Learning Stateful Preconditions Modulo a Test Generator

Angello Astorga
University of Illinois at Urbana-Champaign
USA
aastorg2@illinois.edu

P. Madhusudan
University of Illinois at Urbana-Champaign
USA
madhu@illinois.edu

Shambwaditya Saha
University of Illinois at Urbana-Champaign
USA
ssaha6@illinois.edu

Shiyu Wang
University of Illinois at Urbana-Champaign
USA
shiyuw3@illinois.edu

Tao Xie
University of Illinois at Urbana-Champaign
USA
taoxie@illinois.edu

Abstract

In this paper, we present a novel learning framework for inferring stateful preconditions (i.e., preconditions constraining not only primitive-type inputs but also non-primitive-type object states) modulo a test generator, where the quality of the preconditions is based on their safety and maximality with respect to the test generator. We instantiate the learning framework with a specific learner and test generator to realize a precondition synthesis tool for C#. We use an extensive evaluation to show that the tool is highly effective in synthesizing preconditions for avoiding exceptions as well as synthesizing conditions under which methods commute.

CCS Concepts • **Theory of computation** → **Program specifications**; • **Software and its engineering** → *Dynamic analysis*; • **Computing methodologies** → Classification and regression trees.

Keywords Specification Mining, Data-Driven Inference, Synthesis

ACM Reference Format:
Angello Astorga, P. Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. 2019. Learning Stateful Preconditions Modulo a Test Generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3314221.3314641

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314641

1 Introduction

Reliable and robust software needs to function well even when given illegal inputs. One common way to handle illegal inputs is to equip the software with a precondition: any inputs violating the precondition are classified as illegal inputs, and further executions on these inputs beyond the precondition-checking point are prevented. To address error-proneness and tediousness of manually deriving and specifying preconditions, researchers have proposed various existing automatic approaches of precondition inference based on static analysis (e.g., [4, 7–9, 12, 23, 24, 26, 34, 36]) or dynamic analysis (e.g., [5, 6, 10, 11, 13, 15, 29, 33]).

Given that static analysis is conservative in nature and often results in many false positives, existing approaches based on dynamic analysis have their advantages, being broadly classified into two major categories: white-box ones [6, 10, 11, 13][29, VPregen] and black-box ones [15, 29, 33][29, PIE]. Both categories learn preconditions from runtime information collected from software execution. For white-box approaches, runtime information typically includes program states in between statements inside the software, whereas for black-box approaches, runtime information typically includes inputs and outputs of the invoked methods defined on the interface of a class.

However, existing approaches of precondition generation based on dynamic analysis typically do not tackle two major challenges. First, most of these approaches do not give any guarantee on the quality of the synthesized preconditions. If preconditions are learned *passively* using feature vectors of states observed on some fixed set of test inputs, the learning is intrinsically incomplete and can lead to overfitting the given test inputs, producing preconditions that are not guaranteed to generalize to unseen test inputs. Certain recent white-box approaches (e.g., [29, VPreGen]) can prove that preconditions are safe with the help of a static verifier, but the required verification is a hard problem to automate, requiring synthesis of inductive loop invariants, etc. In this

paper, we explore a different guarantee—to ensure that the synthesized precondition is safe (and maximal) with respect to a *test generator*, which is typically a lot more scalable than a static verifier.

Second, the target preconditions are *stateful* by nature for object-oriented programs: the target preconditions constrain not only primitive-type inputs (such as integers and strings) but also non-primitive-type inputs, such as the receiver-object states and object states of a non-primitive-type argument for the method under consideration.

To address these two challenges, in this paper, we present a novel active learning framework for *testing-assisted inference of stateful preconditions* that are guaranteed to be safe and maximal with respect to a given test generator. Safety and maximality are both parameterized with respect to a test generator. We want a precondition that is safe—the test generator cannot find a precondition-allowing (i.e., precondition-satisfying) input whose execution leads to a failure, and a precondition that is maximal—the test generator cannot find an input disallowed by the precondition whose execution does not lead to any failure. We define this requirement through a formal notion of *ideal preconditions* with respect to a test generator.

To synthesize stateful preconditions, our framework includes an abstraction based on observer methods defined for various classes, namely an observer abstraction. This abstraction enables the learned precondition to express abstract properties of non-primitive-type inputs 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).

