Optimal Learning of Specifications from Examples

Dana Drachsler-Cohen
Technion
Israel

Martin Vechev
ETH Zürich
Switzerland

Eran Yahav
Technion
Israel

Abstract
A fundamental challenge in synthesis from examples is designing a learning algorithm that poses the minimal number of questions to an end user while guaranteeing that the target hypothesis is discovered. Such guarantees are practically important because they ensure that end users will not be overburdened with unnecessary questions.

We present SPEX—a learning algorithm that addresses the above challenge. SPEX considers the hypothesis space of formulas over first-order predicates and learns the correct hypothesis by only asking the user simple membership queries for concrete examples. Thus, SPEX is directly applicable to any learning problem that fits its hypothesis space and uses membership queries.

SPEX works by iteratively eliminating candidate hypotheses from the space until converging to the target hypothesis. The main idea is to use the implication order between hypotheses to guarantee that in each step the question presented to the user obtains maximal pruning of the space. This problem is particularly challenging when predicates are potentially correlated.

To show that SPEX is practically useful, we expressed two rather different applications domains in its framework: learning programs for the domain of technical analysts (stock trading) and learning data structure specifications. The experimental results show that SPEX’s optimality guarantee is effective: it drastically reduces the number of questions posed to the user while successfully learning the exact hypothesis.

1. Introduction
Over the last few years, programming by example (PBE) techniques have proved useful in a variety of application domains (e.g., [8, 20, 21, 23, 27, 28, 34, 37, 41, 42, 45, 46]). The goal of PBE approaches is to synthesize a hypothesis (e.g., a program [34] or a specification [23]) desired by an end user from answers of questions the synthesizer poses to that user. Thus, a prime objective for any PBE approach is to reduce the burden placed on the user. This means that it is critical to reduce the number of questions the end user has to answer, ideally, to a minimum. While existing approaches have focused on PBE engines that learn hypotheses in interesting domains, there has been little work on guaranteeing that the target hypothesis can be discovered with a minimal number of user questions. In fact, existing PBE approaches (e.g., [34]) may ask the user an exponential number of questions even when a linear number of questions would have sufficed, limiting the practical benefits of PBE.

This Work We present SPEX, a new approach which ensures that for a given hypothesis space, the PBE engine will find the target solution with a minimal number of questions posed to the end user. To obtain this result we had to address two challenges: (i) define the hypothesis space and identify key properties on its shape; in turn, this allows our search procedure to detect hypotheses whose testing enables maximum pruning of the search space, and (ii) uncover the place in the search where involving the user is most beneficial and thus are guaranteed their involvement is reduced to a minimum.

Concretely, SPEX considers: (i) a hypothesis space defined by formulas over first-order predicates, and (ii) membership questions which are posed to the end user; we note that simple membership questions of various flavors are a staple of PBE approaches and are suitable for end users to answer. In our setting, membership questions are simple questions on concrete examples with the answer determining whether a predicate is relevant to the hypothesis we are trying to learn. Any synthesis problem which has the same hypothesis space and considers membership questions like SPEX can immediately benefit from our results.

SPEX Operation To learn a hypothesis, SPEX maintains a strict formula $\phi$ which logically implies the hypothesis we are trying to discover, and gradually attempts to relax it. At each step, SPEX: (i) considers a minimally relaxed hypothesis $\phi'$, (ii) generates a distinguishing input $e$ for $\phi$ and $\phi'$, (iii) asks the user for $e$’s correct output (via a membership question), and (iv) accordingly decides whether $\phi$ can be relaxed to $\phi'$. Unfortunately, this approach only works for independent predicates, since for dependent predicates step (ii) can fail. In fact, an even more restricted version of this approach (one that considers special cases of independent predicates) was proposed by [7]. They show that for other cases (e.g., dependent predicates), not only this approach fails but also that it cannot be accomplished with a polynomial number of questions.

This is where SPEX’s key technical novelty lies in: we show how to proceed at step (ii) even in the case of dependent predicates while guaranteeing we ask a minimal number of questions at step (iii). The beauty of this approach is that the number of questions is fully adaptable to the choice of predicates. That is, SPEX’s guarantees are not obtained by the general worst-case, but by the worst-case of the hypothesis space determined by the given set of predicates. Such guarantees are also known as the teaching dimension of the hypothesis space [25]. In addition, we present another result which further characterizes hypothesis spaces where the number of questions presented by SPEX is guaranteed to be linear (our result subsumes works that consider independent predicates).

In addition to minimality guarantees, we show our framework can accommodate interesting application domains: we expressed two different synthesis problems in SPEX: one where we learn technical patterns (i.e., programs used in stock trading) and one where we learn data structure specifications. We also show experiments demonstrating that SPEX significantly reduces the number of questions posed to the user, when compared to current approaches.

Main Contributions The main contributions are:
- SPEX: an interactive PBE system that learns a target hypothesis expressed by formulas over first-order predicates using a minimal number of membership questions.
- A result which states that for a certain useful class of hypothesis spaces, the number of examples presented to the user is linear.
In this section, we informally explain SPEX.

2. Overview

In this section, we informally explain SPEX. Formal details are provided in later sections.

2.1 Exact Learning from Examples

We address the problem of exact learning from examples (ELE). In ELE, a synthesizer (learner) tries to learn a concept by presenting examples for classification by a user (teacher). The user may also provide initial sets of positive and negative examples. Technically, given a domain of examples $D$, a concept $C \subseteq D$ is a subset of the example domain. An example $e \in C$ is referred to as a positive example, and an example $e \in D \setminus C$ is a negative example.

For instance consider:

- A domain $D = \{(x, y) \mid 0 \leq x \leq 4, 0 \leq y \leq 4\}$.
- A concept $C = \{(1, 1), (1, 2), (2, 1), (2, 2)\}$.
- A single initial example $(2, 2)$, which is positive.

Fig. 1 shows $D$ and $C$ (whose points are marked with bold points).

**The Challenge of Exact Learning** Exact learning algorithms have to learn a single concept. However, the initial examples the user provides are often consistent with many concepts. In our example, there are many subsets of $D$ that contain the example $(2, 2)$. To isolate the correct concept, exact learning algorithms are allowed to present questions to the user. This enables pruning concepts inconsistent with the new examples until a single concept remains.

**Membership Queries** Two common kinds of questions presented by exact learning algorithms are membership queries and validation queries. Membership queries present examples (elements from $D$) and ask whether they belong to the concept, while validation queries present concepts and ask whether these are the correct one.

Unfortunately, in many domains, validation queries are complex, error-prone, or impossible for the user to understand. For such domains, it is desirable to limit the questions to membership queries only, i.e., limit the setting to ELE: exact learning from examples.

**Predicate-defined Concepts** Often, concepts are conveniently specified using their features or properties. In this work, we assume that concepts are defined using arbitrary predicates. For example, we can express the concepts of $D$ from our running example with the following set of predicates:

$$S = P_x \cup P_y \cup \{x = y\}$$

where $P_x$ consists of predicates capturing intervals of $x$: $P_x \equiv \{\{0 \leq x \leq 2\}, \{1 \leq x \leq 3\}, \{2 \leq x \leq 4\}\}$ and $P_y$ is defined identically with respect to $y$. A concept satisfies the predicate $a \leq x \leq b$ if all its points $(x, y)$ satisfy that $x$ is between $a$ and $b$ and a concept satisfies the predicate $x = y$ if all its points take the form of $(x, x)$. Using these predicates, the concept we wish to learn (the one depicted in Fig. 1) is expressible by the formula:

$$\varphi_C = (1 \leq x \leq 3) \land (0 \leq y \leq 2) \land (1 \leq y \leq 3) \land (0 \leq y \leq 2)$$

**Finding the Correct Concept** To learn the concept formula $\varphi_C$, one can present examples to the user and prune the inconsistent concepts until a single concept remains. Many classical exact learning algorithms may be used only if the predicates are independent (see Section 7). Unfortunately, practical application domains often define specifications over abstract properties that may be dependent.

One approach [34] does address this challenge by iteratively picking two non-equivalent concepts, showing an input that distinguishes them, asking the user for the correct output, and pruning the inconsistent concepts accordingly. Unfortunately, since the two concepts are selected arbitrarily, the number of concepts pruned after a single question may be small, which can result in presenting an exponential number of questions to the user even when a linear number would suffice (see Section 6). In contrast, we leverage the partial-order between the concepts to pick two “close” concepts at each step. This guarantees that, overall, our approach always asks the user the minimal number of questions.

In the next two sections, we focus on learning concepts that can be expressed as conjunctions. We then show that learning disjunctions is dual and that learning conjunctions may be used to learn DNF formulas (and thus any specification).

2.2 Our Approach: a Guided Traversal in the Concept Space

Intuitively, we follow the classic techniques as described in Section 1, which use the structure of the formula to efficiently check if a candidate hypothesis can be relaxed by asking the user about a distinguishing input, namely an input whose output changes when relaxing the formula. However, in the general setting of predicate-defined concepts, some hypotheses may not have such distinguishing input.

The main idea of our approach is to leverage a partial order between concepts to find a minimal number of inputs that, together, act as the nonexistent distinguishing input.

Conceptually, SPEX traverses along the partially-ordered space of concepts consistent with the examples to find the correct concept formula $\varphi_C$. The partial-order is defined as follows: two formulas $\varphi, \psi$ satisfy $\varphi \leq \psi$ if $\varphi$ contains all of $\psi$’s predicates (and possibly additional predicates). This order induces a graph of the concepts consistent with the observed examples: the nodes are the consistent concepts, captured by formulas. There is an edge between two formulas $\varphi$ and $\psi$ if there is no other formula $\psi'$ satisfying $\varphi \leq \psi'$ $\leq \psi$. This graph is known as the version space [39].

Fig. 2 presents part of this graph corresponding to our running example: the bottom node shows the most specific concept formula, $\varphi_0$, which satisfies all predicates from $S$ satisfied by the initial positive example $(2, 2)$. Edges link $\varphi_0$ to more relaxed formulas (formulas with fewer constraints). We use $\varphi_{R_1 \ldots R_n}$ to denote the formula $\varphi_0$ without the predicates $R_1, \ldots, R_n$. The main idea of our approach is to leverage a partial order between concepts to find a minimal number of inputs that, together, act as the nonexistent distinguishing input.

**Our Approach** To learn the concept formula, one has to present examples to the user and prune the inconsistent nodes, until a single node remains. SPEX performs a guided traversal looking for a path to the concept formula $\varphi_C$. At each step, SPEX examines a specific node and its immediate neighbours to find a step towards $\varphi_C$.

An immediate neighbour of a node $\varphi$ is $\varphi^R$, the formula where a single predicate $R$ has been dropped. Note that $\varphi$ logically implies $\varphi^R$. Examining $\varphi^R$ means checking whether $\varphi_C$ is reachable from it, i.e., whether $R$ is in $\varphi_C$. To check if $R$ is part of $\varphi_C$, SPEX looks for a distinguishing input between $\varphi$ and $\varphi^R$. Since $\varphi \leq \varphi^R$, this means finding an example satisfying $\varphi^R \land \neg \varphi$.

If an example $e$ such that $e \models \varphi_R \land \neg \varphi$ exists, its classification enables progress: if $e$ is a positive example (i.e., $e \models \varphi_C$), SPEX proceeds towards $\varphi_R$ (and prunes the rest of the space), otherwise, $\varphi_R$’s sub-graph is pruned.
Challenge 1. When \( \varphi^R \land \neg \varphi \) is not satisfiable, how can one obtain alternative distinguishing inputs that enable to classify \( R \)?

A naive solution to this challenge is to examine every child of \( \varphi^R \): if all have distinguishing inputs with \( \varphi \) and one of these inputs is a positive example (satisfies \( \varphi_C \)), the traversal proceeds towards this child and prunes the rest of the space; otherwise, if all inputs are negative examples, \( \varphi^R \)'s sub-graph is pruned. However, it is not guaranteed that the children necessarily have distinguishing inputs, in which case their children must be examined similarly. While this solution is correct, it is wasteful in the number of questions. This leads us to the second challenge we address:

Challenge 2. How can one obtain a minimal number of alternative distinguishing inputs?

To present a minimal number of questions, we show that instead of examining all children of \( \varphi^R \), it suffices to examine a subset of children. This subset is the set of predicates in \( \varphi_R \) "preventing" distinguishing inputs with \( \varphi \). That is, predicates preventing the formula \( \varphi^R \land \neg \varphi \) from being satisfiable. Such predicates are known as the unsat core of the formula.

We prove that if there is no distinguishing input for \( \varphi^R \) and \( \varphi \), it suffices to compute an unsat core of the above formula and consider only the children belonging to the unsat core. If the computed unsat cores are guaranteed to be minimal, we prove that a minimal number of questions is presented. Though finding minimal unsat cores in general theories is EXPSPACE-complete, there are approaches to compute small unsat cores (e.g., [15]), and in some theories (such as the ones exemplified in this work), minimal unsat cores can be computed.

