ICE: A Robust Framework for Learning Invariants

Pranav Garg[1], Christof Löding[2], P. Madhusudan[1], and Daniel Neider[2]

[1] University of Illinois at Urbana-Champaign [2] RWTH Aachen University

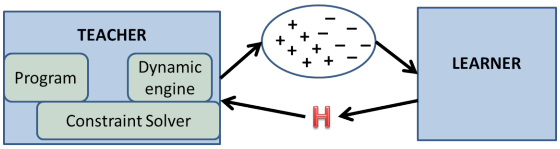
Abstract. We introduce a robust learning paradigm for synthesizing invariants, called ICE-learning, that learns using examples, counter-examples, and *implications*, and show that it admits honest teachers and convergent mechanisms for invariant synthesis. We observe that existing algorithms for black-box abstract interpretation can be interpreted as ICE learning algorithms. We develop new convergent ICE-learning algorithms for two domains, one for learning Boolean combinations of numerical invariants for scalar variables and one for *quantified* invariants for arrays and dynamic lists. We implement these ICE learning algorithms in a verification tool and show they are robust, practical, and effective.

1 Introduction

The problem of generating adequate inductive invariants to prove a program correct is at the heart of automated program verification. Automated program verifiers invariably synthesize invariants; for example, abstract interpretation methods find invariants using fixed-points evaluated over an abstract domain [1,2], counter-example guided predicate abstraction iteratively computes predicates and uses model-checking to establish invariants [3], etc. In deductive verification settings, traditionally, invariants are provided manually, and automated tools need to synthesize invariants to relieve the programmer of this burden [4–6]. Moreover, synthesizing invariants is the *hardest* aspect of program verification— once adequate inductive invariants are synthesized, program verification reduces to checking validity of verification conditions obtained from finite loop-free paths, which is a logic problem that has been highly automated over the years [7–9].

Invariant generation techniques can be broadly classified into two kinds: white-box techniques where the synthesizer of the invariant is acutely aware of the precise program and property that is being proved and black-box techniques where the synthesizer is largely agnostic to the structure of the program and property, but works with a partial view of the requirements of the invariant. Abstract interpretation [1], counter-example guided abstraction refinement, predicate abstraction [3, 10], the method of Craig interpolants [11, 12], IC3 [13], etc. all fall into the white-box category. In this paper, we are interested in the newly emerging black-box techniques for invariant generation.

Learning invariants: One prominent class of black-box techniques for invariant generation is the emerging paradigm of *learning*. Intuitively (see picture on right), we have two components in the verification tool: a white-box *teacher* and a black-box *learner*.



The learner synthesizes suggestions for the invariants in each round. The teacher is completely aware of the program and the property being verified, and is responsible for two things: (a) to check if a purported invariant H (for hypothesis) supplied by the

learner is indeed an invariant and is adequate in proving the property of the program (typically using a constraint solver), and (b) if the invariant is not adequate, to come up with concrete program configurations that need to be added or removed from the invariant (denoted by + and − in the figure). The learner, who comes up with the invariant H is completely agnostic of the program and property being verified, and simply aims to build a simple formula that satisfies the properties the teacher demands.

A crucial restriction here is that the teacher communicates the constraints on H using only a finite set of program configurations. When learning an invariant, the teacher and learner talk to each other in rounds, where in each round the teacher comes up with additional constraints involving new data-points and the learner replies with some set satisfying the constraints, until the teacher finds the set to be an adequate invariant. The above learning approach for invariants has been explored for quite some time in various contexts [14–16], and is gaining considerable excitement and traction in recent years [17–20].

Advantages of learning: There are many advantages the learning approach has over white-box approaches. First, a synthesizer of invariants that works cognizant of the program and property is very hard to build, simply *due* to the fact that it has to deal with the complex logic of the program. When a program manipulates complex data-structures, pointers, objects, etc. in a real language with a complex memory model and semantics, building a set that is guaranteed to be an invariant for the program gets extremely complex. However, the invariant for a loop in such a program may be much simpler, and hence a black-box technique that uses a "guess and check" approach guided by a finite set of configurations is much more light-weight and has better chances of finding the invariant. (See [5] where a similar argument is made for black-box generation of the abstract post in an abstract interpretation setting). Second, the learning procedure, by concentrating on finding the simplest concept that satisfies the constraints, implicitly provides a tactic for generalization, while white-box techniques (like interpolation) need to build in tactics to generalize. Finally, the black-box approach allows us to seamlessly integrate highly scalable machine-learning techniques into the verification framework [21, 22].

ICE-learning: The problem with the learning approach described above is that it is *broken*, as we show in this paper! In particular, the teacher *cannot* come up with examples and counter-examples alone to guide the learner towards an invariant. Approaches to learning invariants have been unduly influenced by algorithmic learning theory, automata learning, and machine learning techniques, which have traditionally offered learning from positive and negative examples. *As we show in this paper, learning using examples and counter-examples does not form a robust learning framework for synthesizing invariants.* To see why example and counter-example configurations are not sufficient, consider the following simple program—

$$pre;\ S;\ \textbf{while}\ (b)\ \textbf{do}\ L;\ \textbf{od}\ S';post$$

with a single loop body for which we want to synthesize an invariant that proves that when the pre-condition to the program holds, the post-condition holds upon exit. Assume that the learner has just proposed a particular set H as a hypothesis invariant. In order to check if H is an adequate invariant, the teacher checks three things:

(a) whether the strongest-post of the pre-condition across S implies H; if not finds a concrete data-point p and passes this as a positive example to the learner.
(b) whether the strongest-post of $(H \wedge \neg b)$ across S' implies the post-condition; if not, pass a data-point p in H that shouldn't belong to the invariant as a *negative* example.
(c) whether H is inductive; i.e., whether the strongest post of $H \wedge b$ across the loop body L implies H; if not, finds two concrete data-points p and p', where $p \in H$ and $p' \notin H$.

In the last case above, the teacher is *stuck*. Since she does not *know* the precise invariant (there are after all many), she has no way of knowing whether p should be excluded from H or whether p' should be included. In many learning algorithms in the literature [14–16, 20], the teacher cheats: she arbitrarily makes one choice and goes with that, hoping that it will result in an invariant. However, this makes the entire framework non-robust, causing divergence, blocking the learner from learning the simplest concepts, and introducing arbitrary bias that is very hard to control. If learning is to be seriously developed for synthesizing invariants, we need to fix this foundationally in the framework itself.

The main contribution of this paper is a new learning framework called ICE-learning, which stands for *learning using Examples, Counter-examples, and Implications*. We propose that we should build learning algorithms that do not take just examples and counter-examples, as most traditional learning algorithms do, but instead also handle *implications*. The teacher, when faced with a refutation of non-inductiveness of the current conjecture H in terms of a pair (p, p'), simply communicates this implication pair to the learner, demanding that the learnt set satisfies the property that if p is included in H, then so is p'. The learner makes the choice, based on considerations of simplicity, generalization, etc., whether it would include both p and p' in its set or leave p out.

We show that ICE-learning is a *robust* learning model, in the sense that the teacher can always communicate to a learner precisely why a conjecture is not an invariant (even for programs with multiple loops, nested loops, etc.). This robustness then leads to new questions that we can formulate about learning. In particular, we can ask whether the iterative learning process, for a particular learner, *strongly converges*— whether the learner will eventually learn the invariant, provided one exists expressible as a concept, no matter how the teacher gives examples, counter-examples, and implications to refute the learner's conjectures. Such questions are, of course, meaningless in the non-robust learning framework based only on examples and counter-examples.

We emphasize that the earlier works in the literature have indeed seen *inductiveness* as an important aspect of synthesizing invariants, and several mechanisms for guiding the search towards an inductive property are known [13, 23–26]. Our work here is however the first that we know that develops a robust learning model that explicitly incorporates the search for inductive sets in black-box invariant generation.

Our main contributions are as follows:

– We propose the ICE-learning framework as a robust learning framework for synthesizing invariants. We study ICE learning algorithms at two levels: ICE-learning for a particular sample as well as the *iterative* ICE model in which the teacher and learner iteratively interact to find the invariant. The complexity of the ICE-learner

for a sample, strong convergence of iterative learning, and the number of rounds of iteration required to learn are pertinent questions.

- It turns out that when the class of concepts forms a *lattice*, ICE learning already exists in the literature. In fact, the Houdini algorithm [4] and the recently proposed *abstract* Houdini algorithms in [5] and [27] for invariant synthesis over abstract numerical domain lattices are in fact ICE-learning algorithms (however, they are not typically strongly convergent and moreover, cannot learn from negative examples at all).
  We hence concentrate on *strongly convergent* ICE learning algorithms, and exhibit two such algorithms, one for numerical domains and one for linear data-structures, as described below.
- We develop a new ICE-learning algorithm for *Boolean combinations of numerical invariants*, which does not form a complete lattice (arbitrarily large subsets do not have a least upper bound). Given an ICE-sample, we show how to find the *simplest* expressible formula that satisfies the sample. Our algorithm *iterates* over all possible template formulas, growing in complexity, till it finds an appropriate formula. In order to scale well, we adapt existing template-based synthesis techniques that use a constraint solver [28–31] to a black-box ICE-learning algorithm for synthesizing invariants. We prove that the resulting iterative ICE algorithm is strongly convergent. Note that the user only specifies the logic for the invariants, and does not need to give templates.
  We build a tool over Boogie [9] for synthesizing invariants over scalar variables and show that it is extremely effective in proving a large corpus of programs correct. It outperforms most other techniques, and furthermore gives guarantees of simplicity and strong convergence that other algorithms do not.
- As a second instantiation of the ICE framework, we develop a new strongly convergent ICE-learning algorithm for *quantified* invariants. We develop a general technique of reducing ICE-learning of quantified properties to ICE-learning of quantifier-free properties, but where the latter is generalized to *sets of configurations* rather than single configurations. We instantiate this technique to build an ICE-learner for quantified properties of arrays and lists. This new learning algorithm (which is the most involved technical contribution of this paper) extends the classical RPNI learning algorithm for automata [32] to learning in the ICE-model and further learns *quantified data automata* [20], which can be converted to quantified logical formulas over arrays/lists. We build a prototype verifier by building this learner and the teacher as well, and show that this results in extremely efficient and robust learning of invariants.

