Two-Variable Logic on Data Words[*]

MIKOŁAJ BOJAŃCZYK   CLAIRE DAVID
Warsaw University   LIAFA, Paris VII

ANCA MUSCHOLL   THOMAS SCHWENTICK
LaBRI, Bordeaux University   Dortmund University

LUC SEGOUFIN
INRIA, ENS-Cachan

October 26, 2010

Abstract

In a *data word* each position carries a label from a finite alphabet and a data value from some infinite domain. This model has been already considered in the realm of semistructured data, timed automata and extended temporal logics.

This paper shows that satisfiability for the two-variable fragment $FO^2(\sim,<,+1)$ of first-order logic with data equality test $\sim$, is decidable over finite and over infinite data words. Here, $+1$ and $<$ are the usual successor and order predicates, respectively. The satisfiability problem is shown to be at least as hard as reachability in Petri nets. Several extensions of the logic are considered, some remain decidable while some are undecidable.

1 Introduction

When developing algorithms for property validation, it can be convenient to have decidable logics at hand, that can refer to data from some unbounded domains. Examples can be found in both program verification and database management. In this paper we reconsider a data model that was investigated both in verification (related to timed languages [BPT03] and extended temporal logics [DLN07, DL09]), as well as XML reasoning [NSV04]. As in these papers, data values are modeled by an infinite alphabet, consisting of a finite and an infinite part. The logic can address the finite part directly, while the infinite part can only be tested for equality. As a first step, this paper considers simple models: words, both finite and infinite.

Our main result is that over word models, satisfiability is decidable for first-order logic with two-variables, extended by equality tests for data values. The problem becomes undecidable when more variables are permitted, or when a linear order on the data values is available, or when more equivalence relations are available.

Following [BPT03], a data word is a finite sequence of positions having each a label over some finite alphabet together with a data value from an unbounded domain. The logic admits the equality test $x \sim y$, which is satisfied if both positions x, y carry the same data value. In addition, the logic uses the linear order $<$ and the successor relation $+1$. It should be noted that in FO^2, first-order logic with only two variables, the successor $+1$ cannot be defined in terms of the order $<$. As usual, we also have a unary predicate corresponding to each letter of the finite alphabet. A typical formula would be $\forall x \forall y \ (x = y + 1) \rightarrow (a(x) \lor x \sim y)$, expressing that for every two successive positions, the right position has label a, or both positions have the same data value.

Perhaps surprisingly, we show that the satisfiability problem for $FO^2(\sim,<,+1)$ is closely related to the well known problem of reachability in Petri nets. More precisely, we show that every language formed by the projection onto the finite alphabet of word models satisfying an $FO^2(\sim,<,+1)$ sentence is recognized by an effectively obtained multicounter automaton, a model whose emptiness is equivalent to Petri net reachability. We give a 2EXPTIME reduction of satisfiability of $FO^2(\sim,<,+1)$ sentences to emptiness of multicounter automata, the latter being decidable by [May84, Kos82]. For the opposite direction we provide a PTIME reduction from emptiness of multicounter automata to satisfiability of $FO^2(\sim,<,+1)$ sentences. Since there is no known elementary upper bound for emptiness of multicounter automata (see e.g. [EN94]), the exact complexity of satisfiability for $FO^2(\sim,<,+1)$ remains a challenging question.

The decidability of $FO^2(\sim,<,+1)$ immediately implies the decidability of $EMSO^2(\sim,<,+1)$. Here $EMSO^2$ stands for formulas of FO^2 prefixed by existential quantification over sets of word positions. Without data values, $EMSO^2(+1)$ has the same expressive power as monadic second-order logic. In this sense, the decidability of $EMSO^2(\sim,<,+1)$ can be seen as an extension of the classical decidability result of monadic second-order logic over words. It should be noted however that full monadic second-order logic (and even first-order logic) over data words is known to be undecidable [NSV04].

We also show that the satisfiability problem for $FO^2(\sim,<,+1)$ remains decidable over data ω-words. In this case we no longer recognize the string projection of definable languages, but we show that it is decidable whether an $FO^2(\sim,<,+1)$ formula is satisfied in a data ω-word whose string projection is ultimately periodic, using again multicounter automata.

Then we show that our decision procedure works even when the logic is extended by predicates $\oplus 1$ and $+2, +3, \ldots$ Here $\oplus 1$ is a binary predicate, which relates two positions if they have the same data value, but all positions between them have a different data value. The $+k$ binary predicate generalizes the successor predicate $+1$ to the k-th successor.

Paper overview. The paper is organized as follows. The main result – a decision procedure for satisfiability of $FO^2(\sim,<,+1)$ sentences – and its main proof steps are stated in Section 3. The proof is presented in sections 4 - 6. The proof introduces the concept of data automaton in Section 4. There are two steps in the proof: first we show in Section 5 that each language definable in $FO^2(\sim,<,+1)$ can be recognized by a data automaton; then we show in Section 6 how emptiness for data automata can be decided using multicounter automata. In Section 7 we study complexity issues. We show that satisfiability of $FO^2(\sim,<,+1)$ is at least as hard as non-emptiness of multicounter automata. However, we show that satisfiability for $FO^2(\sim,<)$, where the successor $+1$ is not allowed, is NEXPTIME-complete. In Section 8, we extend the main decidability result: first by adding new predicates, then by considering ω-words. Finally, in Section 9, we show that the logic becomes undecidable when: a) two equivalence relations are allowed, or b) three variables are allowed (even without the order $<$); or c) a linear order on data values is included. We conclude with a discussion of the results.

Related work. Automata on finite strings of data values (without labels) were introduced in [SF94, KF94]. The automaton model used there is based on registers for storing data values and comparing w.r.t. equality. In [NSV04] register automata and pebble automata over such words were studied. Several versions of these automata (one-way/two-way, deterministic/nondeterministic/alternating) were compared. Most of the results were negative however, i.e., most models are undecidable. Register automata have also been considered by Bouyer et al. [BPT03] in the realm of timed languages. However, as their automata proceed in a one-way fashion, just as the automata in [SF94, KF94], the expressive power is limited, e.g. one cannot test whether all data values are different. In particular the projection of a language recognized by one-way register automata onto the finite alphabet is always regular. This is not the case for the logic considered in this paper.

In [DLN07] an extension of LTL was introduced which can manipulate data values using a *freeze* operator. Decidability of LTL with one *freeze* operator is obtained in [DL09] using a reduction to faulty Minsky machines. This fragment is incomparable in expressive power to $FO^2(\sim,<,+1)$ as it can only process the word left-to-right (in particular it cannot express the formula $\psi_{a,b}$ given in Example 1 below), but can express properties that $FO^2(\sim,<,+1)$ cannot. LTL can be extended

with the past temporal operators, and a syntactic fragment of past-LTL with one freeze operator equivalent to $FO^2(\sim,<,+1)$ was given in [DL09].

Restricting first-order logic to its two-variable fragment is a classical idea when looking for decidability [GO99]. Over graphs or over arbitrary relational structures, first-order logic is undecidable, while its two-variable fragment is decidable [Mor75]. However, this result neither implies the decidability of $FO^2(\sim,<,+1)$, nor the NExpTime-completeness of $FO^2(\sim,<)$, since the equivalence relation $\sim$, the order $<$ and the successor relation $+1$ cannot be axiomatized with only two variables. A related result is the NExpTime-completeness of two-variable logics over *ordered structures* with arbitrary additional binary predicates [Ott01]. This result is not helpful for us, as our linear order is a particular one (the transitive closure of the successor relation). A recent paper generalized the result of [Mor75] in the presence of one or two equivalence relations [KO05]. Again this does not apply to our context as we also have the order and the successor relation. However [KO05] also showed that FO^2 with *three* equivalence relations, without any other structure, is undecidable. This implies immediately that we cannot extend the decidability result to data words with more than two data values per position. In Section 9, we show that in our framework already *two* equivalence relations yield undecidability.

In the context of XML reasoning we considered $FO^2(\sim,+1)$ over unranked ordered data trees in [BDM$^+$06, BMSS09] and showed the decidability of the satisfiability question. For unranked ordered data trees, the predicate $+1$ actually stands for two successor predicates, one for the child axis and one for the next sibling axis. As data words are special cases of data trees, this implies the decidability of $FO^2(\sim,+1)$ over words. The complexity for data trees is in 3NExpTime but can be brought down to 2NExpTime when restricted to data words (or possibly even further down, the best lower bound we know is NExpTime). This should be contrasted with the complexity of satisfiability of $FO^2(\sim,<,+1)$, which is not known to be elementary.

On words over a finite alphabet (without data values), the $FO^2(<,+1)$ fragment of first-order logic is very well understood. A characterization in terms of temporal logic says that it is equivalent to LTL restricted to the unary operators F, G, X and their past counterparts [EVW02]. The satisfiability problem for $FO^2(<,+1)$ is NExpTime-complete with an arbitrary number of atomic predicates [EVW02] (in the presence of $+1$ the lower bound is achieved already with one atomic predicate). Moreover, satisfiability of $FO^2(<)$ is NP-complete if the number of predicates is constant [WI09]. In terms of automata, $FO^2(<)$ is equivalent to partially-ordered, two-way deterministic finite automata [STV01], while in terms of algebra the logic corresponds to the semigroup variety DA [TW98].

This paper is the journal version of [BMS$^+$06]. New results have been included and the proofs of the old ones have been considerably modified and simplified since the conference version.

2 Preliminaries

Let Σ be a finite alphabet of *labels* and D an infinite set of *data values*. A *data word* $w = w_1 \cdots w_n$ is a finite sequence over $\Sigma \times D$, i.e., each w_i is of the form (a_i, d_i) with $a_i \in \Sigma$ and $d_i \in D$. The idea is that the alphabet Σ is accessed directly, while data values can only be tested for equality. So data words are actually words over Σ endowed with an equivalence relation on the set of positions. Two positions are equivalent if their data values are equal. We write $\sim$ for this equivalence relation.

$$
\begin{matrix}
a & b & c & b & c & a & a & b & b & b & c & a & b & a & c & b & b & a & a \\
17 & 5 & 2 & 3 & 3 & 3 & 2 & 2 & 7 & 17 & 17 & 3 & 4 & 5 & 2 & 3 & 3 & 4 & 4
\end{matrix}
\qquad\qquad
\text{abcbcaabbbcabacbbaa}
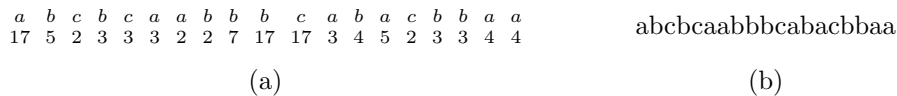$$

(a)   (b)

Figure 1: (a) A data string w and (b) its string projection

The string $str(w) = a_1 \cdots a_n$ is called the *string projection* of w. A data word v of length 19 over alphabet $\{a, b, c\}$ and with data values from $\mathbb{N}$ is shown in Figure 1 (a), its string projection

is displayed in (b). A *data language* is a set of data words, for some Σ. For a data language L, we write $\text{str}(L)$ for $\{\text{str}(w) \mid w \in L\}$.

A *class* is a maximal set of positions in a data word with the same data value. Thus, a class is just an equivalence class of the relation $\sim$. For a class with positions $i_1 < \cdots < i_k$ the *class string* is $a_{i_1} \cdots a_{i_k}$.

In the example of Figure 1, $\{1, 10, 11\}$ is the class of positions carrying the value 17. Its class string is *abc*.

