Form Methods Syst Des (2015) 47:120–157
DOI 10.1007/s10703-015-0231-6

Quantified data automata for linear data structures: a register automaton model with applications to learning invariants of programs manipulating arrays and lists

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

Published online: 25 August 2015

Abstract We propose a new automaton model, called *quantified data automata* (QDA) over words, that can model quantified invariants over linear data structures, and study their theory, including closure properties, canonical minimality, and decidability of emptiness. We 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 show translations to decidable theories of arrays and lists. We also prove that every QDA has a unique minimally-over-approximating elastic QDA, showing a robust technique for abstracting QDA-expressible properties to the decidable fragments expressed by elastic QDAs. We then give an application of these theoretically sound and efficient active learning algorithms to program verification by building a passive learning framework that efficiently learns adequate quantified linear data structure invariants from samples obtained from dynamic executions for a class of programs.

Keywords Quantified data automata · Register automata · Invariants · Linear data structures

✉ Pranav Garg
garg11@illinois.edu

Christof Löding
loeding@automata.rwth-aachen.de

P. Madhusudan
madhu@illinois.edu

Daniel Neider
neider@automata.rwth-aachen.de

[1] Department of Computer Science, University of Illinois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, IL 61801, USA

[2] Lehrstuhl für Informatik 7, RWTH Aachen University, 52056 Aachen, Germany

1 Introduction

Linear data structures, such as arrays and linked lists, are important unbounded data structures used in computer science. Properties of such linear data structures require *quantification* due to the unbounded nature of these structures. For instance, expressing that the data stored in a linked list is sorted requires quantification. Reasoning with such data structures requires expressing such properties, especially in program verification where such properties can encode pre/post conditions or invariants that help prove a program correct.

In many fields of logic, *automata theory* plays an important role as a normal form for logic. For instance, monadic second order logic on labeled words and trees, both finite and infinite, is captured using finite-state automata on these structures [18,36]. A connection to automata theory is often useful as the simple structure of automata in terms of graphs gives a better arena than logic to study algorithmic problems on the associated logic. Decision procedures such as satisfiability on logic (and even model-checking of systems) can be translated to appropriate emptiness-checking algorithms on graphs [37]. Another important algorithmic procedure that automata yield are *learning algorithms*— automata-based learning algorithms give a means to learn the corresponding logical formulae in various learning models [3,17].

In this paper, our main motivation stems from program verification, in particular the problem of synthesizing loop invariants for programs that express properties of linear data structures. Synthesizing invariants for programs is one of the most challenging problems in verification today. While there are several logical theories of linear data structures and several fragments have a decidable validity problem [7,12,24], our main motivation is to synthesize invariants using learning. We hence seek an automaton model for linear data structures that (a) can express properties of the structure as well as the data contained in the data structure, (b) can express typical quantified properties of such linear data structures that arise in verification, and (c) has subfragments that can be translated to known decidable fragments of logics on arrays and lists.

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 [3]) 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) [20].

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 φ is quantifier-free, and which capture 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.

We can 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. We hence seek automata models for expressing quantified properties of such data-words.

1.1 Data-words and quantified data automata (QDA)

Our first technical contribution is a novel representation (normal form) for quantified properties of linear data structures, called *quantified data automata* (QDA).

Quantified data automata are a new model of automata over data words that are powerful enough to express a class of *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 study several properties of QDAs. First, QDAs can be viewed as accepting data-words at one level, but also as acceptors of *valuation words* at another level, which encode a data-word as well as an interpretation of the universally quantified variables. We show that unique minimal (minimal in number of states) QDA do *not* exist for languages of data-words, but unique minimal (and hence canonical) QDAs do exist for languages of valuation words. The view of QDAs as acceptors of valuation words links it closer to classical automata theory, as we can view QDAs as essentially Moore-machines that read valuation words and output data-formulas.

Turning to closure properties, we show that the class of languages accepted by QDAs are not, in general, closed under union and complementation. Quantified data automata implicitly involve universal quantification on the outermost level, and hence it is not surprising that they are not closed under union and complementation. However, when the formula lattice is closed under conjunction, then we prove that the class of languages accepted by such QDAs are closed under intersection.

1.2 Learning quantified data automata

Our second main contribution is a an efficient active learning algorithm for QDAs. Using our result that for any set of valuation words (data words with valuations for the variables Y), there is a *canonical* QDA, we show that learning valuation words can be reduced to learning *formula words* (words with no data but paired with data formulas). In this model, the learner can ask a membership query for any word and the teacher replies with the data formula corresponding to the word, and the learner can ask an equivalence query giving a QDA, and the teacher either answers yes (if the QDA accepts the language she has in mind) or returns a counterexample consisting of a word and the data formula associated with it. Note that the learning algorithm is not in terms of data words (which have concrete data but no interpretation for universal variables) nor in terms of valuation words (which have concrete data and have interpretation of universal variables) but in terms of formula words (which have

no concrete data but have an interpretation for universal variables). Our learning algorithm is based on an extension of Angluin's learning algorithm for DFAs [3], extended to QDAs by viewing QDAs as Moore machines accepting formula words. 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. The learning algorithm we develop can hence be used to learn the quantified logical formulas that QDAs represent.

1.3 Elastic quantified data automata (EQDA), decidable logics, unique minimal over-approximations

The class of quantified properties that can be expressed using QDAs is very powerful, and does not admit decidable satisfiability problems, in general. The validity of the corresponding logical formulas in the theory of arrays and lists are also undecidable, in general.

In the context of program verification, even if we use QDAs to learn invariants, we will be unable to *verify* automatically whether the learned properties are adequate invariants for the program at hand. Even though SMT solvers support heuristics to deal with quantified theories (like e-matching [29]), in our experiments, the verification conditions derived from invariants expressed as QDAs could not be handled by such SMT solvers. The third contribution of this paper is hence to lift QDAs and the learning algorithms to subclasses that admit decidable validity problems.

We identify a subclass of QDAs (called EQDAs) and show two main results for them: (a) EQDAs can be converted to formulas of *decidable* logics, to the array property fragment [7] when modeling arrays and the decidable STRAND fragment [24] 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 EQDA.

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 [24]). 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 [7] and the decidable fragment of STRAND disallows this by permitting only the use of $\rightarrow^*$ or $\rightarrow^+$ in order to compare quantified variables [24,25]). We identify a *structural restriction* of QDAs that permits only elastic properties to be stated.

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 EQDAs (which then yields decidable verification conditions in the context of program verification). The result is proved by showing that there is a unique way to minimally morph a QDA to one that satisfies the elasticity restrictions.

1.4 Application to verification: passive learning of quantified invariants

The active learning algorithm for QDAs 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. We also refer

the reader to recent work on a new learning model called ICE (learning using examples, counter-examples, and implications), which is a much more robust active learning model for synthesizing invariants [15].

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 size of 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 [23], compositional verification [2,11], 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 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.

The paper is structured as follows. Section 2 informally motivates and illustrates quantified invariants over linear data structures, quantified data automata, examples of elastic invariants, and elastic data automata. Section 3 formally introduces the QDA model, data-words, valuation words, and formula words. In Sect. 4, we explore various properties of QDAs and the languages accepted by them. In Sect. 5 we present an active learning algorithm, showing how Angluin-style learning of finite automata can be extended to learn QDAs. Section 6

introduces the subclass of QDAs, called EQDAs, and shows the unique elastification result of QDA to elastic quantified data automata. In Sect. 7 we describe how linear data structures can be modeled using data-words and properties of linear data structures using QDAs/EQDAs, and also gives a translation from EQDAs to decidable fragments of arrays and lists. We describe an application to *passively* learning invariants involving arrays and lists in program verification in Sect. 8, using the active learning algorithm for QDAs/EQDAs, and present experimental results of our evaluation. Finally, Sect. 9 concludes with some future directions of research.

1.5 Related work

For invariants expressing properties on the dynamic heap, *shape analysis* techniques are the most well known [32], 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, [5] 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* [27] 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 [1,19,28,33]. These methods use different heuristics like term abstraction [1], preferring smaller constants [19,28] and use of existential ghost variables [33] to ensure that the interpolant converges on an invariant from a *finite* set of spurious counter-examples. IC3 [6] 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 [10], and also extended to quantified invariants [23]. However, unlike us, [23] 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 [34]. Variants of the Houdini algorithm [14] essentially use conjunctive Boolean learning (which can be achieved in polynomial time) to learn conjunctive invariants over templates of atomic formulas (see also [35]). The most mature work in this area is Daikon [13], 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

2.1 List and array invariants

Consider a typical invariant in a sorting program over lists where the loop invariant is expressed as:

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

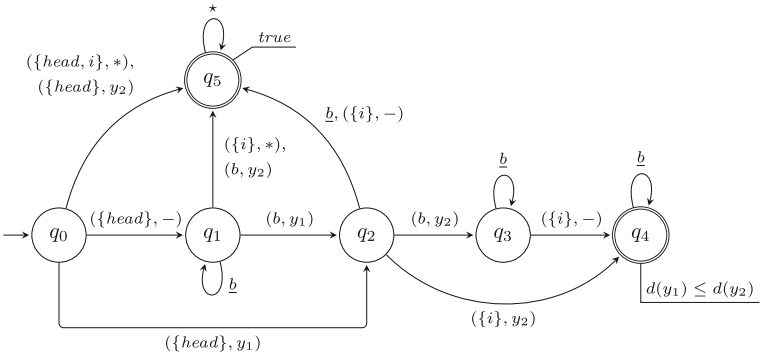
This says that for all cells y_1 that occur somewhere in the list pointed to by *head* and where y_2 is the successor of y_1, and where y_1 and y_2 are before the cell pointed to by a scalar pointer variable i, the data value stored at y_1 is no larger than the data value stored at y_2. This formula is *not* in the decidable fragment of STRAND [24,25] 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:

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

Note that the above formula is not in the decidable array property fragment [7].

2.1.1 Quantified data automata

The key idea in this paper is an automaton model for expressing such constraints called 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.

2.1.2 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 [24,25]).

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

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

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:

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

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 QDA, called *elastic quantified data automata* (EQDA) in Sect. 6, which structurally captures the constraint that quantified variables can be related only using elastic relations (like $\rightarrow^*$ and $\leq$). Furthermore, we show in Sect. 7 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 Preliminaries

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.

Definition 1 *(Data words)* Let $PV = \{p_1, \ldots, p_r\}$ be a finite set of pointer variables, $\Sigma = 2^{PV}$, and D a data domain. A *data word* over PV and D is a word $u \in (\Sigma \times D)^*$ where every $p \in PV$ occurs exactly once in u (i.e., for each $u = a_1 \ldots a_n$ and $p \in PV$, there exists precisely one $j \in \{1, \ldots, n\}$ such that $a_j = (X, d)$ and $p \in X$).

The empty set in the first component of a data word corresponds to a blank symbol indicating that no pointer variable occurs at this position. We also denote this blank symbol by the letter b.

Let $Y = \{y_1, \ldots, y_k\}$ be a nonempty, finite set of *universally quantified variables*. The automata we build accepts 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).

Definition 2 (*Valuation word*) A *valuation word* is a word $v \in (\Sigma \times (Y \cup \{-\}) \times D)^*$ where v projected to its first and third component forms a data word and where each $y \in Y$ occurs exactly once in v.

We use the symbol "$-$" to denote positions in valuation words where no universally quantified variable occurs. Note that the choice of the alphabet ensures that all universally quantified variables have to occur at different positions and is technically convenient.

A valuation word corresponds to a data word with a valuation of Y. This is formalized by the following definition.

Definition 3 Given a valuation word $v \in (\Sigma \times (Y \cup \{-\}) \times D)^*$, the corresponding data word is the word $\mathrm{dw}(v) \in (\Sigma \times D)^*$ resulting from projecting v to its first and third components.

Later, we will also consider a third type of words, called *symbolic words*. In contrast to data and valuation words, symbolic words only capture the structure of a list or array but do not contain data.

Definition 4 (*Symbolic word*) Let $\Sigma = 2^{PV}$ and $\Pi = \Sigma \times (Y \cup \{-\})$. A *symbolic word* is a word $w \in \Pi^*$ where each $p \in PV$ occurs exactly once in w and each $y \in Y$ occurs exactly once in w.

We denote the symbol in Π representing neither a pointer nor a universally quantified variable by $\underline{b} = (b, -)$. The next definition establishes a connection between symbolic and valuation words.

Definition 5 Given a valuation word $v \in (\Sigma \times (Y \cup \{-\}) \times D)^*$, the corresponding symbolic word is the word $\mathrm{sw}(v) \in \Pi^*$ resulting from projecting v to its first two components.

To express the properties on the data, let us 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.

QDA use a *finite* set F of formulas over the atoms $d(y_i), i \in \{1, \ldots, n\}$, which we interpret as the data values of the cells pointed to by the variables $y_1, \ldots, y_n$. We assume that this set is organized in a (bounded semi-)lattice, which leads to the following definition.

Definition 6 (*Formula lattice*) A *formula lattice* $\mathcal{F} = (F, \sqsubseteq, \sqcup, \mathit{false}, \mathit{true})$ is a tuple consisting of a finite set F of formulas over the atoms $d(y_1), \ldots, d(y_n)$, a partial-order relation $\sqsubseteq$ over F, a least-upper bound operator $\sqcup$, and the formulas *false* and *true*, which are required to be in F and correspond to the bottom and top elements of the lattice. Furthermore, we require that whenever $\alpha \sqsubseteq \beta$, then $\alpha \Rightarrow \beta$. Also, we require that the formulas in the lattice are pairwise *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 $\alpha \sqsubseteq \beta$ if and only if $\alpha \Rightarrow \beta$, 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 β also appear in α. The size of a Cartesian lattice is exponential in the number of literals.

We are now ready to introduce the automaton model.

