# Learning Universally Quantified Invariants of Linear Data Structures

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

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

**Abstract.** We propose a new automaton model, called *quantified data automata* over words, that can model quantified invariants over linear data structures, and build poly-time active learning algorithms for them, where the learner is allowed to query the teacher with membership and equivalence queries. In order to express invariants in decidable logics, we invent a decidable subclass of QDAs, called elastic QDAs, and prove that every QDA has a unique minimally-over-approximating elastic QDA. We then give an application of these theoretically sound and efficient active learning algorithms in a passive learning framework and show that we can efficiently learn quantified linear data structure invariants from samples obtained from dynamic runs for a large class of programs.

## 1 Introduction

Synthesizing invariants for programs is one of the most challenging problems in verification today. In this paper, we are interested in using *learning* techniques to synthesize quantified data-structure invariants.

In an *active* black-box learning framework, we look upon the invariant as a set of configurations of the program, and allow the learner to query the teacher for membership and equivalence queries on this set. Furthermore, we fix a particular representation class for these sets, and demand that the learner learn the smallest (simplest) representation that describes the set. A learning algorithm that learns in time polynomial in the size of the simplest representation of the set is desirable. In *passive* black-box learning, the learner is given a sample of examples and counter-examples of configurations, and is asked to synthesize the simplest representation that includes the examples and excludes the counter-examples. In general, several active learning algorithms that work in polynomial time are known (e.g., learning regular languages represented as DFAs [1]) while passive polynomial-time learning is rare (e.g., conjunctive Boolean formulas can be learned but general Boolean formulas cannot be learned efficiently, automata cannot be learned passively efficiently) [2].

In this paper, we build active learning algorithms for *quantified logical formulas describing sets of linear data-structures.* Our aim is to build algorithms that can learn formulas of the kind "$\forall y_1, \ldots, y_k\ \varphi$", where $\varphi$ 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.). Furthermore, we show that we can build learning algorithms that learn properties that are expressible in known decidable logics. We then employ the active learning algorithm in a *passive learning* setting where we show that by building an imprecise teacher that answers the questions of the active learner, we can build effective invariant generation algorithms that learn simply from a finite set of examples.

**Active Learning of Quantified Properties Using QDAs:** Our first technical contribution is a novel representation (normal form) for quantified properties of linear data-structures, called *quantified data automata* (QDA), and a polynomial-time active learning algorithm for QDAs.

We model linear data-structures as *data words*, where each position is decorated with a letter from a finite alphabet modeling the program's pointer variables that point to that cell in the list or index variables that index into the cell of the array, and with data modeling the data value stored in the cell, e.g., integers. Quantified data automata (QDA) are a new model of automata over data words that are powerful enough to express *universally* quantified properties of data words. A QDA accepts a data word provided it accepts *all possible* annotations of the data word with valuations of a (fixed) set of variables $Y = \{y_1, \ldots, y_k\}$; for each such annotation, the QDA reads the data word, records the data stored at the positions pointed to by $Y$, and finally checks these data values against a data formula determined by the final state reached. QDAs are very powerful in expressing typical invariants of programs manipulating lists and arrays, including invariants of a wide variety of searching and sorting algorithms, maintenance of lists and arrays using insertions/deletions, in-place manipulations that destructively update lists, etc.

We develop an efficient active learning algorithm for QDAs. By using a combination of *abstraction* over a set of data formulas and Angluin's learning algorithm for DFAs [1], we build a learning algorithm for QDAs. We first show that for any set of valuation words (data words with valuations for the variables $Y$), there is a *canonical* QDA. Using this result, we show that learning valuation words can be reduced to learning *formula words* (words with no data but paired with data formulas), which in turn can be achieved using Angluin-style learning of Moore machines. The number of queries the learner poses and the time it takes is bound polynomially in the size of the canonical QDA that is learned. Intuitively, given a set of pointers into linear data structures, there is an exponential number of ways to permute the pointers into these and the universally quantified variables; the learning algorithm allows us to search this space using only polynomial time in terms of the actual permutations that figure in the set of data words learned.

**Elastic QDAs and a Unique Minimal Over-Approximation Theorem:** The class of quantified properties that we learn in this paper (we can synthesize them from QDAs) is very powerful. 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. The goal of this paper is hence to also offer mechanisms to *learn invariants that are amenable to decision procedures.*

The second technical contribution of this paper is to identify a subclass of QDAs (called elastic QDAs) and show two main results for them: (a) elastic QDAs can be converted to formulas of *decidable* logics, to the array property fragment when modeling arrays and the decidable STRAND fragment when modeling lists; (b) 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). The result is proved by showing that there is a unique way to minimally morph a QDA to one that satisfies the elasticity restrictions. For the former, we identify a common property of the array property fragment and the syntactic decidable fragment of STRAND, called *elasticity* (following the general terminology in the literature on STRAND [3]). Intuitively, both the array property fragment and STRAND prohibit quantified cells to be tested to be bounded distance away (the array property fragment does this by disallowing arithmetic expressions over the quantified index variables [4] and the decidable fragment of STRAND disallows this by permitting only the use of $\rightarrow^*$ or $\rightarrow^+$ in order to compare quantified variables [3,5]). We finally identify a *structural restriction* of QDAs that permits only elastic properties to be stated.

