Discovering Relational Specifications

Calvin Smith
University of Wisconsin–Madison
USA

Gabriel Ferns
University of Wisconsin–Madison
USA

Aws Albarghouthi
University of Wisconsin–Madison
USA

ABSTRACT

Formal specifications of library functions play a critical role in a number of program analysis and development tasks. We present Bach, a technique for discovering likely *relational specifications* from data describing input–output behavior of a set of functions comprising a library or a program. Relational specifications correlate different executions of different functions; for instance, commutativity, transitivity, equivalence of two functions, etc. Bach combines novel insights from program synthesis and databases to discover a rich array of specifications. We apply Bach to learn specifications from data generated for a number of standard libraries. Our experimental evaluation demonstrates Bach's ability to learn useful and deep specifications in a small amount of time.

CCS CONCEPTS

• **Theory of computation** → Logic and verification; Logic and databases; • **Software and its engineering** → **Dynamic analysis**; • **Security and privacy** → *Logic and verification*;

KEYWORDS

Hyperproperties, Datalog, Specification Mining

ACM Reference Format:
Calvin Smith, Gabriel Ferns, and Aws Albarghouthi. 2017. Discovering Relational Specifications. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106279

1 INTRODUCTION

Formal specifications of library functions play a critical role in a number of settings: In program analysis and verification, specifications are essential to efficiently and precisely analyzing applications that use libraries whose code is unavailable or too complex to analyze. In software engineering, formal specifications can be used to unambiguously document libraries and APIs, as well as aid developers in program evolution. We address the problem of discovering a rich class of specifications defining behaviors of a set of functions.

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106279

Problem setting Imagine we are given a set of functions $f_1, \ldots, f_n$ along with a dataset D representing a partial picture of the input–output behavior of each function f_i, perhaps collected through random testing or instrumentation. We ask the following question:

> *What can we learn about the set of functions $f_1, \ldots, f_n$ by simply analyzing the dataset D?*

We present a novel and expressive algorithm, called Bach, that is able to discover likely *relational specifications* that correlate (*i*) different executions of a single function or (*ii*) different executions within collections of functions. In other words, Bach learns *hyperproperties* [6]—this is in contrast to traditional techniques that discover properties of single executions (invariants) [8]. For instance, Bach may learn the following specifications from some input–output data, where all variables are implicitly universally quantified:

$$\texttt{add}(x, y) = z \Leftrightarrow \texttt{add}(y, x) = z \tag{1}$$

$$\texttt{gt}(x, y) = t \wedge \texttt{gt}(y, z) = t \Rightarrow \texttt{gt}(x, z) = t \tag{2}$$

$$\texttt{trim}(\texttt{uppercase}(x)) = y \Leftrightarrow \texttt{uppercase}(\texttt{trim}(x)) = y \tag{3}$$

$$x > 0 \wedge \texttt{abs}(x) = y \Rightarrow \texttt{abs}(y) = x \tag{4}$$

Specification 1 is a *bi-implication* specifying that `add` is a commutative function. Specification 2, on the other hand, is an *implication* specifying transitivity of `gt` (greater than). Bach may also discover specifications that correlate different functions; e.g., Specification 3 specifies that the composition $\texttt{trim} \circ \texttt{uppercase}$ is equivalent to $\texttt{uppercase} \circ \texttt{trim}$ (where `trim` removes whitespace from a string and `uppercase` turns all characters to uppercase). Further, Bach may discover sophisticated specifications that are *refined* with additional constraints; for instance, Specification 4 specifies that the function `abs` (absoulte value) is invertible on positive integers.

Primary challenges There are three primary challenges that arise in learning relational specifications from a dataset: (*i*) *What does it mean for a specification to explain (partial) input–output behavior?* (*ii*) *How do we efficiently handle large amounts of input–output data?* (*iii*) *The space of possible relational specifications is vast, so how do we efficiently traverse the search space?*

We now describe how Bach tackles these challenges. Figure 1 provides a high-level overview of Bach.

Consistency verifier The first piece of the puzzle is defining what it means for a specification to explain a dataset. We view a specification as a first-order formula $\mathcal{F}$, and the given dataset as a *partial interpretation* D. We formalize what it means for D to be a model of $\mathcal{F}$. Bach employs a notion of *evidence* to rank specifications. If there exists any *negative evidence*—e.g., a counterexample to transitivity of a function—then the specification is considered inconsistent with the data and discarded. Otherwise, a specification is considered *more likely* to be true depending on a measure of the *positive evidence* that is available for it.

The *specification consistency verifier* checks specifications on a given dataset. Since we are potentially dealing with thousands of

input–output examples per function, we could easily incur a prohibitive polynomial blowup when evaluating a specification, e.g., evaluating $f(g(x), h(y))$ requires us to evaluate f on the Cartesian product of the available outputs of g and h in the dataset. To efficiently handle large amounts of examples, we exploit the insight that we can encode the specification consistency checking problem as a set of *non-recursive, relational Horn clauses* (a *union of conjunctive queries*, which is a subset of SQL), allowing us to delegate the problem to efficient, scalable databases or Datalog solvers.

Induction and abduction engines The second piece of the puzzle is how to automatically discover specifications. Our first insight is that we are searching for a specification (a logical formula) that is comprised of a *set* of "programs" (compositions of functions) and *connections* between them. Consider the transitivity formula:

$$\underbrace{f(x,y)=t}_{p_1} \wedge \underbrace{f(y,z)=t}_{p_2} \Rightarrow \underbrace{f(x,z)=t}_{p_3}$$

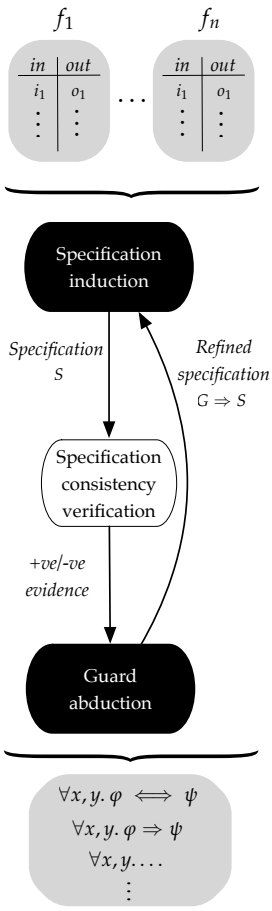
Here we have 3 programs, two on the left of the implication (p_1, p_2) and one on the right (p_3). The programs are *connected* by sharing their inputs and outputs through quantified variables.

Following this observation, to discover specifications, we utilize a *specification induction engine* that traverses the space of *sets of programs* and connections between them. This is analogous to how an inductive synthesis algorithm searches for a *single* program satisfying some property—here, we search for a set of programs.

If the induction engine discovers a specification S that is *too strong* to hold on the given dataset, the *guard abduction engine* asks the question: *what do we need to know in order to make the specification hold?* Viewed logically, the abduction engine *weakens* the specification S by qualifying it with some guard G, resulting in $G \Rightarrow S$. In practice, we exploit the insight that the guard abduction problem can be reduced to a *classification problem* by splitting data into positive and negative sets, those that satisfy the specification and those that do not. By alternating between induction and abduction, Bach is able to learn a rich array of specifications.

Implementation We implemented Bach and applied it to learn specifications of a range of Python libraries, including a geometry module and an SMT solver's API. Our results demonstrate Bach's ability to discover useful and elegant specifications explaining interactions between functions. While Bach learned a number of expected specifications, we were pleasantly surprised by some of the non-obvious specifications it managed to infer (see Section 6).

Most related work The most closely related work to ours is Claessen et al.'s [5], which discovers equational specifications through random testing. The class of specifications learnable by Bach is richer in a number of dimensions: In addition to learning equivalences (as bi-implications) between pairs of programs, Bach is able to (*i*) learn implications between pairs of programs; (*ii*) learn equivalences and implications over sets of programs, e.g., for properties like transitivity, which correlate executions between more than two *copies* of a program; and (*iii*) abduce guards on specifications. Further, Bach operates in a *black-box* setting: we do not assume access to library code or a random test generator. Instead, we directly operate on data, making our approach general, perhaps even beyond software specifications, e.g., hardware components and networks.



Dataset

input–output data

The input to Bach is a relational data set $\mathcal{D}$ where each relation describes a subset of the input–output relation induced by some function f_i.

