the same feature value or the feature values in $\vec{f}$ and $\vec{g}$ are both larger than b or both smaller than $-b$. The implementation of this algorithm is straightforward.

5.2.3.3 Convergence of Learning

We argue earlier that if the learning framework, instantiated with any learner, terminates, then it has computed an ideal precondition. In this section, we consider settings where the learning framework is also *convergent* (i.e., is guaranteed to terminate).

Let us fix a testing-based teacher TBT and let us assume that there is (at least) one target concept $\varphi*$ in $\mathcal{L}$ such that if C is the set of all counterexamples returned by the TBT (in response to any possible precondition), then $\varphi*$ is consistent with C.

We consider the case when the hypothesis space H of preconditions is *finite*, i.e., when the number of possible preconditions is finite, and when the logic is closed under Boolean operations. For the logic $\mathcal{L}_{\mathbb{B},\mathbb{Z}}$, this finite space naturally occurs when the features are all Boolean or when we fix a certain finite set of constants for thresholds for numerical inequalities, e.g., $[-b, +b]$ for some $b \in \mathbb{N}$. We can now show that our learning framework where the learner is *any* learner that learns consistent formulas is guaranteed to find an ideal precondition with respect to the TBT.

Theorem 5.1. *Let the logic for preconditions be any finite hypothesis space of formulas $\mathcal{H}$ that is closed under conjunctions, disjunctions, and negation. Consider any instantiation of the learning framework with any conflict resolver and any learner that always returns a concept consistent with all the given labeled feature vectors, if one exists. Then for any method $m(\overline{p})$, the learning framework is guaranteed to terminate and return an ideal precondition for m, provided that m has an ideal precondition expressible in the logic.*

Proof. We argue that the learner can always return a hypothesis consistent with the samples in each round, and that when it first *repeats* the conjecture of a hypothesis H, the hypothesis H must be an ideal precondition. The reason for the latter is that when H was first proposed, the teacher returned a set of counterexamples. Later, if the learner proposed H, it must be that H is consistent with those counterexamples; this situation would happen since in the interim when H was proposed, the teacher would have returned at least one indistinguishable negative counterexample for each positive counterexample disallowed by H. Hence H would be ideal. Given that the hypothesis space is finite, the learner must eventually repeat a hypothesis, and hence always converges.

The reason why any consistent learner always finds some logical formula that satisfies the set of (conflict-resolved) samples is as follows. First, let $\hat{\mathcal{H}}$ denote the tightest preconditions, and hence any hypothesis in $\mathcal{H}$ is a disjunction of preconditions in $\hat{\mathcal{H}}$. The preceding is true since the logic is closed under Boolean operations. Let $\equiv$ be an equivalence relation on the set of input states that relate any two input states not distinguishable by the formulas in $\mathcal{H}$ (equivalently by $\hat{\mathcal{H}}$). Then we know that the conflict resolver would ensure that feature vectors in each equivalence class are pure—that there are no positive and negative vectors in the same class. Consequently, the disjunction of the formulas corresponding to the equivalence classes containing the positive samples is consistent with the samples, and is in $\mathcal{H}$. This concludes the proof. Q.E.D.

5.2.4 Construction of a Testing-Based Teacher

In this section, we describe our techniques for adapting a test generator to a testing-based teacher (TBT) that actively tries to find counterexamples to safety and maximality of given preconditions. We also describe how the featurizer can be implemented.

5.2.4.1 Extracting Counterexamples

The first issue is to adapt the test generator to return negative counterexample inputs for showing that a precondition is not safe and positive counterexample inputs for showing that a precondition is potentially not maximal. A test generator's goal is slightly different than a TBT's (see Definition 5.1). Given a method, $m(\vec{p})$, and a precondition, φ, the goal of a test generator is to find samples of the form (s, v) allowed by φ, and to generate valid and invalid input states, typically trying to find invalid ones.

Extracting negative counterexamples is easy—we keep the same precondition (the precondition needs to be evaluated by calling the various observer methods) and we ask the test generator to find inputs that cause exceptions. Valid and invalid inputs found by the test generator can be returned.

To extract positive counterexamples, we instrument the method as follows:

- replace the precondition φ with its negation $\neg\varphi$,
- for every `assert` statement, we insert an `assume` statement for the same condition right before the `assert` statement,
- add an `assert(false)` statement at the end of the method, and before every `return` statement (if any).

The valid/invalid inputs found by the test generator for the instrumented method are returned (as valid/invalid inputs to the original method).

5.2.4.2 Implementing the Featurizer

To form feature vectors from inputs generated by the test generator, we insert additional statements at the beginning of the method for computing the features. The features are computed by calling the various observer methods and storing their return values in variables of appropriate type.

Although in theory we assume that observer methods are pure, this assumption may not always be true in practice. In our evaluation, we manually ensure that the chosen observer methods are pure.

5.2.5 Evaluation

We prototype an implementation of our framework, called the Proviso tool, for synthesizing preconditions for C# programs. We adapt an industrial test generator, Pex [76],

Project / Classes	**#Classes**	**#LOC**
.NET Data structures: Stack, Queue, Set, Map, ArrayList	46	14886
QuickGraph: Undirected Graph, Binary Heap	319	34196
Lidregen Network: NetOutGoingMessage, BigInteger	59	14042
DSA: Numbers	60	5155
Hola Benchmarks	1	933
Code Contract Benchmarks	1	269

Table 5.1: Statistics of evaluation subjects

to a testing-based teacher, choose the logic $\mathcal{L}_{\mathbb{B},\mathbb{Z}}$, over Booleans and integers (introduced in Section 5.2.3.2), and a variant of Quinlan's C 5.0 decision tree algorithm [59, 75] to an exact learner [46]. The conflict resolver is the one for $\mathcal{L}_{\mathbb{B},\mathbb{Z}}$ described in Section 5.2.3.2. We instantiate the framework for two tasks of specification inference: learning preconditions for preventing runtime failures and learning conditions for method commutativity in data structures. The Proviso tool terminates only when it finds an ideal precondition modulo the test generator.

In our evaluation, we intend to address the following main research question:

How effectively can the Proviso tool learn truly *ideal preconditions?*

The purpose of this research question is to investigate how effective our framework is in learning preconditions that are *truly* ideal—truly safe and truly maximal. In Section 5.2.3, we show that our learning algorithm when it terminates will converge on a safe and maximal precondition, with respect to the test generator. However, it may be the case that the learned precondition is neither safe nor maximal when compared to the ground truth (as determined by a programmer examining the code). This situation can happen for multiple reasons: ineffectiveness of the test generator to generate counterexamples, lack of observer methods to capture sufficient detail of objects, and inexpressiveness of the logic to express the right preconditions. To answer this research question, we manually inspect all of the cases and derive ground truths to the best of our abilities and compare them with the preconditions synthesized by Proviso.