Related Work: Prominent white-box techniques for invariant synthesis include abstract interpretation [1], interpolation [11, 12] and IC3 [13]. Abstract interpretation has been used for generating invariants over mostly convex domains [2, 33], some non-convex domains [34, 35] and more recently even over non-lattice abstract domains [36]. Template based approaches to synthesizing invariants using constraint solvers have been explored in a white-box setting in [28–31], and we adapt these techniques in Section 4 for developing an ICE-learning algorithm for numerical invariants. Several white-box techniques for synthesizing quantified invariants are also known. Most of them are ei-

ther based on abstract interpretation or are based on interpolation theorems for array theories [37–45].

In contrast to the above mentioned white-box techniques, there are black-box learning-based techniques for synthesizing invariants and rely-guarantee contracts. First, Daikon [46] proposed conjunctive Boolean learning to learn *likely* invariants from program configurations recorded along test runs. Learning was introduced in the context of verification by Cobleigh et al. [14], which was followed by applications of Angluin's L^* algorithm [47] to finding rely-guarantee contracts [15] and stateful interfaces for programs [16]. Houdini [4] uses essentially conjunctive Boolean learning (which can be achieved in polynomial time) to learn conjunctive invariants over templates of atomic formulas. In Section 3, we show that the Houdini algorithm along with its generalization by Thakur et al. [5] and [27] to arbitrary abstract domains like intervals, octagons, polyhedrons, linear equalities, etc. can be adapted to ICE-learning algorithms.

Recently, there has been a renewed interest in the application of learning to program verification, in particular to synthesize invariants [17–19] by using scalable machine learning techniques [21,22] to find classifiers that can separate good states that the program can reach (positive examples) from the bad states the program is forbidden from reaching (counter-examples). Quantified *likely* invariants for linear data-structures and arrays are found from dynamic executions using learning in [20], but these aren't necessarily adequate. Boolean formula learning has also been applied recently for learning quantified invariants in [48]. In addition, learning has been applied towards inductive program synthesis [49,50] and model extraction and testing of software [51,52].

Counterexamples to inductiveness of an invariant have been handled in the past [24–26], but only in the context of lattice domains where the learned concepts grow monotonically and implications essentially yield positive examples. Recently, [23] tries to find inductive invariants by finding *common* interpolants for same program locations. Though [18] mentions a heuristic for handling implication samples in their algorithm for learning invariants their tool does not implement that heuristic. As far as we know, our work here is the first to explicitly incorporate the search for inductive sets in black-box invariant generation.

2 Illustrative Example

Consider the C program on the right. This program requires a scalar invariant ($i > p \Rightarrow j = 1$) for its verification using VCC [53]. Even to synthesize such a scalar invariant, white-box techniques would need to reason about the array `a` in the program, and in general have to deal with language features like objects, pointers, a complex memory model and its semantics, etc. A black-box approach can however learn such an invariant from a small set of program configurations.

Consider a black-box engine that calls `foo` with the values for p– $25, 26, \ldots$ and that unrolls the loop a few times to find positive examples for (i, j, p) in the kind $(0,0,25), (1,0,25), (1,1,25), \ldots$ for a small number of

```
#include <vcc.h>
int foo(int a[], int p)
_(requires (p>=25 && p<75))
_(requires a[p]==1)
_(requires \thread_local_array(a, 100))
{
  int i=0, j=0;
  while (i<100)
  _(invariant (i>p ==> j==1))
  {
    if (a[i]==1)
       j = 1;
    i = i+1;
  }
  _(assert j==1);
}
```

values of i, and counter-examples of the form $(100, 0, 25)$, $(100, 2, 25)$, $\ldots$ $(99, 0, 25)$, $(99, 2, 25)$, $\ldots$ (values close to 100 for i and different from 1 for j). From these positive and negative examples, the learner could naturally come up with a conjecture such as (i > 50 => j = 1) (machine learning algorithms tend to come up with such an invariant).

Now notice that the teacher is *stuck* as all positive and negative examples are satisfied by the conjecture, though it is not inductive. Consequently, when using a learner from only positive and negative samples, the teacher cannot make progress. However, in ICE learning, the teacher can give an implication pair, say of the form $((50, 0, 25), (51, 0, 25))$, and proceed with the learning. Hence we can make progress in learning, and a learner that produces the simplest conjectures satisfying the samples would eventually generalize a large enough sample to come up with a correct invariant. Our tool from section 4 in fact precisely learns the above mentioned invariant for this program.

3 ICE-Learning

When defining a (machine) learning problem, one usually specifies a domain D (like points in the real plane or finite words over an alphabet), and a class of concepts C (like rectangles in the plane or regular languages), which is a class of subsets of the domain. In classical learning frameworks (see [22]), the teacher provides a set of (positive) examples in D that are part of the target concept, and a set of counter-examples (or negative examples) in D that are not part of the target concept. Based on these examples and counter-examples, the learner has to construct a hypothesis, which best approximates the actual target concept the teacher has in mind.

ICE learning: In our setting, the teacher does *not* have a precise target concept from C in mind, but is looking for an inductive set which meets certain additional constraints. Consequently, we extend this learning setting with a third type of information that can be provided by the teacher: implications. Formally, let D be some domain and $C \subseteq 2^D$ be a class of subsets of D, called the concepts. The teacher knows a triple (P, N, R), where $P \subseteq D$ is an (infinite) set of positive examples, $N \subseteq D$ is an (infinite) set of counter-examples (or negative examples), and $R \subseteq D \times D$ is a relation interpreted as an (infinite) set of implications. We call (P, N, R) the *target description*, and these sets are typically *infinite* and are obtained from the program, but the teacher has the ability to query these sets effectively.

The learner is given a finite part of this information (E, C, I) with $E \subseteq P$, $C \subseteq N$, and $I \subseteq R$. We refer to (E, C, I) as an (ICE) *sample*. The task of the ICE-learner is to construct *some* hypothesis $H \in C$ such that $P \subseteq H$, $N \cap H = \emptyset$, and for each pair $(x, y) \in R$, if $x \in H$, then $y \in H$. A hypothesis with these properties is called a *correct hypothesis*. Note that a target description (P, N, R) may have several correct hypotheses (while H must include P and exclude R and be R-closed, there can be several such sets).

Iterative ICE learning: The above ICE-learning corresponds to a passive learning setting, in which the learner does not interact with the teacher. In general, the quality of the hypothesis will heavily depend on the amount of information contained in the sample. However, when the hypothesis is wrong, we would like the learner to gain

information from the teacher using new samples. Since such a learning process proceeds in rounds, we refer to it as *iterative ICE-learning*.

The iterative ICE learning happens in rounds, where in each round, the learner starts with some sample (E, C, I) (from previous rounds or an initialization) and constructs a hypothesis $H \in C$ from this information, and asks the teacher whether this is correct. If the hypothesis is correct (i.e., if $P \subseteq H$, $H \cap N = \emptyset$, and for every $(x, y) \in R$, if $x \in H$, then $y \in H$ as well), then the teacher answers "correct" and the learning process terminates. Otherwise, the teacher returns either some element $d \in D$ with $d \in P \setminus H$ or $d \in H \cap N$, or an implication $(x, y) \in R$ with $x \in H$ and $y \notin H$. This new information is added to the sample of the learner.

The learning proceeds in rounds and when the learning terminates, the learner has learnt *some* R-closed concept that includes P and excludes N.

Aside: In more traditional active learning, especially in automata theory, learners are allowed to ask *membership* queries (asking whether a particular element in the domain belongs to the target concept). However, in our setting the teacher does not have a precise target concept in mind and therefore cannot, in general, answer membership queries. The only query we can expect the teacher to answer is the correctness query above.

Using ICE Learning to Synthesize Invariants: Honesty and Progress
Given an ICE-learning algorithm for a concept class, we can build algorithms for synthesizing invariants that fall into this concept class by building a (white-box) teacher that can check whether hypotheses given by the learner are adequate invariants, and if not, find concrete examples, counter-examples, and implications to explain why the hypothesis is not an invariant.

We can apply such learning for finding invariants in programs with multiple loops, nested loops, etc. The learning of the invariant will simultaneously learn all these invariant annotations. The teacher can check the hypotheses by generating verification conditions for the hypothesized invariants and by using automatic theorem provers to check their validity. In general, the positive examples will arise from propagating the pre-condition for the program across code, the negative examples will arise from (weakest pre-conditions of) the specifications, including the assertions and post-condition of the program, and the implications will arise from non-inductiveness of the hypothesis.

The salient feature of ICE-learning is it ensures *progress*: no matter which verification conditions are proven invalid, the teacher can always add an example/counter-example/implication to the sample such that H (and any other hypothesis returned in earlier rounds) does not satisfy the new sample. Furthermore, while augmenting the sample, the teacher does not preclude *any* possible adequate invariant of the program in future. In other words, the teacher doesn't lie in any round, and gives only constraints consistent with *all* adequate invariants; we call this *honesty*. These two salient properties of progress and honesty are what makes ICE-learning of invariants robust: the teacher can always make progress and never has to make arbitrary choices that causes valid inductive invariants to be excluded. When using learning just from positive and negative examples, progress and honesty cannot both be maintained (when the teacher finds no new examples or counter-examples to furnish and the hypothesis is not inductive, there is no way to make progress without making an arbitrary dishonest choice).

Note that honesty and progress do not, of course, imply convergence to an adequate invariant.

Convergence: The setting of iterative ICE-learning naturally raises the question of convergence of the learner, that is, does the learner find a correct hypothesis in a finite number of rounds? We say that a learner *strongly converges*, if for every target description (P, N, R) it reaches a correct hypothesis (from the empty sample) after a finite number of rounds, no matter what information is provided by the teacher (of course, the teacher has to answer correctly according to the target description (P, N, R)).

Note that the definition above demands convergence for arbitrary triples (P, N, R), and allows the teacher in each round to provide *any* information that contradicts the current hypothesis, and is hence a very strong property.

Observe now that for a *finite* class C of concepts, a learner strongly converges if it never constructs the same hypothesis twice. This assumption on the learner is satisfied if it only produces hypotheses H that are consistent with the sample (E, C, I), that is, if $E \subseteq H$, $C \cap H = \emptyset$, and for each pair $(x, y) \in I$, if $x \in H$, then $y \in H$. Such a learner is called a *consistent learner*. Since the teacher always provides a witness for an incorrect hypothesis, the next hypothesis constructed by a consistent learner must be different from all the previous ones.

