ICE: A Robust Learning Framework for Synthesizing Invariants

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

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

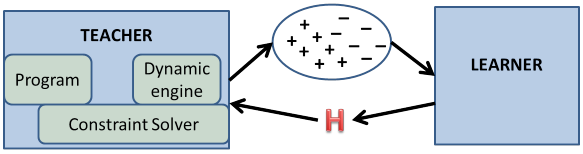
Abstract

Invariant generation lies at the heart of automated program verification, and the learning paradigm for synthesizing invariants is a new promising approach to solve this important problem. Unlike white-box techniques that try to generate an invariant by analyzing the program, learning approaches try to synthesize the invariant given concrete configurations that the invariant must include and exclude, and are algorithmically based on learning theory and scalable machine learning algorithms. In this paper we argue that traditional learning paradigms that use concrete examples and counter-examples are inherently non-robust for synthesizing invariants. We introduce a more general learning paradigm, called ICE-learning, that learns using examples, counter-examples, and *implications*, and show that this paradigm allows building honest teachers and convergent mechanisms for invariant synthesis. We study the new paradigm of ICE learning, develop several monotonic ICE-learning algorithms, and two classes of non-monotonic domains for learning numerical invariants for scalar variables as well as *quantified* invariants for arrays and dynamic lists, and establish convergence results for them. We implement these ICE algorithms in a prototype verifier and show that the robustness of ICE-learning is practical and effective by evaluating them on a class of programs.

1. Introduction

The problem of generating adequate invariants to show a program correct is at the heart of automated program verification. Automated program verifiers invariably synthesize invariants: abstract interpretation [16] finds invariants using fixed-points evaluated over an abstract domain, counter-example guided predicate abstraction iteratively computes predicates and uses model-checking to establish invariants [7], etc. In Floyd-Hoare style deductive verification settings, automated tools need to find invariants to replace the invariants traditionally provided by users [8, 14, 23, 33].

Invariant generation techniques can be broadly classified into two classes: 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 [16], counter-example guided abstraction refinement, predicate abstraction [7], the method of Craig interpolants [35, 42], IC3 [11], etc. all fall into the white-box category. In this paper, we are interested in the less well-studied black-box techniques for invariant generation.



One prominent class of black-box techniques for invariant generation is the emerging paradigm that we call *data-point driven learning*. Intuitively (see picture), we have two components in the verification tool: a white-box *teacher* and a black-box *learner*. 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, and (b) if the invariant is not adequate, to come up with an explanation using concrete *data-points* (program configurations) that tell the learner why the invariant is not adequate, and what further properties it needs to satisfy. The learner, who comes up with the invariant H is completely agnostic of the program and property being verified. Hence its aim is not really to build an invariant but simply to build some set 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 *data-points*, which are concrete 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 *data-point driven* learning approach for invariants has been explored for quite some time in various contexts, and is gaining considerable excitement and traction in recent years [24, 50–52]. The hope for success of data-point driven learning methods are, in our opinion, based on the following arguments.

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., 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 focuses on constraints expressed using a finite set of configurations of the program could have considerably better chances of finding the invariant. Note that the outer iteration anyway ensures that the learnt set is a correct invariant, absolving the learner from proving the set to be an invariant.

Second, the learning procedure provides a simple and natural way to *generalize* from partial knowledge. White-box program analysis using forward and backward symbolic execution using

bounded unfolding of loops, etc., gives a set of partial constraints the invariant needs to satisfy. The program-agnostic learner typically learns a *simple* concept that fits these constraints, and hence generalizes the partial knowledge; in white-box approaches, generalization is very hard simply by the fact that the synthesis engine knows so much that it needs to forget (interpolation, for example, is explicitly designed to forget information in order to generalize).

Finally, there is some hope that we can leverage extremely scalable techniques from the literature to solve the problem of the learner, especially using the large class of scalable machine-learning algorithms available today [45]. Data-point driven learning facilitates the application of these algorithms seamlessly into a verification platform, which the white-box model does not.

ICE-learning

Algorithmic learning theory and machine learning techniques, traditionally, offer learning from *positive examples*, and in some cases, *positive and negative examples*. Consequently, most learning algorithms in the literature for learning invariants have used such learners. The teacher hence gives concrete data-points of positive and negative examples, and the learner constructs a concept H that includes the positive examples and avoids the negative ones.

However, as we argue in this paper, learning using examples and counter-examples does not form a robust learning framework for synthesizing invariants. In this paper, we will define a new learning framework, called ICE-learning, that is a robust framework for synthesizing invariants.

To see why example and counter-example data-points 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 she finds a concrete data-point p that the invariant must contain, 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, she finds a concrete data-point p in H that should not belong to the invariant, and passes this as a *negative* example to the learner

(c) whether H is inductive; i.e., whether the strongest post of $H \wedge b$ across the loop body L implies H; if not, she finds two concrete data-points p and p', where $p \in H$ and $p' \notin H$.

In the last case above, the teacher is actually *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, 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 and bias that is very hard to control. If we are to take learning as a serious paradigm for synthesizing invariants, we need to fix this foundationally within 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 set the learner comes up with must satisfy the property that if p is included in H, then so should p'. The learner then 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, procedures, 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.

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 class of concepts 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, strong convergence of iterative learning, and the number of rounds of iteration required to learn are pertinent questions.
- We study first *monotonic* concept classes, and show that they are amenable easily to ICE learning. In particular, we show that several classes of invariants, including conjunctive Boolean formulas over predicates, k-CNF formulas, rectangular constraints, and more generally all abstract numerical data-domains can be easily adapted to ICE-learning. We show that the popular Houdini [22] algorithm for invariant synthesis *is* in fact an ICE-learning algorithm for conjunctions.
- We develop a non-monotonic ICE-learning algorithm for Boolean combinations of numerical constraints. We adapt existing white-box techniques that use a constraint solver to find invariants using templates [15, 28–30] to an ICE-learning algorithm. Furthermore, we prove that this algorithm is strongly convergent. We build a prototype verifier for synthesizing invariants over scalar variables and show that it is effective in proving several programs correct. The robustness of ICE-learning is hence practically feasible.
- We develop techniques for ICE-learning *quantified* properties. 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 *data-sets* rather than data-points. We instantiate this technique to build an ICE-learner for quantified properties of arrays and lists. This learner extends the classical RPNI learning algorithm for automata [47] to learning in the ICE-model and further learns *quantified data automata* [24], which can be converted to quantified logical formulas over arrays/lists. We build a prototype verifier building this learner and a teacher as well, and show that ICE-learning is effective for a class of benchmark programs.

Related Work

First, there are several white-box techniques that implicitly compute invariants. In abstract interpretation [16], invariant generation is achieved by computing fixed-points on abstract lattices of finite height, or using *widening* techniques for lattices of infinite height, combined with forward/backward analysis and narrowing

techniques (see [37] and [17] for abstract interpretation over affine equalities and inequalities and [44] for abstract interpretation over octagons). Abstract interpretation over powerset extensions of abstract domains [21, 48] can handle some form of disjunctions, in the synthesized invariants, as in our template based approach.

Abstraction-refinement approaches based on guidance by counter-examples strive to be property driven. Predicate abstraction [7, 32], for instance, tunes the abstract lattice according to the property being verified, and the Boolean program model-checker computes the reachable set of predicate-states, and hence essentially computes an invariant. Recent techniques based on Craig's interpolation [35, 42] and IC3 [11] have been proposed for generalizing results of symbolic bounded model-checking to find invariants.

Template based approaches to synthesizing numerical invariants using constraint solvers has been explored in the past in white-box settings [15, 28–30], and the technique is similar to the one we use in Section 5 for template-based ICE-learning.