Let $\text{FO}(\sim, <, +1)$ be first-order logic with the following atomic predicates: $x \sim y$, $x < y$, $x = y + 1$, and a predicate $a(x)$ for every $a \in \Sigma$. A data word can be seen as a model for this logic, where the carrier of the model is the set of positions in the word. The interpretation of $a(x)$ is that the label in position x is a. The order $<$ and successor $+1$ are interpreted in the usual way. Two positions satisfy $x \sim y$ if they have the same data value. We write $L(\varphi)$ for the set of data words that satisfy a sentence φ. A formula satisfied by some data word is called *satisfiable*. We write FO^k for formulas with at most k variables. Note that the examples below use 2 variables only.

Example 1 We present here a formula φ such that $\text{str}(L(\varphi))$ is exactly the set of words over $\{a, b\}$ that contain the same number of a and b.

- The formula φ_a says all a-positions are in different classes:

$$\varphi_a = \forall x \forall y (x \neq y \ \wedge a(x) \wedge a(y)) \to x \nsim y \ .$$

  Similarly we define φ_b.

- The formula $\psi_{a,b}$ says each class with an a also contains a b:

$$\psi_{a,b} = \forall x \exists y \big(a(x) \to (b(y) \wedge x \sim y) \big) \ .$$

  Similarly we define $\psi_{b,a}$.

- Hence, in a data word satisfying $\varphi = \varphi_a \wedge \varphi_b \wedge \psi_{a,b} \wedge \psi_{b,a}$ the numbers of a and b-labeled positions are equal.

This can be easily extended to describe data words with an equal number of a, b and c, hence a language with a string projection that is not even context-free.

For a data word w and a formula α with one free variable (which will be usually quantifier-free) we denote by $\text{Lst}_\alpha(w)$ the last position of w in which α holds, if such position exists (undefined otherwise). Similarly, $\text{Fst}_\alpha(w)$ denotes the first position of w that satisfies α, if it exists (undefined otherwise). For example, if α is the formula $b(x)$ then $\text{Lst}_\alpha(w) = 17$ and $\text{Fst}_\alpha(w) = 2$. By $\text{Llst}_\alpha(w)$ we denote the last position of w satisfying α that belongs to a different class than $\text{Lst}_\alpha(w)$, if it exists (undefined otherwise). The formula $\text{Ffst}_\alpha(w)$ is defined similarly, as the first position satisfying α, that belongs to a different class than $\text{Fst}_\alpha(w)$. In the example, $\text{Llst}_\alpha(w) = 13$ and $\text{Ffst}_\alpha(w) = 4$.

Example 2 For $a \in \Sigma$, the formula $\psi_{\text{Ffst},a}(x)$ below is satisfied precisely by the position $\text{Ffst}_{a(x)}(w)$.

$$\psi_{\text{Ffst},a}(x) = a(x) \wedge \exists y \ (y < x \ \wedge \ a(y) \ \wedge \ x \nsim y) \wedge$$
$$\forall y \ (y < x \ \wedge \ a(y)) \to [x \nsim y \wedge \forall x \ \big((x < y \ \wedge \ a(x)) \to x \sim y \big)]$$

Note how in this example the variable x is reused.

3 Decidability of $\mathrm{FO}^2(\sim,<,+1)$

The main result of this paper is the following:

Theorem 3 *Satisfiability of* $\mathrm{FO}^2(\sim,<,+1)$ *formulas over data words is decidable.*

The basic idea of the proof of Theorem 3 is to compute for each formula φ a multicounter automaton (defined in Section 6) that recognizes $\mathrm{str}(L(\varphi))$. As an intermediate step, we use a new type of finite automaton that works over data words, called a data automaton (to be defined in Section 4). Theorem 3 follows immediately from the following three statements.

Proposition 4 *Every language definable in* $\mathrm{FO}^2(\sim,<,+1)$ *is recognized by an effectively obtained data automaton.*

Proposition 5 *From each data automaton a multicounter automaton recognizing the string projection of its recognized language can be computed.*

Theorem 6 *[May84, Kos82] Emptiness of multicounter automata is decidable.*

Proposition 4 is shown in Section 5, and Proposition 5 is shown in Section 6. Regarding complexity, satisfiability of an $\mathrm{FO}^2(\sim,<,+1)$ formula is reduced in 2ExpTime to the emptiness of a multicounter automaton of doubly exponential size.

4 Automata over data words

In this section we first recall the definition of *register automata* and then we introduce *data automata*[1]. Both are automata that recognize data languages. We then state a result of [BS07] showing that data automata can simulate register automata. Both kinds of automata will be used in the proof of Theorem 3.

Register automata. A register automaton is a finite state machine equipped with a finite number of registers. These registers can be used to store data values from D. Since the two-way model is undecidable [NSV04], we consider here only one-way register automata. When processing a word, an automaton compares the data value of the current position with values in the registers with respect to equality; based on this comparison, the current state and the label of the position, the automaton can decide on its action. This model has been introduced in [KF94] where it was shown that emptiness is decidable. We present the definition in a way that fits with data words.

A *k-register automaton* has a finite state space Q and a number k of registers. Each transition of the automaton is a tuple from

$$Q \times \Sigma \times 2^{\{1,\dots,k\}} \times Q \times \{1,\dots,k,\perp\} \; .$$

A tuple (p,a,E,q,i) is interpreted as follows: if (1) the current state is p, (2) the label in the current position is a, and (3) the data value d in the current position matches a register j iff $j \in E$, then the automaton moves to the next position, changes the state to q, and stores d in register i unless $i = \perp$. The automaton accepts a data word if it has a run that begins in the designated initial state $q_0 \in Q$ and ends in one of the designated accepting states $F \subseteq Q$. In the initial configuration, the register values are undefined, and will not match any data values until overwritten during the run.

Example 7 For every k, there is a k-register automaton recognizing the following property: "a position x has label a if and only if it has the same data value as position $x + k$". Whenever the automaton reads an a it stores in its state the data value of the current position in some register. Obviously, k registers are sufficient.

---

[1]Our data automata differ from what is called data automata in [BPT03], which are essentially one-way register automata.

Data automata. We now define the second automaton model for data words, data automata. This is a new model. A *data automaton* $\mathcal{D} = (\mathcal{A}, \mathcal{B})$ consists[2] of

- a non-deterministic letter-to-letter word transducer $\mathcal{A}$ called *base automaton*, with input alphabet Σ and output alphabet Γ (letter-to-letter means that each transition reads and writes exactly one symbol), and

- a non-deterministic word automaton $\mathcal{B}$ called the *class automaton*, over input alphabet Γ.

A data word $w = w_1 \cdots w_n \in (\Sigma \times D)^*$ is accepted by $\mathcal{D}$ if there is an *accepting* run of $\mathcal{A}$ on the string projection of w, yielding an output string $b_1 \cdots b_n \in \Gamma^*$, such that, for *each class* $\{x_1, \ldots, x_k\} \subseteq \{1, \ldots, n\}$ in the data word w, with $x_1 < \cdots < x_k$, the class automaton $\mathcal{B}$ accepts the *output class string* $b_{x_1} \cdots b_{x_k}$.

In the above, the transducer $\mathcal{A}$ was chosen letter-to-letter so that the string $b_{x_1} \cdots b_{x_k}$ would be conveniently defined. In general, a transducer with output of varying length could be used, giving the same expressive power. On a similar note, the automaton $\mathcal{B}$ is only used to represent a regular language, and the definition could equivalently use a regular expression, or a deterministic automaton. On the other hand, non-determinism of the base automaton $\mathcal{A}$ is an essential feature, as the following example shows.

Example 8 Let $L_\#$ be the language of data words w fulfilling the following properties: (1) $\mathrm{str}(w) \in a^* \$ a^*$, (2) the data value of the \$-position occurs exactly once, and each other value occurs precisely twice - once before and once after \$, and (3) the order of data values in the first a-block of w is different from the order of data values in the second a-block. A data automaton that checks (1) and (2) is easy to describe (even with a deterministic base automaton). For property (3), the base automaton guesses and marks two positions $x < y$ before \$ and two positions $x' < y'$ after \$, in such a way that x, y' are both marked by 0 and y, x' are both marked by 1. Then the class automaton checks that each class is either unmarked, or has two positions marked by 0, or two positions marked by 1.

For the sake of contradiction, let us assume now that $L_\#$ is accepted by a data automaton $\mathcal{D} = (\mathcal{A}, \mathcal{B})$ with deterministic $\mathcal{A}$.

Let n be the number of states of $\mathcal{A}$ and let $w = (a, d_1) \cdots (a, d_{n+1}) \$ (a, d_1) \cdots (a, d_{n+1})$ with pairwise distinct data values d_i. Clearly, there are $i < j$ such that $\mathcal{A}$ has the same state p after reading $(a, d_1) \cdots (a, d_i)$ as after reading $(a, d_1) \cdots (a, d_j)$. Let v be the data word resulting from w by switching (the first occurrences of) (a, d_i) and (a, d_j). It is easy to observe that (a) the final state assumed by $\mathcal{A}$ is the same for v as for w and, (b) the output class strings of $\mathcal{A}$ are the same for v and for w. As $v \in L_\#$, $\mathcal{D}$ has an accepting run for v. By (a) and (b), this run can be transformed into an accepting run for w, the desired contradiction.

In the lemma below, we use the term *letter projection* for a letter-to-letter morphism, that is a morphism defined as $h : \Sigma \to \Sigma'$, where Σ, Σ' are alphabets. The following lemma is straightforward:

Lemma 9 Languages recognized by data automata are closed under union, intersection, and letter projection.

Remark: Data automata are not closed under complementation. As an example, we show that the complement L of $L_\#$ cannot be recognized by any data automaton. The proof is similar to Example 8. Assuming a data automaton $\mathcal{D} = (\mathcal{A}, \mathcal{B})$ for L (possibly with non-deterministic $\mathcal{A}$), let n be the number of states of $\mathcal{A}$, and choose w as above. Clearly, $\mathcal{D}$ has an accepting run on w. We can construct a data word v as above, and show similarly an accepting run of $\mathcal{D}$ on v.

The following lemma presents a family of data languages recognizable by data automata which will be used later in the proof.

Lemma 10 For any given regular language $L \subseteq \Sigma^*$, there is a data automaton accepting all data words for which each class string belongs to L.

Proof. The base automaton just copies its input and the class automaton checks membership in L. □

One can also verify if *some* class string belongs to L: for each position, the base automaton nondeterministically chooses to either output the input symbol or a special symbol $\perp$. It accepts if it outputs at least one non-$\perp$ symbol. The class automaton accepts the language $L \cup \perp^*$.

We will use the following proposition, which shows that data automata are at least as powerful as register automata.

Proposition 11 *[BS07] For every register automaton an equivalent data automaton can be computed in polynomial time.*

The converse does not hold, since register automata do not capture the language "all positions have different data values". A data automaton can easily recognize this language: the class automaton checks if every class string has length 1.

5 Reduction to data automata

The goal of this section is to prove Proposition 4, i.e. to transform a formula of $\mathrm{FO}^2(\sim,<,+1)$ into an equivalent data automaton of doubly exponential size.

