Algorithmic Analysis of Array-Accessing Programs

Rajeev Alur Pavol Černý Scott Weinstein
University of Pennsylvania
{alur,cernyp,weinstein}@cis.upenn.edu

December 3, 2008

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 array whose length is potentially unbounded, and whose elements range over pairs from $\Sigma \times D$, where Σ is a finite alphabet and D is 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) solvable in EXPSPACE for programs with arbitrarily nested loops if array elements range over a finite data domain, and (3) decidable for a restricted class of programs with doubly-nested loops. The third 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 [2, 17, 14].

A natural question is then whether it is possible to extend the Boolean program model without losing decidability or worsening computational complexity of the reachability problem. The first idea might be to add integer variables. However, adding expressions permitting (Pressburger) integer arithmetic would cause undecidability. Therefore, one can investigate the possibility of adding only equality and order tests on integers to the language. Reachability in such programs is decidable, but perhaps the programs themselves are not too interesting. We show that it is possible to extend the model further. We consider programs that have boolean variables, variables from a potentially infinite domain, and access an array.

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, integer arrays are easily captured by setting D to be $\mathbb{N}$ and Σ to be a singleton set. 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 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. Array is a heavily used data structure. 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. In order to analyze programs statically, it is often necessary to check relationships among values in the array, as well as their relationships to values of other variables and constants. For example, in the case of indirect addressing, it is needed to check that all the values in the array fall within certain bounds. For programs that fall outside the restrictions mentioned above, it is possible to use abstract interpretation [8] 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 consider nondeterministic programs in this paper.

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. The latter is the basis of successful software verification tools, and therefore we believe that, coupled with abstraction techniques, our decision procedure can potentially be the basis of a software model checking tool that better handles data structures with potentially unbounded size.

Our second result establishes decidability of the reachability problem for programs with arbitrary nesting of loops that do not use index variables, under the assumption that the data domain is finite. The algorithm can be used for bounded model checking of such programs. In this case, the array can be viewed as a finite word over the finite alphabet of data values. The traversal order of a program with nested loops and index/data variables does not directly correspond to classical extensions of automata with multiple passes and/or pebbles (see for example [11]). We show that the set of arrays for which a particular Boolean state is reachable is regular, and reachability is solvable in space polynomial in the number of states of the program, which itself is exponential in the number of variables.

Our third 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* [5, 4, 18]. Among different kinds of automata over data words that have been studied, *data automata* [5] 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 word over Γ 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, obtained by replacing each position with data value different from d by the special symbol 0, 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. Bojanczyk et al. [5] 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. 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.

2 Programs

In this section, we define the syntax and semantics of programs that we will consider in this paper.

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.

2.1 Syntax

The programs have one array variable `A`. Variables `b1`, `b2`, ... are boolean, `p1`, `p2`, ... range over $\mathbb{N}$, and are called index variables, `i1`, `i2`, ... range over $\mathbb{N}$ and are called loop variables, `v1`, `v2`, ... range over D and are called data variables. `c1`, `c2`, ... are constants in D, and `s1`, `s2`, ... are constants 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 of the form

```
IE :: = p1 | i1
```

Data expressions `DE` are of the form

```
DE :: = v1 | c1 | A[IE].d
```

where `A[IE].d` accesses the data part of the array.

Σ-expressions `SE` are of the form

```
SE :: = s1 | A[IE].s
```

where `A[IE].s` accesses the Σ part of the array.

Boolean expressions are defined by the following grammar:

```
 B :: =    true | 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
         | b1:=B
         | p1:=IE
         | v1:=DE
         | if B then P else P
         | if * then P else P
         | for i1:=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, 2, 17] 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.

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. For all occurrences of a doubly nested loop in a Restricted-`ND2` program, the following holds. Let `i` be the loop variable of the outer loop and let `j` be the loop variable for the inner loop. The expression `A[j].d` (`A[j].s`) can only be compared with `A[i].d` (`A[i].s`), that is, it cannot be compared to constants.

2.2 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 `v`, we denote the value of `v` by $g[\mathtt{v}]$. The valuation of the array variable A is a word $w \in (\Sigma \times D)^*$. The length of the array at global state g is denoted by $g[l(\mathtt{A})]$ and evaluates to the length of w, and the valuation of the array is denoted by $g[\mathtt{A}]$. 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 can be defined in a standard way: $[\![\mathtt{B}]\!] : G \to \mathbb{B}$, $[\![\mathtt{IE}]\!] : G \to \mathbb{N}$, $[\![\mathtt{DE}]\!] : G \to D$ and $[\![\mathtt{SE}]\!] : G \to \Sigma$.

We define the semantics of commands: $[\![\mathtt{P}]\!] \subseteq G \times G$.

- $(g, g) \in [\![\mathtt{skip}]\!]$, for all g in G
- $(g, g') \in [\![\mathtt{v:=E}]\!]$, iff $g' = g[\mathtt{v} \leftarrow [\![\mathtt{E}]\!](g)]$, for any assignment.
- $(g, g') \in [\![\mathtt{if\ B\ then\ P1\ else\ P2}]\!]$ iff $[\![\mathtt{B}]\!](g) = \mathit{true}$ and $(g, g') \in [\![\mathtt{P1}]\!]$ or $[\![\mathtt{B}]\!](g) = \mathit{false}$ and $(g, g') \in [\![\mathtt{P2}]\!]$.
- $(g, g') \in [\![\mathtt{if\ *\ then\ P1\ else\ P2}]\!]$ iff $(g, g') \in [\![\mathtt{P1}]\!]$ or $(g, g') \in [\![\mathtt{P2}]\!]$.
- $(g, g') \in [\![\mathtt{for\ i1:=1\ to\ length(A)\ do\ P}]\!]$ iff there exist $g_1, g_2, \ldots, g_{l+1}$, where $l = g[\mathtt{l(A)}]$, such that $g_1 = g$, $g_{l+1} = g'$, and for all i such that $1 \le i \le l$, we have that there exists a g'_{i+1}, such that $(g_i, g'_{i+1}) \in [\![\mathtt{P}]\!]$ and $g_{i+1} = g'_{i+1}[\mathtt{i1} \leftarrow i + 1]$.

- $(g, g') \in [\![\texttt{P1;P2}]\!]$ iff there exists g'' such that $(g, g'') \in [\![\texttt{P1}]\!]$ and $(g'', g') \in [\![\texttt{P2}]\!]$.

Given a program, a global state is *initial* if 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. Thus the only non-specified part of the initial state, the part that models input of the program, is the array.

Note that for the programs we have defined, where the only iteration allowed is over the array, the termination is guaranteed. Therefore for all initial global states g_I there exists a global state g such that $(g_I, g) \in [\![\texttt{P}]\!]$.

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 $bool(g)[\texttt{b}] = g[\texttt{b}]$. We denote boolean states by m, m_1 and the set of boolean states by M.

2.3 Examples

We present four illustrative examples for the classes of programs defined in this section. First, we will consider a simple array accessing program that scans through an array to find a minimal data value. It has one index variable, `min`, and it is an `ND1` program, as it does not contain nested loops. Note that by definition `min` is initialized to 1.

Example 1.

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

The correctness requirement for this program is that the index `min` points to a minimal element, that is $\forall$ `i:` `A[i]` $\geq$ `A[min]`. Verifying the correctness of the program can be reduced to checking reachability, as the requirement itself can be expressed as a program.

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

Second, we present an `ND1` program that tests whether the array is sorted. It uses one data variable called `prev` (note that by definition, `prev` is initialized to the same value as the first element of the array).

Example 2.

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

Third, let us construct a Restricted-`ND2` program that tests whether there is a data value d that appears twice in the array:

Example 3.

