Algorithmic Analysis of Array-Accessing Programs *

Rajeev Alur, Pavol Černý, and Scott Weinstein

University of Pennsylvania
{alur,cernyp,weinstein}@cis.upenn.edu

Abstract. For programs whose data variables range over boolean or finite domains, program verification is decidable, and this forms the basis of recent tools for software model checking. In this paper, we consider algorithmic verification of programs that use boolean variables, and in addition, access a single read-only array whose length is potentially unbounded, and whose elements range over a potentially unbounded data domain. We show that the reachability problem, while undecidable in general, is (1) PSPACE-complete for programs in which the array-accessing `for`-loops are not nested, (2) decidable for a restricted class of programs with doubly-nested loops. The second result establishes connections to automata and logics defining languages over data words.

1 Introduction

Verification questions concerning programs are undecidable in general. However, for finite-state programs — programs whose data variables range over finite types such as boolean, the number of bits needed to encode a program state is a priori bounded, and verification questions such as reachability are decidable. This result, coupled with progress on symbolic techniques for searching the state-space of finite-state programs, and abstraction techniques for extracting boolean over-approximations of general programs, forms the basis of recent tools for software model checking [3, 16].

We focus on algorithmic verification of programs that access a single array. The length of the input array is potentially unbounded. The elements of the array range over $\Sigma \times D$, where Σ is a finite set, and D is a data domain that is potentially unbounded and totally ordered. The array is thus modeled as a *data word*, that is, a sequence of pairs in $\Sigma \times D$. For example an array that contains employees' names, and for each name a tag indicating whether the employee is a programmer, a manager, or a director, can be modeled by setting D to be the set of strings, and Σ to be a set with three elements. The program can have Boolean variables, index variables ranging over array positions, and data variables ranging over D. Programs can access Σ directly, but can only perform equality and order tests on elements of D. The expressions in the program can

* This research was partially supported by NSF Cybertrust award CNS 0524059.

use constants in D, and equality tests and ordering over index and data variables. The programs are built using assignments, conditionals, and `for`-loops over the array. Even with these restrictions, one can perform interesting computational tasks including searching for a specific value, finding the minimum data value, checking that all values in the array are within specific bounds, or checking for duplicate data values. For example, Java midlets designed to enhance features of mobile devices include simple programs accessing the address books, and our methods can lead to an automatic verification tool that certifies their correctness before being downloaded. For programs that fall outside the restrictions mentioned above, it is possible to use abstract interpretation techniques such as predicate abstraction [13] to abstract some of the features of the program, and analyze the property of interest on the abstract program. As the abstract programs are nondeterministic, we will consider nondeterministic programs.

Our first result is that the reachability problem for programs in which there are no *nested* loops is decidable. The construction is by mapping such a program to a finite-state abstract transition system such that every finite path in the abstract system is feasible in the original program for an appropriately chosen array. We show that the reachability problem for programs with non-nested loops is PSPACE-complete, which is the same complexity as that for finite-state programs with only boolean variables.

Our second result shows decidability of reachability for programs with doubly-nested loops with some restrictions on the allowed expressions. The resulting complexity is non-elementary, and the interest is mainly due to the theoretical connections with the recently well-studied notions of automata and logics over *data words* [6, 5, 17]. Among different kinds of automata over data words that have been studied, *data automata* [6] emerged as a good candidate definition for the notion of regularity for languages on data words. A data automaton first rewrites the Σ-component to another finite alphabet Γ using a nondeterministic finite-state transducer, and then checks, for every data value d, whether the projection obtained by deleting all the positions in which the data value is not equal to d, belongs to a regular language over Γ. In order to show decidability of the reachability problem for programs with doubly nested loops, we extend this definition as follows: An *extended data automaton* first rewrites the data word as in case of data automata. For every data value d, the corresponding projection contains more information than in case of data automata. It is obtained by replacing each position with data value different from d by the special symbol 0. The projection is required to be in a regular language over $\Gamma \cup \{0\}$. We prove that the reachability problem for extended data automata can be reduced to emptiness of multi-counter automata (or equivalently, to Petri nets reachability), and is thus decidable. We then show that a program containing doubly-nested loops can be simulated, under some restrictions, by an extended data automaton. Relaxing these restrictions leads to undecidability of the reachability problem for programs with doubly-nested loops.

Analyzing reachability problem for programs brings a new dimension to investigations on logics and automata on data words. We establish some new

connections, in terms of expressiveness and decidability boundaries, between programs, logics, and automata over data words. Bojańczyk et al. [6] consider logics on data words that use two binary predicates on positions of the word: (1) an equivalence relation $\approx$, such that $i \approx j$ if the data values at positions i and j are equal, and (2) an order $\prec$ which gives access to order on data values, in addition to standard successor ($+1$) and order $<$ predicates over the positions. They show that while the first order logic with two variables, $\mathrm{FO}^2(\approx, <, +1)$, is decidable, introducing order on data values causes undecidability, that is, $\mathrm{FO}^2(\approx, \prec, <, +1)$ is undecidable. In this context, our result on programs with non-nested loops is perhaps surprising, as we show that the undecidability does not carry over to these programs, even though they access order on the data domain and have an arbitrary number of index and data variables.

Details of proofs are available in the companion report [1].