Definition 7 (*Quantified data automata*) Let PV be a finite set of program variables, Y a finite, nonempty set of universally quantified variables, D a data domain, and $\mathcal{F}$ a formula lattice over a finite set F of formulas. A *quantified data automaton (QDA)* is a tuple $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$ where Q is a finite, nonempty set of states, $\Pi = \Sigma \times (Y \cup \{-\})$ is the input alphabet, $\delta\colon Q \times \Pi \dashrightarrow Q$ is the (partial) transition function, and $f\colon Q \to F$ is the *final-evaluation function*, which maps each state to a data formula.

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 $u \in (\Sigma \times D)^*$ if it accepts *all possible* valuation words v extending u with a valuation over Y. We formalize this below.

A configuration of a QDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$ is a pair (q, r) where $q \in Q$ and $r\colon Y \dashrightarrow D$ is a partial variable assignment. The initial configuration is (q_0, r_0) where the domain of r_0 is empty.

The run of $\mathcal{A}$ on a valuation word $v = (a_1, y_1, d_1) \ldots (a_n, y_n, d_n) \in (\Sigma \times (Y \cup \{-\}) \times D)^*$ is a sequence $(q_0, r_0), \ldots, (q_n, r_n)$ of configurations that satisfies $\delta(q_i, (a_i, y_i)) = q_{i+1}$ and

$$r_{i+1} = \begin{cases} r_i\{y_i \leftarrow d_i\} & \text{if } y_i \in Y; \\ r_i & \text{if } y_i = -; \end{cases}$$

where $i \in [0, n)$, the configuration (q_0, r_0) is the initial configuration, and $r_i\{y_i \leftarrow d_i\}$ corresponds to the mapping r_i in which the argument y_i is mapped to the value d_i. We use $\mathcal{A}\colon (q_0, r_0) \xrightarrow{v} (q_n, r_n)$ as a shorthand-notation.

The QDA $\mathcal{A}$ accepts a valuation word v if $\mathcal{A}\colon (q_0, r_0) \xrightarrow{v} (q, r)$ with $r \models f(q)$; that is, after reading the valuation word, the data stored in the registers satisfies the formula annotating the state finally reached. The language $L_{val}(\mathcal{A})$ is the set of valuation words accepted by $\mathcal{A}$.

The QDA $\mathcal{A}$ accepts a data word $u \in (\Sigma \times D)^*$ if $\mathcal{A}$ accepts all valuation words v with $\mathrm{dw}(v) = u$. The language $L_{dat}(\mathcal{A})$ is the set of data words accepted by $\mathcal{A}$.

To ease working with QDAs and to obtain the intended semantics, we assume throughout this chapter that each QDA satisfies two further constraints:

– Each QDA verifies that its input satisfies the constraints on the number of occurrences of variables from PV and Y. All inputs violating these constraints (i.e., all inputs that are not valuation words) either do not admit a run due to missing transitions or lead to a dedicated state labeled with the data formula *false*. This property implies that the states of an QDA are "typed" with the set of variables that have been read so far. As a consequence, cycles in the transition structure of an QDA can only be labeled with $\underline{b}$-symbols. Note that

this assumption is no restriction because both the language of valuation words and the language of data words are defined in terms of words that satisfy the correct occurrence of variables from PV and Y.

– Each QDA verifies that the universally quantified variables occur in its input in the same fixed order, say $y_1 \prec \cdots \prec y_k$. All valuation words violating this order lead to a dedicated state labeled with the data formula *true* (i.e., all such valuation words are accepted). The rationale behind this assumption is the following: since the variables $y \in Y$ are universally quantified, it is sufficient to check a property with respect to a fixed order and a different order should not change the accepted language of data words. Although this assumption is a restriction in general, each QDA can be transformed into one that accepts the same data language and respects the predetermined variable ordering if the formula lattice is closed under conjunction. The idea for such a construction is to use a subset construction that follows all paths that only differ in the order of Y. For each state in a set of states reached like that, one remembers in which order the variables in Y have occurred. At the final states, one uses the conjunction of all formulas in the set with the appropriate renaming of the variables in Y. Due to the universal semantics of QDAs, this captures a QDA that accepts the same data language as original automaton. Since most natural formula lattices, such as the full lattice and the Cartesian lattice (which we use in this chapter), are closed under conjunction, we can without loss of generality assume that each QDA respects a fixed ordering of the universally quantified variables.

4 Properties of QDAs

In this section, we study properties of QDAs, such as whether QDAs allow for canonical representations, closure under Boolean operations, and decidability results.

4.1 Viewing QDAs as Moore machines

Moore machines are extensions of deterministic finite automata that are equipped with output at their states and define a mapping rather than accept a language. On a syntactical level, QDAs can be viewed as such machines, where the output corresponds to the formulas at the final states of the QDA. Taking this view of QDAs allows us to derive some results by using the theory of Moore machines. Formally, Moore machines are defined as follows.

Definition 8 (*Moore machine*) A *Moore machine* is a tuple $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta, \lambda)$ where Q is a nonempty, finite set of states, Σ is the input alphabet, Γ is the output alphabet, $q_0 \in Q$ is the initial state, $\delta\colon Q \times \Sigma \to Q$ is the transition function, and $\lambda\colon Q \to \Gamma$ is the output function that assigns an output-symbol to each state.

The run of a Moore machine $\mathcal{M}$ on a word $u = a_1 \dots a_n$ is a sequence $q_0, \dots, q_n$ of states that satisfies $\delta(q_i, a_{i+1}) = q_{i+1}$ for all $i \in [0, n)$; as in the case of QDAs, we use the shorthand-notation $\mathcal{M}\colon q_0 \xrightarrow{u} q_n$ to denote the run of $\mathcal{M}$ on u. Each Moore machine defines a total function $f_{\mathcal{M}}$ that maps an input-word $u \in \Sigma^*$ to the output of the state that $\mathcal{M}$ reaches after reading u; more precisely, we define $f_{\mathcal{M}}(u) = \lambda(q)$ where $\mathcal{M}\colon q_0 \xrightarrow{u} q$. Finally, we call a function $f\colon \Sigma^* \to \Gamma$ *Moore machine computable* if there exists a Moore machine $\mathcal{M}$ such that $f = f_{\mathcal{M}}$.

Let us now describe how one can view QDAs as Moore machines. Recall that QDAs define two kind of languages, a language of data words and a language of valuation words. On the

level of valuation words, we can understand a QDA as an automaton that reads the structural part of a valuation word (i.e., a symbolic word) and outputs a data formula capturing the data. To make this intuition more precise, let us introduce another type of words, which we call *formula words*.

Definition 9 (*Formula words*) Let PV be a finite set of pointer variables, Y a finite set of universally quantified variables, and $\mathcal{F}$ a lattice over a set F of formulas. A *formula word* is a finite word $(w, \varphi) \in (\Pi^* \times F)$ where, as before, $\Pi = \Sigma \times (Y \cup \{-\})$, and each $p \in PV$ and each $y \in Y$ occurs exactly once in w.

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 Σ), a valuation for the quantified variables (modeled using $Y \cup \{-\}$), as well as a formula over lattice $\mathcal{F}$ over the data domain. For example, $((\{h\}, y_1)(b, -)(b, y_2)(\{t\}, -), d(y_1) \leq d(y_2))$ 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 $d(y_1) \leq d(y_2)$.

We can now view a QDA as an acceptor of formula words.

Definition 10 A QDA $\mathcal{A} = (Q, q_0, \Pi, \delta, f)$ over the set F of data formulas accepts a formula word $(w, \varphi) \in \Pi^* \times F$ if $\mathcal{A}$ reaches a state $q \in Q$ on reading the symbolic word w and $f(q) = \varphi$. Given a QDA $\mathcal{A}$, we define the language $L_f(\mathcal{A}) \subseteq \Pi^* \times F$ of formula words accepted by $\mathcal{A}$ in the usual way. Moreover, we call a language $L_{for} \subseteq \Pi^* \times F$ of formula words *QDA-acceptable* if there exists a QDA $\mathcal{A}$ with $L_f(\mathcal{A}) = L_{for}$.

Note that not every language of formula words is QDA-acceptable; for instance, consider the language

$$L^{\star}_{for} = \{(\underline{b}^i (h, y) \underline{b}^i, \ true) \mid i \geq 1\}.$$

A standard pumping argument shows that $L^{\star}_{for}$ cannot be accepted by a QDA since the number of blanks at the beginning and at the end of a word have to match. Furthermore, words whose symbolic component is not of the form $\underline{b}^i (h, y) \underline{b}^i$ are not present in $L^{\star}_{for}$ but a QDA necessarily assigns a unique formula to every symbolic word. In fact, every QDA-acceptable language L_{for} of formula words has to fulfill the following constraints:

- For every symbolic word $w \in \Pi^*$, there exists a formula φ such that $(w, \varphi) \in L_{for}$.
- If $(w, \varphi) \in L_{for}$ and $(w, \varphi') \in L_{for}$, then $\varphi = \varphi'$.
- There are only finitely many different formulas occurring in formula words in L_{for}.

These constraints allow us to treat QDAs as Moore machines that read symbolic words and output data formulas. In fact, we make the following observation.

Observation 1 A QDA-acceptable language $L_{for} \subseteq \Pi^* \times F$ is an alternative representation of a Moore machine-computable mapping $f\colon \Pi^* \to F$ (in the sense that $(w, \varphi) \in L_{for}$ if and only if $f(w) = \varphi$).

One easily deduces 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 (assuming that all the formulas in the lattice are pairwise non-equivalent). Thus, we can easily reduce the problem of actively learning QDAs to the problem of actively learning Moore machines, as we show in Sect. 5.

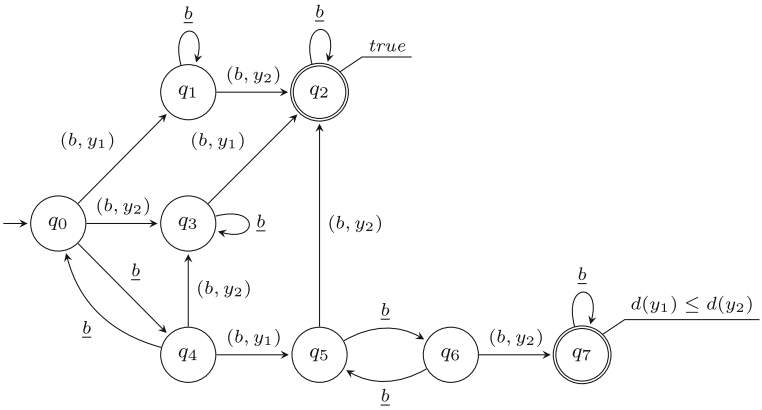


Fig. 1 A QDA expressing that the data on even list positions is sorted. A QDA expressing the property over lists that the data on even positions is sorted. Missing transitions lead to a sink-state labeled with *false*, which is not shown for the sake of readability. All states depicted as a *single circle* are implicitly labeled with the formula *false*

4.2 Canonical QDAs

Recall that QDAs define two kinds of languages, namely a language of data words and a language of valuation words. We begin by observing that we cannot hope for unique minimal QDA on the level of data words.

To see why, consider the QDA $\mathcal{A}$ in Fig. 1 over $PV = \emptyset$ and $Y = \{y_1, y_2\}$. It accepts all valuation words in which

- $d(y_1) \leq d(y_2)$ if y_1 occurs before y_2 and y_1, y_2 are both on even positions; or
- $y_2 < y_1$; or
- at least one of y_1 and y_2 does not occur at an even position.

Hence, $\mathcal{A}$ accepts the language of data words that consist of all data words such that the data on even positions is sorted. Since each QDA has to ensure that each variable occurs exactly once, the number of states of $\mathcal{A}$ is minimal for defining this language of data words.

However, a QDA in which we replace the transition $\delta(q_6, \underline{b}) = q_5$ by the transition $\delta(q_6, \underline{b}) = q_1$ accepts the same language of data words. This new QDA checks the sortedness only for all y_1, y_2 with $y_2 = y_1 + 2$, which is sufficient. This shows that the transition structure of a state-minimal QDA for a given language of data words is not unique.

On the level of valuation words, on the other hand, there exists a minimal canonical QDA, which is formalized next. This is because the automaton model is deterministic and, since all universally quantified variables are in different positions, the automaton cannot derive any relation on the data values during its run. Formally, we can state the following theorem.

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

Proof Consider a language L_{val} of valuation words that can be accepted by a QDA, and let $w \in \Pi^*$ be a symbolic word. Then there must be a formula ψ_w in the lattice that characterizes precisely the valuation words $v \in L_{val}$ that extend w with data (i.e., that satisfy $\mathrm{sw}(v) = w$). Since we assume that all the formulas in the lattice are pairwise non-equivalent, this formula is uniquely determined. This formula ψ_w is obtained by considering for each valuation word

v with $\mathrm{sw}(v) = w$ the greatest-lower bound φ_v of all formulas in the lattice that are satisfied in v, and then taking the least-upper bound of all these φ_v.

In fact, the formula ψ_w is independent of the actual QDA. To prove this, take any QDA $\mathcal{A}$ that accepts L_{val}. Then w leads to some state q in $\mathcal{A}$ that outputs the formula $f(q)$, where f is the final-evaluation function in $\mathcal{A}$. If w leads to any other formula in another QDA $\mathcal{A}'$, then $\mathcal{A}'$ accepts a different language of valuation words.

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 (see [22]), hence the theorem. □

4.3 Boolean operations

Because of the universal semantics of QDAs, it is easy to see that the class of QDA-definable data languages is not closed under complement. Since the universal quantifier does not distribute over disjunctions, the class is also not closed under union.

