Automating Program Verification

Pranav Garg
University of Illinois at Urbana-Champaign
garg11@illinois.edu

Abstract

The problem of generating adequate invariants to show a program correct is at the heart of automated program verification. In a deductive verification setting, manually annotating the program with these invariants places a huge burden on the user. Also, the resulting verification conditions often fall outside decidable fragments, in which case the user has to guide the proof of validity of these verification conditions by using proof tactics, and manually providing various low-level lemmas, etc. In this thesis, we develop techniques to automatically synthesize program invariants (for both sequential and concurrent programs), thereby reducing the annotation burden on the user and thus increasing the automaticity of program verification. We adapt ideas from computational learning theory and machine learning and develop new learning algorithms for synthesizing these invariants. We also develop *natural proofs* in the context of verifying complex heap-manipulating data-structure programs (verification conditions for which usually fall outside decidable fragments) and asynchronously communicating distributed programs. Natural proofs is a fixed set of sound but incomplete proof tactics that are completely automatic and work effectively for most programs encountered in practice. We plan to deploy the techniques we have developed to deductively verify a critical software and, in the process, validate that our techniques help in automating deductive verification and thus help it scale to larger programs.

1 Introduction

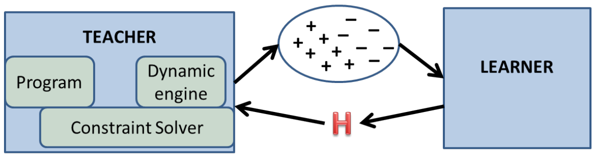
The problem of generating adequate invariants to show a program correct is at the heart of automated program verification. Automated program verifiers invariably synthesize invariants: abstract interpretation [27] finds invariants using fixed-points evaluated over an abstract domain, counter-example guided predicate abstraction [9] iteratively computes predicates and uses model-checking to establish invariants, etc. Floyd-Hoare style deductive verification [33, 45] takes a program annotated with modular pre/post condition annotations for every method, class invariants and loop invariants, etc., and reduces the problem of program verification to checking the validity of certain logical *verification conditions*, which can be discharged by using either automated theorem provers or SMT solvers. With the advent of very scalable SMT solvers in the past decade, deductive verification technology can now scale to very large programs [25, 10, 53, 65, 57].

However, providing these program annotations, to begin with, places a huge burden on the user as writing these annotations can often be very tedious and complex (specially writing loop invariants or pre/post-condition annotations for internal methods, which have to specify properties over intermediate program configurations). Even in the context of concurrent programs, deductive verification a la Owicki and Gries [74] or Jones [51] requires the user to provide program annotations in the form of rely-guarantee contracts, stable program invariants, etc. Providing these annotations requires considerable human effort and is one of the main reasons why the use of automated verification is still languishing in the software industry.

Another challenge to deductive verification arises from the fact that the resulting verification conditions for complex programs do not always fall into decidable theories. Specially, specifying properties of heap-manipulating programs requires *quantification* in some form, which immediately precludes the use of most SMT decidable theories. For the validation of such verification conditions, the user has to guide the verification by providing low-level lemmas, proof tactics, heuristics like quantifier instantiation triggers, etc., to help the proof go through.

In this thesis, we successfully address both these challenges to automated program verification. We develop techniques to automatically synthesize program invariants (for both sequential and concurrent programs), thereby reducing the annotation burden on the user. The problem of automatically synthesizing these program invariants is, in general, undecidable and involves an instrinsically-hard step of *inductive generalization*. We adapt ideas from computational learning theory [52] and machine learning [69] and develop new learning algorithms for automatically synthesizing these invariants. Also, for deductive verification of more complex heap-manipulating programs, we develop a fixed set of sound but incomplete proof-tactics called *natural proofs* that are completely automatic, and work effectively for a large class of these programs encountered in practice. With the same idea of automating program verification, we also study the problem of verification of asynchronously communicating distributed systems and develop natural proofs for their automated verification.

Invariant generation techniques can be broadly classified into two kinds: white-box techniques where the synthesizer of the invariant is acutely aware of the precise program and property that is being proved and black-box techniques where the synthesizer is largely agnostic to the structure of the program and property, but works with a partial view of the requirements of the invariant. Abstract interpretation [27], counter-example guided abstraction refinement, predicate abstraction [9], the method of Craig interpolants [67, 49], IC3 [16], etc. all fall into the white-box category. In this thesis, we explore 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 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 [23, 5, 4], and has gained considerable excitement and traction in recent years [80, 79, 78, 35, 34].

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

However, we found that traditional learning algorithms that offer learning from *positive examples*, and in some case, *positive and negative examples* do not form a robust learning framework for synthesizing program invariants [34]. We propose a new learning framework called *ICE-learning* which stands for learning using *examples*, *counter-examples* and *implications* [34]. We propose that one should build new learning algorithms for synthesizing program invariants that do not just take examples and counter-examples, as most traditional learning algorithms do, but instead also handle implications. We show, by building various ICE-learning algorithms for synthesizing program invariants, that ICE-learning is a robust learning model that ensures progress and leads to honest teachers. Our main contributions are as follows.

- We show that the Houdini algorithm [32] and the recently proposed *abstract* Houdini algorithm [83] for learning invariants over abstract domains are actually ICE-learning algorithms (Section 4). We apply the abstract Houdini learning algorithm (Section 8.1) to learn invariants over numerical domains like intervals and pentagons [58] for concurrent programs, and prove numerical properties like array-index-within-bounds, data race freedom, etc.

- We develop a new non-monotonic ICE-learning algorithm for Boolean combinations of numerical constraints (Section 5). We adapt existing template-based synthesis techniques that use a constraint solver [43, 42, 26] to a black-box ICE-learning algorithm for synthesizing invariants, and prove that this 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 introduce *quantified data automata* (QDA) in Section 6 which is a normal form for representing quantified properties of linear data structures and build a polynomial time active learning algorithm for QDAs. We identify a subclass of QDAs called *elastic* QDAs and show that (a) elastic QDAs can be converted to formulas of decidable logics, to the array property fragment [17] when modeling arrays and the decidable fragment of Strand [59, 61] when modeling lists; (b) the active learning algorithm for QDAs can be modified to obtain an active learning algorithm for elastic QDAs. We deploy this active learning algorithm in a passive setting where we learn quantified invariants over arrays and lists by learning elastic QDAs from configurations that manifest in dynamic test runs.

- We also develop a new non-monotonic ICE-learning technique for *quantified* invariants over arrays and lists (Section 6.3). 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 new learning algorithm extends the classical RPNI learning algorithm for automata [73] to learning in the ICE-model and further learns elastic QDAs, which can be converted to quantified logical formulas over arrays/lists. We build a prototype verifier by building this learner and the teacher as well, and show that robust ICE-learning is still effective.

- For the deductive verification of more complex heap-manipulating programs, whose verification conditions usually always fall outside decidable logics, we introduce natural proofs (Section 7) which is a fixed set of sound but incomplete proof tactics for discharging these verification conditions, that is completely automatic and works effectively for a large class of data-structure programs encountered in practice. As part of future work, we plan to build learning algorithms for synthesizing invariants of these data-structure programs so that, coupled with the natural proofs methodology, we can use them to prove these programs correct in a completely automatic manner.
- We develop natural proofs for verifying asynchronous systems communicating via FIFO message queues, against local specifications like responsiveness of the system, etc. (Section 8.2). Our proof is in the form of a sound exploration strategy that is also complete for most asynchronous systems encountered in practice. We use natural proofs for proving the responsiveness of various windows device drivers as well as several distributed protocols.

As part of future work, I plan to apply the technqiues we have developed, which includes algorithms to learn program invariants and natural proofs for validating the correctness of a proof of heap manipulating programs and asynchronous event-driven programs, in a deductive verification setting to verify some critical software. In the landscape of software verification, most automatic tools like Astree, SLAM [9] or various static analysis tools verify programs only against shallow safety properties. With the technology of automatically synthesizing program invariants and the technique to automatically discharge verification conditions even for complex heap-manipulating programs, we plan to show that automatic program verification is certainly possible for verifying programs even against more-complex and deeper specifications.