Learning engine

specification induction engine

The specification induction engine attempts to learn specifications of the form $\Psi \iff \Phi$ or $\Psi \Rightarrow \Phi$, where Ψ and Φ are conjunctions of function applications.

specification consistency verifier

The specification consistency verifier checks whether a given specification is consistent with the given data set—that is, the data set does not falsify the specification and that there is *some* evidence of it holding.

guard abduction engine

The abduction engine refines specifications produced by induction engine, by augmenting them with additional constraints, e.g., $\Psi \iff \Phi$ becomes $G \Rightarrow (\Psi \iff \Phi)$.

Learned specifications

output stream

The output of Bach is a stream of specifications learned from the given data set.

Figure 1: Main components of the Bach algorithm

In comparison with likely invariant discovery techniques, e.g., Daikon [8], our problem is more general and theoretically more complex. Checking whether an invariant holds on a dataset representing program states requires a linear traversal of the set of observed states, while ensuring that the invariant holds on each state. In our relational setting, however, we need to simultaneously consider multiple executions, which is why we delegate specification checking to a database engine. For instance, to falsify a loop invariant, e.g., $x > 0$, all we need is a single execution where x is negative; however, to falsify transitivity of a function, we need a set of 3 executions that together demonstrate that a function is not transitive. Section 7 makes a detailed comparison with other works.

Summary of contributions To summarize, the primary contributions of this paper are as follows:

- We formally define the relational specification learning problem as that of discovering likely specifications with respect to a dataset of input–output behaviors.
- We present Bach, an automated tool that learns relational specifications from input–output data of a library of functions. Bach utilizes a novel inductive and abductive synthesis technique to learn a rich class of specifications.

– We show that checking if a specification is consistent with the given data can be reduced to conjunctive queries, allowing us to use efficient databases or Datalog solvers to check specifications.
– We describe our implementation of Bach and present a thorough experimental evaluation on a range of libraries, demonstrating Bach's ability to learn a spectrum of useful specifications.

2 ILLUSTRATIVE EXAMPLES

In this section, we demonstrate the operation of Bach through a set of simple illustrative examples. Each function discussed has an associated set of observations in Figure 2.

E1: Properties of Addition

Suppose we have a function `add` that takes two numbers and returns their sum, and we have observed the input–output relation of `add` in Figure 2, where i_1 and i_2 are the inputs and r is the return value.

Induction phase The induction engine of Bach searches the space of specifications and proposes candidate specifications. Suppose that the induction phase proposes the following candidate:

$$\texttt{add}(x, y) = z \Leftrightarrow \texttt{add}(y, x) = z$$

where x, y, z are implicitly universally quantified variables.

Consistency verification Now, the specification goes to the consistency verifier, which checks if the specification is consistent with the provided dataset. The consistency verifier asks two questions:

+ve evidence Is there evidence that the specification holds?
-ve evidence Is there evidence that the specification *does not* hold?

To find positive evidence, the verifier attempts to find three constant values, a, b, and c, such that $\texttt{add}(a, b) = c$ and $\texttt{add}(b, a) = c$, as per the given dataset. The set of all possible values of (a, b, c) is considered the set of positive evidence. To characterize positive evidence, we view the set of input–output examples of `add` as a ternary relation $R_{\texttt{add}}(x, y, z)$, where x and y are the inputs and z is the corresponding output. Now, every possible tuple (a, b, c) that is evidence that the candidate specification holds is in the relation

$$R_{\texttt{add}}(x, y, z) \bowtie R_{\texttt{add}}(y, x, z)$$

where $\bowtie$ is the standard *join* operation from relational algebra. In the rest of the paper, we will use the formalism of Horn clauses (in non-recursive Datalog) to define positive evidence. Specifically, we will say that the set of positive evidence P is defined as follows:

$$P(X, Y, Z) \leftarrow R_{\texttt{add}}(X, Y, Z), R_{\texttt{add}}(Y, X, Z).$$

If the relation P is empty, then there is no positive evidence. Semantically, the above Horn clause defines P as the *smallest* relation such that if $(a, b, c) \in R_{\texttt{add}}$ and $(b, a, c) \in R_{\texttt{add}}$, then $(a, b, c) \in P$. In our example, we see that there is at least one tuple in P: $(3, 4, 7)$.

We now try to find negative evidence, i.e., tuples that falsify the specification. Since the specification is a *bi-implication*, we need to find evidence that holds for one side but not the other. Let us try to find a tuple that satisfies the left hand side but not the right hand side. We do this as follows:

$$N(X, Y, Z) \leftarrow R_{\texttt{add}}(X, Y, Z), R_{\texttt{add}}(Y, X, Z'), Z \neq Z'$$

If the relation N is not empty, then we know that there exists a tuple (a, b, c) such that $\texttt{add}(a, b) \neq \texttt{add}(b, a)$. In our example, the relation N is empty, and therefore Bach infers the specification stating that `add` is a commutative function.

add

i_1	i_2	r
1	2	3
3	4	7
5	6	11
4	3	7
⋮	⋮	⋮

gt

i_1	i_2	r
1	2	F
2	1	T
2	3	F
3	2	T
3	1	T
⋮	⋮	⋮

abs

i_1	r
1	1
2	2
-10	10
-3	3
⋮	⋮

concat

i_1	i_2	r
a	b	ab
a	ϵ	a
⋮	⋮	⋮

len

i_1	r
a	1
ϵ	0
b	1
ab	1
⋮	⋮

Figure 2: Example observed function executions.

Intuitively, our goal is to discover specifications with (*i*) no negative evidence associated with them (we say they are *consistent* with the data), and that (*ii*) have some positive evidence, which we use as a proxy to the likelihood of a specification.

E2: Transitivity of Comparison

Consider `gt`, the function implementing the greater-than operation for integers with data in Figure 2. The specification induction phase will propose the following specification:

$$(\texttt{gt}(x, y) = w \wedge \texttt{gt}(y, z) = w) \Rightarrow \texttt{gt}(x, z) = w$$

Note that this is an implication; Bach is able to infer both implications and bi-implications, as we shall describe in detail in Section 4. The consistency verifier will be able to find positive evidence and no negative evidence for this specification, thus declaring it a possible specification for `gt`. Specifically, it will solve the following two Horn clauses on the given dataset, and discover that the set P is non-empty, while N is empty.

$$P(\ldots) \leftarrow R_{\texttt{gt}}(X, Y, W), R_{\texttt{gt}}(Y, Z, W), R_{\texttt{gt}}(X, Z, W)$$
$$N(\ldots) \leftarrow R_{\texttt{gt}}(X, Y, W), R_{\texttt{gt}}(Y, Z, W), R_{\texttt{gt}}(X, Z, W'), W' \neq W$$

E3: Identity of Absolute Value

Consider the function `abs` (absolute value) with data in Figure 2.

Induction phase Suppose that the induction phase proposes the following candidate specification:

$$\texttt{abs}(x) = x$$

which can be viewed as the implication $\mathit{true} \Rightarrow \texttt{abs}(x) = x$

Consistency verification The consistency verifier will solve the following set of Horn clauses to discover positive and negative evidence, and store them in two relations $P(X)$ and $N(X)$:

$$P(X) \leftarrow R_{\texttt{abs}}(X, X)$$
$$N(X) \leftarrow R_{\texttt{abs}}(X, X'), X \neq X'$$

In this example, both relations will not be empty. Specifically, $P = \{1, 2, \ldots\}$ and $N = \{-10, -3, \ldots\}$.

Guard abduction The guard abduction phase attempts to *weaken* the specification by finding a formula G such that

$$G \Rightarrow \texttt{abs}(x) = x$$

has no negative evidence, and has all (or most) of the positive evidence. In other words, we can view the guard abduction phase as solving a *classification* problem, that of finding a classifier—a formula G—that labels elements of the set P with *true* and elements of the set N with *false*.

To find such a G, we assume we are given a set of predicates and functions with which we can construct G. For instance, if we are learning specifications of an API that operates over integers, we might instantiate our algorithm with standard operations over integers, e.g., $>, <, +, -$. In this example, the abduction phase might return the formula $x \geqslant 0$, resulting in the correct specification

$$x \geqslant 0 \Rightarrow \texttt{abs}(x) = x$$

There are many ways to approach such a classification task, e.g., using decision-tree learning. In practice, we employ a simple algorithm for learning a conjunction of predicates that separates positive and negative evidence.

E4: String Operations

Let us now consider an example with multiple functions. Suppose we are given a dataset (provided in Figure 2) describing the input–output relations of `concat`, which concatenates two strings, and `len`, which returns the length of a string. Here, ϵ is the empty string.

