Algorithmic Analysis of Array-Accessing Programs[1,2]

RAJEEV ALUR
University of Pennsylvania
PAVOL ČERNÝ
IST Austria
and
SCOTT WEINSTEIN
University of Pennsylvania

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 an 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.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

General Terms: Verification, Theory

Additional Key Words and Phrases: algorithmic software verification, arrays, automata

1. INTRODUCTION

Verification questions concerning programs are undecidable in general. However, for finite-state programs — programs whose data variables range over finite types such

Rajeev Alur, Department of Computer and Information Science, University of Pennsylvania, 3330 Walnut Street, Philadelphia, PA 19104, USA. Email: alur@cis.upenn.edu
Pavol Černý, Insitute of Science and Technology Austria, Am Campus 1, A3400 Klosterneuburg, Austria. Email: pavol.cerny@ist.ac.at
Scott Weinstein, Department of Philosophy, University of Pennsylvania, 249 South 36th Street, Philadelphia, PA 19104, USA. Email: weinstein@cis.upenn.edu
[1]Preliminary version of this paper appears in the Proceedings of 18th EACLS Annual Conference on Computer Science Logic, 2009.
[2]This research was supported in part by the NSF Cybertrust award CNS 0524059, by the European Research Council (ERC) Advanced Investigator Grant QUAREM, and by the Austrian Science Fund (FWF) project S11402-N23.

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 [Ball and Rajamani 2002; Henzinger et al. 2002; Gulavani et al. 2006].

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. However, adding integer variables and (Presburger) arithmetic over integers quickly leads to undecidability of verification problems. One can therefore investigate the possibility of adding only equality and order tests on integers to the language. We show that it is possible to extend the model further.

This paper focuses 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. The expressions in the program can use constants in D, and equality tests and ordering over index and data variables. There are no restrictions on operations on elements of Σ. 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 be at the core of an automatic verification tool that certifies their correctness. 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 techniques such as predicate abstraction [Graf and Saïdi 1997] 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* [Bojańczyk et al. 2006; Björklund and Schwentick 2007; Kaminski and Francez 1994]. Among different kinds of automata over data words that have been studied, *data automata* [Bojańczyk et al. 2006] 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 multicounter 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. [Bojańczyk et al. 2006] 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.

2. PROGRAMS

In this section, we define the syntax and semantics of programs that we will consider. 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`. Boolean variables are defined by the following grammar: `bvar ::= b, b1, b2, ....` Index variables are defined by the following grammar: `indvar ::= p, p1, p2, ...` and range over $\mathbb{N}$. Loop variables are defined by the following grammar: `loopvar ::= i, j, i1, i2, ...` and range over $\mathbb{N}$. We make a distinction between loop and index variables because loop variables cannot be modified outside of the loop header. Data variables are defined by the following grammar: `datavar ::= v, v1, v2, ...` and range over D. Constants in D are defined by the grammar: `dconst ::= c, c1, c2, ...`, and constants in Σ are defined by the grammar `sconst ::= s, s1, s2, ....`

Index expressions `IE` are defined by the following grammar:

```
IE ::= indvar | loopvar
```

Data expressions `DE` are of the form:

```
DE :: = datavar | dconst | A[IE].d
```

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

Σ-expressions `SE` are of the form:

```
SE :: = sconst | 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 | bvar
      | 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 }
      | bvar:=B | indvar:=IE | datavar:=DE
      | if B then P else P
      | if * then P else P
      | for loopvar:=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 techniques [Graf and Saïdi 1997; Ball and Rajamani 2002; Henzinger et al. 2002] 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.

2.2 Semantics

A *global state* of a program is a valuation of its boolean, loop, index and data variables, as well as of its 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 expressions (index, data and Σ expressions respectively) is a given by a partial function $[\![\mathtt{B}]\!]$ from G to $\mathbb{B}$. For index expressions, $[\![\mathtt{IE}]\!]$ is a partial function from G to $\mathbb{N}$. For data expressions, $[\![\mathtt{DE}]\!]$ is a partial function from G to D. For Σ-expressions, $[\![\mathtt{SE}]\!]$ is a partial function from G to Σ. The semantics of an expression is not defined only when there is an out-of-bounds array access. 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, $[\![\mathtt{P}]\!] \subseteq G \times G$.

— $(g, g) \in [\![\mathtt{skip}]\!]$, for all g in G

— $(g, g') \in [\![\mathtt{v:=E}]\!]$, iff $g' = g[\mathbf{v} \leftarrow [\![\mathtt{E}]\!](g)]$, 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 $l(g[\mathtt{A}]) = 0$.

— $(g, g') \in [\![\mathtt{for\ i1:=1\ to\ length(A)\ do\ P}]\!]$ iff $l(g[\mathtt{A}]) > 0$ and there exist $g_1, g_2, \ldots, g_{l+1}$, where $l = l(g[\mathtt{A}])$, such that $g_1 = g$, $g_{l+1} = g'$, and for all i such that $1 \leq i \leq l$, we have that there exists a g'_{i+1}, such that $(g_i, g'_{i+1}) \in [\![\mathtt{P}]\!]$ and $g_{i+1} = g'_{i+1}[\mathtt{i1} \leftarrow i + 1]$.

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

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 a 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.

For 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 $(g_I, g) \in [\![\mathtt{P}]\!]$.

2.3 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

