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.

1 INTRODUCTION

Formal specifications of library functions play a critical role in a number of settings: In program analysis and verification, specifica-

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:

\[
\begin{align*}
\text{add}(x, y) = z &\iff \text{add}(y, x) = z \quad (1) \\
\text{gt}(x, y) = t \land \text{gt}(y, z) = t &\implies \text{gt}(x, z) = t \quad (2) \\
\text{trim} \circ \text{uppercase}(x) = y &\iff \text{uppercase} \circ \text{trim}(x) = y \quad (3) \\
x > 0 \land \text{abs}(x) = y &\implies \text{abs}(y) = x \quad (4)
\end{align*}
\]

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

Primary challenges There are three primary challenges that arise in learning relational specifications from a dataset: (i) \textbf{What does it mean for a specification to explain (partial) input–output behavior?} (ii) \textbf{How do we efficiently handle large amounts of input–output data?} (iii) \textbf{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 \textbf{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 \textbf{more likely} to be true depending on a measure of the \textbf{positive evidence} that is available for it.

The \textbf{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 database. 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:

\[
\forall x, y, z \in \mathbb{D}, f(x, y) = t \land f(y, z) = t \Rightarrow f(x, z) = t
\]

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.

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.

![Figure 1: Main components of the Bach algorithm](image-url)
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:

\[
\text{add}(x, y) = z \iff \text{add}(y, x) = z
\]

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:

- **Positive evidence** Is there evidence that the specification holds?
- **Negative 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 \( \text{add}(a, b) = c \) and \( \text{add}(b, a) = c, \) 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 \( R_{\text{add}}(x, y, z) \), 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

\[
R_{\text{add}}(x, y, z) \Rightarrow R_{\text{add}}(y, x, z)
\]

where \( \Rightarrow \) 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:

\[
P(X, Y, Z) \leftarrow R_{\text{add}}(X, Y, Z), R_{\text{add}}(Y, X, Z).
\]

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 \( (a, b, c) \in R_{\text{add}} \) and \( (b, a, c) \in R_{\text{add}} \), 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:

\[
N(X, Y, Z) \leftarrow R_{\text{add}}(X, Y, Z), R_{\text{add}}(Y, X, Z'), Z \neq Z'.
\]

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

E2: Transitivity of Comparison

Consider \( \text{gt} \), the function implementing the greater-than operation for integers with data in Figure 2. The specification induction phase will propose the following specification:

\[
(\text{gt}(x, y) = w \land \text{gt}(y, z) = w) \Rightarrow \text{gt}(x, z) = w
\]

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 \( \text{gt} \). 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.

\[
P(\ldots) \leftarrow R_{\text{gt}}(X, Y, W), R_{\text{gt}}(Y, Z, W), R_{\text{gt}}(X, Z, W)
\]

\[
N(\ldots) \leftarrow R_{\text{gt}}(X, Y, W), R_{\text{gt}}(Y, Z, W), R_{\text{gt}}(X, Z, W'), W' \neq W
\]

E3: Identity of Absolute Value

Consider the function \( \text{abs} \) (absolute value) with data in Figure 2.

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

\[
\text{abs}(x) = x
\]

which can be viewed as the implication \( \text{true} \Rightarrow \text{abs}(x) = x \).

**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 \( N(X) \):

\[
P(X) \leftarrow R_{\text{abs}}(X, X)
\]

\[
N(X) \leftarrow R_{\text{abs}}(X, X'), X \neq X'
\]

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

\[
G \Rightarrow \text{abs}(x) = x
\]

![Figure 2: Example observed function executions.](image-url)
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., $\text{>, <, +, -}$. In this example, the abduction phase might return the formula $x \geq 0$, resulting in the correct specification

$$x \geq 0 \Rightarrow \text{abs}(x) = x$$

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 $\text{concat}$, which concatenates two strings, and $\text{len}$, which returns the length of a string. Here, $\varepsilon$ is the empty string.