Our *active learning framework* combines a black-box learner with a white-box teacher, with the latter realized using a test generator, in order to learn ideal preconditions. Working by actively querying a test generator to produce an ideal precondition alleviates the problem of learning a precondition that overfits a particular set of inputs. Learning ideal preconditions in a logic $\mathcal{L}$ with the aid of a test generator can then be posed as actively learning formulas using positive and negative feature vectors that the test generator produces in rounds of interaction between the test generator and the learner.

However, there are two main issues that need to be addressed when realizing this active learning, mostly due to the fact that inherently the test generator cannot guarantee feature vectors to be positive (but can certify negative feature vectors). First, the test generator can label certain vectors as positive, and later change its mind and label these vectors negative. Second, the limited expressiveness of the logic to state preconditions can also force the exclusion of certain positive inputs from the learned precondition.

To address these issues, our framework includes a component named a *conflict resolver* that effectively relabels positive feature vectors to negative vectors when necessary. The resulting learning framework with the conflict resolver can be instantiated for any logic for expressing preconditions using any learner and any test generator in order to learn ideal preconditions.

We also prove a convergence result—assuming that the logic expresses finitely many formulas closed under Boolean operations, an ideal precondition expressible in the logic exists, and the learner is able to always produce formulas consistent with samples when they exist, we are guaranteed to converge to an ideal precondition.

We instantiate the learning framework with a learner that uses an algorithm for decision-tree learning to synthesize formulas in a particular logic that involves Boolean combinations of predicates and inequalities involving numerical predicates, where the predicates describe both properties of primitive-type inputs and non-primitive-type inputs (such as the receiver objects and non-primitive-type input parameters). This algorithm is a standard machine-learning algorithm for decision-tree learning that uses statistical measures to build trees of small size; these trees correspond to preconditions consistent with the (conflict resolved) counterexamples returned by the test generator in each round, and learning continues till the learner finds an ideal precondition.

We also instantiate the framework for two important tasks in specification inference: runtime-failure prevention and conditional-commutativity inference [37]. 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. Inferring preconditions for commutativity is important for programmers to understand when they can reorder calls to methods while preserving behavior equivalence, and also has applications to downstream tools such as program analysis and tools for instrumenting concurrency control [20–22, 41].

We implement a prototype of our framework in a tool called Proviso using a learner based on the ID3 classification algorithm [31], a powerful classification algorithm in the machine learning community, and Pex [39], an industrial test generator based on dynamic symbolic execution [19, 35], shipped as IntelliTest in the Microsoft Visual Studio Enterprise Edition since Visual Studio 2015/2017/2019.

This paper makes the following main contributions:

- A novel formalization for the inference problem of stateful precondition modulo a test generator, called ideal preconditions, that guarantees that it is safe and maximal with respect to the test generator.
- A novel active learning framework for inferring ideal stateful preconditions, using a conflict resolver component to adaptively mark positive inputs as negative, when necessary, in order to deal with the incompleteness of the test generator and the inexpressiveness of the logic for expressing preconditions.

```
[PexMethod]
public void PUT-CommutativityAddContains(
[PexAssumeUnderTest] ArrayList s1, int x, int y){
DataStructures.ArrayList s2 = new
    DataStructures.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 1. Encoding conditional property: Commutativity conditions for methods `Contains` and `Add` from the ArrayList class in the .NET Library.

- Convergence arguments that the learning framework will eventually synthesize a safe and maximal precondition modulo the test generation, if there is one, when the hypothesis space is finite.
- Instantiations of the framework in a tool Proviso for two important tasks in specification inference (preconditions for preventing runtime failures and conditions for commutativity of methods) and using a machine-learning algorithm for decision trees and an industrial test generator (Pex).
- An extensive evaluation on various C# classes from well-known benchmarks and open source projects that demonstrates the effectiveness of the proposed framework.

2 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 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 [40] in Figure 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 4.1 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 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 paper.

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 $(x = y \wedge s1.Contains(x)) \ \vee \ (x \neq y)$.

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).

3 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 3.2), 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 4 and Figure 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 4.3).

We next formalize the notions of programs, valid and invalid input states, and testing-based teachers (Section 3.1), and then formalize the problem of precondition synthesis modulo a testing-based teacher using the notion of an ideal precondition (Section 3.2).

3.1 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).

Logic for expressing preconditions. The *logic* $\mathcal{L}$, for expressing preconditions for a method $m(\vec{p})$ in this paper 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 paper, 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

[1] Note that in this paper, 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.

testing tools such as Pex are effective, but not testing tools such as Randoop that generate random inputs). Formally,

Definition 3.1 (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.

3.2 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 3.2. 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: 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.

4 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 4.1) and then discuss multiple ways to instantiate the framework (Section 4.2). Adapting a TBT to realize this framework is discussed in Section 5. Finally, in Section 4.3, we discuss general conditions under which we can show that our learners and learning framework converge to an ideal precondition with respect to any TBT.

4.1 Framework Overview

Our learning framework, depicted in Figure 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.

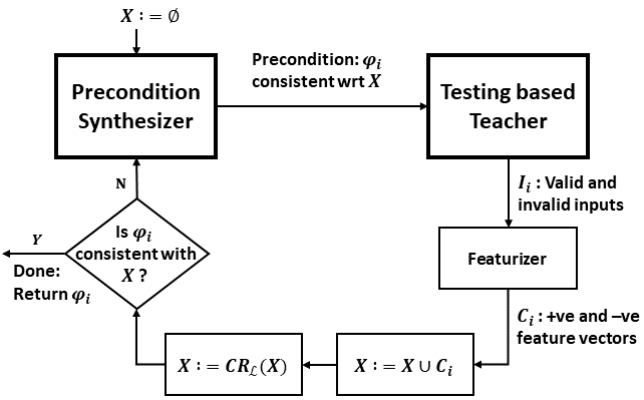

Figure 2. The learning framework for synthesizing ideal preconditions with respect to a TBT.

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. 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

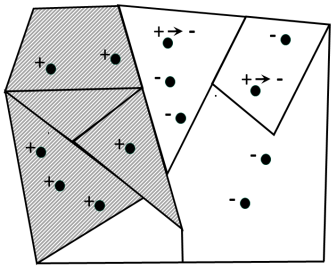

Figure 3. An example of conflict resolution where positive vectors are made negative. Partitions denote equivalence classes of indistinguishable vectors; points denote positive and negative feature vectors. The shaded region denotes a consistent precondition.

properties of ideal preconditions in Definition 3.2: 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 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.

4.2 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

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

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 4.3 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 [25, 31] 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 [17]. In our evaluation (Section 6), 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 [2]). 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 [3, 27, 32]. In fact, one recent tool named PIE [29] 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.

4.3 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 4.1. *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 gist: 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. □

5 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.

5.1 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 3.1). 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 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.

Table 1. Statistics of evaluation subjects

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

6 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 [39], to a testing-based teacher, choose the logic $\mathcal{L}_{\mathbb{B},\mathbb{Z}}$, over Booleans and integers (introduced in Section 4.2), and a variant of Quinlan's C 5.0 decision tree algorithm [25, 31] to an exact learner [17]. The conflict resolver is the one for $\mathcal{L}_{\mathbb{B},\mathbb{Z}}$ described in Section 4.2. 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:

RQ: 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 4, 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 [6, 29, 30] and test generation [38, 42]. 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, Lidgren Network and Data Structures and Algorithms (DSA), and (2) a set of Code Contract benchmarks from the cccheck static analyzer [12] and benchmarks from the Hola engine [14]. 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 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 in our evaluation, we use all and only the public observer methods in the interface of their respective class.

Table 2 summarizes our evaluation results, including statistics on our subjects, statistics on learning, and details on validation with respect to Randoop [28] (a test generator) and ground truth.

6.1 RQ: 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 [28], 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.

Our results shown in Table 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. A sample of learned preconditions can be found on the Proviso website[2]. 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.