Several white-box techniques for synthesizing quantified invariants are also known [1, 10, 18, 27, 31, 36, 43, 49]. Most of them are either based on abstract interpretation like [10, 18, 27, 31] or are based on interpolation theorems for array theories [1, 36, 43, 49].

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 [20] was path-breaking work that used essentially conjunctive Boolean learning to learn *likely* invariants from program configurations recorded along test runs. However, the invariants synthesized by Daikon might not be true invariants of the program. See [46] for recent work extending this work to finding likely polynomials and arrays invariants.

Learning was explicitly introduced in the context of verification by Cobleigh et al. [13], which was then followed by several algorithms based on Angluin's L^* algorithm [4] for learning regular languages applied to finding rely-guarantee contracts [3] as well as learning stateful interfaces for programs [2]. Houdini [22] uses essentially conjunctive Boolean learning (which can be achieved in polynomial time) to learn conjunctive invariants over templates of atomic formulas. In Section 4, we show that the Houdini algorithm along with its generalization by Thakur et al. [53] 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 [50–52] by using scalable machine learning techniques [38, 45] 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 invariants for linear data-structures and arrays are found using learning in [24], where the authors use a variant of Angluin's L^* algorithm [4]. Boolean formula learning has also been applied recently for learning quantified invariants [39] and transition invariants for termination analysis [40].

Turning to general learning theory, the field of algorithmic learning theory is a well-established field [38, 45]. The PAC learning model of learning approximating concepts with high probability using samples drawn randomly is a well-studied model. In the absence of probabilistic sampling, the classical results in this field are on learning using *queries* where the learner is allowed to ask various questions (see [5] for an overview). A celebrated result is the polynomial-time learning of regular languages by Angluin which uses membership and equivalence queries. Gold [25] showed that the problem of finding the smallest automaton consistent with a set of accepted and rejected strings is NP-complete. The RPNI algorithm [47] is a passive learning algorithm for regular languages that learns from positive and negative examples, works in polynomial time, but does not guarantee learning the smallest automaton. This algorithm is the one that we adapt to quantified data automata in order to learn quantified invariants for arrays/lists.

```
P1: i:=0;
    j:=0;
    while (i<=100) {
     if (i=25) then j:=1;
     i:=i+1;
    }
    assert (j=1);
```

```
P2: Input: A[0..n-1]
pre: 0 <= x < n & A[x]=1;
  i:=0; b:= -1;
  while (i < n) {
   if A[i]=1 then b:=i;
   A[i]:=0;
   i++;
}
post: (b!=-1 & forall y.
       (0<=y<n => A[y]=0))
```

2. Illustrative Examples

We motivate ICE-learning and the types of invariants that we consider in this paper using two simple examples. Consider the programs P1 and P2 (right above).

In program P1, the inductive invariant ($i>25 \Rightarrow j=1$) verifies the assertion. However, notice that a white-box engine that unrolls the loop a few times will only find positive examples for (i, j) in the kind $(0,0), (1,0), (2,0), ...$ for a small number of values of i, and counter-examples of the form $(100,0)$, $(100,2)$, $(100,3)$, $\ldots (99,0)$, $(99,2)$, $(99,3)\ldots$ (values close to 100 for i and values different from 1 for j). From these positive and negative examples, the learner could naturally come up with a conjecture such as $((i \leq 50 \Rightarrow j = 0) \wedge (i > 50 \Rightarrow j = 1))$. Machine learning algorithms that divide the positive examples and negative examples with planes that maximize the distance to both of them will tend to come up with such an invariant.

Now notice that the teacher is *stuck* as the positive examples (even with a few more unrollings) belong to the conjecture, and the teacher can't find a negative example that should be avoided by the conjecture, while the conjecture is not inductive. Consequently, when using a learner that uses only positive and negative samples, the teacher cannot make any progress. However, in ICE learning, the teacher can give an implication pair, say of the form $((25,0),(26,1))$, and proceed with the learning. Note that, despite the fact that in this example the learner was able to get directly at the interesting configuration where $i = 25$, quick convergence is still not guaranteed. However, the crucial point is that there is progress in learning. Also, a learner that tries to produce the simplest conjecture satisfying the samples would eventually generalize a large enough sample set to come up with this invariant.

Learning quantified invariants: Now consider Program P2 above: we have an input array, with the pre-condition asserting that there is some value of the array in the range $[0, n-1]$ that contains the value 1. This pre-condition can be written as $\exists x.(0 \leq x < n \wedge A[x] = 1)$, but instead, we have introduced a *ghost* variable x, which is unknown to the program, that witnesses the fact that $A[x] = 1$. The program finds some index with value 1 and simultaneously initializes the array to 0, and the post-condition asserts that b would not remain -1 across the loop and that the array is completely initialized to 0. An adequate invariant would hence be

$$(b = -1 \Rightarrow i \leq x \wedge A[x] = 1) \wedge 0 \leq i \leq n \wedge x < n \wedge \forall y.(0 \leq y < i \Rightarrow A[y] = 0).$$

Now, even assuming that we want to use an ICE-learning framework, we need the learner to learn a quantified invariant from examples, counter-examples, and implications that a teacher provides. This turns out to be much more complex than learning quantifier-free facts over numerical domains. Notice that the formula inside the universal quantifier for the invariant above is an *implica-*

tion of the form *guard* $\Rightarrow$ *data-formula*, and is inherently a non-conjunctive formula.

In this paper, we develop a general ICE-learning mechanism for handling quantified properties that sets up the problem of learning quantified invariants by mapping quantified variables as *new program variables* that extend the notion of a configuration. In this model, an example/counter-example is not a single configuration, but rather a *set* of valuation configurations. Intuitively, a single configuration c, when extended with valuations for a set of universally quantified variables, results in a *set* of valuation configurations V. We therefore refer to this model as set-based ICE-learning, which works on finding *quantifier-free* concepts that learn from such sets, and results in synthesizing universally quantified concepts; Section 6 introduces this model and provides an instantiation of this model for finding quantified invariants for arrays and lists.

3. ICE-Learning

When defining a (machine) learning problem, one usually specifies a domain (like points in the real plane or finite words over an alphabet), and a class of concepts (like rectangles in the plane or regular languages), which is a class of subsets of the domain. The teacher has such a target concept in mind and the task is to build an algorithm (the learner) that can infer (or approximate) this target concept from information that is provided by the teacher.

In classical learning frameworks (see [38]), the teacher provides a set of (positive) examples, which are elements of the domain that are part of the target concept, and a set of counter-examples (or negative examples), which are elements of the domain that are not part of the target concept. Based on these examples and counter-examples, the learner has to construct a hypothesis (which has to satisfy certain criteria w.r.t. approximations of the actual target concept the teacher has in mind).

In our setting, the teacher does *not* have a precise target concept from C in mind, but is looking for an inductive set with some additional constraints. Consequently, we extend this 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 a set of positive examples, $N \subseteq D$ is a set of counter-examples (or negative examples), and $R \subseteq D \times D$ is a relation interpreted as a set of implications. We call (P, N, R) the *target description*, and these sets typically have *infinite* descriptions, 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 a 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*.

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. In active learning, the learner can ask questions of certain type, which are then answered by the teacher. Since such a learning process proceeds in rounds, we refer to it as *iterative ICE-learning* (it-ICE). A famous active learning setting is the one of Angluin for learning regular languages [4]. In this setting, the learner can ask whether concrete elements of the domain belong to the target concept (membership query), and the learner can propose a hypothesis (equivalence query), which is then answered by the teacher with an element of the domain that is treated incorrectly by the hypothesis.

To define an active learner in our setting, note that the teacher does not have a precise target concept in mind and therefore cannot, in general, answer membership queries. We thus only allow equivalence queries, which we call correctness queries because there is no unique target concept to which the hypothesis could be equivalent. More formally, in one round of iterative ICE-learning, 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. 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 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.

