Learning commutativity specifications

Timon Gehr Dimitar Dimitrov Martin Vechev

Department of Computer Science
ETH Zürich

Abstract. In this work we present a new sampling-based "black box" inference approach for learning the behaviors of a library component. As an application, we focus on the problem of automatically learning *commutativity specifications* of data structures. This is a very challenging problem, yet important, as commutativity specifications are fundamental to program analysis, concurrency control and even lower bounds.
Our approach is enabled by three core insights: (i) *type-aware sampling* which drastically improves the quality of obtained examples, (ii) *relevant predicate discovery* critical for reducing the formula search space, and (iii) an *efficient search* based on weighted-set cover for finding formulas ranging over the predicates and capturing the examples.
More generally, our work learns formulas belonging to fragments consisting of quantifier-free formulas over a finite number of relation symbols. Such fragments are expressive enough to capture useful specifications (e.g., commutativity) yet are amenable to automated inference.
We implemented a tool based on our approach and have shown that it can quickly learn non-trivial and important commutativity specifications of fundamental data types such as hash maps, sets, array lists, union find and others. We also showed experimentally that learning these specifications is beyond the capabilities of existing techniques.

1 Introduction

In this work we present a new and scalable "black box" technique for learning complex library specifications. Our technique is based on sampling of library behaviors, is fully automatic, and quickly learns succinct and precise specifications of complex interactions beyond the reach of current techniques. Concretely, our approach learns specifications in fragments of the quantifier-free formulas over a finite number of relation symbols. Such fragments are expressive enough to capture useful specifications yet are amenable to automated inference. Note that even though the fragment is quantifier-free, the relations in the fragment can be defined using quantifiers and hence the learned formulas may include quantifiers.

We have instantiated our approach to learning *commutativity specifications* of data structures, a hard yet practically important problem as these specifications are fundamental to concurrency (e.g., program analysis [4], concurrency control [23,14,9,13], and lower bounds [1]). This is the first automatic approach that can precisely and quickly infer commutativity specifications for useful data types such as hash map, union find, array list and others[1].

[1] Specifications and source code available at `http://www.srl.inf.ethz.ch/core`

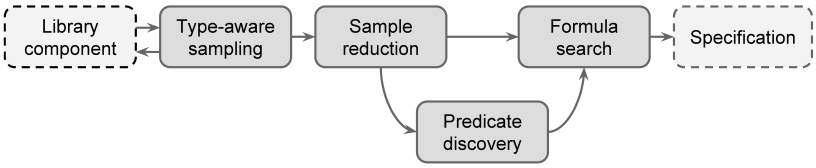

Fig. 1. Our approach to specification inference.

The flow and ingredients of our approach are shown in Fig. 1. Given a library component (e.g., a data structure), instead of blindly sampling its behaviors and obtaining redundant examples, a key insight is to introduce the notion of *type-aware sampling* which allows us to obtain a diverse set of quality examples (informally, two examples have the same type if they are indistinguishable by a formula in the logical fragment). However, even with advanced sampling, the sheer number of examples can overwhelm the search. That is why we reduce the size of this set by filtering out examples indistinguishable by the logical fragment. Once the final set of examples is obtained, a critical step is to reduce the search space of candidate formulas by *discovering relevant predicates* (a fundamental step in many static analysis approaches [15]). The key insight of this step is to filter out a predicate if it cannot be distinguished from its negation by any positive-negative example pair. Finally, we search for formulas over the relevant predicates that cover our set of examples. We show that a greedy algorithm based on weighted set-cover is quite effective in finding non-trivial and optimal specifications quickly – it infers complex commutativity conditions in seconds.

Main contributions. The main contributions of this work are:

- A new sampling-based "black box" approach for learning specifications in fragments consisting of quantifier-free formulas over a finite number of relation symbols. The key insights of our technique are: type-aware sampling, sample reduction, relevant predicate discovery and efficient formula search.
- An instantiation of the approach for learning commutativity specifications including a specialized sampling procedure.
- An experimental evaluation illustrating that our approach quickly learns practical commutativity specifications of fundamental data types such as hash map, set, array list, union find and others. We further show that learning these specifications is beyond the reach of current approaches.

2 Overview

We next illustrate our approach on an example: inferring the conditions for when two insertions in a `Map` data type commute. The aim of this section is to provide an intuitive understanding of the general framework presented later.

Commutativity. Consider a standard `Map` data type, supporting the methods $\texttt{get}(k)/r$ and $\texttt{put}(k,v)/r$, where `put` returns the old value r under the key k. We seek to infer the commutativity specification for the method pair `put`/`put`. Two method invocations *commute* if they can always be reordered without any observable effect. In our case, $\texttt{put}(k_1,v_1)/r_1$ and $\texttt{put}(k_2,v_2)/r_2$ commute if and only if they access different keys or both leave the map unmodified, captured as:

$$\varphi(k_1,v_1,r_1,k_2,v_2,r_2) := k_1 \neq k_2 \vee v_1 = r_1 \wedge v_2 = r_2. \tag{1}$$

When inferring logical specifications, we assume a fixed logical fragment that defines our search space. Let us set this fragment to consist of arbitrary boolean combinations of constraints over the numeric relations $<$ and $=$.

Type-aware sampling. We treat the data type as a black box, sampling it at random in order to obtain examples of commuting/non-commuting behaviors. We first prepare a random data type state, then choose random arguments for the methods, and finally invoke them in both orders. This way we obtain tuples of arguments and return values which are classified as positive (commuting) or negative (non-commuting). Fig. 2 shows a sample of several examples for `put`/`put`. As a pure random sample may contain many redundant examples, we use a sampling method that tries to rule out a large number of candidate specifications.

#	k_1	v_1	r_1	k_2	v_2	r_2	+/-
1.	3	2	2	4	2	1	+
2.	1	0	0	2	0	-1	+
3.	2	0	0	1	0	-1	+
4.	2	0	-2	1	0	-1	+
5.	1	2	2	1	1	2	-
6.	1	-1	-1	1	1	-1	-

Fig. 2. Commutativity and non-commutativity examples of `put`/`put`.