6.2 On Empirical Comparison with Related Work

It is hard to provide a fair comparison with two closely related approaches by Padhi et al. [29] and Gehr et al. [18]. 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. [29] 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. [29, 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.

[2] http://madhu.cs.illinois.edu/proviso

7 Related Work

Black-box approaches. Ernst et al. [15] 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. [29], Gehr et al. [18], and Sankaranarayanan et al. [33]. The last two approaches [18, 33] 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. [29] 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) [2], which in fact resembles online learning. In this setting, the target expression is

Table 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.

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

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 [1]).

Decision-tree learning. Decision-tree learning has been used in several contexts in program synthesis before — in precondition synthesis [33], in invariant synthesis [16, 17], in synthesizing piece-wise linear functions [27], 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.

8 Conclusion

In this paper, 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 specification inference and evaluated our framework on various C# classes from well-known benchmarks and open source projects. Our evaluation results demonstrate the effectiveness of the proposed framework.

Acknowledgment

This work was supported in part by National Science Foundation under grant no. CCF-1527395, CNS-1513939, CNS-1564274, CCF-1816615 and the GEM fellowship.

References

[1] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. 2016. Maximal specification synthesis. In *POPL 2016*.

[2] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-guided synthesis. In *Dependable Software Systems Engineering*. NATO Science for Peace and Security Series, D: Information and Communication Security, Vol. 40.

[3] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling enumerative program synthesis via divide and conquer. In *TACAS 2017*.

[4] Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. 2005. Synthesis of interface specifications for Java classes. In *POPL 2005*.

[5] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *POPL 2002*.

[6] Angello Astorga, Siwakorn Srisakaokul, Xusheng Xiao, and Tao Xie. 2018. PreInfer: Automatic inference of preconditions via symbolic analysis. In *DSN 2018*.

[7] David Brumley, Hao Wang, Somesh Jha, and Dawn Xiaodong Song. 2007. Creating vulnerability signatures using weakest preconditions. In *CSF 2007*.

[8] Raymond P. L. Buse and Westley Weimer. 2008. Automatic documentation inference for exceptions. In *ISSTA 2008*.

[9] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *PLDI 2009*.

[10] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. 2007. Bouncer: Securing software by blocking bad input. In *SOSP 2007*.

[11] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. 2005. Vigilante: End-to-end containment of Internet worms. In *SOSP 2005*.

[12] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic inference of necessary preconditions. In *VMCAI 2013*.

[13] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic symbolic execution for invariant inference. In *ICSE 2008*.

[14] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive invariant generation via abductive inference. In *OOPSLA 2013*.

[15] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically discovering likely program invariants to support program evolution. In *ICSE 1999*.

[16] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. 2018. Horn-ICE learning for synthesizing invariants and contracts. In *OOPSLA 2018*.

[17] Pranav Garg, P. Madhusudan, Daniel Neider, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *POPL 2016*.

[18] Timon Gehr, Dimitar Dimitrov, and Martin T. Vechev. 2015. Learning commutativity specifications. In *CAV 2015*.

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *PLDI 2005*.

[20] Maurice Herlihy and Eric Koskinen. 2008. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP, 2008*.

[21] Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. 2011. Exploiting the commutativity lattice. In *PLDI 2011*.

[22] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic parallelism requires abstractions. In *PLDI 2007*.

[23] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. 2014. Sound input filter generation for integer overflow errors. In *POPL 2014*.

[24] Ravichandhran Madhavan and Raghavan Komondoor. 2011. Null dereference verification via overapproximated weakest precondition analysis. In *OOPSLA 2011*.

[25] Thomas M. Mitchell. 1997. *Machine Learning* (1 ed.).

[26] Mangala Gowri Nanda and Saurabh Sinha. 2009. Accurate interprocedural null-dereference analysis for Java. In *ICSE 2009*.

[27] Daniel Neider, Shambwaditya Saha, and P. Madhusudan. 2016. Synthesizing piece-wise functions by learning classifiers. In *TACAS 2016*.

[28] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed random testing for Java. In *OOPSLA 2007*.

[29] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-driven precondition inference with learned features. In *PLDI 2016*.

[30] Nadia Polikarpova, Carlo A. Furia, Yu Pei, Yi Wei, and Bertrand Meyer. 2013. What good are strong specifications?. In *ICSE 2013*.

[31] J. R. Quinlan. 1986. Induction of Decision Trees. *Mach. Learn.* 1, 1 (1986).

[32] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2017. Refutation-based synthesis in SMT. *Formal Methods in System Design* (2017).

[33] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. 2008. Dynamic inference of likely data preconditions over predicates by tree learning. In *ISSTA 2008*.

[34] Mohamed Nassim Seghir and Daniel Kroening. 2013. Counterexample-guided precondition inference. In *ESOP 2013*.

[35] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. In *ESEC/FSE 2005*.

[36] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. 2009. Fault localization and repair for Java runtime exceptions. In *ISSTA 2009*.

[37] Calvin Smith, Gabriel Ferns, and Aws Albarghouthi. 2017. Discovering relational specifications. In *ESEC/FSE 2017*.

[38] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. 2011. Synthesizing method sequences for high-coverage testing. In *OOPSLA 2011*.

[39] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: White box test generation for .NET. In *TAP 2008*.

[40] Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. In *ESEC/FSE 2005*.

[41] W. E. Weihl. 1988. Commutativity-Based concurrency control for abstract data types. *IEEE Trans. Comput.* 37, 12 (1988).

[42] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. 2013. Characteristic studies of loop problems for structural test generation via symbolic execution. In *ASE 2013*.