### 2.3 A Running Example

We now demonstrate SPEX on the concept defined in Section 2.1. Given the initial user-provided example \((2,2)\), SPEX first computes the most strict consistent concept, \( \varphi_0 \) (which implies every consistent concept), which is \( \bigwedge_{l \in S_0=\{(0,0)\}} l \), that is:

\[
\varphi_0 = (0 \leq x \leq 2) \land (1 \leq x \leq 3) \land (2 \leq x \leq 4) \land (0 \leq y \leq 2) \land (1 \leq y \leq 3) \land (2 \leq y \leq 4) \land (x = y)
\]

**Distinguishing Inputs** After constructing \( \varphi_0 \), SPEX looks for a predicate \( R \) that can be classified with a single example. Unfortunately, none of its immediate neighbours in the concept graph has a distinguishing input with \( \varphi_0 \). For example, for \( R = (0 \leq x \leq 2) \therefore \) there is no distinguishing input, because such an input has to satisfy the (unsatisfiable) formula, \( \varphi^R_0 \land \neg \varphi_0 \), which is simplified to:

\[
\psi_{S_0}^{[0 \leq x \leq 2]} = \bigwedge_{l \in Q_0 \setminus \{R \}} l \land \bigwedge_{l \in R \setminus S_0} \neg l
\]

In the following, we use the notation \( \psi_{S_0}^{[0 \leq x \leq 2]} \) to refer to the formula satisfying the predicates in \( Q_0 \) and not in \( R \) and the negations of the predicates in \( R \), that is: \( \psi_{Q_0}^{[0 \leq x \leq 2]} = \bigwedge_{l \in Q_0 \setminus \{R \}} l \land \bigwedge_{l \in R \setminus S_0} \neg l \).

Back to our example, the formula \( \psi_{S_0}^{[0 \leq x \leq 2]} \) is unsatisfiable due to the dependency between the predicates \( 0 \leq x \leq 2) \), \( 0 \leq y \leq 2 \), and \( x = y \). If there were such examples they would satisfy that \( x \) is greater than 2, \( y \) is at most 2, and \( x \) equals \( y \), which clearly cannot be satisfied together.

**Finding Alternative Distinguishing Inputs** To find alternative satisfiable formulas, SPEX computes an unsat core of the above formula. For each predicate \( R \) in the unsat core (except for the one at hand, \( R \)), SPEX constructs a formula that negates \( R \), in addition to \( R \). If some of these formulas are still unsatisfiable, SPEX repeats this process, computes a new unsat core, and generates a set of formulas from it. Finally, SPEX presents the user an example for each of these formulas. In our example, SPEX computes an unsat core of \( \psi_{S_0}^{[0 \leq x \leq 2]} \), which is \( \{(0 \leq x \leq 2), (0 \leq y \leq 2), x = y\} \) and generates the formulas \( \psi_{S_0}^{[0 \leq x \leq 2], (x=y)}, \psi_{S_0}^{[0 \leq x \leq 2], (0 \leq y \leq 2)} \). These formulas are satisfiable by (3, 2) and (3, 3) respectively. Therefore, SPEX presents these points to the user.

**Inferring Classifications from the Alternative Formulas** If one of these examples, corresponding to \( \psi_{S_0}^{[0 \leq x \leq 2]} \), is classified by the user as a positive example, then none of the negated predicates is part of \( \varphi_C \), and thus \( R, R_1, \ldots, R_k \) are dropped from the current formula. However, if all of them are negative, then it is only guaranteed that the predicate at hand, \( R \), is part of the correct concept \( \varphi_C \). For example, in our example, both points (3, 3) and (3, 2) are negative, and thus SPEX infers that \( 0 \leq x \leq 2 \) is part of \( \varphi_C \). Note that although these formulas negate additional predicates, \( x = y \) and \( 0 \leq y \leq 2 \), these cannot be classified
at this point. Indeed, eventually \( x = y \) will be dropped, while
\((0 \leq y \leq 2)\) will be part of \( \varphi_C \). However, the next step of SPEX,
which considers the predicate \((2 \leq x \leq 4)\), infers differently.
As before, there is no distinguishing input for \( \varphi_0 \) and \( \varphi_0 \)
\((2 \leq x \leq 4)\) (i.e., \( \psi_3 \)) is unsatisfiable). Therefore, SPEX considers the unsat
\( 1 \leq x \leq 3 \) and \( x = y \)
\( (x = y) \) and generates the relaxed formulas \( \psi_3 \) and \( \psi_3 \).
Both formulas are satisfiable, by \((1, 2)\) and \((1, 1)\), and both are positive.
However, this time after the first example, \((1, 2)\), is presented to
the user, SPEX infers immediately (without presenting \((1, 1)\))
that \((2 \leq x \leq 4)\) and \( x = y \) are not in \( \varphi_C \), and thus it updates
the current candidate formula to \( \varphi_0 \). In the next step, SPEX
looks for distinguishing inputs from the new candidate formula,
and so it constructs \( \psi_3 \cdot \psi_3 \cdot \psi_3 \cdot \psi_3 \), namely it ignores the
predicates \((2 \leq x \leq 4), x = y \) as they no longer affect the
classifications. We note that in fact \((2 \leq x \leq 4)\) is implied by the
predicate \((0 \leq x \leq 2)\), and thus is classified as redundant by SPEX
immediately after learning \((0 \leq x \leq 2)\) – we ignore this step here
to exemplify how SPEX classifies a predicate as not part of \( \varphi_C \).

2.4 SPEX Extensions

We use the logic described for learning formulas over conjunctions,
to learn other concept classes: disjunctions and DNFs.

D-SPEX The disjunctive variation of SPEX is dual to the conjunctive.
While the conjunctive variation, C-SPEX, generalizes from the
positive examples and learns which constraints must be met by ex-
amples in the concept, D-SPEX generalizes from the negative ex-
amples, and learns which constraints eliminate examples from being
part of the concept. Fig. 3 visually demonstrates the difference
between the classes: C-SPEX learns a consecutive region in the
concept space that contains all positive examples, while D-SPEX
learns the same region only for the negative examples.

Gen-SPEX Gen-SPEX learns more complex formulas that can cap-
ture general concepts, in which there is no single consecutive reg-
ion for the positive examples or the negative examples (as illustrated in Fig. 3).
Ideally, to learn such concepts, Gen-SPEX would
simply invoke C-SPEX to learn a conjunction for each region (inde-
pendently) and then return the disjunction over these conjunctions.
However, there are two main issues with this approach that Gen-
SPEX has to address:

- How to guarantee that every region has been covered? It cannot assume
that the user provides enough examples.
- How to handle intersecting regions? C-SPEX may over-
generalize such regions, resulting in an incorrect specification.

To address the first challenge, Gen-SPEX maintains two formu-
las: (i) \( \varphi_P \), satisfied by the positive examples, and (ii) \( \varphi_N \), satisfied
by the negative examples. While there is an example not satisfying
any of them, Gen-SPEX asks the user for the example’s classification,
and accordingly adds a conjunction to \( \varphi_P \) or \( \varphi_N \).

To address the second challenge, we first identify the pitfall of
employing C-SPEX as-is: C-SPEX relies on the fact that every ex-
ample “outside” of the (single) region is classified as negative ex-
ample. However, this is not true for Gen-SPEX as examples “out-
side” of a certain region may be classified as positive if they belong to
a different region. Since C-SPEX generates examples that are
“close” the current candidate hypothesis, it can learn regions
that are “sufficiently apart” from others. For regions that are “too
close” or even intersect, over-generalization may still occur in two cases:
(i) if an example is in the intersection of two regions, and
(ii) if several examples are generated to classify a predicate \( R \) (due
to dependency between predicates), which leads to removing \( R \)
if one of them is positive. In the first case, C-SPEX will not be able to
isolate the regions and will return an over-generalization contain-
ing them, and thus Gen-SPEX has to detect this and ignore the
conjunction. In the second case, Gen-SPEX avoids over-generalization
by modifying C-SPEX to examine all examples and eliminate the
negative examples with a disjunction for each negative example.
While this results in a formula which is not a DNF (as its conjunc-
tion may be over disjunctions and not only literals), it can be easily
transformed to a DNF, and thus we refer to learning such formulas
as learning a DNF. We provide further details in Section 5.

3. Exact Learning from Examples

In this section, we define formally the problem of learning an exact
specification from examples.

Specifications We consider three types of specifications: (i) DNF
specifications where the formula is in disjunctive normal form,
(ii) conjunctive specifications, a restricted case of DNF where there
is a single cube (i.e., there are no disjunctions), and (iii) disjunctive
specifications, a restricted case of DNF where each cube contains
a single literal (i.e., there are no conjunctions). The specifications
are defined over arbitrary predicates, defined over the example domain.
We next formally define them.

DEFINITION 1 (A DNF Specification). Let the example domain \( D \)
be a set and \( S \) be a set of predicates over \( D \), namely \( \forall R \in S, \exists n \in \mathbb{N} : R \subseteq D^n \). A DNF specification is a formula
\( \varphi(d) = \bigvee_{A \subseteq R, \neg R \mid R \subseteq S} l(d) \), where each \( A \) is a subset of literals
over \( S \), that is \( A \subseteq \{ R, \neg R \mid R \subseteq S \} \).

DEFINITION 2 (A Conjunctive Specification). A conjunctive
specification is a DNF specification with a single cube, namely
\( \varphi(d) = \bigwedge_{A \subseteq R, \neg R \mid R \subseteq S} l(d) \) where \( A \subseteq \{ R, \neg R \mid R \subseteq S \} \).

DEFINITION 3 (A Disjunctive Specification). A disjunctive
specification is a DNF specification where each cube has a single literal,
namely \( \varphi(d) = \bigvee_{A \subseteq R, \neg R \mid R \subseteq S} l(d) \) where \( A \subseteq \{ R, \neg R \mid R \subseteq S \} \).

Exact Learning of Specifications We address the problem of exact
learning of specifications. In exact learning, the goal is to pre-
cisely classify every example in the domain \( D \), without necessar-
ily explicitly seeing every input-output example. We consider the
teacher-student model where the student (i.e., the algorithm) can
ask the teacher (i.e., the oracle or user) only membership questions,
that is ask for the output of a given input. We further allow the
teacher to provide some initial positive and/or negative examples,
however the teacher need not provide examples, and in any case
the student obtains the examples it needs by interacting with the
teacher. We next formally state this (interactive) learning problem.

DEFINITION 4 (Exact Learning from Examples (ELE)). Let \( D \) be a
domain, \( S \) be a set of predicates over \( D \), \( \varphi_C \) be an unknown
specification over \( S \) (to be discovered), \( E_P, E_N \subseteq D \) be initial
sets of positive and negative examples (i.e., \( \forall d \in E_P, \varphi_C(d) \) and

Figure 3. Illustration of the concept spaces.
We refer to the above problem as a C-ELE, D-ELE, or DNF-ELE if the specification is conjunctive, disjunctive, or DNF (resp.).

**ELE’s Complexity Class** ELE was extensively studied and in particular it was shown to be EXPTIME for the special case where the domain is a set of boolean vectors and the predicates are monomials over the vectors [7]. This implies that our general setting of DNF-ELE, which does not restrict the domain or the predicates, is also EXPTIME. The work of [7] also implies that D-ELE is EXPTIME since it can be seen as a special setting of DNF-ELE where the domain is the boolean vectors and S contains conjunctions. To show that C-ELE is also EXPTIME we prove the following claim:

**Claim 1. Let**
- $D$ be a set of boolean vectors: $\{(x_0, ..., x_k) \mid \forall i, x_i \in \{0, 1\}\}$.
- $S$ be a set of conjunctions: $\{(x_0 \lor \neg x_j), (x_0 \lor \neg x_j) \mid 1 \leq j \leq k\}$.
- $E_P = E_N = 0$.

*For any ELE algorithm there is a conjunctive specification $\varphi_C$ which presents $\Omega(2^{|S|})$ membership queries to the oracle.*

Intuitively, the specification is a CNF, which here is equivalent to learning DNF. Proof is provided in Appendix A.

4. **The C-SPEX and D-SPEX Algorithms**

In this section, we present our exact learning algorithm for the restricted classes of conjunctive and disjunctive specifications. We begin with the high-level algorithm, then show the algorithm itself to the C-ELE and D-ELE algorithms, and finally prove that C-SPEX and D-SPEX generate a minimal number of examples and that for a useful class this number is linear.

4.1 **C-SPEX and D-SPEX in a Nutshell**

In this section, we present the pseudo code of SPEX and discuss the main differences between its conjunctive and disjunctive variations.

**The Guided Traversal** Algorithm 1 shows the pseudo code of SPEX’s guided traversal. The algorithm takes as arguments the set of predicates $S$ and the initial set of positive and negative examples $E_P$ and $E_N$ (which may be empty). It begins by constructing the most specific formula and storing its literals in $S_0$, which provides the “alphabet” of the concept formula $\varphi_C$ to be learned.