Lemma 1. *For a finite class C of concepts, every consistent learner strongly converges.*

For various iterative ICE algorithm classes, where class of concepts may be infinite, we will study strong convergence.

ICE Learning over Lattice Domains: It turns out that (non-iterative) ICE algorithms are especially easy to build when the class of concepts forms a lattice, as typical in an abstract interpretation setting.

Consider an abstract domain that is a lattice. Then given any sample (E, C, I), we can compute the *best* (smallest) abstract element that satisfies the constraints (E, C, I) as follows. First, we take the least upper bound of the set of all $\alpha(e)$, for each $e \in E$. Then we see if these satisfy the implication constraints; if not, then for every pair $(p, p') \in I$ that is not satisfied, we know that p' *must* be added to the set (since p belongs to every set that includes E). Hence all these elements p' can be added by applying α to them, and we can take the lub with respect to the existing set. We continue in this fashion till we converge to an abstract element that is the smallest satisfying E and I. Now, we can check if C is excluded from it; if yes, we have computed the best set, else there is no set satisfying the constraints. The above is an ICE-algorithm for any abstract domain.

We can, using this argument, establish polynomial-time (non-iterative) ICE learning algorithms for conjunctive formulas (in fact, this is what the classical Houdini algorithm does [4, 22]), k-CNF formulas [22], and for abstract domains such as octagons, polyhedra, etc. as in [24, 25]

However, note that the iterative extension of the above ICE algorithm may not halt (unless the domain has finite height). One can of course use a widening heuristically after some rounds to halt, but then clearly the iterative ICE algorithm will not be necessarily *strongly convergent*. In fact, the iterative ICE algorithm with widening is, in

fact, precisely the *abstract Houdini* algorithm proposed recently in [5], and is similar to another recent work in [27]. Note, however, that these are not *strongly convergent* iterative ICE learning schemes.

The iterative ICE learning algorithms we develop in this paper *are strongly convergent*. While the above derived iterative ICE algorithms essentially ignore counter-examples, and fail to use counter-examples and implications as a way to *come down* the lattice after a widening/over-generalization, the algorithms we propose in the next two sections are more general schemes that truly utilize examples, counter-examples, and implications to find succinct expressions.

4 An ICE-learning algorithm for Numerical Invariants

In this section, we describe a learning algorithm for synthesizing invariants that are arbitrary Boolean combinations of numerical atomic formulas. Since we want the learning algorithm to generalize the sample (and not capture precisely the finite set of implication-closed positive examples), we would like it to learn a formula with the *simplest* Boolean structure. In order to do so, we iterate over templates over the Boolean structure of the formulas, and learn a formula in the given template.

Note that the domain is a join-semilattice (every pair of elements has a least upper bound) since formulas are closed under disjunction. Hence we can employ the generic abstract Houdini algorithm [5] to obtain a passive ICE learning algorithm. However, using the vanilla algorithm will learn only the precise set of positive and implication-closed set, and hence not generalize without a widening. Widening for disjunctive domains is not easy, as there are several ways to generalize disjunctive sets [54]. Furthermore, even with a widening, we will not get a *strongly convergent* iterative ICE algorithm that we desire (see experiments in this section where abstract Houdini diverges even on conjunctive domains on some programs for this reason). The algorithm we build in this section will not only be strongly convergent but also will produce the *simplest* expressible invariant.

Let $Var = \{x_1, \cdots, x_n\}$ be the set of (integer) variables in the scope of the program. For simplicity, let us restrict atomic formulas in our concept class to octagonal constraints, over program configurations, of the general form:

$$s_1 v_1 + s_2 v_2 \leq c, \;\; s_1, s_2 \in \{0, +1, -1\}, \;\; v_1, v_2 \in Var, \; v_1 \neq v_2, \;\; c \in \mathbb{Z}.$$

(We can handle more general atomic formulas as well; we stick to the above for simplicity and effectiveness.)

Our ICE learning algorithm will work by iterating over more and more complex *templates* till it finds the simplest formula that satisfies the sample. A *template* fixes the Boolean structure of the desired invariants and also restricts the constants $c \in \mathbb{Z}$ appearing in the atomic formulas to lie within a finite range $[-M, +M]$, for some $M \in \mathbb{Z}^+$. Bounding the constants leads to strong convergence as we show below. For a given template $\bigvee_i \bigwedge_j \alpha^{ij}$, the iterative ICE-learning algorithm we describe below learns an adequate invariant φ, of the form:

$$\varphi(x_1, \cdots, x_n) = \bigvee_i \bigwedge_j (\; s_1^{ij} \, v_1^{ij} \; + \; s_2^{ij} \, v_2^{ij} \; \leq \; c^{ij} \;), \quad | \, c^{ij} \, | \; \leq \; M.$$

Given a sample (E, C, I), the learner iterates through templates, and for each template, tries to find concrete values for s_k^{ij}, v_k^{ij} ($k \in \{1, 2\}$) and c^{ij} such that the formula φ is consistent with the sample; i.e., for every data-point $p \in E$, $\varphi(p)$ holds; for $p \in C$, $\varphi(p)$

does not hold; and for every implication pair $(p, p') \in I$, $\varphi(p')$ holds if $\varphi(p)$ holds. Unfortunately, finding these values in the presence of implications is hard; classifying each implication pair (p, p') as both positive or p as negative tends to create an exponential search space that is hard to search efficiently. Our ICE-learner uses a constraint solver to search this exponential space in a reasonably efficient manner. It does so by checking the satisfiability of the formula Ψ (below), over free variables s_k^{ij}, v_k^{ij} and c^{ij}, which precisely captures all the ICE-constraints. In this formula, b_p is a Boolean variable which tracks $\varphi(p)$; the Boolean variables b_p^{ij} represent the truth value of $(s_1^{ij}\, v_1^{ij} + s_2^{ij}\, v_2^{ij} \leq c^{ij})$ on point p, r_{kp}^{ij} encode the value of $s_k^{ij} \cdot v_k^{ij}$ (line 2 of the formula); and d_{kp}^{ij} encode the value of v_k^{ij} (line 3).

$$\Psi(s_k^{ij}, v_k^{ij}, c^{ij}) \equiv \left(\bigwedge_{p\in E} b_p\right) \wedge \left(\bigwedge_{p\in C} \neg b_p\right) \wedge \left(\bigwedge_{(p,p')\in I} b_p \Rightarrow b_{p'}\right) \wedge \left(\bigwedge_{p} \left(b_p \Leftrightarrow \bigvee_i \bigwedge_j b_p^{ij}\right)\right) \wedge$$

$$\left(\bigwedge_{p,i,j} \left(b_p^{ij} \Leftrightarrow \left(\sum_{k\in\{1,2\}} r_{kp}^{ij} \leq c^{ij}\right)\right)\right) \wedge \left(\bigwedge_{i,j} \left(-M \leq c^{ij} \leq M\right)\right) \wedge \left(\bigwedge_{\substack{p,i,j \\ k\in\{1,2\}}} \begin{pmatrix} s_k^{ij} = 0 \Rightarrow r_{kp}^{ij} = 0 \\ s_k^{ij} = 1 \Rightarrow r_{kp}^{ij} = d_{kp}^{ij} \\ s_k^{ij} = -1 \Rightarrow r_{kp}^{ij} = -d_{kp}^{ij} \end{pmatrix}\right) \wedge$$

$$\left(\bigwedge_{\substack{p,i,j \\ k\in\{1,2\}}} \bigwedge_{l\in[1,n]} (v_k^{ij} = l \Rightarrow d_{kp}^{ij} = p(l))\right) \wedge \left(\bigwedge_{\substack{i,j \\ k\in\{1,2\}}} \left(-1 \leq s_k^{ij} \leq 1\right)\right) \wedge \left(\bigwedge_{\substack{i,j \\ k\in\{1,2\}}} (1 \leq v_k^{ij} \leq n)\right) \wedge \left(\bigwedge_{i,j} (v_1^{ij} \neq v_2^{ij})\right)$$

Note that Ψ falls in the theory of quantifier-free linear integer arithmetic, the satisfiability of which is decidable. A satisfying assignment for Ψ gives a *consistent* formula that the learner conjectures as an invariant. If Ψ is unsatisfiable, then there is no invariant for the current template consistent with the given sample. In this case we iterate by increasing the complexity of the template. For a given template, the class of formulas conforming to the template is finite. Our enumeration of templates *dovetails* between the Boolean structure and the range of constants in the template, thereby progressively increasing the complexity of the template. Consequently, the ICE learning algorithm always synthesizes a consistent hypothesis if there is one, and furthermore synthesizes a hypothesis of the simplest template.

A similar approach can be used for learning invariants over linear constraints, and even more general constraints if there is a solver that can effectively solve the resulting theory.

Convergence: Our iterative ICE algorithm conjectures a consistent hypothesis in each round, and hence ensures that we do not repeat hypotheses. Furthermore, the enumeration of templates using dovetailing ensures that all templates are eventually considered, and together with the fact that there are a finite number of formulas conforming to any template ensures strong convergence.

Theorem 1. *The above ICE-learning algorithm always produces consistent conjectures and the corresponding iterative ICE algorithm strongly converges.*

Our learning algorithm is quite different from earlier white-box constraint based approaches to invariant synthesis [28–31]. These approaches directly encode the adequacy of the invariant (encoding the entire program's body) into a constraint, and use Farkas' lemma to reduce the problem to satisfiability of quantifier-free non-linear arithmetic formulas, which is harder and in general undecidable. On the other hand, we split

the task between a white-box teacher and a black-box learner, communicating only through ICE-constraints on concrete data-points. This greatly reduces the complexity of the problem, leading to a simple teacher and a much simpler learner. Our idea is more similar to [19] which use algebraic techniques to guess the coefficients.