Proposition 1 *There is a lattice $\mathcal{F}$ that is closed under all Boolean operations, such that the class of QDA-definable languages of data words over this lattice is not closed under complement and union.*

Proof Take the data domain of the integers, and all Boolean formulas using the binary predicate $\leq$. The set of pointer variables is empty. We have already seen that the set L of data words in which the data is sorted in ascending order is QDA definable. The complement of this language is the set of data words in which there are two positions y_1 and y_2 such that $y_1 < y_2$ and $d(y_1) > d(y_2)$. Assume that there is a QDA $\mathcal{A}$ accepting this language. We assume here that the QDA uses only two variables y_1, y_2 but the argument can easily be extended to any number of variables. Consider the two data words $w_1 = (b, 2)(b, 1)(b, 3)(b, 4)$ and $w_2 = (b, 1)(b, 2)(b, 4)(b, 3)$. Both have to be accepted by $\mathcal{A}$. However, $\mathcal{A}$ then also accepts the data word $w = (b, 1)(b, 2)(b, 3)(b, 4)$ because for each valuation y_1, y_2 in w there is a valuation in w_1 or w_2 that cannot be distinguished from the valuation of w by $\mathcal{A}$ (i.e., the valuation word leads to the same state and satisfies the same data formulas); for instance, the valuation $(b, y_1, 1)(b, -, 2)(b, y_2, 3)(b, -, 4)$ in w cannot be distinguished from $(b, y_1, 2)(b, -, 1)(b, y_2, 3)(b, -, 4)$ in w_1. Thus, all valuations of w are accepted but w is in L and not in its complement.

For the non-closure under union consider the set L from above, and the set L' of data words in which the data is sorted in descending order. An argument similar to the one from above shows that the union of these two languages is not QDA definable. □

Since universal quantification distributes over conjunction, we obtain a positive result for intersection of data languages.

Proposition 2 *Let $\mathcal{F}$ be a formula lattice. If $\mathcal{F}$ is closed under conjunction, then the class of QDA-definable languages of data words is closed under intersection.*

Proof A standard product construction for $\mathcal{A}_1$ and $\mathcal{A}_2$ with $f(q_1, q_2) = f(q_1) \wedge f(q_2)$ results in a QDA for the desired language. □

As for the case of canonical QDAs, we now consider closure properties on the level of valuation words.

Proposition 3 *Let $\mathcal{F}$ be a formula lattice. The class of QDA-definable languages of valuation words is closed under*

1. *Complement if $\mathcal{F}$ is closed under negation;*
2. *Union if $\mathcal{F}$ is closed under disjunction; and*
3. *Intersection if $\mathcal{F}$ is closed under conjunction.*

Proof For the complement, just take the negation of the final formulas. For union and intersection use a product construction and combine the formulas by disjunction for union, and conjunction for intersection. □

This shows that, on the level of valuation words, QDAs behave much more like standard automata, given that the lattice has the corresponding properties. For the case of union and intersection, we additionally obtain the following weaker version of the results if we do not assume the corresponding closure properties of the lattice.

Proposition 4 *Let $\mathcal{F}$ be a formula lattice (with least upper bound and greatest lower bound operators), and let $\mathcal{A}_1, \mathcal{A}_2$ be two QDAs. There exists a unique minimal QDA-definable language of valuation words containing $L_{val}(\mathcal{A}_1) \cup L_{val}(\mathcal{A}_2)$, and there is a unique maximal QDA-definable language of valuation words contained in $L_{val}(\mathcal{A}_1) \cap L_{val}(\mathcal{A}_2)$.*

Proof As in Proposition 3, we use product constructions, now combining the final formulas using the least upper bound and greatest lower bound instead of disjunction and conjunction.□

4.4 Decidability results

The expressive power of QDAs depends on the data domain and the formula lattice for testing properties of the data. The formula lattices used for expressing nontrivial properties of data words usually lead to the undecidability of the emptiness problem for QDAs. For instance, using the integers as data domain, and an appropriate signature, it is easy to reduce the halting problem for two-counter machines to the emptiness problem of QDAs. Using blocks of three successive positions, one encodes the line number, and the two counter values in the data. The formulas at the final states are used to check that the data encoding the configurations faithfully simulates the computation of the given two-counter machine (a data domain with linear arithmetic would suffice). With a bit more effort, this result can even be extended to formulas that only use Boolean combinations of equality tests.

In contrast, the universality problem, that is, whether a given QDA accepts all data words (with the appropriate restrictions on the labeling by pointer variables), is decidable, provided the quantifier-free fragment used to express the data formulas is decidable. This amounts to a simple check whether there is a symbolic word that does not admit a run, or leads to a final state with a formula which is not *true* (i.e., not a tautology). In this case, one can construct a valuation word that is not accepted by the QDA, and thus the corresponding data word is also rejected.

5 Learning QDAs

The goal of this section is to develop an learning algorithm for QDAs that operates in Angluin's active learning setting [3]. To this end, we proceed as follows: we begin this section by recapping Angluin's active learning setting for regular languages. Then, we briefly describe how to learn Moore machines in an Angluin-style active learning setting. Finally,

we reduce the problem of actively learning QDAs to the problem of actively learning Moore machines.

5.1 Angluin's active learning setting

Angluin's active learning setting, which she has introduced in [3], is a framework in which the task is to "learn" a regular language $L \subseteq \Sigma^*$ over a fixed alphabet Σ—called *target language*—by actively querying an external source for information. The learning takes place between a learning algorithm—abbreviated *learner*—and the information source—called *teacher*. The teacher can answer two types of queries: membership and equivalence queries.

Membership query	On a membership query, the learner provides a word $u \in \Sigma^*$ and the teacher replies "yes" if $u \in L$ and "no" if $u \notin L$.
Equivalence query	On an equivalence query, the learner provides a regular language, usually given as a DFA $\mathcal{A}$, and the teacher checks whether $\mathcal{A}$ is equivalent to the target language. If this is the case, he returns "yes". If this is not the case, he returns a *counterexample* $u \in L(\mathcal{A}) \Leftrightarrow u \notin L$ as a witness that $L(\mathcal{A})$ and L are indeed different.

Given a teacher for a regular target language L, the learner's task is to find a DFA (usually of minimal size) that passes an equivalence query.

In [3], Angluin has not only introduced the active learning framework but also developed a learning algorithm that learns the unique minimal deterministic automaton that accepts the target language in polynomial time. This algorithm is based on the *Myhill–Nerode congruence* of the target language: given a language $L \subseteq \Sigma^*$, the Myhill-Nerode congruence is the equivalence relation $\sim_L$ over words defined by $u \sim_L v$ if and only if $uw \in L \Leftrightarrow vw \in L$ for all $w \in \Sigma^*$. Angluin's pivotal idea is to start with a coarse approximation of the Myhill-Nerode congruence and refine the approximation, using membership and equivalence queries, until the Myhill-Nerode congruence has been computed exactly; since the number of equivalence classes is finite for every regular language, this approach is guaranteed to terminate eventually.

Internally, Angluin's algorithm stores the data learned so far in a so-called *observation table* $O = (R, S, T)$; the set $R \subseteq \Sigma^*$ is a finite, prefix-closed set of *representatives* that serve to represent equivalence classes, the set $S \subseteq \Sigma^*$ is a finite set of *samples* that are used to distinguish representatives, and $T\colon (R \cup R \cdot \Sigma) \cdot S \to \{\text{"yes"}, \text{"no"}\}$ is a mapping that stores the actual table entries and is filled using membership queries.

Angluin's algorithm proceeds in rounds: In each round, the algorithm extends the observation table until it is *closed* and *consistent*, which roughly corresponds to the situation that the data stored in the table forms a congruence. Then, Angluin's algorithm derives a conjecture DFA from the table (similar to the construction of the minimal DFA from the Myhill-Nerode congruence) and submits this conjecture on an equivalence query. If the teacher replies "yes", the learning terminates; if the teacher returns a counterexample, on the other hand, Angluin's algorithm adds the counterexample along with all of its prefixes as new representatives to the table and proceeds with the next iteration.

We refer the reader to [3] for an in-depth presentation of Angluin's active learning setting and Angluin's algorithm. Here, we just want to summarize the main results.

Theorem 2 (Angluin [3]) *Given a teacher for a regular target language $L \subseteq \Sigma^*$, Angluin's algorithm learns the minimal DFA accepting L in time polynomial in the size n of this DFA and the length m of the longest counterexample returned by the teacher. It asks $\mathcal{O}(n)$ equivalence queries and $\mathcal{O}(mn^2)$ membership queries.*

5.2 Actively learning QDAs as Moore machines

For actively learning QDAs we take the view of QDAs as Moore machines, as described in Sect. 4.1. We first describe how to adapt Angluin's setting to Moore machines and then explain how to apply this to learning QDAs.

In the context of actively learning Moore machines, the target concept is a Moore machine computable function $f\colon \Sigma^* \to \Gamma$. Note that we obtain Angluin's original setting for learning regular languages by letting $\Gamma = \{0, 1\}$.

Given A Moore machine computable function $f\colon \Sigma^* \to \Gamma$, a teacher for f answers queries as follows.

Membership query	On a membership query with a word $u \in \Sigma^*$, the teacher replies the classification $f(u)$.
Equivalence query	On an equivalence query with a Moore machine $\mathcal{M}$, the teacher checks whether $f_{\mathcal{M}} = f$ is satisfied. Is this the case, he returns "yes". If this is not the case, he returns a counterexample $u \in \Sigma^*$ with $f_{\mathcal{M}}(u) \neq f(u)$.

Note that the learner and the teacher do not need to agree a priori on the output alphabet since the learner can obtain this knowledge through membership queries.

One can, in a straight forward manner, adapt Angluin's algorithm—in fact any observation table-based learning algorithms, such as Rivest and Schapire's algorithm [31]—to learn Moore machines. The idea is to lift the Myhill–Nerode congruence to Moore machine computable mappings $f\colon \Sigma^* \to \Gamma$ by defining

$$u \sim_f v \quad \text{if and only if } \forall w \in \Sigma^*\colon f(uw) = f(vw),$$

where $u, v \in \Sigma^*$. Then, it is indeed enough to adapt the mapping T of an observation table to $T\colon (R \cup R \cdot \Sigma) \cdot S \to \Gamma$ and the way conjectures are generated. For the latter, we do no longer produce a DFA as a conjecture but a Moore machine whose output is defined by the function value $f(u)$ of the representatives $u \in R$. Chen et al. [9] demonstrate this adaptation for the case $|\Gamma| = 3$.

In analogy to Angluin's algorithm (see Theorem 2), an algorithm adapted this way learns the unique minimal Moore machine for the target function in time polynomial in this minimal Moore machine and the length of the longest counterexample returned by the teacher. Thus, we obtain the following remark.

Remark 1 Given a teacher for a Moore machine computable function that can answer membership and equivalence queries, the unique minimal Moore machine for this function can be learned in time polynomial in the size of this minimal Moore machine and the length of the longest counterexample returned by the teacher.

We can now simply apply this setting to QDAs viewed as Moore machines. Reformulating the setting for this specific case, we assume that the teacher has access to a QDA-acceptable language $L_{for} \subseteq \Pi^* \times F$ of formula words and answers queries as follows.

Membership query	On a membership query, the learner provides a symbolic word $w \in \Pi^*$, and the teacher returns the unique formula $\varphi \in F$ with $(w, \varphi) \in L_{for}$. Note that such a formula word is guaranteed to exist since L_{for} is a QDA-acceptable language.
Equivalence query	On an equivalence query with a QDA $\mathcal{A}$, the teacher checks whether $L_f(\mathcal{A}) = L_{for}$ is satisfied. If this is the case, he returns "yes". If this is not the case, then there exists a formula word (w, φ) such that $(w, \varphi) \in$

$L_f(\mathcal{A}) \Leftrightarrow (w, \varphi) \notin L_{for}$ (since both $L_f(\mathcal{A})$ and L_{for} contain a formula word of the form (w', φ') for every $w' \in \Pi^*$), and the teacher returns w as counterexample.

Such a teacher for QDAs answers queries in the same manner as a teacher for Moore machines, hence we have reduced the learning of QDAs to learning of Moore machines (using the correspondence from Observation 1 in Sect. 4.1). This allows us to adapt off-the-shelf learning algorithms, such as Angluin's or Rivest and Schapire's algorithm, and we immediately obtain the following result.

Theorem 3 *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 the size of this minimal QDA and the length of the longest counterexample returned by the teacher.*

6 Elastic quantified data automata

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 Sect. 2.

Definition 11 (*Elastic quantified data automata*) A QDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$ is called elastic if each transition on $\underline{b}$ is a self-loop (i.e., whenever $\delta(q, \underline{b}) = q'$ is defined, then $q = q'$).

If a state in an EQDA does not have any outgoing $\underline{b}$-transition, it might seem that the EQDA could still test whether two universally quantified variables, say y_1 and y_2, are bounded distance away (which is the reason for the undecidability of the emptiness problem for QDAs). However, because of the universal semantics of the automaton model, such a test is not possible. This is discussed in more detail in the translation from EQDAs to logic formulas in Sect. 7, where we introduce the notion of irrelevant self-loop.

The learning algorithm that we use to synthesize QDAs does not construct EQDAs in general. However, we can show that every QDA *uniquely over-approximated* by a language of valuation words that can be accepted by an EQDA, as stated in the following theorem. This result crucially relies on the particular structure that elastic automata have, that forces a unique set of words to be added to the language in order to make it elastic. We will refer to the construction in Definition 12 as *elastification*.

To ease the following definition, we introduce a few auxiliary notations: Given a QDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$, let $R_{\underline{b}}(q)$ be the set of state reachable from q via a (possibly empty) sequence of $\underline{b}$-transitions and $R_{\underline{b}}(S) = \bigcup_{q \in S} R_{\underline{b}}(q)$ for a set $S \subseteq Q$. Moreover, we lift the transition function of $\mathcal{A}$ to sets of states: for $S \subseteq Q$ and $a \in \Pi$, let $\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$.

Definition 12 (*Elastification*) Given a QDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$, we define the EQDA $\mathcal{A}_{\text{el}} = (Q_{\text{el}}, \Pi, S_0, \delta_{\text{el}}, f_{\text{el}})$ by

- $Q_{\text{el}} = \{S \mid S \subseteq Q\}$;
- $S_0 = R_{\underline{b}}(q_0)$;
- $f_{\text{el}}(S) = \bigsqcup_{q \in S} f(q)$; and

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

Note that this construction is similar to the usual powerset construction except that we take the "$\underline{b}$-closure" after applying the transition function of $\mathcal{A}$. Moreover, $\mathcal{A}_{\mathrm{el}}$ loops in a state S as soon as a $\underline{b}$-transition is defined for a state $q \in S$.