```
b:=false
for i:= 1 to length(A) do {
  for j:= 1 to length(A) do
    if (A[i].d == A[j].d) and (i != j) {
      b:=true;
    }
}
```

Fourth, let us consider an `ND2` program that checks the following property: (1) The projection of the array to the Σ component of each element is $(\Sigma \setminus \{\$\})^*\$(\Sigma \setminus \{\$\})^*$, and (2) the data value of the \$-position occurs exactly once, and each other data value occurs precisely twice — once before and once after the \$ sign. There is an (easy to write) `ND2` program that checks this property. This fact is used in the proof of undecidability of `ND2` programs. Note however, that this property cannot be checked with Restricted-`ND2` programs. The reason is that the inner loop of such a program cannot compare the values in the array to constants, and thus does not 'see' the \$ sign. Therefore it cannot check that one of the occurrences appeared to the left of the \$ sign, while the other appeared to the right.

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 $(g_I, g) \in [\![\mathtt{P}]\!]$ and $bool(g) = m$. The reachability problem is to determine, for a given program and a given boolean state m, whether m is reachable.

In this section, we show that reachability is decidable for programs with nesting depth equal to 1 (`ND1`) programs, and it is decidable for programs with arbitrary nesting of loops (but without index variables) as well if the data domain D is finite.

Local states. 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`.

For a given global state g, we denote the corresponding local state by $loc(g)$. For any variable v of the program, we have that $loc(g)[\texttt{v}] = g[\texttt{v}]$. If `v` is an index or a loop variable, we also have that $loc(g)[\texttt{A\_v}] = [\![\texttt{A[v]}]\!](g)$. We denote local states by q, q_1, and the set of local states by Q. A local state q is *initial* if there exists an initial global state g_I such that $loc(g_I) = q$.

Normal form. In order to simplify the presentation of proofs of the decidability results, we will first translate the programs into a normal form. A program is in normal form if the branches of `if` statements do not contain loops.

We define a translation function *norm*(`P`), that given a program `P` returns an equivalent program in normal form. We use an auxiliary function *assume*(`B, P`), and we set *norm*(`P`) = *assume*(`true, P`). The function *assume*(`B, P`) is defined inductively as follows:

- *assume*(`B, skip`) = `skip`.
- *assume*(`B, v:=E`) = `if B then v:=E else skip`,
  if `B` is not `true`, and `v:=E` otherwise.
- *assume*(`B, if B1 then P1 else P2`) =
  `b := B1;`
  assume(`B and b, P1`);*assume*(`B and (not b), P2`),
  where `b` is a new boolean variable
- *assume*(`B, if * then P1 else P2`) =
  `if * then b:=true else b:=false;`
  assume(`B and b, P1`);*assume*(`B and (not b), P2`),
  where `b` is a new boolean variable
- *assume*(`B, for i1:=1 to length(A) do P`) =
  `for i1:=1 to length(A) do` *assume*(`B, P`).
- *assume*(`B, P1;P2`) =
  assume(`B, P1`);*assume*(`B, P2`).

The program *norm*(`P`) has more variables than the program `P`. However, intuitively the programs *norm*(`P`) and `P` compute the same function on the common variables. We now formalize this notion. Let `P`, `P'` be two programs, and let G (G') be the set of global states of `P` (`P'`). For a function $f : G' \to G$, we define $[\![\texttt{P}]\!] \sim_f [\![\texttt{P'}]\!]$ iff we have that for all $g'_1, g'_2 \in G'$ we have that $(g'_1, g'_2) \in [\![\texttt{P'}]\!]$ iff $(f(g'_1), f(g'_2)) \in [\![\texttt{P}]\!]$.

Let P be a program, let G be its set of global states and let V be its set of variables. Let V' be the set of variables of *norm*(P), and let G' be its set of global states. We have that $V \subseteq V'$. We define a function $\pi : G' \to G$ as follows: $\pi(g') = g$ iff g and g' agree on variables from V.

Proposition 1. *For all programs* P*, we have that* $[\![norm(P)]\!] \sim_\pi [\![P]\!]$*. Furthermore, the nesting depth of loops is the same in norm*(P) *as it is in* P*. The number of boolean variables in norm*(P) *increased by at most the number of* `if` *statements in* P.

3.1 Programs with non-nested loops

The goal of this subsection is to prove the following theorem:

Theorem 2. *Reachability for* ND1 *programs is decidable. The problem is* PSPACE-*complete.*

The structure of the proof will be as follows. We first characterize the semantics of a program P in terms of a transition system T whose states will be tuples of local states of P. Secondly, we construct a finite state system T^α that abstracts the infinite part of the local states, that is the values of index and data variables, and keeps only the order of these values. 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. The key part of the proof that relates the transition systems T and T^α is Lemma 5. The lemma shows that every finite path in the abstract transition system is feasible in the concrete transition system for an appropriately chosen array.

To simplify the presentation, we will suppose that there are no constants in D in the programs. At the end of this subsection, we will explain how the proof that follows can be extended to programs with constants from D.

Transition system semantics. We show that for programs that contain only non-nested loops and are in normal form, $[\![\texttt{P}]\!]$ can be represented by a triple (e, T, f), where $T = (R, \delta \subseteq R \times (\Sigma \times D) \times R, F)$ is a transition system whose set of states is R. The set $F \subseteq R$ is the set of final states. The transition relation will simulate executions of the loops that appear in the program. Its input will be, in addition to a state from R, also a pair (a, d) from $(\Sigma \times D)$ representing the current element of the input array. The relation e is a subset of $Q \times R$ and the relation f is a subset of $F \times Q$. The relation e will represent the loop-free part of the program before the first non-nested loop, and the relation f will represent the loop-free part of the program after the last non-nested loops. Recall that for program in normal form, loops do not appear in branches of `if` statements.

We define a function $[\![\texttt{P}]\!]^t$ which for loop-free programs returns a binary relation over Q, and returns a triple (e, T, f) for programs that contain non-nested loops. Intuitively, a loop free program P will be represented by a binary relation over Q. For a loop command we use the relation representing the (loop free) body of the loop to construct a transition system. For sequential composition of commands, a product construction augmented with some bookkeeping is used. We explain the construction for two sequentially composed loops that iterate through the array. The transition system is a product of the transition systems defined by the two loops, and the bookkeeping part ensures that the second loop starts from a state where the first loop finished.

For the following commands `P`: `skip`, `v := E`, `if B then P1 else P2`, `if * then P1 else P2`, $[\![\texttt{P}]\!]^t$ is defined by

$$[\![\texttt{P}]\!]^t = \{(loc(g), loc(g'))|(g, g') \in \ [\![\texttt{P}]\!]\}.$$

Note that for the conditionals, we have that `P1` and `P2` are loop-free. For loops and sequential composition we have:

- $[\![\texttt{for i1:=1 to length(A) do P}]\!]^t = (e, (Q, \delta, Q), f)$, where e and f are identity relations on Q, and $\delta(q, (a, d), q')$ if there exists a local state $q'' \in Q$ such that $(q, q'') \in [\![\texttt{P}]\!]^t$ and $q' = q''[\texttt{i1} = i + 1, \texttt{A\_i1} = (a, d)]$, where $i = q[\texttt{i1}]$. (Note that `P` is loop free.)
- $[\![\texttt{P1;P2}]\!]^t$ is defined as follows:
  1. If $[\![\texttt{P1}]\!]^t = f_1$ and $[\![\texttt{P2}]\!]^t = f_2$, then $[\![\texttt{P1;P2}]\!]^t = f_1 \circ f_2$.