2 Related Work

Prominent white-box techniques for invariant synthesis include abstract interpretation [27], interpolation [67, 49] and IC3 [16]. Template based approaches to synthesizing invariants using constraint solvers has been explored in a white-box setting in [26, 42, 43], 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. Most of them are either based on abstract interpretation [44, 15, 28, 41] or are based on interpolation theorems for array theories [50, 68, 3, 77].

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 [31] proposed conjunctive Boolean learning to learn *likely* invariants from program configurations recorded along test runs. Learning was introduced in the context of verification by Cobleigh et al. [23], which was followed by applications of Angluin's L^* algorithm [7] to finding rely-guarantee contracts [5] and stateful interfaces for programs [4]. Houdini [32] 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. [83] 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 [80, 79, 78] by using scalable machine learning techniques [69, 52] 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). Boolean formula learning has also been applied recently for learning quantified invariants in [54].

Turning to general learning theory, the field of algorithmic learning theory is a well-established field [52, 69]. 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 [8] for an overview). A celebrated result is the polynomial-time learning of regular languages by Angluin which uses membership and equivalence queries. Gold [39] showed that the problem of finding the smallest automaton consistent with a set of accepted and rejected strings is NP-complete. The RPNI algorithm [73] 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 Section 6.3 in order to learn quantified invariants for arrays/lists.

There is a rich literature on analysis of heaps in software. We omit discussing literature on general interactive theorem provers (like Isabelle [71]) that require considerable manual guidance. We also omit a lot of work on analyzing shape properties of the heap [64, 85, 20, 30, 11], as they do not handle complex functional properties. There are several proof systems and assistants for separation logic [76, 72] that incorporate proof heuristics and are incomplete. Symbolic execution with separation logic has been used in [13, 12, 14] to prove structural specifications for various list and tree programs. These tools come hardwired with a collection of axioms and their symbolic execution engines check the entailment between two formulas modulo these axioms. Verifast [48], on the other hand, chooses flexibility of writing richer specifications over complete automation, but requires the user to provide inductive lemmas and proof tactics to aid verification. Similarly, Bedrock [22] is a Coq library that aims at mostly automated (but not completely automated) procedures that requires some proof tactics to be given by the user to prove verification conditions. A work that comes very close to our work on natural proofs in Section 7 is a paper by Chin et al. [21], where the authors allow user-defined recursive predicates (similar to ours) and build a terminating procedure that reduces the verification condition to standard logical theories. However, their procedure does not search for a proof in a well-defined simple and decidable class, unlike our natural proof mechanism; in fact, the resulting formulas are quantified and incompatible with decidable logics handled by SMT solvers.

The idea of unfolding recursive definitions and formula abstraction, which is one of the main components of natural proofs, also features in the work by Suter et al. [81, 82], where a procedure for algebraic data-types is presented. However, this work focuses on soundness and completeness, and is not terminating for several complex data structures like red-black trees. Moreover, the work limits itself to functional program correctness; in our opinion, functional programs are very similar to algebraic inductive specifications, leading to much simpler proof procedures. The natural proof methodology was introduced in [63], but was exclusively built for tree data-structures. In particular, this work could only handle *recursive* programs, i.e., no while-loops, and even for tree data-structures, imposed a large number of restrictions on pre/post conditions for methods— the input to a procedure had to be only a single tree, the method can only return a single tree, and even then must havoc the input tree given to it. Also, structures such as doubly-linked lists, trees with parent pointers, etc. are out of scope of this work. In our work, we can handle user-defined structures expressible in separation logic, multiple structures and their separation, programs with while-loops, etc., because of our *logical* treatment of separation logic using classical logic.

3 ICE: A Robust Learning Framework for Synthesizing Invariants

Algorithmic learning theory [52] and machine learning techniques [69] 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, learning using examples and counter-examples does not form a robust learning 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 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 passes a data-point p in H that shouldn't belong to the invariant as a *negative* example.

(c) whether H is inductive; i.e., whether the strongest post of $H \wedge b$ across the loop body L implies H; if not, 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 *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 [23, 5, 4, 35], the teacher cheats: she arbitrarily makes one choice and goes with that, hoping that it will result in an invariant. However, this makes the entire framework non-robust, causing divergence, blocking the learner from learning the simplest concepts, and introducing arbitrary bias that is very hard to control. To investigate learning as a serious paradigm for synthesizing invariants, one needs to fix this foundationally within the framework itself.

In our work [34], we introduce a new learning framework called ICE-learning, which stands for *learning using Examples, Counter-examples, and Implications*. We propose in [34] 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.

3.1 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.

In classical learning frameworks (see [52]), 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 $\mathcal{C}$ 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 $\mathcal{C} \subseteq 2^D$ 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 are typically *infinite*, 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 $H \in \mathcal{C}$ 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*.

A natural active learning question is *membership* (as in Angluin's learning algorithm [7]), where the learner is allowed to ask whether an element belongs to the target concept. However, in our setting the teacher does not have a precise target concept in mind and therefore cannot, in general, answer membership queries. In order to define an active learning framework in the invariant generation setting, we only allow *correctness queries*. 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 $H \in \mathcal{C}$ 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 the teacher answers "correct" and the learning process terminates. Otherwise, the teacher returns either some element $d \in D$ with $d \in P \setminus H$ or $d \in H \cap N$, or an implication $(x, y) \in R$ with $x \in H$ and $y \notin H$. This new information is added to the sample of the learner.

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

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

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

Observe now that for a *finite* class $\mathcal{C}$ 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.*

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

We can apply such learning for finding invariants in a variety of settings, including programs with recursion, where we find not only loop invariants but pre/post conditions for methods. The learning of the invariant will simultaneously learn all these annotations, and the teacher can find the answers by generating verification conditions and using automatic theorem provers (since the parts of the program connecting annotations will be loop-free). In general, the positive examples will arise from propagating the pre-condition for the program across code, the negative examples will arise from (weakest pre-conditions of) the specifications, including the assertions and post-condition of the program, and the implications will arise from non-inductiveness of the hypothesis. The ICE-learning algorithm has the following salient features:

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 the safety specification 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.

In this thesis we have developed several new ICE-learning algorithms for learning program invariants, belonging to different concept classes, ranging from scalar invariants to more-complex quantified invariants over arrays and lists. We now describe these algorithms briefly in the following sections.

4 Monotonic ICE Frameworks

We study in [34] a class of ICE-learning frameworks, which we call *monotonic* frameworks, that are extremely simple to build ICE-algorithms for. We 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.

We argue in [34] that the learner described above terminates in a finite number of steps and in each round, when it produces a hypothesis H, it produces the least concept that meets the constraints imposed by the samples. Consequently, the above ICE-learner is *consistent*; it produces hypotheses that satisfy all constraints. Also 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 briefly describe various 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 [52]. The derived ICE-learning algorithm, in essence, is close to the well-known technique called *Houdini* in program verification [32]. 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 [52]. 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 [52] 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.

ICE-learning over abstract lattices: We show that one can in fact extend monotonic ICE learning to *any* abstract interpretation [27] 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. It immediately follows that for a large class of numerical domains such as intervals, octagons, convex polyhedra, linear equalities, etc., we obtain ICE-learning algorithms. Moreover, the iterative ICE-learner converges whenever the lattice has finite height.

The above scheme of using samples to learn an element of the abstract domain is in essence the *abstract Houdini* algorithm proposed recently by Thakur et al. [83], and is similar to another recent work [38]. In the past, a similar idea of using concrete samples has been explored in the context of building precise abstract transformers [86]. We simply observe that these can in fact be seen as ICE learning algorithms. In Section 8.1 we describe our current work of using these ICE-learners to perform static analysis of concurrent programs over numerical abstract domains. We use ICE-learning to prove safety properties like data race freedom, array indices within bounds, etc. in concurrent (linux and windows) device drivers.