Evaluation setup. We evaluate our framework on a combination of small and large projects studied in previous work related to precondition inference [133, 159, 160] and test generation [161, 162]. We consider classes with methods from these projects whose parameters are of primitive types currently supported by our learner (i.e., int, bool) or whose parameters are of complex types that have observer methods (defined in their interface) whose return types are int or bool. For the task of learning preconditions for preventing runtime failures, our evaluation subjects include (1) two open source projects, Lidregen Network and Data Structures and Algorithms (DSA), and (2) a set of Code Contract benchmarks from the cccheck static analyzer [163] and benchmarks from the Hola engine [120]. For the task of learning conditions for method commutativity in data structures, our evaluation subjects include data structures available in two open source projects, QuickGraph and .NET Core. Table 5.1 shows the data structures/classes used as our evaluation subjects. The table also shows the number of classes and the size of code for the entire project (the individual methods that we consider in our evaluation are smaller, but they can call various other methods and our test generator does analyze the larger code base).

In total, our evaluation subjects include 105 method pairs for learning commutativity conditions and 121 methods for learning conditions to prevent runtime failures.

For each data structure and non-primitive type, we implement abstract equality methods and factory methods. The equality methods compare object states for equivalence, and the factory methods (which Pex exploits) create objects from primitive types. For each method or method pair, we use all and only the public observer methods in the interface of their respective class.

Table 5.2 summarizes our evaluation results, including statistics on our subjects, statistics on learning, and details on validation with respect to Randoop [164] (a test generator) and ground truth.

5.2.5.1 Learning Ideal Preconditions

We assess the effectiveness of our framework in two main aspects: one being *quality* of the learned preconditions while the other being the efficiency of precondition learning.

Quality of learned preconditions. We examine in two ways whether the learned preconditions are indeed truly ideal. First, we use another test generator compatible with C# programs, namely Randoop [164], to check whether a precondition is safe. If Randoop can generate an invalid input allowed by the learned precondition, then it is clear that the learned precondition is not truly safe (despite the fact that Pex did not find such input). After this first step, we manually inspect each case where Randoop cannot generate inputs to show unsafety, deriving the truly ideal precondition manually and checking whether it is equivalent to the learned precondition.

Subjects				**Learning Framework**				**Validation**			
								Randoop			**Manual**
Project/Class	**LOC**	**Met.**	**Obs.**	**#CE**	**#Rnd.**	**Size**	**Time(s)**	**#Test**	**#Fail**	**#Safe**	**#Corr.**
Commutativity											
Stack	502	10	3	7.9	4.9	1.0	418.2	10117	0	10	8
Set	1847	10	2	11.3	2.5	2.1	493.3	10922	0	10	10
Queue	584	10	3	21.9	10.3	1.0	1646.5	10020	0	10	8
Map	1382	10	3	30.5	5.1	2.7	1230.1	8809	13	9	9
ArrayList	2963	10	4	12.0	4.7	2.6	1212.0	9072	75	9	9
Undirected Graph	327	36	7	13.7	7.0	2.7	1045.2	5708	104	30	16
Binary Heap	335	19	4	42.2	4.6	6.2	872.2	7456	25	17	17
Exceptions											
NetOutGoingMessage	785	47	3	9.5	2.8	1.7	515.0	1067	70	44	42
BigInteger	2334	39	2	13.6	3.7	2.6	214.5	2214	178	38	34
Numbers	284	4	0	20.3	60.5	1.8	4589.2	3626	0	4	2
CodeContract	269	21	0	22.4	16.0	1.6	376.9	14170	0	21	21
Hola	933	10	0	47.5	20.9	2.9	428.7	19236	0	10	7

Table 5.2: Evaluation results on benchmarks and open source programs using Proviso. Abbreviations: LOC=total number of Lines of Code of the class, Met.=total number of methods/method pairs, Obs.=total number of observer methods, #CE=average number of counterexamples, #Rnd.=average number of rounds, Size=average size of the preconditions, time=average time taken per method (in seconds), #Test=total number of tests generated by Randoop, #Fail=total number of failing tests found by Randoop, #Safe=number of methods/method pairs whose preconditions are found safe by Randoop, #Corr.= number of methods/method pairs found by manual inspection to be truly safe and maximal.

Our results shown in Table 5.2 suggest that learning modulo a test generator can be effective in learning truly safe and maximal preconditions, despite the test generator's incompleteness.

Out of the 105 commutativity cases, we find that Proviso can learn 73 (∼70%) truly safe and maximal preconditions. In addition, Proviso learns 24 other preconditions that are only truly safe. Overall, ∼92% of the preconditions learned by Proviso for the commutativity cases are truly safe. For the 121 exception-prevention cases, we find that Proviso can learn 105 (∼87%) truly safe and maximal preconditions. Proviso learns additional 4 preconditions that are only truly safe.

In multiple cases, Proviso does not learn a truly ideal precondition due to lacking appropriate observer methods. For example, a commutativity precondition synthesized for a .NET benchmark involves checking whether a setter and getter on a dictionary commute, and Proviso learns a precondition that is neither truly safe nor truly maximal. However, if we implement an additional observer method `ContainsValueAt(x)`, which returns the value at x, then Proviso learns `(s1.ContainsValueAt(x) && s1.ContainsKey(y)) || (!(x == y) && s1.ContainsKey(y))`, which is a truly safe and maximal precondition.

Another example is the commutativity of methods `peek()` and `pop()` in a stack—they commute when the top two elements in the stack are identical. However, this property turns out to be not expressible using the available observer methods and the learned precondition is *false*.

In most cases, however, Proviso does learn truly safe and maximal preconditions that are natural and easy to read. For example, for the commutativity of `push(x)` and `push(y)`, Proviso learns the precondition `x == y`, which is indeed truly safe and maximal.

For learning preconditions to prevent runtime failures, Proviso performs very well. Proviso performs well also on the larger open source programs in Lidregen Network in terms of both correctness and time spent in learning. A particular case of interest in DSA is where Proviso is able to learn a truly ideal precondition [`number < 1024`] for the `toBinary` method, which converts an integer to its binary representation (the constant 1024 is discovered by Proviso). For values of 1024 or higher, an integer overflow exception occurs deep in .NET library code.

Efficiency of precondition learning. We also measure the time efficiency of Proviso in learning preconditions. Proviso takes on average ~740 seconds per method/method pair to synthesize preconditions.

5.2.5.2 On Empirical Comparison with Related Work

It is hard to provide a fair comparison with two closely related approaches by Padhi et al. [133] and Gehr et al. [165]. The approach by Gehr et al., strictly speaking, does not learn preconditions. It learns conditions under which two methods have commuted after their execution; the learned conditions are expressed over primitive-type parameters *and return values* of these two methods (note that, by definition, preconditions for these two methods should not be expressed using their return values). In addition, the learned conditions cannot capture properties of object states.