Theorem 4 *For every QDA $\mathcal{A}$ one can construct an EQDA $\mathcal{A}_{el}$ such that*

- $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{A}_{el})$*; and*
- *for every EQDA $\mathcal{B}$ such that $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{B})$, the inclusion $L_{val}(\mathcal{A}_{el}) \subseteq L_{val}(\mathcal{B})$ holds.*

Proof We begin by observing that $\mathcal{A}_{\mathrm{el}}$ is elastic by definition of δ_{el}. Moreover, a standard induction over the length of valuation words $v = a_1 \dots a_n \in (\Pi \times D)^*$ shows the following: if the run of $\mathcal{A}$ on v is

$$\mathcal{A}\colon q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n,$$

then the run of $\mathcal{A}_{\mathrm{el}}$ on v is

$$\mathcal{A}_{\mathrm{el}}\colon S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} S_n$$

such that $q_i \in S_i$ for all $i \in \{1,\dots,n\}$. This implies $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{A}_{\mathrm{el}})$ because the implication $f(q_n) \to f_{\mathrm{el}}(S_n)$ holds by definition of f_{el}.

Let us now show that the language $L_{val}(\mathcal{A}_{\mathrm{el}})$ is indeed the most precise elastic over-approximation of $L_{val}(\mathcal{A})$. To this end, let $\mathcal{B} = (Q_{\mathcal{B}}, \Pi, q_0^{\mathcal{B}}, \delta_{\mathcal{B}}, f_{\mathcal{B}})$ be an EQDA with $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{B})$. Additionally, let $v \in L_{val}(\mathcal{A}_{\mathrm{el}})$. Thus, the task is to prove that $v \in L_{val}(\mathcal{B})$ holds, too.

Let S be the state reached by $\mathcal{A}_{\mathrm{el}}$ on reading v and p be the state reached by $\mathcal{B}$ on reading v. We now show that $f(q)$ implies $f_{\mathcal{B}}(p)$ for every $q \in S$. Once we have established this, we obtain that $f_{\mathrm{el}}(S)$ implies $f_{\mathcal{B}}(p)$ because $f_{\mathrm{el}}(S)$ is the least formula in the formula lattice that is implied by all formulas $f(q)$ for $q \in S$. Since $v \in L_{val}(\mathcal{A}_{\mathrm{el}})$, the valuation word v satisfies $f_{\mathrm{el}}(S)$ and, hence, also $f_{\mathcal{B}}(p)$. Thus, $v \in L_{val}(\mathcal{B})$.

To prove that $f(q)$ implies $f_{\mathcal{B}}(p)$ for every $q \in S$, pick a state $q \in S$. Following the definition of δ_{el}, we now construct a valuation word $v' \in (D \times \Pi)^*$ that satisfies the following properties:

- $v' \in L_{val}(\mathcal{A})$.
- The run of $\mathcal{A}$ on v' leads to q.
- The run of $\mathcal{B}$ on v' leads to p.

In order to obtain v', we insert symbols of the form $(\underline{b}, d)$ into v. Since the data values at such positions do not occur together with variables, their actual value is unimportant.

For the construction, let $v = a_1 \cdots a_n$ and let

$$\mathcal{A}_{\mathrm{el}}\colon S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} S_n$$

be the run of $\mathcal{A}_{\mathrm{el}}$ on v (so $S = S_n$). Let $q \in S$ and let $q'_n := q$. Since $\delta_{\mathrm{el}}(S_{n-1}, a_n) = S_n$ and $q'_n \in S_n$, there is some state $q'_{n-1} \in S_{n-1}$ and $q_n \in S_n$ such that $\delta(q'_{n-1}, a_n) = q_n$, and $\mathcal{A}\colon q_n \xrightarrow{\underline{b}^{i_n}} q'_n$ for some $i_n \geq 0$. We continue this construction: if $q'_j \in S_j$ is defined for

$j \in \{1, \ldots, n\}$, we construct q'_{j-1}, q_j and i_j as above. For $j = 0$ we finally pick i_0 such that $\mathcal{A} : q_0 \xrightarrow{\underline{b}^{i_0}} q'_0$.

Let $v' = \underline{b}^{i_0} a_1 \underline{b}^{i_1} a_2 \underline{b}^{i_2} \cdots a_n \underline{b}^{i_n}$. By construction, the run of $\mathcal{A}$ on v' leads to $q = q'_n$:

$$\mathcal{A}\colon q_0 \xrightarrow{\underline{b}^{i_0}} q'_0 \xrightarrow{a_1} q_1 \xrightarrow{\underline{b}^{i_1}} q'_1 \ldots \xrightarrow{a_n} q_n \xrightarrow{\underline{b}^{i_n}} q'_n$$

Since v' is obtained from v by inserting $\underline{b}$, the word v' also satisfies the formula $f(q)$ and thus $v' \in L_{val}(\mathcal{A})$. It remains to show that the run of $\mathcal{B}$ on v' leads to p. Since $\mathcal{B}$ is elastic, the only possibility that v' does not lead to p in $\mathcal{B}$ is a missing $\underline{b}$-loop at a position at which we inserted a non-empty sequence of $\underline{b}$. However, since $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{B})$, such a position cannot exist.

We conclude that $f(q)$ implies $f_{\mathcal{B}}(p)$ using the following argument: If $f(q)$ does not imply $f_{\mathcal{B}}(p)$, then there exists an assignment of data values to the variables $y_1, \ldots, y_k$ such that $f(q)$ is satisfied but $f_{\mathcal{B}}(p)$ is not. By changing the data values in v' accordingly, we can produce a valuation word that is accepted by $\mathcal{A}$ but not by $\mathcal{B}$. However, this contradicts the assumption $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{B})$. Thus, $f(q)$ implies $f_{\mathcal{B}}(p)$. □

7 Linear data structures to words and EQDAs to decidable logics

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

7.1 Modeling program configurations as data words

We model program configurations consisting of scalar variables, pointer or index variables,[1] and one (or more) linear data structures—lists or arrays in our case—as data words over a finite set of variables. The resulting data word is over the same domain D as the data in the cells of the data structure.

To simplify our modeling, we replace each scalar variable with an auxiliary pointer variable that points to a cell containing the data of the scalar variable. More precisely, for each scalar variable, we introduce a new pointer variable and extend the data structure with a new cell, which is located before the actual data structure begins and contains the data of the scalar variable; the order in which scalar variables are represented in the data structure is arbitrary but needs to be fixed. To be able to access the data at these positions (recall that QDAs can only access the data at position pointed to by universally quantified variables), we amend QDAs with a register for each such pointer variable and extend the set F of formulas over which the considered QDA works with the atom $d(x)$ for each scalar variable x.

Let c be a program configuration over a linear data structure and a finite set PV of pointer or index variables, and let $\Sigma = 2^{PV}$. We model c as the data word

$$u_c = (a_1, d_1) \ldots (a_n, d_n) \in (\Sigma \times D)^*,$$

such that the i-th symbol of the data word corresponds to the i-th cell of the data structure. In particular, the symbol $a_i \subseteq PV$ contains all pointer or index variables referencing the i-th cell, and d_i is the data stored in that cell.

In the case of lists, some of the pointer variables might be *null* or point to unallocated memory, which cannot be referenced. We capture this situation in the data word by introducing

[1] Index variables occur in the case of arrays and index into arrays.

an auxiliary pointer variable *nil* that points to a new cell at the beginning of the list. All pointer variables that are *null* or point to unallocated memory occur together with *nil*. The data value of the *nil* cell in the data word is not important and can be set to an arbitrary element of D.

Similarly, we introduce two new index variables *index_le_zero* and *index_geq_size* for arrays to capture index variables that are out-of-bounds (we assume that arrays are indexed starting at 0). The variable *index_le_zero* occurs together with all index variables that are less than zero, and *index_geq_size* occurs with those index variables that are either equal to or exceed the size of the array. Let the set *Aux* contain all auxiliary variables that may occur in our encoding.

To model configurations of programs that manipulate more than one data structure, one can use one of the following two approaches: the first approach concatenates the data structures using a special pointer variable $\star_i$ to demarcate the end of the i-th data structure; the second approach models several data structures as one single combined data structure over an extended data domain by convolution of the original data words (i.e., by transforming a pair of words into a word over pairs); we refer the reader to standard textbooks (e.g., Khoussainov and Nerode [21]) for more details about convolution.

Let us illustrate the described translation with an example.

Example 1 Consider a program that takes as input a scalar variable *key* and a list l and partitions l into two separate lists: the first list contains all nodes whose data value is less than *key* and the second list contains all the remaining nodes of l. The program maintains pointer variables h_1 (corresponding to "head"), p, and c_1 (corresponding to a "current" pointer) to point into the first list and h_2 and c_2 to point into the second list. All nodes from h_1 through p in the first list are less than *key*. Similarly, all nodes in the second list (h_2 through c_2) are greater than or equal to *key*.

Let us consider a concrete scenario where l is a list with data values $1, 2, \ldots, 10$ in increasing order, and let $key = 6$. Moreover, consider the program configuration where the first seven nodes of the list have been processed and c_1 is pointing to the node with data value 8. Two data words corresponding to this program configuration—one using concatenation and one using convolution—are depicted in Fig. 2.

7.2 Converting EQDAs to STRAND and the array property fragment

We now describe a translation of an EQDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$ into a formula $\varphi_{\mathcal{A}}$ (in the decidable syntactic fragment of STRAND, respectively in the Array Property Fragment) such that the data word language $L_{dat}(\mathcal{A})$ corresponds to the set of program configurations that model $\varphi_{\mathcal{A}}$. For brevity, we only consider the case of EQDAs working over a single list or array; for multiple lists or arrays, the translation is analogous.

Our translation is based on the notion of *simple paths* in EQDAs. A simple path is a sequence

$$\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$$

of states connected by transitions starting in the initial state such that $\delta(q_i, a_{i+1}) = q_{i+1}$ is satisfies for all $i \in [n]$, no state occurs more than once, and all pointer and universally quantified variables occur exactly once; in particular, this implies $a_i \neq \underline{b}$. Note that there exist only finitely many simple paths in an EQDA because each state is allowed to occur at most once. We denote the set of all simple paths in the EQDA $\mathcal{A}$ by $P_{\mathcal{A}}$.

To simplify the translation, we assume without restricting the class of formulas represented by EQDAs that any EQDA $\mathcal{A}$ fulfills two structural properties:

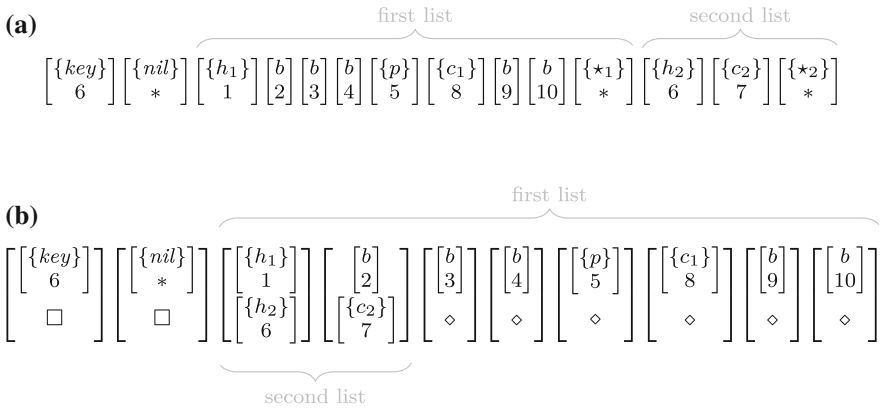


Fig. 2 Two datawords modeling the program configuration described in Example 1. A $*$-entry can be populated with an arbitrary data value. **a** Dataword modeling the concatenation of the two lists of Example 1. **b** Dataword modeling the convolution of the two lists of Example 1. Both $\square$ and $\diamond$ are new auxiliary padding symbols (that must occur at the beginning and the end of the lists, respectively) used to equal the length of the lists

1. Auxiliary variables, such as *nil* or scalar variables, which might have been introduced by the encoding of Sect. 7.1, occur in the beginning of any simple path in the exact same order. Although the exact order is unimportant, we fix one for the sake of simplicity: scalar variables occur first (in some fixed order), followed by *nil* in the case of lists, respectively *index_le _zero* and *index_geq _size* in the case of arrays.
2. Any simple path in $\mathcal{A}$ along which a universally quantified variable occurs together with auxiliary variables leads to a dedicated state labeled with the formula *true*. This means that the acceptance of a data word depends only on such valuations where no universally quantified variable occurs together with auxiliary variables. Since auxiliary variables were introduced for technical reasons only, valuation words in which a universally variable occurs together with auxiliary variables should, therefore, not influence the formula $\varphi_{\mathcal{A}}$.

EQDAs can check properties of the beginning and the end of a data structure, such as whether a pointer variable points to the head or tail of a list. In order to capture such properties, we use the constants 0 and *size* in the case of arrays, respectively *head* and *tail* in the case of lists, that point to the beginning and the end of the considered data structure. We assume that the *size* of an array is available as a variable in the scope of the program. For programs over list structures, the QDA only models the part of the list that can be accessed by traversing the *next* pointer from other pointer variables in the scope of the program. This implies that the head of the list is always available for reference in the EQDAs. Finally, we can express *tail* of the list in STRAND by the following formula: $\exists tail.\ (succ(tail, nil) \wedge head \rightarrow^* tail)$.

We are now ready to describe the actual translation. Roughly speaking, our translation considers each simple path of an EQDA individually, records the structural constraints of the variables along the path, and relates these constraints to the data formula of the final state of the path. By doing so, we construct a path formula φ_π for each simple path π in $\mathcal{A}$. The resulting formula $\varphi_{\mathcal{A}}$ is then the union of all such path formulas and an additional subformula that captures the valuation words not accepted by $\mathcal{A}$. Since there exists only finitely many simple path in $\mathcal{A}$, the resulting formula is finite.