The guided traversal classifies each literal in $S_0$ as part of $\varphi_C$ or not. It maintains two sets, $S_P$ and $S_N$, storing the predicates classified so far as part of $\varphi_C (S_P)$ or not ($S_N$). SPEX iteratively classifies literals until all are at $S_P$ or $S_N$. At each step, it invokes a function that returns a literal, which minimizes the number of examples needed for classification, and its classifying examples. It then gradually asks the oracle for their output, until it can classify the literal. If the literal was classified to $S_P$, literals that are implied by $S_P$ and the new literal are classified to $S_P$, too. Finally, SPEX generates $\varphi_C$ by constructing a conjunctive or disjunctive formula from $S_P$, cleans it by removing redundant literals, and returns it.

**C-SPEX and D-SPEX** The above pseudo code is the framework of variations the C-SPEX and D-SPEX. While the framework is identical, the two variations are not identical but dual: C-SPEX generalizes from positive examples, whereas D-SPEX generalizes from negative examples. We next informally present the differences:

- **Initialization:** both variations initialize $S_0$ such that the conjunction (in C-SPEX) or disjunction (in D-SPEX) over its elements implies $\varphi_C$. In C-SPEX, $S_0$ contains the literals from $S$ satisfied by all positive examples. In D-SPEX, $S_0$ contains the negations of literals from $S$ satisfied by all negative examples.

### Algorithm 1: SPEX Pseudo Code $(S, E_P, E_N)$

1. $S_0 = \text{get literal set of the most specific hypothesis from } S, E_P, E_N$
2. $S_P = S_N = \emptyset$
3. while $S_P \cup S_N \not\subseteq S_0$ do
   4. $l, e$ = find a literal that requires a minimal number of examples
   5. Get feedback on $e$s until $l$ can be classified to $S_P$ or $S_N$
   6. if $l \in S_P$ then add to $S_P$ all literals $l$ implied by $S_P$ and $l$
   7. Construct $\varphi_C$ by collecting all the literals in $S_P$
   8. Clean $\varphi_C$ by removing implied literals
   9. return $\varphi_C$

- Constructing examples: in both variations, the goal is to learn which literals from $S_0$ are part of $\varphi_C$, and thus to classify the literals, SPEX constructs distinguishing inputs, however those are constructed differently. In C-SPEX, to infer whether a literal $l$ in $S_0$ is in $\varphi_C$, a distinguishing example for the conjunction over $S_0 \setminus S_N$ (the most strict hypothesis consistent with the current positive examples) and the same hypothesis only without $l$ satisfies all the literals in $S_0 \setminus S_N$ (but $l$) and $\neg l$. If such example is positive, $l$ is not in $\varphi_C$, otherwise it is. Intuitively, correctness follows because if the example $e$ is positive, i.e., $e \models \varphi_C$, but does not satisfy $l$, i.e., $e \not| l$, $l$ cannot be part of the conjunction $\varphi_C$. In contrast, in D-SPEX, to infer whether a literal $\neg l$ is in $\varphi_C$, a distinguishing example satisfies $\neg l$ and none of the other literals in $S \setminus S_N$. If such example is negative, $\neg l$ is not in $\varphi_C$, otherwise it is. Intuitively, correctness follows because if the example $e$ is negative, i.e., $e \not| \varphi_C$, then if $\varphi_C$ would have contained $\neg l$, $e$ should have been a positive example since it satisfies the disjunction. We note that in case there are no such distinguishing inputs, SPEX considers alternative formulas whose distinguishing inputs enable to infer the classification similarly (as will be described later).

- Implications: in C-SPEX, if a literal $l$ is added to $S_P$, any other literal implied by $S_P \cup \{l\}$ is also in $S_P$ (since any positive example satisfies $S_P$ and $l$, and thus this literal). In D-SPEX, if a literal $\neg l$ is added to $S_P$, any other literal that is implied by the disjunction is added to $S_P$ (since there are no positive examples satisfying the disjunction but not this literal) and any other literal that implies the disjunction is added to $S_P$ (since there are no positive examples satisfying this literal but not the disjunction and adding it to the disjunction does not strengthen or relaxes the disjunction). Removing the implied literals is only required to complete the classification of each literal in $S_0$, and these literals do not affect the final formula $\varphi_C$, since it is cleaned from redundant literals.

4.2 **The SPEX Algorithm**

In this section, we present the actual algorithm of SPEX, shown in Algorithm 2, which instantiates the template of Algorithm 1. SPEX takes as arguments the set of predicates $S$, the initial sets $E_P$ and $E_N$ of positive and negative examples, and the isCon flag indicating whether to learn a conjunctive or a disjunctive specification.

To instantiate it, two operations are required, init and implied, implemented differently by C-SPEX and D-SPEX. SPEX begins with initializing $S_0$ to the set of all possible literals that $\varphi_C$ may contain (using init) and the two literal sets, $S_P$ and $S_N$, to the empty sets. Then, SPEX iteratively generates examples to classify literals in $S_0$ until all are in $S_P$ or $S_N$ (Lines 3–10). At each iteration, SPEX invokes getMinMaxLiteralNEExamples that picks the next literal to classify $l$ and returns $l$ along with the examples that imply its classification. Each example is accompanied with a set of literals $Rs$ containing the relaxed literals (in C-SPEX this means literals that are negated, and in D-SPEX this means literals for which no classification has been made yet).
that are not negated). Then, SPEX gradually iterates the examples to classify $l$ (Lines 6–9). First, it obtains their output (Line 7) using `askUser` (whose code is omitted) that gets the example’s classification (positive or negative) either from the available examples or, if the example is new, from the oracle and adds the new example to $E_P$ or $E_N$ accordingly. After obtaining the output, SPEX classifies according to its duality. In C-SPEX, if the example is positive, this indicates that $l$ and the rest of the literals in $R_s$ (which includes $l$) are not in $\varphi_C$, and thus all are added to $S_N$, and SPEX continues to classify the next literal. Otherwise, if all the examples are negative, this indicates that $l$ is in $\varphi_C$, and thus the set of literals implied by $S_P$ and $l$ (which includes $l$) is computed using `implied` and added to $S_P$ (Line 10). D-SPEX is dual: it adds $l$ to $S_P$ if one of the examples is negative, or adds $l$ to $S_P$ if all the examples are positive. Finally, SPEX generates $\varphi_C$ from $S_P$ and cleans it by removing implied literals (Lines 11–13).

### 4.3 Computing the Next Literal to Classify

In this section, we present `getMinLiteralNExamples` (Algorithm 3), abbreviated to `getMin`, that picks the next literal to classify and returns it along with the examples implying its classification. Each example is accompanied with the set of its picking the literals. We first describe how to compute for a given literal a minimal set of examples that implies its classification, and then describe `getMin` that finds a literal whose example set is of minimal size.

**Computing the Minimal Example Set of a Literal**

Ideally, a literal $l$ can be classified using a single example. To check if there is such an example, `getMin` constructs a formula $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$, where $R_s = \{l\}$, “isolating $l$’s effect”:

- In C-SPEX, $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$ is the conjunction of $\neg l$ and the literals in $S_N$, except for $l$ and the literals classified to $S_N$.
- In D-SPEX, $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$ is the conjunction of $l$ and the negations of the other literals in $\varphi_C \setminus S_N$.

If $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$ is satisfiable, any example in $D$ satisfying it is an example whose classification implies $l$’s classification. If $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$ is unsatisfiable, but there is a single way to relax $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$ (by removing specific literals), then similarly any example satisfying the relaxed formula can serve as the single classifying example.

However, if there are multiple ways to relax $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$ (e.g., $l_1$ or $l_2$ may be removed from it), `getMin` has to consider every relaxed formula and generate an example for each (it does not necessarily mean that all will be presented to the oracle). To find a minimal number of relaxed formulas, `getMin` uses UNSAT cores (i.e., unsatisfiable sets of literals from the formula) that may be computed from the unsatisfiable formula $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$, for example using an SMT-solver (e.g., [17]). The UNSAT cores must contain $l$ (because $S_N \setminus S_N$ is satisfiable) and some literals from $S_N$ not in $S_P$ (otherwise, $l$ is implied from $S_P$, but then it would have been removed by SPEX before invoking `getMin`). Each of these literals is a possibility to consider except for $l$ and literals from $S_P$ (as these dominate the examples’ classification, regardless of $l$’s classification). Thus, for each `getMin` generates a formula extending $R_s$ with this literal. If the new $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$ is still unsatisfiable, an additional core is computed and new relaxed formulas replace the former relaxed formula. Since the relaxed formulas are uniquely identified by their relaxed literals, `getMin` maintains the relaxed literal set of each relaxed formula, which are stored in sets.

**Computing the Minimal Literal**

To find a literal requiring a minimal number of examples, `getMin` sets a bound on this number with the variable `max` and increases it only if all literals require more examples (Line 1). After fixing `max`, every unclassified literal is checked whether it can be classified using at most `max` examples (Line 2). To this end, `getMin` initializes `sets` to contain the initial `Rs` set, $\{l\}$, (Lines 3) and replaces each literal whose formula $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$ is unsatisfiable with its relaxed sets, as previously described. Then, a loop updates `sets` until: (i) every $Rs$ satisfies that $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$ is satisfiable, or (ii) the size of `sets` exceeds `max` (Lines 4–7). To determine whether for a given $Rs$, $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$ is satisfiable, `getMin` uses an SMT-solver (Line 5). If $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$ is unsatisfiable, an UNSAT core is obtained from the SMT-solver (we also reduce it to be minimal by removing redundant literals, we omit this part from the code), and `sets` is updated to exclude $R_s$ and include all the sets consisting of $R_s$ and a single literal from the unsat core that is not in $S_P$ or $R_s$ (Lines 5–6). If the extension of `sets` results in exceeding `max`, the loop terminates (Line 7) and the next literal is examined (Line 8). Otherwise, $l$ is returned along with the set of pairs consisting of the `Rs` sets and their corresponding examples. The examples are obtained from the SMT-solver, denoted $\text{ex}(\varphi)$.

### 4.4 Implementing C-SPEX and D-SPEX

In this section, we describe the operations that instantiate C-SPEX and D-SPEX: (i) `init`, (ii) imply, and (iii) $\psi_{\varphi_C \setminus \{l\}}^{R_s \setminus S_N}$ listed in Table 1.

**C-SPEX**

This variation implements these operations as follows:

- `C-init` returns the conjunction over all literals in $S$ satisfied by all positive examples. This formula implies the specification $\varphi_C$: literals not in it are not satisfied by one of the positive examples and thus are not in $\varphi_C$.
- `C-implied` returns the set of literals implied by the conjunction of $S_P$ and the literal $l$.

1 More precisely, in C-SPEX the core actually contains negations of literals from $Rs$, and in D-SPEX negations of literals from $Sp$, but the core is cleaned from these literals without the negations.
Table 1. The template functions of the two SPEX variations.

| Variation | init(S, E_P, E_N) | implied(S_P, l) | \(\psi_{RS}^E\) |
|-----------|------------------|-----------------|------------------|
| C-SPEX    | \(\{l \in S \mid \forall e \in E_P.e \models \psi\}\) | \(\{l \mid \bigforall_{g \in S_P \cup \{l\}} q \models \bigforall_{g \in S_P \cup \{l\}} q\}\) | \(\bigwedge_{Q \in Rs} \neg Q \land \bigwedge_{Q \in Os} \neg Q\) |
| D-SPEX    | \(\{l \in S \mid \forall e \in E_N.e \models \psi\}\) | \(\{l \mid \bigforall_{g \in S_P \cup \{l\}} q \models \bigforall_{g \in S_P \cup \{l\}} q\}\) | \(\bigwedge_{Q \in Rs} Q \land \bigwedge_{Q \in Os} \neg Q\) |

\[E\]

- \(\psi_{RS}^E\) returns the conjunction of: (i) the negations of the literals in Rs, i.e., l and the literals relaxed to obtain a satisfiable formula, and (ii) the other literals in \(S \setminus S_N\). Literals in \(S_N\) may be determined arbitrarily as they do not affect the classification: SPEX observed positive and negative examples satisfying them.

**THEOREM 1.** Given \(D, S, \varphi_C\) an unknown conjunctive specification over \(S\), and initial positive and negative examples \(E_P\) and \(E_N\). Let C-SPEX be SPEX with \(C\)-init, \(C\)-implied, and \(C\)-\(\psi_{RS}^E\). C-SPEX is a C-ELE algorithm, i.e., it learns a conjunctive specification \(\varphi\) over \(S\) such that: \(\forall d. \varphi_C(d) \leftrightarrow \varphi(d)\).

Proof is provided in Appendix A.

