Synthesizing Contracts Correct Modulo a Test Generator

ANGELLO ASTORGA, University of Illinois at Urbana-Champaign, U.S.A
SHAMBWADITYA SAHA, Tufts University, U.S.A
AHMAD DINKINS, University of Illinois at Urbana-Champaign, U.S.A
FELICIA WANG, University of Illinois at Urbana-Champaign, U.S.A
P. MADHUSUDAN, University of Illinois at Urbana-Champaign, U.S.A
TAO XIE, Peking University, China

We present an approach to learn contracts for object-oriented programs where guarantees of correctness of the contracts are made with respect to a test generator. Our contract synthesis approach is based on a novel notion of tight contracts and an online learning algorithm that works in tandem with a test generator to synthesize tight contracts. We implement our approach in a tool called Precis and evaluate it on a suite of programs written in C#, studying the safety and strength of the synthesized contracts, and compare them to those synthesized by Daikon.

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

Additional Key Words and Phrases: Specification Mining, Data-Driven Inference, Synthesis, Software Testing, One-Class Classification

ACM Reference Format:
Angello Astorga, Shambwaditya Saha, Ahmad Dinkins, Felicia Wang, P. Madhusudan, and Tao Xie. 2021. Synthesizing Contracts Correct Modulo a Test Generator. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 104 (October 2021), 27 pages. https://doi.org/10.1145/3485481

1 INTRODUCTION

Research in academia and industry has argued for annotating code using formal contracts to engineer and maintain reliable software [Barnett et al. 2004; Fähndrich 2010; Leavens et al. 2006; Meyer 1988; Spivey 1988]. Annotations for a method in a class typically consist of preconditions and summaries (also named postconditions). Preconditions are assumptions that must be satisfied by the input parameters and the receiver object before a client invokes the method on it. Summaries capture the effect of executing the method both in terms of the return value and the receiver object. In other words, formal contract specifications describe the assumptions that classes/methods make on their clients as well as the commitments that they promise to their clients in terms of their functional behaviors.

Annotating software components with contracts has many potential benefits. Contracts can be used for understanding the intended usage of software libraries and for documenting legacy

Authors' addresses: Angello Astorga, aastorg2@illinois.edu, University of Illinois at Urbana-Champaign, U.S.A; Shambwaditya Saha, shambwaditya.saha@tufts.edu, Tufts University, U.S.A; Ahmad Dinkins, ahmadid2@illinois.edu, University of Illinois at Urbana-Champaign, U.S.A; Felicia Wang, felicia5@illinois.edu, University of Illinois at Urbana-Champaign, U.S.A; P. Madhusudan, madhu@illinois.edu, University of Illinois at Urbana-Champaign, U.S.A; Tao Xie, taoxie@pku.edu.cn, Peking University, China.

2475-1421/2021/10-ART104
https://doi.org/10.1145/3485481

code. Contracts also facilitate modular software development and evolution—components can be developed independently and swapped in as long as they maintain the contracts that they agree upon [Meyer 1988]. Furthermore, contracts documenting how modules behave in a formal language can enable various downstream analyses such as unit testing (to supply strong test oracles for checking the module under test without knowing its clients), runtime monitoring [Chen and Roşu 2007] (to detect runtime behavior deviating from specifications), and formal verification [Floyd 1960; Hoare 1969] (to conduct deductive verification using contracts as formal specifications). Various specification languages for contracts have been developed and used in industry, including Design by Contract (DbC) [Meyer 1988], Java Modeling Language (JML) [Leavens et al. 2006], Spec# [Barnett et al. 2004], and CodeContract for C# and other .NET languages [Fähndrich 2010].

Over the years, various research efforts have focused on *specification mining*, i.e., automatically inferring contracts for code [Alur et al. 2005; Ammons et al. 2002; Ernst et al. 1999; Henzinger et al. 2005], to alleviate the programmer's burden of writing specifications. One basic question that arises when inferring contracts is on the guarantees given on their correctness. At one extreme, we can demand that the contracts be formally proven to be correct. However, this requirement mandates automatic formal verification [Alur et al. 2005; Henzinger et al. 2005], which is a hard problem (undecidable) requiring approaches that do not scale well for large code such as inferring loop invariants and solving sophisticated logical validity of verification conditions. On the other end, various approaches infer contracts from dynamic executions of the target program on a fixed set of tests. A classic approach that falls in this category is Daikon [Ernst 2000; Ernst et al. 1999]. Most of these approaches [Ammons et al. 2002; Astorga et al. 2018; Csallner et al. 2008; Ernst et al. 1999] give no measurement of the accuracy of synthesized contracts, let alone a guarantee of correctness.

In this paper, we follow a third approach that guarantees correctness *with respect to a test generator* (introduced for preconditions in recent work [Astorga et al. 2019]). More precisely, we demand that a deterministic test generator with a fixed set of resources should not be able to find faults with correctness of the synthesized contracts. Contract synthesis algorithms that give such an assurance could work in tandem with a test generator in order to mine contracts. The salient feature of this approach is that it avoids the hardness of automatic program verification while at the same time giving some guarantees of correctness that cannot be provided by approaches based on dynamic analysis on a fixed finite set of tests. In this paper, we undertake the problem of synthesizing contracts whose safety is guaranteed modulo a test generator.

A contract, in this paper, is meant to summarize the input-output behaviors of methods in classes in an object-oriented programming language *given preconditions for these methods*. Preconditions can be given by users, or taken to be the trivial precondition *true*. They can also be synthesized automatically to avoid errors/exceptions. The contract for a method is a summary of its behavior, and captures the *effect* of calling the method. It is expressed as a property relating input parameters (which may include other objects) and the state of the receiver object before the method is called with the return value and the state of the receiver object after the call. Properties about an object are expressed using *observer methods* that are pure methods for revealing properties of the object. The contract summarizes the effect of calling the method on all inputs satisfying the given precondition and on which the method terminates (inputs that cause exceptions or errors are ignored). A contract for a method is said to be *safe* if on any possible input state that satisfies the precondition, the method's behavior, if terminating, satisfies the contract.

The general architecture that we propose for contract synthesis modulo a test generator is to pair a contract synthesizer (aka learner) with the test generator (teacher), similar to the recent work [Astorga et al. 2019]. Inference proceeds in rounds, where in each round the synthesizer proposes contracts that the test generator tries to refute for safety, and where refutations result in concrete

input-output behaviors of the method. The contract synthesizer utilizes these counterexamples of safety to synthesize a safe contract modulo the test generator.

One-class Classification. The primary challenge in instantiating this general architecture is in determining how to ensure that contracts are *strong*.[1] A trivial contract such as *true* is obviously safe for any method, but is the weakest contract and is useless. From a learning perspective, the example behaviors exposed by a test generator can be seen as *positively* labeled samples. Note that a test generator cannot be reasonably expected to give *negatively* labeled samples (examples of behaviors, abstracted by observations, that cannot be manifested by the method). This characteristic makes the learning problem significantly different from existing related work. For instance, work on precondition synthesis modulo a test generator studied in the recent work [Astorga et al. 2019] is a setting where the learner learns from both positive and negative examples (since the test generator can produce inputs that cause exceptions as well as inputs that do not).

The learning problem in contract synthesis is in fact best seen as a *one-class classification (OCC) learning problem* [Moya and Hush 1996], which is the problem of learning from only positive samples in Boolean classification. The most important question that we tackle in this paper is how to learn strong contracts from (positive) samples of observations of behaviors, while at the same time guaranteeing that each round of learning converges to a tight contract.

We address the preceding problem using primarily two technical contributions. First, we define *tight* contracts (more generally, k-tight contracts, for $k \in \mathbb{N}$), a new technical notion that negotiates a balance between simplicity of contracts and their strength, to avoid both extremely weak contracts as well as contracts that overfit. Second, we design a new learning algorithm that synthesizes tight contracts using state-of-the-art expression synthesis techniques that combine constraint solving and syntax-guided synthesis.

Tight Contracts. The contract synthesis algorithm that we develop automatically synthesizes atomic predicates and using quantifier-free logic, captures relationships between integer and Boolean variables that capture input parameters, output values, and properties of object states. For the purpose of describing tight contracts, let us fix a particular set of predicates (these would have been synthesized for the current round of learning). Contracts can then be seen as Boolean combinations of this set of atomic predicates.

We require contracts to be safe with respect to the set of behaviors exposed thus far by the test generator, of course. We also would like contracts to be strong in that they capture small semantic spaces (the semantic space of a contract φ is $[\![\varphi]\!]$, the set of models satisfying φ).

However, given a set of behaviors, it may not be wise to always synthesize the *strongest* Boolean formula that includes the sample behaviors. Learning such strongest formulae (called *overfitting* in learning) is unlikely to result in safe contracts, and hence would increase the number of rounds to converge to a safe contract.

Our proposal in this paper is a novel notion called *tight contracts* that aim to be *strong while utilizing a small number of disjunctions*. We express contracts using *decision trees with conjunctive leaves* and use the number of conditionals in a decision tree to measure how "disjunctive" a concept is. Intuitively, there are two extreme kinds of contracts. We can use *no* disjunctions, and take the strongest contract expressible as a conjunction— but this contract would be a weak one. On the other hand, the strongest contract can be obtained by taking a disjunction of conjuncts, one disjunct for each sample that is the conjunction of all predicates true in that sample. This contract would have, however, a lot of disjuncts. We say that a concept decision tree φ with n conditionals is *k-tight* with respect to a set of samples S *if there is no stronger decision tree that includes the samples S and*

[1] A contract φ is said to be stronger than a contract ψ if $\varphi \Rightarrow \psi$. The notion of a strong contract is not a formal notion.

has at most $n + k$ conditionals. In particular, we are interested in 1-tight contracts (which we also simply call *tight* contracts).

Consider a set of samples S and a contract φ that includes S (each element of S satisfies φ). If there exists a stronger contract than φ (expressible over the current predicates) that includes S and that uses at most one more conditional, then φ is *not* tight. Our learning algorithm will prefer to use an extra conditional to obtain a stronger contract, and hence avoid proposing φ. If more than one more conditional is required to capture stricter concepts than φ that include S, we declare φ tight.

The goal of the learning algorithm is to learn in each round (as well as finally) a tight (or k-tight) contract with respect to the set of behaviors exposed by the test generator. This negotiation between the number of disjunctions in concepts and their strength is designed to avoid extreme overfitting, allowing the number of disjunctions to grow dynamically only when justified to capture stronger contracts.

The proof that learning tight contracts is effective ultimately lies in our experimental evaluation. We show in our evaluation that the learning algorithm for learning tight contracts converges to safe and strong contracts (see Section 6).

Learning Tight Contracts. We develop a novel learning algorithm for synthesizing tight contracts. We consider contracts expressed in quantifier-free Boolean-closed logics over a class of atomic predicates, where atomic predicates can involve arbitrary linear arithmetic constraints.