3.1 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 each 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 obey the rules for her answers).

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. This is in general a stronger requirement than asking for convergence when the teacher knows a unique target concept $L \in C$, and hence strong convergence is a hard requirement for a learning engine to satisfy.

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

3.2 ICE-learning for synthesizing program invariants

We describe now how to use an iterative ICE-learner to learn invariants that prove a program correct with respect to assertions in the program.

Consider a program or system S and let us model the configurations of S as a set D. This will be the domain for the learning setup. Let us assume a set of initial configurations *Init* for S and that the semantics of S is given by a transition relation *post*, a binary relation on configurations. D can be arbitrarily complex, where each configuration in D models the state of the system: for example, the valuation of program variables, the stack of local variables capturing the call stack of a program with procedures, the global dynamic heap of the program, and, in the case of concurrency, the interleaved state of the program across the various concurrent components.

A safety specification φ for such a system is, in general, a subset of the configurations D, and the system is said to satisfy the specification if every reachable state (from the set of initial configurations) is contained in φ.

The set of reachable configurations, *Reach* is the smallest set that contains the initial configurations and is *post*-closed. An invari-

ant is any subset $Inv \subseteq D$ that contains *Reach* (i.e., $Reach \subseteq Inv$). An invariant *Inv* is said to be an *inductive invariant* if it is *post*-closed. An invariant *Inv verifies the safety specification* φ if *Inv* is contained in φ. Clearly, *Reach* is an inductive invariant, and if the system is correct, verifies the specification. However, there may be inductive invariants that are larger than *Reach* but *simpler* than *Reach* (in terms of expressibility using finite descriptions such as logic) that verify the specification.

We can solve the synthesis problem for invariants using the following setup. We first define a concept class that contains the class of subsets that we want invariants to express (this typically involves finding a syntactic form for expressing invariants). The teacher uses the target description $(P, N, post)$, where the positive set P is the set of initial configurations and the negative set N is the *complement* of the set φ. Furthermore, she knows the relation *post*. The teacher wants the learner to find an invariant *Inv* that includes P, excludes N, and is *post*-closed; note that P, N, and *post* are infinite sets in general. The learner is an (iterative) ICE-learner for learning sets of configurations. In the beginning, we can start with the teacher simply proposing the sample (E, C, I), where all these sets are empty. In each round, when the learner proposes a hypothesis H, the teacher checks if $P \subseteq H$ (if not, she produces an example configuration $c \in P \setminus H$ and adds it to E), checks if $H \cap N$ is empty (if not, she produces a counter-example configuration $c \in H \cap N$ and adds it to the set C), and finally checks if H is *post*-closed (if not, she finds a pair of configurations (c, c') such that $post(c, c')$ holds, c is in H, $c' \notin H$, and adds this pair to I).

Note the following salient features of the above algorithm. First, if the learner learns a concept H that satisfies the three properties of being an inductive invariant that proves the safety specification, then we are, of course, done with verifying S. Moreover, in the learning process, the learning has two properties:

Progress: In each round, the teacher proposes a new sample (E, C, I) that is consistent with all the previous sets it has proposed and refutes every previous concept H proposed by the learner. In other words, assuming a consistent learner, if C_i is the class of concepts that are consistent with the sample of round i, then C_i is a strict superset of C_{i+1}, for every i.

Honesty of teacher: The teacher never imposes any constraint on the set to be learned that contradicts any possible inductive invariant that proves φ. In other words, if *Inv* is any inductive invariant that proves φ, then *Inv* is consistent with all the samples in any round.

Note that progress and honesty do not imply convergence in finite time. However, the above two properties are very important for the robustness of a learning-based approach for synthesizing invariants. For example, in many learning-based algorithms for learning invariants using examples and counter-examples only, when the teacher gets a hypothesis H that is not inductive, she can find a pair (c, c') as above that witness non-inductiveness, and then *add both* to the set of examples. Such a teacher would not be honest, as there could be an inductive invariant proving φ that excludes c, which the teacher precludes by adding both of them to the positive examples. Similarly, adding both to the negative set is also dishonest. Dishonesty of the teacher could lead to an evolution of the learning process where eventually there are no inductive invariants left that prove φ and that are consistent with sample; furthermore, we cannot detect when this happens, and hence cannot backtrack effectively to undo previous decisions.

The above ICE-learning scheme requires building two components. First, we need an ICE-learning algorithm for a set of concepts that is expressive enough to represent invariant sets of configurations. Second, to build the teacher, we need effective procedures that check the properties of whether a hypothesis satisfies the three conditions for being an inductive invariant verifying φ, and also construct concrete examples, counter-examples, and implication-pairs when these conditions are not satisfied.

Learning Floyd-Hoare style proof annotations:
While the above is a general scheme of learning invariants, we will be more interested in this paper in restricted forms of expressing inductive invariants that use Floyd-Hoare style annotations for sequential imperative programs [23, 33]. In Floyd-Hoare style verification for sequential while-programs with function calls, the annotation mechanism requires pre- and post-conditions for all functions/methods in the program, and loop invariants for every while-loop in the program. (Object-oriented programs could have more sophisticated annotation schemes such as class invariants, etc., which we will ignore for now, but these can be easily incorporated.) Note that these annotations define in turn a particular form of an invariant over the set of global configurations of the program, but are restricted in the sense that they talk only about the local configuration in scope. Floyd-Hoare style reasoning hence has several sources of incompleteness (the ability to talk only about the local state in scope, though this can be mitigated by using ghost-variables, the logic used to express the properties of the state, etc.) Post-conditions can express properties not just about the post-state, but relate it to the pre-state as well, and hence we will think about post-conditions encoding properties of pairs of configurations.

Assuming specifications are given in the form of assertions, ICE-learning can now be used to find the annotations of the program that prove the program correct. We first set up a way to represent sets of configurations and set up an iterative ICE-framework to learn such sets of configurations. We can add a new variable l to configurations that denotes the annotation location, and learn a set of configurations which can then be partitioned using this variable into the various places of annotation in the program. When confronted with a hypothesis H, the teacher will check whether the annotations defined by H prove the program correct (by generating verification conditions that are then validated using an automated theorem prover), and if not, find witnesses (again, using the theorem prover) that show why H does not define an inductive invariant proving the assertions, and appropriately modify the sample (E, C, I) in the next round.

Note that most of the verification conditions, when invalid, will result in an *implication* constraint to be added to the constraint set. However, the entire process of deriving an invariant is property-based (dependent on φ) as the counter-examples stem for the inability of the invariant to prove the safety property at hand. Also, note that all normal program constructs can be handled using this general scheme (multiple loops, nested loops, function calls, recursion, etc.); however *synthesizing new ghost variables*, updating them, and using them in invariants cannot be done in this framework, and in general, is out of scope of our current work.

4. Monotonic ICE Frameworks

We now study a class of ICE-learning frameworks, which we call *monotonic* frameworks, that are extremely simple to build ICE-algorithms for. The aim of this section is to also show that there are *existing* approaches in the literature for program verification that can be seen as or easily modified to be ICE-learning algorithms.

In a monotonic framework, we assume that given a set of positive samples E, there is a *smallest* concept (set) that contains E. Furthermore, let us assume that we can build a (passive) learner who can learn this smallest concept containing E. We now show that such a learner can be turned into an ICE-learner.

The ICE learner can be built as follows. Given finite sets (E, C, I), the learner does the following:

(1) Computes the closure E' of E with respect to the implications in I. In other words, we compute the smallest set E' such that $E \subseteq E'$ and for every $e \in E'$, if there is some e' such that $(e, e') \in I$, then e' also belongs to E'. Since I is finite, it's clear that E' is also finite and can be computed.