Before we can enter the detailed definition of the formulas, we need another preprocessing of the path under consideration. Basically, we remove self-loops that on this path do not

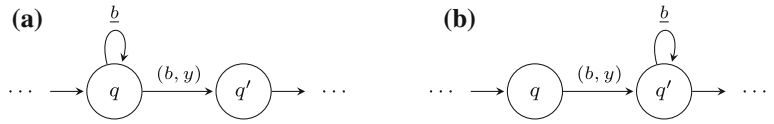


Fig. 3 Base cases of the inductive definition of irrelevant self-loops. **a** An irrelevant self-loop in q. **b** An irrelevant self-loop in q'

contribute to the acceptance of data words, which we call *irrelevant self-loops*. The precise reason for removing these loops becomes clear in Case 5 of the translation below.

For the formal definition of irrelevant self-loops, let π be a simple path in $\mathcal{A}$, and let q, q' be two states on π such that q' is the direct successor of q and the transition connecting q and q' is $\delta(q, (b, y)) = q'$. If q has a self-loop on $\underline{b}$ (i.e., $\delta(q, \underline{b}) = q$), then we define this self-loop inductively to be *irrelevant on* π if either q' has no self-loop on $\underline{b}$ or if this self-loop is irrelevant on π; the former situation is sketched in Fig. 3a. Symmetrically, we define a self-loop on $\underline{b}$ at q' inductively as irrelevant on π if either q has no self-loop on $\underline{b}$ or this self-loop is irrelevant on π (see Fig. 3b).

If a self-loop is irrelevant on π, it cannot contribute to the acceptance of a data word. To see why, consider two valuation words

$$v = v_1(b, y)\underline{b}v_2 \quad \text{and} \quad v' = v_1\underline{b}(b, y)v_2$$

with $\mathrm{dw}(v) = \mathrm{dw}(v')$ (i.e., v and v' only differ in the valuation of the universally quantified variable y by one position). Moreover, assume that v is accepted along π using an irrelevant self-loop in q' on the $\underline{b}$ before v_2 (as in Fig. 3b). In this situation, $\mathcal{A}$ reaches q after reading v_1 and hence rejects v' since q has no transition on $\underline{b}$. Thus, $\mathcal{A}$ rejects $\mathrm{dw}(v') = \mathrm{dw}(v)$. A similar argument applies to the other cases of the definition of irrelevant self-loop.

This reasoning shows that one can safely remove irrelevant loops from a path without changing the accepted language of data words. However, since being an irrelevant self-loop is a property depending on a path, it can happen that there are paths π, π' such that a self-loop in a state q is irrelevant on π but not on π'. We thus cannot remove irrelevant self-loops from $\mathcal{A}$ itself, but have to work on the level of paths.

For the actual translation into a formula, let $\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$ be a simple path in $\mathcal{A}$ with $a_i \in \Sigma \times (Y \cup \{-\})$ and $a_i \neq \underline{b}$ for $i \in \{1, \dots, n\}$. The path formula corresponding to π is the implication

$$\varphi_\pi := \psi_\pi \rightarrow \chi_\pi,$$

where the antecedent ψ_π (which we define shortly) serves as a guard that captures the relative positions of the variables along π and the consequent $\chi_\pi = f(q_n)$ is the data formula decorating the final state q_n of π (in the case of a translation into the Array Property Fragment, an overapproximation of $f(q)$ might be necessary).

We define the path guard ψ_π as follows: at each state q_i on the path, we construct *local constraints*, which describe how individual variables encoded in the incoming and outgoing transitions of q_i are related, and collect them in the set C_i; the path guard then is the conjunction

$$\psi_\pi := \bigwedge_{i=1}^{n} \bigwedge_{\psi \in C_i} \psi.$$

For the construction of path guards, we use the following two notations: First, we use the notation $q_{i-1} \xrightarrow{a_i} q_i \in \pi$, respectively $q_{i-1} \xrightarrow{a_i} q_i \xrightarrow{a_{i+1}} q_{i+1} \in \pi$, to denote parts

of the simple path $\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$. Second, we use the input-symbol $a = (\sigma, y) \in \Sigma \times (Y \cup \{-\})$ and the set $(\Sigma \cup \{y\}) \setminus \{-\}$ of all variables (either pointer variables or universally quantified variables) occurring in a interchangeably; for instance, we write $x \in a$ to denote that the variable x occurs in a.

We divide the construction of path guards into two parts: The *auxiliary part* (i.e., Cases 1 and 2 below) covers the beginning of the path where pointer variables occur together with auxiliary variables, such as *nil*; recall that our encoding of Sect. 7.1 asserts that auxiliary variables occur always in the beginning of valuation words (and, correspondingly, in simple paths). The *data structure part* (i.e., Cases 3 to 6 below) deals with the remainder of the path, which is related to the actual data structure. The local constraints at state q_i are constructed according to the following (nonexclusive) case distinction:

Case 1 $q_{i-1} \xrightarrow{a_i} q_i \in \pi$ and $a_i \cap Aux \neq \emptyset$
Let $z \in a_i \cap Aux$ be the unique auxiliary variable.

- If z models a scalar variable, we set $C_i \leftarrow C_i \cup \{x = z\}$ for all $x \in a_i \setminus \{z\}$. (This case covers the second assumed structural property of EQDAs, described on Page 24. Note that x can only be a universally quantified variable and the state q_n of the simple path is labeled with the data formula *true*.)
- If $z = nil$, we set $C_i \leftarrow C_i \cup \{x = nil\}$ for all $x \in a_i \setminus \{z\}$.
- If $z = index\_le\_size$, we set $C_i \leftarrow C_i \cup \{x < 0\}$ for all $x \in a_i \setminus \{z\}$.
- If $z = index\_geq\_size$, we set $C_i \leftarrow C_i \cup \{x \geq size\}$ for all $x \in a_i \setminus \{z\}$.

Case 2 $q_{i-1} \xrightarrow{a_i} q_i \xrightarrow{a_{i+1}} q_{i+1} \in \pi$, $a_i \cap Aux \neq \emptyset$, and $a_{i+1} \cap Aux = \emptyset$
This case covers the boundary between the auxiliary and the data structure part of a simple path (i.e., processing the actual data structure starts at q_i). Here, we distinguish two cases:

- If $\delta(q_i, \underline{b})$ is undefined, we set $C_i \leftarrow C_i \cup \{x = 0\}$ for all $x \in a_{i+1}$ in the case of arrays, respectively $C_i \leftarrow C_i \cup \{x = head\}$ for all $x \in a_{i+1}$ in the case of lists.
- If $\delta(q_i, \underline{b}) = q_i$, we set $C_i \leftarrow C_i \cup \{0 \leq x\}$ for all $x \in a_{i+1}$ in the case of arrays, respectively $C_i \leftarrow C_i \cup \{head \rightarrow^* x\}$ for all $x \in a_{i+1}$ in the case of lists.

Cases 3 to 6 below only apply if no auxiliary variables occur in the incoming or outgoing transitions. Note that such situations indeed occur since we assume that Y contains at least one variable (which occurs on every simple path after all auxiliary variables).

Case 3 $q_{i-1} \xrightarrow{a_i} q_i \in \pi$
For all $x, x' \in a_i$ with $x \neq x'$, we set $C_i \leftarrow C_i \cup \{x = x'\}$.

Case 4 $q_{i-1} \xrightarrow{a_i} q_i \xrightarrow{a_{i+1}} q_{i+1} \in \pi$ and $\delta(q_i, \underline{b}) = q_i$
Let $x_1 \in a_i$ and $x_2 \in a_{i+1}$. In the case of arrays, we consider two cases:

- If $x_1 \notin Y$ or $x_2 \notin Y$, then we set $C_i \leftarrow C_i \cup \{x_1 < x_2\}$.
- If $x_1 \in Y$, $x_2 \in Y$, and $(a_i \cup a_{i+1}) \cap \Sigma = \emptyset$ (i.e., only universally quantified variables occur), then the Array Property Fragment forbids two adjacent universally quantified variables to be related by the relation $<$; in this case, we set $C_i \leftarrow C_i \cup \{x_1 \leq x_2\}$ and $\chi_\pi \leftarrow \chi_\pi \vee (d(x_1) = d(x_2))$. At this point, the translation does not capture the exact semantics of the EQDA (we comment on this shortly). Note that $\leq$ is an elastic relation.

In the case of lists, we set $C_i \leftarrow C_i \cup \{x_1 \rightarrow^+ x_2\}$ where $\rightarrow^+$ is the transitive closure of the successor relation $\rightarrow$. Note that $\rightarrow^+$ is an elastic relation.

Case 5 $q_{i-1} \xrightarrow{a_i} q_i \xrightarrow{a_{i+1}} q_{i+1} \in \pi$ and $\delta(q_i, \underline{b})$ is undefined
Let $x_1 \in a_i$ and $x_2 \in a_{i+1}$. We distinguish two cases:

- Let $x_1 \notin Y$ or $x_2 \notin Y$. In the case of arrays, we set $C_i \leftarrow C_i \cup \{x_2 = x_1 + 1\}$. In the case of lists, we set $C_i \leftarrow C_i \cup \{x_1 \rightarrow x_2\}$.
- Let $x_1 \in Y$ and $x_2 \in Y$. Since both STRAND and the Array Property Fragment forbid expressing that two universally quantified variables are a fixed distance away, we express their relation indirectly: we identify a state q on the path π that is closest to q_i (the direction is not important) and has a transition containing a pointer variable $p \in PV$ (if $PV = \emptyset$, we use *head* or *tail*). Since $\mathcal{A}$ does not contain any irrelevant self-loops, the subpath from q_i to q has no self-loops. Thus, we can constrain the universally quantified variables at q_i to be a fixed distance away from the pointer variable p. For a translation into the Array Property Fragment, we achieve this using arithmetic on the pointer variables. For a translation into the decidable syntactic fragment of STRAND, we obtain the same effect by existentially quantifying monadic predicates $x_1, x_2, \cdots, x_d$ that track the distance d from the universally quantified variable y at q_i to the pointer variable p as follows: $succ(y, x_1) \wedge succ(x_1, x_2) \wedge \cdots \wedge succ(x_{d-1}, x_d) \wedge x_d = p$. Since the distance d between q and q_i is bounded, a finite number of such predicates suffices.

Case 6 $q_{n-1} \xrightarrow{a_n} q_n \in \pi$
In this case, q_n is the last state of π and $\delta(q_{n-1}, a_n) = q_n$ the last transition. We distinguish two cases:

- If $\delta(q_n, \underline{b})$ is undefined, we set $C_n \leftarrow C_n \cup \{x = size - 1\}$ for all $x \in a_n$ in the case of arrays, respectively $C_n \leftarrow C_n \cup \{x = tail\}$ for all $x \in a_n$ in the case of lists.
- If $\delta(q_n, \underline{b}) = q_n$, we set $C_n \leftarrow C_n \cup \{x < size\}$ for all $x \in a_n$ in the case of arrays, respectively $C_n \leftarrow C_n \cup \{x \rightarrow^* tail\}$ for all $x \in a_n$ in the case of lists.

Since the Array Property Fragment lacks the ability to check whether two universally quantified variables are different, Case 4 needs to introduce an overapproximation of the real constraints along a simple path if two universally quantified variables, say y and y', are adjacent at a state with a self-loop on $\underline{b}$ (i.e., the path guard is incorrectly satisfied even if $y = y'$ holds). In order to compensate for this, we amend the formula χ_π by disjointly adding the constraint $d(y) = d(y')$, which ensures that the path formula is satisfied if $y = y'$ holds (since $y = y'$ implies $d(y) = d(y')$). This way, the path formula checks the structural and data constraints of the path if the valuation satisfies $y_1 < \cdots < y_k$, but also when universally quantified variables are equal (which cannot be checked by an EQDA due to fact that the input alphabet of EQDAs requires universally quantified variables to be at different position). Note that a path formula with such an approximation is imprecise in general.

The complete translation functions as follows: It collects the sets C_i along every simple path $\pi \in P_\mathcal{A}$ and constructs the formulas ψ_π and χ_π. For a translation into the decidable syntactic fragment of STRAND, it returns the formula

$$\varphi_\mathcal{A} := \forall y_1 \colon \ldots \forall y_k \colon \Bigg[\underbrace{\Big(\bigwedge_{\pi \in P_\mathcal{A}} \psi_\pi \rightarrow \chi_\pi \Big)}_{\varphi_{sp}} \wedge \underbrace{\Big(\Big[(head \rightarrow^* y_1 \rightarrow^+ \cdots \rightarrow^+ y_k \rightarrow^* tail) \wedge \neg \Big(\bigvee_{\pi \in P_\mathcal{A}} \psi_\pi \Big) \Big] \rightarrow false \Big)}_{\varphi_{\neg sp}} \Bigg];$$

For a translation into the Array Property Fragment, it returns

$$\varphi_{\mathcal{A}} := \forall y_1 : \ldots \forall y_k : \Bigg[\underbrace{\Big(\bigwedge_{\pi \in P_{\mathcal{A}}} \psi_\pi \rightarrow \chi_\pi \Big)}_{\varphi_{sp}} \wedge$$

$$\underbrace{\Bigg(\Big[(0 \leq y_1 \leq \cdots \leq y_k < size) \wedge \neg \Big(\bigvee_{\pi \in P_{\mathcal{A}}} \psi_\pi \Big) \Big] \rightarrow \bigvee_{\substack{y, y' \in Y, \\ y \neq y'}} d(y) = d(y') \Bigg)}_{\varphi_{\neg sp}} \Bigg].$$

The subformula φ_{sp} is the conjunction of all path formulas whereas the subformula $\varphi_{\neg sp}$ captures valuation words that have the right ordering of the universally quantified variables but do not admit a run of $\mathcal{A}$ (i.e., that are rejected by $\mathcal{A}$). As in the case of path formulas, the Array Property Fragment formula $\varphi_{\neg sp}$ only approximates the correct semantics of $\mathcal{A}$. Again, the disjunction constituting the consequent compensates for the necessary overapproximation in the antecedent ($y_1 \leq \cdots \leq y_k$ instead of $y_1 < \cdots < y_k$).