Our learning algorithm synthesizes tight contracts by a careful dovetailing between two tasks. The first task is to synthesize *tight* decision trees over a fixed set of atomic predicates (that are synthesized by the second task). Finding tight contracts for a set of sample behaviors is nontrivial since the notion of tightness is logically complex (involving universal quantification). We need to find a decision tree T with n conditionals (preferably for a small n) that includes all samples such that *all* decision trees with at most $n + 1$ conditionals and that includes all samples not stronger than T.

We solve the synthesis problem of tight contracts precisely. We propose a novel algorithm that uses repeated calls to SyGuS [Alur et al. 2015] (SYntax-GUided Synthesis) solvers (which tackle SyGuS synthesis problems), and that ensures that there are no $k + 1$ conditional trees capturing a stronger concept. Using recent progress in solving Boolean SyGuS problems [Alur et al. 2015; Reynolds et al. 2019], we develop an effective learning algorithm for tight concepts. Furthermore, by increasing the number of disjunctions iteratively, the learning algorithm tries to minimize the number of disjuncts in the tight decision tree that it synthesizes.

The preceding task interleaves with the second task of expression synthesis to infer atomic predicates that capture functional relationships between output states and input states. We synthesize such expressions on the leaves of tight decision trees constructed as above, growing the set of predicates on which decision trees are constructed. The atomic predicates that we synthesize can be from various grammars and logics; in our implementation, we synthesize expressions over arbitrary linear arithmetic on integer parameters and observer method returns. Expression synthesis is also achieved using SyGuS solvers [Alur et al. 2015; Reynolds et al. 2019].

We prove that the synthesizer, despite the preceding dovetailing between tight decision tree synthesis and predicate synthesis over an infinite grammar, terminates for any set of sample behaviors (i.e., each round of learning terminates). Moreover, when plugged into the architecture, working in tandem with a test generator, if learning terminates, we can make the following guarantees of the produced contract (1) that it is safe with respect to the test generator, and (2) that it is tight (or k-tight) with respect to the set of all behaviors exposed by the test generator in various rounds.

Evaluation. We implement a prototype for our approach in a tool called Precis using an industrial test generator Pex [Tillmann and De Halleux 2008] (aka IntelliTest), available in the Visual Studio 2019 Enterprise Edition. We evaluate our prototype on our benchmarks including a set of C# classes. We study three research questions for evaluating Precis: (a) how safe synthesized contracts are, (b) how strong synthesized contracts are, and (c) ablation studies for studying the effect of predicate synthesis, the ability to express disjunctions, as well as the choice of 1-tightness. Our results show that Precis, which by design always synthesizes tight (i.e., 1-tight) contracts, is effective in synthesizing safe and strong contracts. We also compare contracts synthesized by Precis with contracts synthesized by Daikon paired with a test generator. Our results show that contracts synthesized by Precis are stronger than contracts synthesized by Daikon for about half of our benchmark subjects.

In summary, this paper makes the following main contributions:

- A novel notion of tight (and k-tight) contracts that balance strength and expression simplicity of contract summaries.
- A novel problem formulation of synthesizing contracts in an object-oriented setting modulo a test generator. Given a program, a precondition for the program, and a test generator, the problem is to find a contract summary (for inputs that satisfy the precondition) that is both a tight contract and also is safe with respect to the test generator.
- A novel online decision-tree *one-class classifier* learning algorithm that synthesizes tight decision trees. This algorithm combines predicate synthesis and tight formula synthesis, and has a nontrivial proof of termination.
- An implementation of our learning algorithm in conjunction with the state-of-the-art testing tool Pex [Tillmann and De Halleux 2008] as the test generator, and an evaluation of the strength and safety of synthesized contracts on a set of benchmarks, as well as a comparison with Daikon.

2 AN ILLUSTRATIVE EXAMPLE

In this section, we illustrate our contributions through an example. Consider a dictionary object d and its *Setter* method (shown in Figure 1) that takes as input integer parameters *key* and *value*. Given this key-value pair, the *Setter* method updates the value indexed by *key* if *key* is already present (Lines 10-19). Otherwise, *Setter* computes the bucket and index in the bucket where the pair can be inserted (Lines 21-22) and then adds the pair to the dictionary (Lines 24-28).

Given this *Setter* method, our approach aims to learn a contract that is *strong* in that it tries to precisely describe the set of input-output behaviors of *Setter*. In this example, let us assume that the precondition is the trivial one *true* that allows calling the method in any object state and with any possible input parameters. Viewing the contract summarizing the behavior of the method as a formula satisfied by the combined input and output state, we want this formula to be as strong as possible.

The algorithm fixes a set of Boolean and integer features. The Boolean features include $ContainsKey_{pre}(key)$ and $ContainsKey_{post}(key)$, obtained by calling a Boolean observer method on the input parameter key, that evaluate to whether key is in the dictionary in the pre- and post-states of the object. Similarly, the Boolean features $ContainsVal_{pre}(value)$ and $ContainsVal_{post}(value)$ record the observer method call that evaluates whether there exists a key in the dictionary that maps to $value$ in the pre- and post-states. The integer features include key and $value$, as well as $Count_{pre}$ and $Count_{post}$, which are observer method calls to $Count()$ that returns the number of key-value pairs in dictionary object d.

```
1  private void Setter(TKey key, TValue value, bool add){
2      if (key == null)
3          throw new ArgumentNullException("nameof(key)");
4      if (buckets == null)
5          Initialize(0);
6
7      int hashCode = this.GetHashCode(key);
8      int targetBucket = hashCode % this.buckets.Length;
9
10     for (int i = this.buckets[targetBucket]; i >= 0;
11               i = this.entries[i].next) {
12
13         if (this.entries[i].hashCode == hashCode &&
14             this.entries[i].key == key) {
15
16                 this.entries[i].value = value;
17                 return;
18         }
19     }
20
21     int index;
22     index, targetBucket = this.GetNextIndexBucket(hashCode);
23
24     this.entries[index].hashCode = this.hashCode;
25     this.entries[index].next = this.buckets[targetBucket];
26     this.entries[index].key = key;
27     this.entries[index].value = value;
28     this.buckets[targetBucket] = index;
29 }
```

Fig. 1. Details of the Setter method (simplified for ease of illustration) used in the working example.

The algorithm starts with a set of *base predicates* R_b over integer features; these predicates include simple predicates over integers such as $x \leq y$, $x = y$, $x < y$, where x, y range over the integer features described earlier. The set of predicates grow as we synthesize new predicates to describe the behaviors of the method.

Our synthesis of contracts proceeds in rounds, where in each round we have a test generator, with a fixed set of resources, for exploring a (finite) set of input-output behaviors of the method, given the current conjectured contract for it. We represent an input-output behavior as a feature vector *fv* over integer and Boolean features (which include observer methods applied on all relevant objects, including the pre and post objects of the receiver object).

The goal of the learner in each round is to synthesize a strong formula that includes all the observed input-output behaviors, i.e., all the feature vectors should make the synthesized formula *true*.

One trivial strong contract that can be learned is to simply interpret each feature vector as a formula and take the disjunction of these formulas. However, this contract will have a large number of disjunctions. Such a strong contract will also likely be unsafe, leading to further rounds of learning. In learning, this situation is called *overfitting* as the learned concept does not generalize well.

One way to avoid this kind of overfitting is to restrict to formulas with only conjunctions. We can in fact learn the strongest *conjunctive* concept that includes the set of behaviors. For example, over a particular set of observed behaviors, the learner could learn the following conjunctive contract:

$$(ContainsKey(key)_{post} \wedge ContainsVal(value)_{post} \wedge Count_{pre} \geq 0 \wedge Count_{post} \geq 1 \wedge Count_{post} \geq Count_{pre})$$

This contract says that (1) (*key*, *value*) pair will be in d after invoking *Setter*, (2) that the number of key-value pairs in the pre state is at least zero and in the post state is at least one, and (3) the number of key-value pairs in the post state is greater than or equal to the number of key-value pairs in the pre state. In fact, this contract is the strongest conjunctive concept (w.r.t to our base predicates) and it is also the concept that Daikon [Ernst et al. 1999] infers for this case. The contract is simple and does not have any disjunctions; however, it is not as strong as we would like, because the base predicates are not expressive enough and because the expression does not use disjunctions.

The main dilemma that we face is in negotiating the precision that disjunctions bring and overfitting of the example behaviors that prevents effective learning. Our work proposes and defines a notion of *tight contracts* to address the preceding dilemma. Intuitively, tight contracts allow disjunctions when justified. More precisely, a formula φ that includes the feature vectors is said to be tight if there is *no other* formula with *one* more conditional that captures a stronger approximation of the feature vectors. Intuitively, we accept formulas as tight if we cannot capture a smaller semantic space by adding one more disjunction. If there is a formula that has one more disjunction and is stronger than φ, we would prefer this formula over φ. In this sense, the preceding conjunctive contract (and the one inferred by Daikon) is *not* tight.

Our learning algorithm also has a predicate synthesis engine that synthesizes new functional relationships between input and output variables, addressing the problem of inexpressiveness of the base predicates. On termination, our algorithm synthesizes the tight contract:

$$(ContainsVal(value)_{post} \wedge ContainsKey(key)_{post}) \wedge$$

$$(\neg\, ContainsKey(key)_{pre} \Rightarrow Count_{post} = Count_{pre} + 1 \wedge\ Count_{post} \geq 1) \wedge$$

$$(ContainsKey(key)_{pre} \Rightarrow Count_{post} = Count_{pre} \wedge Count_{pre} \geq 1)$$

This contract more closely relates the value of *Count* accounting for how the value changes depending on whether the dictionary already contained the key or not. The split on the case of whether the key being inserted already exists is a crucial disjunction to express this contract. The predicate synthesizer, which synthesizes linear relationships between inputs and an output feature, is responsible for synthesizing the predicate $Count_{post} = Count_{pre} + 1$.

The preceding contract is learned by our tool after 6 rounds of interaction. In each round, the learner fits the observed behaviors with a tight contract, and the test generator examines the contract to determine whether it can produce a test that violates the contract. The learner takes these new tests, enlarging the contract till it is deemed to be safe by the test generator.

3 PROGRAMS AND SAFE CONTRACTS

Methods and Observer 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/objects of type t range over. In the following, we assume that each variable v has an implicit type t associated with it, and use $\mathcal{D}(v)$ to denote its data domain.

We assume that we have a target method $m()$ with formal input parameters $\vec{p}$ and formal output parameters $\vec{q}$ (typically a single output) that we want to synthesize a contract for. Let us assume that these two sets of parameters are disjoint, without loss of generality.