The approach by Padhi et al. [133] learns preconditions while also synthesizing auxiliary predicates. In this case, the languages for the programs are different (ours is for C# while theirs is for OCaml), and a direct tool comparison is hard. However, since our framework allows any passive learner to be plugged in, we plug in the passive learner used by Padhi et al. [133, PIE] and re-produce our evaluation results. The results show that when the feature sets are fixed, Proviso equipped with Padhi et al.'s learner has similar effectiveness as Proviso (by default equipped with the decision-tree learner), but when features are not provided, Proviso equipped with Padhi et al.'s learner takes much longer time and even diverges in some cases.

5.3 SYNTHESIZING CONJUNCTIVE POSTCONDITIONS

Synthesizing postconditions can be seen as one form of specification mining, which has applications in various fields. The precondition and the postcondition of a method forms its contract, such that if the method is executed in a state that satisfies the precondition, then it is guaranteed to terminate in a state satisfying the postcondition. While preconditions are used to define the input state space on which the method would terminate without throwing an exception or causing an assertion violation, postconditions are used to summarize the behavior of the method, capturing it as a logical formula. Such summarization using the contracts is especially useful in verification, as methods can be verified modularly, by substituting each method call by its contract instead of inlining. Moreover, code snippets for a variety of tasks are available online on websites like GitHub or StackOverflow. Summarizing such code snippets by synthesizing their contracts, can allow us to search/autocomplete the implementation of a method just from its specification.

In this section, we propose a CEGIS based learning framework to synthesize conjunctive postcondition of methods using a test generator as the oracle. The synthesized postconditions need to be *strong*, and ideally the strongest postcondition expressible in a given logic. We assume that, along with other annotations, the method has been annotated with a precondition that prevents all exception failure. This can be done using the precondition synthesis techniques mentioned in earlier sections.

Our framework works as follows: in each round, the learner proposes a new postcondition. The teacher asserts the postcondition at the end of the method and generates pairs of input and output states such that the input states satisfy the precondition. Hence, all the counterexamples in this framework are labeled positively. We next use a SyGuS solver to synthesize predicates, which are equality expressions that define an output as a function of the input. Finally, we use the classical elimination algorithm to learn conjunctive postconditions from the synthesized predicates. The elimination algorithm fits perfectly in this scenario as it can learn conjunctive concepts from positive counterexamples. The conjunction it learns is the semantically tightest formula (or the formula with the most predicates), which prevents our framework from learning trivial postconditions.

5.3.1 Problem Formulization

In this section, we formalize the problem of synthesizing postconditions with the aid of a test generator.

Recall from Section 5.2.2, that we have a method $m(\vec{p})$ with formal input parameters $\vec{p}$, where $\vec{p}$ consists of primitive type input parameters and objects of classes. Let us assume the class the method m is a member of and classes of the objects in the input parameters, have a set of *pure* (i.e., side-effect free) observer methods that return primitive types. These observer methods query the properties of the state of the objects. Let us also assume that, along with other annotations, the method has been annotated with the precondition synthesized using the techniques mentioned earlier sections.

We formalize the test generator as a deterministic function that given a method returns pairs of feasible input-output states. Each input state consists of the values of the input parameters at the beginning of the method. Note that the input states satisfy the precondition. The output states consist of the valuation of all the variables and objects at the end of the method. The output states also include the updated valuations of the input parameters and the value of a special variable *ret* containing the return values of the method.

We define feature vectors as an abstraction of the input states or the output states. The abstraction replaces an object by the values of its observer methods while keeping all the primitive type values unchanged. Hence, the feature vectors only contain primitive type values (integer and Boolean in our tool). Let us call the feature vector constructed from an input state as fv_{in}, and fv_{out} for the output state. Let us call the variables corresponding to the features in the feature vector fv_{in} as $vars_{in}$, and $vars_{out}$ for the feature vector fv_{out}. Let $vars$ be the union of $vars_{in}$ and $vars_{out}$.

The counterexamples in this framework are the abstractions of pairs of input and output states. In other words, a counterexample is a concatenation of the feature vectors fv_{in} and fv_{out}. A counterexample is labeled *negative* if the corresponding input and output states do not cause an exception or an assertion failure. However, as the precondition prohibits the test generator from creating such a pair of input and output states, all the counterexamples in this framework are labeled *positive*.

In this framework, we aim to synthesize a conjunctive postcondition φ, which is a quantifier-free first-order logic formula over the free variables in $vars$. Additionally, each atomic predicate in this formula φ can contain equality, and if-then-else(ite) expressions along with the usual functions, relations, and constants in the underlying logic.

We can now state the problem statement:

Problem Statement 5.2. Given a method $m(\vec{p})$ with input parameters $\vec{p}$, an ideal precondition for this method, a logic $\mathcal{L}$ for expressing postconditions, and a test generator, find the strongest postcondition for $m(\vec{p})$ expressible in the given logic $\mathcal{L}$.

5.3.2 Learning Framework for Synthesizing Postconditions

We now describe the general learning framework to synthesize postconditions for methods. Our framework has the following components: (1) a test generator that given a hypothesis postcondition, generates pairs of input and output states such that the input states satisfy the precondition, (2) a featurizer that abstracts these input states into feature vectors and hence, generates the counterexamples, (3) a predicate synthesizer, and (4) a learner for conjunctive Boolean formulas that learns from positive examples. The framework maintains a global set CE of counterexamples to accumulate them across rounds. In every round, the learner proposes a postcondition consistent with the set of counterexamples CE. The test generator then returns pairs of input and output states, which are abstracted into counterexamples by the featurizer and added to the set CE. We next invoke the predicate synthesizer to synthesize predicates on the variables in $vars$. The conjunctive learner, which consists of the elimination algorithm, is invoked next, thus proposing a new hypothesis postcondition for the next round, and the loop continues. We declare a proposed postcondition φ as the solution if the test generator cannot find any new counterexamples.

5.3.2.1 Synthesizing Predicates

We next synthesize a set of predicates on the variables in $vars$. We synthesize two classes of template-based predicates. For the first class, we enumerate all possible octagonal constraints on the variables in $vars$. Let us call this set of predicates P_{oct}. Here each predicate consists of two variables and a constant, all of whose magnitude range from $+1$ to -1. Formally,

$$P_{oct} = \{s_1 v_1 + s_2 v_2 \leq c,\ s_1, s_2, c \in \{1, 0, -1\},\ v_1, v_2 \in vars,\ v_1 \neq v_2\} \tag{5.2}$$

We synthesize the second class of predicates using external synthesis engines. Often in our experiments, we found that equality predicates, which define an output as a function of the input, describe the intended behavior of the program in a better way, and hence result in stronger postconditions. For example, in the method *pop* of the class *stack*, adding the predicate $size_{out} = ite(size_{in} > 0, size_{in} - 1, size_{in})$, where ite denotes if-then-else expressions, $size_{in}$ and $size_{out}$ denote the value of the variable $size$ in the input state and the output state, respectively. This predicate captures the functional behavior of the method more accurately.