2 Programs

In this section, we define the syntax and semantics of programs that we will consider. We start by defining arrays. Let D be an infinite set of data values. We will consider domains D equipped with equality $(D, =)$, or with both equality and linear order $(D, =, <)$. Let Σ be a finite set of symbols. An array is a data word $w \in (\Sigma \times D)^*$. The program can access the elements of the array via indices into the array.

Syntax. The programs have one array variable `A`. Variables `b`, `b1`, `b2`, ... are boolean. Variables `p`, `p1`, `p2`, ... range over $\mathbb{N}$, and are called index variables. Variables `i`, `j`, `i1`, `i2`, ... range over $\mathbb{N}$ and are called loop variables. Variables `v`, `v1`, `v2`, ... range over D and are called data variables. Constants `c`, `c1`, `c2`, ... are in D, and constants `s`, `s1`, `s2`, ... are in Σ. We make a distinction between loop and index variables because loop variables cannot be modified outside of the loop header. Index expressions `IE` are defined by the following grammar `IE ::= p | i`. Data expressions `DE` are of the form `DE :: = v | c | A[IE].d`, where `A[IE].d` accesses the data part of the array. Σ-expressions `SE` are of the form `SE :: = s | A[IE].s`, where `A[IE].s` accesses the Σ part of the array. Boolean expressions are defined by the following grammar: `B :: = true | false | b | B and B | not B | IE = IE | IE < IE | DE = DE | DE < DE | SE = SE`. The programs are defined by the grammar:

```
P :: = skip | { P } | b:=B | p:=IE | v:=DE
       | if B then P else P | if * then P else P
       | for i:=1 to length(A) do P | P;P
```

The commands include a nondeterministic conditional. We consider nondeterministic programs in this paper, in order to enable modeling of abstracted programs. Software model checking approaches [13, 3, 16] often rely on predicate abstraction. For example, if the original program contains an assignment of the form `b := E`, where `E` is a complicated expression that falls out of scope of the intended analysis, the assignment is abstracted into a nondeterministic assignment to `b`. This is modeled as `if * then b:=true else b:=false` in the language presented here.

Semantics. A *global state* of the program is a valuation of its boolean, loop, index and data variables, as well as of the array variable. We denote global states by g, g_1, and the set of global states by G. For a boolean, index, loop or data variable $\mathtt{v}$, we denote the value of $\mathtt{v}$ by $g[\mathtt{v}]$. The value of the array variable A is a word $w \in (\Sigma \times D)^*$. It is denoted by $g[\mathtt{A}]$. The length of the array at global state g is denoted by $l(g[\mathtt{A}])$ and evaluates to the length of w. Note that the length and the contents of the array do not change over the course of the computation.

Semantics of boolean, index, data and Σ expressions is a partial function: $[\![\mathtt{B}]\!] : G \to \mathbb{B}$, $[\![\mathtt{IE}]\!] : G \to \mathbb{N}$, $[\![\mathtt{DE}]\!] : G \to D$ and $[\![\mathtt{SE}]\!] : G \to \Sigma$. It is not defined only in cases when there is an array access out of bounds. For example, in a state g where $g[\mathtt{A}]$ is a word of length 10 and $g[\mathtt{p}]$ is 20, the semantics of the expression `A[p].d` is undefined. The semantics of commands is defined as a relation on G, $[\![\mathtt{P}]\!] \subseteq G \times G$. The definition is presented in detail in [1].

Given a program, a global state is *initial* if either i) the array variable contains a nonempty word, all boolean variables are set to *false*, all index and loop variables are set to 1, and all data variables are set to the same value as the first element of the array; or ii) the array variable contains an empty word, all boolean variables are set to *false*, all index and loop variables are set to 1, and all data variables are set to constant $c_D \in D$. The intention is that the only unspecified part of the initial state, the part that models input of the program, is the array. A boolean state is a valuation of all the boolean variables of a program. For a given global state g, we denote the corresponding boolean state by $bool(g)$. For any boolean variable b of the program, we have that $bool(g)[\mathtt{b}] = g[\mathtt{b}]$. We denote boolean states by m, m_1 and the set of boolean states by M.

Restricted fragments. We classify programs using the nesting depth of loops. We denote programs with only non-nested loops by `ND1`, programs with nesting depth at most 2 by `ND2`, etc. Restricted-`ND2` programs are programs with nesting depth at most 2, that do not use index or data variables, and do not refer to order on data or indices. Furthermore, a key restriction, such that if it is lifted, the reachability problem becomes undecidable, is a restriction on the syntax of the code inside the inner loop. Let `P1` be the code inside an inner loop, and let `i` be the loop variable of the outer loop and let `j` be the loop variable for the inner loop. `P1` must be of the following form: `if A[i].d=A[j].d then P2 else P3`. Furthermore, `P3` cannot refer to `A[j]`, i.e. it does not contain occurrences of `A[j].d` or `A[j].s`.

Examples. We present three examples illustrating these classes of programs.