(2) The ICE-learner then uses its *passive learning algorithm* to find the smallest hypothesis H that contains E'. If H does not satisfy the implication constraints I, then we find the pairs (e, e') in I where $e \in H$ and $e' \notin H$, and add the element e' to E', and repeat the passive learning algorithm for the new set. If H does not satisfy the counter-example constraints C, then the learner exits stating that there is *no concept* that meets the constraints of the sample.

To see why the above is a consistent ICE-learner that terminates, we will show that the learner in each round, when it produces a hypothesis H, produces the least concept that meets the constraints imposed by the samples. In Step 1 above, the set E' is clearly the set of positive examples that is implied by the constraints. When the ICE-learner uses the passive learner in Step 2 for the first time and finds a hypothesis H, since H is the smallest concept containing E', it follows that for any implication pair (e, e') that is not satisfied by H, *any* concept that satisfies the samples must include e' as well. This is so because every concept that satisfies the constraints is a superset of H, and hence must include e. This justifies adding e' to the positive set and repeating the passive learning algorithm. The same reasoning shows that if there are any counter-examples $ce \in C$ that belong to H, then *no concept* that satisfies the constraints can exist, since all of them are supersets of H and will include ce. In this case, we can terminate and declare that there is no concept that satisfies the samples. To argue termination of ICE-learning, note that the iteration in Step 2 terminates because the passive learner, since it's given new examples that aren't in the current hypothesis, is forced to produce larger and larger sets. Since I is finite, this process will converge in a finite number of steps.

It is also easy to see that the above ICE-learner is *consistent*; it produces hypotheses that satisfy all constraints. Consequently, it follows by Lemma 3.1 that if the class of concepts is finite, then the *iterative ICE-learning* using the above algorithm strongly converges.

We will now quickly look at a variety of domains that admit monotonic ICE-learning.

ICE-learning conjunctions of predicates: Assume a finite set of predicates *Pred* over configurations, and let C be the class of concepts that consist of conjunctions of literals over these predicates, (i.e., monomials over *Pred*). We can now build a monotonic ICE-learning algorithm for this class. In order to do this, notice that given a finite set E of sample configurations, we can build a monomial that consists precisely of the conjunction of literals l such that *every* element of E satisfies l. Clearly, this defines the monomial that defines the smallest set of configurations that includes the set E. We can even do this efficiently (in polynomial time): we start with the set of all literals, and for every example $e \in E$, examine each literal to see whether e satisfies the literal, and remove it from the set if it doesn't. This learning algorithm hence can be extended to an ICE-learning algorithm using the scheme described above.

The above passive learning from examples is not new, and is in fact a well-known learning algorithm for conjunctions, and also works in the *PAC*-learning setting [38]. The derived ICE-learning algorithm, in essence, is close to the well-known technique called *Houdini* in program verification [22]. The Houdini algorithm typically works using symbolic strongest-posts as opposed to concrete data-points as in the learning algorithm above, but otherwise is essentially the same algorithm.

Note that the iterative ICE-learning algorithm is strongly convergent because the number of concepts is finite, and uses only polynomially many rounds to converge.

ICE-learning k-CNF: The standard PAC learning algorithm for k-CNF formulas, for a fixed k also extends to ICE-learning [38]. This algorithm essentially enumerates all possible k-CNF clauses (there are only polynomially many) and then using these as propositions, learns a conjunction of them using the conjunctive-learning algorithm above. The same algorithm works in the ICE-setting as well, works in polynomial time, and strongly converges in polynomially many rounds.

ICE-learning rectangles: Assume that the set of concepts involves learning rectangles in a 2D plane, modeling constraints on two rational/integer variables in a program (this also extends to cubes in higher dimensions). Rectangles are closed regions with four borders parallel to the x and y axes. It turns out that the well-known PAC-learning algorithm for rectangles [38] is in fact an algorithm that learns the *smallest* bounding rectangle for the positive examples. This algorithm is hence a passive learning algorithm from examples, and immediately extends using our technique to an (iterative) ICE-learning algorithm.

In this case, the number of concepts is *not* finite, and in fact, the above algorithm is not strongly convergent over the rational domain. However, for integer domains, we can prove that the above algorithm is strongly convergent. Note that a rectangle contains a finite number of points; let's call this its size. Notice that in each round, when the teacher proposes a new set of samples that contradicts the last conjecture, the learner will necessarily call the passive learner on a larger set of examples E' in Step 2 to come up with the final rectangle. Now, if there is some inductive invariant *Inv* expressible as a rectangle, it is clear that the passive learner can never be using any positive set of samples that is larger than the size of *Inv*, and will hence have to converge. As far as we know, this is the first time such a convergent learning algorithm has been described for even such a simple domain as this.

ICE-learning over abstract lattices: We can in fact extend monotonic ICE learning to *any* abstract interpretation domain D that over-approximates sets of concrete configurations and has *least upper bounds*. Intuitively, given a set of examples E, we can learn the set $\sqcup_{e \in E} \alpha(\{e\})$, where α is the abstraction operator. This is clearly a passive learner that finds the smallest abstract element that contains E. Using our general extension above, we can turn this into an ICE-learning algorithm, and when these operations (α and $\sqcup$) can be done in poly-time, we get a poly-time learner. Moreover, the iterative ICE-learner converges whenever the lattice has finite height. It immediately follows that for a large class of numerical domains such as intervals, octagons, convex polyhedra, linear equalities, etc., we immediately obtain ICE-learning algorithms.

A similar idea of using samples to learn an element of the abstract domain has also been proposed recently by Thakur et al. [53], but in the context of building precise abstract transformers.

Note that, in some sense, the monotonic frameworks above achieve ICE-learning simply from a passive learner for examples, exploiting monotonicity of the class of concepts. In the rest of the paper, we will study more complex concept classes, such as those that use *disjunctions*, which are not monotonic, and yet admit ICE-learning.

5. Template-based ICE-learning for Numerical Constraints

In this section, we describe a learning algorithm for synthesizing invariants that are arbitrary Boolean combinations of numerical constraints over program variables. Since disjunctions are allowed, the

concept class is not monotonic. 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 find a formula with the *simplest* Boolean structure. In order to build an effective algorithm, we will iterate over templates of the Boolean structure, and for each template, use a constraint solver to efficiently search for a formula of that template. The atomic formulas are also restricted in syntactic form. The core of the algorithm is hence one that, for a fixed template, performs iterative ICE-learning to learn an adequate invariant expressible in the form of the template, provided one exists. If no adequate invariant exists in the given template, we progressively increase the complexity of the template till an invariant is found. Even though the constants used in the atomic formulas can be found for such a general template, in order to achieve strong convergence, we will further restrict the templates to use constants within a particular bound, which will be iteratively increased when no invariant is found.

Let $Var = \{x_1, \cdots, x_n\}$ be the set of (integer) variables in the scope of the program. Then a program configuration is an n-tuple with a concrete value for every variable $x_i \in Var$. We want to design a learning algorithm that learns Boolean combinations of octagonal and box constraints over such program configurations, i.e., it learns invariants which are Boolean combinations of atomic formulas of the form

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

A template, as mentioned before, fixes the Boolean structure of the desired invariants by specifying the number of disjunctions and conjunctions in the DNF representation of the invariants. It also restricts the constants $c \in \mathbb{Z}$, which appear in the numerical constraints, to lie within a finite range $[-Max, +Max]$, for $Max \in \mathbb{Z}^+$. For a given template which fixes the Boolean structure of the invariants to be of the form $\bigvee_i \bigwedge_j a^{ij}$, we describe an iterative ICE-learning algorithm which learns an octagonal or box constraint (of the form specified above) for each a^{ij} such that the learned invariant

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

is adequate and is inductive.