Program	Invariant	White Box		Black Box			Program	Invariant	White Box		Black Box		
		InvGen [31]	CPA [55]	absH [5]	ML [18]	ICE			InvGen [31]	CPA [55]	absH [5]	ML [18]	ICE
w1 [29]	$x \leq n$	0.1	×	0.1	0.2	**0.0**	w2 [29]	$x \leq n-1$	0.1	×	0.2	0.1	**0.0**
fig6 [56]	*true*	0.1	1.3	0.1	0.1	**0.0**	fig1 [29]	$x \leq -1 \vee y \geq 1$	×	4.5	×	×	**0.1**
fig8 [56]	*true*	0.0	1.4	0.0	0.0	**0.0**	fig3 [56]	$lock = 1 \vee x \leq y-1$	0.1	1.4	×	0.1	**0.0**
ex14 [57]	$x \geq 1$	×	1.5	0.2	0.2	**0.0**	fig9 [56]	$x = 0 \wedge y \geq 0$	0.1	1.4	0.0	0.2	**0.0**
finf1	$x = 0$	0.1	1.5	0.1	0.4	**0.0**	ex23 [57]	$0 \leq y \leq z \wedge$ $z \leq c + 4572$	×	90.5	0.2	×	**14.2**
finf2	$x = 0$	0.1	1.4	0.0	0.1	**0.0**							
sum3	$sn = x$	0.1	1.5	0.1	0.1	**0.0**	ex7 [57]	$0 \leq i \wedge y \leq len$	×	1.6	0.2	0.4	**0.0**
term2	*true*	0.0	1.6	0.0	0.0	**0.0**	sum1	$sn = i-1 \wedge$ $(sn = 0 \vee sn \leq n)$	×	×	×	×	**1.8**
term3	*true*	0.0	1.4	0.0	0.0	**0.0**							
trex1	$z >= 1$	0.1	1.5	0.1	0.4	**0.0**	sum4	$sn = i-1 \wedge sn \leq 8$	0.1	2.8	×	×	**2.6**
trex2	*true*	0.0	1.4	0.0	0.0	**0.0**	tcs [12]	$i \leq j-1 \vee i \geq j+1 \vee$ $x = y$	0.1	1.4	×	0.5	**1.4**
trex4	*true*	0.0	1.4	0.0	0.0	**0.0**							
winf1	$x = 0$	0.0	1.4	0.0	0.0	**0.0**	trex3	$0 \leq x1 \wedge 0 \leq x2 \wedge$ $0 \leq x3 \wedge d1 = 1 \wedge$ $d2 = 1 \wedge d3 = 1$	0.5	×	×	×	**2.2**
winf2	$x = 0$	0.0	1.4	0.0	0.0	**0.0**							
winf3	$x = 0$	×	1.4	0.3	0.1	**0.1**							
vmail	$i \geq 0$	×	1.4	0.1	0.3	**0.0**	matrix	$a[0][0] \leq m \vee j \leq 0;$ $a[0][0] \leq m \vee j + k \leq 0$	×	×	×	×	**5.8**
lucmp	$n = 5$	×	77.0	0.0	0.1	**0.0**							
n.c11	$0 \leq len \leq 4$	0.1	2.2	×	0.2	**0.6**	cgr2 [29]	$N \leq 0 \vee (x \geq 0 \wedge$ $0 \leq m \leq N-1)$	×	1.8	×	×	**7.3**
cgr1 [29]	$x - y \leq 2$	0.1	1.5	0.1	0.2	**0.0**	array	$j \leq 0 \vee m \leq a[0]$	×	×	×	0.2	**0.3**
oct	$x + y \leq 2$	0.0	1.3	0.2	0.1	**0.2**	vbsd	$pathlim \leq tmp$	×	1.6	0.5	×	**0.0**

Table 1. Results for ICE-learning numerical invariants. ICE is the total time taken by our tool. All times reported are in seconds. × means an adequate invariant was not found.

Experimental Results: We have implemented our learning algorithm as an invariant synthesis tool in Boogie [9]. In our tool we enumerate templates in an increasing order of their complexity. For a given Boolean structure of the template B_i, we fix the range of constants M in the template to be the greater value out of i and the maximum integer in the program. If an adequate invariant is not found, we increase i. If an adequate invariant is found, we use binary search on M to find an invariant that has the same Boolean structure but the smallest constants. This enumeration of templates is complete and it ensures that we learn the *simplest* invariant. In our tool, ICE samples discovered while learning an invariant belonging to a simpler template are not wasted but used in subsequent rounds. As already mentioned, our learner uses an incremental Z3 [58] solver that adds a new constraint for every ICE sample discovered by the Boogie based teacher. A graphical representation of an ICE sample where implications dictate the invariant learnt by our tool is shown in Figure 2 in the Appendix.

We evaluate our tool on SV-COMP benchmarks[1] and several other programs from the literature (see Table 1). We use SMACK [59] to convert C programs to Boogie and use our tool to learn loop invariants for the resulting Boogie programs. We use inlining to infer invariants for programs with multiple procedures. In Table 1 we compare our tool to invariant synthesis using abstractHoudini [5] (called absH in Table 1), [18]

[1] https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp13/loops/

(called ML), Invgen [31] and interpolation based Impact algorithm [60] implemented in CPAchecker (called CPA) [55]. We implemented the octagonal domain in abstractHoudini for a comparison with our tool. As mentioned in Section 3, abstractHoudini is an ICE learning algorithm but is not strongly convergent. Unlike our tool, abstractHoudini is not able to learn disjunctive octagonal invariants. In addition, it is unable to prove programs like trex3 and n.c11 where it loses precision due to widening. InvGen [31] uses a white-box constraint-based approach to invariant synthesis. Unlike our tool that enumerates all templates, InvGen requires the user to specify a template for the invariants. Being white-box, it cannot handle programs with arrays and pointers, even if the required invariants are numerical constraints over scalar variables. Being incomplete, it is also unable to prove several scalar programs like fig1 and cegar2. Finally, [18] is a machine learning algorithm for inferring numerical invariants. From our experience, the inference procedure in [18] is very sensitive to the test harness used to obtain the set of safe/unsafe program configurations. For several programs, we could not learn an adequate invariant using [18] despite many attempts with different test harnesses.

The experiments show that our tool outperforms [5, 18, 31, 55] on most programs, and learns an adequate invariant for all programs in reasonable time. Though we use the more complex but more robust framework of ICE-learning that promises to learn the simplest invariants and is strongly convergent, it is generally faster than other learning algorithms like [17, 18] that learn invariants from just positive and negative examples, and lack any such promises.

5 Learning Universally Quantified Properties

In this section we describe a setting of ICE-learning for *universally quantified* concepts. For a typical such scenario consider programs that manipulate dynamic heaps. A configuration of a program can be described by the heap structure (locations, the various field-pointers, etc.), and a finite set of pointer variables pointing into the heap. Since the heap is unbounded, typical invariants for programs manipulating heaps require universally quantified formulas. For example, a list is sorted if the data at all pairs y_1, y_2 of successive positions is sorted correctly. In general, we consider universal properties of the form $\psi = \forall y_1, \ldots, y_k \varphi(y_1, \ldots, y_k)$, where φ is a quantifier-free formula. We now describe how to modify the learning setting such that we can use a learner for the quantifier-free property described by $\varphi(y_1, \ldots, y_k)$.

We consider for each concrete program configuration c the set S_c of *valuation configurations* of the form (c, val), where val is a valuation of the variables $y_1, \ldots, y_k$. For example, if the configurations are heaps, then the valuation maps each quantified variable y_i to a cell in the heap, akin to a scalar pointer variable. Then $c \models \psi$ if $(c, val) \models \varphi$ for all valuations val, and $c \not\models \psi$ if $(c, val) \not\models \varphi$ for some valuation val.

This leads to the setting of *data-set based ICE-learning*. In this setting, the target description is of the form $(\hat{P}, \hat{N}, \hat{R})$ where $\hat{P}, \hat{N} \subseteq 2^D$ and $\hat{R} \subseteq 2^D \times 2^D$. A hypothesis $H \subseteq D$ is correct if $P \subseteq H$ for each $P \in \hat{P}$, $N \not\subseteq H$ for each $N \in \hat{N}$, and for each pair $(X, Y) \in \hat{R}$, if $X \subseteq H$, then also $Y \subseteq H$. The sample is a finite part of the target description, that is, it is of the form $(\hat{E}, \hat{C}, \hat{I})$, where $\hat{E}, \hat{C} \subseteq 2^D$, and $\hat{I} \subseteq 2^D \times 2^D$.

An ICE-learner for the data-set based setting corresponds to an ICE-learner for universally quantified concepts in the original data-point based setting using the following

connection. Given a standard target description (P, N, R) over D, we now consider the domain D_{val} extended with valuations of the quantified variables $y_1, \ldots, y_k$ as described above. Replacing each element c of the domain by the set $S_c \subseteq D_{val}$ transforms (P, N, R) into a set-based target description. Then a hypothesis H (described by a quantifier-free formula $\varphi(y_1, \ldots, y_k)$) is correct w.r.t. the set-based target description iff the hypothesis described by $\forall y_1, \ldots, y_k \varphi(y_1, \ldots, y_k)$ is correct w.r.t. the original target description. Unlike [40] that uses "Skolem constants", learning over data-sets allows us to learn from not only examples, but also from counter-examples and implications (where simple Skolem constants will not work).

Recap of Quantified Data Automata and related results [20]:
We will develope ICE learning algorithms for universally quantified invariants over arrays and lists that can be expressed by an automaton model called quantified data automata (QDA) introduced by Garg et al in [20]. We briefly recall the main ideas concerning this model and refer the reader to Appendix C and [20] for more detailed definitions.

Let us illustrate QDAs with an example. Consider a typical invariant in a sorting program over an array A: $\forall y_1, y_2.((0 \leq y_1 \wedge y_2 = y_1 + 1 \wedge y_2 \leq i) \Rightarrow A[y_1] \leq A[y_2])$. This says that for all successive cells y_1, y_2 that occur somewhere in the array A before the cell pointed to by a scalar pointer variable i, the data value stored at y_1 is no larger than the data value stored at y_2.

We model arrays (or other linear data structures) by *data words*, in which each position corresponds to an element or cell in the data structure. Each position in such a word is labeled with a tuple of a set of pointer variables of the program that indicates their position in the data structure and a data value from some data domain (e.g., integers) that indicates the value contained in the cell of the data structure. A QDA defines a set of data words. However, to capture the idea of expressing universally quantified properties, a QDA reads *valuation words*, which are additionally annotated with universally quantified variables. The alphabet of a QDA is a pair in which the first component corresponds to the pointer variables, and the second component contains the universally quantified variable at that position (if any).