  2. If $[\![\texttt{P1}]\!]^t = f_1$ and $[\![\texttt{P2}]\!]^t = (e_2, T_2, f_2)$, then $[\![\texttt{P1;P2}]\!]^t = ((f_1 \circ e_2), T_2, f_2)$.
  3. If $[\![\texttt{P1}]\!]^t = (e_1, T_1, f_1)$ and $[\![\texttt{P2}]\!]^t = f_2$, then $[\![\texttt{P1;P2}]\!]^t = (e_1, T_1, (f_1 \circ f_2))$.
  4. If $[\![\texttt{P1}]\!]^t = (e_1, T_1, f_1)$ and $[\![\texttt{P2}]\!]^t = (e_2, T_2, f_2)$, then $[\![\texttt{P1;P2}]\!]^t = (e, T, f)$, where the components are defined as follows. Let $T_1 = (R_1, \delta_1, F_1)$ and $T_2 = (R_2, \delta_2, F_2)$. The transition system $T = (R, \delta, F)$ is defined as follows: $R = R_1 \times R_2 \times R_2$, $\delta((r_1, r_2, r_3), (a, d), (r'_1, r'_2, r'_3))$ iff $\delta_1(r_1, (a, d), r'_1)$, $r_2 = r'_2$, and $\delta_2(r_3, (a, d), r'_3)$. A state (r_1, r_2, r_3) is in F if and only if $r_1 \in F_1$, $r_3 \in F_2$, and $(r_1, r_2) \in (f_1 \circ e_2)$. The function e is defined in the following way: $(q, (r_1, r_2, r_3)) \in e$ if and only if $r_2 = r_3$ and $(q, r_1) \in e_1$. For the function f, we have $((r_1, r_2, r_3), q) \in f$ if $(r_1, r_2, r_3) \in F$ and $(q, r_3) \in f_2$.

We now show that $[\![\texttt{P}]\!]^t = (e, T, f)$ captures the semantics of `P`. In what follows, we suppose that the program that we analyze contains at least one non-nested loop, and therefore $[\![\texttt{P}]\!]^t$ has the form (e, T, f).

Given a transition system $T = (R, \delta, F)$, where δ is a subset of $R \times (\Sigma \times D) \times R$, we extend the definition of δ to words in $(\Sigma \times D)^*$. We define a relation δ^* on $R \times (\Sigma \times D)^* \times R$ as follows: for $w = w_1 \dots w_l$ we have that $\delta^*(r, w, r')$ iff $\exists r_1, \dots r_{l+1}$ such that $r = r_1$, $r' = r_{l+1}$ and for all i such that $1 \leq i \leq l$ we have that $\delta(r_i, w_i, r_{i+1})$.

Given a word w in $(\Sigma \times D)^*$, we say that q_2 *is w-reachable from q_1 in $[\![P]\!]^t$* iff $[\![\texttt{P}]\!]^t = (e, T, f)$, $T = (R, \delta, F)$ and there exist $r_1, r_2 \in R$ such that $(q_1, r_1) \in e$, $(r_2, q_2) \in f$, and $\delta^*(r_1, w, r_2)$.

Lemma 3. *A local state q_2 is w-reachable from q_1 in $[\![P]\!]^t$ if and only if there exist states g_1 and g_2 such that $loc(g_1) = q_1$, $g_1[A] = w$, $loc(g_2) = q_2$, $g_2[A] = w$ and $(g_1, g_2) \in [\![P]\!]$.*

Proof. The proof uses induction on the structure of the program `P`. □

A boolean state m is *w-reachable in* $[\![P]\!]^t$ if there exist an initial local state q_I, a local state q such that $bool(q) = m$ and q is w-reachable from q_I in $[\![\mathtt{P}]\!]^t$.

The next lemma follows from Lemma 3.

Lemma 4. *Given a program P, a boolean state m is reachable if and only if there exist a word $w \in (\Sigma \times D)^*$ such that m is w-reachable in $[\![P]\!]^t$.*

Furthermore, if $[\![\mathtt{P}]\!]^t = (e, T, f)$ and $T = (R, \delta, F)$, we have that $R = Q^{2k-1}$, where k is the number of loops in `P`.

Abstract transition system. We fix a program `P` for the rest of this subsection. Let $[\![\mathtt{P}]\!]^t$ be (e, T, f), where $T = (R, \delta, F)$, and $R = Q^{2k-1}$. We show that we can find a finite state system T^α (and corresponding relations e^α and f^α) such that we can reduce reachability in T to reachability in T^α. The main idea in the construction of the abstract transition system is that it will keep track of only the order of index and data variables, not their values.

We will need an abstract version of the set Q. 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 IV and SD is a total order on $DV \cup IV$. For example, if a program has an index variable `p1`, a loop variable `i1` and a data variable `d1`, a possible abstract state is $(m, \mathtt{p1} < \mathtt{i1}, \mathtt{p1} = \mathtt{i1} < \mathtt{d1})$. This means that the program is in a boolean state m, `p1` is less than `i1`, and `A[p1]` is equal to `A[i1]` and is less than `d1`. Let Q^α be the set of abstract states.

We will also need an abstract version of R, the set of states of T. We consider sets IV^{2k-1} and DV^{2k-1}, where there are $2k-1$ copies of each variable. Let SI_R be a total order on IV^{2k-1} and let SD_R be a total order on $DV^{2k-1} \cup IV^{2k-1}$. We will consider the set $U = M^{2k-1}$. Let R^α be the set of abstract states of the form (u, SI_R, SD_R), where u is in U.

The abstraction function $\alpha_Q : Q \to Q^\alpha$ can be defined straightforwardly: $\alpha_Q(q) = (m, SI, SD)$ iff $bool(q) = m$ and for all index and loop variables `p1`, `p2`, we have that `p1 < p2` in SI iff $q[\mathtt{p1}] < q[\mathtt{p2}]$, and `p1 = p2` in SI iff $q[\mathtt{p1}] = q[\mathtt{p2}]$. The definition is similar for SD. We present the case of one index variable `p1` and one data variable `v1`. We have that `p1 < v1` in SD if and only if $[\![\mathtt{A[p1]}]\!] < q[\mathtt{v1}]$, and `p1 = v1` in SD if and only if $[\![\mathtt{A[p1]}]\!] = q[\mathtt{v1}]$. We define the abstraction function $\alpha_R : R \to R^\alpha$ similarly.

We now define the abstract transition system. More precisely, we define $[\![\mathtt{P}]\!]^\alpha = (e^\alpha, T^\alpha, f^\alpha)$ using $[\![\mathtt{P}]\!]^t$ as follows: Let $T^\alpha = (R^\alpha, \delta^\alpha, F^\alpha)$. The transition relation $\delta^\alpha \subseteq R^\alpha \times R^\alpha$ is defined in a standard way: $\delta^\alpha(r_1^\alpha, r_2^\alpha)$ iff there exist r_1, r_2 and a pair $(a, d) \in (\Sigma \times D)^*$, such that $\delta(r_1, (a, d), r_2)$ and $\alpha(r_1) = r_1^\alpha$ and $\alpha(r_2) = r_2^\alpha$. The set F^α of final states is defined as follows: $r^\alpha \in F^\alpha$

iff there exists $r \in F$ and $\alpha(r) = r^\alpha$. The relation $\delta^{\alpha *}$ denotes the transitive closure of δ^α. Given a relation e on $Q \times R$, we define its abstract version e^α on $Q^\alpha \times R^\alpha$ similarly to the definition of the abstract transition relation. Also, given a relation f on $R \times Q$, we define its abstract version f^α on $R^\alpha \times Q^\alpha$.

The following lemma is the key part of the proof. It relates reachability of a boolean state in the abstract and concrete systems.

Note first that the abstraction function does not define a bisimulation relation between the abstract and concrete states. We demonstrate this using an example. Let us consider a program `P` and let us focus on two data variables `v1` and `v2`. We set D to be $\mathbb{N}$, the set of natural numbers. 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. We do not need to consider the values of the other variables. The abstract state corresponding to r_1, $r_1^\alpha = \alpha_Q(r_1)$ is thus m, SI, SD, where SD, the order on data and index variables, includes $\mathtt{d1} < \mathtt{d2}$. The abstract state r_1^α, can transition 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.

However, we show that for each run of the abstract transition system, we can find a run in the concrete transition system leading to the same boolean state, by appropriately choosing the input array.

Lemma 5. *For all r_1^α, r_2^α in R^α,we have that $\delta^{\alpha *}(r_1^\alpha, r_2^\alpha)$ if and only if there exist $r_1, r_2 \in R$ and a word $w \in (\Sigma \times D)^*$ such that $\alpha(r_1) = r_1^\alpha$, $\alpha(r_2) = r_2^\alpha$, and $\delta^*(r_1, w, r_2)$.*

Proof. It is straightforward to prove that if there exist r_1, r_2 and w such that $\alpha(r_1) = r_1^\alpha$, $\alpha(r_2) = r_2^\alpha$, and $\delta^*(r_1, w, r_2)$ then $\delta^{\alpha *}(r_1^\alpha, r_2^\alpha)$. We only need to apply the definition of δ^α inductively.