The iterative ICE-learning algorithm consists of an ICE-learning algorithm paired with a white-box teacher who analyzes the program to give examples, counter-examples, and implications, as explained in Section 3.2. The task of the ICE-learner is to construct, given a sample (E, C, I), a hypothesis formula, which essentially means finding concrete values for s_k^{ij}, v_k^{ij} ($k \in \{1, 2\}$) and c^{ij} such that the learned formula $\varphi(x_1, \cdots, x_n)$ is consistent with the sample (we detail on this in a moment).

Unfortunately, learning in the presence of implications is hard. The uncertainty of 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-learning algorithm uses a constraint solver which, due to recent advances in SAT/SMT, can search this exponential space in a reasonably efficient manner.

Given a sample (E, C, I), the learner uses a constraint solver to find values for s_k^{ij}, v_k^{ij} and c^{ij} such that the hypothesis φ is consistent with the sample, i.e., for every data-point $p = (p(1), \cdots, p(n)) \in E$, $\varphi(p)$ holds; for every data-point $p \in C$, $\varphi(p)$ does not hold; and for every implication pair $(p, p') \in I$, $\varphi(p')$ holds if $\varphi(p)$ holds. The learner does so by constructing an SMT formula over the free variables s_k^{ij}, v_k^{ij} and c^{ij} which precisely captures these ICE-constraints on the data-points p in the sample (E, C, I). The precise SMT formula Ψ that the learner constructs is provided in Figure 1. In this formula, b_p is a Boolean variable which tracks $\varphi(p)$ and Ψ asserts that b_p is true for $p \in E$, $\neg b_p$ is true for $p \in C$ and $b_p \Rightarrow b_{p'}$ is true for $(p, p') \in I$. 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 p (line 2 of the formula), the variables r_{kp}^{ij} encode the value of $s_k^{ij} v_k^{ij}$ (line 5 of the formula), and d_{kp}^{ij} the value of v_k^{ij} (line 6 of the formula).

$$\begin{aligned}
\Psi(s_k^{ij}, v_k^{ij}, c^{ij}) \equiv \;\; & \left(\bigwedge_{p \in E} b_p \right) \wedge \left(\bigwedge_{p \in C} \neg b_p \right) \wedge \left(\bigwedge_{(p,p') \in I} b_p \Rightarrow b_{p'} \right) \\
& \bigwedge_p \left(b_p \Leftrightarrow \bigvee_i \bigwedge_j b_p^{ij} \right) \\
& \bigwedge_{p,i,j} \left(b_p^{ij} \Leftrightarrow \Big(\sum_{k \in \{1,2\}} r_{kp}^{ij} \leq c^{ij} \Big) \right) \\
& \bigwedge_{i,j} -Max \leq c^{ij} \leq Max \\
& \bigwedge_{\substack{p,i,j \\ k \in \{1,2\}}} \begin{pmatrix} s_k^{ij} = 0 \Rightarrow r_{kp}^{ij} = 0 \\ s_k^{ij} = 1 \Rightarrow r_{kp}^{ij} = d_{kp}^{ij} \\ s_k^{ij} = -1 \Rightarrow r_{kp}^{ij} = -d_{kp}^{ij} \end{pmatrix} \\
& \bigwedge_{\substack{p,i,j \\ k \in \{1,2\}}} \bigwedge_{l \in [1,n]} (v_k^{ij} = l \Rightarrow d_{kp}^{ij} = p(l))
\end{aligned}$$

Figure 1. The SMT formula Ψ constructed by the learner. Ψ additionally asserts that $s_k^{ij} \in \{0, +1, -1\}$, $v_k^{ij} \in [1, n]$ and $v_k^{ij} \neq v_{k+1}^{ij}$ for all i, j.

Note that the formula Ψ falls in the theory of quantifier-free linear integer arithmetic, the satisfiability of which is decidable. A satisfying assignment for Ψ gives a formula that the learner conjectures as an invariant. Our learner always produces a hypothesis which is *consistent* with its sample. If Ψ is unsatisfiable, it implies that there is no invariant in the current template which is consistent with the given sample. In this case, assuming that the program is safe, we increase the size of the template by dovetailing between increasing the size of the Boolean structure of the template and increasing the range of the constant, Max, specified in the template. A similar approach can be used for learning linear constraints, and even more general constraints if there is a solver that can effectively solve the resulting theory.

Our learning algorithm is quite different from the earlier constraint based approaches to invariant synthesis [15, 28–30]. These approaches directly encode the adequacy of the invariant (in a white-box manner, encoding the entire program's body) into the constraint, and use Farkas' lemma to reduce the problem to satisfiability of formulas in the quantifier-free theory of non-linear arithmetic, which is harder and in general undecidable. On the other hand, we split the task of invariant synthesis between a white-box teacher and a black-box learner, both of which communicate only through ICE-constraints on concrete data-points. This greatly reduces the complexity of the problem, leading to simpler teachers and a simpler learner.

Convergence

A template fixes the Boolean structure of the invariants and also restricts the constants appearing in the numerical constraints in the invariants to a finite range. Hence, for a given template, the concept class consisting of invariants belonging to that template is finite. Additionally, our ICE-learner always conjectures a hypothesis which is *consistent* with its sample. Hence, the learner never conjectures the same invariant twice. Therefore, in a finite number of rounds, our iterative ICE-learning algorithm either learns an adequate invariant or discovers that no such invariant exists in the given template and maximum value of constants. In the latter case,

we dovetail between increasing the size of the Boolean structure of the template or increasing the range of the constants specified in the template. Since, for a given template the iterative ICE-learning algorithm runs for a finite number of rounds, if there exists an adequate invariant in some template our learning algorithm will eventually discover it. This leads to the following result.

THEOREM 5.1. *The above template-based ICE-learning algorithm for numerical constraints always produces consistent conjectures and strongly converges.*

Note that the constraint solver can handle (in a decidable way) constraints that encode unbounded constants. However, such a learner for increasing Boolean templates may not converge, which is why we resort to bounding the constants. In practice, searching for invariants with unbounded constants as well would be useful in cases where the invariant does require large constants.

Experimental Results

We implemented a prototype [1] of the template-based iterative ICE-learning algorithm for learning numerical invariants. To start with, the teacher in the beginning runs the program on a few random inputs and collects the program configurations that manifest at the program point for which we want to synthesize an inductive invariant. These program configurations form the set of positive *examples* with which the iterative ICE-learning algorithm is initialized. The set of *counter-examples* and *implication* pairs is empty in the first round of iterative ICE-learning.

At any given stage, given a sample (E, C, I), the learner uses an SMT solver, as described in this section, to learn a hypothesis which is consistent with the sample. The teacher then generates verification conditions for every basic path in the program and uses an SMT solver to check the adequacy of the conjectured hypothesis. If the hypothesis is not adequate, the teacher adds more information in the form of examples, counter-examples or implications to the sample of the learner and this process iterates till an adequate invariant is found. To speed up this process, the teacher usually adds multiple examples, counter-examples or implication pairs to the sample, in every round of iterative ICE.

We evaluated our ICE-learning algorithm on various programs from the literature (see Table 1). We report the number of variables in the scope of the program and the template arguments: the number of disjunctions, conjunctions and the maximum range *Max* of the constants appearing in the invariants. Next, we report the number of rounds of iterative ICE-learning required; the number of examples, counter-examples and implication pairs in the final sample of the learner; the total time taken by the learner to conjecture invariants in all the rounds combined and the total time taken by the teacher to refute inadequate conjectures and add ICE-constraints to the sample of the learner. We also report the final inductive invariants learned.

The programs `w1` and `w2` from [28] are simple iterative programs. However, standard narrowing techniques [16] are unable to regain the precision lost to widening during static analysis of these programs. On the other hand, our learning algorithm is property driven and can refine the invariants by adding more counter-examples. `tacas06` [35] is a program where naïve predicate abstraction [7] fails due to divergence. However, limiting the search to invariants from a restricted template prevents divergence in our approach. The programs `ex7` and `ex23` are array programs. Proving even simple properties on integer variables for such complex programs using white-box techniques like interpolation [42] is hard and very heavy-weight. Our algorithm also has advantages over earlier white-box, constraint based approaches to invariant synthesis [15, 28–30]. Being black-box, we reduce the problem to satisfiability of simpler constraints. Invgen [29], for example, is not able to synthesize an adequate invariant for the program `fig1` [28].