In some sense, the monotonic frameworks above achieve ICE-learning simply from a passive learner for examples, exploiting monotonicity of the class of concepts. The algorithms we describe in the next two sections are more general schemes that truly use examples, counter-examples, and implications.

5 Template-based ICE-learning for Numerical Constraints

We now describe a learning algorithm for synthesizing invariants that are arbitrary Boolean combinations of numerical constraints [34]. Since we want the learning algorithm to generalize the sample (and not capture precisely the finite set of implication-closed positive examples), we would like it to learn a formula with the *simplest* Boolean structure. In order to do so, we iterate over templates over the Boolean structure of the formulas, and learn a formula in the given template.

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

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

A template fixes the number of disjuncts and conjuncts in the DNF representation of the desired formulas. It also restricts the constants $c \in \mathbb{Z}$ appearing in the atomic constraints to lie within a finite range

$[-Max, +Max]$, for $Max \in \mathbb{Z}^+$ (in order to achieve strong convergence as described later). For a given template $\bigvee_i \bigwedge_j \alpha^{ij}$, the iterative ICE-learning algorithm described in [34] learns an adequate invariant φ, of the form:

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

The iterative learning algorithm consists of an ICE-learner paired with a white-box teacher. Given a sample (E, C, I), the learner's task is to construct 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 φ is consistent with the sample; i.e., for every data-point $p \in E$, $\varphi(p)$ holds; for $p \in C$, $\varphi(p)$ does not hold; and for every implication pair $(p, p') \in I$, $\varphi(p')$ holds if $\varphi(p)$ holds. Unfortunately, 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-learner uses a constraint solver to search this exponential space in a reasonably efficient manner. It does so by checking the satisfiability of Ψ (see Figure 1), over the free variables s_k^{ij}, v_k^{ij} and c^{ij}, which precisely captures all the ICE-constraints.In this formula, b_p is a Boolean variable which tracks $\varphi(p)$; the Boolean variables b_p^{ij} represent the truth value of $(s_1^{ij}\, v_1^{ij} \; + \; s_2^{ij}\, v_2^{ij} \; \le \; c^{ij})$ on p (line 2 of the formula); the variables r_{kp}^{ij} encode the value of $s_k^{ij} v_k^{ij}$ and d_{kp}^{ij} the value of v_k^{ij} (line 3 of the formula).

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

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 Ψ falls in the theory of quantifier-free linear integer arithmetic, the satisfiability of which is decidable. A satisfying assignment for Ψ gives a *consistent* formula that the learner conjectures as an invariant. If Ψ is unsatisfiable, then there is no invariant in the current template consistent with the given sample. In this case we iterate by increasing the complexity of the template as explained below. 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 white-box constraint based approaches to invariant synthesis [42, 43, 26]. These approaches directly encode the adequacy of the invariant (encoding the entire program's body) into a constraint, and use Farkas' lemma to reduce the problem to satisfiability of quantifier-free non-linear arithmetic formulas, which is harder and in general undecidable. On the other hand, we split the task between a white-box teacher and a black-box learner, communicating only through ICE-constraints on concrete data-points. This greatly reduces the complexity of the problem, leading to a simple teacher and a much simpler learner. The idea is somewhat similar to [78] which use algebraic techniques to guess the coefficients.

Convergence: A template fixes the Boolean structure of the invariants and also restricts the constants appearing in the invariants to a finite range. For a given template, the concept class is hence finite. As our ICE-learner always conjectures a *consistent* hypothesis, it follows from Lemma 3.1 that our learning algorithm, in a finite number of rounds, either learns an adequate invariant or discovers that no such invariant exists in the given template. In the latter case, we dovetail between the Boolean structure of the template and the range of constants specified in the template. Since for a given template our algorithm runs for only a finite number of rounds, 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 bound the constants. In practice, searching for invariants with unbounded constants would be useful in cases where the invariant does require large constants.

Program	Black-Box												White-Box
	#V	#D	#C	#R	#E	#C	#I	Learned Invariant	L (s)	T (s)	ICE (s)	[79] (s)	InvGen (s)
cegar1 [42]	2	1	1	2	3	1	4	$x - y \leq 2$	0.0	0.2	0.2	0.2	0.1
cegar2 [42]	3	1	3	5	9	10	12	$x \leq N \wedge 0 \leq m \leq x - 1$	1.3	0.8	2.1	Fail	Fail
fig1 [42]	2	2	1	4	18	8	9	$x \leq -9 \vee x \leq y - 1$	0.4	0.6	1.0	Fail	Fail
w1 [42]	2	1	1	3	13	8	0	$x \leq n$	0.0	0.4	0.5	0.2	0.1
w2 [42]	2	1	1	3	21	8	0	$x \leq n - 1$	0.1	0.3	0.4	0.1	0.1
tacas06 [49]	4	3	1	5	18	12	12	$i \leq j - 1 \vee i \geq j + 1 \vee$ $x = y$	3.9	0.9	4.8	0.5	0.1
fig3 [40]	3	2	1	1	15	10	4	$lock = 1 \vee x = y - 1$	0.2	0.3	0.5	0.1	0.1
fig6 [40]	1	1	1	1	5	0	0	*true*	0.0	0.0	0.0	0.1	0.1
fig8 [40]	6	1	1	2	5	4	0	$lock = 0$	0.0	0.2	0.2	0.0	0.0
fig9 [40]	2	1	2	5	4	8	12	$x \geq 0 \wedge y \geq 0$	0.3	0.6	0.9	0.2	0.1
ex23 [47]	3	1	4	8	53	20	16	$0 \leq y \leq z \wedge$ $c \leq z \leq c + 4572$	6.0	1.3	7.3	Fail	-
ex7 [47]	4	1	2	6	10	20	0	$0 \leq i \wedge y \leq len$	0.4	0.7	1.1	0.4	-
ex14 [47]	2	1	1	2	15	4	0	$x \geq 1$	0.0	0.2	0.2	0.2	-
array [1]	3	2	1	4	17	8	12	$j = 0 \vee m \leq array[0]$	0.6	0.7	1.3	0.2	-
for-inf.- -loop-1 [1]	4	1	1	2	7	4	4	$x = 0$	0.1	0.2	0.3	0.4	0.1
n.c11 [1]	2	1	2	3	5	8	4	$0 \leq len \leq 4$	0.1	0.4	0.5	0.2	0.1
trex01 [1]	4	1	1	2	7	4	0	$z >= 1$	0.0	0.2	0.2	0.4	0.1

Table 1: Results for template-based iterative ICE-learning for numerical constraints. #V is the number of variables; #D, #C is the number of disjunctions and conjunctions in the invariant template; #R is the number of rounds of iterative-ICE learning required; #E, #C and #I are the number of examples, counter-examples and implications in the final sample; L and T is the time taken by the learner and the teacher respectively and ICE is the total time taken by our prototype.

Experimental Results: We implemented a prototype of the above ICE-learning algorithm for numerical invariants. Given a sample (E, C, I), the ICE-learner uses an SMT solver, as described above, to learn a hypothesis consistent with the sample. The teacher generates verification conditions for the program and uses an SMT solver to check whether the hypothesis is adequate. If not, the teacher adds more constraints to the sample and this process iterates till an adequate invariant is found.

We evaluate our learning algorithm on various programs[1] from the literature (see Table 1). We compare our ICE-algorithm to InvGen [43], which uses a white-box constraint-based approach to invariant synthesis, and [79], which is a machine learning algorithm to learn numerical invariants. InvGen is quite mature and quickly learns an adequate invariant for most of the programs. However, being white-box, it cannot handle programs with arrays (`-` in the table) even though the required invariants are numerical constraints over scalar variables. Being incomplete, it is also unable to prove programs `fig1` and `cegar2`. Our tool learns an adequate invariant in time comparable to [79] for most programs. For programs that involve specific integral values, like the size of the static array is `4608` in `ex23` or the assignment `x := -50` just before the entry to the loop in `fig1`, invariants learned by [79] are very sensitive to the test harness. For `ex23, fig1` and `cegar1`, we could not learn adequate invariants using [79] despite many attempts with different test harnesses.