The proof of Proposition 4 starts with a classical step for 2-variable first-order logic: transforming the given $\mathrm{FO}^2(\sim,<,+1)$ formula φ into an equivalent linear size formula in *Scott normal form* (see e.g. [GO99]). The latter is a formula

$$\exists R_1 \cdots R_n \big(\forall x \forall y \ \chi \wedge \bigwedge_i \forall x \exists y \ \chi_i \big) \ ,$$

where the R_i are unary predicates, and χ and each χ_i are quantifier-free $\mathrm{FO}^2(\sim,<,+1)$ formulas. [3] The idea behind the transformation is that each unary predicate R_i corresponds to a subformula of φ with one free variable, and marks the positions where the subformula holds. The conjuncts χ and χ_i ensure that these new predicates are consistent with each other.

Since data automata are closed under letter projection and intersection, it suffices to construct a data automaton for each of the formulas $\forall x \forall y \ \chi$ and $\forall x \exists y \ \chi_i$. This is done in Lemmas 12 and 13 below. Note that the vocabularies (alphabets) of these formulas are larger than the one of the original formula φ, since they include the new unary predicates $R_1, \ldots, R_n$.

Before describing these automata, however, we would like to clarify how words are represented. The signature of the logic uses unary predicates for labels, while automata work with a finite input alphabet. Technically, a letter of the input alphabet of the automaton is a bit vector, telling which predicates hold in a given position. This will be important for the estimation of the automaton's size: given a unary predicate P, there are exponentially many distinct letters corresponding to the cases where P is true. Thus, in order to test whether P holds or not, the automaton needs exponentially many distinct transitions.

In our constructions, we will often use some particular data automata, which we describe next.

The first data automaton checks that a predicate S_{+1} marks exactly those positions x that have the same data value as their successor $x+1$. From Example 7, we know that this check can be done by a register automaton. By Proposition 11, the register automaton can be transformed into a data automaton, which we denote as $\mathcal{D}_+$. Predicate S_{+1} will be used in the two technical lemmas 12, 13 below, when doing a case analysis over the mutual relationship between two positions x, y. In these lemmas we guess predicate S_{+1} and check it using $\mathcal{D}_+$. If the guess is correct, i.e. if $\mathcal{D}_+$ accepts, then S_{+1} tells us whether neighboring positions are data equivalent.

---

[3]It should be noted that the name Scott Normal Form usually refers to the inner first-order part. This inner formula is in general only equivalent with respect to satisfiability to the original formula.

A second kind of data automaton will be used for distinguishing non-deterministically a fixed number k of classes. The base automaton uses the output alphabet $\Gamma_k = \{\bot\}\cup(\{1,\ldots,k\}\times\{0,1\})$. For each position i, it guesses an output symbol $b_i \in \Gamma_k$. It makes sure that for each $j \in \{1,\ldots,k\}$, the symbol $(j,1)$ is chosen at most once. The class automaton accepts a word if it is either in $\bot^*$, or in $(j,1)(j,0)^*$, for some j. Therefore, for each class, either the output symbol $\bot$ is chosen for all positions, or the first output symbol is $(j,1)$ and all others are $(j,0)$, for some j. As each $(j,1)$ is used at most once, the number j uniquely identifies a class. Thus, the class automaton "knows", for each position, to which of the k classes it belongs (if any). Note that this automaton is not useful on its own, but will be used in conjunction with some other automaton, which will be verifying properties of the classes we guess.

In the following, a *type* is a conjunction of unary predicates or their negations. These unary predicates are either $a \in \Sigma$, or additional predicates like the R_i that were introduced by the transformation into Scott normal form (some more such predicates will be introduced below). Note that a type need not use all predicates of the signature.

Lemma 12 For each $\mathrm{FO}^2(\sim,<,+1)$ formula $\varphi = \forall x \forall y\ \chi$, with χ quantifier-free, an equivalent data automaton of doubly exponential size can be constructed.

Proof. It is straightforward to first bring χ into CNF, and then to rewrite φ as conjunction of exponentially many formulas of the following form:

$$\psi = \forall x \forall y\ \big((\alpha(x)\ \wedge\ \beta(y)\ \wedge\ \delta(x,y))\ \rightarrow\ \gamma(x,y)\big) \tag{1}$$

where α and β denote types, $\delta(x,y)$ is either $x \sim y$ or $x \not\sim y$, and $\gamma(x,y)$ is a disjunction of atomic and negated atomic formulas that use only the order predicates $<$ and $+1$. It suffices now to describe a data automaton for each such formula ψ. For each such formula, we will give an automaton with $O(1)$ many states. Thus, it suffices to take the intersection of all these automata, for the exponentially many conjuncts. This results in an automaton of doubly exponential size.

The proof is a case analysis on the (small number of) possible kinds of δ and γ. By w we will refer to the input data word.

We start with the simpler case where $\delta(x,y)$ is $x \sim y$. The formula ψ requires for every α-position x and β-position y belonging to the same class, that x,y are ordered according to γ. Since γ may use the successor relation, the base automaton guesses the predicate S_{+1}. The correctness of the guess is ensured by running the data automaton $\mathcal{D}_+$ (cf. beginning of the section) in parallel. When S_{+1} is guessed correctly, the formula ψ amounts to a regular property that must be satisfied by each class. Using the data automaton of Lemma 10, we obtain a data automaton for ψ.

We now turn to the case where $\delta(x,y)$ is $x \not\sim y$. Here we need to analyse the possible $\gamma(x,y)$. As in this case we have $x \neq y$, it is enough to consider the following cases:

$$false, \quad x < y, \quad x = y+1, \quad x \neq y+1, \quad x \leq y+1, \quad y = x+1 \vee x = y+1 \tag{2}$$

(or symmetrically, with x and y interchanged).

- $\gamma(x,y) = false$ means that w cannot have an α- and a β-position in different classes. In particular, there can be at most one class containing α, and this class must contain all β in the word. (Or there are no β, but this can be checked by the base automaton alone.) Using the second technique explained at the beginning of the section, a data automaton guesses (at most) one class, and checks that there are no α and β outside this class.

- $\gamma(x,y) = (x < y)$. Recall the notations Lst and Llst introduced in Section 2. Formula ψ holds if and only if (a) there is no β up to position $\mathrm{Llst}_\alpha(w)$; and (b) the β-positions between $\mathrm{Llst}_\alpha(w)+1$ and $\mathrm{Lst}_\alpha(w)$ are in the same class as $\mathrm{Lst}_\alpha(w)$. The case where $\gamma(x,y)$ is $x \leq y+1$ is handled similarly.

  Thus, the base automaton simply guesses whether w has 0, 1 or more classes containing α and marks the classes containing $\mathrm{Lst}_\alpha(w)$ and $\mathrm{Llst}_\alpha(w)$ (if defined). It then checks that (a) and (b) hold.

- $\gamma(x,y) = (x = y + 1)$ implies in particular that there are at most two classes with α or β. This is dealt with by marking the class(es), checking that α and β do not occur outside, and verifying the distance condition with the base automaton. Likewise, $y = x + 1 \lor x = y + 1$ implies that there are at most three classes with α and β, and this case can be handled analogously.

- $\gamma(x,y) = (x \neq y+1)$ is handled directly by the base automaton, after guessing (and checking in parallel) the predicate S_{+1}.

$\square$

Lemma 13 For each $\mathrm{FO}^2(\sim,<,+1)$ formula $\varphi = \forall x \exists y \, \chi$, with χ quantifier-free, an equivalent data automaton of doubly exponential size can be constructed.

Proof. First, χ can be written in disjunctive normal form

$$\bigvee_i \big(\alpha_i(x) \wedge \beta_i(y) \wedge \delta_i(x,y) \wedge \epsilon_i(x,y) \big),$$

where α_i, β_i are types, δ_i is either $x \sim y$ or $x \not\sim y$, and ϵ_i is one of $x + 1 < y$, $x + 1 = y$, $x = y$, $x = y + 1$ or $x > y + 1$. We will first eliminate the disjunction. To this end, we add for each disjunct above a new unary predicate S_i with the intended meaning that if S_i holds at a position x with α_i, then there is a y such that $\beta_i(y) \wedge \delta_i(x,y) \wedge \epsilon_i(x,y)$ holds. Formally, we rewrite each $\forall x \exists y \, \chi$ as

$$\exists S_1 \, \exists S_2 \cdots (\forall x \bigvee_i S_i(x)) \; \wedge \; \bigwedge_i \forall x \exists y \, \big(S_i(x) \rightarrow (\alpha_i(x) \wedge \beta_i(y) \wedge \delta_i(x,y) \wedge \epsilon_i(x,y)) \big) \,.$$

The subformula $\forall x \, (\bigvee_i S_i(x))$ ensures that for each x, one of the disjuncts holds. It can be rewritten equivalently as $\forall x \exists y \, (\bigwedge_i \neg S_i(x) \rightarrow \mathit{false})$.

It remains to show that we can construct a data automaton for any formula ψ of type

$$\forall x \exists y \, \big(\alpha(x) \rightarrow \; (\beta(y) \; \wedge \; \delta(x,y) \; \wedge \; \epsilon(x,y)) \big) \,.$$

For each such formula, we will give an automaton with $O(1)$ states. We then need to take an intersection of all these automata, for the exponentially many i. This gives the doubly exponential size from the statement of the lemma.

The case where $\delta(x,y)$ is $x \sim y$, is treated as in Lemma 12, by guessing S_{+1} and checking a regular condition on each class. As in Lemma 12, S_{+1} is needed when ϵ uses the successor relation.

We now consider the case when $\delta(x,y)$ is $x \not\sim y$. Note that this implies again that $x \neq y$. We denote as before by w the input data word.

Assume first that $\epsilon(x,y)$ is $x + 1 < y$ or $x > y + 1$. We describe the case of $x + 1 < y$, the other one is analogous. In this case, ψ expresses that each α-position of a data word w needs a β-position in a different class to its right, but not as its right neighbor. First there are two special cases: if w contains no β, then it contains no α either; if all β are in the same class (i.e., $\mathrm{Llst}_\beta(w)$ is undefined), then all α are outside this class and before $\mathrm{Lst}_\beta - 1$. We can use in the latter case the data automaton that marks the class with β and let the base automaton check the previous regular condition. In the last case, there are at least two classes with β. Here, notice that every α-position before $\mathrm{Llst}_\beta - 2$ is guaranteed to have an appropriate β to its right. Hence, it suffices to require the following properties: (a) w contains no α after position $\mathrm{Lst}_\beta - 1$; and (b) all α between $\mathrm{Llst}_\beta - 1$ and $\mathrm{Lst}_\beta - 2$ are not in the same class as Lst_β. This involves guessing the classes of Lst_β and Llst_β and using the base automaton for (a) and (b).