The proof of the other implication uses induction on the length of the path from r_1^α to r_2^α that witnesses $\delta^{\alpha *}(r_1^\alpha, r_2^\alpha)$. We will also need the following notion: The relation $Gap(r, o)$ holds for $r \in R$ and $o \in \mathbb{N}$ iff for all data variables (and values pointed to by index variables) `v1`, `v2`, we have that if $r[\mathtt{v1}] > r[\mathtt{v2}]$, then $r[\mathtt{v1}] - r[\mathtt{v2}] \geq o$. The relation $\delta_k^\alpha(r_1^\alpha, r_2^\alpha)$ is defined as follows: $\delta_k^\alpha(r_1^\alpha, r_2^\alpha)$ if there exists a state $r_3^\alpha \in R^\alpha$ such that $\delta^\alpha(r_1^\alpha, r_3^\alpha)$ and $\delta_{k-1}^\alpha(r_3^\alpha, r_2^\alpha)$ for $k > 1$; and $\delta_1^\alpha = \delta^\alpha$.

We will prove the following inductive claim: If $\delta_k^\alpha(r_1^\alpha, r_2^\alpha)$, then for all r_1 such that $\alpha_R(r_1) = r_1^\alpha$ and $Gap(r_1, 2^k)$, there exists r_2 and a word $w \in (\Sigma \times D)^k$, such that $\delta(r_1, w, r_2)$, and $\alpha_R(r_2) = r_2^\alpha$.

The base case, where $k = 0$ is straightforward. For the inductive case, suppose that $\delta_k^\alpha(r_1^\alpha, r_2^\alpha)$. Then there exists a state $r_3^\alpha \in R^\alpha$ such that $\delta^\alpha(r_1^\alpha, r_3^\alpha)$ and $\delta_{k-1}^\alpha(r_3^\alpha, r_2^\alpha)$. Let r_1 be such that $\alpha(r_1) = r_1^\alpha$ and $Gap(r_1, 2^k)$. (It is easy to show that such r_1 exists for all r_1^α.) We need to find a state $r_3 \in R$ and a pair $(a, d) \in \Sigma \times D$ such that $\delta(r_1, (a, d), r_3)$, $\alpha_R(r_3) = r_3^\alpha$ and $Gap(r_3, 2^{k-1})$. This is done by case analysis of the transition $\delta^\alpha(r_1^\alpha, r_3^\alpha)$. Informally, the transition can require that the data value d of the current position (the position pointed to by the loop variable) has to be between two stored values, but as $Gap(r_1, 2^k)$ holds, we can always choose d such that we ensure that $Gap(r_3, 2^{k-1})$. We can conclude by using induction hypothesis for $\delta^{\alpha}_{k-1}(r^{\alpha}_3, r^{\alpha}_2)$. □

A boolean state m is *reachable in* $[\![P]\!]^{\alpha}$ if there exists an initial state g_I, an abstract state q^{α}_I such that $\alpha(loc(g_I)) = q^{\alpha}_I$, and states $q^{\alpha}_2 \in Q^{\alpha}$, $r^{\alpha}_1, r^{\alpha}_2 \in R^{\alpha}$ such that $(q^{\alpha}_I, r^{\alpha}_1) \in e^{\alpha}$, $\delta^{\alpha *}(r^{\alpha}_1, r^{\alpha}_2)$, $(r^{\alpha}_2, q^{\alpha}_2) \in f^{\alpha}$, and $bool(q^{\alpha}_2) = m$.

Lemma 6. *A boolean state m is reachable if and only if it is reachable in $[\![P]\!]^{\alpha}$.*

Proof. The proof uses Lemmas 4 and 5. □

Complexity The proofs of the preceding lemmas give rise to an algorithm for deciding reachability of a boolean state m. The algorithm tests reachability of m in the abstract transition system. We show that the algorithm is in PSPACE. The number of states in T^{α} depends exponentially the number of variables in the program. Furthermore, given two abstract states, r^{α}_1 and r^{α}_2, one can decide (in polynomial time in the number of variables), whether the tuple $(r^{\alpha}_1, r^{\alpha}_2)$ is in δ^{α}.

In order to show that the problem is PSPACE-hard, we can reduce SUCCINCT-REACHABILITY (see [21]) to our reachability problem. Note that the resulting instance will use only boolean variables, not data or index variables.

This completes the proof of Theorem 2.

As noted above, we presented the proof for programs without constants in D. The proof can be extended to programs with constants in a straightforward way: Let c_1 be the smallest and let c_2 be the greatest constant that a given program `P` uses. The abstract system $[\![\mathtt{P}]\!]^{\alpha}$ will need to track the values between c_1 and c_2 precisely, and track only the order between the stored values for values less than c_1 or greater than c_2. The resulting system will thus still have a finite number of states. The reachability problem can be solved in space polynomial in the number of variables and the size (number of bits) of the largest constant.

3.2 Finite data domain

In this subsection, we consider the case when the data domain D is finite. We also syntactically restrict the programs: we consider programs which do not have index variables and which do not contain expressions of the form `IE = IE` and `IE < IE`, that is the index expressions (consisting now only of loop variables) are not compared. We call these programs *index-free*. The reason we consider this restriction is that in this case, the local state needs only to store a fixed number of data values. As the data values are from a finite domain, the set of local states is finite.

We will show that in this case, we can allow the nesting depth of loops to be arbitrary while maintaining the decidability of the reachability problem.

Theorem 7. *Reachability is decidable for index-free programs if the data domain D is finite. The problem is in* EXPSPACE.

Note that if we make the data domain finite and the state of the program (apart from the loop variable) finite as well, there is a natural question about how such programs are related to finite state automata on words. Let us consider an execution of such a program. The traversal order of this execution is different from standard finite state automata, as the program reads the input array many times. The number of times `P` scans the array in fact depends also on the length of the input word, and is therefore unbounded. If n is the length of the input word, and k is the nesting depth of loops in a program `P`, then `P` scans the array n^k times.

We will show how all these traversals of the array can be simulated by a finite state system. The main idea of the construction is that even though the number of iterations through the array depends on the length of the input word, each such scan by a particular loop can be characterized by a pair of states of the program - a state in which it begins, and a state in which it ends.

Construction of a finite-state transition system. In the rest of this subsection, we fix a program `P`. A program in normal form can be seen as a sequence $f_1 L_1 f_2 L_2 \ldots f_k L_k f_{k+1}$, where f_i is a binary relation over Q representing a loop-free part of the program, and L_i is a loop, i.e. a command of the form `for i1:=1 to length(A) do P1`. We present the construction of such a sequence $\mathit{Seq}(\mathtt{P})$.

For the following commands `P`: `skip`, `v := E`, `if B then P1 else P2`, `if * then P1 else P2`, $\mathit{Seq}(\mathtt{P})$ is defined by

$$\mathit{Seq}(\mathtt{P}) = \{(\mathit{loc}(g), \mathit{loc}(g')) | (g, g') \in [\![\mathtt{P}]\!]\}.$$

Note that for the conditionals, we have that `P1` and `P2` are loop free. For loops and sequential composition we have:

- $\mathit{Seq}(\texttt{for i1:=1 to length(A) do P}) = f_1 L f_2$, where $(q, q) \in f_1$ and $(q, q) \in f_2$, for all $q \in Q$, and `L = for i1:=1 to length(A) do P`.
- $\mathit{Seq}(\mathtt{P1;P2})$ is defined as follows: If $\mathit{Seq}(\mathtt{P1}) = f_1 L_1 f_2 L_2 \ldots f_k L_k f_{k+1}$ and $\mathit{Seq}(\mathtt{P2}) = f'_1 L'_1 f'_2 L'_2 \ldots f'_{k'} L'_{k'} f'_{k'+1}$, then $\mathit{Seq}(\mathtt{P1;P2}) = f_1 L_1 f_2 L_2 \ldots f_k L_k (f_{k+1} \circ f'_1) L'_1 f'_2 L'_2 \ldots f'_{k'} L'_{k'} f'_{k'+1}$.

Given a program `P`, we construct a finite-state transition system $\mathit{FS}(\mathtt{P}) = (S, D, \delta, s_0)$. Let L be the set of all loop commands that appear in the program. Let S_P be a set of all tuples of the form $Q \times Q \times L \times Q$. The set S is then $2^{S_P} \cup \{s_0\}$, where s_0 will be the initial state.

Given two states q_1 and q_2, we say that a state s of $\mathit{FS}(\mathtt{P})$ models a triple $(q_1, \mathtt{P}, q_2)$ (denoted by $s \models (q_1, \mathtt{P}, q_2)$) iff $\mathit{Seq}(\mathtt{P}) = f_1 L_1 f_2 L_2 \ldots f_{k+1}$ and there exist $q_1^1 q_1^2 q_2^1 q_2^2 \ldots q_k^1 q_k^2 q_{k+1}^1$ such that for all i, if $1 \leq i \leq k$, then $(q_i^1, q_i^2) \in f_i$, there exists a state q such that $(q_i^2, q, L_i, q_{i+1}^1)$ is in s, $q_1^1 = q_1$ and $q_{k+1}^1 = q_2$.