The preliminary experiments show that our algorithm learns an adequate invariant in reasonable time on these programs.Though we use the more complex but more robust framework of ICE-learning, the time to learn is comparable to other learning algorithms like [79, 80] that learn invariants from just positive and negative examples (but lack robustness).

6 Learning Universally Quantified Invariants of Linear Data Structures

Verification of heap-manipulating programs often involves reasoning about an *unbounded* heap, manipulated by the program. Therefore, invariants of these programs usually use quantifiers to represent properties over such unbounded heaps. Synthesizing such quantified invariants is a huge challenge. In this section, we describe learning algorithms for such *quantified logical formulas describing sets of linear data-structures*. We have built algorithms [35, 34] that can learn formulas of the kind "$\forall y_1, \ldots, y_k\ \varphi$", where φ is quantifier-free, and that captures properties of arrays and lists (the variables range over indices for arrays, and locations for lists, and the formula can refer to the data stored at these positions and compare them using arithmetic, etc.).

We first describe a novel representation (normal form) for quantified properties of linear data-structures, called *quantified data automata* (QDA) and show that the problem of learning such quantified properties can be reduced to learning Moore machines, which are finite-state automata with output on states. We then describe a sub-class of QDAs (called elastic QDAs) which have a decidable emptiness-problem and consequently capture *quantified invariants that are amenable to decision procedures*. Using these automata-based representations, we describe two algorithms for learning universally quantified invariants: a polynomial-time passive learning algorithm based on Angluin's L* algorithm [7] and an active ICE-algorithm based on regular positive negative inference (RPNI) [73]. We also show in [37] that elastic QDAs form an abstract domain over lists and they can also be used for synthesizing quantified list invariants, in a white-box setting, using abstract interpretation [27].

[1]Available at `http://web.engr.illinois.edu/~garg11/tacas14/numerical_domain/`

6.1 Quantified Data Automata

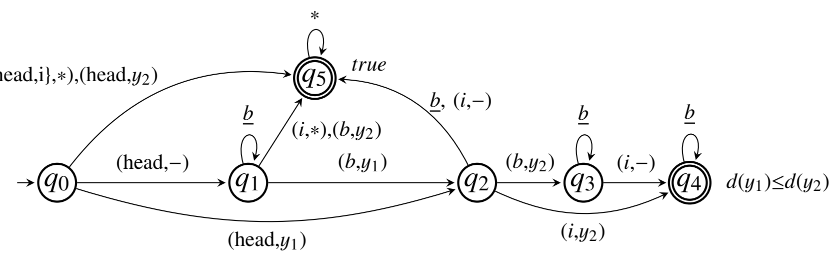
We are interested in learning invariants of typical programs manipulating lists and arrays, including various searching and sorting algorithms, algorithms that maintain lists and arrays using insertions/deletions, in-place manipulations that destructively update lists, etc. Consider a typical invariant in a sorting program over lists where the loop invariant is expressed as:

$$head \rightarrow^* i \;\; \wedge \;\; \forall y_1, y_2.((head \rightarrow^* y_1 \wedge succ(y_1, y_2) \wedge y_2 \rightarrow^* i) \Rightarrow d(y_1) \leq d(y_2)) \quad (1)$$

This says that for all cells y_1 that occur somewhere in the list pointed to by *head* and where y_2 is the successor of y_1, and where y_1 and y_2 are 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; in other words, the sublist from *head* to i is sorted. Such an invariant for a program manipulating arrays can be similarly expressed as:

$$\forall y_1, y_2.((0 \leq y_1 \wedge y_2 = y_1 + 1 \wedge y_2 \leq i) \Rightarrow A[y_1] \leq A[y_2]) \qquad (2).$$

We introduce in [35] an automaton model called *quantified data automata* (QDA) that expresses such quantified constraints. The above two invariants are expressed by the following QDA:



The automaton view models program configuration consisting of a linear data structure as a data word, which reads the positions of the pointer variables pointing into the data structure (for eg., variables *head* and i in the above example) as well as the data values in the cells of the structure. The QDA reads (deterministically) data words, along with a valuation of the quantified variables, y_1 and y_2 in the above example. We use two *blank* symbols that indicate that no pointer variable ("b") or no variable from Y ("−") is read in the corresponding component; moreover, $\underline{b} = (b, -)$. Missing transitions go to a sink state labeled *false*. The above automaton 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. 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 above automaton hence accepts precisely those set of data words that satisfy the invariant formula.

It is important to note that QDAs are not necessarily a blown-up version of the formulas they correspond to. For a formula, the corresponding QDA can be exponential, but for a QDA the corresponding formula can be exponential as well (QDAs are like BDDs, where there is sharing of common suffixes of constraints, which is absent in a formula).

Learning Quantified Program Properties using QDAs: In [35], we show that for any regular set of valuation words (data words extended with valuations for the universal variables Y), there is a *canonical* QDA. Using this result, we show that learning valuation words can be reduced to learning *formula words* (which are valuation words with no data but paired with data formulas), which in turn can be reduced to learning Moore machines.

Decidable Fragments and Elastic Quantified Data Automata: The emptiness problem for QDAs is undecidable; in other words, the logical formulas that QDAs express fall into undecidable theories of lists and arrays. Consequently, even if they are learnt in an invariant-learning application, we will be unable to *verify* automatically whether the learnt properties are adequate invariants for the program at hand. Even though SMT solvers support heuristics to deal with quantified theories (like e-matching), in our experiments, the verification conditions could not be handled by such SMT solvers. Our goal is hence to also offer mechanisms to *learn invariants that are amenable to decision procedures*.

A common restriction in the decidable array property fragment [17] as well as the syntactic decidable fragments of Strand [59] is that quantification is not permitted to be over elements that are only a *bounded* distance away. The restriction allows quantified variables to only be related through *elastic* relations (following the terminology in Strand [59, 61]).

For instance, a formula equivalent to the formula in Eq. 1 but expressed in the decidable fragment of Strand over lists is:

$$head \rightarrow^* i \;\;\wedge\;\; \forall y_1, y_2.((head \rightarrow^* y_1 \wedge y_1 \rightarrow^* y_2 \wedge y_2 \rightarrow^* i) \Rightarrow d(y_1) \leq d(y_2)) \quad (3)$$

This formula compares data at y_1 and y_2 whenever y_2 occurs sometime after y_1, and this makes the formula fall in a decidable class. Similarly, a formula equivalent to the formula Eq. 2 in the decidable array property fragment is:

$$\forall y_1, y_2.((0 \leq y_1 \wedge y_1 \leq y_2 \wedge y_2 \leq i) \Rightarrow A[y_1] \leq A[y_2]) \qquad (4)$$

The above two formulas are captured by a QDA that is the same as in the figure above, except that the $\underline{b}$-transition from q_2 to q_5 is replaced by a $\underline{b}$-loop on q_2.

We, consequently, identify a restricted form of quantified data automata, called *elastic quantified data automata* (EQDA) [35], which structurally captures the constraint that quantified variables can be related only using elastic relations (like $\rightarrow^*$ and $\leq$). This is achieved by allowing blank transitions only to occur as self-loops. Furthermore, we show in [35] that EQDAs can be converted to formulas of decidable logics, the decidable fragment of Strand and the array property fragment, and hence expresses invariants that are amenable to decidable analysis across loop bodies. Also we show a surprising *unique minimal over-approximation theorem* that says that for every QDA, accepting say a language L of valuation-words, there is a *minimal* (with respect to inclusion) language of valuation-words $L' \supseteq L$ that is accepted by an elastic QDA. The latter result allows us to learn QDAs and then apply the unique minimal over-approximation (which is effective) to compute the best over-approximation of it that can be expressed by elastic QDAs (which then yields decidable verification conditions).