Suppose that the induction phase proposes the specification

$$\texttt{len}(\texttt{concat}(x, y)) = z \Leftrightarrow \texttt{len}(x) = z$$

Obviously, this is not true. However, using the positive and negative evidence, the guard abduction phase will discover that the specification holds when $y = \epsilon$, resulting in the following specification:

$$y = \epsilon \Rightarrow (\texttt{len}(\texttt{concat}(x, y)) = z \Leftrightarrow \texttt{len}(x) = z)$$

Additionally, Bach will infer other properties of `concat` and `len`, e.g., that the order of concatenation does not change the length of the resulting string.

$$\texttt{len}(\texttt{concat}(x, y)) = z \Leftrightarrow \texttt{len}(\texttt{concat}(y, x)) = z$$

We have illustrated the operation of Bach through a series of simple examples. In Sections 3-5, we formalize Bach. In Section 6, we thoroughly evaluate Bach on a range of Python libraries.

3 SPECIFICATIONS AND EVIDENCE

We now formalize the core definitions needed for our algorithm.

Formulas We assume formulas are over an interpreted theory, where we have a finite set of *uninterpreted functions* $\Sigma = \{f_1, \ldots, f_n\}$, where each f_i has arity $ar(f_i)$. A *formula* $\mathcal{F}$ is of the form

$$\forall V.\, G \Rightarrow (\Psi \Leftrightarrow \Phi) \quad \text{or} \quad \forall V.\, G \Rightarrow (\Psi \Rightarrow \Phi)$$

where (*i*) V is a set of variables. (*ii*) G, the *guard*, is a formula over an interpreted set of predicate and function symbols. (*iii*) Ψ (analogously, Φ) is defined as $\bigwedge_i \psi_i$, where each ψ_i is an *atom* of the form $t = o$, where o is a variable in V and t is a *nested* function application over the functions Σ and variables V. We assume that $\mathcal{F}$ has no free variables. For simplicity, and w.l.o.g., we assume that all variables and functions are over the same domain $\mathcal{D}$ (e.g., integers).

Example 3.1. Consider the following formula with $\Sigma = \{f, g\}$:

$$\forall x, y.\, \underbrace{x > 0}_{G} \Rightarrow \left(\underbrace{f(g(x)) = y}_{\Psi} \Leftrightarrow \underbrace{g(f(x)) = y}_{\Phi} \right)$$

Observe that G is a subformula using an interpreted predicate over integers, and each of Ψ and Φ are composed of a single atom containing nested uninterpreted function applications. ■

Interpretations and models An *interpretation* I gives a definition to each function $f \in \Sigma$; i.e., for each f and $\boldsymbol{i} \in \mathcal{D}^{ar(f)}$, I assigns a value $o \in \mathcal{D}$ such that $f(\boldsymbol{i}) = o$. For each $f \in \Sigma$, we use I_f to denote the definition of f in I:

$$I_f = \{\boldsymbol{i} \mapsto o \mid f(\boldsymbol{i}) = o\}$$

We will consider the interpretation I as $I = \bigcup_{f \in \Sigma} I_f$, a union of sets indexable by functions in Σ.

Given a formula $\mathcal{F}$, an interpretation I is a *model* of $\mathcal{F}$, denoted $I \models \mathcal{F}$, if I satisfies $\mathcal{F}$, using the standard definition of first-order satisfiability.

Datasets Intuitively, a *dataset* D in our setting is a *partial interpretation*, that is, an interpretation that defines each function $f \in \Sigma$ on a *finite* subset of the domain $\mathcal{D}^{ar(f)}$. Given a dataset D and interpretation I, we say I is a *completion* of D, denoted $D \subseteq I$, if for all $f \in \Sigma$, $D_f \subseteq I_f$.

Consistency Our goal is to define what it means for a formula $\mathcal{F}$ to explain a dataset D. We thus define a notion of *consistency*. A formula $\mathcal{F}$ is *inconsistent* with D, denoted $D \models_c \mathcal{F}$, if

$$\forall I \supseteq D.\, I \not\models \mathcal{F}$$

Otherwise, we say that D is *consistent* with $\mathcal{F}$, or $D \not\models_c \mathcal{F}$.

In other words, if no matter how we complete a dataset it results in an interpretation that falsifies $\mathcal{F}$, then we say that $\mathcal{F}$ is inconsistent with D. Otherwise, we say it is consistent.

Positive and negative evidence Note that while a formula $\mathcal{F}$ can be consistent with D, this could happen vacuously. The simplest case is the empty dataset, which is consistent with any satisfiable formula $\mathcal{F}$. Our goal is not only to find a consistent formula, but one that explains the data well. We therefore define the notions of *positive* and *negative* evidence. First, we define D-restricted assignments.

Definition 3.2 (D-restricted assignment). Given quantifier-free formula ϕ with variables V, a *D-restricted assignment* σ_D is a map from each $v \in V$ to a constant that appears in the dataset D. Additionally, for every term $f(\boldsymbol{i}) \in \sigma_D(\phi)$, $\boldsymbol{i} \mapsto o \in D_f$, for some $o \in \mathcal{D}$, where $\sigma_D(\phi)$ is ϕ with all variables replaced by their assignment in σ_D. ■

Example 3.3. Let $D_f = \{1 \mapsto 2\}$. $\sigma_D = \{x \mapsto 1\}$ is a D-restricted assignment to $f(x) = x$. This is because $\sigma_D(f(x) = x)$ is $f(1) = 1$, and 1 is in the domain of D_f. On the other hand, $\sigma'_D = \{x \mapsto 2\}$ is not a valid assignment, because f is not defined on 2 in D_f.

In case of nested terms, e.g., $f(g(x))$, a D-restricted assignment needs to set x to a value c in the domain of D_g such that $D_g(c)$ is in the domain of f. ■

Definition 3.4 (Positive evidence). Given D and $\mathcal{F}$, where $\mathcal{F}$ is of the form $\forall V.\, G \Rightarrow (\Psi \Leftrightarrow \Phi)$ or $\forall V.\, G \Rightarrow (\Psi \Rightarrow \Phi)$, we define *positive evidence* as:

$$pos(D, \mathcal{F}) = \{\sigma_D \mid \forall I \supseteq D.\, I \models \sigma_D(G \wedge \Psi \wedge \Phi)\}$$

Informally, positive evidence is the set of instantiations of variables V that non-vacuously satisfy the body of $\mathcal{F}$, i.e., $G \wedge \Psi \wedge \Phi$. ■

Definition 3.5 (Negative evidence). We define negative evidence as the set of D-restricted assignments such that

$$neg(D, \mathcal{F}) = \{\sigma_D \mid \forall I \supseteq D.\, I \models \sigma_D(G \wedge \neg(\Psi \Leftrightarrow \Phi))\}$$

or with $\sigma_D(G \wedge \neg(\Psi \Rightarrow \Phi))$ in the case $\mathcal{F}$ is an implication. Informally, we can view negative evidence as the set of *witnesses* to the fact that $\mathcal{F}$ is inconsistent with D. ■

Example 3.6. Consider formula $\mathcal{F} \triangleq \forall x, y.\, f(x) = y \Leftrightarrow g(x) = y$ and dataset D, where $D_f = \{1 \mapsto 1\}$ and $D_g = \{2 \mapsto 3\}$. Here, there is no negative evidence. However, there is no positive evidence either—i.e., there is no witness to the fact that f and g are equivalent. If we update D_g to $\{2 \mapsto 3, 1 \mapsto 1\}$, then $pos(D, \mathcal{F})$ will be the singleton set with the assignment that sets x to 1 and y to 1. Alternatively, if we update D_g to $\{2 \mapsto 3, 1 \mapsto 2\}$, then negative evidence will be the set of two assignments $\{\{x \mapsto 1, y \mapsto 1\}, \{x \mapsto 1, y \mapsto 2\}\}$. ■

The following lemma captures the fact that existence of negative evidence implies that the formula is inconsistent with the dataset.

Lemma 3.7. $neg(D, \mathcal{F}) \neq \emptyset \implies D \not\models_c \mathcal{F}$

The proof follows immediately from the definitions of consistency and negative evidence. We will see the utility of this lemma in our problem specification.

Specification learning problem Given a dataset D, $\mathcal{F}$ is a *likely specification* if $pos(D, \mathcal{F}) \neq 0$ and $neg(D, \mathcal{F}) = 0$. In other words, D supports the specification $\mathcal{F}$, and by the above lemma, we cannot show that $D \not\models_c \mathcal{F}$.