We say that examples have the same *logical type* if they cannot be distinguished by any formula in the fragment. For instance, examples 1 and 2 in Fig. 2 have the same logical type, as all basic predicates of the form $x = y$ and $x < y$ have the same truth value on both 1 and 2 (where $x, y \in \{k_i, v_i, r_i\}$). To rule out as many candidates as possible, we need to diversify the logical types observed in a sample. Thus, we first choose a formula representing the logical type of the method arguments (e.g., $v_1 = v_2 < k_1 < k_2$ for examples 1 and 2), and then select concrete values that satisfy that formula. We finally test for commutativity with these arguments.

Sample size reduction. We would ideally like to diversify the logical type of the complete example, and not only the arguments. However, we cannot control what the return values are, as we obtain these *after* executing the commutativity test. This can lead to redundancy in the sample as some examples will have the same logical type. We filter out such uninformative examples and obtain a significantly smaller sample which covers exactly the same variety of logical types. Inference over this reduced sample is much faster than over the larger one.

Discovery of relevant predicates. Logical specifications in our fragment consist of disjunctions and conjunctions of literals. For example, when inferring a specification for `put`/`put` we search for a formula built from all possible literals over

the relations $<$, $=$, and the variables k_i, v_i, r_i, namely, $k_1 = k_2$, $k_1 \neq k_2$, $k_1 < k_2$, $k_1 \geq k_2$, $v_1 = r_1$, $v_2 = r_2$, etc. As we can see, literals such as $k_1 < k_2$ and $k_1 \geq k_2$ do not occur in the target specification (1), and are therefore irrelevant for the search. Thus, to reduce the formula search space, we introduce a procedure that identifies irrelevant literals from the information provided by the sample. Informally, the idea is to look at a contradicting pair of literals such as $k_1 < k_2$ and $k_1 \geq k_2$. We then consider those pairs of commutativity examples that can be distinguished only by the two contradicting literals. Examples 2 and 3 in Fig. 2 form such a pair and can be distinguished only by $k_1 < k_2$ and $k_1 \geq k_2$. If the two examples are either both positive or both negative, then the pair of literals is irrelevant for distinguishing these examples. We rule out the pair if it is irrelevant for all pairs of examples (later we describe a more elaborate technique that is able to rule out more literals). In our example, we obtain a reduced set of literals: $k_1 \neq k_2$, $v_1 = r_1$, $v_2 = r_2$, ...

Formula search. Finally, given a data sample (as in Fig. 2) and a set of literals, we search for a formula which is consistent with the sample. We aim to infer *sound* specifications. In the case of commutativity, this means that the specification should always imply that two invocations do commute. Therefore, we search for a formula which does not evaluate to $true$ on negative examples yet evaluates to $true$ in as many positive examples as possible. To prevent overfitting, we search for the smallest such formula. This is the reason why our approach requires both positive and negative examples, for otherwise the formulas $true$ and $false$ would be trivial solutions. To infer a formula meeting or objective, we developed a procedure that interleaves exhaustive and greedy search.

Determining sample size. For the inference to be self-contained, we need an automatic way to determine the sample size. We employ an intuitive approach: new examples are drawn until the result of the formula search stabilizes. To confirm the stabilization, we draw examples in blocks, each block being twice as large as the previous one. For every new block we run the formula search, and if it produces the same outcome twice in a row, we declare the formula as stable. The exponential increase of the block size ensures that we restart the search at most a logarithmic number of times, and that we draw at most linearly more examples than required.

In what follows, we first describe a general framework for learning specifications belonging to fragments consisting of quantifier-free formulas over a finite number of relation symbols. Then, we show how to instantiate our approach for learning practical commutativity specifications (belonging to a restriction of the general fragment), and finally we discuss an experimental evaluation on a number of real world data structures.

3 Background

In this section we introduce several key notions from logic that are essential for addressing the problem of specification inference. In particular, the two concepts

that we will rely upon later sections are *definable relations* and *logical types*. The family of definable relations forms the hypothesis class that an inference algorithm considers to explain the observed data, and the complexity of that class governs the difficulty of inference. A logical type abstracts all data points carrying the same information with respect to the hypothesis class. Among other uses, types are crucial when estimating the quality of a data sample.

Definition 1. *A* structure $\mathfrak{X}$ *consists of a carrier set, a set of relations and functions over the carrier, and relation and function symbols naming them.*

We think of a structure $\mathfrak{X}$ as providing the context for interpreting logical formulas. The first-order *language* of $\mathfrak{X}$ consists of all first-order formulas built using equality and the symbols mentioned by the structure. For a formula $\varphi(\boldsymbol{u})$ from the language and a tuple $\boldsymbol{u}$ of elements from the carrier, we shall use the standard notation $\mathfrak{X} \models \varphi(\boldsymbol{u}/\boldsymbol{x})$ to state that $\varphi(\boldsymbol{x})$ holds true for $\boldsymbol{u}$.

In general, not all relations over the carrier can be expressed by formulas over a structure's language. In the present work, we consider relations expressible in a boolean closed fragment $\mathcal{L}$ of the full language of $\mathfrak{X}$:

Definition 2. *A relation h over $\mathfrak{X}$ is $\mathcal{L}$-definable if there exists a formula $\varphi(\boldsymbol{x}) \in \mathcal{L}$ such that for any tuple $\boldsymbol{u} \in h$ we have that $\boldsymbol{u} \in h \iff \mathfrak{X} \models \varphi(\boldsymbol{u}/\boldsymbol{x})$.*

Looking for a specification in the fragment $\mathcal{L}$ implies that we need to approximate an unknown relation c with an $\mathcal{L}$-definable relation h, given a finite sample of c. Thus, the family $\mathcal{H}$ of all $\mathcal{L}$-definable relations forms our hypothesis class.