```
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.

```
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.

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`.

2.4 Examples

We present two illustrative examples for the classes of programs we defined.

Example 2.1. 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 2.2. The Restricted-`ND2` program in Figure 2 tests whether there is a data value that appears twice in the array.

2.5 Program verification

Our results on decidability of reachability for `ND1` and Restricted-`ND2` can be applied to assertion checking. Consider assertions of the form `assert` φ, where φ is a propositional formula over boolean variables of the program. If such an assertion appears in the program, it can be checked as follows: introduce a new boolean variable `f` (which is set to `false` initially, as all the other boolean variables), and replace the assert statement by the assignment `f :=` φ. The assertion holds in all executions of the program if and only if a boolean state where `f == true` is reachable.

More generally, the results on decidability of reachability are applicable if the assertion for a program P is expressed as a program of the same class (`ND1` and Restricted-`ND2`) as P. This is illustrated in the next example.

Example 2.3. 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. By the definition of program semantics, `p` is initialized to 1.

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

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:

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

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 can suggest 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.

3. REACHABILITY

Boolean states. A *boolean state* is a valuation of all the boolean variables of a program. For a given global state g, the corresponding boolean state is denoted 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.

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 P and a given boolean state m, whether m is reachable. In this section, we show that reachability is decidable (PSPACE-complete) for programs with only non-nested loops (ND1-programs).

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 $\mathtt{A_v}$ that stores the value of the array element at position given by v. The main difference between local and global states is that local states do not contain a valuation of the array, the local states only store a finite number of values from the unbounded domain D.

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 $loc(g)[\mathtt{v}] = g[\mathtt{v}]$. If v is an index or a loop variable, we also have that $loc(g)[\mathtt{A_v}] = [\![\mathtt{A[v]}]\!](g)$. 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 be a program, let G be its set of global states and let V be its set of variables. Similarly, let G' and V' be the sets of states and variables of a program P'. Furthermore, we will assume that $V \subseteq V'$. We define a relation $\sim_{V,V'}$ as follows. We have that $[\![\mathtt{P}]\!] \sim_{V,V'} [\![\mathtt{P'}]\!]$ if and only if for all $g'_1, g'_2 \in G'$ it holds that $(g'_1, g'_2) \in [\![\mathtt{P'}]\!]$ iff $(\pi_{V,V'}(g'_1), \pi_{V,V'}(g'_2)) \in [\![\mathtt{P}]\!]$, where $\pi_{V,V'} : G' \to G$ as follows: $\pi_{V,V'}(g') = g$ iff g and g' agree on variables from V.

LEMMA 3.1. *Let* P *be a program, let* V *be its set of variables, and let* V' *be the set of variables of norm*(P). *We have that* $[\![P]\!] \sim_{V,V'} [\![norm(P)]\!]$. *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. *If* P *is an* ND1 *(Restricted-*ND2*) program, then norm*(P) *is an* ND1 *(Restricted-*ND2*) program.*

PROOF. Let B be a boolean expression. Let *vars*(P) (*vars*(B)) denote the set of variables in P (B). We assume that *vars*(P) ∩ *vars*(B) is empty.

Claim: $[\![$*assume*(B, P)$]\!] \sim_{V,V'} [\![$if B then P else skip$]\!]$.
We prove the claim by induction on the structure of the program P. If P is skip or v:=E, the claim follows easily. The inductive cases are similar to each other; we

present the case of the loop construct.

Let P be the program `for i1:=1 to length(A) do P1 else skip`. Let V be the set of variables and G the set of global states of `if B then P else skip`. Let V' be the set of variables of *assume*(B, P) and G' its set of states. Let us now suppose that there are two states $g'_1, g'_2 \in G'$ such that $(g'_1, g'_2) \in [\![\mathit{assume}(\texttt{B, P})]\!]$. By definition, we have $(g'_1, g'_2) \in [\![\texttt{for i1} := \texttt{1 to length(A) do } \mathit{assume}(\texttt{B, P1})]\!]$. By definition of the semantics of for-loops, we obtain that there exist states $h_1, h_2, \ldots, h_{l+1} \in G'$, where $l = l(g'_1[\texttt{A}])$, such that $h_1 = g'_1$, $h_{l+1} = g'_2$, and for all i such that $1 \leq i \leq l$, we have that there exists a h'_{i+1}, such that $(h_i, h'_{i+1}) \in [\![\mathit{assume}(\texttt{B, P1})]\!]$ and $h_{i+1} = h'_{i+1}[\texttt{i1} \leftarrow i+1]$. We now use induction hypothesis for the program P1 to obtain that there exist states $k_1, k_2, \ldots, k_{l+1} \in G$, such that (a) $\pi_{V,V'}(k_i) = h_i$, (b) $(k_i, k'_{i+1}) \in [\![\texttt{if B then P1 else skip}]\!]$, (c) $k_{i+1} = k'_{i+1}[\texttt{i1} \leftarrow i+1]$. This implies that there exist states $g_1, g_2 \in G$ such that $(g_1, g_2) \in [\![\texttt{for i1} := \texttt{1 to length(A) do } \{\texttt{if B then P1 else skip}\}]\!]$, $\pi_{V,V'}(g_1) = g'_1$, and $\pi_{V,V'}(g_2) = g'_2$. The last step consists in showing that $(g_1, g_2) \in [\![\texttt{if B then } \{\texttt{for i1} := \texttt{1 to length(A) do P1}\} \texttt{ else skip}]\!]$.

The claim above implies that we have that $[\![\mathit{norm}(\texttt{P})]\!] \sim_{V,V'} [\![\texttt{P}]\!]$. The proofs of the other parts of the lemma are by simple structural induction. □

The following theorem is one of the two main results of the paper. It states that the reachability problem for ND1 programs is PSPACE-complete, where the size of the input is the size of the program.

THEOREM 3.2. *Reachability for* ND1 *programs is decidable. The problem is* PSPACE*-complete.*

We start by describing the structure of the proof. First, the semantics of a program P in terms of a transition system T whose states are (tuples of) local states. Intuitively, this is not surprising, as an ND1 program P that contains only one loop (e.g. `for i1:=1 to length(A) do P1`) 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 (by applying P1). For sequential composition of loops, a product construction (augmented with some bookkeeping) is used.

The transition system 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 away 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. More precisely, T^α stores an *ec-order*, that is, a total order on equivalence classes of a set. (We define the notion of ec-order formally below.) 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 an ec-order on IV and SD is an ec-order 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 T^α is feasible in the concrete transition system T.

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 ec-order on data and index variables, includes v1 $<$ 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. v1 > 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 there are at least $2^k - 1$ elements between v1 and v2.

We note that the proof applies to arbitrary linear orders (i.e. dense or not) on unbounded domains. The proof only requires (as explained in the previous paragraph) the existence of strictly increasing sequences of elements of unbounded length.

Remark. In order to simplify the presentation of the proof, we make two assumptions: first, 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.) Second, we assume 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.

We now present the proof according to the above outline.

Transition system semantics. We show that for programs that contain only non-nested loops and are in normal form, the semantics $[\![\mathtt{P}]\!]$ 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 δ 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 loop. Recall that for program in normal form, loops do not appear in branches of `if` statements.

We define a function $[\![\mathtt{P}]\!]^t$ 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. 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. Consider the case of 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.

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

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

Note that for the conditionals, we have that `P1` and `P2` are loop-free. For loops and sequential composition we have the following, where $\circ$ denotes composition of relations.

—$[\![\mathtt{for\ i1:=1\ to\ length(A)\ do\ P}]\!]^t = (e, (Q, \delta, Q), f)$, where e and f are identity relations on Q, and $(q, (a, d), q') \in \delta$ if there exists a local state $q'' \in Q$ such that $(q, q'') \in [\![\mathtt{P}]\!]^t$ and $q' = q''[\mathtt{i1} = i + 1, \mathtt{A}_{\mathtt{i1}} = (a, d)]$, where $i = q[\mathtt{i1}]$. (Note that `P` is loop free.)

—$[\![\mathtt{P1;P2}]\!]^t$ is defined as follows:

(1) If $[\![\mathtt{P1}]\!]^t = f_1$ and $[\![\mathtt{P2}]\!]^t = f_2$, then $[\![\mathtt{P1;P2}]\!]^t = f_2 \circ f_1$.
(2) If $[\![\mathtt{P1}]\!]^t = f_1$ and $[\![\mathtt{P2}]\!]^t = (e_2, T_2, f_2)$, then $[\![\mathtt{P1;P2}]\!]^t = ((e_2 \circ f_1), T_2, f_2)$.
(3) If $[\![\mathtt{P1}]\!]^t = (e_1, T_1, f_1)$ and $[\![\mathtt{P2}]\!]^t = f_2$, then $[\![\mathtt{P1;P2}]\!]^t = (e_1, T_1, (f_2 \circ f_1))$.
(4) If $[\![\mathtt{P1}]\!]^t = (e_1, T_1, f_1)$ and $[\![\mathtt{P2}]\!]^t = (e_2, T_2, f_2)$, then $[\![\mathtt{P1;P2}]\!]^t = (e, T, f)$, 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 by: $R = R_1 \times R_2 \times R_2$, $((r_1, r_2, r_3), (a, d), (r_1', r_2', r_3')) \in \delta$ iff $(r_1, (a, d), r_1') \in \delta_1$, $r_2 = r_2'$, and $(r_3, (a, d), r_3') \in \delta_2$. 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 $(r_1, r_2) \in (e_2 \circ f_1)$. 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 $[\![\mathtt{P}]\!]^t = (e, T, f)$ captures the semantics of `P`.

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 $(r, w, r') \in \delta^*$ 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 $(r_i, w_i, r_{i+1}) \in \delta$.

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

state m is *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 reachable from q_I in $[\![\mathtt{P}]\!]^t$.

LEMMA 3.3. *Given a program* P*, a boolean state* m *is reachable if and only if* m *is reachable in* $[\![P]\!]^t$.