Let us also fix a set of *pure* (i.e., side-effect free) *observer methods* that *have primitive return type.* These methods help query properties of the state of objects. Apart from the target method and observer methods, we have other methods for classes, including constructors and methods that mutate the object. These methods implicitly define the set of valid states V that the object can have, i.e., an object state is valid if and only if it can be reached by calling these methods.

Preconditions. For a target method m with input parameters $\vec{p}$ and output parameters $\vec{q}$, a precondition is a property of primitive parameters in $\vec{p}$ and properties of objects in $\vec{p}$ as well as the receiver object that receives m. Consequently, we express preconditions using a logic that refers to the primitive parameters in $\vec{p}$ and values of observer methods on objects in $\vec{p}$ as well as observer methods applied to the receiver object.

The role of preconditions is to restrict the input parameters and object state on which m is intended to be called. The contracts that we synthesize are with respect to a given precondition. We do not expect given preconditions to satisfy any particular property. The goal of contract synthesis is to find a contract for the given precondition. In particular, the contract will ignore inputs that do *not* satisfy the precondition. Also, contracts will ignore inputs satisfying the precondition but on which the method does not terminate or throws an exception.

Preconditions can be written by programmers, or simply set to *true* (to allow all inputs). They can also be *synthesized* automatically, e.g., to prevent the method m from throwing an exception [Astorga et al. 2019].

Contracts. In this paper, given a precondition, we consider the problem of synthesizing the contract for a method. The contract intuitively represents a *summary* of the method m, namely what *effect* it has on m's return and the object manipulated by m. More precisely, the contract relates input parameters and the object state before m is called with the return value of m and the object state after m exits.

Denoting the pre object state before m is called as o and the resulting post object state after m returns as o' and the return values as $\vec{q}$, a contract for m under a precondition $Precond(\vec{p}, o)$ is a predicate $Con(\vec{p}, o, \vec{q}, o')$. The contract says what kinds of behavior are permitted by m when m is called in a state satisfying the given precondition $Precond(\vec{p}, o)$ (the behavior of m when called on inputs and pre object states not satisfying the precondition is entirely ignored).

Contracts can express properties of objects (whether they be objects in $\vec{p}$ or $\vec{q}$ or the pre or the post object state of the receiver object o and o') using their respective observer methods. Note that we do not allow contracts to refer directly to the internal state of the objects. This requirement is a conscious choice to preserve modular design and development, as contracts are meant to be used by clients who can observe objects through only observer methods.

We can view our contracts also as postcondition assertions. Given a program P, we can modify P so that it records properties of the pre state in a set of fresh variables before it begins execution; let us call this transformed program P'. Then we can see a contract Con as an *assertion* at the exit of the program P' that we expect to hold on all input states that satisfy the precondition.

A contract is said to be *safe* if for every value of input parameters and valid pre object state satisfying the given precondition, if m executes and halts, the contract is satisfied by the input and output parameter values and the pre and post object states, i.e., the Hoare triple $\{Precond\}P'\{Con\}$ is valid.

4 TIGHT CONTRACTS: STRONG BUT SIMPLE

One of the challenges in learning contracts is to determine which of the multitude of safe contracts should be synthesized. Safe but trivial contracts such as *true* are too liberal/weak. The stronger a contract is, the more it says about a method. However, the strongest contract may not be expressible in any reasonable logic that we choose, and in this case, defining what the "best" contract should be is non-trivial.

For example, consider a program that returns an integer y that can be and always is an odd prime number. Assume that our logic is quantifier-free and admits only $=, \leq, \geq$ relations and arbitrary constants. Then the strongest contract, which is the set of all odd primes, is not expressible in the

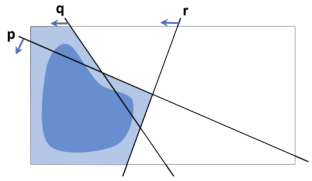

Fig. 2. Example space where $ite(p, r, q \wedge r) \lessdot r$.

logic. Moreover, the sequence of contracts $(y \geq 3)$, $(y = 3 \vee y \geq 5)$, $(y = 3 \vee y = 5 \vee y \geq 7)$, $(y = 3 \vee y = 5 \vee y = 7 \vee y \geq 11)$, . . ., is an infinite sequence of stronger and stronger contracts. Which one do we synthesize?

In this work, we concentrate on logics that are quantifier free and work by using Boolean combination of a finite set of atomic formulas (these atomic formulas are themselves *synthesized* for each method from an infinite set of formulas, as we describe later in Section 5). Consequently, we can think of contracts simply as a Boolean formula over a finite set of (synthesized) predicates $R = \{r_1, \ldots, r_n\}$.

Decision Trees. From now on, let us assume that contracts are written in the form of *decision trees* with conjunctive leaves. More precisely, for a set of Boolean variables V, the set of decision trees with conjunctive leaves is defined by the following grammar:

$$DT ::= ite(v, DT, DT) \mid C$$
$$C ::= true \mid v \wedge C \mid \neg v \wedge C$$

where $v \in V$. In the preceding grammar, "ite" terms stand for "if-then-else" terms, which evaluate to the second or third terms depending on whether the first formula is true or false, respectively. Note that the leaves are conjunctions of predicates and negated predicates. We refer to such formulas simply as decision trees in the sequel.

For a decision tree φ, let $d(\varphi)$ denote the number of *conditionals* in the decision tree—more precisely, the number of subterms of φ of the form $ite(\cdot, \cdot, \cdot)$. Note that for a purely conjunctive formula φ, $d(\varphi) = 0$.

Also, let $[\![\varphi]\!]$ denote the semantic space of valuations that satisfy φ. We say a formula φ is stronger (or strictly stronger) than φ' if $[\![\varphi]\!] \subseteq [\![\varphi']\!]$ (or $[\![\varphi]\!] \subset [\![\varphi']\!]$, respectively).

For example, consider the space depicted in Figure 2, where the strongest contract is depicted by the dark-blue shaded area, and three predicates p, q, and r each divide the space into two (the area to the left denoting the space satisfied by each predicate). If we allow no conditionals, then the tightest conjunctive formula that captures the contract is r, as the contract has states that satisfy (and do not satisfy) p as well as states that satisfy (and do not satisfy) q. However, if we are allowed one conditional, we find that the formula $ite(p, r, q \wedge r)$ (shown as the light-blue shaded region below) is a safe contract and is stronger than the formula r.

A Notion of Tight Contracts. In general, more conditionals in decision trees allow capturing smaller semantic spaces. However, it is also true that formulas with a large set of disjunctions can cause overfitting, making learning a safe contract take many rounds, and can lead to divergence.

We say a formula φ is *simpler* than φ' if it has fewer conditionals, i.e., $d(\varphi) \leq d(\varphi')$. There is hence a natural partial order $\preceq$ between formulas: $\varphi \preceq \varphi'$ iff

- $[\![\varphi]\!] \subseteq [\![\varphi']\!]$ (i.e., φ is stronger than φ'), and
- $d(\varphi) \leq d(\varphi')$ (i.e., φ is simpler than φ').

How do we define a notion of tightness of contracts for capturing strong *and* simple contracts? One possibility is that we ask for a Pareto optimal contract based on the preceding ordering. That is,

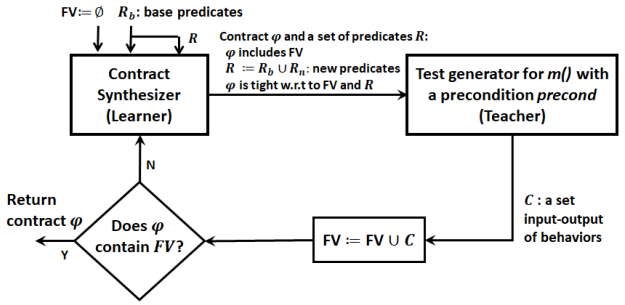

Fig. 3. Architecture for synthesizing safe and tight contracts modulo a test generator

we could ask for a *minimal* safe contract with respect to the preceding ordering. However, doing so is not always desirable. For example, a technique could learn only conjunctive formulas (i.e., have no conditionals) and learn the tightest conjunctive contract. The learned contract would be minimal by the preceding ordering, but is not necessarily good (as shown by the example in Figure 2, where the tightest formula with no conditionals is r, as well as the illustrative example in Section 2).

We propose a novel notion of tightness of contracts to combine how strong a contract is and how simple it is. Intuitively, if φ is a safe contract, but there is a contract that has *one more conditional* than φ and captures a smaller space, then we would reject φ as being tight (preferring the stronger formula with one more disjunct). However, if there exists no such stronger contract with an equal number or *one more conditionals*, then we declare φ a tight contract.

For instance, in the example in Figure 2, r is not a tight contract as with one more conditional we find that the formula $ite(p, r, q \wedge r)$ is a safe contract that is stronger than r. Moreover, it is easy to see that $ite(p, r, q \wedge r)$ is tight, as there is no formula with two conditionals that defines a stronger and safe contract.

We define the following "immediate ordering" $\lessdot$ on formulas: $\varphi \lessdot \varphi'$ iff (a) $d(\varphi) \leq d(\varphi') + 1$ and $[\![\varphi]\!] \subset [\![\varphi']\!]$.

Intuitively, $\varphi \lessdot \varphi'$ if φ uses at most one more disjunct than φ' and captures a strictly smaller space than φ does. For the example in Figure 2, $ite(p, r, q \wedge r) \lessdot r$, but $ite(p, ite(p, r, q \wedge r), r) \not\lessdot r$. Note that $\lessdot$ is not necessarily a partial order nor a preorder (it is not transitively closed, in general), but is acyclic. We are now ready to define tight formulas.

Definition 4.1 (Tight formulas). Let R be a finite set of Boolean variables and let S be a set of Boolean valuations of R. We say a formula φ is a *tight approximation* of S (or simply *tight*) if

- $[\![\varphi]\!] \supseteq S$ and
- There is no formula φ' such that $\varphi' \lessdot \varphi$ and $[\![\varphi']\!] \supseteq S$.

The above says that a formula φ is tight with respect to S if it overapproximates S and there is no stronger formula with at most one more conditional that overapproximates S.

We propose that the learning algorithm returns tight formulas for a given set of observed behaviors.

5 LEARNING TIGHT CONTRACTS MODULO A TEST GENERATOR

In this section, we describe our online learning architecture for synthesizing tight contracts (Definition 4.1) that are safe with respect to a test generator. We also present the core learning algorithm that learns tight concepts from sample behaviors.

5.1 Architecture for Online Learning of Contracts Safe Modulo a Test Generator

Our architecture, depicted in Figure 3, pairs a test generator (with fixed resources and assumed to be deterministic) with a contract synthesizer that works in rounds of communication. In each round, the contract synthesizer takes as input the set of input-output behaviors, denoted as FV, exposed by the test generator thus far, and proposes a conjecture contract φ that includes FV. The test generator then checks whether φ is an unsafe contract by finding input-output behaviors that witness that φ is not safe— i.e., the test generator finds valid input states that satisfy the precondition of the target method such that the execution of the method on this state halts and the resulting output state does not satisfy the contract φ. The generated input-output behaviors are accumulated in FV for the contract synthesizer to use in the next round. This process continues until the test generator is unable to find a witness that shows φ is unsafe. At this point, we can obviously deem φ to be safe *modulo the test generator* and return φ as the learned contract.