6.2 Passive-Learning of Universally Quantified Invariants of Linear Data-Structures

We now describe an algorithm for passively learning quantified invariants of linear data structures based on the Angluin's L* algorithm for learning Moore machines. Note that the L* algorithm is an active learning algorithm, consisting of a teacher which can answer membership and equivalence queries of the learner. Consequently, the active learning algorithm can itself be used in a verification framework, where the membership and equivalence queries are answered using under-approximate and deductive techniques (for instance, for iteratively increasing values of k, a teacher can answer membership questions based on bounded and reverse-bounded model-checking, and answer equivalence queries by checking if the invariant is adequate using a constraint solver). However we do not pursue an implementation of active learning as above, but instead build a passive learning algorithm that uses the active learning algorithm.

Our motivation for doing passive learning is that we believe (and we validate this belief using experiments) that in many problems, a lighter-weight passive-learning algorithm which learns from a few randomly-chosen small data-structures is sufficient to find the invariant. Note that passive learning algorithms, in general, often boil down to a guess-and-check algorithm of some kind, and often pay an exponential price in the property learned. Designing a passive learning algorithm using an active learning core allows us to build more interesting algorithms; in our algorithm, the inacurracy/guessing is confined to the way the teacher answers the learner's questions.

The passive learning algorithm works as follows. Assume that we have a finite set of configurations S, obtained from sampling the program (by perhaps just running the program on various random small inputs). We are required to learn the simplest representation that captures the set S (in the form of a QDA). We now use an active learning algorithm for QDAs; membership questions are answered with respect to the set S (note that this is imprecise, as an invariant I must include S but need not be precisely S). When asked an equivalence query with a set I, we check whether $S \subseteq I$; if yes, we can check if the invariant is adequate using a constraint solver and the program.

It turns out that this is a good way to build a passive learning algorithm. First, enumerating random small data-structures that get manifest at the header of a loop fixes for the most part the structure of the invariant, since the invariant is forced to be expressed as a QDA. Second, our active learning algorithm for QDAs promises never to ask long membership queries (queried words are guaranteed to be less than the diameter of the automaton), and often the teacher has the correct answers. Finally, note that the passive learning algorithm answers membership queries with respect to S; this is because we do not know the true invariant, and hence err on the side of keeping the invariant semantically small. This inaccuracy is common in most learning algorithms employed for verification (e.g, Boolean learning [54], compositional verification [24, 6], etc). This inaccuracy could lead to a non-optimal QDA being learnt, and is precisely why our algorithm need not work in time polynomial in the simplest representation of the concept (though it is polynomial in the invariant it finally learns).

The proof of the efficacy of the passive learning algorithm rests in the experimental evaluation [35]. We implement the passive learning algorithm (which in turn requires an implementation of the active learning algorithm). By building a teacher using dynamic test runs of the program and by pitting this teacher against the learner, we learn invariant QDAs, and then over-approximate them using elastic QDAs (EQDAs). These EQDAs are then transformed into formulas over decidable theories of arrays and lists. Using a wide variety of programs manipulating arrays and lists, ranging from several examples in the literature involving sorting algorithms, partitioning, merging lists, reversing lists, and programs from the Glib list library, programs from the Linux kernel, a device driver, and programs from a verified-for-security mobile application platform, we show that we can effectively learn adequate quantified invariants in these settings. In fact, since our technique is a black-box technique, we show that it can be used to infer pre-conditions/post-conditions for methods as well.

6.3 ICE-Learning of Universally Quantified Invariants of Linear Data-Structures

In the passive-learning algorithm we just described, the teacher is not honest, the learner learns only from positive examples obtained using test runs, and hence synthesizes only *likely* invariants. In contrast, we present in this section an ICE-learning algorithm that uses an honest teacher, and guarantees convergence to the actual invariant regardless of the way the teacher answers equivalence queries. To this end, we first extend ICE-learning, described in Section 3, to *universally quantified* concepts. We will then describe an ICE-algorithm for learning universally quantified invariants in linear data structures.

Data-set based ICE-learning: In general, consider universal properties of the form $\psi = \forall y_1, \ldots, y_k \varphi(y_1, \ldots, y_k)$, where φ is a quantifier-free formula. We now describe how to modify the learning setting such that we can use a learner for the quantifier-free property described by $\varphi(y_1, \ldots, y_k)$. We consider for each concrete program configuration c the set S_c of *valuation configurations* of the form (c, val), where val is a valuation of the variables $y_1, \ldots, y_k$. For example, if the configurations are heaps, then the valuation maps each quantified variable y_i to a cell in the heap, akin to a scalar pointer variable. Then $c \models \psi$ if $(c, val) \models \varphi$ for all valuations val, and $c \not\models \psi$ if $(c, val) \not\models \varphi$ for some valuation val.

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

An ICE-learner for the data-set based setting corresponds to an ICE-learner for universally quantified concepts in the original data-point based setting using the following connection. Given a standard target description (P, N, R) over D, we now consider the domain D_{val} extended with valuations of the quantified variables $y_1, \ldots, y_k$ as described above. Replacing each element c of the domain by the set $S_c \subseteq D_{val}$ transforms (P, N, R) into a set-based target description. Then a hypothesis H (described by a quantifier-free formula $\varphi(y_1, \ldots, y_k)$) is correct w.r.t. the set-based target description iff the hypothesis described by $\forall y_1, \ldots, y_k \varphi(y_1, \ldots, y_k)$ is correct w.r.t. the original target description.

In this section, we briefly describe an iterative ICE-learner for concepts represented by EQDAs (see [34] for more details). We start by a result that we cannot hope for a polynomial time iterative ICE-learner for EQDAs, when the set of pointers and quantified variables is unbounded [34] .

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

The above theorem 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.

Adapting the heuristic RPNI algorithm for EQDAs. Since we cannot hope for a polynomial time iterative ICE-learner, we develop a heuristic that constructs an EQDA from a given sample in polynomial time. In the iterative setting this yields a learner for which each round is polynomial, while the number of rounds is not polynomial, in general. We start from the RPNI algorithm [73], which takes as input a sample (E, C) of positive example words E and counter-example words C, and constructs a DFA consistent with (E, C), that is, 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 appending the corresponding letter, and the final states are the words in E. Then, RPNI tries to increase the language accepted by the automaton by successively trying to merge pair of states, according to the canonical ordering of words. Stated compactly, the rough structure of RPNI is as follows:

- 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.

In [34] we explain how we adapt RPNI to the setting of ICE-learning for EQDAs. Here we only list the differences to the setting of the original RPNI algorithm and leave the explanation of how to adapt the parts Init, Merge, and Test to deal with these differences, to [34].

1. The samples contains implications in addition to the examples and counter examples, and we 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. Final states are labeled by formulas instead of being simply accepting or rejecting.
3. We need to adapt RPNI such that it constructs elastic automata.

We show in [34] that the hypothesis constructed by our ICE-algorithm adapted from RPNI is an EQDA that is consistent with the sample. Hence we have described a consistent learner. For a fixed set of pointer variables and universally quantified variables there are only a finite number of EQDAs, and therefore by Lemma 3.1 we conclude that the above learning is strongly convergent (though number of rounds need not be polynomial).

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

Experiments: Table 2 presents the results of our ICE-learning algorithm for EQDAs on typical programs manipulating arrays. The results show that our algorithm learns the correct quantified invariant (represented as an EQDA) for all these programs, in a reasonable time. We compare our results to SAFARI [3], the state-of-the-art verification tool based on interpolation in array theories. SAFARI, in general, cannot handle list programs, and also array programs like *sorted-find* and *sorted-insert* which have quantified pre-conditions. From the remaining, SAFARI diverges for three programs (marked *div.*) and one probably needs to manually provide a term abstraction list for them to achieve convergence. From the results in Table 2, we conclude that the robustness of ICE seems practically feasible even for synthesizing quantified invariants of arrays. More details about our experiments are present in [34].