Example 1. We consider a simple array accessing program `Min` that scans through an array to find a minimal data value. It has one index variable, `p`, and it is an `ND1` program, as it does not contain nested loops:

```
for i:= 1 to length(A) do { if A[i].d < A[p].d then p := i }
```

Note that by definition of program semantics, `p` is initialized to 1. The correctness requirement for this program is that the index `p` points to a minimal element, that is $\forall$ `i: A[i].d` $\geq$ `A[p].d`. Verifying the correctness of the program can be reduced to checking reachability, as the requirement itself can be expressed as a program, by appending to the program `Min` above the following:

```
b:=true;
for i:= 1 to length(A) do {
  if A[i].d < v then b:=false
                else skip;
  v := A[i].d
}
```

Fig. 1. Example 2

```
b:=false;
for i:= 1 to length(A) do {
  for j:= 1 to length(A) do
    if (A[i].d = A[j].d) then {
       if (not (i = j)) then b:=true
                        else skip
    } else skip
}
```

Fig. 2. Example 3

```
b:=true;
for i:= 1 to length(A) do { if A[i].d < A[p].d then b:=false }
```

We can now ask a reachability question: Does the control reach the end of the program in a state where `b == false` holds?

Example 2. Figure 1 shows an `ND1` program that tests whether the array is sorted. It uses one data variable called `v` (note that by definition of the semantics, `v` is initialized to the same value as the first element of the array).

Example 3. The Restricted-`ND2` program in Figure 2 tests whether there is a data value that appears twice in the array.

3 Reachability

Given a program `P`, a boolean state m is *reachable* if and only if there exists an initial global state g_I and a global state g such that $(g_I, g) \in [\![\texttt{P}]\!]$ and $bool(g) = m$. The reachability problem is to determine, for a given program `P` and a given boolean state m, whether m is reachable. We will use a notion of a *local state*. Given a program, a local state is a valuation of all its boolean, index, loop, and data variables, as well as the values of array elements corresponding to index and loop variables. For each index and loop variable `v`, local states have an additional variable `A_v` that stores the value of the array element at position given by `v`. The main difference between local and global states of a program `P` is that local states do not contain valuation of the array, they store instead at most a fixed finite number k_P of values from the unbounded domain D, where k_P is bounded by the total number of index and loop variables occurring in `P`.

Theorem 1. *Reachability for* `ND1` *programs is decidable. The problem is* PSPACE*-complete.*

The structure of the proof is as follows. We first characterize the semantics of a program in terms of a transition system T whose states are (tuples of) local states. Let us first consider the following simple program `P`: `for i1:=1 to length(A) do P1`. Here, and in the rest of the proof, we assume that the length of the array is non-zero. (In the case the length of the array is zero, the program effectively contains no loops, and reachability can be computed in time polynomial in number of variables.) The program `P` can be seen as a transition system whose states are local states of `P` and which processes an input

word in $\Sigma \times D$, with each iteration consuming one symbol of the word. For sequential composition of commands, a product construction (augmented with some bookkeeping) is used.

Note that T is still an infinite-state system, as its states store values from D. Therefore, we construct a finite state system T^α that abstracts the infinite part of the local states, that is the values of index, loop and data variables. The abstract state transition system T^α keeps only order and equality information on the index, loop and data variables. Let IV be the set of index and loop variables of `P`. Let DV be the set of data variables of `P`. An abstract state is a tuple (m, SI, SD), where m is a boolean state in M, SI is a total order on equivalence classes on IV and SD is a total order on equivalence classes on $DV \cup IV$. An abstract state represents a set of local states. For example, if a program has an index variable `p1`, a loop variable `i1` and a data variable `v1`, a possible abstract state is $(m, \texttt{p1} < \texttt{i1}, \texttt{p1} = \texttt{i1} < \texttt{v1})$. This abstract state represents a set of concrete states whose boolean state is m and, the value of `p1` is less than the value of `i1`, the value of the array at position `p1` is the same as the value of the array at position `i1`, which is less than the value of `v1`.

We show that reachability of a boolean state m can be decided on the abstract system, in the sense that m is reachable in T if and only if it is reachable in T^α. (A boolean state m is reachable in T^α iff there exist SI and SD such that (m, SI, SD) is reachable in T^α.) The main part of the proof shows that every finite path in the abstract transition system is feasible in the concrete transition system. The first idea for a proof might be to show that the abstraction defines a bisimulation between abstract and concrete transition systems. However, this is not the case. We present a simple counterexample. Let us consider a program `P` and let us focus on two data variables `v1` and `v2`. Let q_1 be a local state such that its boolean component is m, the value of `v1` at q is 5 and the value of `v2` at q is 6. The abstract state corresponding to r_1, r_1^α is thus (m, SI, SD), where SD, the order on data and index variables, includes $\texttt{v1} < \texttt{v2}$. Furthermore, let us suppose that the program is such that the abstract state r_1^α can transition (in a way that does not change the values of `v1` and `v2`) to an abstract state r_2^α that requires that another data variable `v3` has a value greater than the value of `v1`, but smaller than the value of `v2`. Note now that the concrete state r_1 cannot transition to any state that would correspond to the order on data variables required by r_2^α, because there is no value between 5 and 6.

In a key part of the proof, we show that if an abstract state r_2^α is reachable from r_1^α, then there exists a state r_1 (abstracted by r_1^α) and a state r_2 (abstracted by r_2^α) such that r_2 is reachable from r_1. The main idea for proof by induction is that we can choose r_1 in such a way that the gaps between values are large enough. More precisely, if (1) r_1^α requires that e.g. $\texttt{v1} > \texttt{v2}$ for two data variables `v1` and `v2` and (2) r_2^α is reachable from r_1^α in k steps, then it is sufficient to choose r_1 such that $\texttt{v1} - \texttt{v2} > 2^k$.