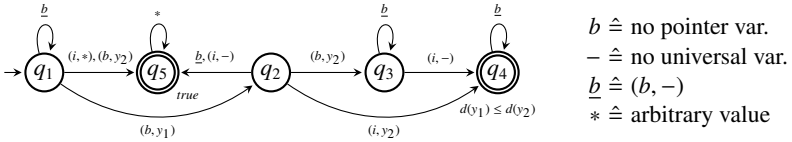


Fig. 1. An example QDA representing an invariant of a sorting routine.

The sortedness invariant above is captured by the QDA in Figure 1 (some abbreviations for transition labels are explained on the right side of the figure). The QDA accepts a valuation word if the data values at the positions of y_1 and y_2 satisfy the formula at the final state it reaches. Moreover, the automaton accepts a data word w if for *all* possible valuations of y_1 and y_2, the automaton accepts the corresponding valuation word.

We assume that the set of formulas used to label the final states of a QDA forms a finite lattice in which the order $\sqsubseteq$ is compatible with implication of formulas, that is, if $\varphi_1 \sqsubseteq \varphi_2$, then $\varphi_1 \Rightarrow \varphi_2$.

In [20] the subclass of elastic QDAs (EQDAs) is considered because they have a decidable emptiness problem and can be translated into decidable logics, like the array property fragment (APF) [61] for arrays, or a decidable fragment of the logic STRAND [62] for lists. The key property of these logics is their inability to express that quantified variables are only a bounded distance away from each other. This is captured at the automaton level by only allowing self loops on $\underline{b}$ in EQDAs. The example QDA in Figure 1 is *not* an elastic QDA because there is a $\underline{b}$-transition from q_2 to q_5. However, there is an EQDA for an equivalent invariant in which the sortedness property is checked for every pair of cells y_1, y_2 such that $y_1 \leq y_2$. Note that since each variable can occur only once, the blank symbol is the only one that can appear arbitrarily often in an input word. Therefore, there are only finitely many EQDAs for a fixed alphabet (set of variables). We refer the reader to [20] for more details on EQDAs.

ICE-learning algorithms for EQDAs. The goal of this section is to develop an iterative ICE-learner for concepts represented by EQDAs. The first relevant question is whether there is a *polynomial time* iterative ICE-learner. We show that this is *impossible* when the set of pointers and quantified variables is unbounded (see Appendix B for a proof sketch).

Theorem 2. *There is no polynomial time iterative ICE-learner for EQDAs, when the alphabet size is unbounded.*

The theorem can be proven by adapting a result from [63], namely that there is no polynomial time learning algorithm for DFAs that only uses equivalence queries. This shows that there is no hope of obtaining an iterative ICE-learner for EQDAs (or even QDAs) in the style of the well-known L^* algorithm of Angluin, which learns DFAs in polynomial time using equivalence and membership queries.

Though we cannot hope for a polynomial time iterative ICE-learner, we develop a (non-iterative) ICE-learner that constructs an EQDA from a given sample in polynomial time. In the iterative setting this yields a learner for which each round is polynomial, while the number of rounds is not polynomial, in general. Our ICE-learning algorithm is adapted from the classical RPNI passive learning algorithm [32], which takes as input a sample (E, C) of positive example words E and counter-example words C, and constructs a DFA consistent with (E, C); that is, the resulting DFA accepts all words in E and rejects all words in C.

RPNI can be viewed as an instance of an abstract state merging algorithm that is sketched as Algorithm 1. In this general setting, the algorithm takes a finite collection $\mathcal{S}$ of data, called *sample*, as input (we make this precise later) and produces a *Moore machine* (i.e., a DFA with output) that satisfies a given (decidable) property p that depends on $\mathcal{S}$ as output. In the case of RPNI, $\mathcal{S} = (E, C)$ consists of two finite sets of words, the resulting Moore machine is interpreted as a DFA, and the property p states that all words in E are to be accepted whereas all words in C are to be rejected.

Algorithm 1 proceeds in two consecutive phases. In Phase 1 (Lines 1 and 2), the algorithm calls `init(`$\mathcal{S}$`)` to construct an initial Moore machine $\mathcal{A}_{\text{init}}$ from $\mathcal{S}$ that satisfies p (assuming that this is possible). Then, it picks a total order $q_0 < \ldots < q_n$ on the states of $\mathcal{A}_{\text{init}}$, which determines the order in which the states are to be merged in the subsequent phase. The actual state merging then takes place in Phase 2 (Lines 3

to 14). According to the given order, Algorithm 1 tries to merge each state q_i with a "smaller" state q_j (i.e., $j < i$) and calls `test` on the resulting Moore machine to check whether this machine still satisfies p; since a merge can cause nondeterminism, it might be necessary to merge further states in order to restore determinism. A merge is kept if the Moore machine passes `test` (otherwise the merge is discarded), which guarantees that the final Moore machine still satisfies p. Note that we represent merging of states by means of a congruence relation $\sim \subseteq Q \times Q$ over the states (i.e., $\sim$ is an equivalence relation that is compatible with the transitions) and the actual merging operation as constructing the quotient Moore machine $\mathcal{A}_{\text{init}}/_{\sim}$ in the usual way. Note that in the case of DFAs each merge increases the language and thus can be seen as a generalization step in the learning algorithm.[2]

Algorithm 1: Generic State Merging algorithm.

Input: A sample $\mathcal{S}$
Output: A Moore machine $\mathcal{A}$ that passes `test`$(\mathcal{A})$

```
1  A_init = (Q, Σ, Γ, q_0, δ, f) ← init(S);
2  (q_0, ..., q_n) ← order(Q);
3  ~_0 ← {(q, q) | q ∈ Q};
4  for i = 1, ..., n do
5      if q_i ≁_{i-1} q_j for all j ∈ {0, ..., i − 1} then
6          j ← 0;
7          repeat
8              Let ~ be the smallest congruence that
               contains ~_{i-1} and the pair (q_i, q_j);
9              j ← j + 1;
10         until test(A_init/~);
11         ~_i ← ~;
12     else
13         ~_i ← ~_{i-1};
14 end
15 return A_init/~_n;
```

We are now ready to describe how to instantiate Algorithm 1 to work in the EQDA setting. In this setting, a sample is of the form $(\hat{E}, \hat{C}, \hat{I})$ where $\hat{E}, \hat{C}$ are sets of sets of valuation words and $\hat{I}$ contains pairs of sets of valuation words, and the task is to compute an EQDA that is consistent with the given sample. From [20] we know that EQDAs can be viewed as Moore machines that read valuation words and output data formulas. Hence we can adapt the RPNI algorithm to learn EQDAs as explained below.

For the initialization `init`$(\mathcal{S})$ we construct an EQDA whose language is the smallest (w.r.t. inclusion) EQDA-definable language that is consistent with the sample $\mathcal{S}$. For this purpose, we consider the set of all positive examples, that is, the set $E := \bigcup \hat{E}$. This is a set of valuation words, from which we strip off the data part, obtaining a set E' of symbolic words only made up of pointers and universally quantified variables. We start with the prefix tree of E' using the prefixes of words in E' as states (as the original RPNI does). The final states are the words in E'. Each such word $w \in E'$ originates from a set of valuation words in E (all the extensions of w by data that result in a valuation word in E). If we denote this set by E_w, then we label the state corresponding to w with the least formula that is satisfied in all valuation words in E_w (recall that the formulas form a lattice). This defines the smallest QDA-definable set that contains all words in E. If this QDA is not consistent with the sample, then either there is no such QDA, or the QDA is not consistent with an implication, that is, for some $(X, Y) \in \hat{I}$ it accepts everything in X but not everything in

[2] We refer the reader to Appendix C for an in-depth description of the generic state merging algorithm and its instantiations.

Y. In this case, we add X and Y to $\hat{E}$ and restart the construction (because every QDA consistent with the sample needs to accept all of X and all of Y).

To make this QDA $\mathcal{A}$ elastic, all states that are connected by a $\underline{b}$-transition are merged. This defines the smallest EQDA-definable set that contains all words accepted by $\mathcal{A}$ (see [20]). Hence, if this EQDA is not consistent with the sample, then either there is no such EQDA, or an implication $(X, Y) \in \hat{I}$ is violated, and we proceed as above by adding X and Y to $\hat{E}$ and restarting the computation. The result of this adapted initialization is an EQDA whose language is the *smallest EQDA-definable language that is consistent with the sample.*

Once Phase 1 is finished, our algorithm proceeds to Phase 2, in which it successively merges states of $\mathcal{A}_{\text{init}}$, to obtain an EQDA that remains consistent with the sample but has less states. When merging accepting states, the new formula at the combined state is obtained as the least upper bound of the formulas of the original states. Note that merging states of an EQDA preserves the self-loop condition for $\underline{b}$-transitions. Finally, the `test` routine simply checks whether the merged EQDA is consistent with the sample.

It follows that the hypothesis constructed by this adapted version of RPNI is an EQDA that is consistent with the sample. Hence we have described a consistent learner. For a fixed set of pointer variables and universally quantified variables there are only a finite number of EQDAs, and therefore by Lemma 1 we conclude that the above learning is strongly convergent (though number of rounds need not be polynomial).

Theorem 3. *The adaption of the RPNI algorithm for iterative set-based ICE-learning of EQDAs strongly converges.*

Experiments: We built a prototype of the set-based ICE-learning algorithm for EQDAs, which consists of an implementation of the learner and the teacher. The learner is an adaptation of the RPNI algorithm from the libALF library [64]. Our implementation of the teacher works as follows. Given an EQDA conjectured by the learner, the teacher converts it to a quantified formula in the APF [61] or decidable Strand for lists [62], and uses a constraint solver to check adequacy of invariants. Since there was no implementation of the decision procedure for Strand, we evaluate our prototype on array programs only. More details about the implementation are in appendix D.