We will make heavy use of a natural abstraction that a logic induces over tuples of elements. Two tuples $\boldsymbol{u}$ and $\boldsymbol{v}$ have the same *logical type*[2][3] if they satisfy the same $\mathcal{L}$-formulas:

Definition 3. *The $\mathcal{L}$-type $tp_{\mathcal{L}}(\boldsymbol{u})$ of a any tuple $\boldsymbol{u}$ over the carrier of $\mathfrak{X}$ is the set of formulas $\Phi(\boldsymbol{x}) = \{\varphi(\boldsymbol{x}) \in \mathcal{L} \mid \mathfrak{X} \models \varphi(\boldsymbol{u}/\boldsymbol{x})\}$ that it satisfies.*

In other words, tuples having the same $\mathcal{L}$-type are indistinguishable by formulas in the fragment $\mathcal{L}$. In turn, this determines the structure of the $\mathcal{L}$-definable relations. Let us call the set $\{\boldsymbol{u} \in X^n \mid tp_{\mathcal{L}}(\boldsymbol{u}) = \Phi(\boldsymbol{x})\}$ the *preimage* of $\Phi(\boldsymbol{x})$. The collection of all such preimages partitions the set of tuples, and moreover

Observation 1. *Every $\mathcal{L}$-definable relation is an unique disjoint union of preimages of $\mathcal{L}$-types.*

Therefore, a relation is $\mathcal{L}$-definable if and only if it does not separate any type preimage into two parts. Later, we will make extensive use of this fact.

In our work, we often need a tangible way to manipulate logical types: we would like to replace a type (a potentially infinite collection of formulas) with a finite description (i.e., just a single formula).

[2] Logical types should not be confused with the concept of types in type theory.
[3] To find more about logical types the reader can consult [17,2], or the classic [3,10].

Definition 4. *A type $\Phi(\boldsymbol{x})$ is called* isolated *when some formula $\varphi(\boldsymbol{x}) \in \Phi(\boldsymbol{x})$* generates *it, that is for all $\psi(\boldsymbol{x}) \in \Phi(\boldsymbol{x})$ we have that $\mathfrak{X} \models \forall \boldsymbol{x}.\, \varphi(\boldsymbol{x}) \to \psi(\boldsymbol{x})$.*

However, we would like an even stronger property to hold true, namely that the preimage of the type $\Phi(\boldsymbol{x})$ be $\mathcal{L}$-definable. This is important for obtaining data samples with a chosen type. Fortunately, the boolean closedness of the fragment ensures that a preimage is $\mathcal{L}$-definable if and only if the type is isolated:

$$tp_{\mathcal{L}}(\boldsymbol{u}) = \Phi(\boldsymbol{x}) \iff \mathfrak{X} \models \Phi(\boldsymbol{u}/\boldsymbol{x}) \iff \mathfrak{X} \models \varphi(\boldsymbol{u}/\boldsymbol{x}) \tag{2}$$

In order to guarantee that we work with isolated types we will impose certain restrictions on the family of definable relations. Let $\mathcal{L}(\boldsymbol{x})$ denote the subfragment of $\mathcal{L}$ consisting of all the formulas $\varphi(\boldsymbol{x}) \in \mathcal{L}$ having free variables among $\boldsymbol{x}$, where as usual $\boldsymbol{x}$ is a finite list of distinct variables.

Observation 2. *If the family of $\mathcal{L}(\boldsymbol{x})$-definable relations is finite, then there are finitely many $\mathcal{L}(\boldsymbol{x})$-types, and all of them are isolated.*

We shall focus on the setting where: (i) the structure $\mathfrak{X}$ is relational, i.e., it mentions no function symbols (including constants); (ii) the relation symbols are finitely many; and (iii) $\mathcal{L}(\boldsymbol{x})$ is the quantifier-free fragment of $\mathfrak{X}$. Combined, these conditions guarantee that there are finitely many $\mathcal{L}(\boldsymbol{x})$-definable relations in the structure $\mathfrak{X}$, and therefore we can leverage Observation 2. We will further assume that formulas are in *negation normal form*, i.e., that all negations are pushed in front of atomic subformulas.

4 Learning formulas

We next describe our approach to learning logical formulas from examples over a structure $\mathfrak{X}$. We first state our learning objective, i.e., which formula to select given a sample. Then, we discuss how we search for such a formula. Finally, we present a way to reduce the formula search space by discarding irrelevant literals.

4.1 Learning objective

Given a sample (s_+, s_-) of positive and negative examples from an unknown relation c over the carrier of $\mathfrak{X}$, we would like to infer a definition of c in the logical fragment $\mathcal{L}(\boldsymbol{x})$. It is important to note that such a definition need not exist in general. That is why, our goal will be to find a formula that defines a relation $h \in \mathcal{H}$ that best approximates c in our *hypothesis class* $\mathcal{H}$ of $\mathcal{L}(\boldsymbol{x})$-definable relations. Two natural approximation criteria are: best under-approximation, i.e., the maximal $h \subseteq c$, and best over-approximation, i.e., the minimal $h \supseteq c$. We shall work with under-approximations but all the machinery can be used directly for over-approximations directly as the two notions are dual. Recalling Observations 1 and 2 we establish the existence of best approximations:

Theorem 1. *The best under-approximation $h \in \mathcal{H}$ to any relation c over $\mathfrak{X}$ equals the disjoint union $\bigcup\{t \subseteq c \mid t \text{ is a preimage of some type}\}$.*

However, we can search for this h only indirectly, for we merely have an access to the finite sample (s_+, s_-) instead of the complete relation c. Our learning objective will be to find a hypothesis h that includes a maximum number of positive examples, while excluding all negative ones. In other words, we have the following optimization problem over the hypothesis class $\mathcal{H}$:

$$\text{maximize } |h \cap s_+|, \quad \text{subject to } h \cap s_- = \varnothing \tag{3}$$

In general, an optimal solution to this objective is not unique. Moreover, for a learning procedure we need an effective criterion telling us when a candidate hypothesis satisfies (3).

Definition 5 (Observed status). *Call the* observed status *of a type $\Phi(\boldsymbol{x})$ with a preimage t:* positive *if $s_+ \cap t \neq \varnothing$ and $s_- \cap t = \varnothing$;* negative *if $s_+ \cap t = \varnothing$ and $s_- \cap t \neq \varnothing$;* ambiguous *if $s_+ \cap t \neq \varnothing$ and $s_- \cap t \neq \varnothing$. Otherwise, call it* missing.