The above argument gives rise to a PSPACE algorithm for deciding reachability of a boolean state m.

4 Programs, automata and logics on data words

In this section, we will examine the decidability boundary for array-accessing programs, and compare the expressive power of these programs to that of logics and automata on data words. We will show that the reachability problem for Restricted-ND2 programs is decidable, and that it is undecidable for full ND2 programs. We start by reviewing the results on automata and logics on data words, as these will be needed for the decidability proof. We will reduce the reachability problem for Restricted-ND2 programs to the nonemptiness problem of extended data automata, a new variation of data automata. The latter is a definition intended to correspond to the notion of regular automata on finite words.

4.1 Background

We briefly review the results on automata and logics on data words from [6]. Recall that a data word is a sequence of pairs $\Sigma \times D$. A *data language* is a set of data words. Let w be a data word $(a_1, d_1)(a_2, d_2) \dots (a_n, d_n)$. The string $str(w) = a_1 a_2 \dots a_n$ is called the string projection of w. Given a data language L, we write $str(L)$ to denote the set $\{str(w) \mid w \in L\}$. A class is a maximal set of positions in a data word with the same data value. Let $\mathcal{S}(w)$ be the set of all classes of the data word w. For a class X in $\mathcal{S}(w)$ with positions $i_1 < \dots < i_k$, the class string $str(w, X)$ is $a_{i_1} \dots a_{i_k}$.

Data automata. A *data automaton* (DA) $\mathcal{A} = (G, C)$ consists of a transducer G and a class automaton C. The transducer G is a nondeterministic finite-state letter-to-letter transducer from Σ to Γ and C is a finite-state automaton on Γ. A data word $w = (a_1, d_1)(a_2, d_2) \dots (a_n, d_n)$ is accepted by a data automaton $\mathcal{A}$ if there is an accepting run of G on the string projection of w, yielding an output string $b = b_1 \dots b_n$, and for each class X in $\mathcal{S}(w')$, the class automaton C accepts $str(w', X)$, where $w' = w'_1 \dots w'_n$ is defined by $w'_i = (b_i, d_i)$, for all i such that $1 \leq i \leq n$. Given a DA $\mathcal{A}$, $L(\mathcal{A})$ is the language of data words accepted by $\mathcal{A}$. The nonemptiness problem for data automata is decidable. The proof is by reduction to a computationally complex problem, the reachability problem in Petri nets.

Example 4. We present a data automaton $\mathcal{A}$ such that $str(L(\mathcal{A}))$, the set of string projections, is exactly the set of all words over $\{a, b, c\}$ that contain the same number of as, bs, and cs. The transducer of $\mathcal{A}$ computes the identity function, i.e. it accepts all words and its output string is the same as its input string. The class automaton ensures, for each class, that the class contains exactly one occurrence of a, one occurrence of b and one occurrence of c.

Logics on data words. We define logics whose models are data words. Following [6], we consider two predicates on positions in a data word whose definition also involves the data values at these positions. The predicate $i \approx j$ is satisfied if both positions i and j have the same data value. The predicate $i \prec j$ is satisfied if the data value at position i is smaller than the data value at position j. Furthermore, standard successor and order predicates on positions in a data word are used.

Let us first consider logics that use the $\approx$ predicate and not the $\prec$ predicate. We first note that for a first order logic $\mathrm{FO}(\approx, <, +1)$ satisfiability is undecidable, even if we restrict the number of variables to three. If we restrict the number of variables to two, the logic becomes decidable, and the proof is by reduction to the nonemptiness problem of data automata. The decidability naturally extends to existentially quantified second order monadic logic with two first order variables, denoted by $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$. Moreover, $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$ is precisely equivalent in expressive power to data automata. The predicate $\oplus 1$ denotes the class successor, and $i \oplus 1 = j$ is satisfied if i and j are two successive positions in the same class of the data word. Furthermore, the logic $\mathrm{EMSO}^2(\approx, <, +\omega, \oplus 1)$ is included in $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$. The symbol $+\omega$ represents all predicates of the form $+k$, $k \in \mathbb{N}$, i.e. the logic includes all predicates $i + 2 = j$, $i + 3 = j$, etc.

4.2 Extended data automata

Position-preserving class string Note that the class automaton does not know the positions of symbols in the word w. The symbols from other classes have simply been erased. However, let us consider a program with a doubly-nested loop where `i` is the loop variable of the outer loop and `j` is the loop variable of the inner loop, and let us suppose that the program inside the inner loop is of the form: `if (A[i].d=A[j].d) then P1 else P2`. The inner loop of the program scans the array from left to right and and modifies the state in two different ways (given by `P1` and `P2`), depending on whether `(A[i].d=A[j].d)` holds or not. Simply erasing the positions from other classes seems therefore not good enough. We thus define an extension of the notion of class string and a corresponding extension of the class automaton.

Given a data word $w \in (\Sigma \times D)^*$, a *position-preserving class string* $pstr(w, X)$ is a string over $\Sigma \cup \{0\}$. (We assume that $0 \notin \Sigma$.) Let $w = w_1 w_2 \dots w_n$, let i be a position in w, and let w_i be (a_i, d_i). The string $v = pstr(w, X)$ has the same length as w, and for v_i we have that $v_i = a_i$ iff $i \in X$, and $v_i = 0$ otherwise. That is, for each position i which does not belong to X, the symbol from Σ at the position i is replaced by 0.