The experiments show that our learning algorithm can learn an adequate invariant in reasonable time, requiring only a few number of rounds and a small sample size. More pertinently, though we use the more complex but more robust framework of ICE-learning, the time taken to learn is comparable to other learning algorithms like [50, 52] that learn invariants from just positive and negative examples (but lack robustness).

[1] available as supplementary material

6. Universally Quantified Properties

In this section we describe a setting of ICE-learning for *universally quantified* concepts, where the universal quantifiers. 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, or a tree has the max-heap property if the data at any nodes is larger than the data at its children. 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.

The formula ψ describes a set of program configurations with a certain property. In the plain learning setting, the class of all sets that can be described by such universal formulas would form the concept class for the learning problem (over the domain D of all program configurations). We now describe how this setting can be modified in a fairly generic way such that we can use a learner for the quantifier-free property (described by the formula $\varphi(y_1, \ldots, y_k)$). For this purpose, we add valuations of the quantified variables to the elements of the domain (the program configurations), and modify the learning setting a bit to reflect the universal semantics of the variables $y_1, \ldots, y_k$.

More precisely, each concrete program configuration c corresponds to a 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. Now consider a configuration c. 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 $\hat{I} \subseteq 2^D \times 2^D$.

The connection between ICE-learning for universally quantified concepts over some domain D and the data-set based setting is as follows. 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. Then each element c of the domain can be replaced by the set $S_c \subseteq D_{val}$. This transforms (P, N, R) into a set-based target description. Then a hypothesis H (described by some quantifier-free formula $\varphi(y_1, \ldots, y_k)$) is correct w.r.t. the set-based target description if, and only if, 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.

In other words, 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.

Table 1. Results for template-based iterative ICE-learning for numerical constraints.

Program	#Vars	#Disj.	#Conj.	Max	#Rounds	#E	#C	#I	Time (s) Learner	Time (s) Teacher	Total Time (s)	Learned Invariant
cegar1 [28]	2	1	1	10	3	9	1	6	0.1	0.5	0.6	$x \geq 0 \wedge x - y \leq 3$
cegar2 [28]	3	1	3	10	6	17	12	24	2.2	1.2	3.4	$x \leq N \wedge 0 \leq m \leq x - 1$
fig1 [28]	2	2	1	10	7	26	30	18	0.8	1.6	2.4	$x \leq -9 \vee x \leq y - 1$
w1 [28]	2	1	1	10	3	15	12	0	0.0	0.5	0.5	$x \leq n$
w2 [28]	2	1	1	10	4	27	18	0	0.1	0.6	0.7	$x \leq n - 1$
tacas06 [35]	4	3	2	10	5	26	18	18	6.7	1.4	8.1	$i \leq j - 1 \vee i \geq j + 1 \vee x = y$
fig3 [26]	3	2	2	10	2	17	12	6	0.3	0.4	0.7	$lock = 1 \vee x = y - 1$
fig6 [26]	1	1	1	10	1	5	0	0	0.0	0.0	0.0	$true$
fig8 [26]	6	1	2	10	2	7	6	0	0.0	0.3	0.3	$lock = 0$
fig9 [26]	2	1	2	10	4	4	12	12	0.3	0.6	0.9	$x \geq 0 \wedge y \geq 0$
ex23 [34]	3	1	4	5000	8	61	24	36	9.0	2.0	11.0	$0 \leq y \leq z \wedge c \leq z \leq c + 4572$
ex11 [34]	2	1	2	10	3	7	12	4	0.1	0.6	0.7	$0 \leq len \leq 4$
ex7 [34]	4	1	2	10	7	29	36	0	1.1	1.4	2.5	$0 \leq i \wedge y \leq len$
ex14 [34]	2	1	1	10	2	17	6	0	0.0	0.3	0.3	$x \geq 1$

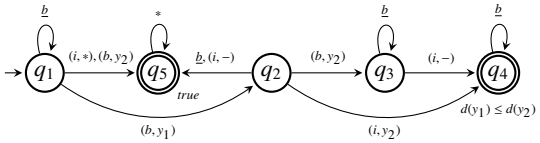


Figure 2. An example QDA representing an invariant of a sorting routine.

6.1 Linear data structures and quantified data automata

We now illustrate the data-set based learning for a specific class of universally quantified properties that can be expressed by an automaton model called quantified data automata (QDA). This model was introduced by Garg et al in [24]. We briefly recall the main ideas concerning this model and refer the reader to [24] for more detailed definitions (additional information in the appendix in the supplementary material).

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.

To describe such kinds of invariants by automata, arrays (or other linear data structures) are modeled by *data words*. Each position in the word corresponds to an element or cell in the data structure. The words are labeled with sets of pointer variables of the program, indicating the value of the pointer. Furthermore, each cell contains a data value from some data domain (e.g., the integers). A QDA defines a set of data words. However, to capture the idea of expressing universally quantified properties, the words that are actually read by a QDA are also annotated with variables $y_1, \ldots, y_k$.

The above invariant is captured by the QDA in Figure 2. The letters at the transitions are pairs, in which the first component indicates the pointer variables pointing to this position of the word, and the second component contains the universally quantified variable at this position (if any). The symbol b in the first component represents blank positions that are not pointed to by any pointer, and the $-$ in the second component means that no variable y_i is evaluated to this position. The symbol $\underline{b}$ is an abbreviation for $(b, -)$. A $*$ in the picture represents an arbitrary value.

The QDA from Figure 2 accepts a data word w with a valuation v for the universally quantified variables y_1 and y_2 as follows: it stores the value of the data at y_1 and y_2 in two registers, and then checks whether the formula annotating the final state it reaches holds for these data values. Moreover, the automaton accepts the data word w if for *all* possible valuations of y_1 and y_2, the automaton accepts the corresponding word with valuation. The QDA hence accepts precisely those set of data words that satisfy the invariant formula.

We refer to data words extended with valuations as *valuation words*. Note that the data at the individual positions is not explicitly read by the automaton, it only plays a role for the evaluation of the formula at the final state.

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 [24] the subclass of elastic QDAs (EQDAs) is studied. EQDAs have a decidable emptiness problem and can be translated into decidable logics, like the array property fragment [12] for arrays, or a decidable fragment of the logic Strand [41] 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 as follows. A QDA $\mathcal{A}$ is called *elastic* if each transition on $\underline{b}$ is a self loop, that is, whenever $\delta(q, \underline{b}) = q'$ is defined, then $q = q'$. The example QDA in Figure 2 is *not* an elastic QDA because there is a $\underline{b}$-transition from q_2 to q_5. However, an equivalent EQDA, which allows y_2 to occur arbitrarily after y_1, can express the invariant. 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. Since on blank symbols only loops are allowed in EQDAs, there are only finitely many EQDAs for a fixed alphabet (set of variables). We refer the reader to [24] for more details on EQDAs.

A lower bound for ICE-learning of EQDAs.

The goal of this section is to develop an iterative ICE-learner for concepts represented by EQDAs. We start by a result showing that we cannot hope for a polynomial time iterative ICE-learner, when the set of pointers and quantified variables is unbounded.

THEOREM 6.1. *There is no polynomial time iterative ICE-learner for EQDAs, when the alphabet size is unbounded.*

The theorem can be proven by adapting a lower-bound result by Angluin, from [6], namely that there is no polynomial time learning algorithm for DFAs that only uses equivalence queries (no membership queries). The gist of the 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.