The case where $\epsilon(x,y)$ is $x + 1 = y$ or $x = y + 1$ is solved using again the predicate S_{+1}. Given this predicate, ψ reduces to a regular property that can be checked by the base automaton. $\square$

9

We would like to note that the converse of Proposition 4 does not hold, i.e. not every data automaton can be transformed into a formula. There are two reasons for this. First, a data automaton can verify arbitrary regular properties of classes, which cannot be done with first-order logic. For instance, no $\mathrm{FO}^2(\sim,<,+1)$ formula captures the language: "each class is of even cardinality". This problem can be solved by adding a prefix of monadic second-order existential quantification. Formally speaking, we consider then formulas of the logic $\mathrm{EMSO}^2(\sim,<,+1)$. Recall that the latter is the extension of $\mathrm{FO}^2(\sim,<,+1)$ by existential second-order quantification over monadic predicates in front of $\mathrm{FO}^2(\sim,<,+1)$ formulas. Note also that as far as satisfiability is concerned $\mathrm{FO}^2(\sim,<,+1)$ and $\mathrm{EMSO}^2(\sim,<,+1)$ are equivalent. However, even with $\mathrm{EMSO}^2(\sim,<,+1)$, it is difficult to write a formula that describes accepting runs of a data automaton. The problem is that describing runs of the class automaton requires comparing successive positions in the same class, which need not be successive positions in the word. That is why we consider a new predicate $\oplus 1$, called the *class successor*, which is true for two successive positions in the same class of the data word. The following result can be shown by extending the proof of Proposition 4 in a straightforward way to include $\mathrm{EMSO}^2(\sim,+1,\oplus 1)$:

Proposition 14 *Data automata and* $\mathrm{EMSO}^2(\sim,+1,\oplus 1)$ *have the same expressive power, and the translations are effective. In particular, satisfiability of* $\mathrm{EMSO}^2(\sim,+1,\oplus 1)$ *formulas over data words is decidable.*

Proof. It is easy to extend the proof of Proposition 4 to the logic $\mathrm{EMSO}^2(\sim,+1,\oplus 1)$. The other direction follows the same lines as the classical simulation of word automata by monadic second-order logic. $\square$

Another normal form is obtained by using the same ideas as in the proof of Proposition 4. Each formula of $\mathrm{EMSO}^2(\sim,<,+1)$ is equivalent to one where the FO part is a Boolean combination of formulas of the form (where α and β are types):

(a) A formula that does not use $\sim$ (i.e., an $\mathrm{FO}^2(<,+1)$ formula).

(b) Each class contains at most one occurrence of α.

(c) In each class, all occurrences of α are located strictly before all occurrences of β.

(d) In each class with at least one occurrence of α, there must be a β, too.

(e) If x is not in the same class as $x+1$, then it is of type α.

6 From data to counters

In this section we complete the proof of Theorem 3 by showing Proposition 5. We first introduce multicounter automata. A *multicounter automaton* is a finite, non-deterministic automaton extended by a number n of counters. It can be described as a tuple $(Q,\Sigma,n,\delta,q_I,F)$. The set of states Q, the input alphabet Σ, the initial state $q_I \in Q$ and final states $F \subseteq Q$ are as in a usual finite automaton. The transition relation δ is a subset of

$$Q \times (\Sigma \cup \{\epsilon\}) \times \{inc(i),\ dec(i) \mid 1 \leq i \leq n\} \times Q\,.$$

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.

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 $(p,\epsilon,inc(i),q) \in \delta$ can be applied if the current state is p. For $a \in \Sigma$, a transition $(p,a,inc(i),q) \in \delta$ 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 $c_i \neq 0$. In the successor configuration, all counter values are unchanged, except for counter i, which now has value $c_i - 1$. A *run* over a word w is a sequence of configurations that is consistent with the transition relation δ.

A run is *accepting* if it starts in the state q_I with all counters equal to zero and ends in a configuration where all counters are zero and the state is final.

The key idea in the reduction from data automata to multicounter automata, is that acceptance of data automata can be expressed using shuffles of regular languages. A word $v \in \Sigma^*$ is a *shuffle* of n words $u_1, \ldots, u_n \in \Sigma^*$ if its positions can be colored with n colors such that for every color i, the subword corresponding to color i is u_i. For a language $L \subseteq \Sigma^*$, we write Shuffle(L) for the set of words $v \in \Sigma^*$ such that for some $n \in \mathbb{N}$ and some $u_1, \ldots, u_n \in L$, the word v is a shuffle of $u_1, \ldots, u_n$.

The link between shuffle languages and multicounter automata is given by the following result:

Proposition 15 *[Gis81, Lemma (IV.6)] If $L \subseteq \Sigma^*$ is regular then Shuffle(L) is recognized by a multicounter automaton of size bounded by the size of an NFA recognizing L.*

We are now ready to prove Proposition 5, completing the proof of Theorem 3.

Proposition 5 *From each data automaton $\mathcal{D}$ one can compute in quadratic time a multicounter automaton recognizing the string projection of the language recognized by $\mathcal{D}$.*

Proof. Fix a data automaton $\mathcal{D} = (\mathcal{A}, \mathcal{B})$ and consider a word $v \in \Sigma^*$ that belongs to the *string projection* str($L(\mathcal{D})$). Then there exists a data word w with $v = \text{str}(w)$ and an accepting run of the base automaton $\mathcal{A}$ on v with output v', such that for each class c in w, the substring of v' corresponding to c is accepted by the class automaton $\mathcal{B}$. If L is the language of $\mathcal{B}$, then this implies $v' \in$ Shuffle(L).

Conversely, let $v \in \Sigma^*$ such that on input v, the base automaton $\mathcal{A}$ outputs a word v' that belongs to Shuffle(L). Then, by the definition of the shuffle operator, there are some n and words $u_1, \ldots, u_n \in L$ such that v' is a shuffle of $u_1, \ldots, u_n$. Let l be a coloring of v' witnessing the fact that v' is a shuffle of $u_1, \ldots, u_n$. Let w be the data word formed from v by assigning to each position w a data value determined uniquely by the coloring l. By definition, w is accepted by $\mathcal{D}$ and hence $v \in \text{str}(L(\mathcal{D}))$.

Hence str($L(\mathcal{D})$) is the set of words v such that some output of $\mathcal{A}$ on v belongs to Shuffle(L).

By Proposition 15, Shuffle(L) is recognized by a multicounter automaton $\mathcal{M}$ of the same size as $\mathcal{B}$. We can now compose the base automaton $\mathcal{A}$ and this automaton $\mathcal{M}$ into a multicounter automaton that recognizes the string projection of $L(\mathcal{D})$. The idea is to store in each state of the new multicounter automaton a state of $\mathcal{M}$ and a state of $\mathcal{A}$. Thus the automaton can simulate in parallel $\mathcal{A}$ and $\mathcal{M}$, using non-determinism to guess the output of $\mathcal{A}$. $\qquad\square$

7 Complexity results

The proof of Theorem 3 does not yield an immediate upper bound on the complexity of satisfiability of $\text{FO}^2(\sim,<,+1)$. From the bounds given in Lemmas 12, 13 and Proposition 5, we can only conclude that the problem can be reduced in doubly exponential time to the emptiness of multicounter automata. For the latter problem, no elementary upper bound is known.

In this section, we first show that satisfiability of $\text{FO}^2(\sim,<,+1)$ has about the same complexity as emptiness of multicounter automata, thus destroying the hope for efficient algorithms. We then initiate the search for more tractable logics by considering the fragments of $\text{FO}^2(\sim,<,+1)$ obtained by dropping $+1$ or $<$, respectively. It will turn out that satisfiability for $\text{FO}^2(\sim,<)$ is NEXPTIME-complete. For $\text{FO}^2(\sim,+1)$ we can currently only give a doubly-exponential upper bound following from our results in [BDM+06].

We first turn to the lower bound for $\text{FO}^2(\sim,<,+1)$. We show that satisfiability for $\text{FO}^2(\sim,<,+1)$ is at least as hard as non-emptiness of multicounter automata. The best lower bound known for

the latter problem is ExpSpace [Lip76] and no elementary, or even primitive recursive, upper bound is known.

Theorem 16 *Emptiness of multicounter automata can be reduced in polynomial time to the satisfiability problem of* $\mathrm{FO}^2(\sim,<,+1)$.

Proof. Given a multicounter automaton $\mathcal{A}$, we construct an $\mathrm{FO}^2(\sim,<,+1)$ formula whose models are the encodings of accepting runs of the automaton. In particular, the formula is satisfiable if and only if the automaton accepts a nonempty language. As far as emptiness is concerned, we may assume that the automaton $\mathcal{A}$ has a one letter input alphabet, so we omit the input letters from the transitions.

Let Q be the set of states, n the number of counters, and δ the transition relation of $\mathcal{A}$. We define the alphabet Σ as $Q \cup \{D_1, I_1, \ldots, D_n, I_n\}$. A transition $(p, inc(k), q) \in \delta$ is encoded by the word pI_kq, and a transition $(p, dec(k), q) \in \delta$ is encoded by pD_kq. A run ρ of $\mathcal{A}$ is encoded by a data word w_ρ such that the string projection of w_ρ is the sequence of encodings of transition steps of ρ and the data values of w_ρ are constrained in order to check the evolution of the counters along the run and enforce the validity of the run ρ in $\mathrm{FO}^2(\sim,<,+1)$ as explained below.

The $\mathrm{FO}^2(\sim,<,+1)$ formula we construct from $\mathcal{A}$ needs to enforce that: (i) the sequence of encoding transition steps is consistent with δ, (ii) the counters never drop below zero and, (iii) they are empty at the end of the computation.

It is easy to enforce (i) by a formula of $\mathrm{FO}^2(\sim,<,+1)$ of polynomial size: only the successor relation is needed here. For (ii) and (iii) we use data values to match each decrement with exactly one previous increment of the same counter and vice-versa. To do this, we constrain the data values in the following way: (1) positions labeled by the symbols $I_1, \ldots, I_n$ have pairwise distinct data values, (2) positions labeled by the symbols $D_1, \ldots, D_n$ have pairwise distinct data values, (3) each position labeled by some D_k has the same data value as a position labeled by some I_k to the left, and (4) each position labeled by some I_k has the same data value as a position labeled by some D_k to the right. The conditions (1)-(4) can easily be defined in $\mathrm{FO}^2(\sim,<,+1)$ by a formula of polynomial size.

From any run ρ satisfying (ii) and (iii) one can easily set up the data values of w_ρ such that (1)-(4) hold by matching the decrement with its corresponding increment. Conversely, assume w satisfies (i) and (1-4). By (i) the word w represents a run ρ of the automaton. The conditions (1-4) induce a bijection between I_k and D_k positions, such that any prefix of the word w contains at least as many I_k as D_k positions. Thus the corresponding run ρ satisfies (ii) and (iii).

Hence $\mathcal{A}$ has an accepting run if and only if the conjunction of the formulas above is satisfiable. □

Now we turn to the logic $\mathrm{FO}^2(\sim,<)$, where the successor relation is not allowed. Removing the successor notably improves the complexity of satisfiability, making it NExpTime-complete.

The following result should be compared with results of [EVW02], which show that satisfiability is NExpTime-complete for $\mathrm{FO}^2(<)$ over word models. In the latter logic, there is no $\sim$ relation to compare data values, so the word positions only have labels, and no data values. The upper bound we give in Theorem 17 strengthens the results from [EVW02] by showing that a NExpTime decision procedure works even if data values are introduced in the logic. Our lower bound is slightly different from the one in [EVW02], since in the presence of data we don't need to use any unary predicates, while the lower bound in [EVW02] needs arbitrarily many unary predicates. In other words, we trade data values for predicates. This tradeoff is necessary: satisfiability of $\mathrm{FO}^2(<)$ is NP-complete if the number of predicates is fixed [WI09].

Theorem 17 *Satisfiability of* $\mathrm{FO}^2(\sim,<)$ *formulas over data words is* NExpTime-*complete. The lower bound holds even for formulas without unary predicates.*

We first show the lower bound:

Lemma 18 Satisfiability for $\mathrm{FO}^2(\sim,<)$ formulas without unary predicates is NExpTime-hard.

Proof. The reduction is from the *origin-constrained tiling problem* [Für84]. The input to this problem is given as (T, H, V, t), where T is a finite set of tiles ($|T| = k$), the sets $H, V \subseteq T \times T$ are horizontal and vertical matching relations, and $t \in T$ is a starting tile. The question is whether there exists a tiling of the square $2^n \times 2^n$ with the tile t in the upper left corner.