In each round, the learner synthesizes, simultaneously, a set of predicates R (being larger than a set of base predicates R_b and capturing more accurate input-output relationships) as well as a contract φ such that φ is tight with respect to R and the sample behaviors FV. This algorithm is the heart of our technical contribution and is next described in Section 5.2. Notice that when the algorithm terminates, we are guaranteed to have synthesized a tight contract that is safe modulo the test generator. We assume that the test generator is deterministic (when it uses randomization, we will fix random seeds) and has no persistent state between calls.

In the preceding architecture, the test generator is an oracle that can potentially be replaced by a *complete verifier* that truly checks whether the conjectured contracts are safe. However, what we crucially need are systems that also return counterexamples when contracts are unsafe. Sound verification engines typically use some form of abstraction and do not provide such counterexamples (for programs with recursion), and tools that can provide counterexamples are typically incomplete ones that can be seen as test generators. Our learning architecture and implementations are hence best understood to be meant to particularly work with test generators.

5.2 Synthesizer of Tight Contract (Learner)

We next describe one of our main technical contributions: our learning algorithm for tight contracts. Given a set of Boolean and integer features, a set of base predicates R_b, and a set of feature vectors FV, the goal of our algorithm named `SynthContract` is to synthesize a contract φ (a quantifier-free Boolean combination of predicates in R) that is tight with respect to FV. `SynthContract` synthesizes a larger set of predicates $R \supseteq R_b$ (Definition 4.1). The newly synthesized predicates are over a logic $\mathcal{L}$ that is infinite, and in our case arbitrary linear arithmetic expressions over integer features. This algorithm is the contract synthesizer (see Figure 3). We describe the algorithm as it operates in a single round.

The (passive learning) algorithm iterates using the following two sub-algorithms till it reaches a fixed point and finds a tight decision tree with respect to a set of predicates R:

`SynthStrongerDT` constructs a tight decision tree with respect to FV and the current set of predicates R. Note that this notion of tightness (Definition 4.1) is based on a Boolean abstraction of the predicates.

`SynthPredicate` grows the set of predicates R by synthesizing new predicate expressions over the logic $\mathcal{L}$ at a leaf of the decision tree to better capture the samples that flow to that leaf. In particular, the sub-algorithm synthesizes expressions at leaves that precisely characterize functional relationships of the output state with respect to the input state.

We describe these sub-algorithms *later*; we first describe the high-level algorithm of `SynthContract`, which orchestrates the construction by calling the preceding sub-algorithms.

Function `SynthContract` $(\overline{b}, \overline{i}, R_b, FV)$**:**
- **Input** $\overline{b}$, $\overline{i}$: Set of Boolean and integer features
- **Input** R_b: Set of (base) predicates over $\overline{b}$ and $\overline{i}$
- **Input** FV: Set of feature vectors over $\overline{b}$ and $\overline{i}$
- **Output** R: $R = R_b \cup R_n$, where R_n is a set of new predicates over $\overline{i}$, including all predicates returned by `SynthPredicate` on each leaf of ϕ
- **Output** ϕ: Formula over R such that if S is the set of Boolean feature vectors obtained by evaluating R on FV, then ϕ is a tight decision tree wrt S
- **Parameterized by** : `SynthPredicateDT`, `SynthStrongerDT`

// B is a set of Boolean variables, one for each predicate in R_b

$B \leftarrow \{b_r \mid r \in R_b\}$

$S \leftarrow$ Set of Boolean feature vectors obtained by evaluating R_b on FV; S is a feature vector over B

$R \leftarrow R_b$

$\phi \leftarrow true$

// ψ_B and ψ'_B will be trees over Boolean variables B

$\psi_B \leftarrow true$, $\psi'_B \leftarrow true$

// k is the number of conditionals in the constructed decision tree

$k \leftarrow 0$

while *true* **do**

  // find a strongest tree with k conditionals

  do

    $\psi_B \leftarrow \psi'_B$

    $\phi \leftarrow$ `ReplaceBoolWithPredicate`(ψ_B, B, R)

    $\phi, Q \leftarrow$ `SynthPredicateDT`$(\phi, \overline{i}, FV)$

    $R \leftarrow R \bigcup Q$

    $S \leftarrow S \oplus Q$ // Expand S by eval Q on FV

    $B \leftarrow B \cup \{b_q \mid q \in Q\}$

    $\psi_B \leftarrow$ `ReplacePredicateWithBool`(ϕ, B, R)

    $\psi'_B \leftarrow$ `SynthStrongerDT`(S, B, ψ_B, k)

  while $(\psi'_B \neq \psi_B)$

  // check whether there is a stronger tree with k+1 conditionals

  $\psi'_B \leftarrow$ `SynthStrongerDT`$(S, B, \psi_B, k{+}1)$

  // If there is not one, we return the current tree; otherwise, we increment k

  if $\psi'_B == \psi_B$ **then**

    return R,ϕ;

  $k \leftarrow k{+}1$

Function `SynthPredicateDT`(CDT, f_I, FV)**:**
- **Input** CDT: A decision tree
- **Input** f_I: Set of integer features
- **Input** FV: Set of of feature vectors
- **Output** DT: Decision tree with newly synthesized predicates at leaves
- **Output** P: Set of synthesized features

$P \leftarrow \emptyset$

$DT \leftarrow CDT$

foreach *leaf* l *in* DT **do**

  $FV' \leftarrow$ the set of feature vectors in FV that flow to l

  $Q \leftarrow$ `SynthPredicate` (f_I, FV')

  $P \leftarrow P \cup Q$

  Add the conjunct $\bigwedge Q$ to the formula at the leaf l of DT

return DT, P

Fig. 4. The Tight Contract Synthesis Algorithm

The `SynthContract` algorithm (Figure 4): We fix a set of integer features $\bar{i}$ and Boolean features $\bar{b}$. These integer and Boolean features are drawn from input parameters of the method under analysis, return variables, integer and Boolean observer methods evaluated on input objects, the receiver object (both pre and post), and returned objects.

The algorithm takes a set of *base predicates* R_b over $\bar{b}$ and $\bar{i}$. The caller constructs the set of base predicates R_b that include the Boolean features $\bar{b}$ as well as a bounded set of simple predicates over integer features (e.g., $x \geq 0$, $x = 0$, $x \leq 0$, $x = y$, $x \leq y$, where x and y are integer features in $\bar{i}$).

The algorithm takes as input (Lines 2–4) the sets $\bar{b}$, $\bar{i}$, R_b, as well as a set FV of feature vectors, where each vector represents values of features $\bar{b}$ and $\bar{i}$. The algorithm needs to produce a set R_n of newly synthesized predicates and a decision tree ϕ (over $R_b \cup R_n$) that is tight with respect to FV and $R_b \cup R_n$. These new predicates will be synthesized using the routine `SynthPredicate`, which is described later, at each leaf.

`SynthContract` works as follows. It works in iterations (the outer loop, Lines 14–28), where in each iteration, it tries to find a tight decision tree with k conditionals, incrementing k in each round. In each such iteration, for the fixed k and a current decision tree ψ_B, the inner loop in Lines 15–24 will try to find a decision tree stronger than ψ_B with k conditionals, and moving to it till a fixed point is reached. When exiting this loop, we know that there is no decision tree stronger than ψ_B with k conditionals. We then check whether there is a stronger decision tree than ψ_B with $k+1$ conditionals (Line 25). If not, ψ_B is in fact tight, and we return it. Otherwise, we take the stronger decision tree over $k+1$ conditionals as the current decision tree, increment k and iterate. Note that when we increment k, we do not know that there is no tight decision tree with k conditionals—all we know is that one strongest decision tree that we constructed with k conditionals is not tight.

The loop in Lines 15–24 does more than synthesizing stronger decision trees–it also synthesizes new predicates at leaves of decision trees, accumulating these predicates over time. We take the current decision tree ψ_B and synthesize predicates at each of its leaves (Line 18, which calls `SynthPredicateDT`, which in turn calls `SynthPredicate` on all leaves of the decision tree; see the bottom of Figure 4 for the procedure `SynthPredicateDT`). We then ask whether there is a decision tree that is stronger than ψ_B and uses only k conditionals over the newly expanded set of predicates (Line 23). This routine returns a stronger decision tree, if one exists, or returns ψ_B if it does not.

The algorithm uses multiple procedures. The procedure `ReplacePredicateWithBool` does the Boolean abstraction, replacing predicates in R with a corresponding Boolean variable in B, while the procedure `ReplaceBoolWithPredicate` does the reverse, replacing Boolean variables with their corresponding predicate.

The algorithm also calls the procedure `SynthPredicateDT`, which is given in Lines 29–42 in Figure 4. This procedure synthesizes predicates at the leaves of a decision tree. The routine first computes the set of feature vectors that *flow* to the leaves (Line 38). This set is the set of feature vectors that satisfy the conditionals corresponding to the path from the root to this leaf. For each leaf, the corresponding feature vectors are passed to `SynthPredicate` to synthesize new predicates that more accurately capture functional relationships between the output state and input state for the feature vectors that flow to that leaf. The procedure returns the new decision tree where the leaves have new conjuncts corresponding to newly synthesized predicates that hold at each leaf (Line 41).

The algorithm calls two procedures `SynthStrongerDT` and `SynthPredicate`, which do the following:

`SynthStrongerDT`: The problem here is, given a current decision tree and k, to find a strictly stronger decision tree (with k conditionals) that includes all the samples from S. This finding process requires searching through all possible decision trees with k conditionals, and doing

an inclusion check, between the current decision tree and the tree to be synthesized and vice versa. The inclusion check itself is a validity problem, and hence it turns out that finding such a tree can be formulated as a 2QBF problem. Section 5.3 below details this formulation.

`SynthPredicate`: This routine is called to find functional relationships between the input and output behaviors in a given subset of feature vectors S. The expressions that we synthesize are in $\mathcal{L}$, the logic of linear integer arithmetic. Section 5.4 details this formulation.

We express both of the preceding problems as syntax guided synthesis (SyGuS) [Alur et al. 2015] problems and solve them using SyGuS solvers [Alur et al. 2017; Reynolds et al. 2019]. A SyGuS specification is a pair consisting of a grammar, $\mathcal{G}$, and logical formula $\exists f\ \forall \vec{x}\ \varphi(f, \vec{x})$, interpreted over a theory T. A solution for such a synthesis specification is an expression e belonging in the language defined by the grammar $\mathcal{G}$ such that the logical formula still holds when we substitute the synthesized expression e for f, i.e. $\forall \vec{x}\ \varphi(e/f, \vec{x})$ is *valid* in T.