Since the decidable syntactic fragment of STRAND allows negating atomic formulas, $\varphi_{\mathcal{A}}$ is in this fragment. Though the Array Property Fragment also allows negation over atomic formulas that relate two pointer variables or a pointer variable and a universally quantified variable, negation of an atomic formula of the form $y \leq y'$ is not allowed [7]. However, since we assume both a fixed variable ordering on Y along simple paths and that all other paths with a different ordering lead to the formula *true*, we can remove formulas of the form $\neg(y \leq y')$ from $\neg\big(\bigvee_{\pi \in P_{\mathcal{A}}} \psi_\pi\big)$; as before, considering a different ordering of the variables in Y is not necessary because these variables are universally quantified. After removing such subformulas, the formula $\varphi_{\mathcal{A}}$ falls into the Array Property Fragment.

When we apply our translation to an EQDA to obtain a formula in the syntactic decidable fragment of STRAND over lists, the obtained formula exactly characterizes the set of program configurations that correspond to the language of data words accepted by the given EQDA. However, due to the necessary abstractions introduced by our translation into the Array Property Fragment, the formula obtained from translating the EQDA over arrays might not characterize the semantics of the given EQDA exactly. However, we can at least assert that all data words accepted by this EQDA correspond to a program configuration satisfying the formula.

To make this intuition precise , let us introduce the following notations: Given a program configuration c, let (c) denote the natural translation of c into an interpretation for formulas in the Array Property Fragment, respectively in the decidable syntactic fragment of STRAND. [2] Moreover, let $(c, y_1, \ldots, y_k)$ denote the interpretation (c) in which the universally quantified variables are fixed to the values $y_1, \ldots, y_k$.

The following theorem now summarizes the main result of our translation.

Theorem 5 *Let $\mathcal{A}$ be an EQDA, c a program configuration, u_c the data word corresponding to c, and $\varphi_{\mathcal{A}}$ the formula obtained after the translation (either in the decidable syntactic fragment of* STRAND *or the Array Property Fragment).*

(a) *For a translation into the decidable syntactic fragment of* STRAND*, the equivalence*

$$u_c \in L_{dat}(\mathcal{A}) \; if \; and \; only \; if \; (c) \models \varphi_{\mathcal{A}}$$

[2] We make sure that the type of an interpretation (i.e., whether it is for formulas in the Array Property Fragment or in the decidable syntactic fragment of STRAND) is always clear from the context.

holds.

(b) *For a translation into the Array Property Fragment, the implication*

$$u_c \in L_{dat}(\mathcal{A}) \text{ implies } (c) \models \varphi_{\mathcal{A}}$$

holds.

The abstraction along simple paths with $y < y'$ introduced by our translation is the reason why Theorem 5 only holds in one direction for the Array Property Fragment. For this reason, we first prove Theorem 5 for the translation into the decidable syntactic fragment of STRAND; based on the insight gained in the proof, it becomes much easier to prove Theorem 5 for the translation into the Array Property Fragment.

Decidable syntactic fragment of STRAND The pivotal fact on which Theorem 5 relies is that the path guard ψ_π exactly captures the structural constraints along π. The next lemma formalizes this intuition.

Lemma 1 *Let $\mathcal{A}$ be an EQDA over the finite set PV of pointer variables and the finite, nonempty set Y of universally quantified variables, π a simple path in $\mathcal{A}$, and ψ_π the corresponding path guard in the decidable syntactic fragment of* STRAND. *Moreover, let c be a program configuration, $y_1, \ldots, y_k$ a valuation of Y, and v the valuation word modeling c and $y_1, \ldots, y_k$. Then, the following equivalence holds:*

$$\textit{The unique run of } \mathcal{A} \textit{ on } v \textit{ is along } \quad \pi \quad \textit{if and only if } (c, y_1, \ldots, y_k) \models \psi_\pi.$$

Proof We split the proof into two parts: we first show the direction from left to right and subsequently the reverse direction. The direction from left to right is straightforward and simply exploits the fact we only add such local constraints to a path guard that are obviously satisfied along the given path. The direction from right to left, however, is more elaborate to prove.

From left to right Let

$$\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$$

a simple path in $\mathcal{A}$ and assume that the unique run of $\mathcal{A}$ on v is along π. Since the path guard is the conjunction $\bigwedge_{i=1}^{n} \bigwedge_{\psi \in C_i} \psi$ of all local constraints along π, it is enough to prove that $(c, y_1, \ldots, y_k)$ satisfies each individual local constraint. To this end, let ψ be a local constraint, say constructed at state q_i of π.

In order to show $(c, y_1, \ldots, y_k) \models \psi$, we have to distinguish due to which case of the translation the constraint ψ has been constructed. However, since most cases are similar, we do not give a thorough proof here but exemplary consider Case 4.

If ψ has been introduced in Case 4, then $\psi := x_1 \rightarrow^+ x_2$ with $x_1 \in a_i$ and $x_2 \in a_{i+1}$. Since the run of $\mathcal{A}$ on v is along π, we know that all variables $x \in a_i$ occur before the variables $x' \in a_{i+1}$. Thus, $(c, y_1, \ldots, y_k)$ satisfies $x \rightarrow^+ x'$ for all such x, x'. This in turn means $(c, y_1, \ldots, y_k) \models \psi$.

From right to left Let

$$\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_m} q_m$$

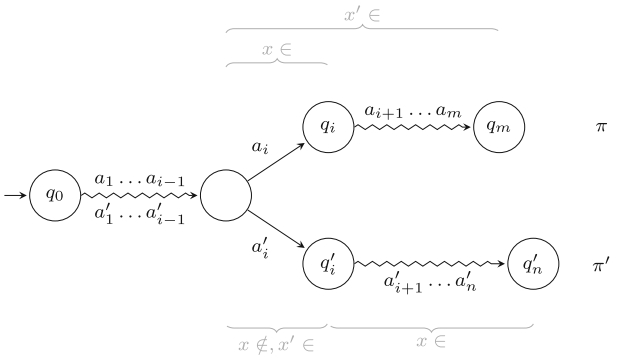


Fig. 4 Two diverging simple paths π, π'

be a simple path in $\mathcal{A}$ and $(c, y_1, \dots, y_k)$ a model of ψ_π. Towards a contradiction, assume that the run of $\mathcal{A}$ on v is along a different simple path, say

$$\pi' = q_0 \xrightarrow{a'_1} q'_1 \xrightarrow{a'_2} \dots \xrightarrow{a'_n} q'_n.$$

Then, there exists a position $i \in \mathbb{N}_+$ at which both paths diverge; that is, $a_j = a'_j$ and $q_j = q'_j$ for all $j \in [i]$, $a_i \neq a'_i$, and $q_i \neq q'_i$. Note that such a position always exists because the states of $\mathcal{A}$ are "typed" (i.e., $\mathcal{A}$ has to remember which variable it has already read). Figure 4 depicts such a situation.

We observe that all input symbols along the paths π and π' are different from $\underline{b}$ because $\mathcal{A}$ is elastic. Thus, if $a_i \neq a'_i$, then there exists a variable $x \in PV \cup Y$ that is missing in exactly one of a_i and a'_i (i.e., $x \in a_i$ if and only if $x \notin a'_i$). Without loss of generality, let us assume $x \in a_i$ and $x \notin a'_i$.

Since $a'_i \neq \underline{b}$, there also exists a variable $x' \in a'_i$ that is different from x. Moreover, since π' is a simple path (which implies that all pointer and universally quantified variables occur exactly once), the variable x also occurs in π', but only in one of the inputs $a'_{i+1}, \dots, a'_n$; note that x' might or might not occur together with x on π.

We now distinguish two cases:

1. An auxiliary variable, say z, occurs in the input-symbol a_i on π; that is, q_i belongs to the auxiliary part of π. We first observe that x cannot be an auxiliary variable because we assume that auxiliary variables appear never together and always in the same, fixed order. Thus, the following two cases remain:

   (a) The variable x occurs on π' together with an auxiliary variable, say z', that is different from z. Since we assume the run of $\mathcal{A}$ on v to be along π', this means $(c, y_1, \dots, y_k) \models x = z'$. Consequently, $(c, y_1, \dots, y_k) \not\models x = z$ because $x = z \wedge x = z'$ is unsatisfiable if $z \neq z'$. However, the path guard ψ_π contains the local constraint $x = z$ (see Case 1). Thus, $(c, y_1, \dots, y_k) \not\models \psi_\pi$, which yields a contradiction.

   (b) The variable x does not occur together with an auxiliary variable on π'. Since we assume the run of $\mathcal{A}$ on v to be along π', this means $(c, y_1, \dots, y_k) \models \mathit{head} \rightarrow^* x$. Consequently, $(c, y_1, \dots, y_k) \not\models x = z$ because z is an auxiliary variable that occurs before *head*. However, the path guard ψ_π contains the local constraint $x = z$ (again, see Case 1). Thus, $(c, y_1, \dots, y_k) \not\models \psi_\pi$, which yields a contradiction.

2. The input-symbol a_i on π does not contain an auxiliary variable; that is, q_i belongs to the data structure part of π. Since we assume the run of $\mathcal{A}$ on v to be along π', the variable x' points to a cell that is located before the cell pointed to by x. Hence, $(c, y_1, \ldots, y_k) \models x' \rightarrow^+ x$. Consequently, $(c, y_1, \ldots, y_k) \not\models x \rightarrow^* x'$ because $x \rightarrow^* x' \wedge x' \rightarrow^+ x$ is unsatisfiable. However, the path guard ψ_π implies $x \rightarrow^* x'$ (see Cases 4 and 5) although it might not contain this subformula explicitly. Thus, $(c, y_1, \ldots, y_k) \not\models \psi_\pi$, which yields the desired contradiction. □

Using Lemma 1, we can now prove Part (a) of Theorem 5.

Proof (of Theorem 5(a)) Let $\mathcal{A}$ be an EQDA over PV and Y, $y_1 \prec \cdots \prec y_k$ the predetermined order in which the universally quantified variables have to occur in the input of $\mathcal{A}$, and $\varphi_\mathcal{A}$ the formula in the decidable syntactic fragment of STRAND resulting from our translation. In addition, let c be a program configuration and u_c the data word modeling c.

We first show the direction from left to right (i.e., $u_c \in L_{dat}(\mathcal{A})$ implies $(c) \models \varphi_\mathcal{A}$) and subsequently the reverse direction (i.e., $(c) \models \varphi_\mathcal{A}$ implies $u_c \in L_{dat}(\mathcal{A})$).

From left to right Let $u_c \in L_{dat}(\mathcal{A})$. In order to prove that the interpretation (c) satisfies $\varphi_\mathcal{A} := \forall y_1 \colon \ldots \forall y_k \colon (\varphi_{sp} \wedge \varphi_{\neg sp})$, we fix an arbitrary valuation $y_1, \ldots, y_k$ of Y and show

$$(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}.$$

In the case that $head \rightarrow^* y_1 \rightarrow^+ \cdots \rightarrow^+ y_k \rightarrow^* tail$ does not hold, we first observe that $(c, y_1, \ldots, y_k)$ does not satisfy any path guard because each path guard implies $head \rightarrow^* y_1 \rightarrow^+ \cdots \rightarrow^+ y_k \rightarrow^* tail$. Hence, $(c, y_1, \ldots, y_k) \models \varphi_{sp}$ since the antecedent of each path formula is unsatisfied. Moreover, $(c, y_1, \ldots, y_k)$ does not satisfy the antecedent of $\varphi_{\neg sp}$ and, consequently, $(c, y_1, \ldots, y_k) \models \varphi_{\neg sp}$. Thus, $(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

In the case that $head \rightarrow^* y_1 \rightarrow^+ \cdots \rightarrow^+ y_k \rightarrow^* tail$ holds, let v be the valuation word resulting from extending u_c with the valuation $y_1, \ldots, y_k$ (which implies $\mathrm{dw}(v) = u_c$). We proceed the proof by first showing that $(c, y_1, \ldots, y_k)$ satisfies φ_{sp} and subsequently that it satisfies $\varphi_{\neg sp}$.

1. Since $u_c \in L_{dat}(\mathcal{A})$, the valuation word v is also accepted by $\mathcal{A}$, say along the simple path π. This particularly means that the unique run of $\mathcal{A}$ on v ends in a configuration (q, r) with $r \models f(q)$. By Lemma 1, we know $(c, y_1, \ldots, y_k) \models \psi_\pi$. Moreover, since $f(q) = \chi_\pi$ and $r \models f(q)$, we also know $(c, y_1, \ldots, y_k) \models \chi_\pi$ and, thus, $(c, y_1, \ldots, y_k) \models \psi_\pi \rightarrow \chi_\pi$. On the other hand, Lemma 1 asserts that no other path guard is satisfied by $(c, y_1, \ldots, y_k)$. Thus, $(c, y_1, \ldots, y_k) \models \varphi_{sp}$.
2. The fact that $(c, y_1, \ldots, y_k) \models \psi_\pi$ holds (see above) implies $(c, y_1, \ldots, y_k) \not\models \neg(\bigvee_{\pi \in P_\mathcal{A}} \psi_\pi)$. Hence, the antecedent of $\varphi_{\neg sp}$ is not satisfied and, therefore, $(c, y_1, \ldots, y_k) \models \varphi_{\neg sp}$.

Thus, $(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

In total, $u_c \in L_{dat}(\mathcal{A})$ implies $(c) \models \varphi_\mathcal{A}$.

From right to left Let u_c be a data word with $u_c \notin L_{dat}(\mathcal{A})$ and c the corresponding program configuration. We need to show that c does not satisfy $\varphi_\mathcal{A}$.

Since $u_c \notin L_{dat}(\mathcal{A})$, there exists a valuation $y_1, \ldots, y_k$ and a corresponding valuation word v (i.e., u_c extended by $y_1, \ldots, y_k$ results in v) such that $v \notin L_{val}(\mathcal{A})$. This valuation word is rejected either

1. due to a missing transition; or
2. due to the fact that the run of $\mathcal{A}$ on v ends in a configuration (q, r) with $r \not\models f(q)$.