Hence, for every variable $v \in vars_{out}$, we call a SyGuS solver to synthesize a function fn on all the variables in $vars_{in}$, such that $v = fn(vars_{in})$ holds. If the SyGuS solver is able to synthesize such a function, we add this expression as an equality predicate; else, we skip this predicate. We call this set of predicates P_{eq}. Note that the variable denoting the return value of the method *ret* is also included in the set of variables $vars_{out}$.

The SyGuS solvers allow us also to specify a grammar to restrict the search space and also to control the syntax of the synthesized function. In our tool, we specified the syntax of the functions to be a tree of height at most 1. The leaf nodes are a linear combination of the variables with coefficients and constants from a bounded range. We start the synthesis process with a small bound ([-1,1] in our tool), and increase it for every subsequent round of the framework. The conditional nodes consist of comparisons between the variables.

The call to the SyGuS solver will always terminate. This is because, as the number of variables is finite, and coefficients and constants of the linear expression are from a bounded range, the search space of candidate functions is also finite. Also, if the solution requires a linear expression with higher coefficients (or constants), then the SyGuS solver will eventually find it in subsequent rounds as the bounds are increased.

5.3.2.2 Learning Conjunctive Boolean Formulas

Once we have synthesized the predicates and created the set P, we then evaluate them on all the counterexamples accumulated across rounds in the set CE, resulting in a set containing vectors of Boolean values, which constitutes our sample for learning. Formally if $P = \{p_1, p_2, \ldots, p_n\}$, and each counterexample $c \in CE$ is of the form $(c_1, c_2, \ldots, c_m)$ (note that $|vars| = m$), then the sample $S = \bigcup_{c \in CE} \{(p_1(c), p_2(c), \ldots, p_n(c))\}$, where $p_i(c)$ evaluates the predicate p_i on the counterexample c. Note that the predicates have as free variables the variables from the set $vars$, and the counterexamples are concrete valuations of the variables in $vars$. Thus, each such evaluation results in a Boolean value.

Due to the nature of our problem where we have only positive counterexamples, and we want to learn the *tightest* conjunction on a set of Boolean variables, the classical *elimination algorithm* poses an excellent fit. The elimination algorithm learns the largest conjunctive formula in terms of the number of predicates in the final conjunct. This also implies that the formula is semantically the tightest. Learning the tightest functional concept is useful in our scenario, as any superset of the tightest set is also a solution and contains less information than the tightest set, with *true* being the most trivial postcondition. The elimination algorithm guarantees convergence in at most $|P|$ rounds, which performs extremely well in practice.

Elimination Algorithm: Let us now describe the elimination algorithm. Given a sample S, the elimination algorithm first assigns P, the set containing all the predicates, to a temporary variable X. The algorithm then proceeds as follows:

Datastructure	**Met.**	**Obs.**	**#CE**	**#Rnd.**	**Size**	**Teacher Time(s)**	**Learner Time(s)**	**Total Time(s)**
Stack	5	3	10.0	3.0	2.8	107.9	5.2	113.1
Queue	5	3	15.0	3.0	3.2	67.8	2.9	70.7
Hash Set	4	2	14.3	2.3	2.0	268.3	1.8	270.1
Dictionary	7	3	17.3	2.6	3.1	284.9	2.9	287.8
Array List	9	4	13.7	3.0	7.2	67.0	5.9	72.9
Binary Heap	7	4	37.7	3.0	6.3	205.2	14.5	219.7
Undirected Graph	4	7	20.5	3.0	7.3	369.5	8.2	377.7

Table 5.3: Evaluation results on benchmarks from open source projects using Precis. Abbreviations: Met.=total number of methods, Obs.=total number of observer methods, #CE=average number of counterexamples, #Rnd.=average number of rounds, Size=average number of conjuncts in the postconditions, Teacher(Learner) time= average time taken by teacher(learner) per method (in seconds), Total time=average time taken per method (in seconds).

1. It removes all predicates $p \in X$ from X that violates one of the counterexamples, i.e., the predicate p evaluates to *false* when evaluated on a counterexample $c \in S$.

2. Continue step 1 until a fixed point is reached. Once this happens, X is the unique largest set of predicates that is consistent with the sample S. The final postcondition is the conjunction of all the predicates in X. If the set X of predicates becomes empty, no conjunctive formula exists that is consistent with the sample, and *true* is returned as the postcondition.

It is not hard to verify that the time the elimination algorithm spends in each round is *polynomial* in the number of predicates in P, and the number of counterexamples in the sample S (provided predicates can be evaluated in constant time).

5.3.3 Evaluation

We implemented a prototype of our framework, in a tool called Precis, to synthesize conjunctive postconditions of C# programs. Precis uses an industrial test generator PEX [76] over the logic of Booleans and integers. We implemented the elimination algorithm in Python. To synthesize the functional components of the equality predicates, we used the enumerative solver [65] from the SyGuS competitions [1]. We specified a grammar to synthesize base expressions or if-then-else(ite) expressions with base expressions as its leaves. The base expressions are linear integer arithmetic expressions on the variables with bounded coefficients, while the conditionals of the ite expressions consist of simple comparisons between the variables. We also implemented a module to simplify linear integer arithmetic formulas, using the various tactics provided by the Z3 [88] SMT solver.

We evaluated our framework on datastructure methods from two open-source projects QuickGraph and the .NET Core. Table 5.1 summarizes the datastructure classes used from these projects. In our evaluation, we used 41 methods from these classes. For every method, we first synthesized their preconditions using our tool Provisio and then added the preconditions as *assume* statements at the beginning of the methods. For each of the classes, we only used public observer methods in their interface.

All experiments were performed on a system with an Intel Core i7 quad-core CPU with 1.8GHz frequency, and 8GB RAM, running 64-bit Windows 10 OS. Table 5.3 summarises our evaluation results. Precis was able to synthesize postconditions of reasonable size, very efficiently taking an average of $\sim 200s$ per method. The majority of the time was spent by PEX to generate counterexamples, while the learning phase took comparatively negligible time. The postconditions synthesized by Precis were of reasonable length (average of 4.6 conjuncts), and it took on average three rounds per method. We also manually verified all the postconditions to verify their validity.

5.4 RELATED WORK

Black-box approaches. Ernst et al. [166] proposed Daikon for dynamically detecting *conjunctive* Boolean formulas as likely invariants from black-box executions that gather runtime state information (method-entry states, method-exit states); Daikon, seen as a learning algorithm, learns using only positive counterexamples, and unlike our approach, does not make any guarantees of safety or maximality.