Program	White-Box	Black-Box		
	SAFARI (s)	R	|Q|	**ICE (s)**
copy	0.0	4	8	**0.7**
copy-lt-key	×	5	13	**1.2**
init	0.7	4	8	**0.6**
init-partial	×	8	12	**1.5**
compare	0.1	9	8	**1.3**
find	0.2	9	8	**1.2**
max	0.1	3	8	**0.4**
increment	×	5	8	**0.7**
sorted-find	×	8	17	**5.1**
sorted-insert	×	6	21	**2.0**
sorted-reverse	×	18	17	**9.4**
devres [48]	0.1	3	8	**0.7**
rm_pkey [48]	0.3	3	8	**0.7**

Table 2. Results for RPNI-based ICE-learning for quantified array invariants. R: # of rounds of iterative-ICE; $|Q|$: # of states in final EQDA. × means a timeout of 5 min.

Table 2 presents the results of our prototype on typical programs manipulating arrays. We compare our results to SAFARI [44], the state-of-the-art verification tool based on interpolation in array theories. SAFARI, in general, cannot handle list programs, and also array programs like *sorted-find* that have quantified pre-conditions. From the remaining, SAFARI diverges for some more programs and one probably needs to manually provide a term abstraction list for them to achieve convergence.

The results in Table 2 show that our ICE-learning algorithm for quantified invariants is effective, in addition to promising polynomial-per-round efficiency, promising invariants that fall in decidable theories, and promising strong convergence of the iterative learning.

6 Conclusions

The argument in this paper is a simple one: in order to build robust learning algorithms for invariant synthesis, we need the learner to be able to process implication samples, in addition to positive and negative samples. Traditional machine learning algorithms do not support implications and we must adapt them to the ICE-framework. Using new ICE learning algorithms for numerical domains as well as quantified invariants for linear data-structures, we have illustrated that it leads to much more robust and efficient algorithms for invariant synthesis.

References

1. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, ACM (1977) 238–252
2. Miné, A.: The octagon abstract domain. In: WCRE. (2001) 310–
3. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL, ACM (2002) 1–3
4. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: FME. Volume 2021 of LNCS., Springer (2001) 500–517
5. Thakur, A., Lal, A., Lim, J., Reps, T.: PostHat and all that: Attaining most-precise inductive invariants. Technical Report TR1790, University of Wisconsin, Madison, WI (Apr 2013)
6. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: FoVeOOS. (2010) 10–30
7. Floyd, R.: Assigning meaning to programs. In Schwartz, J.T., ed.: Mathematical Aspects of Computer Science. Number 19 in Proceedings of Symposia in Applied Mathematics, AMS (1967) 19–32
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10) (1969) 576–580
9. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO. (2005) 364–387
10. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, ACM (2002) 58–70
11. McMillan, K.L.: Interpolation and SAT-Based model checking. In: CAV. Volume 2725 of LNCS., Springer (2003) 1–13
12. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: TACAS. Volume 3920 of LNCS., Springer (2006) 459–473
13. Bradley, A.R.: SAT-based model checking without unrolling. In: VMCAI. Volume 6538 of LNCS., Springer (2011) 70–87
14. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: TACAS. Volume 2619 of LNCS., Springer (2003) 331–346
15. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: CAV. Volume 3576 of LNCS., Springer (2005) 548–562
16. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL, ACM (2005) 98–109
17. Sharma, R., Nori, A.V., Aiken, A.: Interpolants as classifiers. In: CAV. Volume 7358 of LNCS., Springer (2012) 71–87

18. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Verification as learning geometric concepts. In: SAS. Volume 7935 of LNCS., Springer (2013) 388–411
19. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: ESOP. Volume 7792 of LNCS., Springer (2013) 574–592
20. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. In: CAV. (2013) 813–829
21. Mitchell, T.M.: Machine learning. McGraw-Hill (1997)
22. Kearns, M.J., Vazirani, U.V.: An introduction to computational learning theory. MIT Press, Cambridge, MA, USA (1994)
23. Albarghouthi, A., McMillan, K.L.: Beautiful interpolants. In: CAV. (2013) 313–329
24. Reps, T.W., Sagiv, S., Yorsh, G.: Symbolic implementation of the best transformer. In: VMCAI. (2004) 252–266
25. Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: better together! In: ISSTA. (2006) 145–156
26. van Eijk, C.A.J.: Sequential equivalence checking without state space traversal. In: DATE. (1998) 618–623
27. Garoche, P.L., Kahsai, T., Tinelli, C.: Incremental invariant generation using logic-based automatic abstract transformers. In: NASA Formal Methods. (2013) 139–154
28. Colón, M., Sankaranarayanan, S., Sipma, H.: Linear invariant generation using non-linear constraint solving. In: CAV. Volume 2725 of LNCS., Springer (2003) 420–432
29. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI, ACM (2008) 281–292
30. Gupta, A., Majumdar, R., Rybalchenko, A.: From tests to proofs. In: TACAS. (2009) 262–276
31. Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: CAV. Volume 5643 of LNCS., Springer (2009) 634–640
32. Oncina, J., Garcia, P.: Inferring regular languages in polynomial update time. Pattern Recognition and Image Analysis (1992) 49–61
33. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL, ACM (1978) 84–96
34. Filé, G., Ranzato, F.: The powerset operator on abstract interpretations. Theor. Comput. Sci. **222**(1-2) (1999) 77–111
35. Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: SAS. Volume 4134 of LNCS., Springer (2006) 3–17
36. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Abstract interpretation over non-lattice abstract domains. In: SAS. (2013) 6–24
37. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. In: POPL, ACM (2011) 105–118
38. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, ACM (2008) 235–246
39. Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: On inter-procedural analysis of programs with lists and data. In: PLDI. (2011) 578–589
40. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL. (2002) 191–202
41. Lahiri, S.K., Bryant, R.E.: Predicate abstraction with indexed predicates. ACM Trans. Comput. Log. **9**(1) (2007)
42. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: CAV. Volume 4590 of LNCS., Springer (2007) 193–206
43. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: TACAS. Volume 4963 of LNCS., Springer (2008) 413–427

44. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Safari: Smt-based abstraction for arrays with interpolants. In: CAV. Volume 7358 of LNCS., Springer (2012)
45. Seghir, M.N., Podelski, A., Wies, T.: Abstraction refinement for quantified array assertions. In: SAS. Volume 5673 of LNCS., Springer (2009) 3–18
46. Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D.: Quickly detecting relevant program invariants. In: ICSE, ACM (2000) 449–458
47. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2) (1987) 87–106
48. Kong, S., Jung, Y., David, C., Wang, B.Y., Yi, K.: Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In: APLAS, Springer (2010)
49. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD. (2013) 1–17
50. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: ASPLOS. (2006) 404–415
51. Ackermann, C., Cleaveland, R., Huang, S., Ray, A., Shelton, C.P., Latronico, E.: Automatic requirement extraction from test cases. In: RV. (2010) 1–15
52. Choi, W., Necula, G.C., Sen, K.: Guided gui testing of android apps with minimal restart and approximate learning. In: OOPSLA. (2013) 623–640
53. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: Vcc: A practical system for verifying concurrent c. In: TPHOLs. (2009) 23–42
54. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. In: VMCAI. (2004) 135–148
55. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: CAV. (2011) 184–190
56. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: SIGSOFT FSE, ACM (2006) 117–127
57. Ivancic, F., Sankaranarayanan, S.: NECLA Benchmarks. `http://www.nec-labs.com/research/system/systems_SAV-website/small_static_bench-v1.1.tar.gz`
58. de Moura, L.M., Bjørner, N.: Z3: An efficient smt solver. In: TACAS. (2008) 337–340
59. Rakamaric, Z., Emmi, M.: SMACK: Static Modular Assertion Checker. `https://github.com/smackers/smack`
60. McMillan, K.L.: Lazy abstraction with interpolants. In: CAV. (2006) 123–136
61. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI. Volume 3855 of LNCS., Springer (2006) 427–442
62. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL, ACM (2011) 611–622
63. Angluin, D.: Negative results for equivalence queries. Machine Learning **5** (1990) 121–150
64. Bollig, B., Katoen, J.P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: libalf: The Automata Learning Framework. In: CAV. Volume 6174 of LNCS., Springer (2010) 360–364

A An ICE-learning algorithm for Numerical Invariants

The ICE-samples returned by the teacher and the corresponding invariant learnt by our learner for a program is shown in Figure 2.

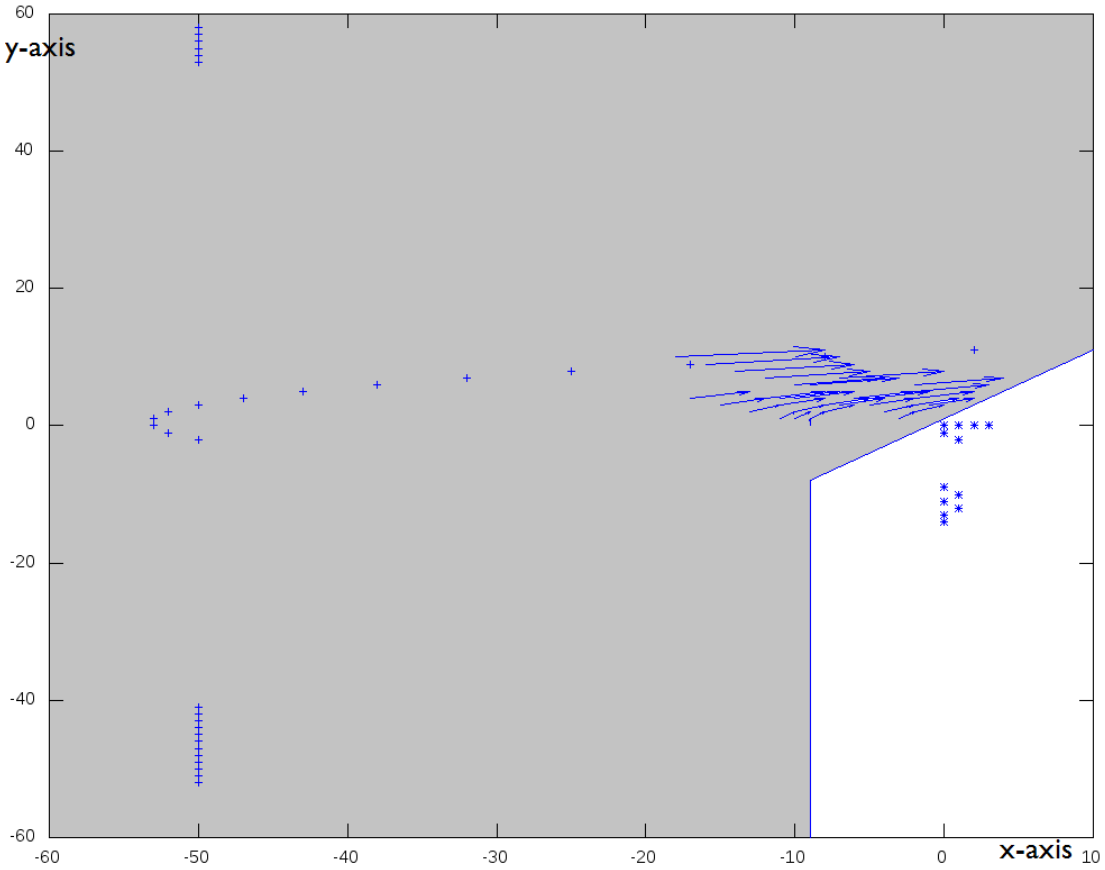


Fig. 2. The shaded region corresponds to the invariant $x \leq -9 \vee x \leq y - 1$. In the figure + are the positive examples, $*$ are negative counter-examples and arrows ($\rightarrow$) indicate an implication from the tail to the head of the arrow.