An *extended data automaton* (EDA) $\mathcal{E} = (G, C)$ consists of a transducer G and a class automaton C. The transducer G is a finite-state letter-to-letter transducer from Σ to Γ and C is a finite-state automaton over $\Gamma \cup \{0\}$. A data word $w = w_1 \dots w_n$ is accepted by the EDA $\mathcal{E}$ if there is an accepting run of G on the string projection of w, yielding an output string $b = b_1 \dots b_n$, and for each class X in $\mathcal{S}(w')$, the class automaton C accepts $pstr(w', X)$, where $w' = w'_1 \dots w'_n$ is defined as follows: $w'_i = (b_i, d_i)$, for all i such that $1 \leq i \leq n$. Given an EDA $\mathcal{E}$, $L(\mathcal{E})$ is the language of data words accepted by $\mathcal{E}$.

Example 5. We consider L, a language of data words defined by the following property: A data word w is in L iff for every class X in $\mathcal{S}(w)$, we have that between every two successive positions in the class, there is exactly one position from another class. We show that there exists an EDA $\mathcal{E} = (G, C)$ such that $L(\mathcal{E}) = L$. The transducer G computes the identity function. The class automa-

ton C is given by the following regular expression: $0^*(\Sigma 0)^*0^*$. It is easy to see that $\mathcal{E}$ accepts L.

We first note that for each DA $\mathcal{A}$, it is easy to find an EDA $\mathcal{E}$ such that $L(\mathcal{E}) = L(\mathcal{A})$. We just modify the class automaton C, by adding the tuple $(q, 0, q)$, for each q, to the transition relation. This means that on reading 0 the state of the class automaton does not change.

We will also show in this section that for each EDA $\mathcal{E}$ we can find an equivalent DA $\mathcal{A}$. This might not be obvious at a first glance, as class automata of DAs do not get to see the distances between positions in a class. Indeed, we show that the language from Example 5 cannot be captured by a deterministic DA. However, we show that $\text{EMSO}^2(\approx, +1, \oplus 1)$ and EDAs are expressively equivalent, and since $\text{EMSO}^2(\approx, +1, \oplus 1)$ and DAs are also expressively equivalent, we conclude that for every EDA there exists a DA that accepts the same language. Showing that for every EDA there exists an equivalent $\text{EMSO}^2(\approx, +1, \oplus 1)$ formula also establishes that non-emptiness is decidable for EDAs. However, the proof of decidability of satisfiability of $\text{EMSO}^2(\approx, +1, \oplus 1)$ formulas is rather involved. We present a direct proof for decidability of emptiness for EDAs, as it also gives an intuitive reason why emptiness is decidable fro EDAs.

Theorem 2. *Given an EDA $\mathcal{E}$, it is decidable whether $L(\mathcal{E}) = \emptyset$.*

Proof. Let $\mathcal{E} = (G, C)$ be an EDA, let G be defined by a tuple $(Q_G, \Sigma, \Gamma, \delta_G, q_0^G, F_G)$, and let C be defined by a tuple $(Q_C, \Gamma, \delta_C, q_0^C, F_C)$. We start by describing a more operational view of EDAs. A *run* of an EDA on a data word w is a function ϱ from positions in w to tuples of the form (q, o, c), where $q \in Q_G$ is a state of the transducer G, o (a symbol from Γ) is the output of the transducer, and c is a function from $\mathcal{S}(w)$ to Q_C, the set of states of C. Furthermore, we require that ϱ is consistent with δ_G and δ_C, the transition functions of G and C. We define $\varrho(0)$ to be $(q_0^G, \gamma, \lambda X.q_0^C)$, i.e. the transducer and all the copies of the class automaton are in initial states. Furthermore, for each position i, $\varrho(i)$ is equal to (q', o', c') if and only if $w_i = (a, d)$, $\varrho(i-1) = (q, o, c)$ and (i) $(q', o') \in \delta_G(q, a)$, (ii) for the unique X such that $i \in X$ we have $c'(X) \in \delta_C(c(X), o')$, (iii) for X such that $i \notin X$ we have $c'(X) \in \delta_C(c(X), 0)$.

A run is *accepting* iff $\varrho(n) = (q, o, c)$, q is a final state of G and for all X in $\mathcal{S}(w)$, we have that $c(X)$ is a final state of C.

Let us consider the class automaton C. Without loss of generality, we suppose that C is a complete deterministic automaton on $\Gamma \cup \{0\}$. The transition function δ_C defines a directed graph C_0 with states of C as vertices and 0-transitions as edges, i.e. there is an edge (p_1, p_2) in C_0 if and only if $\delta_C(p_1, 0) = p_2$. Every vertex in C_0 has exactly one outgoing edge (and might have multiple

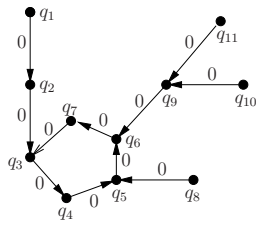

Fig. 3. A connected component of a graph C_0 corresponding to an EDA $\mathcal{E}$

incoming edges). Therefore, each connected component of C_0 has exactly one cycle. A vertex is called cyclic if it is part of a cycle, and it is called non-cyclic otherwise. It is easy to see that each connected component is formed by the cyclic vertices and their 0-ancestors. An example of a connected component is in Figure 3. The vertex labeled q_6 is cyclic, its ancestors q_9, q_{10}, q_{11} are non-cyclic.