Correctness of `SynthContract`: It is easy to see that the `SynthContract` algorithm is correct when it terminates. The decision tree output is guaranteed to include S and be tight with respect to S and the returned set of predicates R.

The argument that it terminates (assuming that the `SynthStrongerDT` and `SynthPredicate` procedures terminate) is more tricky. The crucial property to note is that the `SynthPredicate` procedure is called on feature vectors that flow to the leaves of some decision tree. Since the number of subsets of feature vectors is finite, this procedure is called only finitely many times, and hence the total number of synthesized predicates over $\mathcal{L}$ is finite (for a given set of feature vectors). Consequently, the number of decision trees over such predicates is finite. Furthermore, since the algorithm always synthesizes and moves to decision trees that are successively stronger, across both the outer and inner loops in Figure 4, the algorithm always terminates.

One subtle point to note is that the algorithm is *not* guaranteed to find the tightest decision tree over the *smallest* number of conditionals. When the algorithm finds a strongest decision tree with k conditionals, and finds a stronger decision tree with $k+1$ conditionals (Line 25), and increments k, we are not guaranteed that there is *no* tightest decision tree with k conditionals (we just know that the current decision tree is not tight). However, the algorithm does heuristically try to return decisions trees with a small number of conditionals (since it increments k gradually).

5.3 Synthesizing Stronger Decision Trees

We next describe the `SynthStrongerDT` algorithm, which synthesizes stronger decision trees with k conditionals than a given decision tree. The inputs to the routine are a set of Boolean feature vectors S (feature vectors of length n), a set of Boolean variables $B = \{b_1, \ldots, b_n\}$, CDT, the "current" decision tree (over B) that includes S, and $k \in \mathbb{N}_{>=0}$. The i'th element in a feature vector corresponds to the Boolean variable b_i, and hence each feature vector is a valuation of B. Our goal is to find a decision tree DT (if one exists) that has k conditionals, includes S, and captures a strictly smaller space than CDT does (i.e., $DT \Rightarrow CDT$ holds and $CDT \Rightarrow DT$ does not hold).

For a feature vector $s \in S$, we denote by $s[i]$ (where $1 \leq i \leq n$), the i'th element of the vector. To accomplish this goal, we formulate a synthesis problem that asks whether there is a decision tree with four requirements: (1) it has at most k conditionals (branching inner nodes in the decision tree), (2) it includes all the feature vectors in S, and (3) the decision tree captures a smaller semantic space than CDT does (i.e., $DT \Rightarrow CDT$ holds), and (4) there is one witnessing valuation that belongs to CDT but not to DT. If all four requirements can be satisfied, this routine returns this newly synthesized tree.

We formulate the above problem as a SyGuS problem. Note that the preceding requirement (3) is an implication constraint, and hence expressed using universal quantification. Since the

constraints are Boolean, we can also think of this problem as a 2-QBF satisfiability problem of the form $\exists \vec{x}\ \forall \vec{y}\ \varphi(\vec{x}, \vec{y})$.

The existential variables $\vec{x}$ consist of two sets of variables, $\vec{c}$ and $\vec{w}$. The variables $\vec{c}$ are $k \cdot |B|$ in number, and denote the various choices for the conditionals in the decision tree. The variables in $\vec{c}$ are c_i^j, which stands for whether the j'th conditional in the decision tree (chosen in some order, say infix order) is filled with the i'th Boolean variable (we require for each j, precisely one c_i^j to be true). The vector $\vec{w}$ is of length n, and is meant to witness a valuation of the Boolean variables in B for showing that the synthesized formula is *strictly* stronger than the current decision tree CDT ((d) above). More precisely, we will require that CDT include $\vec{w}$ (seen as a valuation of B) while the synthesized decision tree DT excludes it.

We do *not* capture the precise conjunction of Booleans at the leaves of the decision tree using variables. Since we are aiming to synthesize strong decision trees, we can assume that at each leaf we have the strongest conjunction of Boolean variables that hold on the set of feature vectors that flow to the leaf.

We hence consider only such decision trees that have the tightest conjunction at the leaves. For a set of Boolean feature vectors S, the tightest conjunct at a leaf of a decision tree whose conditionals are fixed is *unique*— it is precisely the conjunction of all Boolean variables $b_i \in B$ such that for every sample $s \in S$ that reaches the leaf (i.e., every sample that satisfies the conditions that lie along the path to that leaf in the decision tree) has $s[i]$ to be true. Note that such a decision tree *automatically* includes all the samples $s \in S$, by design (i.e., satisfies the requirement (b) above).

Given a shape of a decision tree and conditionals that inhabit its interior nodes (such as $ite(b_3, ite(b_5, \cdot, \cdot), \cdot)$ or $ite(b_2, \cdot, ite(b_1, \cdot, \cdot))$), the decision tree is uniquely fixed as the leaves are the tightest conjuncts. This is why the existential variables $\vec{c}$ combined with the shape of the tree entirely determine the tree. Hence, for a given shape h of the decision tree, we can write a formula $eval_h(\vec{c}, \vec{v})$ that essentially checks whether the decision tree with conditionals chosen according to the choice determined by $\vec{c}$ and with tightest conjuncts at leaves includes a valuation $\vec{v}$ over B. The formulation of $eval_h$ is a bit technical, and we postpone its definition. Let us assume that we have this function defined logically.

We can now state the synthesis problem using the 2-QBF formula $\exists \vec{c}\ \exists \vec{w}\ \varphi$ where φ has the following conjuncts:

- $\forall \vec{u}.\ (\ eval_h(\vec{c}, \vec{u}) \Rightarrow CDT(\vec{u})\ )$
  In other words, the synthesized decision tree defines a space that is a subset of the space defined by the current decision tree (condition (c)). Note that the universal quantification over $\vec{u}$ is necessary to express this inclusion criterion.
- $CDT(\vec{w}) \wedge \neg eval_h(\vec{c}, \vec{w})$
  This conjunct says the valuation w witnesses that the synthesized decision tree excludes at least one valuation (namely $\vec{w}$) that is included by the current decision tree (condition (d)).

By enumerating various shapes h of decision trees with k conditionals and taking the disjunction of these decision trees, we formulate the synthesis problem for tighter decision trees than CDT that use k conditionals.

We then give this $\exists^*\forall^*$ formula to a solver in order to synthesize stronger decision trees. While we can use any 2-QBF solver, we use a SyGuS solver CVC4 [Reynolds et al. 2019] for uniformity with other synthesis tasks in our tool. If the solver gives a valuation for $\vec{c}$, we create a decision tree using the shape and by computing the tightest conjunct at each leaf. It is easy to see that this procedure terminates and is correct.

Formulating $eval_h$: We now show more details on the SyGuS encoding by giving the formulation of the function $eval_h(\vec{c}, \vec{u})$.

Let us first assume that for each leaf l in the shape h, we have a function expressed in logic $Reaches(\vec{u}, \vec{c}, \vec{l})$ that determines whether the valuation $\vec{u}$ reaches the leaf l in the decision tree determined by $\vec{c}$. This is expressed in logic easily: we take the path π from the root to l, and for each internal node n along π, assert that if $\vec{c}$ fills the node n with the predicate b_i, then $u[i]$ is true iff the path π takes the "true" direction from n.

We can now formulate $eval_h$ as

$$eval(\vec{c}, \vec{v}) : \bigwedge_{leaves\ l} \left[Reaches(\vec{v}, \vec{c}, l) \Rightarrow \bigwedge_{i} \left(\neg v[i] \Rightarrow \bigvee_{samples\ s\ |\ s[i]=F} Reaches(s, \vec{c}, l) \right) \right]$$

where i ranges from 1 to the number of Boolean variables in feature vectors.

The above says that $\vec{v}$ evaluates to true in the decision tree defined by $\vec{c}$ provided the leaf l that $\vec{v}$ reaches has the following property: for every i such that $\vec{v}$ is false, there is at least one sample feature vector s that has its i'th variable set to *false* that reaches the leaf l.

The argument of why the above is correct is as follows. Assume $\vec{v}$ reaches the (unique) leaf l and let i be an index. If $v[i]$ is true, then v will satisfy b_i, in case it is mentioned in the conjunct at l. If i is such that $v[i]$ is false, then in order for $\vec{v}$ to be accepted by the decision tree, the implicit conjunct at the leaf l should *not* mention b_i. Recall that b_i is not mentioned in the implicit conjunct at l iff there is some sample s that flows to the leaf l and has its i'th Boolean variable set to false. Hence we require that there is at least one sample s, with $s[i]$ set to false, that reaches the leaf l.

5.4 Synthesizing Predicates

We describe the algorithm `SynthPredicate` for synthesizing predicates. The inputs to this routine are CDT, a set of integer features $\bar{i}$, and a set of feature vectors FV, where CDT is the *current decision tree* over R that expresses a contract that includes FV (i.e., CDT holds true under every feature vector in FV). The goal of this routine is to synthesize predicates that express functional relationships between inputs and output features. More precisely, we synthesize functions of the integer features of the output state in terms of the integer features of the input state using expression synthesis techniques such as SyGuS. For each integer feature of the output state o (which can either be a return variable of type integer or the value of an observer method on an output object that returns an integer), we ask whether o can be realized as a function of the integer features of the pre state (which includes input parameters of integer type and values of observer methods on input objects, including the receiver object, i.e., written as `this`).

We specifically aim to discover functional relationships expressed in a logic of linear integer arithmetic. Let us fix the following grammar $\mathcal{G}_I$ of linear integer arithmetic expressions over the set of input integer features $I \subseteq \bar{i}$:

$$E \ ::= \ i \mid c \mid E + E \mid E - E$$

where $i \in I$ and $c \in \mathbb{Z}$. Let $\vec{I}$ denote an ordering of I.

We now want to synthesize, for any output integer feature o, a function $\mathcal{F}(\vec{I})$ expressible in the grammar $\mathcal{G}_I$ so that for every feature vector $fv \in FV$, $fv(o) = \mathcal{F}[fv[\vec{I}]/\vec{I}]$. In other words, we want the output value of o in each feature vector to be equal to the function computed by $\mathcal{F}$ using the input values of the feature vector.

We use SyGuS solvers [Alur et al. 2015, 2017; Reynolds et al. 2019] to find such arithmetic expressions. Note that a SyGuS (syntax guided synthesis) specification consists of a grammar and a

logical formula, $(\mathcal{G}, \varphi(\vec{x}))$ interpreted over a theory T. A solution for such a synthesis specification is an expression e such that e belongs to the language defined by the grammar $\mathcal{G}$ and such that $\forall \vec{x}\ \varphi(\vec{x})$ is *valid* in T.