PROOF. We prove the following inductive claim. The lemma immediately follows.

Claim: A local state q_2 is reachable from q_1 in $[\![\mathtt{P}]\!]^t$ if and only if there exist states g_1 and g_2 such that $loc(g_1) = q_1$, $loc(g_2) = q_2$, and $(g_1, g_2) \in [\![\mathtt{P}]\!]$.

The proof of the claim is by induction on the structure of the program P. For the commands `skip`, `v := E`, `if B then P1 else P2`, `if * then P1 else P2` and `for i1:=1 to length(A) do P`, the lemma follows directly from the definition of $[\![\mathtt{P}]\!]^t$.

Let us consider the inductive case, the sequential composition, i.e. P is `P1;P2`. We prove the left-to-right implication, i.e. we assume that a local state q_2 is reachable from q_1 in $[\![\mathtt{P}]\!]^t$. Let us assume that $[\![\mathtt{P1}]\!]^t = (e_1, T_1, f_1)$ and $[\![\mathtt{P1}]\!]^t = (e_2, T_2, f_2)$ (This is the case when both `P1` and `P2` contain loops. The other cases are simpler.) If q_2 is reachable from q_1 in $[\![\mathtt{P}]\!]^t$, then by construction of $[\![\mathtt{P}]\!]^t$, there exists a state q_m, such that q_m is reachable from q_1 in $[\![\mathtt{P1}]\!]^t$ and q_2 is reachable from q_m in $[\![\mathtt{P2}]\!]^t$. By applying induction hypothesis twice, we obtain that there exist states g_1, g_m and g_2 such that $loc(g_1) = q_1$, $loc(g_m) = q_m$, $loc(g_2) = q_2$, and $(g_1, g_m) \in [\![\mathtt{P1}]\!]$ and $(g_m, g_2) \in [\![\mathtt{P2}]\!]$. Using the definition of $[\![\mathtt{P1;P2}]\!]$, we obtain that (g_1, g_2) is in $[\![\mathtt{P}]\!]$, which concludes the proof of left-to-right implication. The proof of the other implication is similar. □

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, as it can easily be shown that sequential composition is associative.

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. First, we define the notion of ec-order on a set of variables V. An ec-order $\rho = (\equiv_\rho, <_\rho)$ is a pair where the first component is an equivalence relation on V, and the second component is a total order on equivalence classes of $\equiv_\rho$. For variables v_1, v_2, we write $v_1 <_\rho v_2$, if v_1 belongs to an equivalence class c_1, v_2 belongs to an equivalence class c_2, and $c_1 <_\rho c_2$. For example, if $V = \{v_1, v_2, v_3\}$, all variables have a defined value, then a possible ec-order on V can be represented as $v_1 \equiv_\rho v_3 <_\rho v_2$.

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 an ec-order on IV and SD is an ec-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 $(m, \mathtt{p1} <_{SI} \mathtt{i1}, \mathtt{p1} \equiv_{SD} \mathtt{i1} <_{SD} \mathtt{d1})$. 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_R and DV_R, where there are $2k-1$ copies of each variable. Let SI_R be an ec-order on IV_R and let SD_R be an ec-order on $DV_R \cup IV_R$. 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 $\texttt{p1} <_{SI} \texttt{p2}$ iff $q[\texttt{p1}] < q[\texttt{p2}]$, and $\texttt{p1} \equiv_{SI} \texttt{p2}$ iff $q[\texttt{p1}] = q[\texttt{p2}]$. The definition is similar for SD. We present the case of one index variable `p1` and one data variable `v1`. We have that $\texttt{p1} <_{SD} \texttt{v1}$ iff $[\![\texttt{A[p1]}]\!] < q[\texttt{v1}]$, and $\texttt{p1} = \texttt{v1}$ iff $[\![\texttt{A[p1]}]\!] = q[\texttt{v1}]$. The abstraction function $\alpha_R : R \to R^\alpha$ is defined similarly.

We now define the abstract transition system. More precisely, we define $[\![\texttt{P}]\!]^\alpha = (e^\alpha, T^\alpha, f^\alpha)$ using $[\![\texttt{P}]\!]^t$ 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 $(r_1, (a, d), r_2) \in \delta$ 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.

LEMMA 3.4. *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 $r[\texttt{v1}] > r[\texttt{v2}]$, then the cardinality of the set $\{d \in D \mid r[\texttt{v1}] > d > r[\texttt{v2}]\}$ is greater than or equal to o. Furthermore, if the set D has a minimal element b, we require that for all data variables (and values pointed to by index variables) `v1`, we have that the cardinality of the set $\{d \in D \mid r[\texttt{v1}] > d > m\}$ is greater than or equal to o. There is an analogical requirement for the case when D has a maximal element.

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 $Gap(r_1, 2^k - 1)$, there exists r_2 and a word $w \in (\Sigma \times D)^k$, such that $(r_1, w, r_2) \in \delta$, 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 $Gap(r_1, 2^k - 1)$. (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 (i) $(r_1,(a,d),r_3) \in \delta$, (ii) $\alpha_R(r_3) = r_3^\alpha$ and (iii) $Gap(r_3, 2^{k-1}-1)$.

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 $Gap(r_1, 2^k-1)$ holds, we can always choose d such that we ensure that $Gap(r_3, 2^{k-1}-1)$. More precisely, we construct r_3 as follows: By definition of δ^α, there exists r_3', and a pair (a',d') such that the properties (i) and (ii) hold for these values. Let us now compare the value d' with the data values of r_1. Let $Data(r)$ be the set of data values that appear in the valuation of data variables given by a state r. Let $r_1^1, r_1^2, \ldots, r_1^z$ be all the distinct data values in $Data(r_1)$. Let us assume that $r_1^j < r_1^{j+1}$, for all j. We now proceed by case analysis. First, if d' is equal to one of the values $r_1^1, r_1^2, \ldots, r_1^z$, then the property (iii) holds for r_3', and (a',d'). This is because in this case, $Data(r_3) \subseteq Data(r_1)$. Second, let us consider the case where there exists j such that $r_1^j < d' < r_1^{j+1}$. In this case, we consider the data value d such that the cardinality of $\{d'' \in D \mid d > d'' > r_1^j]\}$ is greater than or equal to $2^{k-1}-1$, and the cardinality of $\{d'' \in D \mid r_1^{j+1} > d'' > d]\}$ is also is greater than or equal to $2^{k-1}-1$. Finding such a value d is possible, as we have that $Gap(r_1, 2^k-1)$. Let us now consider a', d and r_3 such that $(r_1,(a',d),r_3) \in \delta$ (such an r_3 exists, as we have $(r_1,(a',d'),r_3') \in \delta$). We now have that all three properties (i), (ii), and (iii) hold for a', d and r_3, because $Data(r_3) \setminus \{d\} \subseteq Data(r_1)$, and because of how d was chosen. The value d is chosen analogically if d' is smaller (greater) then the smallest (greatest) value in the set $Data(r_1)$.

As we constructed r_3 such that the property (iii) holds, we can conclude by using induction hypothesis for $\delta^\alpha_{k-1}(r_3^\alpha, r_2^\alpha)$. □

A boolean state m is *reachable in* $[\![P]\!]^\alpha$ if there exists an initial state g_I, an abstract state q_I^α such that $\alpha(loc(g_I)) = q_I^\alpha$, and states $q_2^\alpha \in Q^\alpha$, $r_1^\alpha, r_2^\alpha \in R^\alpha$ such that $(q_I^\alpha, r_1^\alpha) \in e^\alpha$, $\delta^{\alpha *}(r_1^\alpha, r_2^\alpha)$, $(r_2^\alpha, q_2^\alpha) \in f^\alpha$, and $bool(q_2^\alpha) = m$.

The next lemma follows from Lemmas 3.3 and 3.4.

LEMMA 3.5. *Given a program* P*, a boolean state m is reachable if and only if it is reachable in $[\![P]\!]^\alpha$.*

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 number of states will be polynomial in the largest constant c_2).

PROOF (OF THEOREM 3.2). Using Lemma 3.5, we reduce the reachability problem for ND1 programs to a reachability problem in a finite state system. Reachability in a finite state system is in NLOGSPACE. This means that the reachability problem for ND1-programs is in PSPACE, as the number of states in T^α is exponential in the number of variables in the program. Furthermore, given two abstract states, r_1^α and r_2^α, one can decide (in polynomial time in the number of variables), whether the tuple (r_1^α, r_2^α) is in δ^α.

In order to show that the problem is PSPACE-hard, we can reduce Succinct-Reachability (see [Papadimitriou 1994]) to our reachability problem. Note that the resulting instance will use only boolean variables, not data or index variables.

A succinct representation of a graph with n nodes, where $n = 2^b$ is a power of two, is a Boolean circuit C with $2b$ input gates. The graph represented by C, denoted G_C is defined as follows: The nodes of G_C are $\{1, 2, \ldots, n\}$. And $[i, j]$ is an edge of G_C if and only if C accepts the binary representation of the b-bit integers i and j as inputs. The Succinct-Reachability problem is as follows: given a succinct representation of a graph, and a vertex r represented as a b-bit integer, is r reachable from vertex 1?

Given a succinct representation of a graph G_C by a circuit C, we construct a program P as follows: P consists of a single loop over an input array. The body of the loop nondeterministically encodes the edges of G_C, using $2b$ boolean variables. More precisely, there are b variables encoding the current vertex, and in the body of the loop there is a program that can be described on a higher level as follows: (i) nondeterministically choose vertex j followed by (ii) if (current == i and C[i,j]) then current = j. Each execution of the body of the loop thus amounts to traversing one edge of the graph. Furthermore, at the end of the body of the loop, there is a test whether the execution reached vertex r. If the test succeeds, a a specific boolean state m is entered. In m, a boolean variable f is set to *true* and all the other variables are set to *false*. The ND1-program can be constructed in polynomial time given the circuit C. The state m is reachable if and only if the vertex r is reachable in G_C. This completes the proof of the lower bound. □

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 [Bojańczyk et al. 2006]. 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 $(a_1, d_1)(a_2, d_2) \ldots (a_n, d_n)$. The string $str(w) = a_1 a_2 \ldots a_n$ 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 $i_1 < \ldots < i_k$, the class string $str(w, X)$ is $a_{i_1} \ldots a_{i_k}$.

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. 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 ($+1$) and order ($<$) predicates on positions in a data word are used. We consider first order logic with k variables, denoted by FO^k. Similarly, EMSO^k stands for FO^k formulas preceded by a number of existential quantifications over sets of word positions.

The results in this paragraph are from [Bojańczyk et al. 2006]. 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.

Example 4.1. Consider the following data automaton $\mathcal{A}$. 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. For $\mathcal{A}$, we have 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.

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:

```
for i:= 1 to length(A) do
  for j:= 1 to length(A) do {
    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. The inner loop tests `(A[i].d=A[j].d)` and can thus be seen operating in a similar setting as the class automaton of a DA. The analogy does not quite work, because the class string is obtained by simply erasing all positions from other classes, whereas the inner loop scans the whole array. 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 4.2. 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 $(q, 0, q)$, 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 4.2 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 gives an intuitive reason why emptiness is decidable for EDAs. This is the second main result of the paper.

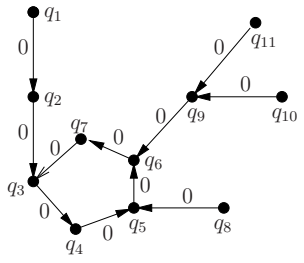

Fig. 3. A connected component of a graph C_0 corresponding to an EDA $\mathcal{E}$

THEOREM 4.3. *Given an EDA $\mathcal{E}$, it is decidable whether $L(\mathcal{E}) = \emptyset$.*

PROOF. Given an EDA $\mathcal{E}$, we construct a multicounter automaton V, such that $L(\mathcal{E})$ is empty if and only if $L(V)$ is empty. The emptiness problem is decidable for multicounter automata. The proof in [Bojańczyk et al. 2006] of decidability of emptiness of data automata also uses reduction to emptiness of multicounter automata. We extend the reduction.