An *optimal specification* $\mathcal{F}^\star$ is a likely specification that maximizes some function h of D and $\mathcal{F}$. Formally,

$$\mathcal{F}^\star = \underset{\mathcal{F} \text{ st } neg(D,\mathcal{F})=\emptyset}{\arg\max} \; h(D, \mathcal{F})$$

An immediate choice for h is given by $h(D, \mathcal{F}) = |pos(D, \mathcal{F})|$. Intuitively, this uses the amount of positive evidence as a proxy for how well $\mathcal{F}$ explains D. Of course, h can also be adjusted to bias our search further if necessary, e.g., to formulas of smaller size.

4 SPECIFICATION LEARNING ALGORITHM

We are now ready to formalize Bach. The algorithm is shown in Figure 3 as a set of non-deterministic rules: if the premise above the horizontal line is true, then the instruction below the line is executed. At a high-level, the operation of Bach is simple: it (*i*) iteratively constructs specifications and (*ii*) checks whether they are consistent with the data.

The state maintained by Bach consists of two sets: (*i*) $\mathcal{J}$, a set of conjunctions of atoms, which are used to construct specifications; e.g., given $\Psi, \Phi \in \mathcal{J}$, we can construct $\forall V.\, \Psi \Leftrightarrow \Phi$. (*ii*) $\mathcal{S}$, a set of discovered likely specifications. Both sets grow monotonically.

Search We assume there is a fixed signature Σ, dataset D, and set of variables V. Recall that an atom is of the form $f(t_1, \ldots, t_n) = v$, where each t_i is a function application or a variable. The rules ADD, EXP_v, and EXP_f construct new atoms and conjoin them to formulas in the set $\mathcal{J}$. The symbol ● is used to denote a *wildcard*, a hole that can be filled to *complete* an atom. A conjunction $\Phi \in \mathcal{J}$ is *complete* if it contains no wildcards (denoted *cmp* in Figure 3).

Induction and abduction The rules $\text{IND}_\Leftrightarrow$ and $\text{ABD}_\Leftrightarrow$ form the core of the algorithm. They apply when a specification is learned, which they add to the set of likely specifications $\mathcal{S}$. The analogous rules $\text{IND}_\Rightarrow$ and $\text{ABD}_\Rightarrow$ learn specifications with implications (not shown in the figure due to similarity).

Let us walk through $\text{IND}_\Leftrightarrow$. It picks two conjunctions of complete atoms, Ψ and Φ, from the set $\mathcal{J}$. It then constructs a formula $\mathcal{F} = \forall V.\, \Psi \Leftrightarrow \Phi$. If $\mathcal{F}$ has positive but no negative evidence, then it is added to the set of specifications $\mathcal{S}$. For now, we use *pos* and *neg* declaratively; in Section 5, we present an algorithm that constructs the sets of evidences.

The rule $\text{ABD}_\Leftrightarrow$ applies when (*i*) $\mathcal{F}$ has non-empty sets of negative and positive evidence, and (*ii*) the two sets can be *separated*. We assume we have an oracle *classify* that returns a formula G that separates the positive and negative evidence. Formally, *classify* returns a formula G without uninterpreted functions and with free variables in V, such that:

(1) $\forall \sigma \in neg(D, \mathcal{F}).\, \sigma(G)$ is unsatisfiable.
(2) $\exists X \subseteq pos(D, \mathcal{F}).\, X \neq \emptyset \wedge \forall \sigma \in X.\, \sigma(G)$ is satisfiable.

In other words, G eliminates all negative evidence (point 1), and maintains *some* of the positive evidence (point 2). Observe that we do not need G to maintain *all* positive evidence—we only need a non-empty set, and that gives us a likely specification.

Soundness We view the soundness of Bach as only adding likely specifications to the set $\mathcal{S}$. This is maintained by construction through (*i*) the rules $\text{IND}_\Leftrightarrow$ and $\text{ABD}_\Leftrightarrow$, (*ii*) and the definition of *classify*, which ensures that all negative evidence is excised and some positive evidence is preserved.

Rule-application schedule Our presentation of the algorithm as a set of rules allows us to dictate the search order by varying the scheduling of rule application. For instance, if we are interested in learning relations between pairs of programs, we can restrict applications of the rule ADD to formulas Φ that are *true*. This ensures that there is only a single conjunct on either side of the (bi-)implication.

We must also decide when to apply $\text{IND}_\Leftrightarrow$ and $\text{ABD}_\Leftrightarrow$. In practice, we apply $\text{ABD}_\Leftrightarrow$ right after a failed application of an induction rule. Specifically, if a failed application of $\text{IND}_\Leftrightarrow$ results in positive evidence *and* negative evidence, then we apply $\text{ABD}_\Leftrightarrow$ with the hope that we can find a guard that eliminates the negative evidence.

5 CONSISTENCY VERIFICATION

We now describe our technique for verifying consistency of a formula $\mathcal{F}$ with respect to a dataset D.

5.1 Background and Overview

The principal idea underlying our technique is that positive and negative evidence of a formula $\mathcal{F}$ and dataset D can be characterized using a *union of conjunctive queries* (UCQ) [1]. A *conjunctive query* (CQ) is a first-order logic query that can model a subset of database queries written in SQL—specifically, a conjunctive query corresponds to a non-recursive Horn clause. Therefore, a UCQ corresponds to a non-recursive *Datalog* program—a set of Horn clauses—whose evaluation results in the positive and negative evidence. Our formulation of consistency verification as database query evaluation allows us to leverage efficient, highly engineered database engines and Datalog solvers.

$$\frac{}{\mathcal{J} \leftarrow \{true\} \quad \mathcal{S} \leftarrow \emptyset}\ \textsc{init}$$

$$\frac{f \in \Sigma \qquad \Phi \in \mathcal{J} \qquad \Phi' = \Phi \wedge f(\bullet_1 \ldots \bullet_{ar(f)}) = \bullet}{\mathcal{J} \leftarrow \mathcal{J} \oplus \Phi'}\ \textsc{add}$$

$$\frac{\Phi \in \mathcal{J} \quad \bullet \in \Phi \quad v \in V}{\mathcal{J} \leftarrow \mathcal{J} \oplus \Phi[\bullet \mapsto v]}\ \textsc{exp}_v$$

$$\frac{\Phi \in \mathcal{J} \quad \bullet \in \Phi \quad f \in \Sigma}{\mathcal{J} \leftarrow \mathcal{J} \oplus \Phi[\bullet \mapsto f(\bullet_1, \ldots, \bullet_{ar(f)})]}\ \textsc{exp}_f$$

$$\frac{\Psi, \Phi \in cmp(\mathcal{J}) \qquad \mathcal{F} = \forall V.\, \Psi \Leftrightarrow \Phi \qquad pos(D, \mathcal{F}) \neq \emptyset \qquad neg(D, \mathcal{F}) = \emptyset}{\mathcal{S} \leftarrow \mathcal{S} \oplus \mathcal{F}}\ \textsc{ind}_\Leftrightarrow$$

$$\frac{\Psi, \Phi \in cmp(\mathcal{J}) \qquad \mathcal{F} = \forall V.\, \Psi \Leftrightarrow \Phi \qquad P = pos(D, \mathcal{F}) \neq \emptyset \qquad N = neg(D, \mathcal{F}) \neq \emptyset \qquad G = classify(P, N)}{\mathcal{S} \leftarrow \mathcal{S} \oplus \forall V.\, G \Rightarrow (\Psi \Leftrightarrow \Phi)}\ \textsc{abd}_\Leftrightarrow$$

Notes: (*i*) $S \oplus x$ is short for $S \cup \{x\}$, (*ii*) $\{\bullet_1 \ldots \bullet_{ar(f)}, \bullet\}$ in ADD and $\textsc{exp}_f$ are fresh, and (*iii*) $\bullet$ in $\textsc{exp}_f$ is assumed to be an argument to a function

Figure 3: Bach's main algorithm (the rule INIT is only applied at initialization)

We provide a brief description of Datalog and refer the reader to Abiteboul et al.'s textbook for a formal presentation of Datalog semantics [1]. A Horn clause is of the form:

$$H(\boldsymbol{X}_0) \leftarrow B_1(\boldsymbol{X}_1), \ldots, B_n(\boldsymbol{X}_n).$$

where $H, B_1, \ldots, B_n$ are relation symbols; each $\boldsymbol{X}_i$ is a vector of variables of size equal to the arity of the corresponding relation; the *atom* $H(\boldsymbol{X}_0)$ is the *head* of the clause; and the *set of atoms* $\{B_i(\boldsymbol{X}_i)\}_i$ is the *body* of the clause. A Datalog program C is a set of Horn clauses. Semantically, a solution of a Datalog program is the least interpretation of the relations that satisfies all the clauses. For our purposes, we will enrich our language with inequalities of the form $X \neq Y$, where X and Y are variables, which can appear in the bodies of clauses. We will use underscores, e.g., $R(X, \_)$, to denote that the second argument of R is *unbound*, i.e., can take any value.