Note that we do not need to use the quantified variables $\vec{x}$, and instead we use a formulation of the form:

$$\bigwedge_{fv \in FV} fv(o) = e[fv[\vec{I}]/\vec{I}]$$

where e is constrained to belong to the grammar $\mathcal{G}_I$. Note that the preceding specification has only constant integers and the function symbol e. We then use SyGuS solvers (see Section 6 for specific ones) to synthesize these expressions. Note that the above can be extended to any logic of quantifier-free expressions, say non-linear arithmetic, etc. as long as we have a synthesizer for expressions that fit input-output specifications.

When evaluating our tool, we synthesize the predicate, $Count_{post} = Count_{pre}+1$, for the method *Setter* in Figure 1. In previous work that evaluated Daikon for synthesizing contracts [Polikarpova et al. 2009], the authors in fact argue that the inability of Daikon to find such functional relationships is one of the key drawbacks of Daikon that prevent it from learning strong contracts.

5.5 Correctness

We formalize correctness, parameterized over the procedure `SynthPredicate` for synthesizing predicates in any logic $\mathcal{L}$.

Theorem 5.1. *Let the procedure* `SynthPredicate` *be deterministic and terminating. Then the algorithm* `SynthContract` *always terminates and produces a set of predicates $R \supseteq R_b$ and a decision tree DT over R that is tight with respect to R and the given feature vectors. Furthermore, the set of predicates R will include all predicates returned by* `SynthPredicate` *when called on the feature vectors that flow to each leaf of the returned decision tree DT.*

We emphasize that the termination argument works for learning a tight concept in any round and works for *any* grammar for producing predicates (the grammar can generate an infinite language and is not restricted to be linear arithmetic). All that we need to assume is that predicate synthesis is deterministic and terminating. Note however that the final contract learned and its quality depend on many aspects— the tests generated by the test generator, the predicates synthesized, the decision trees synthesized by synthesis engines, etc.

Note that our algorithm is nontrivial because not only does it find *a* decision tree that describes the behavior (this is what many related approaches using decision trees do) but it finds a provably *tight* decision tree, which calls for an iterative procedure, whose termination in each round is subtle and is argued in the Correctness of `SynthContract` subsection of Section 5.2.

6 EVALUATION

We implement our learning architecture in a tool named Precis. We adapt an industrial test generator Pex [Tillmann and De Halleux 2008] as the testing oracle. The contract synthesizer component adapts the CVC4SY [Reynolds et al. 2019] SyGuS solver to synthesize trees and the enumeration-based synthesis solver [Alur et al. 2015] to synthesize predicates.

To assess the effectiveness of our approach, we investigate the following six main research questions (RQs), where RQ4, RQ5, and RQ6 are *ablation studies*:

- **RQ1: How effectively can Precis learn *safe* contracts?**
- **RQ2: How effectively can Precis learn *strong* contracts?**

- **RQ3: How effective is Precis in comparison to Daikon^{AL} (Daikon [Ernst et al. 1999] adapted to an active learning setting paired with a test generator; see Section 6.2 for details)?**
- **RQ4: How much does the use of disjunctions (decision trees) contribute to the overall effectiveness of Precis?**
- **RQ5: How much does the use of predicate synthesis contribute to the overall effectiveness of Precis?**
- **RQ6: How effective is the choice of 1-tightness for contract synthesis? In particular, would 2-tight contracts be much better?**

6.1 Evaluation Subjects

We evaluate our approach on the large projects studied in previous work [Astorga et al. 2019] (for precondition synthesis). We conduct our evaluation on a total of 52 methods from 7 classes; 30 of these methods are from the .NET Runtime open source project and the remaining 22 are from the QuickGraph open source project. Table 1 shows the projects and number of methods used in our evaluation.

.NET Runtime[2] is an open source implementation of the .NET runtime, libraries, and shared host, and powers the .NET Framework[3] across major systems (MacOs, Windows, and Ubuntu). We evaluate on five commonly used classes from the foundational libraries, namely Stack, Queue, Set, Map, and ArrayList. **QuickGraph**[4] is a popular C# open source project. It provides mature implementations of graph-based data structures and algorithms. We evaluate on two key classes namely UndirectedGraph and BinaryHeap. We do not include classes from the CodeContract [Cousot et al. 2013] and HOLA [Dillig et al. 2013] benchmarks studied in previous work [Astorga et al. 2019] because these benchmarks are not representative of object-oriented programs and in fact do not manipulate objects.

6.2 Evaluation Setup

Test Generators. For the purpose of synthesizing contracts in our evaluation, our Precis tool uses the default values of configuration parameters in Pex. For the purpose of evaluating RQ1, we configure Pex-High, a version of Pex with higher resources.

The relevant configurations for PEX include Timeout (default: 120 seconds), MaxConstraintSolverTime (default: 2 seconds), MaxRunsWithoutNewTest (default: 100), and MaxRuns (default: 100). For Pex-High, these values are increased to 500, 10, 2147483647, and 2147483647, respectively. For RQ1, we also use Randoop [Pacheco and Ernst 2007] with its timeout option set to 600 seconds (default: 120). We want to have high confidence that our synthesized contracts are truly safe so we allow Randoop to run for a longer time compared to Pex.

Making these test generators effective typically requires supplying them with factory methods. We implement these factory methods (also known as data generators) manually to create objects of a specific type from primitive types or objects of other types. We give all the test generators the same factory methods. There is a line of work that helps developers write factory methods (i.e., data generators) [Gligoric et al. 2010; Nie et al. 2020].

Daikon^{AL}. Since Daikon works usually as a passive learner, we enhance Daikon to produce a new tool Daikon^{AL} that works in an online learning setting interacting with the test generator

[2] https://github.com/dotnet/runtime
[3] https://docs.microsoft.com/en-us/dotnet/core/introduction
[4] https://github.com/YaccConstructor/QuickGraph

Table 1. Statistics on the number of methods of each class, the average number of features per method, and the average number of base predicates per method

	ArrayList	BinaryHeap	Dictionary	HashSet	Queue	Stack	Un. Graph
Methods	9	7	7	4	5	5	15
Avg.#Features/method	4.70	4.71	3.86	3.25	3.00	3.75	7.74
Avg.#Base-Predicates /method	164.6	299.5	53.9	37.8	75.4	75.4	220.6

Pex. We build Daikon^{AL} using the C# front end for Daikon, namely Celeriac [Schiller et al. 2014]. We also configure Daikon^{AL}'s template language to use only client-visible variables to instantiate its templates.

Preconditions. For our evaluation, we annotate each method for evaluation with its precondition. We use the Proviso tool developed and released by Astorga et. al [Astorga et al. 2019] to derive preconditions automatically. In cases where Proviso does not converge to a "truly ideal precondition" (which is their notion of a correct precondition), we manually improve the precondition derived by Proviso.

Feature Selection. Our approach and Daikon^{AL} require a set of integer and Boolean features, described in Section 5.2, to build a set of base predicates. Features are various combinations of Boolean and integer observer methods applied to formal parameters and return variables (including objects of any type) of the method under analysis. We use the observer methods that are pure methods in the class and static observer methods natively available in C#. Parameter and return variables of integer and Boolean types are also features. We omit any observer methods whose precondition is *false*. This case can occur when the precondition of the method under analysis contradicts the observer method's precondition. The average number of the features and base predicates, per method, used in our evaluation is listed in Table 1.

6.3 RQ1: How effectively can Precis learn *safe* contracts?

Our tool is designed such that the returned synthesized contract is always safe with respect to the test generator. The purpose of RQ1 is to investigate the effectiveness of Precis in generating *truly safe contracts*, i.e., contracts that include all possible input-output behaviors (independent of the test generator). We employ two mechanisms to check for safety beyond the test generator used to synthesize the contract. Our first safety check uses two other test generators to find violations and our second safety check relies on manual inspection. For the first check, we use Randoop (a random test generator) and Pex-High (Pex with more resources). The manual inspection is done on only contracts on which the test generators cannot find any violation of safety. The average learning time for our tool ranges from as little as 2 minutes per method in a class to as long as 5 hours per method per class. More efficient algorithms for synthesizing tight contracts remain as a valuable future direction, not the focus of this work.

In Table 2, Column 2 (labeled with Precis) shows the evaluation results for RQ1. The column shows the number of contracts learned by Precis, for each subject, that are found safe by both Pex-High and Randoop. Precis is effective in synthesizing safe contracts: Precis learns safe contracts for 82% (43/52) of the methods. For our second safety check, we manually inspect the 43 contracts synthesized by Precis where neither test generator is able to find a safety violation. The manual check of a contract proceeds by understanding the contract to check whether it is an appropriate contract for the method. Apart from doing so, we take each leaf of the decision tree of the contract, and manually strive to analyze and synthesize a test input that leads to an output state that falsifies

Table 2. Results for RQ1 and RQ3: safety and relative strength of contracts inferred by Precis and Daikon^{AL}.

Classes	#Safe by Pex-High and Randoop (RQ1)		Precis vs Daikon^{AL}(RQ3)		
	Precis	Daikon^{AL}	Precis	Daikon^{AL}	Equivalent
ArrayList	9	9	3	0	6
BinaryHeap	5	5	5	0	0
Dictionary	7	7	4	0	3
HashSet	4	4	2	0	2
Queue	5	5	5	0	0
Stack	5	5	5	0	0
UndirectedGraph	8	4	1	0	0
Total	**43**	**39**	**25**	**0**	**11**

the constraint corresponding to the leaf. If our manual best efforts cannot find a reason why the contract is unsafe, we declare it to be safe. The results of our manual inspection (not shown in the table) show that indeed the remaining 43 contracts are safe. This finding shows not only the effectiveness of Precis but also the effectiveness of both test generators in approximating manual inspection. Out of the 43 cases that are safe, 10 are purely conjunctive while 33 have disjunctions. An example of a safe contract learned by Precis is given in Section 2.

The average number of rounds and samples generated by Precis for each subject are ArrayList: 5.5, 38.6; BinaryHeap: 4.3, 40.3; Dictionary: 4.7, 38.6; HashSet: 4.5, 36.5; Queue: 5, 27.6; StackTest: 6.0, 25.4; and UndirectedGraphTest: 4.8, 44.3.

6.4 RQ2: How effectively can Precis learn *strong* contracts?

To answer this question, we manually derive strong contracts for all the 52 methods, and compare the relative strength of the manually derived contracts and the contracts learned by Precis.

We assign one author solely for writing contracts; note that this author is not involved in technique or tool development. This author is given just the code of a subject, the available observer methods, and the grammar for contracts. We then check whether the manually derived contracts are safe. On safe contracts, we check whether they are stronger than the contracts learned by Precis.

We automatically conduct the comparison for strength between the manually derived contracts and the contracts learned by Precis. The automatic comparison leverages an SMT solver, namely Z3 [De Moura and Bjørner 2008], to do the two required inclusion checks.