**D-SPEX** This variation implements the operations as follows:
- \(D\)-init returns the conjunction over the negations of literals in \(S\) satisfied by all negative examples. This formula implies the specification \(\varphi_C\); a negation of a literal not in it is satisfied by a negative example and thus is not in \(\varphi_C\).
- \(D\)-implied returns the set of literals implied by or implying the disjunction of \(S_P\) and the literal \(l\).
- \(D\)-\(\psi_{RS}^E\) returns the conjunction of: (i) the literals in \(R_s\), checked whether they sufficient to satisfy the disjunction, and (ii) the negations of the literals in \(S \setminus S_N\), which include \(S_P\) that contains literals known to be sufficient to satisfy the disjunction.

**THEOREM 2.** Given \(D, S, \varphi_C\) an unknown disjunctive specification over \(S\), and initial positive and negative examples \(E_P\) and \(E_N\). Let D-SPEX be SPEX with \(D\)-init, \(D\)-implied, and \(D\)-\(\psi_{RS}^E\). D-SPEX is a D-ELE algorithm, i.e., it learns a disjunctive specification \(\varphi\) over \(S\) such that: \(\forall d. \varphi_C(d) \leftrightarrow \varphi(d)\).

Proof is provided in Appendix A.

### 4.5 Complexity Analysis

In this section, we present the theorem stating that SPEX asks the minimal number of questions and characterize when this number is linear. Proofs are in Appendix A.

**THEOREM 3.** Given \(D\), a set of literals \(S\) of size \(n\), and initial examples \(E_P\) and \(E_N\). If C-SPEX or D-SPEX present \(\mathbb{O}(f(n))\) questions for some function \(f\), any C-ELE or D-ELE algorithms present \(\mathbb{O}(f(n))\) questions.

**THEOREM 4.** If at any iteration of C-SPEX or D-SPEX there is a literal \(l \in S_0\) such that \(\psi_{RS}^E\) or \(\psi_{RS}^E\) (resp.) are satisfiable, C-SPEX and D-SPEX complete in a linear number of questions.

Intuitively, if this condition is satisfied, at each invocation of getMinLiteralNExamples there is a literal \(l\) for which the formula \(\psi_{RS}^E\) is satisfiable, and thus a single example is generated. This bounds the number of examples to \(|S_0|\), namely linear.

#### 4.5.1 Classes Learned with a Linear Number of Questions

This section focuses on a useful class of predicates: predicates that pertain only to the binary relative comparison of values \(x, y\), i.e., \(x < y, x \leq y, x = y, x \neq y\). For this class, the conditions of Theorem 4 are satisfied for C-SPEX, namely it learns with a linear number of questions.
in $S_N$ are satisfied), and thus the second concept’s literals are also added to $S_N$. To exclude over-generalizing conjunctions, Gen-SPEX invokes the overgen operation, described in Section 5.3.

### 5.1 The Gen-SPEX Algorithm

Gen-SPEX (Algorithm 4) learns two formulas, one that generalizes the positive examples and the other that generalizes the negative examples. Each of these formulas is a disjunction over a set of conjunctions capturing a single region. The conjunctions are over literals or conjunctions of literals, and they are learned using a slightly modified version of C-SPEX (described in Section 5.2).

Gen-SPEX maintains the formulas’ set of conjunctions, stored in $C_P$ and $C_N$, which are initially empty. It begins by examining the initially provided examples in $E_P$ and $E_N$ and while they contain examples not satisfied by any of the conjunctions in $C_P$ or $C_N$, it invokes C-SPEX, checks if the resulted conjunction is an over-generalization (using overgen), and if not, adds the conjunction to $C_P$ or $C_N$, respectively (Lines 2–7). We note that since $C_N$ is satisfied by the negative examples, C-SPEX switches the user’s classifications when generalizing a negative example, and it is invoked with $E_N$ as $E_P$ and $E_P$ as $E_N$.

Then, while there is an example not satisfied by any of the conjunctions, Gen-SPEX obtains such example from the SMT-solver, asks the user for its classification, invokes C-SPEX, and, if the resulted conjunction is not an over-generalization, adds the new conjunction to $C_P$ or $C_N$ (Lines 8–15).

After the loop terminates, the specification is the disjunction over the conjunctions in $C_P$. Before returning it, the conjunctions are cleaned from redundant ones, which are the ones implying the disjunction of the other conjunctions (Lines 16–17).

### 5.2 The Modifications to SPEX

In this section, we describe the two modifications to SPEX and getMinLiteralNExamples.

#### Modifications to SPEX

As discussed at the beginning of this section, SPEX is modified to classify literals to $S_N$, only if all examples returned by getMinLiteralNExamples are positive. While literals may be classified to $S_P$ when a negative example is observed, this is an over-strict classification, since it excludes positive examples that do not satisfy this literal. Though some of these positive examples may be part of a different sub-concept (and thus will be covered later), others may be part of this concept, and excluding them will cause to splitting this sub-concept into two sub-concepts, which will introduce more questions. To avoid this, Gen-SPEX excludes only the negative examples by adding to $S_P$ a disjunction for every negative example, defined over the literals in $R_S$. The disjunctions exclude the examples since the examples satisfy the negations of literals in $R_S$. Additional literals from $S$ cannot be added to the disjunction because they may be part of the final conjunction. The code snippet below shows these modifications.

#### Modifications to getMinLiteralNExamples

As discussed at the beginning of this section, getMinLiteralNExamples cannot assume that literals in $S_N$ do not affect the examples’ classifications and thus they are not ignored and $\psi^S$ is used instead of $\psi^{R_S \setminus S_N}$.

### 5.3 The Overgen Operation

The overgen operation (Algorithm 5) takes a conjunction $con$, the example $e$ from which $con$ was generalized, $E_P$ and $E_N$, and a flag $isPos$ indicating whether $e$ is a positive or a negative example. It returns $true$ or $false$ to indicate whether $con$ is an over-generalization of $e$.

#### Main Idea

If $con$ over-generalizes $e$, there are (at least) two sub-concepts containing $e$ and captured by conjunctions that include $con$ and additional literals. These literals are: (i) satisfied by $e$, and (ii) not implied by $con$. Also, if $con$ is an over-generalization, there are negative examples satisfying it. Namely, there are examples satisfying $con$ but not any of the conjunctions of the sub-concepts. In particular, if there is an example: (i) satisfying $con$, and (ii) not satisfying any of the other literals satisfied by $e$ (which are not implied by $con$), then such example must be a negative example (or a positive example if $isPos$ is false) because it does not satisfy any of the other literals in the sub-concepts’ conjunctions. Thus, such example can be used to determine whether $con$ over-generalizes: if the user classifies it as positive (or negative if $isPos$ is false), $con$ does not over-generalize, and if it is negative, $con$ over-generalizes.

However, due to the dependency, there might not be such example, in which case overgen generates all satisfiable formulas that negate as many literals as possible (which must include a formula that does not satisfy any of the sub-concepts’ conjunctions, if $con$ over-generalizes). Then, it presents the user the corresponding examples, and if one of the examples is negative (or positive if $is-
$\text{Pos}$ is false), it determines that $\text{con}$ over-generalizes; otherwise, it determines that $\text{con}$ does not over-generalize.

**Implementation** The implementation of over$n$gen resembles the getMinLiteralNExamples operation only that it begins with negating all literals and diminishes this set if it is unsatisfiable (instead of beginning with one literal and extending its set). To this end, it maintains a set of tuples called $\text{sets}$ whose tuples consist of:

1. a set $N_{\text{g}}$ of literals that have to be negated, and
2. a subset of $N_{\text{g}}$, $R_{\text{s}}$, whose literals cannot be removed from $N_{\text{g}}$, as they are examining a certain possibility of relaxation (similarly to the $R_{\text{s}}$ sets in getMinLiteralNExamples). Initially, $\text{sets}$ contains a single tuple whose $N_{\text{g}}$ is the set of all tuples satisfied by $e$ and not implied by $\text{con}$ and $R_{\text{s}}$ is the empty set (there are no constraints yet on which literals cannot be removed). Then, a loop iterates the tuples in $\text{sets}$. For each tuple in $\text{sets}$, if $\text{con}$ and the literals in $N_{\text{g}}$ are satisfiable, i.e., $\psi_{\text{con}}$ (in its C-SPEX variation) is satisfiable, an example is presented to the user, and if it is a negative example (or a positive example if $\text{isPos}$ is false), true is returned to indicate that $\text{con}$ over-generalizes. If $\psi_{\text{con}}$ is unsatisfiable, then the tuple is replaced with a set of tuples, each considers a different possibility to relax $N_{\text{g}}$, that is removing a literal from the UNSAT core and obligating the other literals in the UNSAT core to remain in the relaxed $N_{\text{g}}$. Similarly to getMinLiteralNExamples, the UNSAT core is cleaned from $\text{con}$ and $R_{\text{s}}$, from the same reason that getMinLiteralNExamples removes $S_{\text{p}}$ and $R_{\text{s}}$ from the cores.

The loop terminates after all tuples correspond to satisfiable formulas, and their examples were classified as positive by the user (or negative if $\text{isPos}$ is false). In this case, it is guaranteed that $\text{con}$ does not over-generalize, and thus $\text{false}$ is returned.

### 5.4 Gen-SPEX Correctness and Guarantees

We next state that Gen-SPEX learns DNF formulas (more precisely, formulas that are close to DNF and can be easily transformed to DNFs), and that it learns with a minimal number of questions. Proofs are provided in Appendix A.

**Theorem 5.** Given $D$, $S$, $\varphi_{\text{C}}$ an unknown DNF specification over $S$, and initial positive and negative examples $E_{\text{P}}$ and $E_{\text{N}}$. Gen-SPEX is a DNF-ELE algorithm, i.e., it learns a specification $\varphi$ over $S$ such that: $\forall d, \varphi(d) \leftrightarrow \varphi(d)$.

**Theorem 6.** Given $D$, literals $S$ of size $n$, $\varphi_{\text{C}}$ an unknown DNF specification over $S$, and initial examples $E_{\text{P}}$ and $E_{\text{N}}$. If Gen-SPEX presents $\Omega(f(n))$ questions for some function $f$, any DNF-ELE algorithm presents $\Omega(f(n))$ questions.

### 6. Evaluation

In this section, we evaluate SPEX on an extension of the example from Section 2 and on two new applications.

### 6.1 SPEX Guided Traversal vs. Unguided Traversals

In this section, we experimentally demonstrate the importance of a guided traversal dependent on the vocabulary size (i.e., the predicates) instead of the domain size. To this end, we show that unguided traversals can present an exponential number of questions (in the number of predicates), even when a linear number suffices.

**Unguided Traversals** We consider CEGIS [48] and Oracle-Guided Synthesis [34] that perform unguided traversals:

- Oracle-Guided Synthesis [34] has the same interaction model as SPEX, where only membership queries are permitted. It begins by finding the set of formulas consistent with the initial examples, and iteratively prunes the space until only equivalent formulas remain (i.e., the concept formula). To prune the space, it arbitrarily selects two non-equivalent formulas consist-

| $p$ | #Q | Inc. | Avg. Inc. | Max | Min | CEGIS [48] | SPEX | Oracle-Guided [34] |
|-----|----|------|-----------|-----|-----|------------|------|---------------------|
| 18  | 6  | 11   | 12        | 6.9 | 6   | 6          |      |                     |
| 20  | 8  | 1    | 14.2      | 1.6 | 17  | 12         | 8.1  | 0.6                |
| 22  | 10 | 1    | 21.9      | 3.9 | 30  | 17         | 13.7 | 2.8                |
| 24  | 12 | 1    | 24.7      | 1.4 | 27  | 23         | 12.6 | -0.6               |
| 26  | 14 | 1    | 33.4      | 4.4 | 55  | 25         | 26   | 6.7                |
| 28  | 14 | 1    | 32.2      | 0.6  | 51  | 22         | 18.1 | -4.0               |
| 30  | 16 | 1    | 55.2      | 11.5 | 88  | 36         | 59.4 | 20.7               |

Table 2. SPEX VS. Oracle-guided Synthesis [34] and CEGIS [48].

tent with the current examples, presents an example that distinguishes them, and prunes the space based on the user’s output.

- CEGIS [48] has a different interaction model, where validation queries may also be presented, i.e., the user may be asked to confirm the specification. It begins by finding a formula consistent with the initial example, asks the user whether this is the correct concept, if so it terminates, otherwise it asks the user for an example eliminating this candidate, and repeats this process.

We compare to this algorithm even though it has a different interaction model to emphasize that even if the algorithm may present more powerful questions, an unguided traversal may still result in a exponential number of questions.

We consider the following setting:

- Domain: $D = \{ (x, y) \mid 0 \leq x \leq 2000, 0 \leq y \leq 2000 \}$.
- Set of predicates: $S_{\text{g}} = \{ P_{a}, x = y \}$, where $P_{a} = \{ 0 \leq a \leq b \leq 5 \}$ and $P_{a}$ is identical with respect to $y$.
- Concept: $C_{0} = \{ 800 \leq x \leq 1200 \} \land \{ 800 \leq y \leq 1200 \}$.
- Initial (positive) example: $\{ 840, 840 \}$. There are 18 predicates in $S_{\text{g}}$ satisfied by this example.