5.2 Detailed Description

Figure 4 describes the algorithm used to construct a set of Horn clauses encoding the positive/negative evidence of $\mathcal{F}$ with respect to D. We assume $\mathcal{F}$ is of the form $\forall \boldsymbol{x}.\, \Psi \Leftrightarrow \Phi$, where $\Psi = \bigwedge_i \psi_i$, $\Phi = \bigwedge_j \phi_j$, each ψ_i (and ϕ_j) is an atom of the form $f(t_1, \ldots, t_n) = x$, and $\boldsymbol{x}$ is the vector of universally quantified variables.

We assume that input–output data of each n-ary function f is stored in a $(n+1)$-ary relation R_f. The Horn-clause construction decomposes into three steps:

(1) Ψ is encoded in a relation $A_H(\mathbf{X}^a)$ (and Φ in $B_H(\mathbf{X}^b)$) by *flattening* the atoms;
(2) positive evidence is encoded in the relation $P(\mathbf{X})$ by collecting variable assignments satisfying Ψ and Φ; and
(3) negative evidence is encoded in the relation $N(\mathbf{X})$ by satisfying Ψ (Φ) while negating Φ (Ψ).

Procedure `Encode` aggregates all generated Horn clauses into a single Datalog program C. Note that we expect the formula $\mathcal{F}$ to be a bi-implication. If $\mathcal{F}$ is of the form $\forall \boldsymbol{x}.\, \Psi \Rightarrow \Phi$, we construct negative evidence by only considering data satisfying Ψ and $\neg\Phi$.

Given a vector of variables $\boldsymbol{x}$ appearing in a formula $\mathcal{F}$, we will construct a vector of Datalog variables $\boldsymbol{X}$ indexed by $x \in \boldsymbol{x}$ (i.e., $x \in \boldsymbol{x}$ implies $X_x \in \boldsymbol{X}$). We use $H(\boldsymbol{X}) \leftarrow S$, where S is the set of terms $\{R_i(\boldsymbol{X}_i)\}_{i=1}^n$, to denote the Horn clause $H(\boldsymbol{X}) \leftarrow R_1(\boldsymbol{X}_1), R_2(\boldsymbol{X}_2), \ldots, R_n(\boldsymbol{X}_n)$. In addition, we use $\oplus$ for adding a single element to a set. For example, $x \oplus \{y, z\} = \{x, y, z\}$.

Encoding specifications Let us walk through the construction of Horn clauses encoding Ψ and Φ. We focus on Ψ, as the encoding for Φ is symmetric. By assumption, the conjunction Ψ consists of atoms $\{t_i^a = x_i^a\}$, where t_i^a is a term of the form $f(t_1, \ldots, t_n)$; we extract those atoms using the `atoms` subroutine. In the first for-loop of `Encode`, we iterate over every atom and encode it as a Horn clause. The atom $t_i^a = x_i^a$ is encoded in the relation $A_i(\mathbf{X}_i^a, X_i^a)$, where $\mathbf{X}_i^a$ represents the inputs to the term t_i^a and X_i^a represents the output (in this case, the value of the formula variable x_i^a). Because each atom can have nested function applications, this procedure is recursive, and so we make use of the subroutine `Flatten`. Finally, we encode Ψ as the conjunction of each atom, which translates into the Horn clause $A_H(\mathbf{X}^a) \leftarrow \left\{A_i(\mathbf{X}_i^a, X_i^a)\right\}_{i=1}^n$.

Example 5.1. We now demonstrate Horn clause construction on a simple example. Consider the following formula:

$$\forall x, y.\, f(g(x)) = y \Leftrightarrow h(x) = y$$

which states that $f \circ g$ is equivalent to h. Encoding the atom $f(g(x)) = y$ requires three recursive calls to `Flatten` (once for x, $g(x)$, and $f(g(x))$). Working from the inside out, we see `Flatten(`x`)` converts the term x to the pair $\emptyset, X_x$. The call to `Flatten(`$g(x)$`)` uses this pair to construct the pair $\left\{R_g(X_x, O)\right\}, O$. Finally, `Flatten(`$f(g(x))$`)` expands on this value to return the pair $\left\{R_f(O, O'), R_g(X_x, O)\right\}, O'$. The first for-loop in `Encode` uses these results to tie the formula variable y to the output of the term $f(g(x))$ by constructing the clause:

$$A_1(X_x, X_y) \leftarrow O' = X_y, R_f(O, O'), R_g(X_x, O).$$

The right-hand side of $\Leftrightarrow$ is encoded similarly. ■

Positive evidence Positive evidence consists exactly of those variable assignments that non-trivially satisfy both Ψ and Φ. As Ψ and Φ are already fully encoded in the relations $A_H(\mathbf{X}^a)$ and $B_H(\mathbf{X}^b)$, this requirement is immediately encodable as the Horn clause $P(\mathbf{X}) \leftarrow A_H(\mathbf{X}^a), B_H(\mathbf{X}^b)$.

Negative evidence Let us now describe the construction of the Horn clauses encoding negative evidence. Intuitively, negative evidence occurs when we satisfy the left side Ψ but falsify the right side Φ. In other words, we want to falsify *at least one* of the atoms $\phi_1, \ldots, \phi_m$. We thus construct m clauses, each one encoding assignments that falsify one of the ϕ_j's. For instance, assignments that

def Flatten(t : *term*):
  case t *is* x, *where* $x \in \boldsymbol{x}$:
    return $\emptyset, X_x$
  case t *is* $f(t_1, \ldots, t_n)$:
    for $i \in 1, \ldots, n$:
      $R_i, O_i \leftarrow$ Flatten(t_i)
    O = *fresh variable*
    $R = R_f(O_1, \ldots, O_n, O) \oplus \bigcup_{i=1}^{n} R_i$
    return R, O

def Encode($\Psi \Leftrightarrow \Phi$: *spec*):
  $C = \emptyset$
  $\left\{t_i^a = x_i^a\right\}_{i=1}^{n} =$ atoms(Ψ)
  $\left\{t_j^b = x_j^b\right\}_{j=1}^{m} =$ atoms(Φ)
  # encode Ψ
  for $i \in 1, \ldots, n$:
    $R_i^a, O_i^a =$ Flatten(t_i^a)
    $\mathbf{X}_i^a =$ vars(R_i^a)
    $C = C \oplus A_i(\mathbf{X}_i^a, X_{x_i^a}) \leftarrow R_i^a \oplus (O_i^a = X_{x_i^a})$
  $C = C \oplus A_H(\mathbf{X}^a) \leftarrow \left\{A_i(\mathbf{X}_i^a, X_i^a)\right\}_{i=1}^{n}$
  # encode Φ
  for $j \in 1, \ldots, m$:
    # ...omitted...
  $C = C \oplus B_H(\mathbf{X}^b) \leftarrow \left\{B_j(\mathbf{X}_j^b, X_j^b)\right\}_{j=1}^{m}$
  # encode positive evidence
  $C = C \oplus P(\mathbf{X}) \leftarrow A_H(\mathbf{X}^a), B_H(\mathbf{X}^b)$
  # encode left negative evidence
  for $j \in 1, \ldots, m$:
    O = *fresh variable*
    $\text{bad} = \left\{B_j(\mathbf{X}_j^b, O), (O \neq X_j^b)\right\}$
    $C = C \oplus N(\mathbf{X}) \leftarrow \{A_H(\mathbf{X}^a)\} \cup \text{bad} \cup \left\{B_i(\mathbf{X}_i^b, \_)\right\}_{i \neq j}$
  # encode right negative evidence
  for $i \in 1, \ldots, n$:
    # ...omitted...
  return C

Figure 4: Encoding formulas as Horn clauses. Omitted for-loops are symmetric (by exchanging A and B) to those immediately preceding. We use = for assignment and $\leftarrow$ for Datalog implication.

falsify ϕ_1 are encoded by the clause:

$$N(\mathbf{X}) \leftarrow A_H(\mathbf{X}^a), B_1(\mathbf{X}_1^b, O), O \neq X_1^b, B_2(\mathbf{X}_2^b, \_), \ldots, B_m(\mathbf{X}_m^b, \_)$$

The fresh variable O is used to encode the fact that B_1 should output a value that is not equal to X_1^b, thus falsifying the right-hand side of the bi-implication. Recall that B_1 encodes a term of the form $f(\ldots) = x$. Effectively, the above clause states that $f(\ldots) \neq x$.