A state s is called *starting* iff there exist an initial local state q_1 and a local state q_2 such that $s \models (q_1, \mathtt{P}, q_2)$ and for all tuples $(q_1, q_2, \mathtt{L}, q_3)$ in s, we have that $q_1 = q_2$.

A state s is called *ending* iff for all tuples $(q_1, q_2, \texttt{L}, q_3)$ in s, we have that $q_2 = q_3$.

The transition relation $\delta \subseteq S \times ((\Sigma \times D) \cup \{\epsilon\}) \times S$ is defined as follows. The initial state s_0 transitions on ϵ to a state s iff s is a starting state. In addition, we have that $\delta(s_1, (a,d), s_2)$ iff for all tuples $t \in s$, there exists a tuple $t' \in s'$ such that $t \xrightarrow{(a,d),s,s'} t'$, and for all tuples $t' \in s'$, there exists a tuple $t \in s$ such that $t \xrightarrow{(a,d),s,s'} t'$. The auxiliary relation $t \xrightarrow{(a,d),s,s'} t'$ is defined as follows: $(q_1, q_2, \texttt{L}, q_3) \xrightarrow{(a,d),s,s'} (q_1', q_2', L, q_3')$ iff `L = for i1 := 1 to length(A) do P`, $q_1 = q_1'$, $q_3 = q_3'$, and there exist a state q'' such that $s \models (q_2, P, q_2'')$ and $q_2' = q_2''[\texttt{A\_i1} = (a,d)]$.

Given a data word w in $(\Sigma \times D)$, we say that q_2 *is w-reachable from q_1 in $FS(P)$* iff there exist a starting state s, an ending state s', a word $w = w_1 w_2 \ldots w_l$, and states $s_1, s_2, \ldots, s_{l+1}$ such that $s \models (q_1, P, q_2)$, $s = s_1$, $s' = s_{l+1}$, and $\delta(s_i, w_i, s_{i+1})$, for all i such that $1 \leq i \leq l$.

Lemma 8. *A local state q_2 is w-reachable from q_1 in $FS(P)$ if and only if there exist states g_1 and g_2 such that $loc(g_1) = q_1$, $g_1[A] = w$, $loc(g_2) = q_2$, $g_2[A] = w$ and $(g_1, g_2) \in [\![P]\!]$.*

Proof. The proof uses induction on the nesting depth of `P`. The inductive step is proven by induction on the number of sequentially composed loops in the program. $\square$

A boolean state m is *w-reachable in $Seq(P)$* if there exist an initial local state q_I and a local state q, such that $bool(q) = m$, and q is w-reachable from q_I in $FS(\texttt{P})$.

The proof of the following lemma uses Lemma 8.

Lemma 9. *A boolean state m is reachable if and only if it is reachable in $FS(P)$.*

Lemma 9 reduces the reachability problem to the reachability problem in a finite state transition system whose size is doubly exponential in the number of variables of the program. We also have that given two states of $FS(\texttt{P})$, s_1 and s_2, and a pair $(a,d) \in (\Sigma \times D)$, it is possible to decide (in polynomial time in the number of variables), whether $\delta(s_1, (a,d), s_2)$. Therefore we have that the problem of deciding reachability is in Expspace. This concludes the proof of Theorem 7.

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 [5]. 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.

Logics on data words We define logics whose models are data words. Following [5], 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 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. Moreover, 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. The proof of the previous fact implies that 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.

Example 4. Let us consider a language L of data words such that $str(L)$, the set of string projections, is exactly the the set of all words over $\{a, b, c\}$ that contain the same number of as, bs, and cs.

It is easy to find a data automaton $\mathcal{A}$ such that $L(\mathcal{A}) = L$. The transducer 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 contain exactly one occurrence of a, one occurrence of b and one occurrence of c.

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. This is in contrast to programs that scan the array linearly from left to right. We therefore 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 automaton 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

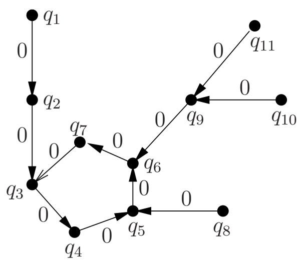

Figure 1: A connected component of a graph C_0 corresponding to an EDA $\mathcal{E}$

$(q, 0, q)$, 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. However, the proof that satisfiability of $\text{EMSO}^2(\approx, +1, \oplus 1)$ formulas is decidable uses a reduction to DAs, and is rather involved. Therefore we present a direct proof for decidability of reachability for EDAs.

Theorem 10. *Given an EDA $\mathcal{E}$, it is decidable whether $L(\mathcal{E}) = \emptyset$.*

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 L 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 $S(w)$ to Q_C, the set of states of C. Furthermore, we require that L is consistent with δ_G and δ_C, the transition functions of G and C. We define $L(0)$ 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, $L(i)$ is equal to (q', o', c') if and only if $w_i = (a, d)$, $L(i-1) = (q, o, c)$ and

- $(q', o') \in \delta_G(q, a)$,
- for the unique X such that $i \in X$ we have $c'(X) \in \delta_C(c(X), o')$,
- for X such that $i \notin X$ we have $c'(X) \in \delta_C(c(X), 0)$.

A run is *accepting* iff $L(n) = (q, o, c)$, 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 $\delta(p_1, 0) = p_2$. Every vertex in C_0 has exactly one outgoing edge (and might have multiple 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 1. 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 11. *Let L be a run of $\mathcal{E}$ on w. Let i be a position in w. Let $L(i)$ 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 state, 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 number of active classes is at most $D(C_0)$, and we conclude immediately.

Let us consider the case $i > D(c_0)$. Let $L(i)$ 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 $\mathcal{S}(w)$:

- $\mathit{pstr}(s, X) = 0^{D(C_0)}$. Let $L(i - D(C_0)) = (q', o', c')$, and let $c'(X) = v$. We can easily prove that $\delta_C^*(p, 0^e)$ is not in W, for all $e \geq D(p)$. By definition, $D(C_0) \geq D(p)$. Therefore, we can conclude that $c(X) \notin W$.
- $\mathit{pstr}(s, X) \neq 0^{D(C_0)}$. 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.

Therefore 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 [10], and thus the emptiness of multicounter automata is decidable. We use the definition of multicounter automata from [5].

Multicounter automata 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 $Q \times (\Sigma \cup \{\varepsilon\}) \times \{inc(i), dec(i)\} \times Q$. The idea is that in each step, the automaton can change its state and modify the counters, by incrementing 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.

Lemma 12. *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:

1. Simulates the transducer G using the finite state part (i.e. not the counters).

2. 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 each C_0^j. 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 11.

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

- If q, q' are cyclic states, the counter corresponding to q is decreased and the counter corresponding to q' is increased.
- If q, q' are non-cyclic state, a transition that changes the state of V is made.
- 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.
- If q is a noncyclic state and q' is a cyclic state, the transition is simulated similarly.

□

This concludes the proof of Theorem 10.

4.3 Restricted double nested loops

We will reduce the reachability problem of Restricted-`ND2` programs to the emptiness problem of EDAs. We will use the following notion: For a given program `P` and a given boolean state m, we consider a language of data words such that the execution of `P` on a word from this language ends in a global state whose boolean component is m. More precisely, 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 $(g_I, g) \in [\![\mathtt{P}]\!]$.

Theorem 13. *Reachability for Restricted-*`ND2` *programs is decidable.*

Proof. In this proof, we fix a program `P` of the following form:

```
for i1 := 1 to length(A) do
   P1;
   for j1: 1 to length(A) do P2;
   P3
```

where P1, P2, P3 are loop free programs. We present the proof for programs of this form. It can be extended for general programs using product construction techniques similar to those from proofs of Theorems 2 and 7. Recall that according to the definition of Restricted-ND2 programs, the program can test whether A[i1]=c for some constant c, but not whether A[j1]=c.

Given a boolean state m_r, we construct an EDA $\mathcal{E} = (G, C)$ such that $w \in L_{m_r}(\mathtt{P})$ iff $w \in L(\mathcal{E})$.

The task of the finite state transducer $G = (Q_G, \Sigma, \Gamma, \delta_G, q_0^G, F_G)$ is to guess a run of the program P. The output alphabet Γ consists of tuples in $\Sigma \times M \times M \times V$, where M is the set of boolean states of the program P.

The main idea of the construction 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. If a position i is marked with (a_i, m, m', v), the class automaton corresponding to class X such that $i \in X$ will verify that if the inner loop, which ran when the loop variable of the outer loop pointed to i, was started at m, then it will finish at m'.

The set V is defined as $V_C \cup V'_C \cup \{e\}$, with $e \notin V$. The set V_C is the set of all constants from D that appear in the program P. The set V'_C contains a symbol c' for each $c \in V_C$. The symbol e will represent the fact that the current input is not equal to any of the constants in the program.

First, let us summarize the effect of the loop-free subprograms P1 and P3 by relations $f_1, f_3 \subseteq M \times (\Sigma \times V) \times M$. The programs P1 and P3 can access the boolean state, read the value $[\![\mathtt{A[i].s}]\!]$, compare the value $[\![\mathtt{A[i].d}]\!]$ to constants, and modify the boolean state.

The transducer reads a word $a_1 a_2 \dots a_l \in \Sigma^*$, and produces a word $b_1 b_2 \dots b_l \in \Gamma^*$ such that:

- $b_1 = (a_1, m, m', v)$, for some m such that there exists a global state g_I such that $(\mathit{bool}(g_I), (a, v), m) \in f_1$.
- for all i such that $1 \leq i < l$, if $b_i = (a_i, m_1, m_2, v)$ and $b_{i+1} = (a_{i+1}, m'_1, m'_2, v')$, then there exist boolean states m_3, m'_1, m'_2 such that $(m_2, (a_i, v), m_3) \in f_3$ and $(m_3, (a_{i+1}, v'), m'_1) \in f_1$.
- $b_l = (a_l, m, m', v)$, for some $m \in M$ and $v \in V$ such that $(m', (a_l, v), m_r) \in f_3$.
- There is an additional requirement on the fourth component of the tuple (a, m, m', v) that will enable the class automaton to verify that the position of constants has been guessed consistently. The transducer guesses a value in $V_C \cup \{e\}$, but at the rightmost position where it guesses a particular

value $v \in V_C$, it outputs v' instead of v. This enables the class automata to check that each value $v \in V_C$ has been guessed for at most one class.

It is straightforward to show that this is possible to do with a finite state transducer.

We now define the class automaton C. The position preserving class string defined by a data value d looks as follows:

$$00(a_1, m_1, m'_1, v)000(a_2, m_2, m'_2, v) \dots 0(a_l, m_l, m'_l, v)00$$

The task of the class automaton is twofold. First, it checks that if we consider only non-0 elements of the sequence and project to the fourth component of the tuple, the sequence observed is either of the form e^* or v^*v', for a constant v. This ensures that constants have been guessed consistently, i.e. that each constant has been assigned to a unique class, and at most one constant has been assigned to a class.

Second, the class automaton for a class X checks that the inner loops that ran when `i1`, the variable of the outer loop, pointed to one of the positions belonging to X, can run as the transducer has guessed. That is, if the position $i \in X$ has a tuple of the form (a_i, m_i, m'_i, v), the inner loop that started at state m_i, with the value of `i1` equal to i, will finish at state m'_i. We can use a construction similar to the one from the proof of Theorem 7 to construct a regular automaton to check this condition. The construction needs to be extended to allow for expressions that compare the value of the index variables (e.g. `i1 = j1`) expressions (which is possible to do here as the nesting depth is at most 2). □

The proof of Theorem 13 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 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 14. *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.

The following theorem shows that (unrestricted) `ND2` programs have an undecidable reachability problem. The proof is by reduction from the reachability problem of two-counter automata. Two-counter automaton has a finite set of states and two integer counters. The main difference between two-counter automata and multicounter automata presented before is that a two-counter automaton can test whether the value of a counter is equal to 0.

Theorem 15. *The reachability problem for* `ND2` *programs is undecidable.*

We omit the proof of the theorem in the interest of space. We note only that the proof shows that reachability 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 programs obtained by adding access to order on the data domain and adding data or index variables to Restricted-`ND2` programs. We show that if we add order on the data domain as well as at least one data variable, the reachability problem becomes undecidable.

Proposition 16. *Reachability for Restricted-*`ND2` *programs that use order on D and at least one data variable is undecidable.*

Proof. The proof is by reduction from the Post's Correspondence Problem, similar to proof of Proposition 21 of [5]. □

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. An $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$ formula φ defines a set of data words for which it holds. An EDA $\mathcal{E}$ defines the language $L(\mathcal{E})$.

We define the corresponding notion for programs using a final state. Let m be a boolean state of a program `P`. Recall that 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 `P` *accepts* the language $L_m(\mathtt{P})$, where m is the final state.

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

Proposition 17. *EDAs and $EMSO^2(\approx, +1, \oplus 1)$ are equally expressive.*

Proof. The fact that EDAs are at least as expressive follows from two facts mentioned in Section 4.2. First, the logic $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$ and data automata are equally expressive, and second, for each DA there exists an EDA that accepts the same language on data words.

To show that $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$ is at least as expressive as EDAs, we present a construction that given an EDA $\mathcal{E}$ constructs an $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$ formula φ such that for all words $w \in D^*$, $w \models \varphi$ iff $w \in L(\mathcal{E})$.

First, we recall a result of [5] that states that $\mathrm{EMSO}^2(\approx, +1, \oplus 1)$ and $\mathrm{EMSO}^2(\approx, <, +\omega, \oplus 1)$ are expressively equivalent. It is thus sufficient to construct an