To demonstrate that unguided traversals dramatically increase the number of questions as the number of predicates increases, in contrast to SPEX, we consider six steps that modify this setting by extending $S_{\text{g}}$ and refining $C_{0}$. The added predicates are of the form $(z \leq a)$, which is satisfied if $a$ divides $z$. The steps are:

1. $S_{1} = S_{\text{g}} \cup \{ x \geq 2 \}$, $y \geq 2 \}$.
2. $S_{2} = S_{1} \cup \{ x \geq 3 \}$, $y \geq 3 \}$.
3. $S_{3} = S_{2} \cup \{ x \geq 4 \}$, $y \geq 4 \}$.
4. $S_{4} = S_{3} \cup \{ x \geq 5 \}$, $y \geq 5 \}.
5. $S_{5} = S_{4} \cup \{ x \geq 6 \}$, $y \geq 6 \}.
6. $S_{6} = S_{5} \cup \{ x \geq 7 \}$, $y \geq 7 \}$.

All new predicates are satisfied by the initial example, namely each step increases the number of satisfied predicates by two.

We ran SPEX and the algorithms of [34] and [48] on this benchmark and counted the number of questions. For the unguided traversals, which are non-deterministic, we ran 10 experiments and computed the average, maximum, and minimum number of questions. We also computed the increase factor of two consecutive steps, which is the ratio between the increase in the number of questions and the increase in the number of predicates, that is: $\text{Inc} = (q_{j} - q_{i-1})/(p_{j} - p_{i-1})$, where $q_{i}$ is the number of questions at step $i$, and $p_{i}$ is the number of predicates satisfied by the initial example at step $i$.

Table 2 shows the number of questions presented by the algorithms at each step and the increase factor (computed on the average number of questions). The table shows that as $p$ increases (i.e., there are more consistent concepts to consider), SPEX increases its number of questions linearly in $p$ (determined by the column $\text{Inc}$). Moreover, it never introduces more than $p/2$ questions. However, it is not the case for unguided traversals: the number of questions presented by the algorithm of [34] increases drastically and inconsistently. This is also true for CEGIS, though at the first steps it enjoys the advantage of being able to ask validation queries.
6.2 Technical Analysis Patterns

Technical analysis, used for trading assets such as stocks, futures, and commodities, tries to predict future price movement based on: (i) past price changes, often visualized in charts (functions mapping a finite set of consecutive dates to their corresponding prices), and (ii) special forms known as patterns. The occurrence of a pattern in a chart is used as a predictor of future price trends. For example, the head and shoulders pattern in Fig. 4 predicts price decline. Patterns are mainly characterized by the relation between the price and the extreme points of the charts. For example, patterns are captured via conjunctive formulas over the less-than predicate.

Technical analysis, used for trading assets such as stocks, futures, and commodities, tries to predict future price movement based on: (i) past price changes, often visualized in charts (functions mapping a finite set of consecutive dates to their corresponding prices), and (ii) special forms known as patterns. The occurrence of a pattern in a chart is used as a predictor of future price trends. For example, the head and shoulders pattern in Fig. 4 predicts price decline. Patterns are mainly characterized by the relation between the price and the extreme points of the charts. For example, patterns are captured via conjunctive formulas over the less-than predicate.

We employed C-SPEX to learn patterns from charts. The patterns are captured via conjunctive formulas over the less-than predicates, defined over the extreme points of the charts. For example, the head and shoulders pattern is defined over the extreme points of the charts. For example, patterns are captured via conjunctive formulas over the less-than predicates, defined over the extreme points of the charts. For example, patterns are captured via conjunctive formulas over the less-than predicates, defined over the extreme points of the charts. For example, patterns are captured via conjunctive formulas over the less-than predicates, defined over the extreme points of the charts. For example, patterns are captured via conjunctive formulas over the less-than predicates, defined over the extreme points of the charts. For example, patterns are captured via conjunctive formulas over the less-than predicates, defined over the extreme points of the charts.
is no guarantee that all non-commutative executions are sampled, and thus the formula may be unsound. This may occur since this approach has no control over the output part of the input-output examples. To avoid such cases, their evaluation reports that at least 5000 samples are used for every specification. As we next show, Gen-SPEX enables to carefully select the generated examples to both guarantee that no execution is missed with a minimal number of questions (on most tested scenarios this number is less than 10).

We next explain the task of learning commutative specifications with the example of sets.

**Sets** A set stores unique elements and supports the standard operations insert(e), remove(e), and contains(e). The operations return a flag ret indicating whether they were successful: insert(e) succeeds if e was not in the set and was thus inserted, remove(e) succeeds if e was in the set and was thus removed, and contains(e) succeeds if e is in the set.

**Commutative Specifications** A commutative specification of two operations is a formula satisfied when the two operations commute, i.e., when the resulted set and their return values are identical regardless of the order of the operations. For insert(e1) and insert(e2), if e1 ≠ e2, then the insertions do not affect each other and thus commute. If e1 = e2 then they commute only if their element was already in the set and none of them inserted. Namely, their commutative specification is:

$$\varphi_{\text{insert}(e_1), \text{insert}(e_2)} = (e_1 \neq e_2) \lor \neg \text{ret}_{e_1} \land \neg \text{ret}_{e_2}$$

**DNF-ELE of Commutative Specifications** ELE of commutative specifications gets as input two formulas capturing the operations’ behaviour and defined over (i) the data structure before the operation, (ii) the data structure after the operation, (iii) the operation’s parameters, and (iv) the return value. For example, insert(e) is captured by the following formula:

$$\varphi_{\text{insert}(ds, ds', e, ret)} = (e \in ds \Rightarrow ds' = ds \cup \{e\} \land \text{ret}) \land (e \notin ds \Rightarrow ds' = ds \land \neg \text{ret})$$

The data structures (i.e., ds, ds’ in the formula above) are captured using functions. For example, for ds which is a set, it is defined by the function Q : Elem → {0, 1}. Q(e) = 1 if e is in the set, which can be encoded as a formula. We next formalize the task of learning commutative specifications formally.

**Definition 6** (Commutative Specifications ELE). Given two operations’ formulas, $$\varphi_{\text{op_1}}(ds, ds', e_1^1, ..., e_m^1, \text{ret}_1)$$ and $$\varphi_{\text{op_2}}(ds, ds', e_1^2, ..., e_m^2, \text{ret}_2)$$, capturing the operation behaviours and defined over: (i) ds and ds’: the state of the data structure before and after the operation (resp.), (ii) e_1, ..., e_k: the parameters, and (iii) ret: the return value, commutative specifications ELE is defined as follows:

- The domain is the set of feasible executions consisting of a single invocation of each operation. Each execution is captured by a tuple consisting of: (i) ds, ds’, ds’’: the state of the data structure before the operations, after the first executed operation, and after the second executed operation (resp.), (ii) e_1, ..., e_k and e_1’, ..., e_k’: the parameters of the operations, and (iii) ret_1 and ret_2: the return values of the operations. Formally,

$$D = \{(ds, ds', ds'', e_1^1, ..., e_1^2, ..., e_m^2, \text{ret}_1, \text{ret}_2) \mid \begin{align*}
\varphi_{\text{op_1}}(ds, ds', e_1^1, ..., e_m^1, \text{ret}_1) \\
\varphi_{\text{op_2}}(ds', ds'', e_1^2, ..., e_m^2, \text{ret}_2) \\
\varphi_{\text{op_1}}(ds', ds'', e_1^1, ..., e_m^1, \text{ret}_2) \\
\varphi_{\text{op_2}}(ds, ds', e_1^2, ..., e_m^2, \text{ret}_1) \end{align*} \}$$

- The set of predicates S contains the pairwise relative comparisons over <, = of all numeric values (elements and return values) and the two states of boolean values (return values).

### Table 4. Gen-SPEX Evaluation Results.

| DS | Op1 | Op2 | pos | neg | Q | |S_0||S_F| Time |
|---|---|---|---|---|---|---|---|---|---|
| Set | Con | Con | 1 | 0 | 6 | 4 | 0 | 49 |
| Set | Con | Add | 2 | 1 | 10 | 12 | 2 | 236 |
| Set | Con | Rem | 2 | 1 | 11 | 12 | 2 | 224 |
| Set | Con | Size | 1 | 0 | 5 | 3 | 0 | 31 |
| Set | Add | Add | 2 | 2 | 9 | 16 | 3 | 403 |
| Set | Add | Rem | 1 | 1 | 9 | 8 | 1 | 243 |
| Set | Add | Size | 1 | 1 | 6 | 6 | 1 | 113 |
| Set | Rem | Rem | 2 | 2 | 9 | 16 | 3 | 376 |
| Set | Rem | Size | 1 | 1 | 6 | 6 | 1 | 109 |
| Set | Size | Size | 1 | 0 | 1 | 2 | 0 | 8 |
| Queue | Top | Top | 1 | 0 | 1 | 2 | 0 | 13 |
| Queue | Top | Push | 1 | 1 | 3 | 4 | 1 | 75 |
| Queue | Top | Pop | 1 | 1 | 3 | 4 | 1 | 77 |
| Queue | Top | Size | 1 | 0 | 3 | 2 | 0 | 20 |
| Queue | Push | Push | 1 | 1 | 3 | 4 | 1 | 153 |
| Queue | Push | Pop | 0 | 1 | 3 | 2 | 0 | 118 |
| Queue | Push | Size | 0 | 1 | 3 | 2 | 0 | 36 |
| Queue | Pop | Pop | 1 | 1 | 3 | 4 | 1 | 130 |
| Queue | Pop | Size | 1 | 1 | 3 | 4 | 1 | 68 |
| Queue | Size | Size | 1 | 0 | 1 | 2 | 0 | 9 |
| Reg | Get | Get | 1 | 0 | 1 | 2 | 0 | 8 |
| Reg | Get | Set | 1 | 2 | 4 | 18 | 2 | 405 |
| Reg | Set | Set | 1 | 2 | 20 | 48 | 3 | 6s |
| Map | Get | Get | 1 | 0 | 31 | 12 | 0 | 451 |
| Map | Put | Get | 2 | 1 | 154 | 60 | 2 | 17s |
| Map | Put | Put | 2 | 3 | 842 | 150 | 4 | 9m |

- A feasible execution is a positive example if it is commutative, captured by satisfying the formula $$\psi_{\text{comm}}$$:

$$\psi_{\text{comm}} = (\varphi_{\text{op_1}}(ds, ds', e_1^1, ..., e_m^1, \text{ret}_1) \land \varphi_{\text{op_2}}(ds', ds'', e_1^2, ..., e_m^2, \text{ret}_2)) \Rightarrow \begin{align*}
\varphi_{\text{op_1}}(ds, ds', e_1^1, ..., e_m^1, \text{ret}_1) \\
\varphi_{\text{op_2}}(ds', ds'', e_1^2, ..., e_m^2, \text{ret}_2) \\
\varphi_{\text{op_1}}(ds', ds'', e_1^1, ..., e_m^1, \text{ret}_2) \\
\varphi_{\text{op_2}}(ds, ds', e_1^2, ..., e_m^2, \text{ret}_1) \end{align*}$$

- $$E_p$$ and $$E_n$$ are sets of feasible executions. We set both to be empty, and let Gen-SPEX discover the required examples.

**Evaluation** We evaluated Gen-SPEX on commutative specifications of four common data structures: set, map, queue, and max register, with their standard operations:

- Set: contains(k), add(k), remove(k), and size().
- Map: get(k) and put(k, v), both return the value at position k (current or former), or 0 if k was not set before.
- Queue: top(), push(k), pop(), and size(), where top does not affect the queue and push does not return any value.
- Max register: get() and set(k) where set updates the register only if k is greater than its current value.

For each, we learned the specification of every pair of operations.

### Results

Table 4 shows the results. The columns are the data structure (DS), the operations used (Op1,Op2), the number of positive (pos) and negative examples (neg), SPEX used for generalizing to conjunctions (it does not include the examples it discovered while learning in C-SPEX, these examples are presented by the next column), the number of questions (Q), the total number of literals ($$|S_0|$$), the number of literals in the learned formula ($$|S_F|$$), and the time Gen-SPEX ran in milliseconds, unless followed by s or m to indicate that time is in seconds or minutes.

Table 4 shows that Gen-SPEX completes fast when the initial number of literals is small. Also, even when there are many dependencies, the number of questions is significantly lower compared to previous work [23] that required for the very least 5000 examples. Lastly, the specifications that have no negative behaviours
(i.e., the ones that do not modify the data structure) complete after Gen-SPEX observed a single (positive) example: for this example it learned the formula \texttt{true}, determined that \texttt{true} is not an over-generalization, and completed.

7. Related Work

In this section we discuss work that is most closely related to ours.