Example 5.2. Recall Example 5.1. Negative evidence, as constructed by Encode, is written as follows:

$$N(X_x, X_y) \leftarrow A_H(X_x, X_y), B_1(X_x, O), O \neq X_y$$
$$N(X_x, X_y) \leftarrow B_H(X_x, X_y), A_1(X_x, O), O \neq X_y$$

The first clause encodes the requirement that $f(g(x)) = y$ is satisfied, but $h(x) = y$ is not; the second clause encodes the opposite fact. ■

Correctness Once we have constructed the Horn clauses, we evaluate them on the given dataset to construct the relations P and N. The following theorem states correctness of the construction:

Theorem 5.3. *Given a dataset D and specification $\mathcal{F}$ of the form $\forall \boldsymbol{x}. \Psi \Leftrightarrow \Phi$ or $\forall \boldsymbol{x}. \Psi \Rightarrow \Phi$, let $N(\boldsymbol{X})$ and $P(\boldsymbol{X})$ be the relations computed using the Horn clauses C from Figure 4. Then, $P(\boldsymbol{X}) = pos(D, \mathcal{F})$ and $N(\boldsymbol{X}) = neg(D, \mathcal{F})$.*

Complexity It is important to note that the decision problem of solving a conjunctive query is NP-complete (*combined complexity*). If the size of the query is fixed and the only variable is the size of the data, the problem is in PTIME (*data complexity*) [1]. This is why database engines are efficient: queries are typically small, but data is large. These classic results shed light on the difficulty of the problem of finding positive/negative evidence: One could easily reduce conjunctive query solving to finding positive evidence in our setting, thus our consistency verification problem is NP-hard.

6 IMPLEMENTATION AND EVALUATION

In this section, we (*i*) describe our implementation of Bach, (*ii*) present an exploratory study in which we apply Bach to a number of libraries, and (*iii*) present an empirical evaluation to investigate the performance and precision characteristics of Bach.

6.1 Implementation

Bach is implemented in OCaml. It takes as input (*i*) a signature of simply typed functions, (*ii*) input–output data for each function, and (*iii*) a set of predicates to compute the guards. Bach uses the Soufflé Datalog engine [14] to compute positive/negative evidence.

Ordering the search The search rules ADD, EXP$_v$, and EXP$_f$ are scheduled to implement a *frontier search* with respect to the size of specifications. That is, we visit specifications in order from smallest to largest. The search rules are augmented with types, ensuring that only well-typed specifications are explored.

Pruning the search Top-down enumerative synthesis tools typically have exponential branching of the search space, and Bach is no exception. To combat this explosion of the search space, Bach employs a series of search-space pruning techniques: First, Bach uses a representation of specifications that guarantees that each explored specification is unique with respect to conjunct reordering (by commutativity of conjunction) and variable renaming. Second, whenever Bach proves a specification $\mathcal{F} = \Psi \Leftrightarrow \Phi$ correct, it records one of Ψ and Φ (the larger with respect to number and size of atoms, if it is obvious). During the search, Bach will never apply the search rules to generate the recorded term.

Specification preference Bach combines induction and abduction rule application as follows: Given two sets of conjunctions, $\Psi, \Phi \in \mathcal{J}$, it first attempts to learn the bi-implication $\Psi \Leftrightarrow \Phi$, using the IND$_\Leftrightarrow$ rule. If IND$_\Leftrightarrow$ fails to apply due to existence of negative evidence, then Bach examines the negative evidence to determine if it is only *one-sided*. If so, then Bach learns an implication $\Psi \Rightarrow \Phi$, using the rule IND$_\Rightarrow$. If no implication can be learned, Bach resorts to abduction. Specifically, it solves a number of abduction problems to learn guards that make the following specifications likely ones:

$$G_1 \Rightarrow (\Phi \Leftrightarrow \Psi), \quad G_2 \Rightarrow (\Phi \Rightarrow \Psi), \quad G_3 \Rightarrow (\Psi \Rightarrow \Phi)$$

Table 1: List of benchmarks; number of functions is in parentheses.

Benchmark	Description
`list` (7)	standard list operations, including `hd`, `tl`, `cons`, etc.
`matrix` (7)	matrix operations from Python's `sympy` library.
`trig` (4)	trig. functions (`sin`, `cos`, etc.) in Python's `math` module.
`z3` (5)	API to Python's `z3` library, including `sat` and `valid`.
`geometry` (5)	manipulations of shapes from Python's `sympy` library.
`sets` (10)	functions from Python's `set` module.
`dict` (5)	functions from Python's `dict` (dictionary) module.
`fp199` (4)	arithmetic on $\mathbb{F}_{199}$, the finite field of order 199.
`strings` (9)	string operations from Python's `string` module.

Bach then picks the specification with the highest positive evidence.

Abduction Guard abduction is done by a simple classification algorithm that finds a small conjunction of the provided predicates. Each predicate is instantiated with every combination of variables. For instance, if the predicate $a > b$ is provided, and $\mathcal{F}$ contains the variables x and y, abduction will use $x > y$ and $y > x$. Bach learns a conjunction that *separates* the positive and negative evidence of $\mathcal{F}$ while *retaining* as much positive evidence as possible.

6.2 Exploratory Evaluation

Setup In order to test the efficacy of Bach, we targeted a set of 9 Python libraries (Table 1). Each benchmark consists of (*i*) a finite signature, (*ii*) a set of predicates, and (*iii*) a dataset of 1000 randomly sampled executions for each function. These random samples are generated by uniformly sampling function inputs from a subdomain of the relevant type and then evaluating the function.

We are interested in examining a variety of specifications. To cover as much of the search space as possible, we run many independent executions of Bach in parallel. Each execution is configured to search over a different subset of functions from the signature, or at a different initial depth. This gives a mix of large and small likely specifications with a variety of combinations of functions.

After letting each execution run for a short amount of time (1-2 minutes), all the resultant likely specifications are collected and presented together. A partial list of specifications found is provided in Figure 5. The output of Bach contains many specifications that are possibly of interest, a few of which are discussed below.

`z3` specifications `z3` is a high-performance SMT solver with APIs for many programming languages. The `z3` benchmark contains functions from a subset of Python's `z3` API. Bach finds the expected specifications relating `and`, `or`, and `neg` through DeMorgan's laws, distributivity, etc. However, the benchmark also contains `valid` and `sat`, which check for the validity or satisfiability of a formula. Consequently, Bach discovers the specification

$$p = \mathit{true} \Rightarrow (\texttt{valid}(x) = p \Rightarrow \texttt{sat}(x) = p),$$

which states that valid formulas are always satisfiable (but not the opposite). Bach also finds interactions between `valid` and the logical connectives. For example,

$$\texttt{valid}(x) = p \wedge \texttt{valid}(y) = p \Rightarrow \texttt{valid}(\texttt{and}(x, y)) = p,$$

which encodes the fact that validity is preserved by `and`.

`strings` specifications The `strings` benchmark contains the typical set of functions for manipulating strings. Bach finds likely specifications which encode idempotence properties, such as

$$\texttt{lstrip}(x) = y \Rightarrow \texttt{lstrip}(y) = y,$$

Learned specifications for `list`

sorting a list preserves length
$\texttt{length}(x) = a \Leftrightarrow \texttt{length}(\texttt{sort}(x)) = a$

`hd` is the destructor of `cons`
$\texttt{cons}(a, y) = x \Rightarrow \texttt{hd}(x) = a$

`rev` is an involution
$\mathit{true} \Leftrightarrow \texttt{rev}(\texttt{rev}(x)) = x$

Learned specifications for `matrix`

identity matrix is upper triangular
$\texttt{identity}(x) \wedge p = \mathit{true} \Rightarrow (\texttt{upper}(x) = p \Leftrightarrow \mathit{true})$

`transpose` preserves symmetric-ness
$\texttt{symmetric}(x) = p \Leftrightarrow \texttt{symmetric}(\texttt{transpose}(x)) = p$

`transpose` is an involution
$\texttt{transpose}(\texttt{transpose}(x)) = x$

Learned specifications for `trig`

`sin` is the inverse of `arcsin`
$\texttt{arcsin}(z) = x \Rightarrow \texttt{sin}(x) = z$

`sin` has period 2π
$\exists k.x = 2\pi k + y \Rightarrow (\texttt{sin}(x) = z \Leftrightarrow \texttt{sin}(y) = z)$