B Proof sketch for Theorem 2.

In [63] Angluin shows that there is no polynomial time learning algorithm for DFAs that only uses equivalence queries (no membership queries). The gist of this proof is as follows. One constructs a family $\mathcal{L}_n$ of languages, where each language in $\mathcal{L}_n$ is defined by a DFA of quadratic size in n, with the following property. For each learner that runs in polynomial time (meaning that it uses polynomially many rounds in the size of the target concept and each round only takes polynomial time in the size of the current sample), there is an n such that the teacher can answer the equivalence queries in such a way that after the polynomial number of rounds available to the learner, there is more than one language left in $\mathcal{L}_n$ that is still consistent with the answers of the teacher. This means that the learner cannot, in general, identify each target concept from $\mathcal{L}_n$.

This proof can be adapted to EQDAs (using just the formulas *true* and *false*) because the DFAs used to define the classes $\mathcal{L}_n$ are acyclic. However, for each class $\mathcal{L}_n$ a different alphabet is needed because for a fixed alphabet there are only finitely many EQDAs.

C RPNI for QDAs

In this section, we describe in detail how we adapt the RPNI algorithm to the setting of Section 5, i.e., to the setting of ICE-learning for EQDAs. To this end, we look at RPNI from a more abstract perspective and treat it as a *generic state merging* algorithm (GSM) for Moore machines. Generic in this context means that some methods are "templates", which we need to instantiate in order to obtain a concrete algorithm.

This section is structured as follows. We first introduce some definitions and notations that we will need throughout this section; in particular, this entails the definition of QDAs and EQDAs from [20]. Then, we describe the GSM algorithm. Finally, we explain how we instantiate the GSM algorithm to obtain the original RPNI algorithm and the adapted RPNI algorithm of Section 5.

C.1 Definitions and Notations

Let Σ be an alphabet and $L \subseteq \Sigma^*$ a set of words. The set

$$Pref(L) = \{u \in \Sigma^* \mid \exists v \in \Sigma^* \colon uv \in L\}$$

is the set of all prefixes of words in L.

QDAs and EQDAs The definition of QDAs is taken from [20]. We work with a set of program variables PV, a data domain D, and a set of universally quantified variables Y. We furthermore fix a formula lattice $\mathcal{F}$ that is used to express properties over the relative positions of the pointer variables and universally quantified variables, as well as properties of the data values from D at the positions of the variables.

With these parameters fixed, a *quantified data automaton (QDA)* is of the form $\mathcal{A} = (Q, q_0, \Pi, \delta, f)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta \colon Q \times \Pi \to Q$

is the transition function with $\Pi = 2^{PV} \times (Y \cup \{-\})$, and $f\colon Q \to \mathcal{F}$ is a final-evaluation function that maps each state to a data formula. The alphabet Π used in a QDA does not contain data. Words over Π are referred to as *symbolic words* because they do not contain concrete data values. The symbol $(b, -)$ indicating that a position does not contain any variable is denoted by $\underline{b}$.

A configuration of a QDA is a pair of the form (q, r) where $q \in Q$ and $r : Y \to D$ is a partial variable assignment. The initial configuration is (q_0, r_0) where the domain of r_0 is empty. For any configuration (q, r), any letter $a \in 2^{PV}$, data value $d \in D$, and variable $y \in Y$ we define $\delta'((q, r), (a, y, d)) = (q', r')$ provided $\delta(q, (a, y)) = q'$ and $r'(y') = r(y')$ for each $y' \neq y$ and $r'(y) = d$, and we let $\delta'((q, r), (a, -, d)) = (q', r)$ if $\delta(q, (a, -)) = q'$. We extend this function δ' to valuation words in the natural way.

A valuation word v is accepted by the QDA if $\delta'((q_0, r_0), v) = (q, r)$ where (q_0, r_0) is the initial configuration and $r \models f(q)$, i.e., the data stored in the registers in the final configuration satisfy the formula annotating the final state reached. We denote the set of valuation words accepted by $\mathcal{A}$ as $L_v(\mathcal{A})$. We assume that a QDA verifies whether its input satisfies the constraints on the number of occurrences of variables from PV and Y, and that all inputs violating these constraints either do not admit a run (because of missing transitions) or are mapped to a state with final formula *false*.

A data word w is accepted by the QDA if every valuation word v that has w as the corresponding data word is accepted by the QDA. The language $L(\mathcal{A})$ of the QDA $\mathcal{A}$ is the set of data words accepted by it.

Finally, a QDA $\mathcal{A}$ is called *elastic*—or an EQDA—if each transition on $\underline{b}$ is a self loop, that is, whenever $\delta(q, \underline{b}) = q'$ is defined, then $q = q'$.

Moore Machines Broadly speaking, a Moore machine is a deterministic finite automaton equipped with an output on its states. Formally, a *Moore machine* is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \delta, f)$ where Q is a finite, nonempty set of states, Σ is the input alphabet, Γ is the output alphabet, $q_0 \in Q$ is the initial state, $\delta\colon Q \times \Sigma \to Q$ is the transition function, and $f\colon Q \to \Gamma$ is the output function that assigns an output symbol from Γ to every state.

Runs are defined in the usual way, and we denote the unique state reached by $\mathcal{A}$ from some state $q \in Q$ after reading a word $u \in \Sigma^*$ as $\delta^*(q, u)$. A Moore machine defines a function $f_{\mathcal{A}}\colon \Sigma^* \to \Gamma$ where $f_{\mathcal{A}}(u) = f(\delta^*(q_0, u))$. Note that QDAs can be seen as a Moore machines that read symbolic words and output data formulas.

Later, we will also deal with partial Moore machines in which the transition function is not necessarily total. In this case, there might exist inputs that do not admit a run because of missing transitions. To make sense of such situations, we fix a dedicated symbol $\square \in \Gamma$ that the Moore machine outputs in such cases.

Quotient Moore Machine Let $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \delta, f)$ be a (partial) Moore machine and $\sim \subseteq Q \times Q$ an equivalence relation. We call $\sim$ a *congruence* (with respect to δ) if the following is satisfied for all $p, q \in Q$ and $a \in \Sigma$:

$$\text{if } p \sim q \text{ and } \delta(p, a), \delta(q, a) \text{ are defined, then } \delta(p, a) \sim \delta(q, a).$$

Moreover, the equivalence class of a state q is the set

$$[\![q]\!]_\sim = \{p \in Q \mid p \sim q\}.$$

Given a congruence $\sim$, we define the *quotient Moore machine* to be the the Moore machine $\mathcal{A}/_\sim = (Q', \Sigma, \Gamma, q_0', \delta', f')$ where

- $Q' = \{[\![q]\!]_\sim \mid q \in Q\}$,
- $q_0' = [\![q_0]\!]_\sim$, and
- $\delta'([\![p]\!]_\sim, a) = \begin{cases} [\![q]\!]_\sim & \text{if } \exists p' \in [\![p]\!]_\sim \colon \delta(p', a) = q \\ \text{undefined} & \text{else} \end{cases}$.

To define the output function f', we additionally need a means to "combine" the output of several states. To this end, we assume that a function $F \colon 2^\Gamma \to \Gamma$ is given that maps a set of output symbols to a single (combined) output symbol. Then, we have

$$f'([\![q]\!]_\sim) = F(\{f(p) \mid p \in [\![q]\!]_\sim\}).$$

C.2 The GSM algorithm

We are now ready to describe the GSM algorithm of which RPNI is a concrete instance. In the following description, the reader should interpret concepts printed in »*italic*« as "templates", which we will instantiate later.

Roughly speaking, the GSM algorithm proceeds in two phases:

1. It first constructs an »*initial Moore machine*« $\mathcal{A}_{\text{init}}$ that satisfies a »*property p*« from a »*sample*«. This sample might be any collection of words together with their output.
2. Then, it successively »*merges*« states of $\mathcal{A}$ in a »*particular order*«. For each candidate merge, it »*tests*« whether the merged Moore machine still satisfies p. If the merged Moore machine passes the test, the GSM algorithm proceeds with this merge. Otherwise, it discards the merge and proceed with the last successful merge.

A more formal description of this GSM algorithm is given as Algorithm 1 in Section 5 (the algorithm is repeated below). Internally, the algorithm calls three template functions, which have the following effect:

- The function `test`$(\mathcal{A})$ checks whether the Moore machine $\mathcal{A}$ satisfies a property p and returns either `true` or `false`.
- The function `init`$(\mathcal{S})$ constructs a Moore machine $\mathcal{A}$ that passes `test` $(\mathcal{A})$ from a sample $\mathcal{S}$.
- The function `order`(Q) returns an ordered list of all elements of Q with respect to some total order over Q.

Note that Algorithm 1 does not perform a state merge on the transition structure of the Moore machine $\mathcal{A}_{\text{init}}$ itself. For the sake of a simpler description, we rather represent a merge by means of a congruence $\sim$, which describes the merging of states on an abstract level. To perform the actual merge, we construct the quotient Moore machine $\mathcal{A}_{\text{init}}/_\sim$. Let us remind the reader that the computation of a quotient Moore machine requires a function $F \colon 2^\Gamma \to \Gamma$ to combine different outputs.