Learning Exact Specifications from Examples

Oracle-Guided Synthesis [34] is the closest setting to learning exact specifications. In this work, the space of programs (in our setting, specifications) is examined by iteratively searching for two programs with a distinguishing input, asking the user for its outputs, and pruning inconsistent programs, until converging to semantically-equivalent programs. Unfortunately, this may require an exponential number of questions. The reason is that while ideally every question prunes half of the space, this occurs only when the observed examples imply the classification of some predicate. To address this issue, the authors of [34] suggest users to begin with a small number of components (i.e., predicates) and gradually extend it until the resulting programs (specifications) capture their intent. Unfortunately, this requires users to validate the programs, which is undesirable and contradicts the premise of our work. Another work that learns exact specification is CEGIS [48], however it assumes that the user is expert and can read the synthesized solution, confirm if it is correct or provide an example to eliminate this solution. Even though it can enjoy more powerful questions, it may present an exponential number of questions, as the examples the user provide may lead to pruning only a few hypotheses from the hypothesis space.

Program Synthesis

The interest in program synthesis has grown dramatically over the years, and especially in the setting of synthesis from examples (e.g., [2, 8, 16, 20, 26-28, 30, 36-38, 41, 42, 45, 46, 53, 54]). However, these works focus on synthesizing programs consistent with the provided examples, and do not necessarily capture the user intent. Naturally, this implies that the complexity analysis of all these algorithms is dependent on the number of provided examples, which enables them to be polynomial or even linear. However, the task of guaranteeing exactness is more complex as we need to also reason about examples which were not given as input. Such examples may trigger questions by SPEX, and thus the asymptotic complexity worsens. A different line of synthesis work (i.e., constrained-based) guarantees exactness but requires the user to provide the specification (e.g., [3, 10, 47, 49]). Unfortunately, this is known to be complex and error-prone.

Relationship with Learning

Exact learning from examples (ELE) is closely related to query learning [6], which learns functions over input variables. Query learning is close but not identical, since ELE learns boolean functions over arbitrary predicates, which to the best of our knowledge is not the setting of query learning in any of its forms (e.g., DNF over inputs, automata, polynomials).

In the context of query learning, various results have been obtained for different interaction models. In particular, there has been a lot of work on query learning with equivalence queries (e.g., [1, 9]) that ask the user to validate the formula correctness (in addition to membership tests that ask users to classify selected inputs), and are not allowed in ELE. Works that do not use equivalence queries typically do not guarantee exact learning (e.g., [50]).

When the shape of the hypothesis space is restricted to combinations of independent monomials, classical results due to Goldman and Kearns [25] provide lower-bounds on the required number of questions, and define the notion of a teaching dimension. Intuitively, “the teaching dimension of a concept class is the minimum number of examples a teacher must reveal to uniquely identify any concept in the class” [25]. In this paper, we show how to obtain similar results for hypothesis spaces of formulas over first-order predicates. We generalize the results obtained for monomials [25] by providing algorithms that are guaranteed to ask the minimal number of questions, even when there are dependencies between predicates. The beauty of our algorithms is that they do not require the user to understand the dependencies between predicates, and instead rely on the computation of minimal unsat cores during the algorithm. This allows us to guarantee convergence with a minimal number of questions. We are not aware of prior work which can address this problem. Thus, we believe our work is a contribution to query learning as well.

Specifically, for the concept class of conjunctions over monomials, it is known ([25, Theorem 12]) that the teaching dimension is linear in the number of examples. This theorem, and the corresponding simple algorithm, do not work for formulas over first-order predicates due to potential correlations between predicates. The C-SPEX algorithm of Section 4 obtains similar results for the more general case of first-order predicates (see Theorem 4). As can be seen in Section 6.2, this has immediate practical implications, as the algorithm using monomials is not applicable, and the previously known oracle-guided algorithm asks a significantly larger number of questions. The results we obtain for DNF formulas over first-order predicates (Theorem 6) similarly generalize their results for DNF formulas over monomials. The results we obtain for DNF are of direct practical value as can be seen in Section 6.3 where our approach learns the exact specification with significantly fewer queries compared to the previous non-exact approach.

Learning Specifications

The task of learning specifications from a given program was studied using both static and dynamic techniques (e.g., [19, 22, 24, 29, 40, 43, 44]). The setting where a program is provided is inherently different from ours.

Concept Learning

SPEX is inspired by concept learning [39], which is the task of learning a concept from classified examples, where concepts are drawn from a hypothesis space (known as version space). SPEX is a novel algorithm for exact learning, generating examples such that convergence to a single hypothesis is guaranteed with a minimal number of examples.

Stream Pattern Detection

Many trading software platforms provide DSLs for traders (e.g., MetaTrader, MetaStock, NinjaTrader) and further DSLs exist, e.g., CPL [5], a Haskell-based high-level language designed for chart pattern queries and enabling fuzzy constraints and pattern composition. However, all require users to program, including programming (and thus understanding) the patterns’ mathematical specification. Other languages support queries for streams, e.g., SASE [52] for RFID streams, Cayuga [11] for detecting complex patterns, SPL [32], IBM’s stream processing language, StreamInsight [14], Microsoft’s stream processing language, and ActiveSheets [51] processing streams from within spreadsheets. However, all require users to mathematically express the detection condition and program the detector.

8. Conclusion

In this paper, we explored exact learning with a minimal number of examples for specifications over first-order predicates. Learning specifications over first-order predicates is practically important, especially for programming by examples. Learning with a minimal number of examples is important for reducing end user effort.

We show that in this setting, classical results on monomials cannot be used, due to the potential correlations between predicates. We therefore present an interactive learning algorithm SPEX that is guaranteed to ask the user a minimal number of questions, without making a priori assumptions on the relationships between predicates. We present several variations of SPEX that can be applied to conjunctive, disjunctive, and DNF specifications. We further show
that for certain predicate classes C-SPEX is guaranteed to ask a number of questions that is linear in the number of predicates.

We have implemented SPEX and applied it to two different application domains: pattern detection for technical analysts and data structure properties. Experimental results show that our synthesizer learns the exact hypothesis while presenting dramatically fewer questions than previous work.

References

[1] H. Abasi, N. H. Bshouty, and H. Mazzawi. On Exact Learning Monotone DNF from Membership Queries, pages 111–124. Springer International Publishing, Cham, 2014. URL http://dx.doi.org/10.1007/978-3-319-11662-4_9.

[2] A. Albarghouthi, S. Gulwani, and Z. Kincaid. Recursive program synthesis. In N. Sharygina and H. Veith, editors, Proceedings of the 25th International Conference on Computer-Aided Verification, CAV, 2013, pages 934–950, 2013. URL http://dx.doi.org/10.1007/978-3-642-39799-8_67.

[3] R. Alur, R. Bodik, G. Juniwal, M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In Proceedings of Formal Methods in Computer-Aided Design (FMCAD), pages 1–8. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679385.

[4] AmiBroker. https://www.amibroker.com/.

[5] S. Anand, W.-N. Chin, and S.-C. Khoo. Charting patterns on price history. In Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming, ICFP ’01, pages 134–145, 2001. URL http://doi.acm.org/10.1145/507635.507653.

[6] D. Angluin. Queries and concept learning. Machine Learning, 2(4):319–342, 1988. URL http://dx.doi.org/10.1007/BF00116828.

[7] D. Angluin, L. Hellerstein, and M. Karpinski. Learning read-once formulas with queries. J. ACM, 40(1):185–210, Jan. 1993. URL http://doi.acm.org/10.1145/138027.138061.

[8] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’15, pages 218–228, 2015. URL http://doi.acm.org/10.1145/2737924.2737952.

[9] A. Beinæ, F. Bergadano, N. H. Bshouty, E. Kushilevitz, and S. Varricchio. Learning functions represented as multiplicity automata. J. ACM, 47(3):506–530, May 2000. URL http://doi.acm.org/10.1145/337244.337257.

[10] J. Bornholt, E. Torlak, D. Grossman, and L. Ceze. Optimizing synthesis with metaskehtes. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’16, pages 775–788, 2016. URL http://doi.acm.org/10.1145/2837614.2837666.

[11] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: A high-performance event processing engine. In Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD ’07, pages 1100–1102, 2007. URL http://doi.acm.org/10.1145/1247480.1247620.

[12] T. Bulkowski. Visual Guide to Chart Patterns. Bloomberg Financial. 2012.

[13] T. N. Bulkowski. Encyclopedia of Chart Patterns. Wiley, 2nd edition, 2005.

[14] B. Chandramouli, J. Goldstein, and D. Maier. High-performance dynamic pattern matching over disordered streams. In VLDB ’10.

[15] A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. J. Artif. Intell. Res. (JAIR), 40, 2011.

[16] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In Proceedings of the 15th International Conference on Database Theory, ICDT ’10, pages 89–103, 2010. URL http://doi.acm.org/10.1145/1804669.1804683.

[17] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS’08/ETAPS’08, pages 337–340. 2008. URL http://dl.acm.org/citation.cfm?id=1792734.1792766.

[18] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen. Commutativity race detection. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, pages 305–315, 2014. URL http://doi.acm.org/10.1145/2594291.2594322.

[19] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. Sci. Comput. Program., December, 2007.

[20] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’15, pages 229–239, 2015. URL http://doi.acm.org/10.1145/2737924.2737977.

[21] J. Franklin, P.-M. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: A type-theoretic interpretation. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’16, pages 802–815, 2016. URL http://doi.acm.org/10.1145/2837614.2837629.

[22] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A Robust Framework for Learning Invariants, pages 69–87. Springer, 2014. URL http://dx.doi.org/10.1007/978-3-319-08867-9_5.

[23] T. Gehr, D. Dimitrov, and M. Vechev. Learning commutativity specifications. In D. Kroening and S. C. Păsăreanu, editors, Proceedings of the 27th International Conference on Computer Aided Verification, CAV 2015, pages 307–323, 2015. URL http://dx.doi.org/10.1007/978-3-319-21690-4_18.

[24] P. Godefroid and A. Taly. Automated synthesis of symbolic instruction encodings from i/o samples. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’12, pages 441–452, 2012. URL http://doi.acm.org/10.1145/2254064.2254116.

[25] S. Goldman and M. Kearns. On the complexity of teaching. J. Comput. Syst. Sci., 50(1):20–31, Feb. 1995. URL http://dx.doi.org/10.1006/jcss.1995.1003.

[26] S. Gulwani. Dimensions in program synthesis. In Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP ’10, pages 13–24, 2010. URL http://doi.acm.org/10.1145/1836089.1836093.

[27] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’11, pages 317–330, 2011. URL http://doi.acm.org/10.1145/1926385.1926423.

[28] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. Commun. ACM, 55(8):97–105, Aug. 2012. URL http://doi.acm.org/10.1145/2240236.2240260.

[29] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS ’09, pages 262–276, 2009. URL http://dx.doi.org/10.1007/978-3-642-00768-2_24.

[30] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’11, pages 317–328, 2011. URL http://doi.acm.org/10.1145/1993498.1993536.
Claim 1  Proof Let $A$ be an $C$  



\begin{proof}
Base: let $A$ be an example (in which $x_0 = 0$). We set $l_i = x_0$. We set $l_i = x_0$ for some $i$. Let $i$ be an example (in which $x_0 = 0$). We set $l_i = x_0$. We set $l_i = x_0$ for some $i$. Let $i$ be an example (in which $x_0 = 0$).
\end{proof}

A. Proofs

A.1 Section 3

Claim 1 Proof Let $A$ be an CELE algorithm, $D = \{ (x_0, ..., x_k) \mid \forall i, x_i \in \{0, 1\} \}, S = \{ (x_0 \lor x_i), (x_0 \land \neg x_i) \mid 1 \leq j \leq k \},$ and the target formula $\varphi_C = \bigwedge_{0 \leq k \leq k} (x_0 \lor l_i)$ where $l_i \in \{x_0, \neg x_i\}$ are defined later. Assume $E_P \subseteq E_N$, the examples in which $x_0 = 1$ are positive and do not eliminate any predicate from $S$, thus assume $A$ does not present such examples (this only helps $A$ to avoid uninformative questions). We prove that there is a selection of $l_i$ for which the first $2^{k-1} - 1$ examples, $d_1, ..., d_{2^{k-1} - 1}$, are all negative and neither enables to infer which predicates are in $\varphi_C$.

Base: let $d_0$ be an example (in which $x_0 = 0$). Assume $A$ infers some classification:

- If $A$ infers $x_0 \land x_i$, is in $\varphi_C$ (for some $i$), then we set $l_i = x_0$, and for some $i$ we set $l_i = x_0$, and $D_0 = \neg \varphi_C$, or $l_i = \neg x_j$, otherwise. Namely, $A$ inferred incorrectly.
- If $A$ infers $x_0 \lor \neg x_i$, is in $\varphi_C$, we set $l_i = x_0$, and get contradiction similarly.
- If $A$ infers $x_0 \lor x_i$, is not in $\varphi_C$, we set $l_i = x_0$, and get contradiction similarly.
- If $A$ infers $x_0 \lor \neg x_i$, is not in $\varphi_C$, we set $l_i = x_0$, and get contradiction similarly.