`sin` and `cos` are shifted by $\pi/2$
$x = y - \pi/2 \Rightarrow (\texttt{sin}(x) = z \Leftrightarrow \texttt{cos}(y) = z)$

Learned specifications for `strings`

a string is a prefix of itself
$p = \mathit{true} \Rightarrow \texttt{prefix}(x, x) = p$

stripping whitespace is idempotent
$\texttt{lstrip}(x) = y \Rightarrow \texttt{lstrip}(y) = y$

palindromes are preserved by `reverse`
$\texttt{concat}(y, \texttt{reverse}(y)) = x \Rightarrow \texttt{reverse}(x) = x$

Learned specifications for `z3`

validity implies satisfiability
$p = \mathit{true} \Rightarrow (\texttt{valid}(x) = p \Rightarrow \texttt{sat}(x) = p)$

`and` is commutative
$\texttt{and}(x, y) = z \Leftrightarrow \texttt{and}(y, x) = z$

`and` preserves validity
$\texttt{valid}(x) = p \wedge \texttt{valid}(y) = p \Rightarrow \texttt{valid}(\texttt{and}(x, y)) = p$

Learned specifications for `sets`

the empty set contains nothing
$p = \mathit{false} \Rightarrow (\texttt{clear}(x) = y \Rightarrow \texttt{contains}(y, a) = p)$

the empty set is contained in every set
$p = \mathit{true} \Rightarrow (\texttt{clear}(x) = y \Rightarrow \texttt{subset}(y, z) = p)$

the subset relation is inclusive
$p = \mathit{true} \Rightarrow \texttt{subset}(x, x) = p$

Learned specifications for `geometry`

if enclosed shape contains point, then encloser also contains it
$b = \mathit{true} \Rightarrow (\texttt{encl}(x, y) = b \wedge \texttt{encl\_pt}(y, p) = b \Rightarrow \texttt{encl\_pt}(x, p) = b)$

rotating a shape by a multiple of 2π results in same shape
$\exists k.x = 2\pi k \Rightarrow \texttt{rotate}(y, x) = y$

Figure 5: A sample of learned specifications on our benchmark suite. Formulas $\exists k.x = 2\pi k + y$, $\exists k.x = 2\pi k$, $x = y - \pi/2$, and $p = \mathit{true}, \mathit{false}$ are guards.

Table 2: Average correctness results (T_1 is the type I error, T_2 is the type II error, and Size is the number of specifications produced)

	10 observations			50 observations			100 observations			500 observations		
Benchmark	T_1	T_2	Size	T_1	T_2	Size	T_1	T_2	Size	T_1	T_2	Size
`ff199`	1.8	17.6	4.2	5.6	13.2	12.4	6.4	10.2	16.2	6.4	6.2	20.2
`trig`	2	19.2	10.8	2	0.8	29.2	0	0	28	0	0	28
`dict`	5.2	0.2	20	1.4	0	16.4	1	0	16	1	0	16
`geometry`	18	12	25	9	3.8	24.2	4	1	22	1	0	20
`lists`	40	17.6	52.4	4.8	0.4	34.4	1.4	0	31.4	0.4	0	30.4
`matrices`	18.2	11.2	25	15.4	3.2	30.2	7.8	0.6	25.2	5.2	0	19.4
`sets`	52.8	53.4	79.4	6.2	3.8	82.4	0.4	0	80.4	0	0	80
`strings`	159.6	234	215.6	85.6	50.8	324.8	22.2	1.6	310.6	0.2	0	290.2

where $\texttt{lstrip}(x)$ removes all whitespace on the left of x, as well as useful facts like

$$p = \mathit{true} \Rightarrow (\texttt{prefix}(x, x) = p),$$

which states that string prefix is a reflexive relation. *Amusingly, Bach also learns that we can construct palindromes by concatenating a string and its reverse:*

$$\texttt{concat}(y, \texttt{reverse}(y)) = x \Rightarrow \texttt{reverse}(x) = x.$$

`trig` specifications The `trig` benchmark contains trigonometric functions (from Python's `math` module), which have a rich set of semantics. Bach has no problem finding many of these properties as likely specifications. These include the fact that trigonometric functions are periodic,

$$\exists k.x = 2\pi k + y \Rightarrow (\texttt{sin}(x) = z \Leftrightarrow \texttt{sin}(y) = z),$$

where $\exists k.x = 2\pi k + y$ is provided as a predicate on x and y. Bach also discovers that `sin` and `arcsin` are *almost* inverses of each other:

$$\texttt{arcsin}(z) = x \Rightarrow \texttt{sin}(x) = z.$$

Note the implication. This is because `arcsin` is sometimes undefined, and so `sin` and `arcsin` are only inverses on the range of `arcsin` ($[-\pi/2, \pi/2]$).

`geometry` specifications The `geometry` benchmark contains functions from `sympy`'s [23] (a popular Python library) `geometry` module, which supplies operations over 2D shapes on a plane. Bach learns the following specification:

$$b = \mathit{true} \Rightarrow$$
$$(\texttt{encl}(x, y) = b \wedge \texttt{encl\_pt}(y, p) = b \Rightarrow \texttt{encl\_pt}(x, p) = b)$$

which states that if (*i*) 2D shape x encloses shape y, and (*ii*) point p is in shape y, then p is in shape x.

Another insightful property that Bach detects is that rotating a shape by a multiple of 2π results in the shape itself:

$$\exists k.x = 2\pi k \Rightarrow \texttt{rotate}(y, x) = y$$

Finding interesting specifications In order to extract the previous specifications (and those in Figure 5), we rank the output of Bach in decreasing order by a function $h(D, \mathcal{F}) = \langle |\mathcal{F}|^{-1}, pos(D, \mathcal{F})\rangle$, where comparison is done lexicographically and $|\cdot|$ is computed by counting AST nodes. Optimal specifications, in this context, are those that are small yet have large amounts of positive evidence.

While this ranking function worked to produce a variety of interesting specifications, it also obscured a few that we expected to see ranked more highly; associativity of matrix multiplication did not show up until late in the list. Finding improved ranking functions for various domains and tasks is an area of future research.

6.3 Empirical Evaluation

We now investigate (*i*) the scalability of Bach and (*ii*) the significance of Bach's learned specifications.

Scalability The Horn clauses for negative evidence can, in some cases, result in a polynomial increase in the size of the relations. To evaluate the impact of this behavior, we measure the number of checked specifications per second (i.e., calls to Datalog solver).

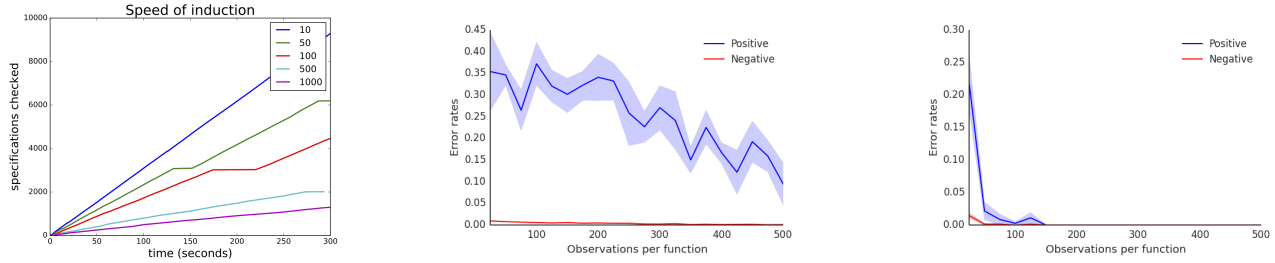
Search performance is dependent more on the structure of the formula and the amount of data than any inherent semantic meaning of the library functions. As such, we test scalability on a representative benchmark, in this case `ff199`. For $k = 10, 50, 100, 500$, and 1000, we sample k observations for each function to construct the dataset D_k. We run Bach for 5 minutes on D_k, and report the number of checked specifications at every point in time. The results are presented in Figure 6(a).

The results are as anticipated: with more data, Bach checks less specifications in the same amount of time. The best-performing benchmark, $k = 10$, checks ≈9x more specifications than the worst-performing benchmark, $k = 1000$. Of note are the plateaus in the $k > 10$ results, which indicate Soufflé getting slowed down with queries with large output. These plateaus suggest that too much data can overwhelm the external Datalog solver to the point of losing performance. In the future, we would like to experiment with *approximate queries*, where we sample a subset of the data with the goal of falsifying a query, before trying the full dataset.