The graph C_0 consists of a number of connected components. We denote these components by C_0^j, for $j \in [1..k]$, where k is the number of the components. Let W be the set of all non-cyclic vertices. For each non-cyclic vertex v, let $D(v)$ be defined as follows: $D(v) = d$ for non-cyclic vertices connected to a cycle, where d is the length of the unique path connecting v to the closest cyclic vertex. For the graph C_0, we define $D(C_0)$ to be $\max_{v \in W} D(v)$.

Let i be a position in a data word w. The data word $w_1 w_2 \dots w_i$ is denoted by $\mathit{prefix}(w, i)$. Let us consider a position i in a data word w and the set of classes $\mathcal{S}(w)$. Let $\mathcal{S}_{act}(w, i)$ be a set of *active* classes, i.e. classes X such that there is a position in X to the left of the position i. More formally, a class $X \in \mathcal{S}(w)$ is in $\mathcal{S}_{act}(w, i)$ if the string $\mathit{str}(\mathit{prefix}(w, i), X)$ is not equal to 0^i.

Lemma 1. *Let ϱ be a run of $\mathcal{E}$ on w. Let i be a position in w. Let $\varrho(i)$ be (q, o, c). The number N of classes X, such that X is in $\mathcal{S}_{act}(w, i)$ and $c(X)$ is a noncyclic vertex, is bounded by $D(C_0)$, i.e. $N \leq D(C_0)$.*

Proof. Let i be a position in a word w. If $i \leq D(C_0)$, then the number of active classes is at most $D(C_0)$, and we conclude immediately. Let us consider the case $i > D(C_0)$. Let $\varrho(i)$ be (q, o, c) and let s be the string of length $D(C_0)$ defined by $s = w_{i-D(C_0)+1}\ w_{i-D(C_0)+2} \dots w_i$. There are two possible cases for each class X in $\mathcal{S}(w)$. The first case is the case when $\mathit{pstr}(s, X) = 0^{D(C_0)}$. Let $\varrho(i - D(C_0)) = (q', o', c')$, and let $c'(X) = v$. We can easily prove that $\delta_C^*(p, 0^e)$ is not in W, for all p and for all $e \geq D(p)$. By definition, $D(C_0) \geq D(q')$. Therefore, we can conclude that $c(X) \notin W$. The second case is the case when $\mathit{pstr}(s, X) \neq 0^{D(C_0)}$. This is true for at most $D(C_0)$ classes, because, for all positions x, there is exactly one class X, such that the symbol at the position x of the class string $\mathit{pstr}(s, X)$ is not 0. Thus we have that $c(X) \in W$ for at most $D(C_0)$ classes.

We reduce emptiness of EDAs to emptiness of multicounter automata. Multicounter automata are equivalent to Petri nets [11], and thus the emptiness of multicounter automata is decidable. We use the definition of multicounter automata from [6]. A *multicounter automaton* is a finite, non-deterministic automaton extended by a number k of counters. It can be described as a tuple $(Q, \Sigma, k, \delta, q_I, F)$. The set of states Q, the input alphabet, the initial state $q_I \in Q$ and final states $F \subseteq Q$ are as in a usual finite automaton. The transition relation is a subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times \{\mathit{inc}(i), \mathit{dec}(i)\} \times Q$. The idea is that in each step, the automaton can change its state and modify the counters, by incrementing ($\mathit{inc}(i)$ increments counter number i) or decrementing them, according to the current state and the current letter on the input (which can be ε). Whenever it tries to decrement a counter of value zero the computation stops and rejects. The transition of a multicounter automaton does not depend on the value of

the counters in any other way. In particular, it cannot test whether a counter is exactly zero. The automaton accepts when the state is final and all the counters are empty.

Lemma 2. *Let $\mathcal{E}$ be an EDA. A multicounter automaton V such that $str(L(\mathcal{E})) = L(V)$ can be computed from $\mathcal{E}$.*

Proof. We present the construction of a multicounter automaton V that simulates $\mathcal{E}$. The multicounter automaton V simulates the transducer G and a number of copies of C. There is one copy per class in $\mathcal{S}(w)$, where w is the word the automaton is reading. We say that a class automaton performs a 0-transition if the input symbol it reads is 0, and it performs a Γ-transition if the input symbol it reads is from Γ. Intuitively, at each step, the automaton V: (i) Simulates the transducer G using the finite state part (i.e. not the counters), and (ii) It guesses to which class the current position belongs, and it executes the Γ-transition of the automaton for that class with the symbol that is the output of the transducer at this step. For all the other simulated automata, V executes the 0-transition. (This is sufficient because each position belongs to exactly one equivalence class.) The counters of the multicounter automaton V correspond to the cyclic vertices in C_0. (In what follows, we call a state of C (non-)cyclic if it corresponds to a (non-)cyclic vertex in C_0.) The value of the counter h corresponds to the number of copies of C currently in the state h. The finite part of the automaton state tracks the number of copies in each non-cyclic state. The key idea of the proof is that the total number of copies in non-cyclic states is finite and bounded (by $D(C_0)$). This fact is implied by Lemma 1.

Furthermore, one copy e of the class automaton is used to keep track of all the classes that are not active yet, i.e. not in $\mathcal{S}_{act}(w, i)$ at step i - thus when a position-preserving class string contains a symbol in Γ for the first time, a new copy of the automaton C is started from the state at which the copy e is.