Our work is most closely related to three black-box approaches by Padhi et al. [133], Gehr et al. [165], and Sankaranarayanan et al. [167]. The last two approaches [165, 167] rely on generating test inputs from sampling feasible truth assignment of input predicates or assignments satisfying representative formulas in a particular logic, followed by Boolean learning from positive and negative examples to infer preconditions. However, these approaches do not provide any guarantees unlike our work, where we guarantee that the final learned precondition is both safe and maximal with respect to a testing-based teacher. Padhi et al. [133] proposed a data-driven learning approach based on feature learning, including black-box and white-box components. Its black-box component, PIE, learns a formula from a fixed set of tests. Its white-box component, VPreGen, includes an iterative refinement algorithm that uses counterexamples returned by a verifier to learn provably safe preconditions. However, the white-box component does not make any guarantees on maximality as we do.

Furthermore, to assure that preconditions are provably safe, inductive loop invariants must be synthesized, further complicating the problem. In our approach, we replace the verifier with a testing-based teacher for practical reasons and handle the accompanying challenges.
Program and expression synthesis. The field of program synthesis deals with the problem of synthesizing expressions that satisfy a specification.

One of the most promising approaches of synthesizing expressions is counterexample-guided inductive synthesis (CEGIS) [1], which in fact resembles online learning. In this setting, the target expression is learned in multiple rounds of interaction between a learner and a verifier. In each round, the learner proposes a candidate expression and the verifier checks whether the expression satisfies the specification, and returns counterexamples otherwise. In this sense, we can view our algorithm also as a CEGIS algorithm, but where the verifier is replaced by an incomplete testing-based tool. However, there are technical differences — in program synthesis, the aim would be to find a formula that precisely classifies the examples, while in our setting, we are required to learn a classifier that classifies negative examples precisely, but is allowed to negatively classify positive examples. Furthermore, we require that a minimal number of positive counterexamples are classified negatively; such maximality constraints are not the norm in program synthesis (indeed some problems involving maximality have been recently considered [168]).
Decision-tree learning. Decision-tree learning has been used in several contexts in program synthesis before — in precondition synthesis [167], in invariant synthesis [46, 126], in synthesizing piece-wise linear functions [157], etc. Many of these algorithms have had to change the ID3 algorithm, similar to our work, so that the algorithm learns a tree consistent with the samples. The crucial differences in our framework from such previous work are that we dynamically modify the classifications of samples from positive to negative when we discover conflicting counterexamples, and ensure maximality of preconditions by learning across rounds using inputs from the testing-based teacher.

5.5 CONCLUSIONS AND FUTURE WORK

In this chapter, we have presented a novel formalization for the inference problem of stateful preconditions modulo a test generator. In this formalization, the quality of the precondition is based on its safety and maximality with respect to the test generator. We have further proposed a novel iterative active learning framework for synthesizing stateful preconditions, and a convergence result for finite hypothesis spaces.

To assess the effectiveness of our framework, we have instantiated our framework for two tasks of inference and evaluated our framework on various C# classes from well-known benchmarks and open source projects. The results demonstrate the effectiveness of the proposed framework.

We also proposed a learning framework to synthesize conjunctive postconditions using a test generator as the oracle. We used SyGuS solvers to synthesize the predicates and used the elimination algorithm to synthesize the postcondition. We implemented the tool PRECIS, which performed effectively on many datastructure benchmarks. One important future research direction is to extend this work for disjunctive concepts, hence learning all Boolean formulas.

REFERENCES

[1] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Dependable Software Systems Engineering*, ser. NATO Science for Peace and Security Series, D: Information and Communication Security. IOS Press, 2015, vol. 40, pp. 1–25.

[2] A. Solar Lezama, "Program synthesis by sketching," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2008.

[3] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," *ACM Sigplan Notices*, vol. 41, no. 11, pp. 404–415, 2006.

[4] S. Gulwani, J. Hernández-Orallo, E. Kitzelmann, S. H. Muggleton, U. Schmid, and B. Zorn, "Inductive programming meets the real world," *Commun. ACM*, vol. 58, no. 11, pp. 90–99, Oct. 2015.

[5] S. Gulwani, O. Polozov, R. Singh et al., "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017.

[6] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, vol. 11, 2011, pp. 62–73.

[7] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Communications of the ACM*, vol. 55, no. 8, pp. 97–105, 2012.

[8] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, "Search-based program synthesis," *Communications of the ACM*, vol. 61, no. 12, pp. 84–93, 2018.

[9] S. Gulwani, "Automating string processing in spreadsheets using input-output example," in *ACM Sigplan Notices*, vol. 46, no. 1. ACM, 2011, pp. 317–330.

[10] V. Le and S. Gulwani, "Flashextract: a framework for data extraction by examples," in *ACM SIGPLAN Notices*. ACM, 2014, pp. 542–553.

[11] R. Singh and S. Gulwani, "Transforming spreadsheet data types using examples," in *Acm Sigplan Notices*, vol. 51, no. 1. ACM, 2016, pp. 343–356.

[12] R. Singh and S. Gulwani, "Synthesizing number transformations from input-output examples," in *International Conference on Computer Aided Verification*. Springer, 2012, pp. 634–651.

[13] R. Singh and S. Gulwani, "Learning semantic string transformations from examples," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 740–751, 2012.

[14] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 317–328.

[15] S. Gulwani, M. Mayer, F. Niksic, and R. Piskac, "Strisynth: synthesis for live programming," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 701–704.

[16] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software.* ACM, 2014, pp. 173–184.

[17] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers, "Using program analysis to improve database applications." *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 48–59, 2014.

[18] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan, "Automated grading of dfa constructions," in *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[19] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 15–26, 2013.

[20] S. Gulwani, "Example-based learning in computer-aided stem education." *Commun. ACM*, vol. 57, no. 8, pp. 70–80, 2014.

[21] S. Gulwani, V. A. Korthikanti, and A. Tiwari, "Synthesizing geometry constructions," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 50–61.

[22] G. Juniwal, A. Donzé, J. C. Jensen, and S. A. Seshia, "Cpsgrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory," in *2014 International Conference on Embedded Software (EMSOFT).* IEEE, 2014, pp. 1–10.

[23] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *International Conference on Computer Aided Verification.* Springer, 2016, pp. 383–401.

[24] S. Saha, S. Prabhu, and P. Madhusudan, "Netgen: synthesizing data-plane configurations for network policies," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research.* ACM, 2015, p. 17.

[25] J. McClurg, H. Hojjat, P. Černỳ, and N. Foster, "Efficient synthesis of network updates," in *Acm Sigplan Notices*, vol. 50, no. 6. ACM, 2015, pp. 196–207.

[26] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic program optimization," *Communications of the ACM*, vol. 59, no. 2, pp. 114–122, 2016.

[27] H. Massalin, "Superoptimizer: a look at the smallest program," in *ACM SIGARCH Computer Architecture News.* IEEE Computer Society Press, 1987, pp. 122–126.