$\mathrm{EMSO}^2(\approx, <, +\omega, \oplus 1)$ formula. The construction is similar to classical simulation of finite state automata in $\mathrm{EMSO}^2(+1)$.

Due to space constraints, we present only the core part of the proof that is different from the classical construction. A formula φ that simulates an accepting run of $\mathcal{E}$ is constructed. It needs to simulate the run of the transducer, as well as the run of a priori unbounded number of copies of the class automaton. We present the simulation of the runs of copies of the class automaton C. Note that we cannot mark (via existentially quantified monadic second order variables) each position in the string with the state of all the copies of C. Instead, monadic second order variables will correspond to single states of C, and each position in a word is marked by exactly one of these state predicates. If the position p is in class X, it will be marked with a state in which the copy of C corresponding to X is at p. The task of the first order part of φ is then to verify, for each class, that the labeling encodes an accepting run of the class automaton. As part of this task, it needs to verify that a correct number of 0 positions appeared between successive class positions. If P_q and P_q' are labels on successive class positions p and p', then one needs to verify that the class automaton that ran with the position-preserving class string as input and thus saw the 0 symbols will indeed be in the state q' after processing the string of 0s followed by the symbol at position p'. The formula that verifies this condition of course depends closely on the transition relation of the class automaton. We will not present the proof for a general transition relation, but will use an illustrative example. Let us suppose that the class automaton (its 0-transitions) are as depicted in Figure 1, and let us suppose that position p is labeled by q_1 and position p' with a Γ symbol a is labeled with a some state s such that there is a transition $\delta_C(q_7, a) = s$. The formula now needs to check that the distance between p and p' is $6 + 5i$, for some i, as this would guarantee that the class automaton transitions to q_4 on the initial string. The part of the formula that checks this property is:

$$\begin{aligned}
&\forall x\ \forall y\ (x \oplus 1 = y \wedge P_q(x) \wedge P_{q'}(y)) \rightarrow \\
&\quad (\bigwedge_{1 \le k \le 5} \forall y\ ((x + k = y) \rightarrow (x \not\approx y))) \wedge \\
&\quad C_0(x) \leftrightarrow C_1(y) \wedge C_1(x) \leftrightarrow C_2(y) \wedge C_2(x) \leftrightarrow C_3(y) \wedge \\
&\quad C_3(x) \leftrightarrow C_4(y) \wedge C_4(x) \leftrightarrow C_0(y)
\end{aligned}$$

where C_0, C_1, C_2, C_3, C_4 are existentially quantified monadic second order predicates that are used for counting modulo the length of the cycle (which is 5 in the example). Note that this is an $\mathrm{FO}^2(\approx, <, +\omega, \oplus 1)$ formula. □

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 18. *Deterministic EDAs are more expressive than deterministic DAs.*

Proof. Let L be the language defined in Example 5. We showed that there is a deterministic EDA $\mathcal{E}$ such that $L(\mathcal{E}) = L$.

We now show that there is no deterministic DA $\mathcal{A} = (G_A, C_A)$ such that $L(\mathcal{A}) = L$. The proof will be by contradiction. We suppose that there is such a data automaton. As the alphabet Σ is a singleton, the Σ part of the data word is determined by the length of the word in this case. We therefore define data words only by their data part in the rest of this proof. Let k be the number of states of the transducer G_A. We consider a data word $w_1 = (d_1 d_2)^{k+1}$, where d_1, d_2 are values in D. This word is in L. There is therefore an accepting run of G_A. Let us consider the even positions in w_1. Clearly, there are two positions $2i$ and $2j$ such that G_A is in the same state at $2i$ as it is at $2j$. We now consider the words $w_2 = (d_1 d_2)^i (d_1 d_2)^{k+1-j} (d_1 d_4)^{j-i}$ and $w_3 = (d_1 d_2)^i (d_1 d_3)^{j-i} (d_1 d_4)^{k+1-j}$. Note that both w_2 and w_3 are in L and the run of the transducer G_A on both of these words is the same as on w_1, as G_A is deterministic and the Σ parts of w_1, w_2, and w_3 are the same.

Now we look at the word $w_4 = (d_1 d_2)^i (d_1 d_3)^{j-i} (d_1 d_2)^{k+1-j}$ and show that it is accepted by $\mathcal{E}$. Again, the run of the transducer is the same as for w_1. The class automaton for the class corresponding to d_1 reads the same input as was the case for w_1. The class automaton for the class corresponding to d_2 reads the same input as was the case for w_2 (here the fact that the transducer is in the same state at $2i$ and $2j$ is used), and the class automaton for the class corresponding to d_3 gets the same input as was the case for w_3. Therefore in each case, the class automaton C_A accepts its input. Thus we have reached a contradiction, as w_4 is not in L. $\square$

We show that nondeterminism adds to the expressive power of EDAs. We will use the following example from [5].

Example 6. Let $L_{\#}$ be the language of data words defined by the following properties: (1) $str(w) = a^* \$ a^*$, (2) the data value of the \$-position occurs exactly once, and each other data value occurs precisely twice — once before and once after the \$ sign, and (3) the order of data values in the first a-block is different from the order of data values in the second a-block.

It is possible to show that there exists a nondeterministic EDA for this language, but not a deterministic DA. This implies the following proposition.

Proposition 19. *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. First, we will show that Restricted-`ND2` programs are not as expressive as $\text{EMSO}^2(\approx, +1, \oplus 1)$.

Proposition 20. *Restricted-*`ND2` *programs are strictly less expressive than EMSO*$^2(\approx, +1, \oplus 1)$.

Proof. For every Restricted-`ND2` program `P` and its boolean state m, we can find an EMSO$^2(\approx, +1, \oplus 1)$ formula φ such that $w \in L_m(\texttt{P})$ iff $w \models \varphi$. The proof of Theorem 13 gives, for each Restricted-`ND2` program `P` and a boolean state m, an equivalent EDA $\mathcal{E}$. In the proof of Proposition 17, we have constructed an EMSO$^2(\approx, +1, \oplus 1)$ formula equivalent to a given EDA.

We will now show that there is a language of data words that can be specified by an EMSO$^2(\approx, +1, \oplus 1)$ formula φ, but not by a Restricted-`ND2` program. We will use Example 6. We have stated that the language $L_{\#}$ can be captured by a nondeterministic EDA, and thus by an EMSO$^2(\approx, +1, \oplus 1)$ formula. There is no Restricted-`ND2` program `P` that captures $L_{\#}$. The reason is that the programs, as opposed to transducers in DAs, cannot mark the input array in any way. □

Proposition 21. *There exists an EMSO*$^2(\approx, +1, \oplus 1)$ *property that is not expressible by an* `ND1` *program.*

Proof. Let us consider the language L of data words w such that every data value that appears in w appears at least twice. It is easy to construct a (deterministic) Restricted-`ND2` program that checks this property. The property can thus be specified in EMSO$^2(\approx, +1, \oplus 1)$.