Theorem 2. *A hypothesis $h \in \mathcal{H}$ is optimal with respect to (3) for a given sample (s_+, s_-) if and only if for every tuple $\boldsymbol{u}$ from $\mathfrak{X}$ with non-missing type*

$$\begin{aligned} \boldsymbol{u} \in h &\iff tp_{\mathcal{L}(\boldsymbol{x})}(\boldsymbol{u}) \text{ is positive} \\ \boldsymbol{u} \notin h &\iff tp_{\mathcal{L}(\boldsymbol{x})}(\boldsymbol{u}) \text{ is negative or ambiguous} \end{aligned}$$

The theorem follows from Observation 1. Note that even if a tuple belongs to s_+, its type might still be ambiguous as another tuple of the same type might belong to s_-. Optimal hypotheses cannot be further distinguished by the sample (s_+, s_-), and so we need an additional principle to select one of them such that we avoid overfitting. We shall rely on the minimum description length principle [18,22,19], which suggests to search for a solution of (3) defined by a formula of minimal size. This is also important for human-readability.

4.2 Formula search

To find a solution to (3) we combine an exhaustive search interleaved with a greedy algorithm. In accordance with Theorem 2 we consider the subset $s'_+ \subseteq s_+$ of positive examples having a type with a positive observed status. We search until we find a formula that evaluates to true on all of s'_+, and evaluates to false on all of s_-. By Theorem 2, the discovered formula satisfies (3). By Theorem 1, at least one optimal hypothesis exists, and therefore the search always terminates. During the search, we try to minimize the size measure given by:

$$\|\mathit{true}\| = \|\mathit{false}\| = 0; \quad \|\text{literal}\| = 1; \quad \|\varphi \wedge \psi\| = \|\varphi \vee \psi\| = 1 + \|\varphi\| + \|\psi\| \tag{4}$$

We enumerate the formulas of $\mathcal{L}(\boldsymbol{x})$ in increasing size, via a simple dynamic programming approach that alone guarantees finding a formula of minimum size. We employ the standard heuristic to consider two formulas equivalent if they produce the same results on the sample $s_+ \cup s_-$. Exhaustive enumeration, however, is feasible for inferring small formulas only. This is why we interleave

it with a greedy algorithm, which does not guarantee minimality, but is much faster in practice. For each formula size i, we consider the set G of conjunctions which have size smaller than i, and also evaluate to false on all of s_-. From those, we try to build a disjunction $\bigvee F$ which covers all of s'_+, where $F \subseteq G$. To find a disjunction of small size, we phrase the problem as an instance of weighted set cover, and use a greedy approximation. We weight each conjunction $\varphi \in G$ with $\|\varphi\|$ and seek a cover $F \subseteq G$ of s'_+ with small total weight. We run a standard greedy algorithm to produce a cover F. If the cover has less than $2i$ formulas, we terminate the search; else, we move on to size $i + 1$.

4.3 Predicate discovery

We now describe how to reduce the formula search space $\mathcal{L}(\boldsymbol{x})$ by restricting the set of literals considered during formula enumeration. We prune formulas that contain literals irrelevant for explaining the sample (s_+, s_-). The approach has to be instantiated for the specific structure $\mathfrak{X}$ under consideration, and we first illustrate it for the two-valued boolean algebra $\{0, 1\}$, or equivalently for the case of learning propositional formulas.

Two-valued case. Here, free variables range over 0–1, and our fragment $\mathcal{L}(\boldsymbol{x})$ has a relation T interpreted as $T(x) \iff x = 1$. The logical type of every tuple $\boldsymbol{u}$ is characterized by a single conjunction $\bigwedge Q_i$, where $Q_i = T(x_i) \iff u_i = 1$ and $Q_i = \neg T(x_i) \iff u_i = 0$. Therefore, in the two-valued case no distinct tuples can have the same logical type, and we can identify the type of a tuple with the tuple itself, i.e., a 0–1 valued vector.

Definition 6. *A sample (s_+, s_-) of n-tuples is* monotone *in the i-th coordinate, if for all tuples $\boldsymbol{u}$ and $\boldsymbol{v}$ with $|\boldsymbol{u}| = i - 1$, $|\boldsymbol{u}| + |\boldsymbol{v}| = n - 1$ we have that $(\boldsymbol{u}, 0, \boldsymbol{v}) \in s_+$ implies $(\boldsymbol{u}, 1, \boldsymbol{v}) \notin s_-$. Similarly, the sample is* antitone *in the i-th coordinate if we instead require that $(\boldsymbol{u}, 1, \boldsymbol{v}) \in s_+$ implies $(\boldsymbol{u}, 0, \boldsymbol{v}) \notin s_-$.*

If a sample (s_+, s_-) is monotone in i, then we can extend it to an optimal hypothesis h with the similar property of $(\boldsymbol{u}, 0, \boldsymbol{v}) \in h$ implying $(\boldsymbol{u}, 1, \boldsymbol{v}) \in h$. A folk theorem states that any formula defining a relation with this property can be converted to an equivalent formula not containing the literal $\neg T(x_i)$. Therefore, in our search for an optimal hypothesis we can prune formulas containing this literal. Analogously, we can prune $T(x_i)$ when (s_+, s_-) is antitone in i.

Generalization. To handle more general logical fragments, we shall abstract the notion of monotonicity from the two-valued case. There, the concept of a tuple and its type essentially coincided; we had a condition over pairs tuples $(\boldsymbol{u}, 0, \boldsymbol{v})$, $(\boldsymbol{u}, 1, \boldsymbol{v})$ that increase (or decrease) in their i-th coordinate. In the more general case, we shall use a condition over logical types and not tuples. As a type assigns a truth value to every literal and (in our fragment) formulas are combinations of literals, the role of coordinates will be played by the literals themselves. For each literal λ we assume a *neighbor relation* $\mathcal{N}_\lambda$ that relates pairs of types for which the truth value of λ increases from *false* to *true*, i.e., $(\Phi, \Psi) \in \mathcal{N}_\lambda$ must imply $\neg\lambda \in \Phi$ and $\lambda \in \Psi$ (the converse need not hold).