In the first case, the run of $\mathcal{A}$ on v does not lead along a simple path. By Lemma 1, this implies $(c, y_1, \ldots, y_k) \not\models \psi_\pi$ for every $\pi \in P_\mathcal{A}$. Hence, $(c, y_1, \ldots, y_k) \models \neg(\bigvee_{\pi \in P_\mathcal{A}} \psi_\pi)$. Since we assume that $\mathcal{A}$ accepts all valuation words that violate the fixed order of the universally quantified variables or where at least one of these variables points to *nil*, we know that $(c, y_1, \ldots, y_k) \models \mathit{head} \rightarrow^* y_1 \rightarrow^+ \cdots \rightarrow^+ y_k \rightarrow^* \mathit{tail}$ holds. Thus, $(c, y_1, \ldots, y_k) \not\models \varphi_{\neg sp}$ and, consequently, $c \not\models \varphi_\mathcal{A}$.

In the second case, the run of $\mathcal{A}$ on v leads along a simple path, say π, ending in the configuration (q, r). By Lemma 1, this implies $(c, y_1, \ldots, y_k) \models \psi_\pi$. However, since $r \not\models f(q) = \chi_\pi$, we have $(c, y_1, \ldots, y_k) \not\models \chi_\pi$. Thus, $(c, y_1, \ldots, y_k) \not\models \varphi_{sp}$ (because $(c, y_1, \ldots, y_k) \not\models \psi_\pi \rightarrow \chi_\pi$) and, consequently, $c \not\models \varphi_\mathcal{A}$.

In total, $u_c \notin L_{dat}(\mathcal{A})$ implies $(c) \not\models \varphi_\mathcal{A}$ (i.e., $(c) \models \varphi_\mathcal{A}$ implies $u_c \in L_{dat}(\mathcal{A})$). □

Array Property Fragment The approximation in Case 4 of our translation is the reason why Theorem 5 holds only in one direction in the case of a translation into the Array Property Fragment. In order to prove this direction, we first show that the path guard ψ_π overapproximates the structural constraints of π. The next lemma formalizes this.

Lemma 2 *Let $\mathcal{A}$ be an EQDA over the finite set PV of pointer variables and the finite, nonempty set Y of universally quantified variables, π a simple path in $\mathcal{A}$, and ψ_π the corresponding path guard in the Array Property Fragment. Moreover, let c be a program configuration, $y_1, \ldots, y_k$ a valuation of Y, and v the valuation word modeling c and $y_1, \ldots, y_k$. Then, the following implication holds:*

$$\textit{if the unique run of } \mathcal{A} \textit{ on } v \textit{ is along } \pi, \quad \textit{then } (c, y_1, \ldots, y_k) \models \psi_\pi.$$

Proof One can prove Lemma 2 in the same way as Lemma 1 (see Page 29): again, we consider each local constraint ψ of a path guard individually and show $(c, y_1, \ldots, y_k) \models \psi$. In fact, we can reuse the proof of Lemma 1 except for a slightly different treatment of Case 4, which we sketch below.

Assume that ψ has been added at state q_i of the simple path $\pi = q_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} q_n$, and let $x_1 \in a_i$ and $x_2 \in a_{i+1}$.

If $x_1 \notin Y$ or $x_2 \notin Y$, then this situation matches Case 4 of the proof of Lemma 1 and immediately yields the desired result.

If $x_1 \in Y$, $x_2 \in Y$, and both variables do not occur together with a pointer variable, then the translation adds $\psi := x_1 \leq x_2$ instead of the "correct" constraint $x_1 < x_2$. However, we know that all variables $x \in a_i$ occur before the variables $x' \in a_{i+1}$ because the run of $\mathcal{A}$ on v is along π. Thus, $(c, y_1, \ldots, y_k) \models x < x'$ for all such x, x', which implies $(c, y_1, \ldots, y_k) \models x_1 \leq x_2$ (i.e., $(c, y_1, \ldots, y_k) \models \psi$). □

We can now prove Part (b) of Theorem 5.

Proof (of Theorem 5(b)) Let $\mathcal{A}$ be an EQDA over PV and Y, $y_1 \prec \cdots \prec y_k$ the predetermined order in which the universally quantified variables have to occur in the input of $\mathcal{A}$, and $\varphi_\mathcal{A}$ the formula in the Array Property Fragment resulting from our translation. Moreover, let c be a program configuration and u_c the data word modeling c. Finally, assume $u_c \in L_{dat}(\mathcal{A})$.

We have to show that (c) is a model of $\varphi_{\mathcal{A}}$. This proof is similar to the direction from left to right of the proof of Theorem 5(a): we again fix an arbitrary valuation $y_1, \ldots, y_k$ of Y and show

$$(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}.$$

In the case that $0 \leq y_1 \leq \cdots \leq y_k < \mathit{size}$ does not hold, we again observe that $(c, y_1, \ldots, y_k)$ does not satisfy any path guard because each path guard implies $0 \leq y_1 \leq \cdots \leq y_k < \mathit{size}$. Hence, $(c, y_1, \ldots, y_k) \models \varphi_{sp}$ since the antecedent of each path formula is unsatisfied. Moreover, $(c, y_1, \ldots, y_k)$ does not satisfy the antecedent of $\varphi_{\neg sp}$ and, consequently, $(c, y_1, \ldots, y_k) \models \varphi_{\neg sp}$. Thus, $(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

In the case that $0 \leq y_1 \leq \cdots \leq y_k < \mathit{size}$ holds, we distinguish two cases:

1. All universally quantified variables are different; that is, $y_i \neq y_j$ holds for all $i, j \in \{1, \ldots, k\}$ with $i \neq j$. In this case, let v be the valuation word resulting from extending u_c with the valuation $y_1, \ldots, y_k$. We proceed the proof by first showing that $(c, y_1, \ldots, y_k)$ satisfies φ_{sp} and subsequently that it satisfies $\varphi_{\neg sp}$.

   (a) Since $u_c \in L_{dat}(\mathcal{A})$, the valuation word v is also accepted by $\mathcal{A}$, say along the simple path π. By Lemma 2, we know that then $(c, y_1, \ldots, y_k) \models \psi_\pi$ holds. Since $v \in L_{val}(\mathcal{A})$, the registers satisfy the data formula of the final state of π. Thus, $(c, y_1, \ldots, y_k) \models \chi_\pi$ and, consequently, $(c, y_1, \ldots, y_k) \models \psi_\pi \rightarrow \chi_\pi$.

   To complete this case, we argue that there exists no other path $\pi' \in P_{\mathcal{A}}$ with $\pi' \neq \pi$ and $(c, y_1, \ldots, y_k) \models \psi_{\pi'}$. Towards a contradiction, assume the contrary and let π' such a simple path. By using arguments similar to those in the direction from right to left of the proof of Lemma 1, one can show that this can only happen due to an overapproximation of the form $y_i \leq y_j$ (rather than $y_i < y_j$). This, in turn, implies that there exists $i, j \in \{1, \ldots, k\}$ with $i < j$ and $y_i = y_k$, which contradicts the assumption that all universally quantified variables are different.

   In total, $(c, y_1, \ldots, y_k)$ satisfies the path formula of each simple path. Hence, $(c, y_1, \ldots, y_k) \models \varphi_{sp}$.

   (b) Since $u_c \in L_{dat}(\mathcal{A})$, we know that there exists a simple path $\pi \in P_{\mathcal{A}}$ such that $(c, y_1, \ldots, y_k) \models \psi_\pi$ (see above). Thus, $(c, y_1, \ldots, y_k) \not\models \neg(\bigvee_{\pi \in P_{\mathcal{A}}} \psi_\pi)$ because removing subformulas of the form $\neg(y \leq y')$ from a path guard potentially results in more interpretations satisfying it and, thus, less satisfying its negation. This implies $(c, y_1, \ldots, y_k) \models \varphi_{\neg sp}$ since we assume $0 \leq y_1 \leq \cdots \leq y_k < \mathit{size}$.

   Thus, $(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

2. There exist $i, j \in \{1, \ldots, k\}$ such that $i < j$ and $y_i = y_j$. In this case, there might be a simple path $\pi \in P_{\mathcal{A}}$ such that $(c, y_1, \ldots, y_k) \models \psi_\pi$. Since universally quantified variables never occur together on a simple path (due to the choice of the input alphabet of QDAs), $(c, y_1, \ldots, y_k)$ can only satisfy ψ_π due to the overapproximation $y_i \leq y_j$ (rather than $y_i < y_j$) introduced by Case 4 of our translation. This means that the formula χ_π is constructed by taking the disjunction of the formulas $f(q)$ (assuming that q is the final state of π), $d(y_i) = d(y_j)$, and potentially other formulas of the form $d(y) = d(y')$ for $y, y' \in Y$. Thus, $(d(y_i) = d(y_j)) \rightarrow \chi_\pi$. Since $y_i = y_j$, we have $d(y_i) = d(y_j)$ and, hence, $(c, y_1, \ldots, y_k) \models \chi_\pi$. This, in turn, means $(c, y_1, \ldots, y_k) \models \psi_\pi \rightarrow \chi_\pi$. Since these arguments are true for all simple paths $\pi' \in P_{\mathcal{A}}$ for which $(c, y_1, \ldots, y_k) \models \psi_{\pi'}$ holds, $(c, y_1, \ldots, y_k) \models \varphi_{sp}$.

On the other hand, $(c, y_1, \ldots, y_k) \models \varphi_{\neg sp}$ because $(c, y_1, \ldots, y_k)$ satisfies the consequent of $\varphi_{\neg sp}$ due to the equality $y_i = y_j$. Thus, $(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

In total, $u_c \in L_{dat}(\mathcal{A})$ implies $(c) \models \varphi_{\mathcal{A}}$. □

8 A case study on learning invariants of linear data structures

We apply the active learning algorithm for QDAs, described in Sect. 5, 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. In this section, we present the implementation details and the experimental results of our evaluation.

8.1 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 the running time of the algorithm is guaranteed to be polynomial in the size of the learned automaton. We show the efficacy of this passive learning algorithm using experimental evidence later in this section.

8.2 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 and 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 in them 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.

8.3 Implementing the learner

We used the LIBALF library [4] as an implementation of the active learning algorithm [3]. 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 Sect. 6. The result is then converted to a quantified formula over STRAND or the APF and we check if the learned invariant was adequate and inductive. Due to the unavailability of an implementation of the STRAND decision procedure, we checked the inductiveness of the invariants learned over list data-structures by manually inspecting the learned QDAs. For programs over arrays, we checked the inductiveness of the learned invariants by manually generating the verification conditions and validating them using the Z3 solver [30] . In the case of arrays, the APF formula that corresponds to a QDA and presented in Sect. 7 over-approximates the semantics of the EQDA. To obtain better results in the implementation, we used a more precise formula in which $\varphi_{\neg sp}$ is replaced by the formula $[(0 \leq y_1 < \cdots < y_k < size) \wedge \neg(\bigvee_{\pi \in P_{\mathcal{A}}} \psi_\pi)] \rightarrow false$. Although this formula does not fall in the APF, the constraint solver was able to handle it in our experiments.

8.4 Experimental results

We evaluate our approach on a suite of programs for learning invariants and preconditions. Our experimental results are tabulated in Table 1. [3] For every program, we report the number of lines of C code, the number of test inputs and the time (T_{teach}) taken to build the teacher from the samples collected along these test runs. We next report the number of equivalence and membership queries answered by the teacher in the active learning algorithm, the size of the final elastic automata in terms of the number of states, whether the learned QDA required any elastification or not and finally, the time (T_{learn}) taken to learn the QDA.

The first part of the table presents results for learning loop invariants. We first report results for programs manipulating arrays like finding a key in an array, copying and comparing two arrays and simple sorting algorithms over arrays. The *inner* and *outer* suffix in insertion and selection sort corresponds to learning loop-invariants for the inner and the outer loops in those sorting algorithms. We next present results for programs that manipulate lists and includes programs to find a key in a sorted list, insert a key in a sorted list such that the resulting list is sorted, initialize all nodes in a list with the value of a key, return the maximum data value in a list, merge two disjoint sorted lists such that the resulting list is also sorted, partition a list into two lists such that one list consists of elements that satisfy a given predicate and the other list consists of nodes that do not, and an in-place reversal of a sorted list where we check whether the output list is reverse-sorted. The programs bubble-sort, fold-split and quick-sort are taken from [5]. 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

[3] The benchmark suite and the source code of our implementation is available at http://www.cs.uiuc.edu/~madhu/cav13/.

Table 1 Experimental results

Program	LOC	#Test inputs	T_{teach} (s)	# Eq.	# Mem.	# States	Elasti ified?	T_{learn} (s)
Learning loop invariants								
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 method preconditions								
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

require a post-condition. Further more, many white-box tools based on *interpolation*, such as SAFARI [1], cannot handle list-structures, and also cannot handle array-based programs with *quantified* preconditions which precludes verifying the array variants of programs like *list-sorted-find*, *list-sorted-insert*, etc., which we can handle.

Next, in Table 1, we present results for verifying methods or code fragments picked from real-world programs. The methods *lookup_prev* and *add_cachepage* are from the module cachePage in ExpressOS, which is a verified-for-security OS platform for mobile applications [26]. The module cachePage maintains a cache of the recently used disc pages as a priority queue based on a sorted list. Next, the method *sort* is a merge sort implementation and *insert_sorted* is a method for inserting a key into a sorted list. Both these

methods are from the Glib library, 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 [23]. 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. In this particular case, we ran the sort utility on an input text file which called the method sortlines multiple times with different array inputs; formula words obtained from these concrete array configurations as described earlier in this Section form the sample S that the teacher uses to learn an invariant. We also used our learning approach for learning method preconditions, given a test suite; the results for those experiments are presented in the second part of the table. Several methods in our collection of programs have the same method preconditions such as the input argument points to a list or that the input argument points to a sorted list; we only report results in the table for three methods that have different preconditions.