Given an instance of the tiling problem (T, H, V, t), we build an $\text{FO}^2(\sim, <)$ formula whose models are encodings of the solutions of the tiling problem. In particular, the formula has a model if and only if the tiling problem has a solution.

Let us describe the encoding of tilings by data words. Let $T = \{t_1, \ldots, t_k\}$ and π be a mapping from the square $2^n \times 2^n$ to T. We encode each position (i, j), $0 \leq i, j < 2^n$, of the square with a distinct data value. This will be done by considering the binary representations of i and j, respectively. An encoding of π is a data word factorized into $2n + k$ blocks $u_1 \cdots u_n v_1 \cdots v_n w_1 \cdots w_k$. The factorization comes from distinguishing the positions where the leftmost data value (the value of the first position in u_1) appears: in each of the blocks the first position, and no other, has the leftmost data value. In particular, we only consider data words where the leftmost data value appears exactly $2n + k$ times. Every data value, except the leftmost one, will be used to encode a position (i, j) in the square. By abuse of language we speak about the square position (and its horizontal/vertical coordinate) associated with a data value different from the leftmost one. The encoding is interpreted as follows:

- The blocks $u_1 \cdots u_n$ encode the horizontal coordinates of square positions: the ℓ-th bit of the horizontal coordinate i of a data value is 1 if and only if the value appears in the block u_ℓ. In particular, the data values associated with the first column ($i = 0$) do not appear in any of these blocks.

- Similarly, the blocks $v_1 \cdots v_n$ encode the vertical coordinates of square positions.

- The blocks $w_1 \cdots w_k$ encode the labels assigned by the mapping π. If $\pi(i, j) = t_l$, then the data value corresponding to (i, j) appears in the block w_m if and only if $l = m$.

Example: Consider $T = \{t_1, t_2, t_3\}$ and a tiling where t_1 labels positions $(0, 0)$ and $(1, 1)$, t_2 labels $(0, 1)$, and t_3 labels $(1, 0)$. One possible coding of this mapping is presented below. The data values $1, 2, 3, 4$ correspond to positions with coordinates $(0, 0)$, $(1, 0)$, $(0, 1)$ and $(1, 1)$. The leftmost data value 0 is used to divide the word into blocks.

$$\overbrace{0 \; 2 \; 4}^{u_1} \quad \overbrace{0 \; 3 \; 4}^{v_1} \quad \overbrace{0 \; 1 \; 4}^{w_1} \quad \overbrace{0 \; 3}^{w_2} \quad \overbrace{0 \; 2}^{w_3}$$

A mapping can have several codings, e.g. by swapping or duplicating non-0 positions inside a block.

We now describe an $\text{FO}^2(\sim, <)$ formula that tests if a data word encodes a solution of the tiling problem. The formula is the conjunction of the following properties:

1. The leftmost data value appears $2n + k$ times.

2. No two (non-leftmost) data values appear exactly in the same blocks $u_1 \cdots u_n$ and $v_1 \cdots v_n$. Every (non-leftmost) data value appears in exactly one block from $w_1 \cdots w_k$.

3. Each position of the square is associated with a data value. The constraints H and V, as well as the origin constraint t, are respected.

By induction on $p \leq 2n + k$, we define a formula $\psi_p(x)$ of polynomial size, which says that to the left of x, there are exactly p occurrences of the leftmost data value. Using this formula, we can express the first property.

We now proceed to express the two remaining properties. Let $\alpha(x)$ be the formula saying that x has a leftmost data value. The formula $\beta_p(x) = \exists y(y \sim x \wedge \psi_p(y))$ says that the data value of position x also occurs in the p-th block. The first clause in property 2 is written as follows:

$$\forall x \forall y \left((x \not\sim y \wedge \neg\alpha(x) \wedge \neg\alpha(y)) \rightarrow \bigvee_{1 \leq p \leq 2n} \beta_p(x) \not\leftrightarrow \beta_p(y) \right)$$

The second clause is written similarly. Property 3 is enforced by using standard binary arithmetics on coordinates. The following formula says that the positions of the squares corresponding to the data values of x and y are in consecutive rows. Here we take the convention that the least significant bit is the rightmost one. Thus there is some $1 \leq p \leq n$ such that the binary representation of these two consecutive rows is respectively $b_1 \cdots b_{p-1} 0 \underbrace{1 \cdots 1}_{n-p-1}$ and $b_1 \cdots b_{p-1} 1 \underbrace{0 \cdots 0}_{n-p-1}$.

$$\bigvee_{1 \leq p \leq n} \left(\neg\beta_p(x) \wedge \beta_p(y) \ \wedge \bigwedge_{1 \leq r < p} \beta_r(x) \leftrightarrow \beta_r(y) \ \wedge \bigwedge_{p < r \leq n} \beta_r(x) \wedge \neg\beta_r(y) \right)$$

A similar formula talks about columns. Using such formulas we can enforce the existence of all the positions of the square and the consistency of the labels in neighboring positions of the tiling. $\square$

We show now the upper bound:

Lemma 19 Satisfiability of $\mathrm{FO}^2(\sim, <)$ formulas is in NExpTime.

Proof. We give a polynomial-time reduction from satisfiability of $\mathrm{FO}^2(\sim, <)$ to satisfiability of $\mathrm{FO}^2(<)$, and then we apply [EVW02], which shows that satisfiability of $\mathrm{FO}^2(<)$ is in NExpTime.

As in the proof for $\mathrm{FO}^2(\sim, <, +1)$, we show that satisfiability of an $\mathrm{FO}^2(\sim, <)$ formula can be reduced to satisfiability of a formula in Scott normal form, a step that is performed in linear time. The Scott normal form formula is of the form

$$\forall x \forall y \ \chi(x, y) \wedge \bigwedge_i \forall x \exists y \ \chi_i(x, y) \ ,$$

where χ and each χ_i are quantifier-free and use only the order $<$, the data equivalence relation $\sim$ and unary predicates. Note that the new formula uses some new unary predicates, but its size—and therefore also the number of new predicates—is linear in the size of the original formula.

We now show that if a formula in Scott normal form is satisfiable, then it has a model with at most 2^{n+2} classes, where n is the number of unary predicates in the formula. Since we can use $n + 2$ new unary predicates for encoding these classes, we obtain a polynomial-time reduction to satisfiability of $\mathrm{FO}^2(<)$, and thus the NExpTime upper bound follows as satisfiability of $\mathrm{FO}^2(<)$ is in NExpTime [EVW02].

Given a model w of the formula in Scott normal form, we build a new model w' by removing all positions except those from some distinguished classes. Let α be a complete type, i.e. a truth assignment for *all* unary predicates. The new data word w' is built from w by keeping (if they exist), for each complete type α, the classes of positions $\mathrm{Fst}_\alpha(w), \mathrm{Ffst}_\alpha(w), \mathrm{Lst}_\alpha(w), \mathrm{Llst}_\alpha(w)$ (as defined in Section 2). All other classes are removed. Since there are 2^n possible complete types, the new data word w' contains at most $4 \cdot 2^n$ classes.

We now show that the data word w' is still a model of the formula. The $\forall x \forall y \ \chi(x, y)$ subformula holds in w', since w' is a substructure of w. For the $\forall x \exists y$ part, we show that the classes that we keep in w' suffice for satisfiability. In the data word w, every position x needs a witness y such that $\chi_i(x, y)$ holds. Consider a position x in w' and a corresponding witness y in w for a formula χ_i. If y is in the class of x, then y belongs to w', so it is also a witness of x in w'. Assume now that y is not in the class of x and that it has the complete type α. Since $\mathrm{Lst}_\alpha(w)$ and $\mathrm{Llst}_\alpha(w)$ are in different classes (and same for $\mathrm{Fst}_\alpha(w)$ and $\mathrm{Ffst}_\alpha(w)$), one of the positions $\mathrm{Lst}_\alpha(w), \mathrm{Llst}_\alpha(w)$, $\mathrm{Fst}_\alpha(w)$ or $\mathrm{Ffst}_\alpha(w)$ is also a witness of x in w. For instance, if $x < y$ then either x has the same value as $\mathrm{Lst}_\alpha(w)$ so we have the witness $\mathrm{Llst}_\alpha(w)$, or they have different values, so $\mathrm{Lst}_\alpha(w)$ is a witness. This shows that x has a witness in w', too. $\square$

8 Decidable extensions

8.1 More successors

It is often useful to talk about patterns in a string, which concern several consecutive positions and their mutual relationships. For instance, we might like to say that there are two positions x and y in the same class such that the substring between x and y is *abc*. To express such properties, it would be convenient to add to the signature relations like $y = x + 2$ and $y = x + 3$.

We denote by $\mathrm{FO}^2(\sim, <, +\omega)$ the logic extending $\mathrm{FO}^2(\sim, <, +1)$ with all predicates $+k$. Of course any given formula of this logic uses only a finite number of predicates $+k$, but the satisfiability algorithm must be prepared for arbitrarily large values. In this section, we show that this extension is still decidable, by adapting the proof for $\mathrm{FO}^2(\sim, <, +1)$.

Theorem 20 *Satisfiability of* $\mathrm{FO}^2(\sim, <, +\omega)$ *formulas over data words is decidable.*

Proof. The proof follows the same lines as the proof of Theorem 3, thus we mention here only the differences.

First, we make now use of predicates S_{+j}, with $1 \leq j \leq k$. Predicate S_{+j} holds at position x iff $x \sim (x + j)$. As for S_{+1}, we can construct a data automaton that guesses and verifies S_{+j}.

Let φ be a formula of $\mathrm{FO}^2(\sim, <, +\omega)$. This formula uses a finite number of successor relations $+1, \ldots, +k$. We transform it first into Scott normal form, as in the $\mathrm{FO}^2(\sim, <, +1)$ case. We need then to argue that Lemmas 12 and 13 still go through, using S_{+k} and the class marking technique.

Consider first formulas of the form (1). As in Lemma 12, the non-trivial case is $\delta(x, y) = (x \not\sim y)$. Here, the different cases for $\gamma(x)$ are more involved than those of (2) since there are more possibilities for the order constraints between x and y. However, the reader should convince herself that the overall proof idea is the same as in Lemma 12.

For formulas as in Lemma 13, the only difference is that the $\epsilon(x, y)$ subformulas are of one of the kinds $x = y \pm j$ or $x < y \pm j$, $0 \leq j \leq k$. Once again, the overall proof idea remains the same. For example, if $\delta(x, y) = (x \not\sim y)$ we use the 2 distinguished classes of Lst_β and Llst_β for $\epsilon(x, y) = (x + j < y)$, and the predicate S_{+j} for $\epsilon(x, y) = (x + j = y)$. $\qquad\square$

The proof of Theorem 20 implies that $\mathrm{FO}^2(\sim, <, +\omega)$ is included in $\mathrm{EMSO}^2(\sim, +1, \oplus 1)$, since we provide a reduction of $\mathrm{FO}^2(\sim, <, +\omega)$ to data automata. Recall that the logic $\mathrm{EMSO}^2(\sim, +1, \oplus 1)$ is the extension of $\mathrm{FO}^2(\sim, <, +1)$ by monadic second-order existential quantification and the class successor, and that this logic is equivalent to data automata (Prop. 14).

8.2 Infinite words

Another extension of interest, which might be useful e.g. for the verification of temporal properties, is the case of data ω-words. A *data ω-word* is simply an infinite sequence over $\Sigma \times D$. In this section we show the following result.