**Passive Learning of Quantified Properties:** 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). In this paper, 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 [6], compositional verification [7,8], 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. 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.

**Related Work:** For invariants expressing properties on the dynamic heap, *shape analysis* techniques are the most well known [9], where locations are classified/merged using *unary* predicates (some dictated by the program and some given as instrumentation predicates by the user), and abstractions summarize all nodes with the same predicates into a single node. The data automata that we build also express an infinite set of linear data structures, but do so using automata, and further allow $n$-ary quantified relations between data elements. In recent work, [10] describes an abstract domain for analyzing list manipulating programs, that can capture quantified properties about the structure and the data stored in lists. This domain can be instantiated with any numerical domain for the data constraints and a set of user-provided patterns for capturing the structural constraints. However, providing these patterns for quantified invariants is in general a difficult task.

In recent years, techniques based on *Craig's interpolation* [11] have emerged as a new method for invariant synthesis. Interpolation techniques, which are inherently white-box, are known for several theories, including linear arithmetic, uninterpreted function theories, and even quantified properties over arrays and lists [12,13,14,15]. These methods use different heuristics like term abstraction [14], preferring smaller constants [12,13] and use of existential ghost variables [15] to ensure that the interpolant converges on an invariant from a *finite* set of spurious counter-examples. IC3 [16] is another white-box technique for generalizing inductive invariants from a set of counter-examples.

A primary difference in our work, compared to all the work above, is that ours is a *black-box technique* that does not look at the code of the program, but synthesizes an invariant from a snapshot of examples and counter-examples that characterize the invariant. The black-box approach to constructing invariants has both advantages and disadvantages. The main disadvantage is that information regarding what the program actually does is lost in invariant synthesis. However, this is the basis for its advantage as well—by *not* looking at the code, the learning algorithm promises to learn the sets with the simplest representations in polynomial time, and can also be much more flexible. For instance, even when the code of the program is complex, for example having non-linear arithmetic or complex heap manipulations that preclude logical reasoning, black-box learning gives ways to learn simple invariants for them.

There are several black-box learning algorithms that have been explored in verification. Boolean formula learning has been investigated for finding quantifier-free program invariants [17], and also extended to quantified invariants [6]. However, unlike us, [6] learns a quantified formula given a set of data predicates as well as the predicates which can appear in the guards of the quantified formula. Recently, machine learning techniques have also been explored [18]. Variants of the Houdini algorithm [19] essentially use conjunctive Boolean learning (which can be achieved in polynomial time) to learn conjunctive invariants over templates of atomic formulas (see also [20]). The most mature work in this area is Daikon [21], which learns formulas over a template, by enumerating all formulas and checking which ones satisfy the samples, and where scalability is achieved in practice using several heuristics that reduce the enumeration space which is doubly-exponential. For quantified invariants over data-structures, however, such heuristics aren't very effective, and Daikon often restricts learning only to formulas of very restricted syntax, like formulas with a single atomic guard, etc. In our experiments Daikon was, for instance, not able to learn an adequate loop invariant for the selection sort algorithm.

## 2 Overview

**List and Array Invariants:** Consider a typical invariant in a sorting program over lists where the loop invariant is expressed as:

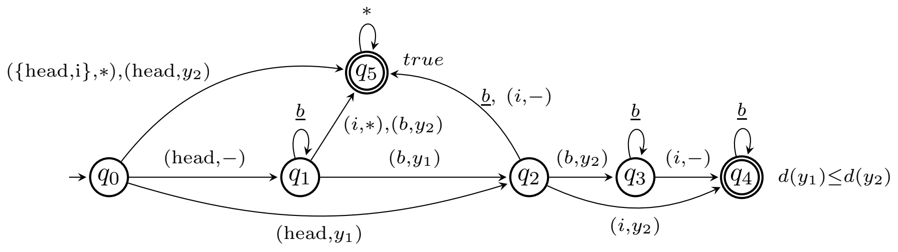
$$head \rightarrow^* i \ \wedge \ \forall y_1, y_2.((head \rightarrow^* y_1 \wedge succ(y_1, y_2) \wedge y_2 \rightarrow^* i) \Rightarrow d(y_1) \leq d(y_2)) \quad (1)$$

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 $y_2$. This formula is *not* in the decidable fragment of Strand [3,5] since the universally quantified variables are involved in a non-elastic relation *succ* (in the subformula $succ(y_1, y_2)$). Such an invariant for a program manipulating arrays can be expressed as:

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

Note that the above formula is not in the decidable array property fragment [4].

**Quantified Data Automata:** The key idea in this paper is an automaton model for expressing such constraints called *quantified data automata* (QDA). The above two invariants are expressed by the following QDA:



The above automaton reads (deterministically) data words whose labels denote the positions pointed to by the scalar pointer variables *head* and $i$, as well as valuations of the quantified variables $y_1$ and $y_2$. 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.

**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. A common restriction in the array property fragment as well as the syntactic decidable fragments of Strand 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 [3,5]).