Definition 7. *Given a literal λ and a neighbor relation $\mathcal{N}_\lambda$, we say that a sample (s_+, s_-) is* $\mathcal{N}_\lambda$-unate *when for all pairs of types $(\Phi, \Psi) \in \mathcal{N}_\lambda$, if Φ has a positive observed status, then Ψ has a positive or a missing observed status.*

In the two-valued case, we implicitly used the relation: $(\Phi, \Psi) \in \mathcal{N}_\lambda$ if and only if the Ψ is obtained from Φ by switching the truth value of λ, but of no other literals. This is too restrictive in general, as switching the truth value of one literal may require a switch in another. For example, consider logical types in the order structure $(\mathbb{Z}, <)$ of the integers. There, switching $x \neq y$ from *false* to *true* also requires switching either $x < y$ or $y < x$ from *false* to *true*.

We now define when a neighbor relation $\mathcal{N}_\lambda$ is admissible. In general such relations have to be derived for the specific structure under consideration. To gain more flexibility, we shall allow $\mathcal{N}_\lambda$ to depend on the sample (s_+, s_-) for which we are doing predicate discovery, i.e., $\mathcal{N}_\lambda = \mathcal{N}_\lambda(s_+, s_-)$.

Definition 8. *A family of neighbor relations $\mathcal{N}_\lambda(s_+, s_-)$ is* admissible *if for every sample (s_+, s_-) with no missing types the best hypothesis is definable without any literals $\neg\lambda$ for which (s_+, s_-) is $\mathcal{N}_\lambda(s_+, s_-)$-unate.*

Neighbors for linear orders. We now give a suitable neighbor relation $\mathcal{N}_\lambda$ for the order structure $(\mathbb{Z}, <)$ of the integers (used in Sec. 6.1). Here, every logical type is equivalent to an ordered partition of the variables $\boldsymbol{x}$: equal variables form a class, and classes are ordered linearly. For example, the type generated by the formula $x_1 < x_2 = x_3 < x_4$ is equivalent to the ordered partition $\{x_1\} < \{x_2, x_3\} < \{x_4\}$. We shall manipulate types via two operations on *adjacent* classes: we can *swap* their position, or we can *merge* them into a single class. Given a sample (s_+, s_-), we say that two types *conflict* if one is positive, while the other is either negative or ambiguous. We are now ready to define $\mathcal{N}_\lambda$ ($[x]$ denotes the class of x):

1. $(\Phi, \Psi) \in \mathcal{N}_{x<y} \iff y < x \in \Phi$, $x < y \in \Psi$; swapping $[x]$ and $[y]$ in Ψ gives Φ; merging $[x]$ and $[y]$ in Ψ gives a type in conflict with Ψ.
2. $(\Phi, \Psi) \in \mathcal{N}_{x\neq y} \iff x = y \in \Phi$, $x \neq y \in \Psi$; merging $[x]$ and $[y]$ in Ψ gives Φ; swapping $[x]$ and $[y]$ in Ψ gives a type not in conflict with Ψ.
3. $(\Phi, \Psi) \in \mathcal{N}_{x\leq y} \iff (\Psi, \Phi) \in \mathcal{N}_{y<x}$ and $(\Phi, \Psi) \in \mathcal{N}_{x=y} \iff (\Psi, \Phi) \in \mathcal{N}_{y\neq x}$.

From an admissible family $\mathcal{N}_\lambda$ we obtain the resulting predicates by pruning literals $\neg\lambda$ for which the sample (s_+, s_-) is $\mathcal{N}_\lambda$-unate. We then search for a formula as described in Section 4.2. Because of missing types, the above method may prune literals which are required for finding the optimal hypothesis. Thus, our pruning approach is a heuristic that relies on good samples.

5 Sampling

When inferring specifications from data, it is important to ensure the data is of sufficient quality. In this section we present a sampling strategy based on a heuristic measure of the informativeness of a sample. Then, we give a simple algorithm for removing redundant observations from a sample. Finally, we describe a method to adaptively determine the sample size.

5.1 Type-aware sampling

Recall that our learning objective is to identify a hypothesis h that best approximates an unknown relation c. We have access to c only as a black-box: we can basically choose a tuple $\boldsymbol{u}$ and test whether it belongs to c or not, thus obtaining a sample of positive s_+ and negative s_- examples. We want to draw tuples such that the obtained sample (s_+, s_-) gives us most information about c. We shall consider one sample more informative than another if it rules out more candidate hypotheses from our class $\mathcal{H}$ of $\mathcal{L}(\boldsymbol{x})$-definable relations.

Measure. We can roughly quantify this notion of informativeness via logical types. Theorem 2 tells us that if the observed status of a type is not missing, we know how to classify all other tuples having that type. This suggests that we should increase the number of types observed in the sample. However, we also need to account for the case where c is not definable in our fragment. It might be that the observed status of some type is positive, but there exists some example that when added to our sample will switch this status to ambiguous, forcing us to reclassify all tuples having this type as negative. Let us call the *true status* of a type $\Phi(\boldsymbol{x})$ its status with respect to the "sample" $c_+ = \{\boldsymbol{u} \mid \boldsymbol{u} \in c\}$, $c_- = \{\boldsymbol{u} \mid \boldsymbol{u} \notin s_-\}$, i.e., when we add all possible tuples. Then our goal is to maximize the measure given by the number of types that have their true status (i.e., w.r.t. (c_+, c_-)) equal to their observed status (i.e., w.r.t. (s_+, s_-)).

Strategy. Of course, we cannot calculate this measure directly, as all we know is the observed status of a type. Thus, in the process of sampling we need to balance two conflicting factors: diversity and confidence. On one hand, we would like the sample to be as diverse as possible and to contain examples from many types. On the other hand, we would like to have high confidence that the observed status of every type matches its true status. To control this trade-off we assume two parameters: the total number m of examples to draw (discussed further in Section 5.3) controls the diversity, and the number k of examples to draw from a single type controls the confidence. Note that once the observed status of a type becomes ambiguous, we can stop drawing more examples from that type, as it will remain ambiguous. The strategy is summarized in Fig. 3.