Step: Assume that the $d_0, ..., d_{2^{k-1} - 1}$ are classified as negative and no predicate was classified as belong or not belong to $\varphi_C$. Assume $A$ infers some classification using the example $d_0$ (in which $x_0 = 0$):

- If $A$ infers $x_0 \lor x_i$, is in $\varphi_C$ (for some $i$): There are $2^{k-1}$ examples in which $x_i = 1$. However, there are $2^{k-1} + 1$ unclassified examples (in which $x_0 = 0$) since $m \leq 2^{k-1} - 1$. Thus, there exists an example $d_i$ in which $x_0 = x_i = 0$, such that $d_i \neq d_j$ for all $0 \leq j \leq m$. We set $l_i = x_0$ if $d_i \equiv x_0$ and if $l_i = x_0$, otherwise.
- For every $0 \leq j \leq m$, $d_j$ is indeed negative, i.e., $d_j \not\in \varphi_C$, since $d_j \neq d_i$ and thus for some $i = x^j \not\in x^j$. However, $A$ inferred incorrectly because $l_i = x_0$. 


• If A infers $x_0 \lor \neg x_i \in \varphi_C$, there is $d$ in which $x_i = 1$ and we contradict A similarly.
• If A infers $x_0 \land x_i \notin \varphi_C$, there is $d$ in which $x_i = 1$ and we contradict A similarly.
• If A infers $x_0 \lor \neg x_i \notin \varphi_C$, there is $d$ in which $x_i = 0$ and we contradict A similarly.

A.2 Section 4

Theorem 1 Proof We prove in induction that during the execution of C-SPEX, $\bigwedge_{l \in S_N} l \models \varphi_C \models \bigwedge_{l \in S_P} l$ and since at the end of the execution $S \setminus S_N = S_P$ and $\varphi = \bigwedge_{l \in S_N} l$, it follows that $\varphi \models \varphi_C$.

Base: initially, $S_P = S_N = \emptyset$, and thus we show $S = = \varphi_C = \emptyset$. $S$ contains all predicates satisfied by all examples in $E_P$ and since $\varphi_C$ contains literals from the initial literals, if we assume that $S \models \varphi_C$, then there is a literal $l$ in $\varphi_C$ not in $S$. This means that there is $e \in E_P$ not satisfying $l$, and thus not satisfying $\varphi_C$, in contradiction to the fact that $e \in E_P$.

Step: We assume $S \setminus S_N = \varphi_C = S_P$ and show that updates to $S_P$ or $S_N$ preserve these implications. Let $l$ be a literal classified.

We split to cases:

• If $l$ is classified to $S_N$ then it must because an example $e$ was generated in which $l$ is negated and was classified positive. In this case, we show $S \setminus (S_N \cup \{l\}) = \varphi_C$ (as $S_P$ does not change, and from the induction hypothesis it continues to hold $\varphi_C \models S_P$). To show $S \setminus (S_N \cup \{l\}) = \varphi_C$, it suffices to show that $l \notin \varphi_C$. Assume in contradiction $l \in \varphi_C$, then since $e$ is positive, $e \models \varphi_C$, however $e \models \neg l$ - contradiction.

• If $l$ is classified to $S_P$ and it was classified following another literal $l'$ was added to $S_P$ and $S_P \cup \{l\}$, then $S_P \models S_P \cup \{l\}$ and thus $\varphi_C \models S_P \cup \{l\}$.

• If $l$ is classified to $S_P$, Then we define the following example each classified negative. We show $l \epsilon \varphi_C$ and thus conclude $\varphi_C \models S_P \cup \{l\}$.

Assume in contradiction $l \notin \varphi_C$. Every example was classified negative, and thus for each example $e$ either $l$ is in $\varphi_C$ or one of the other literals of $R_S$ is in $\varphi_C$ (otherwise the example should have been classified as positive, since the rest of the predicates of $\varphi_C$ are in $S \setminus S_N$, since $S \setminus S_N = \varphi_C$). Assume that when examining $l$ and extracting the $l^{th}$ UNSAT core, the result was $\neg l, l_1, \ldots, l_k$ (note that $\neg l$ must be in the UNSAT core, since $S$ is satisfiable and $l_1, \ldots, l_k \in S$), namely $l_1 \land \cdots \land l_k \models l$. If $l_1, \ldots, l_k \in \varphi_C$, then $l \models \varphi_C$ which is equivalent to saying that $l$ is in $\varphi_C$, in contradiction to our assumption. Thus, in each iteration of extracting the UNSAT core, one of the literals is not in $\varphi_C$, denote it by $Q'$. However, in this case the example satisfying $\bigwedge_{l \in S_N \cup \cup \{l_1, \ldots, l_k\}} l' \models \varphi_C$ may be a positive example since it satisfies all the literals in $\varphi_C$. However, this example was classified as negative, and thus our assumption is contradicted again and $l$ must be in $\varphi_C$.

Theorem 2 Proof Dual to the previous proof.

Theorem 3 Proof We prove the theorem for the C-SPEX, the proof for D-SPEX is similar. Let A be an CELE algorithm, $D$ a domain, $S$ predicates over $D$ of size $n$, an unknown target formula $\varphi_C$, $E_P$ and $E_N$ sets of positive and negative examples, and a literal $l \in S$. We prove the following claim:

CLAIM 3. To determine $l$’s classification (i.e., in or not in $\varphi_C$) at some moment during the execution, where:

• $S_P$ contains all literals already known to be part of $\varphi_C$, i.e., for all $Q \in S_P$, all positive examples satisfy $Q$.
• $S_N$ contains all literals already known to be in $\varphi_C$, i.e., for all $Q \in S_N$, there exist a positive example $e'$ not satisfying (equivalently, all positive examples satisfy every predicate in $S \setminus S_N$).

A must either have:

• (A new) positive example $e_l$ not satisfying $l$, in which case $l$ is classified to $S_N$ (as $l$ is not in $\varphi_C$ if $e_l$ is positive), or
• All (new) negative examples, $e_1, \ldots, e_k$, satisfying:
  • $e_1, \ldots, e_k \models S_P$, for all $1 \leq i \leq k$, $e_i \notin l$,
  • for all $1 \leq i \leq k$, $S_{l_i}$ is minimal, where $S_{l_i} = \{Q \in S \setminus S_N | e_i \notin Q\}$ (i.e., every subset $S'$ of $S_{l_i}$ does not have an example satisfying $l$ and the predicates in $S \setminus S'$).
  • for all $1 \leq i < j \leq k$, $S_{l_i} \neq S_{l_j}$, in which case $l$ is classified to $S_P$.

Using this claim, the theorem is proven as follows: SPEX presents for every $l$ the examples $e_1, \ldots, e_k$ as described in the claim (up to the property that they are negative, which is known only after the user classifies them), and may only stop if one of them is classified as positive. Namely, these examples consist the superset of the examples SPEX presents to classify $l$. Thus, and since every literal in $S$ is potentially in $\varphi_C$ (more precisely, $S_0$ of Line 1 in Algorithm 2), this claim implies that A requires at least as many examples as SPEX, namely SPEX presents a minimal number of questions. Note that if an example $e$ is required for the classification of two literals $l_1$ and $l_2$, then it is counted only once since if $e$ was classified as positive, both $l_1$ and $l_2$ are classified as not belonging to $\varphi_C$ (in which case $e$ is not presented again), and otherwise $e$ is stored in $E_N$ and will not be presented to the user again. We note that if $l$ is not in $\varphi_C$, SPEX presents in the worst case all examples (if all but $l_i$ are negative examples), and thus it may be that occasionally A “gets lucky” and asks fewer questions (e.g., if its first question is $e_1$. A need not present $e'_1, \ldots, e'_{k-1}$), however, up to the order of $e'_1, \ldots, e'_k$. A asks as many questions (examples) as SPEX.

Proof of the Claim. Let $S_P$ and $S_N$ be the sets as described in the claim during some point of the execution, and let $e_1, \ldots, e_k$ be the positive examples observed and $e'_1, \ldots, e'_k$ be the negative example observed. Assume that A classifies $l$. We split to cases:

• if $l$ is classified to $S_N$: we show that A must have at least one positive example not satisfying $l$. Suppose otherwise, then we show that there is a specification $\varphi_C$ containing $l$ which is consistent with the previous examples’ classifications and the previous predicates’ classifications, and this contradicts A’s classification of $l$. Assume that A does not have a positive example not satisfying $l$, namely all other positive examples $e_1, \ldots, e_t$ satisfy $l$. We set $\varphi_C = \bigwedge_{Q \in S \setminus \{e_1 \ldots e_k\}} Q$. $\varphi_C$ is consistent with the examples:
  • By construction all positive examples satisfy $\varphi_C$.
  • Every negative example $e'$ does not satisfy $\varphi_C$: suppose otherwise and suppose that there is another specification $\varphi_C$ consistent with the classification of all the observed examples. For every two positive examples, literals that are satisfied by one but not by the other cannot be in $\varphi_C$ (because otherwise one of them will not be a positive example). Thus, $e'$ cannot be a negative example because one of the literals satisfied by one of the positive examples but not by another positive example, and there must be a predicate in $\varphi_C$ satisfied by all positive examples but not by $e'$, however in this case this predicate is also in $\varphi_C$, i.e., $e' \notin \varphi_C$.

$\varphi_C$ is consistent with the literals:

• From $S_P$ definition, every positive example satisfies all literals in $S_P$, and thus every such literal in $S_P$ is also in $\varphi_C$, as required.
• Every literal in $S_N$ has a positive example not satisfying it, and thus by construction it is not in $\varphi_C$, as required.
Theorem 4 Proof. \(l\) belongs to exactly one sub-concept, the modified C-SPEX learns a conjunction which is not an over-generalization.

Proof. To prove that its learned conjunction is not an over-generalization, we prove that it is not satisfied by any negative example (because a conjunction is an over-generalization only if it is satisfied by a negative example). If \(e\) belongs to exactly a single sub-concept, captured by a conjunct \(c\), then it satisfies all the conjunction's constraints, and for each other sub-concept it has at least one constraint which \(e\) does not satisfy. Let \(e'\) be the conjunction learned by C-SPEX. We prove that every example that satisfies \(e'\) either satisfies \(e\) or is a positive example satisfying a different sub-concept. Let \(l\) be a literal in \(e\). Modified C-SPEX classifies \(l\) either by generating examples for \(l\) or if detecting that \(l\) is implied by the learned conjunction. In the latter case, \(e\) satisfies \(l\), and thus \(l\) is added to \(S_P\) and the claim holds. We focus on the case where C-SPEX generates examples to classify \(l\). If \(l\) can be classified using a single example \(e_l\), then \(e_l\) satisfies all constraints \(e\) satisfies, except for \(l\) whose negation is satisfied. Since \(l\) is part of \(e\), \(e_l\) does not satisfy this sub-concept. Also, since for any other sub-concept \(e\) does not satisfy at least one constraint, \(e_l\) also does not satisfy at least one constraint, too, and thus \(e_l\) does not belong to any of the other sub-concepts. Thus, \(e_l\) is classified as a negative example (positive, in case C-SPEX is given a negative example to generalize), and \(l\) is classified to \(S_P\). Otherwise, if \(l\) cannot be classified with a single example, namely it has multiple examples \(e_l, ..., e_l\), each satisfies the negation of \(l\). For every negative example (or positive, in case C-SPEX is given a negative example to generalize), a disjunction eliminating it is added to \(S_P\). Note that the all these disjunctions contain \(l\) and thus all positive examples of this sub-concept satisfy these disjunctions. Further note that adding the disjunctions enables at most the positive examples among \(e_l, ..., e_l\) to satisfy it. This follows since any other example negates an additional literal \(l'\), which either cannot be satisfied with \(c\) or that it is yet to be classified, and when \(l\) will be classified its negation will be tested and SPEX will discover if negating it results in negative examples.
CLAIM 5. If an example \( e \) belongs to more than one sub-concept then: 1. the modified C-SPEX learns a conjunction which is an over-generalization, and 2. overgen detects this.

Proof of 1: If \( e \) satisfies the constraints of (at least) two concepts, then when C-SPEX negates the constraints of the first concept, the other concept’s constraints are satisfied, resulting in observing only positive examples and thus adding these constraints to \( S_N \). When C-SPEX negates the constraints of the second concept, the constraints of the first concept are satisfied (because the second modification ensures that the literals in \( S_N \) are satisfied), and thus the second concept’s constraints also added to \( S_N \), and thus \( S_P \) is missing constraints of every sub-concept \( e \) belongs to. This implies that the conjunction learned must be an over-generalization. This is because if there are no negative examples satisfied by it, then there is no need in the sub-concept learned by C-SPEX contains all the sub-concepts, in which case these sub-concepts are meaningless and thus can be ignored, and then \( e \) would not have been considered as part of two sub-concepts.