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

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

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:

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

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 identify a restricted form of quantified data automata, called *elastic quantified data automata* (EQDA) in Section 5, which structurally captures the constraint that quantified variables can be related only using elastic relations (like $\rightarrow^*$ and $\leq$). Furthermore, we show in Section 6 that EQDAs can be converted to formulas in the decidable fragment of STRAND and the array property fragment, and hence expresses invariants that are amenable to decidable analysis across loop bodies.

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).

## 3 Quantified Data Automata

We model lists (and finite sets of lists) and arrays that contain data over some data domain $D$ as finite words, called *data words*, encoding the pointer variables and the data values. Consider a finite set of pointer variables $PV = \{p_1, \ldots, p_r\}$ and let $\Sigma = 2^{PV}$. The empty set corresponds to a blank symbol indicating that no pointer variable occurs at this position. We also denote this blank symbol by the letter $b$. A data word over $PV$ and the data domain $D$ is an element $w$ of $(\Sigma \times D)^*$, such that every $p \in PV$ occurs exactly once in the word (i.e., for each $p \in PV$, there is precisely one $j$ such that $w[j] = (X, d)$, with $p \in X$).

Let us fix a set of variables $Y$. The automata we build accept a data word if for all possible valuations of $Y$ over the positions of the data word, the data stored at these positions satisfy certain properties. For this purpose, the automaton reads data words extended by valuations of the variables in $Y$, called valuation words. The variables are then quantified universally in the semantics of the automaton model (as explained later in this section).

A valuation word is a word $v \in (\Sigma \times (Y \cup \{-\}) \times D)^*$, where $v$ projected to the first and third components forms a data word and where each $y \in Y$ occurs in the second component of a letter precisely once in the word. The symbol '$-$' is used for the positions at which no variable from $Y$ occurs. A valuation word hence defines a data word along with a valuation of $Y$. The data word corresponding to such a word $v$ is the word in $(\Sigma \times D)^*$ obtained by projecting it to its first and third components. Note that the choice of the alphabet enforces the variables from $Y$ to be in different positions.

To express the properties on the data, we fix a set of constants, functions and relations over $D$. We assume that the quantifier-free first-order theory over this domain is decidable. We encourage the reader to keep in mind the theory of integers with constants (0, 1, etc.), addition, and the usual relations ($\leq$, $<$, etc.) as a standard example of such a domain.

Quantified data automata use a *finite* set $F$ of formulas over the atoms $d(y_1), \ldots, d(y_n)$ that is additionally equipped with a (semi-)lattice structure of the form $\mathcal{F} : (F, \sqsubseteq, \sqcup, \mathit{false}, \mathit{true})$ where $\sqsubseteq$ is the partial-order relation, $\sqcup$ is the least-upper bound, and *false* and *true* are formulas required to be in $F$ and correspond to the bottom and top elements of the lattice. Furthermore, we assume that whenever $\alpha \sqsubseteq \beta$, then $\alpha \Rightarrow \beta$. Also, we assume that each pair of formulas in the lattice are *inequivalent*.

One example of such a formula lattice over the data domain of integers can be obtained by taking a set of representatives of all possible inequivalent Boolean formulas over the atomic formulas involving no constants, defining $\alpha \sqsubseteq \beta$ iff $\alpha \Rightarrow \beta$, and taking the least-upper bound of two formulas as the disjunction of them. Such a lattice would be of size doubly exponential in the number of variables $n$, and consequently, in practice, we may want to use a different coarser lattice, such as the Cartesian formula lattice. The Cartesian formula lattice is formed over a set of atomic formulas and consists of conjunctions of literals (atoms or negations of atoms). The least-upper bound of two formulas is taken as the conjunction of those literals that occur in both formulas. For the ordering we define $\alpha \sqsubseteq \beta$ if all literals appearing in $\beta$ also appear in $\alpha$. The size of a Cartesian lattice is exponential in the number of literals.

We are now ready to introduce the automaton model. A quantified data automaton (QDA) over a set of program variables $PV$, a data domain $D$, a set of universally quantified variables $Y$, and a formula lattice $\mathcal{F}$ is of the form $\mathcal{A} = (Q, q_0, \Pi, \delta, f)$ where $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $\Pi = \Sigma \times (Y \cup \{-\})$, $\delta : Q \times \Pi \rightarrow Q$ is the transition function, and $f : Q \rightarrow F$ is a *final-evaluation function* that maps each state to a data formula. The alphabet $\Pi$ used in a QDA does not contain data. Words over $\Pi$ are referred to as *symbolic words* because they do not contain concrete data values. The symbol $(b, -)$ indicating that a position does not contain any variable is denoted by $\underline{b}$.

Intuitively, a QDA is a *register* automaton that reads the data word extended by a valuation that has a register for each $y \in Y$, which stores the data stored at the positions evaluated for $Y$, and checks whether the formula decorating the final state reached holds for these registers. It accepts a data word $w \in (\Sigma \times D)^*$ if it accepts *all possible* valuation words $v$ extending $w$ with a valuation over $Y$.