SAMPLE$(\mathfrak{X}, \mathcal{L}, c, m, k)$
  $s_+, s_- \leftarrow \varnothing, \varnothing$
  while $|s_+| + |s_-| < m$
    choose an unambiguous $\mathcal{L}$-type $\Phi(\boldsymbol{x})$ at random
    while $\Phi(\boldsymbol{x})$ is unambiguous and $\#\{\boldsymbol{u} \in s_+ \cup s_- \mid \mathfrak{X} \models \Phi(\boldsymbol{u}/\boldsymbol{x})\} < k$
      choose $\boldsymbol{u} : \mathfrak{X} \models \Phi(\boldsymbol{u}/\boldsymbol{x})$ at random
      $s_+ \leftarrow s_+ \cup \{\boldsymbol{u}\}$ **if** $\boldsymbol{u} \in c$
      $s_- \leftarrow s_- \cup \{\boldsymbol{u}\}$ **if** $\boldsymbol{u} \notin c$
  return (s_+, s_-)

Fig. 3. Type-aware sampling from a relation c. The algorithm draws m examples in total, with at most k of them having the same logical type.

Sample size reduction. Once we have a sample we can optimize its size without reducing its informativeness by removing examples as long as we preserve the observed status of every type. We need to keep a single example of any type with a positive or a negative observed status, and two examples, one positive and one negative, from any type with an ambiguous observed status. This size reduction plays a role when we decide how many examples to draw (Section 5.3).

Guarantees. If we obtain a sample of maximal informativeness, i.e., in which every type has its observed status equal to its true status, then we are guaranteed to infer (the best) sound approximation. Such a sample always exists (as there are finitely many logical types) but obtaining it is often infeasible in practice. Thus, we can combine our black-box approach with white-box verification, e.g., [12].

5.2 Black-box interface

In our sampling algorithm we assumed that we sample the unknown relation c by generating tuples $\boldsymbol{u}$ and feeding them to a black-box which classifies them as positive or negative. However, this is not always the case in general (e.g, for commutativity, Section 6.2). We might be able to feed the black-box only a part $\boldsymbol{v}$ of $\boldsymbol{u}$, and only then obtain the rest $\boldsymbol{w}$ (i.e., $\boldsymbol{u} = (\boldsymbol{v}, \boldsymbol{w})$). In this case, we cannot control the type of the whole $\boldsymbol{u}$ but only of $\boldsymbol{v}$. This requires only a local change to the type-aware sampling algorithm: we choose a random type $\Phi(\boldsymbol{y}), \boldsymbol{y} \subseteq \boldsymbol{x}$ to generate $\boldsymbol{v}$, and then feed $\boldsymbol{v}$ to the black-box to obtain the complete example $\boldsymbol{u}$. From there on, we continue as before, considering the type of $\boldsymbol{u}$ and not $\boldsymbol{v}$.

5.3 Hypothesis stabilization

We now discuss how to reliably determine the sample size m. Instead of fixing the number m a priori, we draw new examples until the result of the formula search stabilizes. We realize this strategy by interleaving a sampling step with a formula search step. If the search gives the same result two consecutive times, then we return the discovered formula. The sampling begins with an initial number of m_0 examples and, at each subsequent step $i+1$ draws twice as many new examples as the previous step, i.e., $m_{i+1} = 2m_i$. After each sampling step, we run the search from scratch on all examples collected so far. As the search might take too long due to insufficient data, we run each search no longer than the time t_i taken for sampling at the same step. The total running time is not much longer compared to a single run over m examples: we restart the search $\Theta(\log m)$ times. The following theorem guarantees that we terminate:

Theorem 3. *If sample size reduction is applied, the required search time t'_i grows sublinearly with the time limit t_i, i.e, $t'_i = o(t_i)$.*

The theorem holds because t'_i grows with the reduced sample size which in turn is bounded by the number of types in the logical fragment. On the other hand, without sample size reduction we are unlikely to terminate, as then the number of examples for which we perform a search is proportional to m_i, and therefore we have that $t'_i = \Omega(t_i)$ (provided the sampling time t_i is linear in m_i).

6 Inferring commutativity specifications

In this section, we apply the approach discussed so far to the problem of learning commutativity specifications. Given a data structure (described via an abstract specification or a concrete implementation), our goal is to infer a commutativity specification for every pair of its methods.

6.1 Commutativity specifications

A commutativity specification states when two method invocations commute. Consider two executions that start in the same initial state σ:

$$m_1(\boldsymbol{u}_1)/\boldsymbol{v}_1 \ ; \ m_2(\boldsymbol{u}_2)/\boldsymbol{v}_2 \quad m_2(\boldsymbol{u}_2)/\boldsymbol{w}_2 \ ; \ m_2(\boldsymbol{u}_1)/\boldsymbol{w}_1 \tag{5}$$

If both executions end in the same state, $\boldsymbol{v}_1 = \boldsymbol{w}_1$, and $\boldsymbol{v}_2 = \boldsymbol{w}_2$, we say that the two invocations $m_1(\boldsymbol{u}_1)/\boldsymbol{v}_2$ and $m_2(\boldsymbol{u}_1)/\boldsymbol{v}_2$ *commute in* σ. A commutativity specification for m_1, m_2 is a formula $\varphi(\sigma, \boldsymbol{x}, \boldsymbol{y})$, which given a concrete initial state σ, arguments $\boldsymbol{x} = \boldsymbol{u}_1\boldsymbol{u}_2$, and return values $\boldsymbol{y} = \boldsymbol{v_1}\boldsymbol{v_2}$ describes if $m_1(\boldsymbol{u}_1)/\boldsymbol{v}_2$ and $m_2(\boldsymbol{u}_1)/\boldsymbol{v}_2$ commute. *Sound* specifications always imply that invocations commute (cf. the objective from Section 4.1). *State independent* ones do not mention the state σ, i.e., they have the form $\varphi(\boldsymbol{x}, \boldsymbol{y})$. Our work is able to learn optimal state independent approximations of state dependent specifications.