We start by describing a more operational view of EDAs. 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 $(Q_C, \Gamma, \delta_C, q_0^C, F_C)$. 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 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 4.4. *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 $\mathcal{S}(w)$:

—$\mathit{pstr}(s, X) = 0^{D(C_0)}$. 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 $c(X) \notin W$.

—$\mathit{pstr}(s, X) \neq 0^{D(C_0)}$. 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 [Gischer 1981], and thus the emptiness of multicounter automata is decidable. We use the definition of multicounter automata from [Bojańczyk et al. 2006].

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 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. Nevertheless, by decrementing a counter v times and incrementing it again afterward it can check that the value of that counter is at least v.

A *configuration* of a multicounter automaton is a tuple $(q, (c_i)_{i \leq n})$, where $q \in Q$ is the current state and $c_i \in \mathbb{N}$ is the value of the counter i. A transition $(p, \epsilon, \mathit{inc}(i), q) \in \delta$ can be applied if the current state is p. For $a \in \Sigma$, a transition $(p, a, \mathit{inc}(i), q) \in \delta$ can be applied if furthermore the current letter is a. In the successor configuration, the state is q, while each counter value is the same as

before, except for counter i, which now has value $c_i + 1$. Similarly, a transition $(p, a, dec(i), q) \in \delta$ with $a \in \Sigma \cup \{\epsilon\}$ can be applied if the current state is p, the current letter is a, if $a \in \Sigma$, and counter i is non-zero. In the successor configuration, all counter values are unchanged, except for counter i, which now has value $c_i = c_i - 1$. A run over a word w is a sequence of configurations that is consistent with the transition function. A run is accepting if it starts in the state q_I with all counters empty and ends in a configuration where all counters are empty and the state is final.

LEMMA 4.5. *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 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 4.4.

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:

—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 Lemma 4.5. □

Lemma 4.5 reduces the emptiness problem for EDAs to the emptiness problem for multicounter automata. As the latter is decidable, we can conclude the proof of Theorem 4.3. □

4.3 Restricted doubly-nested loops

Language of a program.. Given a program and a boolean state m, 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 $(g_I, g) \in [\![\mathtt{P}]\!]$. In order to define the language of a program, we extend the notion of a program by designating one of its boolean state as the *final state*. The language $L(\mathtt{P})$ is defined by $L(\mathtt{P}) = L_m(\mathtt{P})$ where m is the final state. We say that a program P *accepts* the language $L(\mathtt{P})$.

THEOREM 4.6. *Reachability for Restricted-*ND2 *programs is decidable.*

PROOF. 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.

We present the proof for programs where the loops are not sequentially composed. We assume 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 Restricted-ND2 programs (which allow sequential composition of loops) using product construction techniques similar to those from the proof of Theorem 3.2. Similarly to the proof of Theorem 3.2, we

assume that 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.) Recall that according to the definition of Restricted-ND2 programs, P2 must be of the following form: `if A[i1].d=A[j1].d then P21 else P22`, where P22 cannot contain `A[i1].d` or `A[i1].s`.

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 transducer G is to guess an accepting run of the outer loop. The output alphabet Γ of the transducer consists of tuples in $\Sigma \times M \times M \times V$, where M is the set of boolean states of the program P. 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.

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'.

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 $[\![\mathtt{A[i1].s}]\!]$, compare the value $[\![\mathtt{A[i1].d}]\!]$ 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 $(bool(g_I), (a_1, v), m) \in f_1$.

—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.

We now define the class automaton C. Its task is to check that the inner loop can be executed in a way that is consistent with the guess of the transducer. The position preserving class string defined by a data value d looks as follows:

$$00(a_1, m_1, m'_1, v_1)000(a_2, m_2, m'_2, v_2) \dots 0(a_o, m_o, m'_o, v_o)00$$

In more details, 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. It is not difficult to construct a regular automaton for this condition. □

The proof of Theorem 4.6 gives rise to 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 [Lipton 1976].

Proposition 4.7. *The reachability problem for multicounter automata can be reduced in polynomial time to the reachability problem for Restricted-*`ND2` *programs.*

Proof. Given a multicounter automaton A, we construct a Restricted-`ND2` program `P` operating on data words that encode runs of A. The proof is similar to the proof of Theorem 14 from [Bojańczyk et al. 2006].

More precisely, we will construct a program `P` with a boolean state m such that L_m(`P`) is non-empty if and only if there is an accepting run of A, i.e. iff $L(A)$ is non-empty.

Let A be defined by the tuple $(Q_A, \Sigma_A, k_A, \delta_A, q_I^A, F_A)$. The array of the program P is a word on $\Sigma_A \times D$. The set Σ_A is defined as $Q_A \cup \{I_j \mid 1 \leq j \leq k_A\} \cup \{D_j \mid 1 \leq j \leq k_A\}$. I_j models the increase operation of the counter j, and similarly, D_j models the decrease operation of the counter j. A transition $(q_1, inc(j), q_2) \in \delta_A$ is encoded by having q_1, I_j and q_2 as Σ_A values in successive positions. A transition $(q_1, dec(j), q_2) \in \delta_A$ is encoded by having q_1, D_j and q_2 as Σ_A values in successive positions.

The program `P` consists of two sequentially composed parts. The first part is a single non-nested loop that examines only the Σ_A part of the data word, and check whether: (a) the first symbol is the initial state of A, (b) the word encodes transitions in δ_A, and (c) the last position contains a state in F_A. The second part uses data values to check that each decrement matches exactly one previous decrement, and each increment matches exactly one subsequent decrement, that is, the counters are never less than zero, and are equal exactly to zero at the end of the computation. This is ensured by requiring that each occurrence of I_j has a different data value, while each occurrence of D_j has the same data value as exactly one preceding I_j.

It is easy to write a Restricted-`ND2` program (of polynomial size) that checks the second part above.

```
b:=false;
```

```
b1:=true;
for i:= 1 to length(A) do {
  if (A[i].s = I_j) {
    for j:= 1 to length(A) do {
       b1 := checkForD_j
    }
  }
  ...
}
if b1:=true then b:=true;
```

The code fragment above shows the core structure of the program `P`. The reachability question we will ask is whether a state where `b` is `true` is reachable. The variable `b` is set to true at the end of the program if `b1` is true, which will be the case only if no error was found during the tests such as $\texttt{checkForD}_\texttt{j}$ above. Note that $\texttt{checkForD}_\texttt{j}$ denotes a few lines of code, it is not a procedure call. This particular test is run if the Σ_A part of the current element in the outer loop is equal to I_j and the test checks whether the equivalence class of the current element contains exactly two elements, with the other element at a position greater than the current value of `i` (no order tests need to be used, this can be checked by switching a boolean variable when `i = j` holds) and its Σ_A part contains D_j. □

4.4 Undecidable extensions

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

THEOREM 4.8. *The reachability problem for* ND2 *programs is undecidable.*

PROOF. The proof is by reduction from the reachability problem of two-counter automata [Minski 1962]. We note that the proof also implies that the reachability problem is undecidable even for ND2 programs that do not access the order on the data domain and do not use index or data variables.