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

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

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

## 4 Learning Quantified Data Automata

Our goal in this section is to synthesize QDAs using existing learning algorithms such as Angluin's algorithm [1], which was developed to infer the canonical deterministic automaton for a regular language. We achieve this by relating QDAs to the classical model of Moore machines (an automaton with output on states). Recall that QDAs define two kinds of languages, a language of data words and a language of valuation words. On the level of valuation words, we can view a QDA as a device mapping a symbolic word to a data formula as formalized below.

A *formula word* over $PV$, $\mathcal{F}$, and $Y$ is an element of $(\Pi^* \times \mathcal{F})$ where, as before, $\Pi = \Sigma \times (Y \cup \{-\})$ and each $p \in PV$ and $y \in Y$ occurs exactly once in the word. Note that a formula word does not contain elements of the data domain—it simply consists of the symbolic word that depicts the pointers into the list (modeled using $\Sigma$) and a valuation for the quantified variables in $Y$ (modeled using the second component) as well as a formula over the lattice $\mathcal{F}$. For example, $\big((\{h\}, y_1)(b, -)(b, y_2)(\{t\}, -), d(y_1) \leq d(y_2)\big)$ is a formula word, where $h$ points to the first element, $t$ to the last element, $y_1$ points to the first element, and $y_2$ to the third element; and the data formula is $d(y_1) \leq d(y_2)$.

By using formula words we explicitly take the view of a QDA as a Moore machine that reads symbolic words and outputs data formulas. A formula word $(u, \alpha)$ is accepted by a QDA $\mathcal{A}$ if $\mathcal{A}$ reaches the state $q$ after reading $u$ and $f(q) = \alpha$. Hence, a QDA defines a unique language of formula words. One easily observes that two QDAs $\mathcal{A}$ and $\mathcal{A}'$ (over the same lattice of formulas) that accept the same set of valuation words also define the same set of formula words [22] (assuming that all the formulas in the lattice are pairwise non-equivalent).

Thus, a language of valuation words can be seen as a function that assigns to each symbolic word a uniquely determined formula, and a QDA can be viewed as a Moore machine that computes this function. For each such Moore machine there exists a unique minimal one that computes the same function, hence we obtain the following theorem.

**Theorem 1.** *For each QDA $\mathcal{A}$ there is a unique minimal QDA $\mathcal{A}'$ that accepts the same set of valuation words.*

Angluin [1] introduced a popular learning framework in which a *learner* learns a regular language $L$, the so-called *target language*, over an a priory fixed alphabet $\Sigma$ by actively querying a *teacher* which is capable of answering *membership* and *equivalence queries*. Angluin's algorithm learns a regular language in time polynomial in the size of the (unique) minimal deterministic finite automaton accepting the target language and the length of the longest counterexample returned by the teacher.

This algorithm can be easily lifted to the learning of Moore machines. Membership queries now ask for the output or classification of a word. On an equivalence query, the teacher says "yes" or returns a counter-example $w$ such that the output of the conjecture on $w$ is different from the output on $w$ in the target language. Viewing QDAs as Moore machines, we can apply Angluin's algorithm directly in order to learn a QDA, and obtain the following theorem.

**Theorem 2.** *Given a teacher for a QDA-acceptable language of formula words that can answer membership and equivalence queries, the unique minimal QDA for this language can be learned in time polynomial in this minimal QDA and the length of the longest counterexample returned by the teacher.*

## 5 Unique Over-approximation Using Elastic QDAs

Our aim is to translate the QDAs that are synthesized into decidable logics such as the decidable fragment of STRAND or the array property fragment. A property shared by both logics is that they cannot test whether two universally quantified variables are bounded distance away. We capture this type of constraint by the subclass of *elastic QDAs (EQDAs)* that have been already informally described in Section 2. Formally, a QDA $\mathcal{A}$ is called *elastic* if each transition on $\underline{b}$ is a self loop, that is, whenever $\delta(q, \underline{b}) = q'$ is defined, then $q = q'$.

The learning algorithm that we use to synthesize QDAs does not construct EQDAs in general. However, we can show that every QDA $\mathcal{A}$ can be *uniquely over-approximated* by a language of valuation words that can be accepted by an EQDA $\mathcal{A}_{\mathrm{el}}$. We will refer to this construction, which we outline below, as *elastification.* This construction crucially relies on the particular structure that elastic automata have, which forces a unique set of words to be added to the language in order to make it elastic.

Let $\mathcal{A} = (Q, q_0, \Pi, \delta, f)$ be a QDA and for a state $q$ let $R_{\underline{b}}(q) := \{q' \mid q \xrightarrow{\underline{b}}{}^{*} q'\}$ be the set of states reachable from $q$ by a (possibly empty) sequence of $\underline{b}$-transitions. For a set $S \subseteq Q$ we let $R_{\underline{b}}(S) := \bigcup_{q \in S} R_{\underline{b}}(q)$.