Let $\gamma \in \Gamma$ be the current input symbol. The automaton works as follows: The first step consists of the automaton V nondeterministically guessing the equivalence class X to which the current position belongs. The copy of the class automaton for X is then set aside while the second step is performed. That is, if the copy is in state s, then s is remembered in a separate part of the finite state. In the second step, the automaton V simulates 0-transitions for all the other copies (other than the copy that performed the Γ-transition). For copies in non-cyclic states, this is done by a transition modifying the finite state of V. The copies that transition from a non-cyclic to a cyclic state are dealt with by modifying the finite state and increasing the corresponding counter. The copies in cyclic states are tracked in the counters. Note that if we restrict the graph to only cyclic states, each state has exactly one incoming and one outgoing 0-edge. For all the copies in cyclic states, the 0-transition is accomplished by "relabeling" the counters. This is done by remembering in the the finite state of V for each loop for one particular state to which counter it corresponds. This is then shifted in the direction of the 0-transition.

The third step is to perform the Γ transition for the class X. For the copy of the automaton corresponding to this class, a Γ-transition is performed. That

is, if it is in state q, and $\delta(q, \gamma) = q'$, then there are four possibilities.: (i) if q, q' are cyclic states, the counter corresponding to q is decreased and the counter corresponding to q' is increased; (ii) if q, q' are non-cyclic state, a transition that changes the state of V is made; (iii) if q is a cyclic state and q' is a non-cyclic state, the counter corresponding to q is decreased, and the finite state of V is changed to reflect that the number of copies in q' has increased; (iv) if q is a noncyclic state and q' is a cyclic state, the transition is simulated similarly.

This concludes the proof of Theorem 2.

4.3 Restricted doubly-nested loops

We will reduce the reachability problem of Restricted-ND2 programs to the emptiness problem of EDAs. The main idea of the proof is that the transducer G guesses an accepting run of the outer loop, while the class automaton C checks that the inner loop can be executed in a way that is consistent with the guess of the transducer.

Theorem 3. *Reachability for Restricted-ND2 programs is decidable.*

The proof of Theorem 3 gives a decision procedure, but one whose running time is non-elementary. The reason is that while the problem of reachability in multicounter automata is decidable, no elementary upper bound is known.

However, the following proposition shows that the problem is at least as hard as the reachability in multicounter automata, which makes it unlikely that a more efficient algorithm exists. The best lower bound for the latter problem is EXPSPACE [19].

Proposition 4. *The reachability problem for multicounter automata can be reduced to the reachability problem for Restricted-ND2 programs.*

4.4 Undecidable extensions

We show that if we lift the restrictions we imposed on Restricted-ND2 programs, the reachability problem becomes undecidable.

Theorem 5. *The reachability problem for ND2 programs is undecidable.*

The proof is by reduction from the reachability problem of two-counter automata [20]. We note that the proof also implies that the reachability problem is undecidable even for ND2 programs that do not use order on the data domain and do not use index or data variables.

We investigate the case of Restricted-ND2 programs with access to order on the data domain and with data index variables. We show that if we add order on the data domain and at least one data variable, the reachability problem becomes undecidable. The proof is by reduction from the Post's Correspondence Problem, and is similar to the proof of Proposition 21 of [6].

Proposition 6. *Reachability for Restricted-ND2 programs that use order on D and at least one data variable is undecidable.*

A natural question, which is now open, is whether it is possible to add only one of these features (order on data domain or data (index) variables) to Restricted-ND2 programs without losing decidability of the reachability problem.

4.5 Expressiveness

In this section, we compare expressiveness of logics and automata on data words and array-accessing programs. We make our comparisons in terms of languages of data words these formalisms can define. Due to a lack of space, we present only the results in this subsection.

Language of a program. In order to define the language of a program, we extend the notion of a program by adding a final state. That is, in this section we will assume that every program $\mathtt{P}$ has a final state m_f, where m is a boolean state of $\mathtt{P}$. The language $L_m(\mathtt{P})$ is the set of data words w, such that there exist an initial state g_I and a state g, such that $g_I[\mathtt{A}] = w$, $bool(g) = m$, and $(g_I, g) \in [\![\mathtt{P}]\!]$. We say that a program $\mathtt{P}$ *accepts* the language $L_m(\mathtt{P})$, where m is the final state.

The following proposition shows that EDAs and $\text{EMSO}^2(\approx, +1, \oplus 1)$ are equally expressive. This means that somewhat surprisingly, DAs and EDAs are expressively equivalent.

Proposition 7. *EDAs and $EMSO^2(\approx, +1, \oplus 1)$ are equally expressive.*

The following proposition sheds light on the difference between DAs and EDAs. We saw that DAs and EDAs are expressively equivalent. However, one difference between EDAs and DAs is that deterministic EDAs are more expressive than deterministic DAs. It is the nondeterminism that then levels the difference.

Proposition 8. *Deterministic EDAs are more expressive than deterministic DAs.*

We show that nondeterminism adds to the expressive power of EDAs, as there exists a language accepted by a nondeterministic EDA, but no deterministic EDA can accept it. This implies the following proposition.