Since Algorithm 1 starts with a Moore machine that passes `test` and only merge states if the same is true for the merged Moore machine, we immediately obtain the following remark.

Input: A sample $\mathcal{S}$
Output: A Moore machine $\mathcal{A}$ that passes `test`($\mathcal{A}$)

$\mathcal{A}_{\text{init}} = (Q, \Sigma, \Gamma, q_0, \delta, f) \leftarrow$ `init`($\mathcal{S}$);
$(q_0, \ldots, q_n) \leftarrow$ `order`(Q);
$\sim_0 \leftarrow \{(q, q) \mid q \in Q\}$;
for $i = 1, \ldots, n$ **do**
  if $q_i \nsim_{i-1} q_j$ *for all* $j \in \{0, \ldots, i-1\}$ **then**
    $j \leftarrow 0$;
    repeat
      Let $\sim$ be the smallest congruence that contains $\sim_{i-1}$ and the pair (q_i, q_j);
      $j \leftarrow j + 1$;
    until `test`($\mathcal{A}_{init}/_{\sim}$);
    $\sim_i \leftarrow \sim$;
  else
    $\sim_i \leftarrow \sim_{i-1}$;
  end
end
return $\mathcal{A}_{\text{init}}/_{\sim_n}$;

Remark 1. Algorithm 1 always returns a Moore machine $\mathcal{A}$ that passes `test`($\mathcal{A}$).

Note that the repeat-loop always terminates because if $i = j$, then the constructed quotient automaton is the same as the one from the previous round, which already passed the test.

C.3 Instantiations of the GSM algorithm

To illustrate the GSM algorithm, let us now briefly demonstrate how to instantiate the templates in order to obtain the original RPNI algorithm. Then, we explain how to apply the GSM algorithm to the setting of ICE-learning of EQDAs.

RPNI In the original RPNI setting, the task is to learn a DFA $\mathcal{A}$ that is consistent with a pair (E, C) where $E, C \subseteq \Sigma^*$ are two disjoint, finite sets of words and $\mathcal{A}$ has to satisfy $E \subseteq L(\mathcal{A})$ and $C \cap L(\mathcal{A}) = \emptyset$. We instantiate Algorithm 1 as follows to obtain the original RPNI algorithm:

- We simulate DFAs by using Moore machines with the output alphabet $\Gamma = \{0, 1\}$; 1 corresponds to a final state and 0 corresponds to a nonfinal state.
- For $S \subseteq \Gamma$, we define the function F to be

$$F(S) = \begin{cases} 1 \text{ if } 1 \in S \\ 0 \text{ else} \end{cases}.$$

  That is, if a state with output 1 is merged with some other states, then the resulting merged state has also output 1.
- The sample is $\mathcal{S} = (E, C)$.

- `test`($\mathcal{A}$) checks whether all $u \in E$ have output 1 and all $u \in C$ have the output 0, i.e., whether all words from E are "accepted" and those of C are "rejected".
- `init`($\mathcal{S}$) constructs the *prefix tree acceptor* of E, i.e., the partial Moore machine $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \delta, f)$ where $Q = Pref(E)$, $q_0 = \varepsilon$,

$$\delta(u, a) = \begin{cases} ua & \text{if } ua \in Pref(E) \\ \text{undefined} & \text{else} \end{cases},$$

  and

$$f(u) = \begin{cases} 1 \text{ if } u \in E \\ 0 \text{ else} \end{cases}.$$

- `order`(Q) returns a set ordered with respect to the canonical order over words.

It is not hard to verify that the Moore machine produced by `init` passes `test`. Thus, Remark 1 yields that the Moore Machine returned by Algorithm 1 when interpreted as a DFA is consistent with (E, C).

ICE-learning of EQDAs We now can formally present our ICE-learning algorithm of Section 5.

The first thing to note is that we do not distinguish between (E)QDAs and Moore machines that read symbolic words (i.e., $\Sigma = \Pi$) and output a data formula (i.e., $\Gamma = \mathcal{F}$). Moreover,

- we choose the output combination function F to be the function that maps a set $S \subseteq \mathcal{F}$ of data formulas to the least upper bound of all formulas of S.

Note that the definition of F is sound because we assume that $\mathcal{F}$ forms a lattice.

Our GSM algorithm for EQDAs takes a sample $\mathcal{S} = (\hat{E}, \hat{C}, \hat{I})$ as input where $\hat{E}, \hat{C}$ are sets of sets of valuation words and $\hat{I}$ consists of pairs of sets of valuation words. In the end, we want our algorithm to produce an EQDA $\mathcal{A}$ that is consistent with the sample. In the context of EQDAs, consistency with a sample is defined as follows:

- For all sets of valuation words $S \in \hat{E}$ and each valuation word $w \in S$, the EQDA $\mathcal{A}$ has to accept w.
- For all sets of valuation words $S \in \hat{C}$ there exists a valuation word $w \in S$ such that the EQDA $\mathcal{A}$ rejects w.
- For all pairs of sets of valuation words $(S_1, S_2) \in \hat{I}$, if the EQDA accepts all $w_1 \in S_1$, then it must also accept all $w_2 \in S_2$.

Thus,

- `test`($\mathcal{A}$) checks whether the Moore machine $\mathcal{A}$ is consistent with the sample $\mathcal{S}$.

The process of creating an initial Moore machine (EQDA) from a sample (the function `init`) is more elaborate than in the case of RPNI and requires a fixed-point computation. First, let $E = \bigcup_{S \in \hat{E}} S$ be the set of all valuation words from positive examples and E' the set of all symbolic words that results from stripping the data from every valuation word of E. Then, we execute the following procedure, which is slightly different from the description in Section 5 but yields the same result.

1. We begin with constructing the prefix tree "acceptor" of E', i.e., the (partial) Moore machine $\mathcal{A} = (Q, \Pi, \mathcal{F}, q_0, \delta, f)$ where $Q = Pref(E')$, $q_0 = \varepsilon$,

$$\delta(u, a) = \begin{cases} ua & \text{if } ua \in E' \\ \text{undefined} & \text{else} \end{cases}.$$

   To define the output function f, we consider a symbolic word $w \in E'$ and define the set $E_w = \{v \in E \mid v \text{ without data is } w\}$ to contain all valuation words of E that result in the word w when stripped from their data. Then, $f(w)$ is the least formula in $\mathcal{F}$ that is satisfied for all valuation words in E_w. Due to the definition of f, $\mathcal{A}$ satisfies the first two conditions of consistency.
2. We elastify the resulting QDA $\mathcal{A}$ according to [20] and obtain the unique EQDA $\mathcal{A}'$.
3. If the resulting QDA $\mathcal{A}'$ is not consistent, then either there exists no such EQDA or the EQDA violates an implication. In the first case, we terminate the whole learning process. In the latter case, there exists an $(S_1, S_2) \in \hat{I}$ such that $\mathcal{A}'$ accepts all $w \in S_1$ but rejects at least one $w \in S_2$. We then add S_1 and S_2 to $\hat{E}$.
4. We repeat Steps 1 to 3 until we obtain an EQDA that is consistent with the sample (or we discover that no such EQDA exists).

Since the sample is finite, `init` eventually terminates and either reports that there exists no EQDA consistent with the sample or it produces a consistent EQDA. Thus, if `init` returns an EQDA $\mathcal{A}$, it passes `test(`$\mathcal{A}$`)`.

To finish the description,

- `order(`Q`)` returns an list of states (symbolic words) that is ordered according to the canonical order on words.

Finally, Remark 1 yields that the GSM instance from above produces an EQDA that is consistent with the given sample provided that such an EQDA exists.

D Details of the Prototype Implementation of the Iterative ICE-learner for EQDAs

This section provides a more detailed description of the exact functioning of our prototype including a detailed description of the teacher and learner.

Given a conjectured hypothesis, the role of the teacher is to check whether the conjectured invariant is adequate or not. In our case, the learner conjectures an EQDA as a hypothesis. The teacher first converts the EQDA to a quantified formula in the array property fragment (APF) [61] or the decidable Strand fragment over lists [62]. Then the teacher uses a constraint solver to check if the conjectured EQDA corresponds to an adequate invariant or not. If the answer is no, the teacher finds examples, counter-examples or implications over concrete data words that need to be added to the sample of the learner for the next iteration of iterative-ICE. However, because of the quantified setting, the sample is defined over sets of valuation words and not data words. Therefore, for every data word, the teacher obtains a set of valuation words and then adds these sets, or pair of sets in the case of implications, to the sample.

The learner is an RPNI-based ICE-learner which given a set-based sample $(\hat{E}, \hat{C}, \hat{I})$ conjectures an EQDA that is consistent with the sample. Let us first fix the formula lattice over data formulas to be the Cartesian lattice of atomic formulas over relations $\{=, <, \leq\}$. To check whether a valuation word v is rejected by an EQDA, the learner should just read v and check if its data values satisfy the data formula φ_v that the EQDA outputs on reading v. However, the learner actually implements this check in a slightly different manner. Given a valuation word v, the learner finds the smallest data-formula in the formula lattice which includes the data values in v, and rejects the word only if that formula is unsatisfiable in conjunction with φ_v. With this criterion of rejecting valuation words, words which should be actually rejected by the EQDA might not be rejected under this new criterion. In terms of the RPNI-based learner which merges states only if the EQDA still rejects all $C \in \hat{C}$, the new rejection criterion leads to fewer states being merged. The new criterion is therefore more conservative and it ensures that the EQDA learned by the learner still remains consistent with the sample. Apart from this modification, the learner is implemented exactly as described in the main paper.

To start the learning process, the teacher in the beginning runs the program on a few random input lists/arrays and collects the concrete data words that manifest at the program locations for which we want to synthesize an invariant. Each such data word is converted to a set of valuation words and together they form the set of positive examples $\hat{E}$ in the sample with which the iterative ICE-learning is initialized ($\hat{C}$ and $\hat{I}$ are empty to begin with).

We adapted the RPNI algorithm from the libALF library [64] to support the above described set-based ICE-learning algorithm. We use Z3 (which supports APF) as the constraint solver in the teacher for checking the adequacy of the quantified array invariants. We evaluated the learning algorithm on several array programs (see Table 2). Since we did not have an implementation of the decision procedure for the decidable fragment of Strand for lists, we could not evaluate our prototype on list-manipulating programs.