	ICE Black-Box						SAFARI White-Box
Program	R	$\lvert\hat{E}\rvert$	$\lvert\hat{C}\rvert$	$\lvert\hat{I}\rvert$	$\lvert Q\rvert$	Time (s)	(s)
compare	9	3	2	5	8	1.3	0.1
copy	4	1	1	1	8	0.7	0.0
copy-lt-key	5	0	2	3	13	1.2	*div.*
init	4	1	1	1	8	0.6	0.7
init-partial	8	1	0	6	12	1.5	*div.*
find	9	2	2	6	8	1.2	0.2
max	3	0	1	1	8	0.4	0.1
increment	5	2	2	1	8	0.7	*div.*
sorted-find	8	4	3	0	17	5.1	–
sorted-insert	6	0	6	0	21	2.0	–
devres [54]	3	0	1	1	8	0.7	0.1
rm_pkey [54]	3	0	1	1	8	0.7	0.3

Table 2: Results for RPNI-based ICE-learning for quantified array invariants. R: number of rounds of iterative-ICE; $\lvert\hat{E}\rvert$, $\lvert\hat{C}\rvert$, and $\lvert\hat{I}\rvert$: number of ex, counter-ex, implications in final sample; $\lvert Q\rvert$: number of states in final EQDA.

7 Natural Proofs for Structure, Data, and Separation

In the previous few sections we described various algorithms to automatically learn loop invariants which together with user written modular contracts and automated theorem proving of the resulting verification conditions leads to a powerful deductive verification paradigm for software verification. Automatic synthesis of loop invariants helps to reduce the annotation burden on the user or increase the automaticity of deductive verification and in this manner helps scale deductive verification to larger software. Verification conditions do not, however, always fall into decidable theories. In particular, the verification of properties of the *dynamically modified heap* is a big challenge for logical methods. The dynamically manipulated heap poses several challenges, as typical correctness properties of heaps require complex combinations of structure (e.g., p points to a tree structure, or to a doubly-linked list, or to an almost balanced tree, with respect to certain pointer-fields), data (the integers stored in data-fields of the tree respect the binary search tree property, or the data stored in a tree is a max-heap), and separation (the procedure modifies one list and not the other and leaves the two lists disjoint at exit, etc.).

As we mentioned in Section 6, the fact that the dynamic heap contains an unbounded number of locations means that expressing the above properties requires *quantification* in some form, which immediately precludes the use of most SMT decidable theories. There are a few known decidable theories that can handle quantification; e.g., the array property fragment [18] and the decidable fragment of Strand logic [60, 62]). In Section 6 we addressed the problem of automatically synthesizing loop invariants for programs whose invariants fall in these decidable logics. In this section we focus on more complex specifications of functional correctness for heap-manipulating programs, which usually always fall outside these decidable logics, and describe completely automatic, sound but incomplete techniques [75] that can prove a large class of these properties/programs correct.

In the Boogie [10] line of tools (including VCC [25]) of writing specifications using first-order logic and employing SMT solvers to validate verification conditions, the specification of invariants of even simple methods like *singly-linked-list insert* is tedious. In such code[2], second-order properties (reachability, acyclicity, separation, etc.) are smuggled in using carefully chosen user-provided *ghost code*. Once such a ghost-encoding of the specification is formulated, the validation of verification conditions, which typically have quantifiers, are dealt with using sound heuristics (a wide variety of them including e-matching, model-based quantifier instantiation, etc. are available), but are still often not enough and have to be augmented by *instantiation triggers* from the verification engineer to help the proof go through. Similarly, in tools [48, 22] based on *separation logic* [76, 72], the validation of verification conditions resulting from separation logic invariants is also complex, and has eluded automatic reasoning and exploitation of SMT solvers and require the user to write low-level lemmas and proof tactics to guide the verification. In contrast, natural proofs [75] exploit a fixed set of proof tactics, giving up on completeness but retaining the automated nature of proving validity, which work for most heap-programs encountered in practice. At a high level, our methodology can thus be viewed as a technique to reduce the burden on the user or equivalently increase the automaticity of automatic deductive verification of complex heap-manipulating programs.

Dryad: A separation logic with determined heaplets. We introduce a new, powerful logic called Dryad [75] to express complex data-structure properties of heap-manipulating programs. Dryad is a dialect of separation logic, and borrows varous salient features from separation logic like *strict specifications*, the spatial conjunction operator, frame reasoning, powerful recursive definitions to express *second order* properties like p points to a tree structure, or that K is the set of keys contained in the tree pointed to by p, etc.

However, a key design feature of Dryad is that the heaplet for recursive formulas is essentially *determined* by the syntax as opposed to the semantics. In classical separation logic, a formula of the form $\alpha * \beta$ says that the heaplet can be partitioned into *any* two disjoint heaplets, one satisfying α and the other β. In Dryad, the heaplet for a (complex) formula is *determined* and hence if there is a way to partition the heaplet, there is precisely *one* way to do so. The key advantage is that this eliminates implicit existential quantification the separation operator provides. In a verification condition that combines the pre-condition in the negative and the post-condition in the positive, the classical semantics for separation logic invariably introduces universal quantification in the satisfiability query for the negation of the verification condition, which in turn is extremely hard to handle.

[2]http://vcc.codeplex.com/SourceControl/changeset/view/dcaa4d0ee8c2#vcc /Docs/Tutorial/c/7.2.list.c

Translating Dryad to classical logic with recursion: The second key step in our paradigm [75] is a technique to bridge the gap from separation logic to classical logic in order to utilize efficient decision procedures supported by SMT solvers. We show that heaplet semantics and separation logic constructs of Dryad can be effectively translated to classical logic where heaplets are modeled as *sets of locations*. This translation does not, of course, yield a decidable theory yet, as recursive definitions are still present (the recursion-free formulas are in a decidable theory). The carefully designed Dryad logic with determined heaplet semantics ensures that there is no quantification in the resulting formula in classical logic.

Natural proofs for Dryad: Finally, we develop a natural proof methodology for Dryad [75] by showing a natural proof mechanism for the equivalent formulas in classical logic. The basic proof tactic unfolds recursive definitions precisely across the footprint touched by the program segment being verified, translates the definitions to the frontier of the footprint, and then uses a form of *formula abstraction* that treats recursive formulas on frontier nodes as uninterpreted functions The resulting formula falls in a logic over sets and integers, which is then decided using the theory of uninterpreted functions and arrays using SMT solvers. These natural proofs for Dryad are an extension of natural proofs for tree data-structures first proposed by Madhusudan et al. in [63].

Experimental Evaluation: Our proof mechanisms are essentially a class of decidable proof tactics that result in sound but incomplete validation procedures. To show their practical effectiveness, we show, using a large class of correct programs manipulating lists, trees, cyclic lists, and doubly linked lists as well as *multiple* data-structures of these kinds, that the natural proof mechanism succeeds in proving the verifications conditions automatically [75]. These programs are drawn from a range of sources, from textbook data-structure routines (binary search trees, red-black trees, etc.) to routines from Glib low-level C-routines used in GTK+/Gnome to implement file-systems, from the Schorr-Waite garbage collection algorithm, to several programs from a recent secure framework developed for mobile applications [65]. Our work is by far the only one that we know of that can handle such a large class of programs, completely automatically.