Theorem 21 *Satisfiability of* $\mathrm{FO}^2(\sim, <, +1)$ *formulas over data ω-words is decidable.*

The proof is along very similar lines as that of Theorem 3. We adapt data automata to infinite words, and we construct from an $\mathrm{FO}^2(\sim, <, +1)$ formula a data ω-automaton that recognizes the same language. We show that the string projection of the language defined by such an automaton is recognized by an extension of multicounter automata called *Büchi bag machine* (see below). Emptiness of this model is decidable by a reduction to emptiness of multicounter automata on finite words.

Data ω-automata can be defined in analogy to data automata. We only mention the differences here. A *data ω-automaton* $\mathcal{D} = (\mathcal{A}, \mathcal{B}_f, \mathcal{B}_i)$ consists of (1) a *base automaton* $\mathcal{A}$ which is a Büchi letter-to-letter transducer with output over some alphabet Γ, (2) a *finitary class automaton* $\mathcal{B}_f$ which is a finite word automaton over Γ and (3) an *infinitary class automaton* $\mathcal{B}_i$, which is a Büchi automaton over Γ. A data ω-word w is accepted if the base automaton has an accepting run

over the string projection of w with output $b_1 b_2 \cdots$ such that for every finite class $i_1 < \cdots < i_k$, the word $b_{i_1} \cdots b_{i_k}$ is accepted by $\mathcal{B}_f$; similarly, for every infinite class $i_1 < i_2 < \cdots$, the ω-word $b_{i_1} b_{i_2} \cdots$ is accepted by $\mathcal{B}_i$.

Theorem 21 follows immediately from Propositions 23, 24, and 26 stated below. In the proof of Proposition 23 we use the following result, the proof of which is an easy adaption of the proof of Proposition 11 to ω-words (with Büchi conditions for both kinds of automata).

Proposition 22 *For every register ω-automaton $\mathcal{A}$, there exists a data ω-automaton $\mathcal{B}$, computable in polynomial time from $\mathcal{A}$, which accepts the same language.*

Proposition 23 *Every data ω-language definable in $\mathrm{FO}^2(\sim,<,+1)$ is recognized by an effectively obtained data ω-automaton.*

Proof. The proof follows the lines of the proof of Proposition 4: We first transform the formula into Scott normal form and then transform each formula of the form $\forall x \forall y \xi$ into a data ω-automaton as in Lemma 12 and each formula of the form $\forall x \exists y \xi$ into a data ω-automaton as in Lemma 13. The Scott normalization step being identical, we will only explain the modifications needed in Lemmas 12 and 13. The basic difference when considering data ω-words arises in subcases involving the order relation between x and y, with x, y in different classes as the infiniteness of the word induces more cases than in the finitary case. All other cases are treated similarly in the finitary and infinitary case.

First, let ψ be a formula as in Lemma 12:

$$\forall x \forall y \left((\alpha(x) \ \wedge \ \beta(y) \ \wedge \ \delta(x,y)) \ \rightarrow \ \gamma(x,y) \right).$$

Recall that α and β are types, δ is either $x \sim y$ or $x \not\sim y$, and $\gamma(x,y)$ is a disjunction of atomic and negated atomic formulas that use only the order predicates $<$ and $+1$. The case when $\delta(x,y)$ is $x \sim y$ is solved like in the finitary case, using the predicate S_{+1} which marks all positions x such that the successor position has the same data value. The only difference is that we now deal with ω-words. By Proposition 22 this test can be done by a data ω-automaton. Using the predicate S_{+1} the formula ψ amounts to a regular condition on each (finite or infinite) class, which can be checked by suitable $\mathcal{B}_f$ and $\mathcal{B}_i$.

Thus, we assume now that $\delta(x,y)$ is $x \not\sim y$. We distinguish the same cases as in the proof of Lemma 12.

- The case $\gamma(x,y) = \mathit{false}$ is completely analogous to the finitary case.

- Let $\gamma(x,y)$ be $x < y$. Here we distinguish two cases: (a) either there are only finitely many α-positions, and this case is completely analogous to the finitary case. Or, (b) the ω-data word contains infinitely many α-positions. In the latter case, if $\mathrm{Fst}_\beta(w)$ is defined, then all α-positions after $\mathrm{Fst}_\beta(w)$ must be in the same class c as $\mathrm{Fst}_\beta(w)$. Moreover, there can be no β outside c and all α outside c must be before Fst_β.

  For (b) the base automaton guesses whether $\mathrm{Fst}_\beta(w)$ is defined and marks its class, if this is the case. It also checks that the marked class contains all (infinitely many) α after $\mathrm{Fst}_\beta(w)$, together with the remaining regular conditions above. The case $x \leq y+1$ is handled similarly.

- For the cases where $\gamma(x,y)$ is $x = y+1$, $x \neq y = 1$, or $x = y+1 \vee y = x+1$ we argue exactly the same as in Lemma 12.

It remains to consider formulas ψ as in Lemma 13:

$$\forall x \exists y \left(\alpha(x) \rightarrow \ (\beta(y) \ \wedge \ \delta(x,y) \ \wedge \ \epsilon(x,y)) \right)$$

The case when $\delta(x,y)$ is $x \sim y$ is treated as in Lemma 13, by checking a regular property on each class (using the predicate S_{+1} if needed). Therefore we consider only the case when $\delta(x,y)$ is $x \not\sim y$ (implying $x \neq y$):

- The case $\epsilon(x,y)$ is $x+1 = y$ or $x = y+1$ is solved as in Lemma 13: using S_{+1}, it is the base automaton can check ψ.

- If $\epsilon(x,y)$ is $x+1 < y$, then ψ expresses that each α-position needs a β-position in a different class to its right (but not as its right neighbor). The base automaton $\mathcal{A}$ nondeterministically guesses whether (a) there are only finitely many β, (b) there is one class with infinitely many β, or (c) there are at least two such classes. The verification that the guess is correct is done by the base automaton, using the class marking technique. Case (a) is handled as in the finitary case. For (b), $\mathcal{A}$ marks the class c with infinitely many β and checks that the last α in c is at least two positions before the last β outside c. For (c), there is nothing to do beyond checking that the guess was correct.

- Consider now the case where $\epsilon(x,y)$ is $y+1 < x$. If there is no occurrence of β in the data ω-word w, then ψ holds on w iff there is also no occurrence of α. If there is at least one occurrence of β ($\mathrm{Fst}_\beta(w)$ is defined) then we consider two cases. If $\mathrm{Ffst}_\beta(w)$ is not defined then ψ holds on w iff all α are in a different class than $\mathrm{Fst}_\beta(w)$ and there is no α before $\mathrm{Fst}_\beta(w)+2$. If $\mathrm{Ffst}_\beta(w)$ is defined then for each occurrence of α to the right of $\mathrm{Ffst}_\beta(w)+2$, either $\mathrm{Fst}_\beta(w)$ or $\mathrm{Ffst}_\beta(w)$ is an appropriate value for y in ψ. Hence ψ holds on w in this case iff there is no α before $\mathrm{Fst}_\beta(w)+2$ and all α between $\mathrm{Fst}_\beta(w)+2$ and $\mathrm{Ffst}_\beta(w)+2$ are in a different class than $\mathrm{Fst}_\beta(w)$.

  All these properties are easily testable by a data ω-automaton by guessing the appropriate case and marking the classes of $\mathrm{Fst}_\beta(w)$ and $\mathrm{Ffst}_\beta(w)$ as in the finitary case.

  $\square$

A *Büchi bag machine* is defined as follows. It is a finite automaton with a finite number of bags. Each bag contains (finitely many) tokens, taken from an infinite set. The automaton reads (one-way) ω-words without data values. A transition reads the current letter and, depending on the current state, performs a bag operation and changes the state, going to the right. Bag operations are of the following types:

- new(i): create a new token and place it in bag i.

- move(i,j): move some token from bag i to bag j.

In the move operation above, the token in bag i is chosen nondeterministically. In particular, transitions of the bag machine do not refer explicitly to token identities. For the acceptance condition, a set of accepting bags is distinguished. A run is accepting if each token created during a run is moved into an accepting bag infinitely often.

Before proceeding, we would like to sketch the analogy between Büchi bag machines and multicounter automata. Each counter of a multicounter automaton represents a bag. An increment of a counter can be simulated by creating a new token in the corresponding bag. A decrement is simulated by moving a token from the corresponding bag to a (special) sink bag. We could also define bag machines on finite words; in the analogue a run is accepting if all tokens are in the accepting bags at the end of the run. It is fairly easy to see that this version of bag automata for finite words is equivalent to multicounter automata.

Proposition 24 *From each data ω-automaton $\mathcal{D}$ a Büchi bag machine recognizing the string projection of the language recognized by $\mathcal{D}$ can be computed.*

Proof.
We simulate a data automaton $\mathcal{D} = (\mathcal{A}, \mathcal{B}_f, \mathcal{B}_i)$ over data ω-words with an effectively obtained Büchi bag machine $\mathcal{C}$. The simulating machine $\mathcal{C}$ runs over string projections, and uses tokens to simulate data values. Let S be the set of states of the base automaton and Q be the set of states of the class automata (we assume that $\mathcal{B}_f, \mathcal{B}_i$ have disjoint sets of states). The states of the bag machine $\mathcal{C}$ are the states S of the base automaton and $\mathcal{C}$ has a bag for each $q \in Q \cup S$ plus an

extra one, which is called *looping bag*. As in the proof of Proposition 5, the idea is to guess the output of the base automaton on-the-fly, while simulating the runs of the class automaton. The accepting bags are the looping bag, plus bags corresponding to accepting states of $\mathcal{A}$ and $\mathcal{B}_i$.

The transitions of $\mathcal{C}$ mimic the transitions of the base automaton $\mathcal{A}$, guessing its output for every position of the input word. Because the acceptance condition of $\mathcal{C}$ is defined by a set of accepting bags instead of accepting states, we use an extra token to duplicate the sequence of states of $\mathcal{A}$. This token is created at the first step of the run and moved from bag to bag among the bags corresponding to S, such that when $\mathcal{C}$ is in state s then the token is in the bag associated with s.

We now explain how the runs of the class automata are simulated. The idea is to use tokens to represent the classes of the original data word. More precisely, each class is associated with a different token that is created at the first position of the class. This token is moved from bag to bag among the bags corresponding to Q, in order to simulate the run of the class automaton on the corresponding class string. Thus, each transition of $\mathcal{C}$ creates or moves exactly one token among the Q bags. Intuitively this token simulates the class of the corresponding position in the input of the data automaton. The last step is to deal with the acceptance condition for runs corresponding to finite classes. At the end of such a finite accepting run, the token corresponding to the finite class is moved into the looping bag (where it will stay forever). This bag is accepting and refreshed infinitely often: in every transition, the machine non-deterministically moves a token from the looping bag back into it. This operation ensures that these tokens can go infinitely often into an accepting bag.

To sum up, a transition of $\mathcal{C}$ can be decomposed into three parts: one corresponding to $\mathcal{A}$ guesses an output symbol and moves the token contained in the S-bags; one corresponding to the runs of the class automata creates or moves exactly one token among the Q-bags; and one refreshing the content of the looping bag moves one token from the looping bag back into it.

From an accepting run of the Büchi bag machine one can reconstruct a data ω-word accepted by the data automaton $\mathcal{D}$ by looking at the evolution of tokens in bags from Q along the run.