Our results, shown in Table 3, indicate that there are substantially more cases where Precis learns contracts stronger than the manually derived contracts than the cases of the other way around (29 to 10). Furthermore, 23 out of the 29 stronger contracts learned by Precis are also truly safe. Out of the 52 contracts (for all the 52 methods, respectively), 10 are equivalent and 13 are incomparable between the ones learned by Precis and the ones manually derived. On cases where Precis does not learn a stronger contract, we notice that in some of these cases the learned contract relates two different aspects of the output state (which we do not consider in our logics for synthesis). In another example, the manually derived contract has a predicate that could have been synthesized by Precis but is not (likely because predicate synthesis is used only at leaves).

We compare the contracts learned by Precis given ideal preconditions from the previous work [Astorga et al. 2019] (and reported in Table 3) with contracts learned by Precis where preconditions are set simply to *true*. When given precondition *true*, the test generator can find inputs on which the target method throws exceptions and does not return in a non-error state. Our tool simply

Table 3. Results for RQ2: the number of methods where the contracts learned by Precis are stronger, the manually derived contracts are stronger, and the contracts resulted from the two ways are equivalent (and incomparable), respectively.

Classes	Precis	Manual	Equivalent	Incomparable	Total
ArrayList	3	0	2	4	9
BinaryHeap	4	0	3	0	7
Dictionary	4	0	3	0	7
HashSet	3	0	1	0	4
Queue	2	0	0	3	5
Stack	3	0	0	2	5
UndirectedGraph	10	0	1	4	15
Total	**29**	**0**	**10**	**13**	**52**

discards such inputs, since the goal is only to learn contracts/summaries on which the method does terminate. Note that the preconditions affect only the test generator, and through this, the contract synthesized (the synthesis tool does not consider the precondition directly). With a 5 hour timeout, we are able to generate contracts for 43 out of the 52 methods. Of these 43, on 29 methods the contracts learned with ideal preconditions are equivalent to the contracts learned with preconditions being true. On 5 methods the contracts learned with preconditions being true are stronger, and for 3 methods contracts learned with ideal preconditions are stronger. Some of the contracts learned with preconditions being true end up capturing aspects of the preconditions needed in order for the methods not to throw an exception.

6.5 RQ3: How effective is Precis in comparison to Daikon^{AL}?

We compare Precis with Daikon^{AL}, an enhancement of Daikon placed in an online learning loop with a test generator. We choose Daikon for comparison because it pioneers the research on dynamically inferring likely invariants. The Daikon tool includes fixed heuristics for inferring disjunctions [Dodoo et al. 2003] and the tool is continuously updated (e.g., its latest release by the time of this paper writing was made in March 2021). In particular, we are interested in knowing how often, in our evaluation subjects, Precis is able to learn safer and/or stronger contracts compared to Daikon^{AL}.

For checking safety, we use the same mechanism as the one used to address RQ1, i.e. using the two test generators, namely Randoop and Pex-High. Our results are shown in Columns 2–3 of Table 2.

Our results show that Precis synthesizes more safe contracts than Daikon^{AL} (43 vs 39). We notice that the heuristics used by Daikon to synthesize disjunctive concepts sometimes lead to overfitting the observed behaviors, leading it to produce truly unsafe contracts. For instance, the method $EdgeCount()$, given an undirected graph, g, returns the number of edges in g, and this class has an observer method $VertexCount()$, which returns the number of nodes in the graph. The contract that Daikon^{AL} infers is $\cdots \wedge VertexCount_{pre} = VertexCount_{post} \wedge (VertexCount_{post} = 0 \vee VertexCount_{post} = 1 \vee VertexCount_{post} = 7)$, being clearly unsafe.

For checking the relative strength of contracts, we use Z3 in the same way as in RQ2. Columns 4–6 in Table 2 show the number of the methods where the contracts learned by Precis are stronger, the ones inferred by Daikon^{AL} are stronger, and the number of the methods where the contracts by the two tools are equivalent. Our results show that ~50% (25/52) of contracts learned by Precis are strictly stronger than their Daikon^{AL} counterparts, while none of the contracts inferred by Daikon^{AL} are stronger than those learned by Precis.

Table 4. Results for RQ4: the number of the methods on which PRECIS learns stronger, weaker, equivalent, and incomparable contracts compared to those learned by PRECIS$_{Conj}$, respectively.

Classes	Precis	Precis$_{Conj}$	Equivalent	Incomparable	Total
ArrayList	2	0	7	0	9
BinaryHeap	2	1	4	0	7
Dictionary	6	0	1	0	7
HashSet	3	0	1	0	4
Queue	4	0	1	0	5
Stack	4	0	1	0	5
UndirectedGraph	12	0	0	3	15
Total	**34**	**1**	**14**	**3**	**52**

We further analyze our tool's components that contribute to the strength of our synthesized contracts over the ones inferred by DAIKON^{AL}. Of the 25 methods where the contracts learned by PRECIS is stronger, we find that for 9 methods, PRECIS with predicate synthesis produces a safe and stronger contract than PRECIS without predicate synthesis. For these 9 methods, we run DAIKON^{AL} seeded with the synthesized predicates, and check whether it is now able to produce stronger contracts. We find that on 6 methods DAIKON^{AL} still produces weaker contracts than PRECIS, while on the other 3 it produces equivalent contracts.

On the 25 methods where the contracts learned by PRECIS is stronger, in 2 of them DAIKON^{AL} simply infers *true* and in 14 of them it infers a conjunctive contract while PRECIS learns a contract with disjunctions. In 30% (16/52) of the cases, the contracts by the two tools are incomparable.

6.6 RQ4: How much does the learning of disjunctive contracts (using decision trees) contribute to the overall effectiveness of PRECIS?

To understand the benefits of synthesizing contracts with disjunctions/conditionals, we create a version of our tool namely PRECIS$_{Conj}$ that generates strongest conjunctive contracts, using the elimination learning algorithm [Mitchell 1997]. This conjunctive learner is similar to the one used by DAIKON, except that we keep our predicate synthesis component and we do not have the heuristics that DAIKON uses for mildly disjunctive formula synthesis.

We compare this tool with PRECIS; our results are shown in Table 4, including the total number of stronger contracts learned by PRECIS compared to PRECIS$_{Conj}$ and vice-versa, equivalent contracts, and incomparable contracts. Out of the 43 methods with truly safe contracts learned by PRECIS, 42 of those methods also have safe contract learned by PRECIS$_{Conj}$. Among these 42 methods, 26 (61%) have contracts that are stronger than those contracts learned by PRECIS$_{Conj}$. None of contracts learned by PRECIS$_{Conj}$ are stronger than those learned by PRECIS. Our results indicate that synthesizing contracts with disjunctions/conditionals is crucial in learning stronger contracts, but care is needed as facilitated by the notion of tight techniques in this paper.

6.7 RQ5: How much does the use of predicate synthesis contribute to the overall effectiveness of PRECIS?

To understand the benefits of synthesizing predicates at the leaves of decision trees, we create a version of our tool namely PRECIS$_{w/o\ synthesis}$ that synthesizes decision trees over only the original set of predicates. For this set of experiments, we set a timeout of five hours for each method under analysis. In total, we learn contracts for 45 methods. The results are shown in Table 5. Out of those 45, 6 are not safe contracts. In comparison with PRECIS, 9 contracts are incomparable, 20 are equivalent, and 6 are stronger than those learned by PRECIS. In these 6 cases, 5 of the contracts are not safe while the remaining 1 is safe. On the other hand, PRECIS learns 10 (10/45 = 22%) contracts

Table 5. Results for RQ5: the number of methods on which PRECIS learns stronger, weaker, equivalent, and incomparable contracts compared to those learned by $\text{PRECIS}_{w/o\ synthesis}$, respectively

Classes	Precis	$\text{Precis}_{w/o\ synthesis}$	Equivalent	Incomparable	Total
Stack	3	0	2	0	5
Queue	1	0	3	1	5
HashSet	2	0	2	0	4
Dictionary	3	0	4	0	7
ArrayList	0	2	5	2	9
BinaryHeap	0	0	1	0	1
UndirectedGraph	1	4	3	6	14
Total	**10**	**6**	**20**	**9**	**45**

Table 6. Results for RQ6: the number of methods in which PRECIS learns stronger, weaker, equivalent, and incomparable contracts compared to those learned by $\text{PRECIS}_{k=2}$, respectively

Classes	Precis	$\text{Precis}_{k=2}$	Equivalent	Incomparable	Total
ArrayList	0	4	5	0	9
Dictionary	0	0	7	0	7
HashSet	0	0	4	0	4
Queue	0	1	4	0	5
Stack	1	0	4	0	5
Total	**1**	**5**	**24**	**0**	**30**

that are stronger than $\text{PRECIS}_{w/o\ synthesis}$. Of those 10, 9 are also safe. Our results indicate that synthesizing predicates has a definite positive impact in learning stronger contracts.

6.8 RQ6: How effective is the choice of 1-tightness for contract synthesis? In particular, would 2-tight contracts be much better?

To understand the benefits of our 1-tightness design choice for contract synthesis, we create a version of our tool namely $\text{PRECIS}_{k=2}$ that synthesizes 2-tight contracts. $\text{PRECIS}_{k=2}$ takes much longer for the learning phase in each round, and we set a timeout of five hours. In total, we are able to compare on 30 methods. Table 6 shows our results. For most cases, 2-tight contracts are equivalent in the end to 1-tight contracts (for 24/30 = 80%). In some cases, the 2-tight decision trees are indeed stronger (5/30 = 16%) and in one case 1-tight decision trees are stronger. In all these 5 cases, the synthesized contracts are also safe. However, the gain in expressiveness of 2-tight contracts comes at a performance cost as in 22 of the methods PRECIS_{2k} could not finish in the allotted five hours. Our results show that on these subjects 1-tight contracts are the sweet spot between efficiency and strength.

7 RELATED WORK

The work most closely related to ours is DAIKON [Ernst 2000; Ernst et al. 1999], which also infers specifications in a (mostly) black-box manner by observing dynamic executions (and which we compare our approach against in Section 6). There are multiple crucial differences though. First, DAIKON works by learning *passively* from a set of executions, while our approach works by using *online* learning with a test generator that produces counterexamples to learned concepts. If the passive tests happen to accidentally satisfy a property, DAIKON would learn that as a likely invariant. Our approach is however guaranteed to be safe with respect to the test generator, albeit with respect

to a fixed limit on testing resources. Second, Daikon mostly learns only conjunctive concepts, admitting the simple elimination learning algorithm [Mitchell 1997]. Our approach instead learns arbitrary Boolean combinations of predicates using decision trees, calling for notions such as tightness and learning algorithms for tight formulas that we develop in this paper. Daikon does have some support for learning *implications* [Dodoo et al. 2003; Ernst 2000], but these implications are formed using conditionals that occur in the code of the program under analysis using mild static analysis. More importantly, this kind of support would result in learning conditions involving an object's internal variables that are not visible to the client, which we avoid in our work. The work reported by Polikarpova et al. [2009] compares Daikon-inferred contracts against user-written contracts, and identifies multiple shortcomings with Daikon, including not being able to learn arithmetic expression relations (which we do with expression synthesis using SyGuS) or learn more complex Boolean expressions of the form $p \Rightarrow q$, $\neg(p \wedge q)$ (which we do by learning decision trees).