7.1 Learning Dryad Invariants (Future Work)

Given Dryad annotations in the form of modular pre/post-conditions and loop invariants, the natural proofs methodology can be used to automatically validate verification conditions resulting from the proof of correctness of complex functional properties of heap-manipulating programs. However, writing these Dryad annotations can be difficult for the user, specially loop invariants or pre/post-conditions of internal methods which usually have a more complex boolean structure and often have to describe partial data structures like list segments, tree with holes, etc. Given pre/post-annotations at the level of a module in a data-structure implementation, we would like to develop algorithms to learn Dryad invariants at loop headers and the entry and exit of internal methods so that, coupled with the natural proofs methodology, we can use them to automatically prove correct various data-structure implementation, with respect to their specification.

$$\begin{array}{rrcl} \text{Positive Formula:} & \varphi & ::= & \texttt{true} \mid \texttt{false} \mid q \mid p^{\Delta}(lt) \mid \texttt{emp} \mid lt \longmapsto (\vec{lt}, \vec{vt}) \mid lt \; rop \; lt \mid vt \; rop \; vt \\ & & & \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi * \varphi \\ \text{Formula:} & \psi & ::= & \varphi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi \end{array}$$

Figure 2: Syntax of Dryad

Figure 2 gives the syntax of DRYAD formulas. In the figure, *lt* is a location term and is either a program variable of type *Location* or `nil`; *q* is a program variable of type Boolean; p^{Δ} is a recursively-defined DRYAD predicate; *vt* is a term of type integer, or a set/multi-set of integers or a set of locations and consists of program variables of the appropriate type, constants, recursively defined DRYAD functions and terms formed by binary operators like $+, -$ for integers and $\cup, \cap$ for set of integer/location terms; *rop* is a relational operator, for eg. $=, \leq$, etc., applied on the terms of the appropriate type.

Since DRYAD syntax disallows any quantification, all variables appearing in a DRYAD formula are program variables. Given a finite set of program variables, we can show that the number of DRYAD formulas is finite (modulo an infinite set of terms arising from constants from unbounded domains like integers or set of integers and binary arithmetic or set operations on them). Learning DRYAD formulas can be thus reduced to learning arbitrary Boolean formulas over a finite set of predicates. So, in a passive learning setting like Daikon [31], one can run the program on various test inputs and record the program configurations that manifest themselves in those test runs. CNF or k-CNF learning algorithms [52] can then be used on these recorded program configurations to learn a *likely* program invariant.

Another interesting research direction is to explore ICE-learning algorithms for synthesizing DRYAD invariants. One possible approach is to fix a template of the desired invariants and then reduce the learning problem to constraint solving, as in Section 5 for template-based ICE-learning over numerical domains. The ICE-learner uses a constraint solver to guess the atomic predicates that when plugged into the template holes give an adequate invariant. The constraints evaluate the conjecture on the concrete program configurations present in the sample and assert that the conjecture evaluates to *true* for all the positive examples in the sample, evaluates to *false* for all the negative examples, and satisfies the implication constraint for all implication pairs in the sample.

An efficient learning algorithm for DRYAD formulas would be very powerful. Given specifications in the form of pre/post-conditions, learning DRYAD invariants would help us prove, completely automatically, a wide-range of data structure programs manipulating lists, binary-search trees, avl-trees, etc. This can be potentially very useful, for example in improving online education (MOOCs) where the natural proofs methodology together with learning algorithms for DRYAD formulas can be used to automatically grade student programming assignments implementing various data-structures, in an algorithms course.

8 Concurrency

8.1 Invariant Synthesis for Shared-Memory Concurrent Programs (Current Work)

The verification problem for concurrent programs with procedures (even with a finite number of Boolean variables) is undecidable [66]. One of the ways of proving concurrent programs is by constructing *compositional proofs*, pioneered by Owicki and Gries [74] and Jones [51]. By definition of compositional proofs, a thread cannot refer to the local state or the stack contents of another thread. To achieve completeness (in the case of finite state programs with no recursion), these proof systems have a rule to introduce *auxiliary variables*. Auxiliary variables can be seen as local states that get exposed in order to capture the correlation between the local states of the concurrently executing threads.

Given a concurrent progam P and a set of auxiliary variables A, we defined in [36] a *non-standard compositional semantics*, different from the traditional interleaving semantics, for P with respect to A. The compositional semantics captures when a concurrent program can be argued compositionally to satisfy a specification, by only exposing the auxiliary variables in A. In fact in [36], we presented an effective sequentialization which takes a concurrent program C (with specifications) and a set of auxiliary variables

A and gives a sequential program $S_{C,A}$ (with specifications) which precisely captures the compositional semantics of C with respect to A. Since verifying a concurrent program with respect to its interleaving semantics is undecidable, the idea is to verify an over-approximation/abstraction of it (leading to a sound, but an incomplete analysis much like natural proofs in Section 7). The compositional semantics of a concurrent program with respect to a set of auxiliary variables A presents a nice, computable over-approximation of its precise interleaving semantics [36]. Hence given a concurrent program C, proving $S_{C,A}$ presents a way of proving C with respect to its specifications.

The extent to which $S_{C,A}$ over-approximates the interleaving semantics of C can be controled by A. In fact, for concurrent programs that are "loosely coupled', exposing only a small number of auxiliary variables is often precise enough to prove the concurrent program correct. We fix A to be the local state that needs to be exposed to reason compositionally about the generic communication patterns present in the concurrent program. It includes a *history variable* that captures the thread ID of the last thread that acquired a given lock; the program counters of the threads at each flag-wait synchronization; and a counter which statically tracks the number of instances of each procedure that may run in parallel with a given event. Finally, once A has been fixed, we learn compositional rely-guarantee invariants for concurrent programs C by using monotonic learning frameworks, we introduced in Section 4, to instead learn invariants for the sequential program $S_{C,A}$.

The sequentialization in [36] introduced recursion and was not very amenable to symbolic execution. We have thus developed a new sequentialization which introduces no recursion and is more amenable to symbolic reasoning. We use this new sequentialization and the above sketched methodology to learn invariants over the interval and the pentagon [58] abstract domains for Linux device drivers. Invariants over these domains and the choice of the auxiliary variables we mentioned above suffices to prove the absence of any array index out of bounds errors in these device drivers. We are also currently applying the above methodology to successfully prove data-race freedom in a suite of Windows device drivers. We hope to argue, through experiments, that static analysis can be more efficient as opposed to under-approximate testing techniques like context-bounded exploration [55, 56, 70], for proving simple properties in concurrent programs (the analysis requires only a small set of auxiliary variables which can be manually set or statically mined). Further, compositional reasoning presents a nice, computable abstraction for the static analysis of concurrent programs. And finally, we hope to show that we can use monotonic learning frameworks even in the context of concurrent programs to learn more complex rely-guarantee invariants.

8.2 Natural Proofs for Asynchronous Concurrent Systems (Current Work)

Asynchronous programming or *event-driven programming* or the *actor programming model* is a common idiom provided today by most parallel programming languages. This programming model is used as a natural model for asynchronous systems on a multi-core computer or distributed systems communicating on a network. Such systems are usually modeled, using state machine notations, as network of concurrent processes communicating via message queues. Communication via FIFO message queues forms a very natural subclass of communication in such systems. Motivated by the verification of such systems, the model checking problem for such systems has been extensively studied [2, 19, 46, 84]. It is known that the model checking problem, even for a two-process finite state system communicating via FIFO message queues of unbounded size, is undecidable. In light of this we develop *natural proofs* for these asynchronous systems, which is a sound exploration strategy which is terminating and hence complete for most asynchronous systems encountered in practice (the strategized exploration might not terminate and is therefore incomplete in general).

P is a domain-specific language recently developed at Microsoft Research for writing asynchronous event-driven code [29]. It was designed for modeling and testing Windows device drivers and has been used to test the core of the USB device driver that ships with Windows 8. P has since been also used to model other asynchronously communicating distributed systems. The P language is designed so that a P program can be checked for resposiveness– the ability to handle every event in a timely manner [29]. Resposiveness of a system is an important safety specification in these device drivers as well as in general distributed protocols. Theoretically, the problem of checking the responsiveness of a P program is equivalent to checking violations of local assertions in a finite state system (with push-down stacks) communicating via FIFO message queueus. In this work we develop natural proofs for checking responsiveness of a system.

At a high level, from every reachable configuration of the system, our natural proofs exploration strategy choses a subset of the enabled actions, depending on the current configuration of the system and the statically available global communication topology of the system, such that (1) the chosen actions/transitions are sufficient to discover non-responsive states in the system if any (soundness), and (2) the exploration strategy prioritizes actions such that the size of the message queueus are bounded (for most systems) leading to termination of the exploration (practical effectiveness).