We now show that this property cannot be specified by an `ND1` program. For the sake of contradiction, suppose that there exists an `ND1` program `P` with k index and data variables. Let us consider a word $w = w_1 w_2 \dots w_{2(k+1)}$ of length $2(k+1)$, such that corresponding data values are such that for all $i \leq k+1$, $d_{i+1} > d_i$, and there exists a d'_i such that $d_i < d'_i < d_{i+1}$. The positions greater than $k+1$ are defined by $d_{k+1+i} = d_i$. As w is in L, there is an accepting run of `P`. Let us consider this run after $k+1$ steps. At this point, there is one value d_j among the first $k+1$ values in w that is not stored in a data variable or pointed to by an index variable. Let us now construct a word w' by replacing the value at $k+1+j$ by d'_j. We can show that `P` accepts w' with the same run, even though w' is not in L. We have thus reached a contradiction. □

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 EMSO$^2(\approx, +1, \oplus 1)$. However, if we syntactically restrict `ND1` programs not to use order on D, they can be captured by 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 [18]. For every register automaton, there is an equivalent data automaton ([4]). 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 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 [18] 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 [20, 5, 4, 3]. 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 [5] 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 [20] and extended temporal logics [9]. The connection to verification of programs with unbounded data structures is the first to the best of our knowledge.

Fragments of first order logic on arrays have been shown decidable in [7, 16, 1, 6]. 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. In particular, reachability in programs (that contain loops) is not reducible to these first-order fragments.

Static analysis of programs that access arrays is an active research area, with recent results including [12, 15, 1]. The approach consists in finding inductive invariants for loops using abstraction methods, such as abstract domains that can represent universally quantified facts [15] and a predicate abstraction approach to shape analysis [1]. In contrast, our results yield decision procedures for array-accessing programs, with an interesting feature in the context of previous work being that our method does not need to discover the loop invariants explicitly. However, the methods based on abstraction are applicable to a richer class of programs.

6 Conclusion

We have presented decision procedures for reachability for classes of array-accessing programs. The arrays considered are unbounded in length and have elements from a potentially infinite ordered domain. For programs with non-nested loops, we showed that the problem is PSPACE-complete, i.e. it is in the same complexity class as the reachability problem for boolean programs, which is used in standard software verification tools. Therefore our decision procedure, coupled with abstract interpretation techniques, can potentially be the basis of a software model checking tool that handles data structures with potentially unbounded size.

Furthermore, we have shown that if the data domain is bounded, the problem is decidable in EXPSPACE for programs with arbitrary nesting of loops (but without index variables). We have established connection to well-studied logics ($\mathrm{EMSO}^2(\approx, +1, \oplus 1)$) and automata on data words, and we have shown that the reachability problem is decidable for programs with doubly-nested loops, under some restrictions.

We will investigate the extensions of our decidability results to classes of programs that access a single array. These extensions include (1) programs accessing data structures other than the array, (2) programs that modify the data structure, (3) program accessing more than one data structure, and finally, (4) program with procedures. The extension to linked lists seems straightforward. We will study the extension to data structures with more successors, such as trees. The proof of decidability of the reachability problem for programs with non-nested loops can be extended to programs that modify the contents of the array. However, the proofs of the other other two main decidability results cannot be extended in a straightforward way, and the question of decidability remains open.

We also plan to implement the decision procedure introduced in Section 3.1 and to test its performance, on one hand on programs that directly fall within the fragment (such as Java methods from J2ME midlets for mobile devices, as well as from standard Java library), and, using appropriate abstraction techniques, on larger programs.

References

[1] I. Balaban, A. Pnueli, and L. Zuck. Shape analysis by predicate abstraction. In *Proc. of VMCAI'05*, pages 164–180, 2005.

[2] T. Ball and S. Rajamani. The SLAM project: debugging system software via static analysis. In *Proc. POPL'02*, pages 1–3, 2002.

[3] H. Björklund and M. Bojanczyk. Shuffle expressions and words with nested data. In *Proc. of MFCS'07*, pages 750–761, 2007.

[4] H. Björklund and T. Schwentick. On notions of regularity for data languages. In *Proc. of FCT'07*, pages 88–99, 2007.

[5] M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *Proc. of LICS'06*, pages 7–16, 2006.

[6] A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu. Rewriting systems with data. In *Proc. of FCT'07*, pages 1–22, 2007.

[7] A. Bradley, Z. Manna, and H. Sipma. What's decidable about arrays? In *Proc. of VMCAI'06*, pages 427–442, 2006.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252, Los Angeles, California, 1977.

[9] S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. In *Proc. of LICS '06*, pages 17–26, 2006.

[10] J. Gischer. Shuffle languages, Petri nets, and context-sensitive grammars. *Commun. ACM*, 24(9):597–605, 1981.

[11] N. Globerman and D. Harel. Complexity results for two way and multipebble automata and their logics. *Theoretical Computer Science*, 169(2):161–184, 1996.

[12] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Proc. of POPL '05*, pages 338–350, 2008.

[13] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *Proc. of CAV'07*, pages 72–83, 1997.

[14] B. Gulavani, T. Henzinger, Y. Kannan, A. Nori, and S. Rajamani. Synergy: a new algorithm for property checking. In *FSE'06*, pages 117–127, 2006.

[15] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Proc. of POPL '08*, pages 235–246, 2008.

[16] P. Habermehl, R. Iosif, and T. Vojnař. What else is decidable about integer arrays? In *Proc. of FoSSaCS'08*, pages 474–489, 2008.

[17] T. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proc. of CAV'02*, pages 526–538, 2002.

[18] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[19] R. Lipton. The reachability problem requires exponential space. Technical Report Dept. of Computer Science, Research report 62, Yale University, 1976.

[20] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3):403–435, 2004.

[21] C. Papadimitriou. *Computational Complexity.* Addison-Wesley Publishing, Reading, MA, USA, 1994.