Suppose that the induction phase proposes the specification

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

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

$$y = \varepsilon \Rightarrow (\text{len}(\text{concat}(x, y)) = z \Leftrightarrow \text{len}(x) = z)$$

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

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

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 $\text{ar}(f_i)$. A formula $\mathcal{F}$ is of the form

$$\forall V, \mathcal{G} \Rightarrow (\Psi \Rightarrow \Phi) \text{ or } \forall V, \mathcal{G} \Rightarrow (\Psi \Rightarrow \Phi)$$

where (i) $V$ is a set of variables, (ii) $\mathcal{G}$, the guard, is a formula over an interpreted set of predicate and function symbols. (iii) $\Psi$ (analogously, $\Phi$) is defined as $\bigwedge_{i} \psi_i$, where each $\psi_i$ is an atom of the form $\mathcal{G} \Rightarrow o$, where $o$ is a variable in $V$ and $\mathcal{G}$ is a nested function application over the functions $\Sigma$ 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 3.3

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

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 $\mathcal{D}_g$ 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 $\forall V, \mathcal{G} \Rightarrow (\Psi \Rightarrow \Phi)$ or $\forall V, \mathcal{G} \Rightarrow (\Psi \Rightarrow \Phi)$, we define positive evidence as:

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

Informally, positive evidence is the set of instantiations of variables $V$ that non-vacuously satisfy the body of $\mathcal{F}$, i.e., $G \land \Psi \land \Phi$. ■
We are now ready to formalize Bach. The algorithm is shown in where each will be the set of two assignments \( \text{cmp} \) can be filled to \( J \) in the set \( V \) of variables. We assume there is a fixed signature discovered likely specifications. Both sets grow monotonically.

Conjunctions of atoms, which are used to construct specifications; consistent with the data.

Consider formula \( F \) how well \( F \) explains \( D \). Recall that an atom is of the form \( \Phi \in J \), where \( D \) is inconsistent with \( D \). An optimal specification \( F^{\ast} \) is a likely specification that maximizes some function \( h \) of \( D \) and \( F \). Formally,

\[
F^{\ast} = \arg \max \{ h(D, F) \} \quad \text{subject to} \quad \neg(D, F) = \emptyset
\]

An immediate choice for \( h \) is given by \( h(D, F) = |\text{pos}(D, F)| \). Intuitively, this uses the amount of positive evidence as a proxy for how well \( F \) 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) \( J \), a set of conjunctions of atoms, which are used to construct specifications; e.g., given \( \Psi, \Phi \in J \), we can construct \( \forall V. \Psi \equiv \Phi \). (ii) \( S \), a set of discovered likely specifications. Both sets grow monotonically.

Search We assume there is a fixed signature \( \Sigma \), 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 \( \text{add}, \text{exp}_V, \) and \( \text{exp}_F \) construct new atoms and conjoin them to formulas in the set \( J \). The symbol \( \bullet \) is used to denote a wildcard, a hole that can be filled to complete an atom. A conjunction \( \Phi \in J \) is complete if it contains no wildcards (denoted \( \text{cmp} \) in Figure 3).

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

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

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

1. \( \forall \sigma \in \text{neg}(D, F). \sigma(G) \) is unsatisfiable.
2. \( \exists X \subseteq \text{pos}(D, F). X \neq \emptyset \land \forall \sigma \in X. \sigma(G) \) 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 \( S \). This is maintained by construction through (i) the rules \( \text{ind}_{\Phi} \) and \( \text{abd}_{\Phi} \), (ii) and the definition of \( \text{classify} \), 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 \( \text{add} \) to formulas \( \Phi \) 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 \( \text{ind}_{\Phi} \) and \( \text{abd}_{\Phi} \). In practice, we apply \( \text{abd}_{\Phi} \) right after a failed application of an induction rule. Specifically, if a failed application of \( \text{ind}_{\Phi} \) results in positive evidence and negative evidence, then we apply \( \text{abd}_{\Phi} \) 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 \( F \) 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 \( F \) 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 \text{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.
where \( H \) is the set of \( \phi \) variables of size equal to the arity of the corresponding relation; the \( \phi \) variables are stored in a set \( \Phi \) of clauses encoding the positive/negative evidence of \( \text{body} \). A Horn clause is of the form:

\[
\forall X \left( \Phi \land (f(x_1, \ldots, x_n) = 1) \right) \rightarrow (\Psi \lor (\neg \Phi \land (f(x_1, \ldots, x_n) = 0)))
\]

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

\[
\forall x, y, f(g(x)) = y \iff h(x) = y
\]

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

\[
A_1(x, X_g) \leftarrow O' = X_m, R_f(O, O'), R_g(X_c, O).
\]

The right-hand side of \( \iff \) is encoded similarly.