In a given configuration, a system often has more than one enabled *send* action. A traditional model checker has to consider all those enabled actions in a non-deterministic manner. If a system continously enqueues messages into a queue without recieving or dequeuing them, then the model checking process might not terminate. In contrast, of all the enabled send actions, we prioritize and consider only those actions that enqueue messages that can be immediately recieved, i.e., actions that enqueue only those messages that another process is ready to recieve or is waiting for. This exploration strategy ensures that the message queues do not keep increasing in size. The strategy is based on our observation that in most asynchronous systems, the mode of communication is mostly synchronous and asynchrony is usually present for performance reasons. Of course, asynchronous communication can lead to new behaviors which one would skip if one restricts oneselves to purely synchronous communication. Our exploration strategy dynamically chooses a subset of actions that need to be scheduled from every configuration and can soundly handle even such asynchrony in the system. We develop sound proof rules that show that the subset of actions we schedule from every configuration suffices to check the non-responsiveness of the given system.

We have developed the theory of our natural proofs methodology and have used it to successfully prove responsiveness of an OSR driver that ships with Windows phone and a model of the elevator distributed system. Our technique is sound and can be used for verifying asynchronous systems, with respect to local assertions (or responsiveness), as opposed to finding bugs. From our initial experiments, we observe that our exploration strategy is significantly more scalable as compared to the exisiting state-of-the-art, incomplete search strategies like depth-first search, breadth-first search, etc. We plan to further evaluate our technique on several other Windows device driver models and asynchronous distributed protocols.

References

[1] Competition on Software Verification (SV-COMP) benchmarks. https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp13/loops/.

[2] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *LICS*, pages 160–170. IEEE Computer Society, 1993.

[3] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Safari: Smt-based abstraction for arrays with interpolants. In *CAV*, volume 7358 of *LNCS*. Springer, 2012.

[4] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109. ACM, 2005.

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

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

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

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

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

[10] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.

[11] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV'07*, volume 4590 of *LNCS*, pages 178–192. Springer, 2007.

[12] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO'05*, volume 4111 of *LNCS*, pages 115–137. Springer, 2005.

[13] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS'05*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.

[14] Matko Botinčan, Matthew Parkinson, and Wolfram Schulte. Separation logic verification of C programs with an SMT solver. *ENTCS*, 254:5 – 23, 2009.

[15] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI*, 2012.

[16] Aaron R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[17] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[18] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *VMCAI'06*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[19] Rohit Chadha and Mahesh Viswanathan. Decidability results for well-structured transition systems with auxiliary storage. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 136–150. Springer, 2007.

[20] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *POPL'08*, pages 247–260. ACM, 2008.

[21] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Science of Computer Programming*, 77(9):1006 – 1036, 2012.

[22] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI'11*, pages 234–245. ACM, 2011.

[23] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, volume 2619 of *LNCS*, pages 331–346. Springer, 2003.

[24] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, pages 331–346, 2003.

[25] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.

[26] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, volume 2725 of *LNCS*, pages 420–432. Springer, 2003.

[27] Patrick Cousot and Radhia 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.

[28] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118. ACM, 2011.

[29] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 321–332, New York, NY, USA, 2013. ACM.

[30] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *TACAS'06*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.

[31] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE*, pages 449–458. ACM, 2000.

[32] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.

[33] Robert 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.

[34] P. Garg, C. Loding, P. Madhusudan, and D. Neider. ICE: A Robust Learning Framework for Synthesizing Invariants. Technical report, University of Illinois at Urbana-Champaign, Oct 2013.

[35] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Learning universally quantified invariants of linear data structures. In *CAV*, pages 813–829, 2013.

[36] Pranav Garg and P. Madhusudan. Compositionality entails sequentializability. In *TACAS*, pages 26–40, 2011.

[37] Pranav Garg, P. Madhusudan, and Gennaro Parlato. Quantified data automata on skinny trees: An abstract domain for lists. In *SAS*, pages 172–193, 2013.

[38] Pierre-Loïc Garoche, Temesghen Kahsai, and Cesare Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In *NASA Formal Methods*, pages 139–154, 2013.

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

[40] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127. ACM, 2006.

[41] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.

[42] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292. ACM, 2008.

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

[44] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.

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

[46] Oscar H. Ibarra, Zhe Dang, and Pierluigi San Pietro. Verification in loosely synchronous queue-connected discrete timed automata. *Theor. Comput. Sci.*, 290(3):1713–1735, January 2003.

[47] Franjo Ivancic and Sriram Sankaranarayanan. NECLA Benchmarks. `http://www.nec-labs.com/research/system/systems_SAV-website/small_static_bench-v1.1.tar.gz`.

[48] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM'11*, volume 6617 of *LNCS*, pages 41–55. Springer, 2011.

[49] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.

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

[51] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[52] Michael J. Kearns and Umesh V. Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, USA, 1994.

[53] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*, pages 207–220. ACM, 2009.

[54] Soonho Kong, Yungbum Jung, Cristina David, Bow-Yaw Wang, and Kwangkeun Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In *APLAS*. Springer, 2010.

[55] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. Static and precise detection of concurrency errors in systems code using smt solvers. In *CAV*, pages 509–524, 2009.

[56] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In *CAV*, pages 427–443, 2012.

[57] Dirk Leinenbach and Thomas Santen. Verifying the microsoft hyper-v hypervisor with vcc. In *Proceedings of the 2Nd World Congress on Formal Methods*, FM '09, pages 806–809, Berlin, Heidelberg, 2009. Springer-Verlag.

[58] Francesco Logozzo and Manuel Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.*, 75(9):796–807, 2010.

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

[60] P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable logics combining heap structures and data. In *POPL*, pages 611–622, 2011.

[61] P. Madhusudan and Xiaokang Qiu. Efficient decision procedures for heaps using STRAND. In *SAS*, volume 6887 of *LNCS*, pages 43–59. Springer, 2011.

[62] P. Madhusudan and Xiaokang Qiu. Efficient decision procedures for heaps using STRAND. In *SAS'11*, volume 6887 of *LNCS*, pages 43–59. Springer, 2011.

[63] P. Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive proofs for inductive tree datastructures. In *POPL'12*, pages 123–136. ACM, 2012.

[64] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. THOR: A tool for reasoning about shape and arithmetic. In *CAV'08*, volume 5123 of *LNCS*, pages 428–432. Springer, 2008.

[65] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in ExpressOS. In *ASPLOS*, pages 293–304, 2013.

[66] Roman Manevich, Eran Yahav, Ganesan Ramalingam, and Shmuel Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*, volume 3385 of *LNCS*, pages 181–198. Springer, 2005.

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

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

[69] Tom M. Mitchell. *Machine learning*. McGraw-Hill, 1997.

[70] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multi-threaded programs. In *PLDI*, pages 446–455, 2007.

[71] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[72] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL'01*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[73] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. *Pattern Recognition and Image Analysis*, pages 49–61, 1992.

[74] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[75] Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Parthasarathy Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242, 2013.

[76] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE-CS, 2002.

[77] Mohamed Nassim Seghir, Andreas Podelski, and Thomas Wies. Abstraction refinement for quantified array assertions. In *SAS*, 2009.

[78] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, volume 7792 of *LNCS*, pages 574–592. Springer, 2013.

[79] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. Verification as learning geometric concepts. In *SAS*, volume 7935 of *LNCS*, pages 388–411. Springer, 2013.

[80] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants as classifiers. In *CAV*, volume 7358 of *LNCS*, pages 71–87. Springer, 2012.

[81] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL'10*, pages 199–210. ACM, 2010.

[82] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS'11*, volume 6887 of *LNCS*, pages 298–315. Springer, 2011.

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

[84] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Context-bounded analysis of concurrent queue systems. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2008.

[85] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *CAV'08*, volume 5123 of *LNCS*, pages 385–398. Springer, 2008.

[86] Greta Yorsh, Thomas Ball, and Mooly Sagiv. Testing, abstraction, theorem proving: better together! In *ISSTA*, pages 145–156, 2006.