Error analysis To evaluate correctness of Bach, we need to determine how often Bach is wrong. We proceed by fixing a notion of *ground truth* and computing type I/II error. Type I error is Bach presenting an incorrect specification (false positive), while type II error is Bach failing to present a correct specification (false negative).

Accurately determining ground truth for the domains Bach operates on requires enumerating all possible hypotheses and asking a human expert or an automated verifier to label them as true or false. This is an infeasible: (*i*) there are infinitely many possible hypotheses, as our formulas are not size-bounded, and (*ii*) even if we bound the size of formulas, there are exponentially many specifications to consider. Using an automated verifier is a possibility, but state-of-the-art checking of relational specifications is limited to simple programs and properties [22]. In our setting, we are dealing with non-trivial, dynamic Python code.

To evaluate error rates, we conducted an experiment where we *approximate* ground truth by running Bach on a large—1000 observed executions per function—dataset per benchmark, and ensured Bach checked every formula up to size 7 (measured by AST leaves). We chose 1000 observations to generate ground truth because (*i*) we observed that the number of discovered specifications



(a) (b) (c)

Figure 6: (a) # of specifications per unit time for `ff199`. (b) Error analysis for `ff199` (band is 95% confidence interval). (c) Error analysis for `sets`.

stabilizes at $\geqslant$ 1000, and (*ii*) at 1000 observations, the number of type I errors is very low. For point (*ii*), we corroborated the accuracy of our approximate ground truth by randomly sampling 30 specifications from those discovered for each benchmark with 1000 observations, and manually classifying them as correct/incorrect specifications. Manual inspection showed that ≈100% of the sampled specifications were correct.

To understand how well Bach performs on varying amounts of data, for $k = 10, 50, 100, 500$, we randomly sample k observations per function to construct a dataset D_k that is independent from the approximate ground truth. We run Bach on the same search space ground truth was generated on (formulas up to size 7) and compute type I and type II with respect to ground truth. For each k we repeat this process multiple times, each time randomly sampling a different dataset. See Table 2 for the results.

For every benchmark, type I/II error tends to decrease as we increase the amount of data. By 100 observations, type II errors have nearly disappeared for every benchmark except `ff199`. By 500 observations, the number of type I errors has also fallen dramatically. For example, in `strings`, which has over 290 average specifications produced, Bach generates on average 0.2 type I errors.

To provide a clearer picture, for representative best- and worst-case benchmarks (`ff199` and `sets`, respectively), we also compute error rates at each $k = 25, 50, 75, \ldots, 500$. The results are presented in Figure 6(b,c). Here, false positive error rate is **FP** / (**TP** + **FP**), where **FP** and **TP** are the number of false and true positives. Negative error rate is defined symmetrically. By 500 observations, our worst-performing benchmark, `ff199`, has a positive error rate of ≈0.1, meaning we expect Bach to discover an incorrect specification 10% of the time. Conversely, our best-performing benchmark, `sets`, has converged to ground truth by 150 observations.

These results indicate that, for most benchmarks, Bach can achieve reasonable results before the decrease in performance found in our scalability experiments becomes prohibitive. The exceptions to are `ff199` and `matrices`, which are both numeric benchmarks. This is due to the difficulty of sampling corner cases (e.g., 0, or non-invertible matrices) with low numbers of observations.

7 RELATED WORK

Specification inference In the introduction, we compared Bach with QuickSpec [5] and Daikon [8]. The work of Henkel et al. [11] for documenting Java container classes is also very closely related to QuickSpec, and has the same comparison with Bach.

A number of specification learning techniques use positive and negative examples to learn *safe preconditions* for calling a function [10, 18, 20, 21]. For example, Padhi et al.'s PIE [18] and Sankaranarayanan et al.'s work [20] use input–output data to learn preconditions that ensure a given postcondition is satisfied for a single function. These approaches synthesize a Boolean formula over a fixed set of predicates; PIE can additionally infer new predicates by searching over a given grammar. Gehr et al. [10] use positive and negative examples to synthesize a precondition that ensures that two function calls commute, with the goal of discovering safe parallel execution contexts. Bach discovers specifications that correlate executions of multiple functions, and does not require annotated positive/negative examples. Bach's abduction engine solves a Boolean classification task, like the aforementioned works. Thus, it can technically be instrumented for inferring safe preconditions.

A number of other techniques aim to learn *temporal* specifications (e.g., [2–4, 9, 12, 13, 15, 24, 25]), which specify acceptable sequences of events, e.g., calls to an API.

ILP *Inductive logic programming* (ILP) [17] is a machine learning technique that infers Horn clauses to logically *classify* a set of positive and negative examples. More than twenty years ago, Cohen [7] used ILP to infer specifications by observing behaviors of a switching system. Sankaranarayanan et al. used ILP [21] to infer Horn clauses that explain when exceptions are thrown in various data structure implementations. ILP techniques, like FOIL [19] and Progol [16], are optimized for learning Horn clauses and tend to sacrifice *correctness* (full classification precision) for scalability. Bach, on the other hand, does not require annotated examples—i.e., it is unsupervised—and can, in principle, discover Horn clause specifications, in addition to arbitrary (bi-)implications.

8 CONCLUSION

We presented Bach, an automated technique for learning relational specifications from input–output data. Our evaluation demonstrated Bach's ability to learn interesting specifications of real-world libraries. There are many potential uses of Bach, which we plan on investigating in the future: it could be used to detect axioms useful for verification of applications using library code, for lemma discovery, e.g., in interactive theorem provers like Coq, or to automatically annotate library documentation with specifications.

ACKNOWLEDGEMENTS

This work is supported by NSF awards 1566015, 1652140, and a Google Faculty Research Award.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases: the logical level.* Addison-Wesley Longman Publishing Co., Inc.

[2] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 25–34.

[3] Rajeev Alur, Pavol Černỳ, Parthasarathy Madhusudan, and Wonhong Nam. 2005. Synthesis of interface specifications for Java classes. *ACM SIGPLAN Notices* 40, 1 (2005), 98–109.

[4] Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining specifications. *ACM Sigplan Notices* 37, 1 (2002), 4–16.

[5] Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing formal specifications using testing. In *International Conference on Tests and Proofs*. Springer, 6–21.

[6] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *JCS* 6 (2010).

[7] William W Cohen. 1994. Recovering software specifications with inductive logic programming. In *AAAI*, Vol. 94. 1–4.

[8] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45.

[9] Mark Gabel and Zhendong Su. 2008. Symbolic mining of temporal specifications. In *Proceedings of the 30th international conference on Software engineering*. ACM, 51–60.

[10] Timon Gehr, Dimitar Dimitrov, and Martin Vechev. 2015. Learning commutativity specifications. In *International Conference on Computer Aided Verification*. Springer, 307–323.

[11] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. 2007. Discovering documentation for Java container classes. *IEEE Transactions on Software Engineering* 33, 8 (2007), 526–543.

[12] Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. 2005. Permissive interfaces. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 31–40.

[13] Guofei Jiang, Haifeng Chen, Cristian Ungureanu, and Kenji Yoshihira. 2007. Multiresolution abnormal trace detection using varied-length n-grams and automata. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 37, 1 (2007), 86–97.

[14] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. 422–430.

[15] Claire Le Goues and Westley Weimer. 2009. Specification mining with few false positives. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 292–306.

[16] Stephen Muggleton. 1995. Inverse entailment and Progol. *New generation computing* 13, 3-4 (1995), 245–286.

[17] Stephen Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter A. Flach, Katsumi Inoue, and Ashwin Srinivasan. 2012. ILP turns 20 - Biography and future challenges. *ML* 86, 1 (2012), 3–23. DOI:http://dx.doi.org/10.1007/s10994-011-5259-2

[18] 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*. ACM, 42–56.

[19] J. Ross Quinlan. 1990. Learning logical definitions from relations. *Machine learning* 5, 3 (1990), 239–266.

[20] Sriram Sankaranarayanan, Swarat Chaudhuri, Franjo Ivančić, and Aarti Gupta. 2008. Dynamic inference of likely data preconditions over predicates by tree learning. In *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 295–306.

[21] Sriram Sankaranarayanan, Franjo Ivanci, and Aarti Gupta. 2008. Mining library specifications using inductive logic programming. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 131–140.

[22] Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare logic for verifying k-safety properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 57–69.

[23] Sympy. 2017. Python library for symbolic mathematics. http://www.sympy.org/en/index.html. (2017).

[24] Andrzej Wasylkowski and Andreas Zeller. 2011. Mining temporal specifications from object usage. *Automated Software Engineering* 18, 3-4 (2011), 263–292.

[25] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*. ACM, 282–291.