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. Recall that membership queries cannot be answered honestly by an ICE teacher and thus we only have equivalence (or correctness) queries. Note that in [24] a learning algorithm for (E)QDAs based on Angluin's L^* is developed. However, this algorithm uses a *dishonest teacher* that answers queries in an arbitrarily chosen way if it does not know the answer. The goal of this paper is to develop a robust setting in which the teacher can answer all queries honestly.

Adapting the heuristic RPNI algorithm for EQDAs.

Since we cannot hope for an overall polynomial time algorithm, we develop a heuristic that constructs an EQDA from a given sample in polynomial time. For the case of learning DFAs from samples of positive and negative examples, such a heuristic is the passive RPNI algorithm [47]. It takes as input a sample (E, C), where E is a set of positive example words and C is a set of counter-example words, and constructs a DFA consistent with (E, C), that is, accepting all words in E and rejecting all words in C.

The RPNI algorithm is a state merging algorithm. It starts by constructing the *prefix tree acceptor* for E, i.e., the (partial) DFA whose states are all the prefixes of words of E, the transitions correspond to extending a word by the corresponding letter, and the final states are the words in E. This initial DFA precisely accepts E. Then, RPNI successively tries to merge pairs of states according to the canonical order of words. Each merge of a pair of states might require further merges to preserve determinism of the automaton. If a final and a non-final state are merged, then the combined state becomes final.

For every candidate-merge, RPNI checks whether the DFA resulting from this merge operation is still consistent with (E, C). If the check passes, RPNI keeps this merge and continues trying the next candidate-merges. Otherwise, it simply discards the recent candidate-merge and proceeds with the DFA as it was before the merge. This process continues until all candidate-merges are exhausted. Stated compactly, the rough structure of RPNI is as shown in Figure 3.

It is not hard to verify that RPNI indeed produces a consistent DFA: it starts with the prefix tree acceptor of E, which is clearly consistent with (E, C) because E and C are disjoint (otherwise there is no DFA consistent with (E, C)), and only keeps merges that do not violate the consistency. Moreover, the resulting DFA is never larger than the prefix acceptor for (E, C) because RPNI exclusively merges states. Note that RPNI does not construct the smallest DFA consistent with the sample because an early merge might prevent later merges which could have produced a smaller automaton.

- Init: Construct an initial automaton that is consistent with the sample.
- In some fixed order over pairs of words (u, v) that are prefixes of the examples E
  - Merge: Modify the current DFA by merging the states reached by u and v.
  - Test: Keep the merge if the new DFA is still consistent with the sample, otherwise discard it.

Figure 3. The rough structure of RPNI.

We now explain how we adapt RPNI to the setting of ICE-learning for EQDAs. We list the differences to the setting of the original RPNI algorithm, and then explain how to adapt the parts Init, Merge, and Test (from Figure 3) to deal with these differences.

1. We have to deal with a different type of samples. First of all, we have implications in addition to the examples and counter examples, and since we are learning universally quantified properties, we want to use the set-based formalism introduced at the beginning of this section. This means that the 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.
2. We work with an automaton model in which final states are labeled by formulas, that is, they are not simply accepting or rejecting.
3. We need to adapt RPNI such that it constructs elastic automata.

The initialization Init takes into account all the three aspects above and constructs an EQDA that is consistent with the sample. For this purpose, we consider the set of all positive examples, that is, the set $E := \bigcup_{S \in \hat{E}} S$. This is a set of valuation words, from which we strip off the data part, obtaining a set E' of words only made up of pointers and universally quantified variables. We start with the prefix tree of E' as 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 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 $\mathcal{A}$, a result that we borrow from [24]. Hence, if this EQDA is not consistent with the sample, then either there is no such EQDA, or an implication is violated and we proceed as above.

The result of this adapted initialization phase is thus an EQDA that is consistent with the sample, and whose states correspond to words or sets of words that are prefixes of the positive examples.

The main loop over all pairs of words is kept as in the original algorithm (Figure 3). The Merge of states is similar to the one for plain DFAs. When merging accepting states, the new formula at the combined state is obtained as the least upper bound of the formulas of the original state. It is important to note that merging states of

an EQDA again results in an EQDA because it cannot introduce $\underline{b}$-transitions that are not self-loops.

Finally, the Test routine that checks whether a merge was successful is simply a check whether the merged EQDA is consistent with the sample, i.e., carried out by running the automaton on all valuation words and checking if all words from $\hat{E}$ are accepted, for each $C \in \hat{C}$ there is one word in C that is rejected, and for each $(X, Y) \in \hat{I}$ such that all words in X are accepted, all words from Y are accepted.

As in the case of RPNI, it directly 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 different EQDAs, and therefore by Lemma 3.1 we conclude that the above learning is strongly convergent. Note however, that the number of rounds for convergence is not polynomial, in general.

THEOREM 6.2. *The adaption of the RPNI algorithm for iterative set-based ICE-learning of EQDAs strongly converges.*

Experimental Results

Table 2. Results for RPNI-based ICE-learning for quantified array invariants.

Program	#Rounds	$\#\hat{E}$	$\#\hat{C}$	$\#\hat{I}$	#States	Time (s)
compare	9	3	2	5	8	1.3
copy	4	1	1	1	8	0.7
copy-lt-key	5	0	2	3	13	1.2
find	9	2	2	6	8	1.2
init	4	1	1	1	8	0.6
max	3	0	1	1	8	0.4
sorted-find	8	4	3	0	17	5.1
sorted-insert	6	0	6	0	21	2.0
devres	3	0	1	1	8	0.7
rm_pkey	3	0	1	1	8	0.7

We implemented a prototype [2] of the set-based ICE-learning algorithm for learning quantified invariants over arrays and lists, that we just described above in this section. Now, we first describe the implementation details of the teacher, followed by the description of the implementation of the RPNI-based 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) [12] or the decidable STRAND fragment over lists [41]. 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 previous subsection.

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 [9] to support the above described set-based ICE-learning algorithm. We use Z3 [19] (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-based 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 build the teacher for lists and evaluate the learning algorithm over list-manipulating programs.

In Table 2 we report for each program the number of rounds taken by the iterative learning algorithm; the number of examples, counter-examples and implications added to the sample of the learner, over all rounds; the number of states in the final EQDA conjectured and the total time taken to learn that EQDA. The program *sorted-find* finds the presence of a key in a sorted array; the program *sorted-insert* reads an array which is sorted from the second position onwards and inserts the first element of the array at its correct position such that the entire array becomes sorted; the program *devres* and *rm_pkey* are methods adapted from the Linux kernel and an Infiniband device driver, both mentioned in [39].

In recent work, [24] proposed an L^* based learning algorithm for learning quantified invariants over arrays and lists. First, that algorithm does not use an honest teacher. The teacher does not know an invariant before hand and therefore answers L^* membership queries in an arbitrarily chosen way if when it does not know the answer. Secondly, that algorithm learns from only positive configurations that manifest themselves in test runs. Hence, the invariants synthesized in [24] are only *likely* invariants (and might not actually be invariants). On the other hand, the learning algorithm we present here is based on the ICE paradigm and uses an honest teacher. The algorithm is robust and guarantees convergence to the actual invariant, regardless of the way the teacher answers equivalence queries. Though our learning algorithm is more complex than [24], we show through experiments that our algorithm learns an adequate invariant in reasonable time, requiring only a few number of rounds and a small sample size.

7. 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-

[2] experimental results available as supplementary material

framework. We have shown several extensions to ICE-learning, ranging from monotonic frameworks (conjunctions, k-CNF, rectangles, all abstract domains), a non-monotonic numerical data domain (Boolean formulas of octagonal constraints) and a quantified domain for lists and arrays. We have established strong convergence results for several of these, implemented the latter two, and shown them effective in generating adequate program invariants.