[28] S. Bansal and A. Aiken, "Binary translation using peephole translation rules," May 4 2010, uS Patent 7,712,092.

[29] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati, "Scaling up superoptimization," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2. ACM, 2016, pp. 297–310.

[30] P. Černỳ, K. Chatterjee, T. A. Henzinger, A. Radhakrishna, and R. Singh, "Quantitative synthesis for concurrent programs," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 243–259.

[31] M. Vechev, E. Yahav, and G. Yorsh, "Abstraction-guided synthesis of synchronization," in *POPL*, vol. 10, 2010, pp. 327–338.

[32] M. Kuperstein, M. Vechev, and E. Yahav, "Automatic inference of memory fences," *ACM SIGACT News*, vol. 43, no. 2, pp. 108–123, 2012.

[33] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax-and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 593–604.

[34] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 691–701.

[35] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *International conference on computer aided verification*. Springer, 2005, pp. 226–238.

[36] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[37] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Acm Sigplan Notices*. ACM, 2014, pp. 419–428.

[38] N. Polikarpova, I. Kuraj, and A. Solar-Lezama, "Program synthesis from polymorphic refinement types," in *ACM SIGPLAN Notices*. ACM, 2016, pp. 522–538.

[39] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, "Complete completion using types and weights," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 27–38.

[40] H. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge, and W. Hu, "Bing developer assistant: improving developer productivity by recommending sample code," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 956–961.

[41] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-directed completion of partial expressions," in *ACM Sigplan Notices*, vol. 47, no. 6. ACM, 2012, pp. 275–286.

[42] A. Muscholl and I. Walukiewicz, "Distributed synthesis for acyclic architectures," *arXiv preprint arXiv:1402.3314*, 2014.

[43] P. Madhusudan and P. S. Thiagarajan, "Distributed controller synthesis for local specifications," in *International Colloquium on Automata, Languages, and Programming.* Springer, 2001, pp. 396–407.

[44] A. V. Nori, S. Ozair, S. K. Rajamani, and D. Vijaykeerthy, "Efficient synthesis of probabilistic programs," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 208–217.

[45] 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.

[46] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *ACM Sigplan Notices*, vol. 51, no. 1. ACM, 2016, pp. 499–512.

[47] R. Sharma and A. Aiken, "From invariant checking to invariant inference using randomized search," *Formal Methods in System Design*, vol. 48, no. 3, pp. 235–256, 2016.

[48] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from big code," in *ACM SIGPLAN Notices.* ACM, 2015, pp. 111–124.

[49] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, "Search-based program synthesis," *Commun. ACM*, vol. 61, no. 12, pp. 84–93, Nov. 2018.

[50] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *2013 Formal Methods in Computer-Aided Design.* IEEE, 2013, pp. 1–8.

[51] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, "Results and analysis of sygus-comp'15," *arXiv preprint arXiv:1602.01170*, 2016.

[52] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[53] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," in *Readings in artificial intelligence and software engineering.* Elsevier, 1986, pp. 3–34.

[54] Z. Manna and R. J. Waldinger, "Towards automatic program synthesis," in *Symposium on Semantics of algorithmic Languages.* Springer, 1971, pp. 270–310.

[55] M. Parthasarathy, U. Mathur, S. Saha, and M. Viswanathan, "A decidable fragment of second order logic with applications to synthesis," in *27th Annual EACSL Conference Computer Science Logic, CSL 2018.* Schloss Dagstuhl-Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing, 2018, p. 31.

[56] E. M. Gold, "Language identification in the limit," *Information and control*, vol. 10, no. 5, pp. 447–474, 1967.

[57] P. D. Summers, "A methodology for lisp program construction from examples," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 161–175, 1977.

[58] D. Shaw, W. Wartout, and C. Green, "Inferring lisp programs from examples." in *IJCAI*, vol. 75, 1975, pp. 260–267.

[59] T. M. Mitchell, *Machine learning*, ser. McGraw Hill series in computer science. McGraw-Hill, 1997.

[60] E. Kitzelmann, "Inductive programming: A survey of program synthesis techniques," in *International workshop on approaches and applications of inductive programming.* Springer, 2009, pp. 50–73.

[61] D. Angluin and C. H. Smith, "Inductive inference: Theory and methods," *ACM Computing Surveys (CSUR)*, vol. 15, no. 3, pp. 237–269, 1983.

[62] S. Gulwani, "Dimensions in program synthesis," in *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '10. ACM, 2010, pp. 13–24.

[63] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1.* ACM, 2010, pp. 215–224.

[64] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama, "Sygus-comp 2016: results and analysis," *arXiv preprint arXiv:1611.07627*, 2016.

[65] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2017, pp. 319–336.

[66] T. Akiba, K. Imajo, H. Iwami, Y. Iwata, T. Kataoka, N. Takahashi, M. Moskal, and N. Swamy, "Calibrating research in program synthesis using 72,000 hours of programmer time," *MSR, Redmond, WA, USA, Tech. Rep*, 2013.

[67] S. Krishna, C. Puhrsch, and T. Wies, "Learning invariants using decision trees," *arXiv preprint arXiv:1501.04725*, 2015.

[68] E. Pek, X. Qiu, and P. Madhusudan, "Natural proofs for data structure manipulation in c using separation logic," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 440–451.

[69] X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan, "Natural proofs for structure, data, and separation," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 231–242.

[70] C. Flanagan and K. R. M. Leino, “Houdini, an annotation assistant for esc/java,” in *International Symposium of Formal Methods Europe.* Springer, 2001, pp. 500–517.

[71] 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 *International Symposium on Formal Methods for Components and Objects.* Springer, 2005, pp. 364–387.

[72] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, “Gpuverify: a verifier for gpu kernels,” in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 113–132.

[73] N. Chong, A. F. Donaldson, P. H. Kelly, J. Ketema, and S. Qadeer, “Barrier invariants: a shared state abstraction for the analysis of data-dependent gpu kernels,” in *ACM SIGPLAN Notices*, vol. 48, no. 10. ACM, 2013, pp. 605–622.

[74] D. Neider, P. Garg, P. Madhusudan, S. Saha, and D. Park, “Invariant synthesis for incomplete verification engines,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2018, pp. 232–250.

[75] J. R. Quinlan, “Induction of decision trees,” *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[76] N. Tillmann and J. De Halleux, “Pex–white box test generation for. net,” in *International conference on tests and proofs.* Springer, 2008, pp. 134–153.

[77] C. Smith, G. Ferns, and A. Albarghouthi, “Discovering relational specifications,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* ACM, 2017, pp. 616–626.

[78] M. J. Kearns, U. V. Vazirani, and U. Vazirani, *An introduction to computational learning theory.* MIT press, 1994.

[79] J. R. Quinlan, *C4. 5: programs for machine learning.* Elsevier, 2014.