The set of states of $\mathcal{A}_{\mathrm{el}}$ consists of sets of states of $\mathcal{A}$ that are reachable from the initial state $R_{\underline{b}}(q_0)$ of $\mathcal{A}_{\mathrm{el}}$ by the following transition function (where $\delta(S, a)$ denotes the standard extension of the transition function of $\mathcal{A}$ to sets of states):

$$\delta_{\mathrm{el}}(S, a) = \begin{cases} R_{\underline{b}}(\delta(S, a)) & \text{if } a \neq \underline{b} \\ S & \text{if } a = \underline{b} \text{ and } \delta(q, \underline{b}) \text{ is defined for some } q \in S \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Note that this construction is similar to the usual powerset construction except that in each step we take the $\underline{b}$-closure after applying the transition function of $\mathcal{A}$. If the input letter is $\underline{b}$, $\mathcal{A}_{\text{el}}$ loops on the current set if a $\underline{b}$-transition is defined for some state in the set.

The final evaluation formula for a set is the least upper bound of the formulas for the states in the set: $f_{\text{el}}(S) = \bigsqcup_{q \in S} f(q)$. We can now show that $L_{\text{v}}(\mathcal{A}_{\text{el}})$ is the *most precise elastic over-approximation* of $L_{\text{v}}(\mathcal{A})$.

**Theorem 3.** *For every QDA $\mathcal{A}$, the EQDA $\mathcal{A}_{el}$ satisfies $L_v(\mathcal{A}) \subseteq L_v(\mathcal{A}_{el})$, and for every EQDA $\mathcal{B}$ such that $L_v(\mathcal{A}) \subseteq L_v(\mathcal{B})$, $L_v(\mathcal{A}_{el}) \subseteq L_v(\mathcal{B})$ holds.*

*Proof:* Note that $\mathcal{A}_{\text{el}}$ is elastic by definition of $\delta_{\text{el}}$. It is also clear that $L_{\text{v}}(\mathcal{A}) \subseteq L_{\text{v}}(\mathcal{A}_{\text{el}})$ because for each run of $\mathcal{A}$ using states $q_0 \cdots q_n$ the run of $\mathcal{A}_{\text{el}}$ on the same input uses sets $S_0 \cdots S_n$ such that $q_i \in S_i$, and by definition $f(q_n)$ implies $f_{\text{el}}(S_n)$.

Now let $\mathcal{B}$ be an EQDA with $L_{\text{v}}(\mathcal{A}) \subseteq L_{\text{v}}(\mathcal{B})$. Let $w = (a_1, d_1) \cdots (a_n, d_n) \in L_{\text{v}}(\mathcal{A}_{\text{el}})$ and let $S$ be the state of $\mathcal{A}_{\text{el}}$ reached on $w$. We want to show that $w \in L_{\text{v}}(\mathcal{B})$. Let $p$ be the state reached in $\mathcal{B}$ on $w$. We show that $f(q)$ implies $f_{\mathcal{B}}(p)$ for each $q \in S$. From this we obtain $f_{\text{el}}(S) \Rightarrow f_{\mathcal{B}}(p)$ because $f_{\text{el}}(S)$ is the least formula that is implied by all the $f(q)$, for $q \in S$.

Pick some state $q \in S$. By definition of $\delta_{\text{el}}$ we can construct a valuation word $w' \in L_{\text{v}}(\mathcal{A})$ that leads to the state $q$ in $\mathcal{A}$ and has the following property: if all letters of the form $(\underline{b}, d)$ are removed from $w$ and from $w'$, then the two remaining words have the same symbolic words. In other words, $w$ and $w'$ can be obtained from each other by inserting and/or removing $\underline{b}$-letters.

Since $\mathcal{B}$ is elastic, $w'$ also leads to $p$ in $\mathcal{B}$. From this we can conclude that $f(q) \Rightarrow f_{\mathcal{B}}(p)$ because otherwise there would be a model of $f(q)$ that is not a model of $f_{\mathcal{B}}(p)$ and by changing the data values in $w'$ accordingly we could produce an input that is accepted by $\mathcal{A}$ and not by $\mathcal{B}$. $\square$

## 6 Linear Data-Structures to Words and EQDAs to Logics

In this section, we sketch briefly how to model arrays and lists as data words, and how to convert EQDAs to quantified logical formulas in decidable logics.

**Modeling Lists and Arrays as Data Words:** We model a linear data structure as a word over $(\Sigma \times D)$ where $\Sigma = 2^{PV}$, $PV$ is the set of pointer variables and $D$ is the data domain; scalar variables in the program are modeled as single element lists. The encoding introduces a special pointer variable *nil* which is always read together with all other null-pointers in the configuration. For arrays, the encoding introduces variables *le_zero* and *geq_size* which are read together with all those index variables which are less than zero or which exceed the size of the respective array. Given a configuration, the corresponding data words read the scalar variables and the linear data structures one after the other, in some pre-determined order. In programs like copying one array to another, where both the arrays are read synchronously, the encoding models multiple data structures as a single structure over an extended data domain.

**From EQDAs to STRAND and Array Property Fragment (APF):** Now we briefly sketch the translation from an EQDA $\mathcal{A}$ to an equivalent formula $\mathcal{T}(\mathcal{A})$ in STRAND or the APF such that the set of data words accepted by $\mathcal{A}$ corresponds to the program configurations $\mathcal{C}$ which model $\mathcal{T}(\mathcal{A})$.