**Positive evidence** Positive evidence consists exactly of those variable assignments that non-trivially satisfy both \( \Psi \) and \( \Phi \). As \( \Psi \) and \( \Phi \) are already fully encoded in the relations \( A_H(X^d) \) and \( B_H(X^b) \), this requirement is immediately encodable as the Horn clause \( P(X) \leftarrow A_H(X^d), B_H(X^b) \).

**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 \( \Psi \) but falsify the right side \( \Phi \). 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 \( \phi_1 \)'s. For instance, assignments that
The first clause encodes the requirement that
\[ h(x) = y \]
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 \( F \) of the form \( \forall x. \Psi \equiv \Phi \) or \( \forall x. \Psi \implies \Phi \), let \( N(X) \) and \( P(X) \) be the relations computed using the Horn clauses \( C \) from Figure 4. Then, \( P(X) = \text{pos}(D, F) \) and \( N(X) = \neg \text{neg}(D, F) \).

**Complexity** It is important to note that the decision problem of solving a conjunctive query is \( \text{NP} \)-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 \( \text{PTIME} \) (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 \( \text{NP} \)-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 Souffle Datalog engine [14] to compute positive/negative evidence.

**Ordering the search** The search rules ADD, EXP_\text{eq}, and EXP_\text{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 \( F = \Psi \equiv \Phi \) correct, it records one of \( \Psi \) and \( \Phi \) (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, \( \Psi, \Phi \subseteq F \), it first attempts to learn the bi-implication \( \Psi \equiv \Phi \), using the \text{IND}_\text{eq} rule. If \text{IND}_\text{eq} 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 \( \Psi \implies \Phi \), using the rule \text{IND}_\text{exp}. 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:

\[
G_1 \equiv (\Phi \equiv \Psi), \quad G_2 \equiv (\Phi \equiv \Psi), \quad G_3 \equiv (\Psi \equiv \Phi)
\]
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

\[
p = \text{true} \Rightarrow (\text{valid}(x) = p \Rightarrow \text{sat}(x) = p),
\]

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

\[
\text{valid}(x) = p \land \text{valid}(y) = p \Rightarrow \text{valid}(\text{and}(x,y)) = p,
\]

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

\[
\text{lstrip}(x) = y \Rightarrow \text{lstrip}(y) = y.
\]
where lstrip(x) removes all whitespace on the left of x, as well as useful facts like

\[ p = \text{true} \Rightarrow (\text{prefix}(x, x) = p), \]

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:

\[ \text{concat}(y, \text{reverse}(y)) = x \Rightarrow \text{reverse}(x) = x. \]

**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,

\[ 3k \cdot x = 2\pi k + y \Rightarrow (\sin(x) = z \Leftrightarrow \sin(y) = z), \]

where \( 3k \cdot x = 2\pi k + y \) is provided as a predicate on \( x \) and \( y \). Bach also discovers that \( \sin \) and \( \arcsin \) are almost inverses of each other:

\[ \arcsin(z) = x \Rightarrow \sin(x) = z. \]

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:

\[ b = \text{true} \Rightarrow (\text{encl}(x, y) = b \land \text{encl pt}(y, p) = b \Rightarrow \text{encl pt}(x, p) = b) \]

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 \( 2\pi \) results in the shape itself:

\[ 3k \cdot x = 2\pi k \Rightarrow \text{rotate}(y, x) = y. \]

**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 \( h(D, F) = (|F|^{-1}. \text{pos}(D, F)) \), 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 \( \text{f199} \).

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 \( \approx 9 \times \) 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...
With QuickSpec [5] and Daikon [8]. The work of Henkel et al. [11] on the worst-performing benchmark, ff199, where we repeat this process multiple times, each time randomly sampling 10% of the time. Conversely, our best-performing benchmark, ff199, to are non-invertible matrices) with low numbers of observations. This is due to the difficulty of sampling corner cases (e.g., 0, or strings) 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.

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 \( \frac{FP}{TP + FP} \), 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 \( \approx 0.1 \), 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 Prolog [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