[80] R. Jin and Z. Ghahramani, “Learning with multiple labels,” in *Advances in neural information processing systems*, 2003, pp. 921–928.

[81] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning.* Springer series in statistics New York, 2001, vol. 1, no. 10.

[82] G. Tsoumakas and I. Katakis, “Multi-label classification: An overview,” *International Journal of Data Warehousing and Mining (IJDWM)*, vol. 3, no. 3, pp. 1–13, 2007.

[83] A. Clare and R. D. King, “Knowledge discovery in multi-label phenotype data,” in *European Conference on Principles of Data Mining and Knowledge Discovery.* Springer, 2001, pp. 42–53.

[84] V. Chvatal, “A greedy heuristic for the set-covering problem,” *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.

[85] S. Saha, P. Garg, and P. Madhusudan, "Alchemist: Learning guarded affine functions," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 440–446.

[86] E. W. Weisstein, "Coplanar," 2002.

[87] M. Raghothaman and A. Udupa, "Language to specify syntax-guided synthesis problems," *arXiv preprint arXiv:1405.5590*, 2014.

[88] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[89] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett, "Counterexample-guided quantifier instantiation for synthesis in smt," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 198–216.

[90] C. Barrett, A. Stump, and C. Tinelli, "The smt-lib standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[91] A. Solar-Lezama, "Program sketching," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 5, pp. 475–495, Oct 2013.

[92] R. Alur and N. Singhania, "Precise piecewise affine models from input-output data," in *Proceedings of the 14th International Conference on Embedded Software*. ACM, 2014, p. 3.

[93] A. Bemporad, A. Garulli, S. Paoletti, and A. Vicino, "A bounded-error approach to piecewise affine system identification," *IEEE Transactions on Automatic Control*, vol. 50, no. 10, pp. 1567–1580, 2005.

[94] G. Ferrari-Trecate, M. Muselli, D. Liberati, and M. Morari, "A clustering technique for the identification of piecewise affine systems," *Automatica*, vol. 39, no. 2, pp. 205–217, 2003.

[95] R. Vidal, S. Soatto, Y. Ma, and S. Sastry, "An algebraic geometric approach to the identification of a class of linear hybrid systems," in *42nd IEEE International Conference on Decision and Control (IEEE Cat. No. 03CH37475)*, vol. 1. IEEE, 2003, pp. 167–172.

[96] S. Paoletti, A. L. Juloski, G. Ferrari-Trecate, and R. Vidal, "Identification of hybrid systems a tutorial," *European journal of control*, vol. 13, no. 2-3, pp. 242–260, 2007.

[97] C. Löding, P. Madhusudan, and D. Neider, "Abstract learning frameworks for synthesis," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2016, pp. 167–185.

[98] R. W. Floyd, “Assigning meanings to programs,” in *Program Verification.* Springer, 1993, pp. 65–81.

[99] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[100] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, “Vcc: A practical system for verifying concurrent c,” in *International Conference on Theorem Proving in Higher Order Logics.* Springer, 2009, pp. 23–42.

[101] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *International Conference on Logic for Programming Artificial Intelligence and Reasoning.* Springer, 2010, pp. 348–370.

[102] L. De Moura and N. Bjørner, “Efficient e-matching for smt solvers,” in *International Conference on Automated Deduction.* Springer, 2007, pp. 183–198.

[103] D. Detlefs, G. Nelson, and J. B. Saxe, “Simplify: a theorem prover for program checking,” *Journal of the ACM (JACM)*, vol. 52, no. 3, pp. 365–473, 2005.

[104] Y. Ge and L. De Moura, “Complete instantiation for quantified formulas in satisfiabiliby modulo theories,” in *International Conference on Computer Aided Verification.* Springer, 2009, pp. 306–320.

[105] C. Löding, P. Madhusudan, and L. Pe
textasciitilde na, “Foundations for natural proofs and quantifier instantiation,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 10, 2017.

[106] D.-H. Chu, J. Jaffar, and M.-T. Trinh, “Automatic induction proofs of data-structures in imperative programs,” in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 457–466.

[107] K. L. McMillan, “Interpolation and sat-based model checking,” in *International Conference on Computer Aided Verification.* Springer, 2003, pp. 1–13.

[108] A. R. Bradley, “Sat-based model checking without unrolling,” in *International Workshop on Verification, Model Checking, and Abstract Interpretation.* Springer, 2011, pp. 70–87.

[109] N. Een, A. Mishchenko, and R. Brayton, “Efficient implementation of property directed reachability,” in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design.* FMCAD Inc, 2011, pp. 125–134.

[110] V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen et al., “The 1st verified software competition: Experience report,” in *International Symposium on Formal Methods.* Springer, 2011, pp. 154–168.

[111] D. Neider, P. Garg, P. Madhusudan, S. Saha, and D. Park, "Prototype and benchmarks for "invariant synthesis for incomplete verification engines"," 2 2018. [Online]. Available: https://doi.org/10.6084/m9.figshare.5928094

[112] R. Piskac, T. Wies, and D. Zufferey, "Automating separation logic using smt," in *International Conference on Computer Aided Verification.* Springer, 2013, pp. 773–789.

[113] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv, "Effectively-propositional reasoning about reachability in linked data structures," in *International Conference on Computer Aided Verification.* Springer, 2013, pp. 756–772.

[114] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan, "Verifying security invariants in expressos," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1. ACM, 2013, pp. 293–304.

[115] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "Ironfleet: proving practical distributed systems correct," in *Proceedings of the 25th Symposium on Operating Systems Principles.* ACM, 2015, pp. 1–17.

[116] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures.* ACM, 2009, pp. 233–244.

[117] A. R. Bradley, Z. Manna, and H. B. Sipma, "What's decidable about arrays?" in *International Workshop on Verification, Model Checking, and Abstract Interpretation.* Springer, 2006, pp. 427–442.

[118] 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.* ACM, 1977, pp. 238–252.

[119] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of c programs," in *ACM SIGPLAN Notices*, vol. 36, no. 5. Citeseer, 2001, pp. 203–213.

[120] I. Dillig, T. Dillig, B. Li, and K. McMillan, "Inductive invariant generation via abductive inference," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 443–456.

[121] S. Gulwani, S. Srivastava, and R. Venkatesan, "Program analysis as constraint solving," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 281–292, 2008.

[122] A. Gupta and A. Rybalchenko, "Invgen: An efficient invariant generator," in *International Conference on Computer Aided Verification.* Springer, 2009, pp. 634–640.

[123] M. A. Colón, S. Sankaranarayanan, and H. B. Sipma, "Linear invariant generation using non-linear constraint solving," in *International Conference on Computer Aided Verification*. Springer, 2003, pp. 420–432.

[124] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *Proceedings of the 22nd international conference on Software engineering*. ACM, 2000, pp. 449–458.