$\square$

For the decidability proof for Büchi bag machines we use the following well-known result:

Fact 25 (Dickson's lemma [Dic13]) *For every infinite sequence of vectors $(x_n)_{n \in \mathbb{N}} \subseteq \mathbb{N}^k$ there exists an infinite subsequence $x_{i_1} \leq x_{i_2} \leq \cdots$, where $\leq$ is the component-wise ordering.*

Proposition 26 *Emptiness is decidable for Büchi bag machines.*

Proof. Fix a Büchi bag machine $\mathcal{C}$. We begin by showing that nonemptiness of $\mathcal{C}$ is equivalent to a property $(*)$ holding for some finite words u, v, with v non-empty. Then, we will show that the latter can be reduced to emptiness of a multicounter automaton, and is therefore decidable.

Property $(*)$ of words u, v is defined as follows. There is a run ρ over uv, with i being the position at the end of u and j being the position at the end of uv, such that

1. The states of the automaton as well as the sets of empty bags, are the same in positions i and j.

2. For each non-empty bag in position i there is some token which passes through an accepting bag between positions i and j.

3. Each non-empty bag at position i contains at least the same number of tokens at j as at i.

We first show that property $(*)$ is necessary for nonemptiness. Consider an infinite word w accepted by $\mathcal{C}$ and let ρ be the corresponding run. Since there is a finite number of bags and states, we can extract an infinite sequence of positions where the states and the sets of empty bags are the same. By Dickson's Lemma this sequence has an infinite subsequence S in which, for each bag, the number of tokens is non-decreasing. Let i be the first position in S. Since the run is

accepting, each token that exists in position i eventually passes through an accepting state. Since S is infinite, we can easily find an appropriate position $j \in S$ fulfilling (1)-(3).

We now show that property $(*)$ is sufficient for nonemptiness of $\mathcal{B}$. To this end, let u, v be words that satisfy property $(*)$. We claim that uv^ω is accepted by $\mathcal{C}$. The idea is to use a policy where the move operation between bags is applied to the token in the bag that has not seen an accepting bag for the longest time. Formally, we start with the run ρ_1 of $\mathcal{C}$ on uv. Assume now that we already defined, for $i \in \mathbb{N}$, a run ρ_i of $\mathcal{C}$ on uv^i, and we want to extend it to a run $\rho_{i+1} = \rho_i \rho'$ on uv^{i+1}. The subrun ρ' is obtained from the subrun ρ on v as follows. For each non-empty bag k at the end of ρ_i, let t_k be a token for which the number of steps without being in an accepting bag is maximal within the tokens of k. The subrun ρ' is obtained from ρ such that token t_k goes through an accepting bag.

It is not hard to see that in the resulting run on uv^ω each token passes through an accepting bag infinitely often. Assume otherwise that some token t passes only finitely often through an accepting bag. We can choose t such that the last step i in which t passes through an accepting bag is minimal (if t never does so let i be the step before t is created). Obviously, t will be the "minimal token" in its bag after a finite number of steps and thus will pass through an accepting bag, the desired contradiction.

Note that we have also shown here that every nonempty Büchi bag machine accepts an ultimately periodic word, if any.

It remains to show how it can be decided whether there exist finite words u, v with property $(*)$. We reduce this problem to the nonemptiness of a multicounter automaton $\mathcal{M}$ over finite words. For every pair of bags k, l in the automaton $\mathcal{C}$, the automaton $\mathcal{M}$ will have counters $b_{k,l}, c_{k,l}$ and d_l. At the beginning, $\mathcal{M}$ reads its input—corresponding to the word u—and simulates the run of $\mathcal{C}$, so that for every bag k of $\mathcal{C}$, the value of counter $c_{k,k}$ contains the size of bag k. (The other counters have value zero.) At some point, $\mathcal{M}$ nondeterministically guesses that it finished reading u. Let i refer to this position. It then reads the rest of its input, which corresponds to the word v. Between positions i and j (the end of the input uv), the counter values are interpreted as follows:

$b_{k,l}$: The number of tokens currently in bag l that were in bag k at position i and have passed through an accepting bag since position i.

$c_{k,l}$: The number of tokens currently in bag l that were in bag k at position i and have not passed through an accepting bag since position i.

d_l: The number of tokens currently in bag l that were created after position i.

Here and in the following, we do not distinguish notationally the value of a counter from the counter itself. Note that at any point between positions i and j we can recover the number a_k of tokens that were in bag k at position i, this number is

$$a_k = \sum_l (b_{k,l} + c_{k,l}) \ .$$

We can also know the number of tokens currently in bag l, which is

$$d_l + \sum_k (b_{k,l} + c_{k,l}) \ .$$

It is not difficult to define the transitions of $\mathcal{M}$ so that the intended properties of the counters are satisfied. For example, the transition of $\mathcal{M}$ that decrements $b_{k,l}$ and increments $b_{k,l'}$ simulates the move from the bag l to a bag l' of a token created before position i, which has passed through an accepting bag since position i and which was in bag k at position i. In order to capture property $(*)$, the automaton should accept if: the states of $\mathcal{C}$ were the same in positions i and j, which is easy for the automaton to check; and for each bag k, both properties (3) and (4) below are satisfied:

$$a_k \quad \leq \quad d_k + \sum_l (b_{l,k} + c_{l,k}) \tag{3}$$

$$a_k = d_k = 0 \qquad \text{or} \qquad \sum_l b_{k,l} > 0 \tag{4}$$

We have produced a multicounter automaton where the acceptance condition is a boolean combination of linear inequalities on its counters. As we will explain next, emptiness for this extended model of multicounter automata can be reduced to emptiness for the usual model (introduced above) where the acceptance condition requires all counters to have value zero at the end of the run.

First, we can convert the boolean combination to DNF, so that we may assume that the property to check at the end of the run is a conjunction of linear inequalities. By using copies of counters we may assume that each counter occurring in the acceptance condition appears in a unique inequality. The last step is to simulate each inequality by zero tests. Each inequality $\sum_{c \in C} c \leq \sum_{d \in D} d$ can be checked by decrementing in parallel (and non-deterministically) one counter from C and one from D. Then we stop decrementing C and continue with D only, until all counters are zero. $\qquad \square$

9 Undecidable extensions

In this section we show that many immediate extensions of the logic lead to undecidability. We first consider the case of several equivalence relations. In XML, document nodes may have several different attributes, which are accessed via the query languages. Equality tests between node attributes could be simulated using several equivalence relations. For instance checking that the nodes x and y agree on attribute a could be written as $x \sim_a y$. However, in [KO05] it is shown that two-variable logic is undecidable with *three* equivalence relations and some unary relations. In the presence of the successor and the linear order, two equivalence relations already yield undecidability:

Proposition 27 *Satisfiability of* $\mathrm{FO}^2(\sim_1, \sim_2, <, +1)$ *and of* $\mathrm{FO}^2(\sim_1, \sim_2, +1, +2, +3)$ *formulas over data words is undecidable.*

Proof. In the following, *2-data-words* are finite sequences over $\Sigma \times D \times D$, i.e., each position carries two data values. Below, we write i-class when talking about the equivalence class of the relation $\sim_i$, for $i = 1, 2$.

The proof is by a reduction from Post's Correspondence Problem (PCP). An instance of PCP consists of k pairs (u_i, v_i) of words from Σ^* and the question is whether there exists a non-empty, finite sequence of indices $i_0, \ldots, i_n$ such that $u_{i_0} u_{i_1} \cdots u_{i_n} = v_{i_0} v_{i_1} \cdots v_{i_n}$.

Let $\Sigma' = \Sigma \cup \overline{\Sigma}$ be the alphabet consisting of two disjoint copies of Σ. If w is a word in Σ^*, then $\overline{w} \in \overline{\Sigma}^*$ is obtained from w by replacing each letter with the corresponding one in $\overline{\Sigma}$. Consider a solution $i_0, \ldots, i_n$ for the given PCP instance. Let w be the word $u_{i_0} u_{i_1} \cdots u_{i_n}$, or equivalently the word $v_{i_0} v_{i_1} \cdots v_{i_n}$. Without loss of generality we assume that if there is a solution, then there is one where w is of odd length. To this end, for each pair (u_i, v_i) an additional pair $(\$u_i, \$v_i)$ can be added, where \$ is a new symbol. Thus, for each solution w without \$, the word \$$w$ is a solution as well (and if the original system had no solutions, the new system will not have any solution either). We encode this solution by a data word $\hat{w}$ satisfying the following:

- The string projection $\mathrm{str}(\hat{w})$ is $u_{i_0} \overline{v_{i_0}} \cdots u_{i_n} \overline{v_{i_n}}$. In particular, the sequence of letters from Σ is the same as the sequence of letters from $\overline{\Sigma}$.

- The subsequence of pairs of values in w is the same as the one in $\overline{w}$, and it is of the form $(\alpha_1, \beta_1)(\alpha_1, \beta_2)(\alpha_2, \beta_2)(\alpha_2, \beta_3) \ldots (\alpha_{m-1}, \beta_m)(\alpha_m, \beta_m)$ where all α_i (β_j, respectively) are different.

We describe first a formula of $\mathrm{FO}^2(\sim_1, \sim_2, <, +1)$ such that w is a solution of PCP iff $\hat{w}$ is a model of the formula. The formula is the conjunction of the following properties:

(1) The string projection belongs to $\{u_i \overline{v_i} \mid 1 \le i \le k\}^+$. This is easy to do in $\mathrm{FO}^2(+1)$.

(2) The values in the Σ-subword are such that:

  (a) Each 1-class has 2 elements with a label in Σ, except for the last Σ-position, which has a 1-value that occurs only once.

  (b) Each 2-class has 2 elements with a label in Σ, except for the first Σ-position, which has a 2-value that occurs only once.

  (c) For every Σ-position but the first and the last one: either there exist a Σ-position to its right (relative to $<$) in the same 1-class and a Σ-position to its left (relative to $<$) in the same 2-class, or the same with 1 and 2 interchanged.

(3) The same properties as in (2) are required for the $\overline{\Sigma}$-subword.

(4) Finally, each pair of data values of the Σ-subword appears also in the $\overline{\Sigma}$-subword with the same letter, and conversely.

It is not hard to verify now that property (2) ensures that the sequence of pairs of data in the Σ-subword is of the form

$$(\alpha_1, \beta_1)(\alpha_1, \beta_2)(\alpha_2, \beta_2)(\alpha_2, \beta_3)\dots(\alpha_{m-1}, \beta_m)(\alpha_m, \beta_m)$$

To see this, we can construct a directed graph where the vertices are the Σ-positions and there is an edge from x to y iff $x < y$ and they are in the same 1-class or 2-class. Because of formula (2), every node but the first and the last one has in-degree 1 and out-degree 1 (the first node has indegree 0 and outdegree 1, and the last one has indegree 1 and outdegree 0). Therefore, the graph must corresponds to the natural successor relation on the Σ-positions.

Since we have the same property in the $\overline{\Sigma}$-subword, and together with formula (4), the sequence of data values and letters is the same in the Σ-subword and the $\overline{\Sigma}$-subword, hence we obtain a solution of PCP.