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, the Z3 SMT solver we used was not able to handle such formulas without giving extra triggers, thus suggesting the necessity of the elastification of QDAs. It is important to note that the learned invariants might not always be inductive, even though this situation did not arise in our experiments. Exploring automated test generation techniques such as KLEE [8] to create a more exhaustive test suite that prevents this situation from arising is an interesting direction for future work. In the current setting, if the invariant learned is not inductive or is inadequate we try to learn from an improved test suite by running the program on more test inputs. ICE learning [15] is a new, active learning model in which the teacher answers equivalence queries only and refutes the current invariant hypothesis, if it is not adequate or inductive, by adding a positive, negative or an implication counter-example. The ICE learning algorithms for QDAs are more robust that the one presented here and ensure that the invariants learned are adequate and inductive [15].

Learnt invariants are complex in some programs; for example, Fig. 5 contains a graphical depiction of the invariant EQDA we learned for the program *list-sorted-find*. If we read the rightmost simple path in the EQDA from state q_0 to q_1 to state q_{14}, and then to q_3 and q_9, it handles the case when $head = cur \neq nil$ and $head \rightarrow^+ y_1$ and $y_1 \rightarrow^+ y_2$ and the EQDA asserts that the data at location pointed to by y_1 is less than or equal to the data at y_2. In totality, the EQDA corresponds to the following formula: $head \neq nil \wedge (\forall y_1 y_2 . head \rightarrow^* y_1 \rightarrow^* y_2 \Rightarrow d(y_1) \leq d(y_2)) \wedge ((cur = nil \wedge \forall y_1 . head \rightarrow^* y_1 \Rightarrow d(y_1) < k) \vee (head \rightarrow^* cur \wedge \forall y_1 . head \rightarrow^* y_1 \rightarrow^+ cur \Rightarrow d(y_1) < k))$.

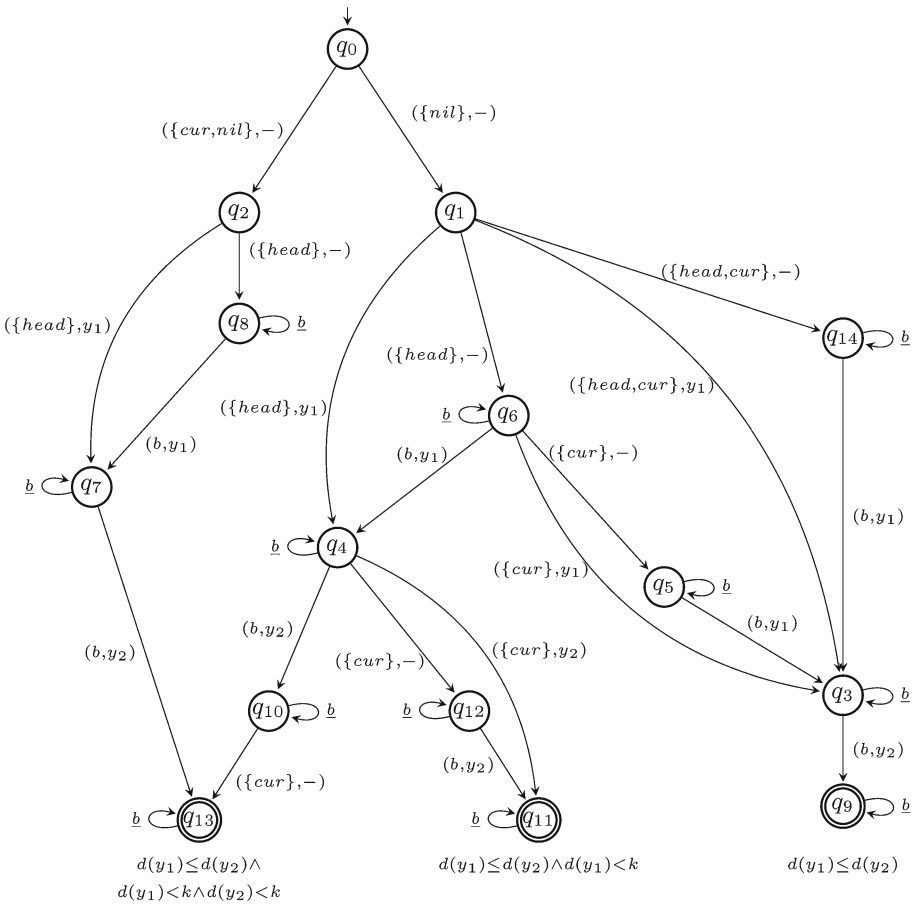


Fig. 5 The learned EQDA that corresponds to the loop invariant of the program *list-sorted-find*

9 Conclusions

We have presented a new automaton model called quantified data automata that can express universally quantified properties of linear data structures, and which can be used to express properties, including invariants, of arrays and lists. We have studied the theory of QDA, defined a subclass called elastic QDA that has decidable emptiness, and built active learning algorithms at the level of formula words. Finally, we have adapted the active learning algorithm for QDAs/EQDAs to learn invariants of programs manipulating arrays and lists, where the decidability of EQDAs and their translation to decidable theories of arrays and lists, yields a verification technique.

EQDA can also be seen as an *abstract domain*, and there has been recent work exploiting this to build an abstract interpretation framework for finding invariants in programs manipulating linear data structures [16].

Neither active learning nor passive learning are robust learning frameworks for synthesizing invariants, since there is no way for the teacher to ensure that the learned invariants are inductive. A new model of learning, called ICE learning, proposes active learning using

examples, counter-examples, and *implication* pairs, with correctness queries only, and is a much more robust model for synthesizing invariants [15]. An ICE learning algorithm for QDAs/EQDAs has also been developed [15].

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 [34], and we would like to explore combining automata-based structural learning (for words and trees) with machine-learning for data-formulas, especially for the ICE learning framework.

Acknowledgments We would like to thank Xiaokang Qiu, who was involved in early discussions on finding an automaton model for the decidable fragment of STRAND. This work was partially supported by NSF CAREER award #0747041 and NSF Expeditions in Computing ExCAPE Award #1138994.

References

1. Alberti F, Bruttomesso R, Ghilardi S, Sharygina N (2012) SMT-based abstraction for arrays with interpolants. In: Madhusudan P, Seshia SA (eds) Computer Aided Verification—24th international conference, CAV 2012, Berkeley, July 7-13, 2012 Proceedings. Lecture notes in computer science, vol 7358. Springer, Berlin, pp 679–685
2. Alur R, Madhusudan P, Nam W (2005) Symbolic compositional verification by learning assumptions. In: CAV, pp 548–562
3. Angluin D (1987) Learning regular sets from queries and counterexamples. Inf Comput 75(2):87–106
4. Bollig B, Katoen JP, Kern C, Leucker M, Neider D (2010) Piegdon, D.R.: libalf: The automata learning framework. In: CAV, Springer, Berlin, pp 360–364
5. Bouajjani A, Dragoi C, Enea C, Sighireanu M (2012) Abstract domains for automated reasoning about list-manipulating programs with infinite data. In: Kuncak V, Rybalchenko A (eds) VMCAI. Lecture notes in computer science, vol 7148. Springer, Berlin, pp 1–22
6. Bradley AR (2011) SAT-based model checking without unrolling. In: Jhala R, Schmidt DA (eds) VMCAI. Lecture notes in computer science, vol 6538. Springer, Berlin, pp 70–87
7. Bradley AR, Manna Z, Sipma HB (2006) What's decidable about arrays? In: Emerson EA, Namjoshi KS (eds) VMCAI. Lecture notes in computer science, vol 3855. Springer, Berlin, pp 427–442
8. Cadar C, Dunbar D, Engler DR (2008) KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves R, van Renesse R (eds.) 8th USENIX symposium on operating systems design and implementation, OSDI 2008, Dec 8–10, USENIX Association, San Diego, pp 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
9. Chen YF, Farzan A, Clarke EM, Tsay YK,Wang BY (2009) Learning minimal separating DFA's for compositional verification. In: Kowalewski S, Philippou A (eds) TACAS. Lecture notes in computer science, vol 5505. Springer, Berlin, pp 31–45
10. Chen YF, Wang BY (2012) Learning boolean functions incrementally. In: In: Madhusudan P, Seshia SA (eds) Computer Aided Verification—24th international conference, CAV 2012, Berkeley, July 7-13, 2012 Proceedings. Lecture notes in computer science, vol 7358. Springer, Berlin, pp 55–70
11. Cobleigh JM, Giannakopoulou D, Pasareanu CS (2003) Learning assumptions for compositional verification. In: Garavel H, Hatcliff J (eds) TACAS. Lecture notes in computer science, vol 2619. Springer, Berlin, pp 331–346
12. Distefano D, O'Hearn PW, Yang H (2006) A local shape analysis based on separation logic. In: Hermanns H, Palsberg J (eds.) Tools and algorithms for the construction and analysis of systems, 12th international conference, TACAS 2006 Held as Part of the joint European conferences on theory and practice of software, ETAPS 2006, Vienna, Austria, Mar 25– Apr 2. Lecture notes in computer science, vol 3920, Springer, Berlin, pp 287–302. Springer doi:10.1007/11691372_19
13. Ernst MD, Czeisler A, Griswold WG, Notkin D (2000) Quickly detecting relevant program invariants. In: ICSE, pp 449–458
14. Flanagan C, Leino KRM (2001) Houdini, an annotation assistant for ESC/Java. In: Oliveira JN, Zave P (eds) FME. Lecture notes in computer science, vol 2021. Springer, Berlin, pp 500–517
15. Garg P, Löding C, Madhusudan P, Neider D (2014) ICE: a robust framework for learning invariants. In: Biere A, Bloem R (eds) CAV. Lecture notes in computer science, vol 8559. Springer, Berlin, pp 69–87

16. Garg P, Madhusudan P, Parlato G (2013) Quantified data automata on skinny trees: An abstract domain for lists. In: Logozzo F, Fähndrich M (eds) SAS. Lecture notes in computer science, vol 7935. Springer, Berlin, pp 172–193
17. Gold EM (1967) Language identification in the limit. Inf Control 10(5):447–474
18. Grädel E, Thomas W, Wilke T (eds) (2002) Automata, logics, and infinite games, vol 2500. Springer, Heidelberg
19. Jhala R, McMillan KL (2007) Array abstractions from proofs. In: Damm W, Hermanns H (eds) CAV. Lecture notes in computer science, vol 4590, Springer, Heidelberg, pp 193–206
20. Kearns MJ, Vazirani UV (1994) An introduction to computational learning theory. MIT Press, Cambridge
21. Khoussainov B, Nerode A (2001) Automata theory and its applications. Birkhauser, Boston
22. Kohavi Z (1970) Switching and finite automata theory. McGraw-Hill, Blacklick
23. Kong S, Jung Y, David C, Wang BY, Yi K (2010) Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In: APLAS, pp 328–343. Springer, Heidelberg
24. Madhusudan P, Parlato G, Qiu X (2011) Decidable logics combining heap structures and data. In: Ball T, Sagiv M (eds.) POPL, pp 611–622. ACM, New York
25. Madhusudan P, Qiu X (2011) Efficient decision procedures for heaps using STRAND. In: Yahav E (ed) SAS. Lecture notes in computer science, vol 6887. Springer, Berlin, pp 43–59
26. Mai H, Pek E, Xue H, King ST, Madhusudan P (2013) Verifying security invariants in ExpressOS. In: V. Sarkar, R. Bodík (eds.) ASPLOS, pp 293–304. ACM, New York
27. McMillan KL (2003) Interpolation and SAT-based model checking. In: Hunt WA Jr., Senzi F (eds.) CAV. Lecture notes in computer science, vol. 2725, Springer, Berlin, pp 1–13
28. McMillan KL (2008) Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan CR, Rehof J (eds) Tools and algorithms for the construction andanalysis of systems. 14th international conference, TACAS 2008, held as part of thejoint European conferences on theory and practice of software, ETAPS 2008, Budapest, Mar 29–Apr 6. Lecture notes in computer science, vol 4963. Springer, Berlin, pp 413–427
29. de Moura LM, Bjørner N, (2007) Efficient E-matching for SMT solvers. In: Pfenning F (ed) CADE. Lecture notes in computer science,vol 4603. Springer, Berlin, pp 183–198
30. de Moura, LM, Bjørner N (2008) Z3: an efficient SMT solver. In: Ramakrishnan CR, Rehof J (eds) Tools and algorithms for the construction andanalysis of systems. 14th international conference, TACAS 2008, held as part of thejoint European conferences on theory and practice of software, ETAPS 2008, Budapest, Mar 29–Apr 6. Lecture notes in computer science, vol 4963. Springer, Berlin, pp 337–340
31. Rivest RL, Schapire RE (1993) Inference of finite automata using homing sequences. Inf Comput 103(2):299–347
32. Sagiv S, Reps TW, Wilhelm R (2002) Parametric shape analysis via 3-valued logic. ACM Trans Progr Lang Syst 24(3):217–298
33. Seghir MN, Podelski A, Wies T (2009) Abstraction refinement for quantified array assertions. In: Palsberg J, Su Z (eds) SAS. Lecture notes in computer science, vol 5673. Springer, Berlin, pp 3–18
34. Sharma R, Nori AV, Aiken A (2012) Interpolants as classifiers. In: Madhusudan P, Seshia SA (eds) Computer Aided Verification—24th international conference, CAV 2012, Berkeley, July 7-13, 2012 Proceedings. Lecture notes in computer science, vol 7358. Springer, Berlin, pp 71–87
35. Srivastava S, Gulwani S (2009) Program verification using templates over predicate abstraction. In: Hind M, Diwan A (eds.) PLDI, pp 223–234. ACM, New York
36. Thomas W (1997) Languages, automata, and logic. In: Rozenberg G, Salomaa A (eds) Handbook of formal language theory, vol III. Springer, Heidelberg, pp 389–455
37. Vardi MY, Wolper P (1986) An automata-theoretic approach to automatic program verification (preliminary report). In: LICS, IEEE Computer Society, pp 332–344