Given an EQDA $\mathcal{A}$, the translation enumerates all simple paths in the automaton to an output state. For each such path $p$ from the initial state to an output state $q_p$, the translation records the relative positions of the pointer and universal variables as a structural constraint $\phi_p$, and the formula $f_{\mathcal{A}}(q_p)$ relating the data value at these positions. Each path thus leads to a universally quantified implication of the form $\forall Y.\ \phi_p \Rightarrow f_{\mathcal{A}}(q_p)$. All valuation words not accepted by the EQDA semantically go to the formula *false*, hence an additional conjunct $\forall Y.\ \neg(\bigvee_p \phi_p) \Rightarrow \mathit{false}$ is added to the formula. So the final formula is $\mathcal{T}(\mathcal{A}) = \big(\bigwedge_p \forall Y.\ \phi_p \Rightarrow f_{\mathcal{A}}(q_p)\big)\ \wedge\ \big(\forall Y.\ \neg(\bigvee_p \phi_p) \Rightarrow \mathit{false}\big)$.

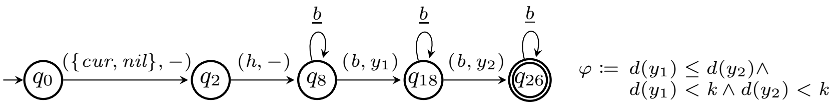


**Fig. 1.** A path in the automaton expressing the invariant of the program which finds a key $k$ in a sorted list.The full automaton is presented in [22].

We next explain, through an example, the construction of the structural constraints $\phi_p$ (for details see [22]). Consider program *list-sorted-find* which searches for a key in a sorted list. The EQDA corresponding to the loop invariant learned for this program is presented in [22]. One of the simple paths in the automaton (along with the associated self-loops on $\underline{b}$) is shown in Fig 1. The structural constraint $\phi_p$ intuitively captures all valuation words which are accepted by the automaton along $p$; for the path in the figure $\phi_p$ is $(cur = nil \wedge h \rightarrow^+ y_1 \wedge y_1 \rightarrow^+ y_2)$ and the formula $\forall y_1 y_2.\ (cur = nil \wedge h \rightarrow^+ y_1 \wedge y_1 \rightarrow^+ y_2) \Rightarrow (d(y_1) \leq d(y_2) \wedge d(y_1) < k \wedge d(y_2) < k)$ is the corresponding conjunct in the learned invariant. Applying this construction yields the following theorem.

**Theorem 4.** *Let $\mathcal{A}$ be an EQDA, $w$ a data word, and $c$ the program configuration corresponding to $w$. If $w \in \mathcal{L}(\mathcal{A})$, then $c \models \mathcal{T}(\mathcal{A})$. Additionally, if $\mathcal{T}(\mathcal{A})$ is a* STRAND *formula, then the implication also holds in the opposite direction.*

APF allows the universal variables to be related by $\leq$ or $=$ and not $<$. Hence, along paths where $y_1 < y_2$, we over-approximate the structural constraint $\phi_p$ to $y_1 \leq y_2$ and, subsequently, the data formula $f_{\mathcal{A}}(q_p)$ is abstracted to include $d(y_1) = d(y_2)$. This leads to an abstraction of the actual semantics of the EQDA and is the reason Theorem 4 only holds in one direction for the APF.

## 7 Implementation and Evaluation on Learning Invariants

We apply the active learning algorithm for QDAs, described in Section 4, in a passive learning framework in order to learn quantified invariants over lists and arrays from a finite set of samples $S$ obtained from dynamic test runs.

**Implementing the Teacher:** In an active learning algorithm, the learner can query the teacher for membership and equivalence queries. In order to build a passive learning algorithm from a sample $S$, we build a teacher, who will use $S$ to answer the questions of the learner, ensuring that the learned set contains $S$.

The teacher knows $S$ and wants the learner to construct a small automaton that includes $S$; however, the teacher does not have a particular language of data words in mind, and hence cannot answer questions precisely. We build a teacher who answers queries as follows: On a membership query for a word $w$, the teacher checks whether $w$ belongs to $S$ and returns the corresponding data formula. The teacher has no knowledge about the membership for words which were not realized in test runs, and she rejects these. She also does not know whether the formula she computes on words that get manifest can be weaker; but she insists on that formula. By doing these, the teacher errs on the side of keeping the invariant semantically small. On an equivalence query, the teacher just checks that the set of samples $S$ is contained in the conjectured invariant. If not, the teacher returns a counter-example from $S$. Note that the passive learning algorithm hence guarantees that the automaton learned will be a superset of $S$ and will take polynomial time in the learnt automaton. We show the efficacy of this passive learning algorithm using experimental evidence.