Two-counter automaton has a finite set of states and two integer counters. The main difference between two-counter automata and multicounter automata is that a two-counter automaton can test whether the value of a counter is equal to 0. More precisely, a two-counter machine is a tuple (Q, δ, q_0, F), where Q is a set of states, $q_0 \in Q$ is an initial state and $F \subseteq Q$ is a set of final states. The transition relation δ is a subset of $Q \times ((\{+\} \times \{1,2\} \times Q) \cup (\{-\} \times \{1,2\} \times Q \times Q))$. A configuration is a tuple in $Q \times \mathbb{N} \times \mathbb{N}$ representing the current state and current values of the two counters. At each step, the machine can either increment one of the counters and transition to a new state, or (try to) decrement one of the counter and transition to one of two possible new states depending on whether the value of the counter was 0. The automaton accepts by final state. It is well-known that the nonemptiness problem for two-counter automata is undecidable [Minski 1962; Lambek 1961].

Given a two-counter automaton A, we construct an ND2 program `P` that has a boolean state m such that for all data words w, we have that $w \in L_m$ iff w encodes runs of A. A configuration (q, i, j) is encoded as a data word as follows: The Σ

part of the data word will be of the form

$$(\#q\$\{a,b\}^{h_1}\$\{a,b\}^{h_2})^*$$

where $\Sigma = \{\#, \$\} \cup Q \cup \{a, b\}$. The first counter is represented between the first and second \$ symbols and its value is given by the number of times the symbol a occurs. Similarly, the second counter is represented between the second and third \$ symbols. A run is a sequence of configurations, and it will be encoded as a sequence of encodings of configurations. Note that the maximal value the counters can have in a run (denoted by h_1, h_2 above) does not change during the run in this encoding.

The ND2 program P checks that the run has the form above, the first configuration has an initial state and the values of counters are zero, the last configuration in the input word has a final state, and that the transition relation δ is respected at every step. We describe how the program checks the last condition.

Let us consider two successive configurations, represented as follows:

$$\ldots \#q\$C_1\$C_2\#q'\$C_1'\$C_2'\# \ldots$$

Let us describe how the program check that the length of the string C_1 is the same as the length of the string C_1'. The program first checks that every data value that appears in the C_1 part appears there exactly once, and appears exactly once in the C_1' part. Similarly, every data value that appears in the C_1' part should appear there exactly once, and should appear exactly once in the C_1 part. The program also checks that every data value d in the C_1 and C_1' parts of the data word appears in pairs (a, d) or (b, d), and not in pairs with other symbols from Σ.

The program guesses a transition and checks that it matches the two successive configurations. We proceed by case analysis on the form of tuples in the transition relation. Let us suppose that the transition guessed is given by the tuple $(q_1, +, 1, q_2)$. The automaton checks that $q = q_1$, $q' = q_2$, the value of the first counter (represented by C_1) has increased by one and the value of the second counter has not changed. We show how a program can check that the value of the first counter increased by one:

—If a data value d appeared in the C_1 part in a pair (a, d), then it appears in the C_1' part in a pair (a, d).

—Let d be the leftmost data value d that appears in the C_1 part in a pair (b, d). The program checks that it appears in the C_1' part in a pair (a, d).

—For all other data values that appear in the C_1 part in a pair (b, d), the program checks that they appear in a pair (b, d) in the C_1' part.

It is easy to see that an ND2 program can check these conditions. In order to check the first condition in the list above, we can construct an ND2 program as follows: Let `i` be the loop variable of the outer loop. The boolean state of the program keeps track of whether the `A[i].s` is #, or is in the C_1 part or in the C_2 part. If it's in the C_1 part and `A[i].s` is a, the inner loop is used to find the data value `A[i].d` in the C_1' part. Once the inner loop finds this value, say at position j, it checks whether $A[j].s$ is a. (Note that the inner loop accesses the `A[j].s` value even when `A[i].d` is not equal to `A[j].d`, thus violating the syntactical restriction from the definition of Restricted-ND2 programs.) The program thus determines that the

number of a's has increased by one. Checking that the value of the second counter has not changed can be done similarly. Conformance to other transition tuples can be verified analogically. Therefore we can conclude that for all words w, we have that $w \in L_m(\mathtt{P})$ if and only if w represents an accepting run of the two-counter automaton. This concludes the proof of Theorem 4.8. □

There are three restrictions in the definition of Restricted-ND2 programs — (i) they do not use data or index variables, (ii) they do not access the order on data values, and (iii) the syntactical restrictions on the inner loop. The proof of Theorem 4.8 shows that if we lift restriction (iii), the reachability problem becomes undecidable.

We now show that lifting restriction (i) or (ii) leads to undecidable reachability problem as well. Let `OrderND2` programs be programs with nesting depth at most 2 that do not use index or data variables. Furthermore, there 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].s` (but note that `P3` can compare `A[i].d` and `A[j].d` e.g. by `A[i].d < A[j].d`). The `DataVarND2` programs are defined similarly, the only difference from `OrderND2` programs is that `DataVarND2` cannot access order on the data values, but can have data and index variables, and can compare the value $A[j].d$ to the value of data variables.

PROPOSITION 4.9. *Reachability for* `OrderND2` *programs is undecidable. Reachability for* `DataVarND2` *programs is undecidable.*

PROOF. We modify the proof of Theorem 4.8. Let `i` be the loop variable of the outer loop, and let `j` be the loop variable of the inner loop, as before. Recall that in the proof of Theorem 4.8, it was necessary to access `A[j].s` even if `A[j].d` was different than `A[i].d`. This falls out of restriction given by `OrderND2` programs. We thus have to modify the encoding of the computation of the two-counter automaton. We will still represent computations as strings of the form:

$$\ldots \#q\$C_1\$C_2\#q'\$C_1'\$C_2'\# \ldots$$

The inner loop had to access the Σ value of a position even if the data values `A[j].d` and `A[i].d` are not equal, because it had to determine if it had passed the \$ sign, in order to see whether the current position is in the C_1 or C_2 part. We will now use a different encoding to enable the inner loop to determine this. The even positions in the string will encode the computation of the two-counter automaton. The odd positions (more precisely, the data values at the odd positions) encode whether the next even position has a sentinel symbol (# or \$) or a value from the set $\{q, a, b\}$ as the Σ-value. Let D_C be a set of data values, with size of D_C being at least $\max(h_1, h_2)$, where h_1 (resp. h_2) is the size of the substrings C_1 (resp. C_2). Let d_1 and d_2 be two values in D, such that d_1 is smaller than all the values in D_C and d_2 is greater than all the values in D_C (it is always possible to find such a set D_C and values d_1 and d_2, as D is infinite). The data values of odd positions will be either equal to d_1 or d_2, with d_1 indicating that the next position contains a sentinel symbol and d_2 indicating that the next position contains a symbol in

$\{q, a, b\}$. When the loop variable of the outer loop is at a position in the C_1 part or in the C_2 part position, the inner loop can test whether the odd position contains d_1 and d_2 by testing $\texttt{A[i].d} < \texttt{A[j].d}$.