The logical fragment we will use consists of quantifier-free formulas built from integer and boolean variables, the predicates $=$ and $<$, and the standard boolean operations. Formulas in this fragment have an arbitrary boolean structure and are expressive enough to capture a large number of commutativity specifications.

6.2 Sampling for commutativity

We now describe an instantiation of the sampling approach from Section 5.1. Here, an example $\boldsymbol{u}_1\boldsymbol{u}_2\boldsymbol{v}_1\boldsymbol{v}_2$ consists of the combined arguments and return values of a pair of commuting or non-commuting method invocations. To produce an example, we generate random arguments $\boldsymbol{u}_1\boldsymbol{u}_2$ and an initial state σ, and then execute the methods in both orders. The outcome is two states σ_1 and σ_2, and two pairs of return values $\boldsymbol{v}_1\boldsymbol{v}_2$ and $\boldsymbol{w}_1\boldsymbol{w}_2$. If the states and the return values match, we have a *positive example* of commutativity. Otherwise, we have two *negative examples*. To compare states we assume that an abstract equality check, $\textsc{Equal}(\sigma_1, \sigma_2)$, is provided (naturally, we reason at the abstract level as opposed to the bit for bit concrete level).

7 Evaluation

We implemented our approach, and experimented with inferring commutativity specifications for method pairs of 21 data types. Some of these (e.g., accumulator, set, map, array list, 1-d tree, union-find) are well-known in the context

of commutativity [23,14,9,12,13,1,4] while others are variants of multiset, partial map, bit list, 6 variations of union-find, etc. We also selected classic data structures such as stack, queue and heap. For all data structures, we aimed to discover a state independent specification in the fragment of Section 6. We used the strategy from Section 5.3, obviating the need for setting the sample size in advance. We have set the initial sample size to 5000 by experimenting with `Set` and `Map`. We have used this value for all other structures. Our tool inferred the best approximation in all cases. For example, we inferred the following specification for the method pair $\texttt{unite}(a_1, b_1)/r_1$, $\texttt{unite}(a_2, b_2)/r_2$ of `UnionFind 6`, where $\texttt{unite}(a, b)/r$ unites the classes of a and b under the representative of the class of a, and also returns whether a modification was actually performed:

$$\big[(a_1 = a_2 \vee a_1 = b_2 \vee a_2 = b_1) \wedge r_1 \wedge r_2\big] \vee a_1 = b_1 \vee a_2 = b_2 \vee (\neg r_1 \wedge \neg r_2) \quad (6)$$

Data structure	Pairs	Size	Disj.	#Samples	#Types	Sampl.	Search	P.d.
`Set`	10	5	3	15 001	11/12	120	0.4	16.0x
`Map`	3	5	3	15 000	2492/4683	1.4s	600	33.3x
`MaxRegister`	3	3	3	15 000	20/75	80	3.2	10.9x
`1DTree`	6	9	3	15 000	31/75	240	2.6	11.9x
`IntProximityQuery`	10	5	3	15 000	19/75	130	1.4	18.6x
`RangeUpdate`	3	11	1	15 000	208/300	330	54	3.7x
`Accumulator`	3	1	1	15 000	3/3	35	0.06	3.3x
`Queue`	10	7	3	15 001	6/6	96	0.6	4.7x
`Stack`	10	7	3	15 001	6/6	83	0.3	3.3x
`MinHeap`	10	7	3	15 001	6/6	85	0.8	4.0x
`MultiSet`	10	9	3	15 000	51/75	170	3.3	2.3x
`PartialMap`	15	5	3	22 500	1987/4683	1.4s	440	43.2x
`UnionFind 1`	6	3	3	17 500	75/75	140	9.7	8.0x
`UnionFind 2`	6	5	3	32 500	75/75	290	31	2.7x
`UnionFind 3`	6	5	1	45 000	75/75	380	59	2.5x
`UnionFind 4`	6	3	3	30 001	247/300	520	110	2.5x
`UnionFind 5`	6	11	3	105 001	247/300	2.3s	470	1.6x
`UnionFind 6`	6	17	9	75 000	247/300	1.6s	550	36.4x
`BitTextEditor`	36	7	3	15 001	4/4	42	0.3	4.3x
`ArrayList`	28	7	3	145 001	1403/4683	2.4s	710	23.9x
`BitList`	120	19	9	75 000	150/150	920	210	19.5x

Fig. 4. Experimental results averaged over 8 runs. Times are in ms (unless indicated).

Fig. 4 summarizes our experimental results over 8 runs. For every data structure we show the averaged maximum over all method pairs. These numbers are: inferred formula size (**Size**), largest disjunct size (**Disj.**), number of drawn examples (**#Sample**), number of observed logical types vs. their upper bound (**#Types**), sampling time (**Sampl.**), formula search time (**Search**), and the search speedup achieved via predicate discovery (**P.d.**). The reduced sample size is proportional to the number of observed types.

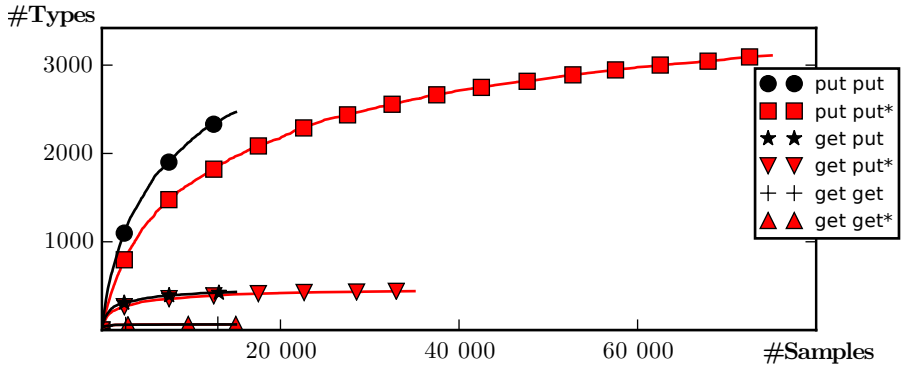

Fig. 5. Type-aware vs. random sampling (*) for 6 method pairs of `Map`. The method pairs produce tuples with 6 (`put`/`put`), 5 (`get`/`put`), and 4 (`get`/`get`) components.