If the order relation $<$ is replaced by relations for the $+2, +3$ successors, the above undecidability proof can be easily adapted. To simplify matters we assume that the PCP instance consists of words of length at most 2. It is well-known that PCP is still undecidable in this case [HU79]. We use the same encoding. The only difference is the way to ensure that the sequence of pairs of data in the Σ-subword (respectively $\overline{\Sigma}$-subword) is of the form

$$(\alpha_1, \beta_1)(\alpha_1, \beta_2)(\alpha_2, \beta_2)(\alpha_2, \beta_3)\dots(\alpha_{m-1}, \beta_m)(\alpha_m, \beta_m)$$

We can now enforce directly that the sequence of 1-values in the Σ-subword is of the form $\alpha_1 \alpha_1 \alpha_2 \alpha_2 \dots \alpha_m$ (and similarly for the 2-values), with distinct α_i. For this, it suffices to replace the last requirement of property (2) by the following one: each 1-class with 2 elements is such that these positions are either adjacent, or separated by $\overline{\Sigma}$-letters only. Since at most 2 such letters can occur contiguously, the claim follows. $\qquad\square$

With only one equivalence relation, three variables already yield undecidability, even without linear order.

Proposition 28 *Satisfiability of* $\mathrm{FO}^3(\sim, +1)$ *formulas over data words is undecidable.*

Note that this implies the undecidability of satisfiability for $\mathrm{FO}^3(\sim, <)$, and of course of $\mathrm{FO}^3(\sim, +1, <)$, since the relation $+1$ is definable from $<$ with three variables.

Proof. One way of proving the proposition would be by a reduction from the undecidable logic $\mathrm{FO}^2(\sim_1, \sim_2, +1, +2, +3)$ (recall Proposition 27). To implement two data values, each position is split into two consecutive positions, with the second position having a special dummy label. Using three variables, we can then simulate any relation $+k$.

Below we present a more direct proof, which will also be referenced in Proposition 29.

We reduce PCP to satisfiability of $\text{FO}^3(\sim, +1)$. We slightly modify the coding of Proposition 27 to deal with the fact that each position of the word carries only one data value.

The idea is that each class will contain two positions, these will be used to match the appropriate positions in the words w and $\overline{w}$. More formally, a solution will be encoded by a data word $\hat{w}$ with labels from Σ' satisfying the following:

- The string projection of $\hat{w}$ belongs to $\{u_i \overline{v_i} \mid 1 \leq i \leq k\}^+$. This can be easily expressed by a formula of $\text{FO}^2(+1)$.

- The Σ-positions ($\overline{\Sigma}$-positions, respectively) carry different data values. Moreover, the sequence of data values in the Σ-subword is the same as the one in the $\overline{\Sigma}$-subword. The former can be expressed by using only two variables; it is the latter condition for which we need a third variable. We express that for every pair of data values in consecutive Σ-positions, their matching occurrences with labels in $\overline{\Sigma}$ are also consecutive:

$$\forall x \forall y \; Succ(y,x) \; \rightarrow (\exists z \; x \sim z \wedge (\exists x \; \overline{Succ}(x,z) \wedge x \sim y))$$

Here, $Succ(y,x)$ ($\overline{Succ(y,x)}$, respectively) expresses that x, y are consecutive Σ-positions ($\overline{\Sigma}$-positions, respectively). This can be done without the order or additional successor predicates, because the distance between x and y is bounded by the maximal length of a PCP word and because we can use three variables.

$\square$

Another possible extension is to add a binary predicate to the signature which is interpreted as a linear order on the data values. However, it is not hard to see that this also yields undecidability, even for FO^2.

Proposition 29 *Satisfiability of* $\text{FO}^2(\sim, \prec, +1, <)$ *formulas over data words is undecidable.*

Proof. We revisit the proof of Proposition 28. We use the same coding of PCP and recall that only the last formula, which checks that the two sequences of data values (the one for symbols in Σ and the one for symbols in $\overline{\Sigma}$) are the same, uses three variables. With the help of $\prec$ this can be replaced by a formula which checks that, for both sequences, the data values are in increasing order: $\forall x, y \; \Sigma(x) \wedge \Sigma(y) \wedge x < y \rightarrow x \prec y$ (and similarly for $\overline{\Sigma}$). $\square$

10 Discussion

We have shown that satisfiability of $\text{FO}^2(\sim, <, +1)$ over data words is decidable. It follows immediately that also $\text{EMSO}^2(\sim, <, +1)$ is decidable over such models.

In the absence of data values, $\text{FO}^2(+1, <)$ has several equivalent characterizations, for instance it corresponds to the fragment of LTL that uses only unary temporal predicates [EVW02]. Still in the absence of data values, $\text{EMSO}^2(+1, <)$ has the same expressive power as MSO. In a sense the decidability of $\text{EMSO}^2(\sim, <, +1)$ can be seen as an extension of classical decidability result of MSO over strings.

An interesting side result is the connection between $\text{FO}^2(\sim, <, +1)$ and multicounter automata (and therefore Petri nets). Indeed, string projections of the languages defined by $\text{FO}^2(\sim, <, +1)$ formulas are recognized by multicounter automata. The converse is true when the logic is extended to $\text{EMSO}^2(\sim, <, +1, \oplus 1)$, which has the same expressive power as $\text{EMSO}^2(\sim, +1, \oplus 1)$. It would be interesting to understand better the connection between the two formalisms. Because of the connection with Petri nets pinpointing the complexity of satisfiability is likely to be difficult.

Our reduction from the decidability of $\text{FO}^2(\sim, <, +1)$ to emptiness of multicounter automata is in 2ExpTime. We do not know whether this is optimal or not.

If only one of the two predicates $+1$ and $<$ is used, the decision problem is elementary. As shown in this paper, satisfiability is NExpTime-complete for $\text{FO}^2(\sim, <)$. In [BDM+06] we studied

the logic $FO^2(\sim, +1)$, and gave a 3NExpTime algorithm for satisfiability over unranked ordered trees. This immediately gives a 3NExpTime upper bound for satisfiability of $FO^2(\sim, +1)$ over words, although we conjecture that the algorithm can be improved to 2NExpTime, or even further, if only words are considered. The logic $FO^2(\sim, +1)$ was also considered in [BB07b], which considered words with several data values. It turns out that if the data values are such that the appropriate equivalence relations are successive refinements, then satisfiability becomes decidable for $FO^2(\sim, +1)$.

Whether $FO^2(\sim, <, +1)$ is decidable over trees is still an open question which was shown in [BDM$^+$06] to be at least as hard as checking emptiness of multicounter automata over trees (stated as an open question in [GGS04]). One decidable variant of satisfiability for $FO^2(\sim, <, +1)$ for trees is the bounded depth case, considered in [BB07a]: if only trees of fixed depth k are considered, then satisfiability is decidable, regardless of the choice of k.

References

[BB07a]   Henrik Björklund and Mikołaj Bojańczyk. Bounded Depth Data Trees. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming, ICALP'07*, volume 4596 of *Lecture Notes in Computer Science*, pages 862–874. Springer-Verlag, 2007.

[BB07b]   Henrik Björklund and Mikołaj Bojańczyk. Shuffle Expressions and Words with Nested Data. In *Proceedings of the 32nd Mathematical Foundations of Computer Science International Symposium, MFCS'07*, volume 4708 of *Lecture Notes in Computer Science*, pages 750–761. Springer-Verlag, 2007.

[BDM$^+$06] Mikołaj Bojańczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data trees and XML reasoning. In *Proceedings of the 25th SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS'06*, pages 10–19. ACM Press, 2006.

[BMS$^+$06] Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-Variable Logic on Words with Data. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science, LICS'06*, pages 7–16. IEEE Computer Society, 2006.

[BMSS09]   Mikołaj Bojańczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-Variable Logic on Data Tree and XML Reasoning. *Journal of the ACM*, 56(3):1–48, 2009.

[BPT03]   Patricia Bouyer, Antoine Petit, and Denis Thérien. An Algebraic Approach to Data Languages and Timed Languages. *Information and Computation*, 182(2):137–162, 2003.

[BS07]   Henrik Björklund and Thomas Schwentick. On Notions of Regularity for Data Languages. In *Proceedings of the 16th International Symposium on Fundamentals of Computation Theory, FCT'07*, volume 4639 of *Lecture Notes in Computer Science*, pages 88–99. Springer-Verlag, 2007.

[Dic13]   Leonard Eugene Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *American Journal of Mathematics*, 35:413–422, 1913.

[DL09]   Stphane Demri and Ranko Lazić. LTL with the Freeze Quantifier and Register Automata. *ACM Transactions on Computational Logic*, 10(3):1–30, 2009.

[DLN07] Stphane Demri, Ranko Lazić, and David Nowak. On the freeze quantifier in constraint LTL: Decidability and complexity. *Information and Computation*, 205(1):2–24, 2007.

[EN94] Javier Esparza and Mogens Nielsen. Decidability Issues for Petri Nets - a survey. *Elektronische Informationsverarbeitung und Kybernetik*, 30(3):143–160, 1994.

[EVW02] Kousha Etessami, Moshe Y. Vardi, and Thomas Wilke. First-Order Logic with Two Variables and Unary Temporal Logic. *Information and Computation*, 179(2):279–295, 2002.

[Für84] Martin Fürer. The computational complexity of the unconstrained limited domino problem (with implications for logical decision problems). In *Logic and Machines: Decision Problems and Complexity*, volume 171 of *Lecture Notes in Computer Science*, pages 312–319. Springer-Verlag, 1984.

[GGS04] Philippe de Groote, Bruno Guillaume, and Sylvain Salvati. Vector Addition Tree Automata. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, LICS'04*, pages 64–73. IEEE Computer Society, 2004.

[Gis81] Jay L. Gischer. Shuffle Languages, Petri Nets, and Context-Sensitive Grammars. *Communications of the ACM*, 24(9):597–605, 1981.

[GO99] Erich Grädel and Martin Otto. On Logics with Two Variables. *Theoretical Computer Science*, 224(1-2):73–113, 1999.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[KF94] Michael Kaminski and Nissim Francez. Finite-Memory Automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[KO05] Emanuel Kieroński and Martin Otto. Small Substructures and Decidability Issues for First-Order Logic with Two Variables. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science, LICS'05*, pages 448–457. IEEE Computer Society, 2005.

[Kos82] S. Rao Kosaraju. Decidability of Reachability in Vector Addition Systems. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, STOC'82*, pages 267–281. ACM Press, 1982.

[Lip76] R.J. Lipton. The reachability problem requires exponential space. Technical Report 62, Yale University, 1976.

[May84] Ernst W. Mayr. An Algorithm for the General Petri Net Reachability Problem. *SIAM J. Comput.*, 13(3):441–460, 1984.

[Mor75] M. Mortimer. On languages with two variables. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 21:135–140, 1975.

[NSV04] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic*, 15(3):403–435, 2004.

[Ott01] Martin Otto. Two Variable First-Order Logic over Ordered Domains. *Journal of Symbolic Logic*, 66(2):685–702, 2001.

[SF94] Yael Shemesh and Nissim Francez. Finite-State Unification Automata and Relational Languages. *Information and Computation*, 114(2):192–213, 1994.

[STV01]   Thomas Schwentick, Denis Thérien, and Heribert Vollmer. Partially-Ordered Two-Way Automata: A New Characterization of DA. In *Proceedings of the 5th International Conference on Developments in Language Theory, DLT'01*, volume 2295 of *Lecture Notes in Computer Science*, pages 239–250. Springer-Verlag, 2001.

[TW98]   Denis Thérien and Thomas Wilke. Over Words, Two Variables Are as Powerful as One Quantifier Alternation. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing, STOC'98*, pages 234–240. ACM Press, 1998.

[WI09]   Philipp Weis and Neil Immerman. Structure Theorem and Strict Alternation Hierarchy for FO^2 on Words. *Logical Methods in Computer Science*, 5(3), 2009.