The proof for the `DataVarND2` programs is similar — we again use two different values for odd positions, and test these values via equality to a data variable. The data variable is set to the first position in the word. □

4.5 Expressiveness

In this subsection, 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. For logics and automata, the notion of expressiveness is standard. For programs, recall that we defined a language corresponding to a program and in Section 4.3.

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 4.10. *EDAs and* $\text{EMSO}^2(\approx, +1, \oplus 1)$ *are equally expressive.*

PROOF. The fact that EDAs are at least as expressive as $\text{EMSO}^2(\approx, +1, \oplus 1)$ follows from two facts mentioned in Sections 4.1 and 4.2. First, the logic $\text{EMSO}^2(\approx, +1, \oplus 1)$ 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 $\text{EMSO}^2(\approx, +1, \oplus 1)$ is at least as expressive as EDAs, we present a construction that given an EDA $\mathcal{E}$ constructs an $\text{EMSO}^2(\approx, +1, \oplus 1)$ 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 [Bojańczyk et al. 2006] that states that $\text{EMSO}^2(\approx, +1, \oplus 1)$ and $\text{EMSO}^2(\approx, <, +\omega, \oplus 1)$ are expressively equivalent. It is thus sufficient to construct an $\text{EMSO}^2(\approx, <, +\omega, \oplus 1)$ formula. The construction is similar to classical simulation of finite state automata in $\text{EMSO}^2(+1)$.

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 3, 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:

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

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

PROOF. Let L be the language defined in Example 4.2. 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 two distinct 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 $k + 1 > 2i > 2j$ and 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. □

We show that nondeterminism adds to the expressive power of EDAs. We will use the following example for the proof.

Example 4.12. 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.

PROPOSITION 4.13. *Deterministic EDAs are strictly less expressive than EDAs.*

PROOF. There exists a nondeterministic EDA for the language $L_{\#}$ from Example 4.12 (we can use the DA constructed in [Bojańczyk et al. 2006] in Example 8). It remains to prove that there is no deterministic EDA that accepts the language $L_{\#}$. The proof is by contradiction. Let us assume that there exists a deterministic EDA $\mathcal{E} = (G, C)$ that accepts $L_{\#}$.

Let n be the number of states G. For the automaton C, we assume (without loss of generality) that it is deterministic. Consider the graph C_0 and its components C_0^j for $j \in [1..k]$, as defined in the proof in Theorem 4.3. For each component C_0^j let $loop(C_0^j)$ be the length of the loop in C_0^j. Let K_0 be the product of the lengths of the loops, i.e. $K_0 = \prod_{j:1\leq j\leq k} loop(C_0^j)$. Recall the definition of $D(C_0)$ from the proof of Theorem 4.3. Let K be the smallest multiple of K_0 greater than $D(C_0)$.

Let w be the word $(a, d_1)(a, d_2)...(a, d_{K(n+2)})(\$, d)(a, d_1)(a, d_2)\dots(a, d_{K(n+2)})$, where $d_1, \dots, d_{K(n+2)}, d$ are distinct. The word w is not in the language, but we show that there is an accepting run of $\mathcal{E}$ on this word. Let us consider the following $n+1$ position in the first part of the word: $K, 2\cdot K, 3\cdot K, \dots, (n+1)\cdot K$. There exist integers p_1' and p_2', such that $1 \leq p_1' < p_2' \leq n+1$ and G has the same state after reading the positions p_1 and p_2, where $p_1 = p_1' \cdot K$ and $p_2 = p_2' \cdot K$. Let v be the data word obtained by switching the data values at p_1 and p_2. We can observe that the run of G on v and w is the same. The position-preserving class strings for classes other than the classes defined by d_{p_1} and d_{p_2} are the same. Consider the position-preserving class string s_v of the word v for the class defined by the value d_{p_1}. It is of the form $0^{p_2-1}(a, d_{p_1})0^{k(n+2)-p_2}0\dots$. The corresponding string s_w in w is of the form $0^{p_1-1}(a, d_{p_1})0^{K(n+2)-p_1}0\dots$. It is easy to see that C does not distinguish between the two strings in the following sense: (a) C has the same state after reading the position p_2 of s_v as it does after reading the position p_1 of s_w, and (b) C has the same state after reading the position $K(n+2)+1$ in both s_v and s_w. (We use (a) to show (b).) We now use the same argument for the class defined by d_{p_2}. We have thus shown that the accepting run of $\mathcal{E}$ on v can be used to construct the accepting run of $\mathcal{E}$ on w, which creates a contradiction. □

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 $\text{EMSO}^2(\approx, +1, \oplus 1)$.

PROPOSITION 4.14. *Restricted-*ND2 *programs are strictly less expressive than* $EMSO^2(\approx, +1, \oplus 1)$.

PROOF. The proof of Theorem 4.6 gives, for each Restricted-ND2 program P and a boolean state m, an equivalent EDA $\mathcal{E}$. In the proof of Proposition 4.10, we have constructed an $\text{EMSO}^2(\approx, +1, \oplus 1)$ formula equivalent to a given EDA. Therefore, for every Restricted-ND2 program P and its boolean state m, we can find an $\text{EMSO}^2(\approx, +1, \oplus 1)$ formula φ such that $w \in L_m(\texttt{P})$ iff $w \models \varphi$.

We will now show that there is a language of data words that can be specified by an $\text{EMSO}^2(\approx, +1, \oplus 1)$ formula φ, but not by a Restricted-ND2 program. We will use Example 4.12. We have stated that the language $L_\#$ can be captured by a nondeterministic EDA, and thus by an $\text{EMSO}^2(\approx, +1, \oplus 1)$ formula. We show that there is no Restricted-ND2 program P that captures $L_\#$.

For a proof by contradiction, let us assume that there is a Restricted-ND2 program P and a state m, such that $L_m(\texttt{P}) = L_\#$. Without the loss of generality, we make the following two assumptions. First, we can assume that the program contains only one (nested) loop, because to every program with sequentially composed loops, we can construct an equivalent program with only one nested loop, by using a construction similar to the one in Theorem 3.2, where we combined sequentially composed loops into a single traversal. Second, we assume that the inner loop of the nested loop is deterministic, as the inner loop can be determinized using a subset construction, as for the classical finite state machines. The nondeterministic choices are then only in the outer loop.

Recall the restriction on the code of the inner loop. The body of the loop must be of the following form: `if A[i].d=A[j].d then P2 else P3`, where `i` is the loop variable of the outer loop, and `j` is the loop variable of the inner loop. Furthermore, `P3` cannot refer to `A[j]`, i.e. it does not contain occurrences of `A[j].d` or `A[j].s`. The computation of the inner loop can therefore be seen as a computation of a finite automaton I on a string over the alphabet $\Sigma \cup \{0\}$, similarly as for the class automaton of an EDA. As in the proof of Proposition 4.13, we analyze the restriction of the automaton I to the letter 0 in order to define the numbers K_0 and K as in that proof.