Our results indicate that the approach is effective for learning non-trivial specifications. Further, stabilizing the inferred formula is a reliable way to determine a good sample size. By filtering examples of the same logical type, we significantly reduced the input sample size for the later formula search, and combining exhaustive and greedy search was fully sufficient for inferring all of the specifications. The results also show that predicate discovery dramatically reduces search time for more complex specifications. Finally, type-aware sampling successfully provided all observations necessary for inference.

We also compared type-aware with pure random sampling. Fig. 5 shows the number of examples vs. observed types of typical inference runs for `Map`. The curve of each run ends when formula stabilization was confirmed. We observe that: (i) type-aware sampling explored new types more quickly than pure random sampling, but only when sampling larger tuples (about 6 components in this particular case); and (ii) type-aware sampling stabilized the inferred formula much earlier (15 000 vs. 75 000 examples). In fact, in our experiments, we observed large variance in stabilization time when sampling purely at random.

8 Related work

There has been substantial interest in learning program invariants from concrete executions [5,8,7,21,20,6,16]. We evaluated several of these approaches, including [5,21,20,16]. Unfortunately, none of them could infer the necessary specifications and match our results.

Daikon [5] infers conjunctions of predefined templates that stay invariant during program execution. DIG [16] infers polynomial invariants over various algebras, e.g., min-plus and max-plus. Even though both tools support some form of disjunctive invariants, in our case they could only infer rather crude approximations to the target specification. A reason for this is that disjunctions abound in the context of commutativity [12]. That is why, in contrast to Daikon and DIG, our approach aims at learning free-form boolean expressions. That

said, our approach is not strictly better. Daikon scales well when the number of relations in the fragment increases, and DIG specializes in polynomial invariants.

Similarly to us, the approaches outlined in [21,20,6] also focus on fragments with rich support for disjunctions. However, their goal is to support program verification, and so they learn invariant properties that separate all positive from all negative examples in a given sample. This is not suitable for learning specifications, due to the fact that: (i) learning fails if a sample cannot be separated by a classifier, even though a good approximation exists (cf. (3) in Section 4.1); (ii) even if the sample can be separated, the inferred classifier can be too approximative to be useful, compared to the best approximation.

These points are especially true for the method in [21] which is tied to an expressive fragment (arbitrary boolean combinations of half-planes) and prone to overfitting, as we observed in our experiments with it. In [20] formula search is performed stochastically. This has the flexibility of supporting a variety of fragments, but can be highly sensitive to randomness, and can also have issues with convergence. We could not observe the approach terminating when inferring commutativity specifications over five or more variables.

Program synthesis methods are also applicable to our problem, i.e., we can simply ask for a program encoding the target specification. The technique in [11] synthesizes a program by querying a black-box input-output oracle. However, it also relies on a verification oracle, and in our setting this requirement can be too strong: the oracle needs to reason about the data type implementation, which in turn can be quite complex. The approach in [7] replaces the verification oracle with "universal examples" which distinguish every possible candidate specification. However, in the case of commutativity, we cannot directly query whether such an example is positive or negative, as a part of the example (the method return values) is actually generated by the query itself. Interestingly, our type-aware sampling can be seen as generalization of the "universal examples".

9 Conclusion

We presented a new "black-box" approach for learning specifications in the fragment of quantifier-free formulas over a finite number of relation symbols. The key insight is to treat uniformly the examples of the same logical type, i.e., examples that are indistinguishable by the logical fragment. Our approach introduces new techniques for obtaining small and informative samples, discovering relevant predicates, fast search procedure, and a way to adaptively determine sample size.

For our evaluation, we focused on automatically learning commutativity specifications. These are fundamental to various areas of computer science, yet are tricky to write manually. Our results indicate that the approach is practically effective – our tool quickly inferred non-trivial, useful commutativity specifications, beyond the reach of any existing work.

References

1. Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, 2011.
2. P.J. Cameron. *Oligomorphic Permutation Groups*. Cambridge Studies in Philosophy. Cambridge University Press, 1990.
3. C.C. Chang and H.J. Keisler. *Model Theory*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1990.
4. Dimitar Dimitrov, Veselin Raychev, Martin T. Vechev, and Eric Koskinen. Commutativity race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014.
5. Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007.
6. Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 69–87. Springer International Publishing, 2014.
7. Patrice Godefroid and Ankur Taly. Automated synthesis of symbolic instruction encodings from i/o samples. PLDI '12, pages 441–452, New York, NY, USA, 2012. ACM.
8. Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. From tests to proofs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 262–276, Berlin, Heidelberg, 2009. Springer-Verlag.
9. Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, 2008.
10. W. Hodges. *Model Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2008.
11. Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM.
12. Deokhwan Kim and Martin C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. PLDI '11, pages 528–541, New York, NY, USA, 2011. ACM.
13. Milind Kulkarni, Donald Nguyen, Dimitrios Prountzos, Xin Sui, and Keshav Pingali. Exploiting the commutativity lattice. *SIGPLAN Not.*, 46(6):542–555, 2011.
14. Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, 2007.

15. Kenneth L. McMillan. Relevance heuristics for program analysis. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, 2008.
16. ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to generate disjunctive invariants. ICSE 2014, pages 608–619. ACM, 2014.
17. B. Poizat. *A Course in Model Theory: An Introduction to Contemporary Mathematical Logic*. Universitext - Springer-Verlag. Springer New York, 2000.
18. J. Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, September 1978.
19. J. Rissanen. *Information and Complexity in Statistical Modeling*. Springer New York, 2010.
20. Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 88–105. Springer International Publishing, 2014.
21. Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and AdityaV. Nori. Verification as learning geometric concepts. In *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*. 2013.
22. V.N. Vapnik. *Statistical learning theory*. Adaptive and learning systems for signal processing, communications, and control. Wiley, 1998.
23. W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 37(12):1488–1505, 1988.