Several future directions are worth pursuing. First, we would like to adapt certain several existing learning algorithms (like [52]) to the ICE-setting. Second, we believe that purely *passive* learning (such as in Daikon [20] and recent work on learning quantified invariants [24]) are extremely scalable techniques that learn likely invariants from positive examples only. We would like to combine these with the ICE-learning techniques presented here in order to build more scalable as well as correct invariant generators. Finally, we would like to build a comprehensive ICE-learner that combines several ICE-domains and evaluate such a learner on programs with complex control and multiple modules.

References

[1] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. Safari: Smt-based abstraction for arrays with interpolants. In *CAV*, volume 7358 of *LNCS*, pages 679–685. Springer, 2012.

[2] R. Alur, P. Cerný, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109. ACM, 2005.

[3] R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *CAV*, volume 3576 of *LNCS*, pages 548–562. Springer, 2005.

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[5] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4): 319–342, 1987.

[6] D. Angluin. Negative results for equivalence queries. *Machine Learning*, 5:121–150, 1990.

[7] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3. ACM, 2002.

[8] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[9] B. Bollig, J.-P. Katoen, C. Kern, M. Leucker, D. Neider, and D. R. Piegdon. libalf: The Automata Learning Framework. In *CAV*, volume 6174 of *LNCS*, pages 360–364. Springer, 2010.

[10] A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI*, 2012.

[11] A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, 2011.

[12] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI*, 2006.

[13] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, volume 2619 of *LNCS*, pages 331–346. Springer, 2003.

[14] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent c. In *TPHOLs*, pages 23–42. Springer, 2009.

[15] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, volume 2725 of *LNCS*, pages 420–432. Springer, 2003.

[16] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[17] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.

[18] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118. ACM, 2011.

[19] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[20] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *ICSE*, 2000.

[21] G. Filé and F. Ranzato. Improving abstract interpretations by systematic lifting to the powerset. In *GULP-PRODE (1)*, 1994.

[22] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, 2001.

[23] R. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, number 19 in Proceedings of Symposia in Applied Mathematics, pages 19–32. AMS, 1967.

[24] P. Garg, C. Löding, P. Madhusudan, and D. Neider. Learning Universally Quantified Invariants of Linear Data Structures. In *CAV*, 2013.

[25] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[26] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127. ACM, 2006.

[27] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246. ACM, 2008.

[28] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292. ACM, 2008.

[29] A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *CAV*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.

[30] A. Gupta, R. Majumdar, and A. Rybalchenko. From tests to proofs. In *TACAS*, volume 5505 of *LNCS*, pages 262–276. Springer, 2009.

[31] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.

[32] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.

[33] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[34] F. Ivancic and S. Sankaranarayanan. NECLA Static Analysis Benchmarks. `http://www.nec-labs.com/research/system/systems_SAV-website/small_static_bench-v1.1.tar.gz`.

[35] R. Jhala and K. L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, pages 459–473. Springer, 2006.

[36] R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV*, volume 4590 of *LNCS*, pages 193–206. Springer, 2007.

[37] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[38] M. J. Kearns and U. V. Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11193-4.

[39] S. Kong, Y. Jung, C. David, B.-Y. Wang, and K. Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In *APLAS*, 2010.

[40] W. Lee, B.-Y. Wang, and K. Yi. Termination analysis with algorithmic learning. In *CAV*, 2012.

[41] P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In *POPL*, pages 611–622. ACM, 2011.

[42] K. L. McMillan. Interpolation and SAT-Based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[43] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, volume 4963 of *LNCS*, pages 413–427. Springer, 2008.

[44] A. Miné. The octagon abstract domain. In *WCRE*, pages 310–, 2001.

[45] T. M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997. ISBN 978-0-07-042807-2.

[46] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *ICSE*, pages 683–693. IEEE, 2012.

[47] J. Oncina and P. Garcia. *Pattern Recognition and Image Analysis*.

[48] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *SAS*, volume 4134 of *LNCS*, pages 3–17. Springer, 2006.

[49] M. N. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. In *SAS*, 2009.

[50] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. In *CAV*, volume 7358 of *LNCS*, pages 71–87. Springer, 2012.

[51] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, volume 7792 of *LNCS*, pages 574–592. Springer, 2013.

[52] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, and A. V. Nori. Verification as learning geometric concepts. In *SAS*, volume 7935 of *LNCS*, pages 388–411. Springer, 2013.

[53] A. Thakur, A. Lal, J. Lim, and T. Reps. PostHat and all that: Attaining most-precise inductive invariants. Technical Report TR1790, CS Dept., University of Wisconsin, Madison, WI, April 2013.

A. RPNI for QDAs

In this section, we describe in detail how we adapt the RPNI algorithm to the setting of Section 6, 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 functions 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 [24]. 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 6.

A.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 [24]. 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 $a \in \Sigma$:

if $p \sim q$ and $\delta(p, a), \delta(q, a)$ 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

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

A.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 works as follows:

- It first constructs an »*initial Moore machine*« $\mathcal{A}$ that satisfies a »*property p*« from »*sample data*«. This sample data might be any collection of words together with their output.
- 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. 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`(C) constructs a Moore machine $\mathcal{A}$ that passes `test` ($\mathcal{A}$) from a collection C of sample data.
- 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.

Algorithm 1: The Generic State Merging algorithm.

Input: A collection C of sample data
Output: A Moore machine $\mathcal{A}$ that passes `test`($\mathcal{A}$)

$\mathcal{A}_{\text{init}} = (Q, \Sigma, \Gamma, q_0, \delta, f) \leftarrow$ `init`(C);
$(q_0, \dots, q_n) \leftarrow$ `order`(Q);
$\sim_0 \leftarrow \{(q, q) \mid q \in Q\}$;
for $i = 1, \dots, n$ **do**
  if $q_i \not\sim_{i-1} q_j$ *for all* $j \in \{0, \dots, i-1\}$ **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 $\mathcal{A}_{\text{init}}/_{\sim_n}$;

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.

Remark A.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.

A.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 that is consistent with a sample $\mathcal{S} = (S_+, S_-)$ where S_+, S_- are two disjoint, finite sets of words. 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 collection of sample data is $C = S_+$.
- `test`($\mathcal{A}$) checks whether all $u \in S_+$ have output 1 and all $u \in S_-$ have the output 0, i.e., whether all words from S_+ are "accepted" and those of S_- are "rejected".
- `init`(C) constructs the prefix tree "acceptor" of S_+, i.e., the partial Moore machine $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \delta, f)$ where $Q = \mathit{Pref}(S_+)$, $q_0 = \varepsilon$,
$$\delta(u, a) = \begin{cases} ua & \text{if } ua \in \mathit{Pref}(S_+) \\ \text{undefined} & \text{else} \end{cases},$$
and
$$f(u) = \begin{cases} 1 & \text{if } u \in S_+ \\ 0 & \text{else} \end{cases}.$$
- `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 A.1 yields that the Moore Machine returned by Algorithm 1 when interpreted as a DFA is consistent with $\mathcal{S}$.

ICE-learning of EQDAs We now can formally present our ICE-learning algorithm of Section 6.

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 $(\hat{E}, \hat{C}, \hat{I})$ 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 $(\hat{E}, \hat{C}, \hat{I})$.

We create the initial Moore machine from the given sample $(\hat{E}, \hat{C}, \hat{I})$, i.e.,

- the collection of sample data is $C = (\hat{E}, \hat{C}, \hat{I})$.

The process of creating an initial Moore machine (EQDA) from a sample (the function `init`) is more elaborate 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 6 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 $Q = \mathit{Pref}(E')$, $q_0 = \varepsilon$,

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

   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 [24] 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 A.1 yields that the GSM instance from above produces an EQDA that is consistent with the given sample provided that such an EQDA exists.