In this work, our approach uses a testing oracle instead of a verifier for practical reasons (see also discussion regarding verification oracles in Section 3). There have been various approaches to mining specifications based on automata learning [Alur et al. 2005; Ammons et al. 2002; Henzinger et al. 2005; Whaley et al. 2002; Xie et al. 2006] or dynamic and symbolic analysis [Astorga et al. 2018; Csallner et al. 2008; DeFreez et al. 2019; Le et al. 2019]. The predicate synthesis aspect of our work is similar to the PIE approach [Padhi et al. 2016] that proposes feature expression synthesis in order to learn preconditions as well as loop invariants, but not strong/tight contracts. The work by Fraser and Zeller [2011] infers assertions for test code by mutation testing to extract negative examples and employ a Daikon-like approach to generate conjunctive assertions. The work by Jahangirova et al. [2016] also uses mutation testing to infer assertions, and iterates over rounds *with the programmer*. The work by Molina et al. [2021] similarly infers program postconditions using an evolutionary algorithm to improve postconditions over generations. It mutates valid input-output pairs derived from test generation to produce potential invalid pairs, albeit in an unsound way. The work by Terragni et al. [2020] similarly uses a co-evolutionary algorithm to improve program postconditions. Given a wrong postcondition, this work infers more precise postconditions. The efficacy of the generated postconditions is also measured by how well they accept valid input-output pairs and how well they reject invalid input-output pairs generated through mutation testing. Our work is similar in that it uses testing, but our work learns purely from positive examples, rather than mutated ones that are unsoundly assumed to be negative. Furthermore, our work can synthesize Boolean contracts and not just conjunctive ones. Another line of work explores mining specifications from natural language available in source code [Pandita et al. 2012; Zhai et al. 2020]. Such work tries to capture the intent of developers but relies on the availability of API documentation or code comments.

Note that the preceding related work is implemented for Java programs while our subjects and tools are for C#, making a direct empirical comparison hard. Various data-driven learning approaches have also been proposed for various synthesis tasks for program analysis [Churchill et al. 2019; Ezudheen et al. 2018; Garg et al. 2016; Neider et al. 2016; Zhu et al. 2018], including synthesizing annotations to help verification and prove equivalence of programs. Other work by Betts et al. [2012] uses a combination of heuristics and the Houdini algorithm for inferring invariants and contracts for verifying race- and divergence-freedom of GPU kernels. Candidate invariants are first heuristically generated and then are passed to Houdini to verify which candidates actually hold and are inductive. In another domain, the work of Newcomb et al. [2020] synthesizes rewrite expressions to prove properties of code required for efficient compilation in Halide.

8 CONCLUSION

In this paper, we have presented a novel approach for learning safe and strong contracts from code with guarantees modulo a test generator. Our synthesis approach is effective on our benchmarks, and seems promising to be applied to aid downstream problems such as unit testing, runtime monitoring (e.g., for anomaly detection), verification, regression checking (e.g., for detecting change of contracts as software evolves), which we think are interesting future directions. On the contract synthesis front, valuable future directions to pursue include exploring a richer class of logics that state more expressive properties such as an abstract datatype with a ghost state that captures more accurate functional descriptions of what methods do. Finally, our passive one-class learning of tight contracts could be useful in other domains (such as learning likely invariants from dynamic traces on inputs), as it is a stand-alone module that need not be paired with a test generator.

ACKNOWLEDGMENTS

This work is supported in part by NSF under grant no. CCF-1816615, a research grant from Amazon, a Discovery Partner's Institute (DPI) science team seed grant, and the GEM fellowship. We thank the anonymous reviewers for their valuable reviews and comments.

REFERENCES

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. IOS Press, 1–25.

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017)*. 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

Rajeev Alur, Pavol Černý, P. Madhusudan, and Wonhong Nam. 2005. Synthesis of Interface Specifications for Java Classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*. 98–109. https://doi.org/10.1145/1047659.1040314

Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*. 4–16. https://doi.org/10.1145/565816.503275

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 2019)*. 775–787. https://doi.org/10.1145/3314221.3314641

Angello Astorga, Siwakorn Srisakaokul, Xusheng Xiao, and Tao Xie. 2018. PreInfer: Automatic Inference of Preconditions via Symbolic Analysis. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2018)*. 678–689. https://doi.org/10.1109/DSN.2018.00074

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. 2004. The Spec# Programming System: An Overview. In *Proceedings of International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*. 49–69. https://doi.org/10.1007/978-3-540-30569-9_3

Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a Verifier for GPU Kernels. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2012)*. 113–132. https://doi.org/10.1145/2384616.2384625

Feng Chen and Grigore Roşu. 2007. MOP: An Efficient and Generic Runtime Verification Framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007)*. 569–588. https://doi.org/10.1145/1297027.1297069

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. 1027–1040. https://doi.org/10.1145/3314221.3314596

Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *Proceedings of International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013)*. 128–148. https://doi.org/10.1007/978-3-642-35873-9_10

Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic Symbolic Execution for Invariant Inference. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*. 281–290. https:

//doi.org/10.1145/1368088.1368127

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V. Thakur. 2019. Effective Error-specification Inference via Domain-knowledge Expansion. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. 466–476. https://doi.org/10.1145/3338906.3338960

Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2013)*. 443–456. https://doi.org/10.1145/2544173.2509511

Nii Dodoo, Lee Lin, and Michael D. Ernst. 2003. *Selecting, Refining, and Evaluating Predicates for Program Analysis.* Technical Report MIT-LCS-TR-914. MIT Laboratory for Computer Science, Cambridge, MA.

Michael D. Ernst. 2000. *Dynamically Discovering Likely Program Invariants.* Ph.D. University of Washington Department of Computer Science and Engineering, Seattle, Washington.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*. 213–224. https://doi.org/10.1145/302405.302467

P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. 2018. Horn-ICE Learning for Synthesizing Invariants and Contracts. In *Proceedings of the Annual ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2018)*. Article 131, 25 pages. https://doi.org/10.1145/3276501

Manuel Fähndrich. 2010. Static Verification for Code Contracts. In *Proceedings of the 17th International Conference on Static Analysis (SAS 2010)*. 2–5. https://doi.org/10.1007/978-3-642-15769-1_2

Robert W. Floyd. 1960. An Algorithm Defining ALGOL Assignment Statements. *Commun. ACM* 3, 3 (1960), 170–171. https://doi.org/10.1145/367149.367170

Gordon Fraser and Andreas Zeller. 2011. Generating Parameterized Unit Tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011)*. 364–374. https://doi.org/10.1145/2001420.2001464

Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants Using Decision Trees and Implication Counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*. 499–512. https://doi.org/10.1145/2837614.2837664

Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test Generation through Programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010)*. 225–234. https://doi.org/10.1145/1806799.1806835

Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. 2005. Permissive Interfaces. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*. 31–40. https://doi.org/10.1145/1081706.1081713

Charles Antony Richard Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test Oracle Assessment and Improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. 247–258. https://doi.org/10.1145/2931037.2931062

Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: Using Dynamic Analysis to Infer Program Invariants in Separation Logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. 788–801. https://doi.org/10.1145/3314221.3314634

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes* 31, 3 (2006), 1–38. https://doi.org/10.1145/1127878.1127884

Bertrand Meyer. 1988. *Object-Oriented Software Construction* (1st ed.). Prentice-Hall, Inc., USA.

Thomas M. Mitchell. 1997. *Machine Learning* (1 ed.). McGraw-Hill Education.

Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. 2021. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE 2021)*. 1223–1235. https://doi.org/10.1109/ICSE43902.2021.00112

Mary M. Moya and Don R. Hush. 1996. Network Constraints and Multi-objective Optimization for One-class Classification. *Neural Networks* 9, 3 (1996), 463–474. https://doi.org/10.1016/0893-6080(95)00120-4

Daniel Neider, Shambwaditya Saha, and P. Madhusudan. 2016. Synthesizing Piece-Wise Functions by Learning Classifiers. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016)*. 186–203. https://doi.org/10.1007/978-3-662-49674-9_11

Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. 2020. Verifying and Improving Halide's Term Rewriting System with Program Synthesis. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2020)*. Article 166, 28 pages. https://doi.org/10.1145/3428234

Pengyu Nie, Marinela Parovic, Zhiqiang Zang, Sarfraz Khurshid, Aleksandar Milicevic, and Milos Gligoric. 2020. Unifying Execution of Imperative Generators and Declarative Specifications. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2020)*. Article 217, 26 pages. https://doi.org/10.1145/3428285

Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA 2007)*. 815–816. https://doi.org/10.1145/1297846.1297902

Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*. 42–56. https://doi.org/10.1145/2908080.2908099

Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring Method Specifications from Natural Language API Descriptions. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*. 815–825. https://doi.org/10.1109/ICSE.2012.6227137

Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. 2009. A Comparative Study of Programmer-Written and Automatically Inferred Contracts. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA 2009)*. 93–104. https://doi.org/10.1145/1572272.1572284

Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2019. cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis. In *Proceedings of International Conference on Computer Aided Verification (CAV 2019)*. 74–83. https://doi.org/10.1007/978-3-030-25543-5_5

Todd W. Schiller, Kellen Donohue, Forrest Coward, and Michael D. Ernst. 2014. Case Studies and Tools for Contract Specifications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. 596–607. https://doi.org/10.1145/2568225.2568285

J. M. Spivey. 1988. *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press, USA.

Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary Improvement of Assertion Oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. 1178–1189. https://doi.org/10.1145/3368089.3409758

Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: White Box Test Generation for .NET. In *Proceedings of the 2008 International Conference on Tests and Proofs (TAP 2008)*. 134–153. https://doi.org/10.1007/978-3-540-79124-9_10

John Whaley, Michael C. Martin, and Monica S. Lam. 2002. Automatic Extraction of Object-Oriented Component Interfaces. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*. 218–228. https://doi.org/10.1145/566172.566212

Tao Xie, Evan Martin, and Hai Yuan. 2006. Automatic Extraction of Abstract-Object-State Machines from Unit-Test Executions. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*. 835–838. https://doi.org/10.1145/1134285.1134427

Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: Translating Natural Language Comments to Formal Program Specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. 25–37. https://doi.org/10.1145/3368089.3409716

He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A Data-Driven CHC Solver. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. 707–721. https://doi.org/10.1145/3192366.3192416