Proposition 9. *Deterministic EDAs are strictly less expressive than EDAs.*

We will now compare the expressive power of array-accessing programs to logics and automata on data words. Specifically, we will use the logic $\text{EMSO}^2(\approx, +1, \oplus 1)$ for comparison. Recall that this logic is expressively equivalent to data automata. We first show that Restricted-`ND2` programs are not as expressive as $\text{EMSO}^2(\approx, +1, \oplus 1)$. We also compare the expressive power of `ND1` programs and $\text{EMSO}^2(\approx, +1, \oplus 1)$.

Proposition 10. *Restricted-*`ND2` *programs are strictly less expressive than $EMSO^2(\approx, +1, \oplus 1)$.*

Proposition 11. *There exists an $EMSO^2(\approx, +1, \oplus 1)$ property that is not expressible by an* `ND1` *program.*

Note that `ND1` programs allow order on the data domain, and thus can check a property specifying that the elements in the input data word are in increasing order. It is easy to see that this property is not specifiable in $\text{EMSO}^2(\approx, +1, \oplus 1)$. However, if we syntactically restrict `ND1` programs not to use order on D, they can

be captured by $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$ formulas. The reason is that `ND1` programs that do not refer to the order on D can be simulated by register automata introduced in [17]. For every register automaton, there is an equivalent data automaton [5]. Another natural question is whether there is an order-invariant property that can be captured by `ND1` programs (that have access to order), but is not expressible in $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$. We leave this question for future work.

5 Related work

Our results establish connections between verification of programs accessing arrays and logics and automata on data words. Kaminski and Francez [17] initiated the study of finite-memory automata on infinite alphabets. They introduced register automata, that is automata that in addition to finite state have a fixed number of registers that can store data values. The results of Kaminski and Francez were recently extended in [21, 6, 5, 4]. Data automata introduced in this line of research were shown to be more expressive than register automata. Furthermore, the logic $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$ was introduced, and [6] shows that $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$ and data automata are equally expressive. The reduction from $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$ to data automata and the fact that emptiness is decidable for data automata imply that satisfiability is decidable for $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$. We show that Restricted-`ND2` programs can be encoded in $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$. However, adding a third variable to the logic or allowing access to order on data variable makes satisfiability undecidable for the resulting logic, even for the first order fragment. We show, perhaps somewhat surprisingly, that the undecidability does not translate into undecidability of reachability for `ND1` programs that access order on the data domain and have an arbitrary number of index and data variables. The results on automata and logics on data words model were applied in the context of XML reasoning [21] and extended temporal logics [9]. The connection to verification of programs with unbounded data structures is the first to the best of our knowledge.

Deutsch et al. [10] consider a model of database-driven systems similar in some aspects to our model of programs. The key difference is that they consider a dense order. They specifically note that the model-checking problem they consider is open for the case of a discrete order. It would be interesting to see if our result on programs on structures with discrete order can be extended to the setting of database-driven systems. Fragments of first order logic on arrays have been shown decidable in [8, 15, 2, 7]. These fragments do not restrict the number of variables (as was the case with $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$), but restrict the number of quantifier alternations. These papers focus on theory of arrays, rather than on analysis of array-accessing programs. Decidability of reachability for polymorphic systems with arrays (PSAs) was studied e.g. in [18]. PSAs use well-typed λ-terms and do not allow iteration over arrays.

Static analysis of programs that access arrays is an active research area, with recent results including [12, 14, 2]. The approach consists in finding inductive invariants for loops using abstraction methods, such as abstract domains that can represent universally quantified facts [14] and a predicate abstraction approach

to shape analysis [2]. In contrast, our results yield decision procedures for array-accessing programs. However, the methods based on abstraction are applicable to a richer class of programs. Note that the abstract domains used for examples and applications also track equality and order on array elements.

References

1. R. Alur, P. Černý, and S. Weinstein. Algorithmic analysis of array-accessing programs. Technical Report MS-CIS-08-35, University of Pennsylvania, 2008.
2. I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *VMCAI*, pages 164–180, 2005.
3. T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
4. H. Björklund and M. Bojańczyk. Shuffle expressions and words with nested data. In *MFCS*, pages 750–761, 2007.
5. H. Björklund and T. Schwentick. On notions of regularity for data languages. In *FCT*, pages 88–99, 2007.
6. M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *LICS*, pages 7–16, 2006.
7. A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu. Rewriting systems with data. In *FCT*, pages 1–22, 2007.
8. A. Bradley, Z. Manna, and H. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.
9. S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. In *LICS*, pages 17–26, 2006.
10. A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *ICDT*, pages 252–267, 2009.
11. J. Gischer. Shuffle languages, Petri nets, and context-sensitive grammars. *Commun. ACM*, 24(9):597–605, 1981.
12. D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2008.
13. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
14. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.
15. P. Habermehl, R. Iosif, and T. Vojnař. What else is decidable about integer arrays? In *FoSSaCS*, pages 474–489, 2008.
16. T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *CAV*, pages 526–538, 2002.
17. M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.
18. R. Lazić. Decidability of reachability for polymorphic systems with arrays: A complete classification. *ENTCS*, 138(3):3–19, 2005.
19. R. Lipton. The reachability problem requires exponential space. Technical Report Dept. of Computer Science, Research report 62, Yale University, 1976.
20. M. Minski. Recursive unsolvability of Post's problem of 'tag' and other topics in theory of Turing machines. *Annals of Mathematics*, 74:437–455, 1962.
21. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, 2004.