**Implementation of a Passive Learner of Invariants:** We first take a program and using a test suite, extract the set of concrete data-structures that get manifest at loop-headers (for learning loop invariants) and at the beginning/end of functions (for learning pre/post conditions). The test suite was generated by enumerating all possible arrays/lists of a small bounded length, and with data-values from a small bounded domain. We then convert the data-structures into a set of formula words, as described below, to get the set $S$ on which we perform passive learning. We first fix the formula lattice $\mathcal{F}$ over data formulas to be the Cartesian lattice of atomic formulas over relations $\{=, <, \leq\}$. This is sufficient to capture the invariants of many interesting programs such as sorting routines, searching a list, in-place reversal of sorted lists, etc. Using lattice $\mathcal{F}$, for every program configuration which was realized in some test run, we generate a formula word for every valuation of the universal variables over the program structures. We represent these formula words as a mapping from the symbolic word, encoding the structure, to a data formula in the lattice $\mathcal{F}$. If different inputs realize the same structure but with different data formulas, we associate the symbolic word with the join of the two formulas.

**Implementing the Learner:** We used the libALF library [23] as an implementation of the active learning algorithm [1]. We adapted its implementation to our setting by modeling QDAs as Moore machines. If the learned QDA is not elastic, we elastify it as described in Section 5. The result is then converted to a quantified formula over Strand or the APF and we check if the learned invariant was adequate using a constraint solver.

**Table 1.** Results of our experiments

| Example | LOC | #Test inputs | $T_{teacher}$ (s) | #Eq. | #Mem. | Size #states | Elastification required ? | $T_{learn}$ (s) |
|---|---|---|---|---|---|---|---|---|
| array-find | 25 | 310 | 0.05 | 2 | 121 | 8 | no | 0.00 |
| array-copy | 25 | 7380 | 1.75 | 2 | 146 | 10 | no | 0.00 |
| array-compare | 25 | 7380 | 0.51 | 2 | 146 | 10 | no | 0.00 |
| insertion-sort-outer | 30 | 363 | 0.19 | 3 | 305 | 11 | no | 0.00 |
| insertion-sort-innner | 30 | 363 | 0.30 | 7 | 2893 | 23 | yes | 0.01 |
| selection-sort-outer | 40 | 363 | 0.18 | 3 | 306 | 11 | no | 0.01 |
| selection-sort-inner | 40 | 363 | 0.55 | 9 | 6638 | 40 | yes | 0.05 |
| list-sorted-find | 20 | 111 | 0.04 | 6 | 1683 | 15 | yes | 0.01 |
| list-sorted-insert | 30 | 111 | 0.04 | 3 | 1096 | 20 | no | 0.01 |
| list-init | 20 | 310 | 0.07 | 5 | 879 | 10 | yes | 0.01 |
| list-max | 25 | 363 | 0.08 | 7 | 1608 | 14 | yes | 0.00 |
| list-sorted-merge | 60 | 5004 | 10.50 | 7 | 5775 | 42 | no | 0.06 |
| list-partition | 70 | 16395 | 11.40 | 10 | 11807 | 38 | yes | 0.11 |
| list-sorted-reverse | 25 | 27 | 0.02 | 2 | 439 | 18 | no | 0.00 |
| list-bubble-sort | 40 | 363 | 0.19 | 3 | 447 | 12 | no | 0.01 |
| list-fold-split | 35 | 1815 | 0.21 | 2 | 287 | 14 | no | 0.00 |
| list-quick-sort | 100 | 363 | 0.03 | 1 | 37 | 5 | no | 0.00 |
| list-init-complex | 80 | 363 | 0.05 | 1 | 57 | 6 | no | 0.01 |
| lookup_prev | 25 | 111 | 0.04 | 3 | 1096 | 20 | no | 0.01 |
| add_cachepage | 40 | 716 | 0.19 | 2 | 500 | 14 | no | 0.01 |
| Glib sort (merge) | 55 | 363 | 0.04 | 1 | 37 | 5 | no | 0.00 |
| Glib insert_sorted | 50 | 111 | 0.04 | 2 | 530 | 15 | no | 0.01 |
| devres | 25 | 372 | 0.06 | 2 | 121 | 8 | no | 0.00 |
| rm_pkey | 30 | 372 | 0.06 | 2 | 121 | 8 | no | 0.00 |
| GNU Coreutils sort | 2500 | 1 File | 0.00 | 17 | 4996 | 5 | yes | 0.07 |
| Learning Function Pre-conditions | | | | | | | | |
| list-sorted-find | 20 | 111 | 0.01 | 1 | 37 | 5 | no | 0.00 |
| list-init | 20 | 310 | 0.02 | 1 | 26 | 4 | no | 0.00 |
| list-sorted-merge | 60 | 329 | 0.06 | 3 | 683 | 19 | no | 0.01 |

**Experimental Results:**[1] We evaluate our approach on a suite of programs (see Table 1) for learning invariants and preconditions. For every program, we report the number of lines of C code, the number of test inputs and the time ($T_{teacher}$) taken to build the teacher from the samples collected along these test runs. We also report the number of equivalence and membership queries answered by the teacher in the active learning algorithm, the size of the final elastic automata, whether the learned QDA required any elastification and finally, the time ($T_{learn}$) taken to learn the QDA.