Proof of 2: Let \( \text{con} \) be the conjunction learned by the modified C-SPEX, namely \( \text{con} \) over-generalizes \( e \). Since there are (at least) two sub-concepts containing \( e \) and captured by conjunctions that include \( \text{con} \) and additional literals. These literals are: (i) satisfied by \( e \), and (ii) not implied by \( \text{con} \). Also, since \( \text{con} \) is an over-generalization, there are negative examples satisfying it. Namely, there are examples satisfying \( \text{con} \) but not any of the conjunctions of the sub-concepts. In particular, there are negative examples: (i) satisfying \( \text{con} \), and (ii) not satisfying some of the other literals satisfied by \( e \) (which are not implied by \( \text{con} \)). In addition, any example satisfying even fewer constraints satisfied by \( e \), is also a negative example. Thus, since overgen constructs all the examples not satisfying a maximal number of constraints, it must encounter a negative example and determine that \( \text{con} \) is an over-generalization.

Theorem 6 Proof Let A be a DNF-ELE algorithm (i.e., an algorithm that can learn arbitrary DNF formulas), \( D \) a domain, \( S \) predicates over \( D \) of size \( n \), an unknown target formula \( \varphi_C \), \( E_P \) and \( E_N \) sets of positive and negative examples. We first prove that the modified C-SPEX generalizes examples as much as possible:

CLAIM 6. Let \( c \) be a conjunction in \( \varphi_C \), C-SPEX captured \( e \) via \( c_{\text{SPEX}} \) which is satisfied by at least all the examples satisfied by \( e \).

Proof of claim. Assume that C-SPEX generalizes from an example \( e \) and assume in contradiction that there is an example \( e' \) satisfying \( c \) but not \( c_{\text{SPEX}} \). This means that there is a literal \( l \) such that \( e' \not\models l \) for which either \( l' \) is in \( c_{\text{SPEX}} \) or in a disjunction in \( c_{\text{SPEX}} \). In the first case, it means that C-SPEX has added \( l \) after generating an example \( e_l \) satisfying all literals from \( S \) that \( e \) satisfies (except for \( l' \)) and \( \neg l \), and this example was classified as negative by the user (or positive if we generalize from a negative example). Since \( c' \)'s literals must be a subset of the literals from \( S \) satisfied by \( e \), it follows that \( c_l \), which is a negative example, satisfies \( e \), but this cannot happen since \( c \) is a correct sub-concept, i.e., includes only positive examples. In the latter case, \( c_{\text{SPEX}} \) excluded exactly the negative behaviours, and thus if \( e' \) is excluded, then the negatable example \( e_N \) that led to adding the disjunction that excludes \( e' \), implies that \( e' \) cannot be in \( c \); if it were, then since it satisfies fewer literals than \( c_N \) (compared to the original example \( e \)), \( e_N \) also must be in \( c \), but it was classified as negative, in contradiction.

The next claim states that modified C-SPEX presents a minimal number of questions, given the assumption that positive examples may be examples which are not part of the hidden concept.

CLAIM 7. If positive classifications of examples do not imply that the examples are part of the learned sub-concept, then the modified C-SPEX generates a minimal number of questions to learn the sub-concept.

Proof of claim. C-SPEX was shown to present a minimal number of questions. Compared to it, the modified C-SPEX introduces more questions only when there is dependency because it does not ignore literals in \( S_N \) when generating the formulas \( \psi \) (in getMinLiteralNExamples). To prove the claim, we demonstrate that ignoring these literals may result in incorrect specifications, and thus the additional questions cannot be avoided. Consider the domain of boolean vectors of size 3, the predicates are monomials, i.e., \( x_0, x_1, x_2 \), and the specification is: \( (x_0 \land x_1) \lor (x_0 \land x_2) \). Suppose that C-SPEX is given the example \((1,1,0)\) which satisfies the first conjunction but not the second, and thus C-SPEX should be able to generalize it correctly (overgen does not detect such overgeneralizations). Initially, C-SPEX tests whether \( \neg x_2 \) is part of the conjunction, and presents the example \((1,1,1)\), which is positive, and thus it infers that \( \neg x_2 \) is not in the conjunction. Next, it tests whether \( x_1 \) is in the specification. Since \( \neg x_2 \) is ignored, an example satisfying \( x_0 \) and \( \neg x_1 \) is \((1,0,1)\) which is positive (because it satisfies the other conjunction), and thus \( x_1 \) erroneously is removed from the conjunction.

The next claim states that any algorithm that does not generalize both positive and negative examples may present is outperformed by Gen-SPEX.

CLAIM 8. Let A be a DNF-ELE. If A only learns a single DNF formula (satisfied by the positive examples only, or by the negative examples only), then A may present \( \Omega(2^n) \) questions, where \( n \) is the size of \( S \) (the set of literals).

Proof. Suppose A only learns a DNF formula satisfied by the positive examples and consider a concept containing exactly a single example from the domain. Then, A has to examine all examples in the domain, or more precisely all non-equivalent examples with respect to the predicates in \( S \). In general, there are \( \Omega(2^n) \) such examples. Even if there are some dependencies and not all combinations of literals from \( S \) are satisfiable, in general this number is still exponential.

In contrast, Gen-SPEX will generalize the negative examples with a minimal number of questions, and in particular will not present more questions than A.

The next claim states that if A generalizes a conjunction, that is by dropping its constraints, then it risks in over-generalization and thus has to trigger at least the questions that overgen introduces to discover this.

CLAIM 9. If at some point of the algorithm A adds to the learned DNF the conjunction \( \text{con} = \bigwedge_{S' \subseteq S} S' \) where \( S' \subseteq S \), it must observed all examples overgen would generate for \( \text{con} \) with the positive example \( e \) used by C-SPEX to compute \( \text{con} \).

Proof. If A generated this conjunction and added it to a DNF, it must have seen a positive example \( e \) satisfying all \( \text{con}' \)s literals (otherwise the DNF specification is incorrect). If A does not examine one of the examples overgen generates for \( e \) and \( \text{con} \), then this example may be negative. This follows because the examples overgen generates do not imply one another nor are implied by other examples satisfying \( \text{con} \), and thus A could not avoid this example through a different example. Namely, A added to the DNF a cube that is satisfied by a negative examples, and thus learned an incorrect DNF.

Theorem Proof. The three first claims imply that the questions submitted to C-SPEX are of a minimal number, the examples C-SPEX presents are of a minimal number, and the learned conjunctions are guaranteed to cover as many examples from the domain as possible. The last claim implies that A cannot reduce the number of questions overgen presents. Thus, A may only have an advantage over Gen-SPEX if it happened to pick better examples to generalize. However, A (as Gen-SPEX) has no knowledge on the
unlearned sub-concepts, and if for some concept it “was lucky” to
draw an e that is not part of two sub-concepts, and Gen-
SPEX picked an example e′ that belongs to more than one sub-concept,
then there are concepts which have the same sub-concepts as A and
Gen-SPEX learned so far, in which the example e is part of two
sub-concept and e′ is not. Thus, overall Gen-SPEX learns concepts
with a minimal number of questions.

B. Technical Analysis Common Patterns

In this section we describe the patterns used to evaluate C-SPEX
in Section 6.2. We used the following patterns: (i) head and
shoulders, (ii) cup with handle, (iii) double tops, (iv) symmetrical
triangle, (v) rectangle, and (vi) flag. The last five patterns are illustrated
in Fig. 5; for further reading see [12, 13, 33]..

The challenge in evaluating C-SPEX is to decide on the pattern
definition to use as pattern definitions are subjective. To overcome
this challenge, we ran several experiments for each pattern, each
with a different formula (but with the same example). The different
definitions, taken from textbooks and online forums, span a range
of possible definitions, from the most permissive to the most re-
strictive. We next provide a general description of the patterns and
the definitions used.

Head and Shoulders Three peaks, the middle is the highest.
(1) Most permissive – three peaks, middle one is the highest.
(2) (1) with shoulders higher than all lows.
(3) (2) where p3, p6 are lower than the other points.
(4) (3) with ascending “neckline” (p2 < p4 < p3) and p0 < p6.
(5) Most restrictive – the given chart is the only valid chart.

Cup with Handle A rise, followed by a cup-shape, then a decline
(“the handle”), and finally another rise.
(1) Most permissive – all four parts exist.
(2) (1) with significant rise; p5 is higher than the other points.
(3) (2) with p0, p5 lower than the other points.
(4) (3) with handle not lower than the cup (¬((p4 < p2)).
(5) Most restrictive – the given chart is the only valid chart.

Two Tops Two peaks of equal height.
(1) Most permissive – there are two equal height tops.
(2) (1) with middle low (p2) not lower than the other lows.
(3) (2) with last point (p4) lower than the other points.
(4) Most restrictive – the given chart is the only valid chart.

The next patterns are captured by constraints that leave little room
for different definitions and thus only two are listed.

Symmetrical Triangle Descending peaks (p1 > p3 > p5), ascending
lows (p2 < p4 < p6), and p2 < p0, p0 < p4.
(1) Most permissive – p0 appears between p1 and p2.
(2) Most restrictive – the given chart is the only valid chart.

Flag A pole followed by descending peaks (p1 > p3 > p5), descend-
ing lows (p2 > p4 > p6), and p0 lower than all points.
(1) Most permissive – p2 and p5 may be equal.
(2) Most restrictive – the given chart is the only valid chart.

Rectangle Peaks (p1, p3) are equal, lows (p2, p4) are equal, and
p0 not higher than p1.
(1) Most permissive – p0 is not higher than p1.
(2) Most restrictive – the given chart is the only valid chart.

C. Full results for C-SPEX

In this section, we provide the full results of Section 6.2 that include
the overall time (in milliseconds).
| Pattern                  | |S₀| | |S₁| |CSPEX| |Oracle Based| |CSPEX| |OB|
|-------------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
|                         | #Questions | Time (millisecc) | CSPEX Num | Avg. | Max | Min | CSPEX Num | Avg. | Max | Min | CSPEX Num | Avg. | Max | Min |
| Head and Shoulders 42   | (1) 6      | 18 | 44.4 | 57 | 38 | 360 | 139 | (1) 5 | 12 | 38.3 | 48 | 27 | 107 | 82|
| (2) 10                  | 18 | 54.8 | 61 | 43 | 223 | 142 | (2) 6 | 12 | 43.6 | 57 | 33 | 93 | 82|
| (3) 10                  | 17 | 61.1 | 88 | 43 | 344 | 155 | (3) 7 | 13 | 35.1 | 41 | 28 | 232 | 70|
| (4) 7                   | 14 | 50.6 | 71 | 40 | 285 | 132 | (4) 7 | 13 | 36.4 | 41 | 32 | 184 | 66|
| (5) 6                   | 12 | 58.4 | 77 | 37 | 267 | 147 | (5) 5 | 10 | 38.6 | 45 | 22 | 111 | 69|
| Cup with Handle 30      | (1) 5      | 12 | 38.3 | 48 | 27 | 107 | 82 | (1) 6 | 12 | 43.6 | 57 | 33 | 93 | 82|
| (2) 6                   | 12 | 43.6 | 57 | 33 | 93 | 82 | (2) 6 | 12 | 43.6 | 57 | 33 | 93 | 82|
| (3) 7                   | 13 | 35.1 | 41 | 28 | 232 | 70 | (3) 7 | 13 | 35.1 | 41 | 28 | 232 | 70|
| (4) 7                   | 13 | 36.4 | 41 | 32 | 184 | 66 | (4) 7 | 13 | 36.4 | 41 | 32 | 184 | 66|
| Two Tops 20             | (1) 6      | 9  | 19.4 | 20 | 18 | 56 | 29 | (1) 6 | 9  | 19.4 | 20 | 18 | 56 | 29|
| (2) 6                   | 9  | 18.3 | 20 | 16 | 49 | 29 | (2) 6 | 9  | 18.3 | 20 | 16 | 49 | 29|
| (3) 6                   | 7  | 18.7 | 19 | 17 | 56 | 28 | (3) 6 | 7  | 18.7 | 19 | 17 | 56 | 28|
| (4) 4                   | 6  | 17   | 17 | 17 | 56 | 25 | (4) 4 | 6  | 17   | 17 | 17 | 56 | 25|
| Symmetrical Triangle 42 | (1) 7      | 16 | 71.4 | 76 | 62 | 23 | 186 | (1) 7 | 17 | 64     | 89 | 52 | 706 | 175|
| (2) 7                   | 14 | 66.5 | 78 | 37 | 239 | 165 | (2) 6 | 16 | 54.3 | 90 | 43 | 67 | 136|
| (3) 7                   | 17 | 64   | 89 | 52 | 706 | 175 | (3) 6 | 16 | 54.3 | 90 | 43 | 67 | 136|
| (4) 6                   | 8  | 27.4 | 28 | 25 | 78 | 36 | (4) 6 | 8  | 27.4 | 28 | 25 | 78 | 36|

Table 5. Results for CSPEX. Number of questions presented by C-SPEX vs. average, maximal, and minimal number of questions presented by the Oracle-Guided approach. Time for generating the next question for both algorithms is shown in milliseconds.