Let us consider a data word $w = (a, d_1)(a, d_2)...(a, d_N)(\$, d)(a, d_1)(a, d_2)\dots(a, d_N)$, for $N = K_0 + K + 2$. Let $e = K_0 + 1$, and let $f = K_0 + K + 1$. Let v be a data word obtained by switching the first two occurrences of d_e and d_f. Clearly, $w \notin L_\#$, but $v \in L_\#$. However, given an accepting execution r_v of P on v, we can construct an accepting execution r_w of P on w, and thus obtain a contradiction. We prove that given any value of `i` (the loop variable of the outer loop), and an execution r_v^i on v of the inner loop with that value of `i`, we can construct an execution r_w^i of the inner loop on w that starts and finishes in the same state as r_v^i. For values of `i` other than e, f, $N+1+e$, $N+1+f$, the execution of the (deterministic) inner loop on w is exactly the same as the execution on v. If the value of `i` is e, it can be shown that the execution r_v^f can be used. More precisely, we show that P is in the same boolean state when the execution r_v^f reaches position $N+1+e$ (resp. $N+1+f$), as P is when the execution r_w^e reaches position $N+1+e$ (resp. $N+1+f$). Similar arguments can be used when the value of `i` is $f, N+1+e, N+1+f$.

We have shown that v is in $L(\texttt{P})$, which is a contradiction. □

We also compare the expressive power of ND1 programs and EMSO$^2(\approx, +1, \oplus 1)$.

PROPOSITION 4.15. *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 [Kaminski and Francez 1994]. For every register automaton, there is an equivalent data automaton [Björklund and Schwentick 2007]. 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 [Kaminski and Francez 1994] 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 [Neven et al. 2004; Bojańczyk et al. 2006; Björklund and Schwentick 2007; Björklund and Bojańczyk 2007]. Data automata introduced in this line of research were shown to be more expressive than register automata. Furthermore, the logic EMSO$^2(\approx, +1, \oplus 1)$ was introduced, and [Bojańczyk et al. 2006] shows that EMSO$^2(\approx, +1, \oplus 1)$ and data automata are equally expressive. The reduction from 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 EMSO$^2(\approx, +1, \oplus 1)$. We show that Restricted-ND2 programs can be encoded in 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 [Bojańczyk et al. 2006]. 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. Similarly, the main difference between our result for ND1 programs and the result of [Kaminski and Francez 1994] about decidability of emptiness for register automata is that the ND1 programs can test order of the elements of the data domain.

The results on automata and logics on data words model were applied in the context of XML reasoning [Neven et al. 2004] and extended temporal logics [Demri and Lazić 2006]. The connection to verification of programs with unbounded data structures is the first to the best of our knowledge.

Deutsch et al. [Deutsch et al. 2009] 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 [Bradley et al. 2006; Habermehl et al. 2008; Balaban et al. 2005; Bouajjani et al. 2007]. 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 [Lazić 2005]. 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. As even simple properties such as reachability are undecidable in general, bulk of the literature is on abstraction. Recent results for this type of programs include [Gopan et al. 2008; Gulwani et al. 2008; Balaban et al. 2005]. The approach consists in finding inductive invariants for loops using abstraction methods, such as abstract domains that can represent universally quantified facts [Gulwani et al. 2008] and a predicate abstraction approach to shape analysis [Balaban et al. 2005]. In contrast, our results yield decision procedures for array-accessing programs. The methods based on abstraction are applicable to a richer class of programs, including programs that access more than one array, use nested loops, and write to the arrays. However, the abstraction based methods are sound but not complete. In contrast, our results lead to a sound and complete decision algorithm. Note also that the abstract domains used for examples and applications in the cited papers also track equality and order on array elements.

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. We have established connection to well-studied logics and automata on data words, and we have shown that the decidability boundary for the reachability problem lies in the class of programs with doubly-nested loops.

REFERENCES

Balaban, I., Pnueli, A., and Zuck, L. 2005. Shape analysis by predicate abstraction. In *VMCAI*. 164–180.

Ball, T. and Rajamani, S. 2002. The SLAM project: debugging system software via static analysis. In *POPL*. 1–3.

Björklund, H. and Bojańczyk, M. 2007. Shuffle expressions and words with nested data. In *MFCS*. 750–761.

Björklund, H. and Schwentick, T. 2007. On notions of regularity for data languages. In *FCT*. 88–99.

Bojańczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., and David, C. 2006. Two-variable logic on words with data. In *LICS*. 7–16.

Bouajjani, A., Habermehl, P., Jurski, Y., and Sighireanu, M. 2007. Rewriting systems with data. In *FCT*. 1–22.

Bradley, A., Manna, Z., and Sipma, H. 2006. What's decidable about arrays? In *VMCAI*. 427–442.

Demri, S. and Lazić, R. 2006. LTL with the freeze quantifier and register automata. In *LICS*. 17–26.

Deutsch, A., Hull, R., Patrizi, F., and Vianu, V. 2009. Automatic verification of data-centric business processes. In *ICDT*. 252–267.

Gischer, J. 1981. Shuffle languages, Petri nets, and context-sensitive grammars. *Commun. ACM 24,* 9, 597–605.

Gopan, D., Reps, T., and Sagiv, M. 2008. A framework for numeric analysis of array operations. In *POPL*. 338–350.

Graf, S. and Saïdi, H. 1997. Construction of abstract state graphs with PVS. In *CAV*. 72–83.

Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., and Rajamani, S. 2006. Synergy: a new algorithm for property checking. In *FSE*. 117–127.

Gulwani, S., McCloskey, B., and Tiwari, A. 2008. Lifting abstract interpreters to quantified logical domains. In *POPL*. 235–246.

Habermehl, P., Iosif, R., and Vojnař, T. 2008. What else is decidable about integer arrays? In *FoSSaCS*. 474–489.

Henzinger, T., Jhala, R., Majumdar, R., Necula, G., Sutre, G., and Weimer, W. 2002. Temporal-safety proofs for systems code. In *CAV*. 526–538.

Kaminski, M. and Francez, N. 1994. Finite-memory automata. *Theoretical Computer Science 134,* 2, 329–363.

Lambek, J. 1961. How to program an infinite abacus. *Canadian Mathematical Bulletin 4*, 295–302.

Lazić, R. 2005. Decidability of reachability for polymorphic systems with arrays: A complete classification. *ENTCS 138,* 3, 3–19.

Lipton, R. 1976. The reachability problem requires exponential space. Tech. Rep. Dept. of Computer Science, Research report 62, Yale University.

Minski, M. 1962. Recursive unsolvability of Post's problem of 'tag' and other topics in theory of Turing machines. *Annals of Mathematics 74*, 437–455.

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

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