[1] More details at `http://web.engr.illinois.edu/~garg11/learning_qda.html`

The names of the programs in Table 1 are self-descriptive and we only describe some of them. The *inner* and *outer* suffix in insertion and selection sort corresponds to learning loop-invariants for the inner and outer loops in those sorting algorithms. The program *list-init-complex* sorts an input array using heap-sort and then initializes a list with the contents of this sorted array. Since heap-sort is a complex algorithm that views an array as a binary tree, none of the current automatic white-box techniques for invariant synthesis can handle such complex programs. However, our learning approach being black-box, we are able to learn the correct invariant, which is that the list is sorted. Similarly, synthesizing post-condition annotations for recursive procedures like merge-sort and quick-sort is in general difficult for white-box techniques, like *interpolation*, which require a post-condition. In fact, SAFARI [14], which is based on *interpolation*, cannot handle list-structures, and also cannot handle array-based programs with *quantified* preconditions which precludes verifying the array variants of programs like *sorted-find*, *sorted-insert*, etc., which we can handle.

The methods *lookup_prev* and *add_cachepage* are from the module cachePage in a verified-for-security platform for mobile applications [24]. The module cachePage maintains a cache of the recently used disc pages as a priority queue based on a sorted list. The method *sort* is a merge sort implementation and *insert_sorted* is a method for insertion into a sorted list. Both these methods are from Glib which is a low-level C library that forms the basis of the GTK+ toolkit and the GNOME environment. The methods *devres* and *rm_pkey* are methods adapted from the Linux kernel and an Infiniband device driver, both mentioned in [6]. Finally, we learn the sortedness property (with respect to the method *compare* that compares two lines) of the method *sortlines* which lies at the heart of the GNU core utility to sort a file. The time taken by our technique to learn an invariant, being black-box, largely depends on the complexity of the property and not the size of the code, as is evident from the successful application of our technique to this large program.

All experiments were completed on an Intel Core i5 CPU at 2.4GHz with 6GB of RAM. For all examples, our prototype implementation learns an adequate invariant really fast. Though the learned QDA might not be the smallest automaton representing the samples $S$ (because of the inaccuracies of the teacher), in practice we find that they are reasonably small (fewer than 50 states). Moreover, we verified that the learned invariants were adequate for proving the programs correct by generating verification conditions and validating them using an SMT solver (these verified in less than 1s). It is possible that SMT solvers can sometimes even handle non-elastic invariants and VCs; however, in our experiments, it was not able to handle such formulas without giving extra triggers, thus suggesting the necessity of the elastification of QDAs. Learnt invariants are complex in some programs; for example the invariant QDA for the program *list-sorted-find* is presented in [22] and corresponds to:

$$head \neq nil \wedge (\forall y_1 y_2 . head \rightarrow^* y_1 \rightarrow^* y_2 \Rightarrow d(y_1) \leq d(y_2)) \wedge ((cur = nil \wedge \forall y_1 . head \rightarrow^* y_1 \Rightarrow d(y_1) < k) \vee (head \rightarrow^* cur \wedge \forall y_1 . head \rightarrow^* y_1 \rightarrow^+ cur \Rightarrow d(y_1) < k)).$$

**Future Work:** We believe that learning of structural conditions of data-structure invariants using automata is an effective technique, especially for quantified properties where passive or machine-learning techniques are not currently known. However, for the data-formulas themselves, machine learning can be very effective [18], and we would like to explore combining automata-based structural learning (for words and trees) with machine-learning for data-formulas.

**Acknowledgements.** We would like to thank Xiaokang Qiu for valuable discussions on the automata model for Strand formulas. This work is partially supported by NSF CAREER award #0747041.

## References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
2. Kearns, M.J., Vazirani, U.V.: An introduction to computational learning theory. MIT Press, Cambridge (1994)
3. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL, pp. 611–622. ACM (2011)
4. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)
5. Madhusudan, P., Qiu, X.: Efficient decision procedures for heaps using STRAND. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 43–59. Springer, Heidelberg (2011)
6. Kong, S., Jung, Y., David, C., Wang, B.-Y., Yi, K.: Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 328–343. Springer, Heidelberg (2010)
7. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
8. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 548–562. Springer, Heidelberg (2005)
9. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
10. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Abstract domains for automated reasoning about list-manipulating programs with infinite data. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 1–22. Springer, Heidelberg (2012)
11. McMillan, K.L.: Interpolation and SAT-Based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
12. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
13. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)

14. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Safari: Smt-based abstraction for arrays with interpolants. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 679–685. Springer, Heidelberg (2012)
15. Seghir, M.N., Podelski, A., Wies, T.: Abstraction refinement for quantified array assertions. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 3–18. Springer, Heidelberg (2009)
16. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
17. Chen, Y.F., Wang, B.Y.: Learning boolean functions incrementally. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 55–70. Springer, Heidelberg (2012)
18. Sharma, R., Nori, A.V., Aiken, A.: Interpolants as classifiers. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 71–87. Springer, Heidelberg (2012)
19. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for eSC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
20. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI, pp. 223–234. ACM (2009)
21. Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D.: Quickly detecting relevant program invariants. In: ICSE, pp. 449–458 (2000)
22. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. CoRR abs/1302.2273 (2013)
23. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: libalf: The Automata Learning Framework. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 360–364. Springer, Heidelberg (2010)
24. Mai, H., Pek, E., Xue, H., King, S.T., Madhusudan, P.: Verifying security invariants in ExpressOS. In: ASPLOS, pp. 293–304 (2013)