[125] A. Champion, T. Chiba, N. Kobayashi, and R. Sato, "Ice-based refinement type discovery for higher-order functional programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018, pp. 365–384.

[126] P. Ezudheen, D. Neider, D. D'Souza, P. Garg, and P. Madhusudan, "Horn-ice learning for synthesizing invariants and contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 131, 2018.

[127] R. Sharma, A. V. Nori, and A. Aiken, "Interpolants as classifiers," in *International Conference on Computer Aided Verification*. Springer, 2012, pp. 71–87.

[128] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A data driven approach for algebraic loop invariants," in *European Symposium on Programming*. Springer, 2013, pp. 574–592.

[129] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori, "Verification as learning geometric concepts," in *International Static Analysis Symposium*. Springer, 2013, pp. 388–411.

[130] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Learning universally quantified invariants of linear data structures," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 813–829.

[131] H. Zhu, A. V. Nori, and S. Jagannathan, "Learning refinement types," in *ACM SIGPLAN Notices*, vol. 50, no. 9. ACM, 2015, pp. 400–411.

[132] Z. Pavlinovic, A. Lal, and R. Sharma, "Inferring annotations for device drivers from verification histories," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 450–460.

[133] S. Padhi, R. Sharma, and T. Millstein, "Data-driven precondition inference with learned features," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 42–56, 2016.

[134] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin, "Automated verification of shape, size and bag properties via user-defined predicates in separation logic," *Science of Computer Programming*, vol. 77, no. 9, pp. 1006–1036, 2012.

[135] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 24, no. 3, pp. 217–298, 2002.

[136] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," in *ACM SIGPLAN Notices*, vol. 44, no. 1. ACM, 2009, pp. 289–300.

[137] Q. L. Le, C. Gherghina, S. Qin, and W.-N. Chin, "Shape analysis via second-order bi-abduction," in *International Conference on Computer Aided Verification.* Springer, 2014, pp. 52–68.

[138] A. Albargouthi, J. Berdine, B. Cook, and Z. Kincaid, "Spatial interpolants," in *European Symposium on Programming Languages and Systems.* Springer, 2015, pp. 634–660.

[139] S. Itzhaky, N. Bjørner, T. Reps, M. Sagiv, and A. Thakur, "Property-directed shape analysis," in *International Conference on Computer Aided Verification.* Springer, 2014, pp. 35–51.

[140] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham, "Property-directed inference of universal invariants or proving their absence," *Journal of the ACM (JACM)*, vol. 64, no. 1, p. 7, 2017.

[141] A. Lal and S. Qadeer, "Powering the static driver verifier using corral," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2014, pp. 202–212.

[142] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "Slam and static driver verifier: Technology transfer of formal methods inside microsoft," in *International Conference on Integrated Formal Methods.* Springer, 2004, pp. 1–20.

[143] A. Lal, S. Qadeer, and S. K. Lahiri, "A solver for reachability modulo theories," in *International Conference on Computer Aided Verification.* Springer, 2012, pp. 427–443.

[144] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *ACM SIGPLAN Notices*, vol. 47, no. 6. ACM, 2012, pp. 405–416.

[145] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations.* Springer, 1972, pp. 85–103.

[146] A. Thakur, A. Lal, J. Lim, and T. Reps, "Posthat and all that: Automating abstract interpretation," *Electronic Notes in Theoretical Computer Science*, vol. 311, pp. 15–32, 2015.

[147] G. Fedyukovich, S. J. Kaufman, and R. Bodík, "Sampling invariants from frequency distributions," in *2017 Formal Methods in Computer Aided Design (FMCAD).* IEEE, 2017, pp. 100–107.

[148] H. Zhu, S. Magill, and S. Jagannathan, "A data-driven chc solver," in *ACM SIGPLAN Notices*, vol. 53, no. 4. ACM, 2018, pp. 707–721.

[149] Y. Vizel, A. Gurfinkel, S. Shoham, and S. Malik, "Ic3-flipping the e in ice," in *International Conference on Verification, Model Checking, and Abstract Interpretation.* Springer, 2017, pp. 521–538.

[150] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with slam," *Communications of the ACM*, vol. 54, no. 7, pp. 68–76, 2011.

[151] A. V. Nori, S. K. Rajamani, S. Tetali, and A. V. Thakur, "The yogi project: Software property checking via static analysis and testing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2009, pp. 178–181.

[152] N. Littlestone, "Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm," *Machine learning*, vol. 2, no. 4, pp. 285–318, 1988.

[153] P. M. Rondon, M. Kawaguci, and R. Jhala, "Liquid types," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 159–169.

[154] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 213–223.

[155] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 263–272.

[156] N. Tillmann and W. Schulte, "Parameterized unit tests," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 253–262.

[157] D. Neider, S. Saha, and P. Madhusudan, "Synthesizing piece-wise functions by learning classifiers," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2016, pp. 186–203.

[158] A. Reynolds, V. Kuncak, C. Tinelli, C. Barrett, and M. Deters, "Refutation-based synthesis in smt," *Formal Methods in System Design*, pp. 1–30, 2017.

[159] A. Astorga, S. Srisakaokul, X. Xiao, and T. Xie, "Preinfer: Automatic inference of preconditions via symbolic analysis," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, 2018, pp. 678–689.

[160] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer, "What good are strong specifications?" in *Proceedings of the 2013 international conference on Software Engineering.* IEEE Press, 2013, pp. 262–271.

[161] X. Xiao, S. Li, T. Xie, and N. Tillmann, "Characteristic studies of loop problems for structural test generation via symbolic execution," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2013, pp. 246–256.

[162] S. Thummalapenta, T. Xie, N. Tillmann, J. De Halleux, and Z. Su, “Synthesizing method sequences for high-coverage testing,” in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 189–206.

[163] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo, “Automatic inference of necessary preconditions,” in *International Workshop on Verification, Model Checking, and Abstract Interpretation.* Springer, 2013, pp. 128–148.

[164] C. Pacheco and M. D. Ernst, “Randoop: feedback-directed random testing for java,” in *OOPSLA Companion*, 2007, pp. 815–816.

[165] T. Gehr, D. Dimitrov, and M. Vechev, “Learning commutativity specifications,” in *International Conference on Computer Aided Verification.* Springer, 2015, pp. 307–323.

[166] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[167] S. Sankaranarayanan, S. Chaudhuri, F. Ivančić, and A. Gupta, “Dynamic inference of likely data preconditions over predicates by tree learning,” in *Proceedings of the 2008 international symposium on Software testing and analysis.* ACM, 2008, pp. 295–306.

[168] A. Albarghouthi, I. Dillig, and A. Gurfinkel, “Maximal specification synthesis,” in *ACM SIGPLAN Notices*, vol. 51, no. 1. ACM, 2016, pp. 789–801.