

CLASS COUNTING AUTOMATA ON DATAWORDS

AMALDEV MANUEL*

Institute of Mathematical Sciences, Taramani, Chennai, India, 600113
amal@imsc.res.in
<http://www.imsc.res.in/~amal>

R. RAMANUJAM

Institute of Mathematical Sciences, Taramani, Chennai, India, 600113
jam@imsc.res.in
<http://www.imsc.res.in/~jam>

Received 31 January 2010

Accepted 19 August 2010

Communicated by Oliver Bournez and Igor Potapov

In the theory of automata over infinite alphabets, a central difficulty is that of finding a suitable compromise between expressiveness and algorithmic complexity. We propose an automaton model where we count the multiplicity of data values on an input word. This is particularly useful when such languages represent behaviour of systems with unboundedly many processes, where system states carry such counts as summaries. A typical recognizable language is: “every process does at most k actions labelled a ”. We show that emptiness is elementarily decidable, by reduction to the covering problem on Petri nets.

Keywords: Automata on infinite alphabet; dataword; decidability.

1991 Mathematics Subject Classification: 22E46, 53C35, 57S20

1. Introduction

Many widely used systems of today, most importantly web servers and other distributed systems, are machines which concurrently run many sequential processes. When there is no a priori bound on the number of processes, though at any point of time only finitely many are active, the necessity of the system to distinguish one process from another involves potentially unbounded data. Typically, system states carry summary information about processes that are known to be active, and hence the set of system configurations is infinite. Such systems arise in the study of web services, communication protocols and software systems with recursive concurrent threads of execution.

*Corresponding author.

Infinite state systems are not unfamiliar in theory of computation; a rich body of results exists on counter systems, pushdown systems and Petri nets. Most reachability properties of such infinite state systems are either undecidable or have such high complexity that algorithmic verification is impractical. On the other hand, if we restrict ourselves to only finite state systems, we can reason only about systems where the set of processes is fixed and known a priori, and we do not (as yet) have clear abstractions that allow us to transfer the results of such reasoning to systems of unbounded processes. Hence there is a clear need for formal models that work with unbounded systems but yet restrict expressiveness to allow decidable verification.

Notice that interesting properties of such systems do not involve process names (or identifiers) explicitly. A specification that restricts attention only to processes P_1, P_2 and P_3 can be implemented by a finite state system. On the other hand, consider a specification such as: “at least k processes get to perform an a action”: this necessitates remembering potentially unboundedly many values, thus leading to infinite state systems.

This paper is situated in such a context and while we have no definitive answers, we consider “state summaries” that allow elementary decidability. The model we use is that of finite automata over infinite alphabets and we use counters to record the intended “summaries”. The main result is that emptiness is EXPSPACE-complete for such a class of automata. Unfortunately, the automata are not closed under complementation, and even the word problem is intractable, suggesting that we have more work to do to further restrict expressiveness.

The study of automaton mechanisms over infinite alphabets has gained interest in recent years, especially from the viewpoint of database theory. In this approach, data values are modelled using a countably infinite domain, and structures are finite words labelled by this infinite alphabet. Typically the alphabet is presented as a product $(\Sigma \times \mathbb{D})$, where Σ is finite and \mathbb{D} is countable. For our purposes, we can think of \mathbb{D} as process names and Σ as the finite set of events they participate in, or conditions that hold.

The study of languages over infinite alphabets was initiated in [2] and [19], where the approach was to define the notion of regularity for languages over infinite alphabets in terms of morphisms to languages over finite alphabets. There are many automaton mechanisms for studying word languages over infinite alphabets: register automata [9], pebble automata [17, 18], data automata [5], nested words [1], class memory automata [4] and automata on Gauss words [16], with different expressive power and complexity. Logic based approaches include monadic second order logics [6, 3], two variable first order logics [5] and temporal logics with special “freeze” quantifiers [8] or predicate abstraction [14, 15]. Algebraic approaches involve quasi-regular expressions [10], or register monoid mechanisms [7]. All these involve interesting trade-offs between expressiveness and complexity of decision procedures. A unifying framework placing all these models in perspective is as yet awaited (see [20] for an excellent survey).

While register automata have polynomial complexity, they are effectively finite state; data automata are more expressive, but emptiness is not known to be elementarily decidable. What we present here is a restriction of class memory automata: these automata can not only test for existence of data values, but can also count the multiplicity of occurrences of data values, subject to constraints on such counts. However, these counters are *monotone*, and hence the constraints are limited in expressive power: we can compare counts against constants, but not much more. We show that such a model of **Class counting automata (CCA)** is interesting, for several reasons; specifically, we get elementary decidability. We see this as “populating the landscape” of classes of data languages, in the sense of [4].

From the viewpoint of reasoning about unbounded systems of processes, it is unclear what exactly is the expressiveness needed. For instance, consider the specification: “No two successive positions carry the same data value”; this is naturally implemented using a register mechanism. But this is a “hard” global scheduling constraint: after any process event is scheduled, the succeeding event must necessarily be from a different process; it is hardly clear that such a constraint is important for loosely coupled systems of processes. This indicates that while we do want to specify combinations of global and local properties, we need to nonetheless allow for sufficient flexibility.

2. Class Counting Automata

Below, for $k > 0$, we denote by $[k]$ the set $\{1, 2, \dots, k\}$. When we say $[k]_0$, we mean the set $\{0\} \cup [k]$. By \mathbb{N} , we mean the set of natural numbers $\{0, 1, \dots\}$. When $f : A \rightarrow B$, $(a, b) \in (A \times B)$, by $f \oplus (a, b)$, we mean the function $f' : A \rightarrow B$, where $f'(a') = f(a')$ for all $a' \in A$, $a' \neq a$, and $f'(a) = b$.

Customarily, the infinite alphabet is split into two parts: it is of the form $\Sigma \times \mathbb{D}$, where Σ is a finite set, and \mathbb{D} is a countably infinite set. Usually, Σ is called the *letter alphabet* and \mathbb{D} is called the *data alphabet*. Elements of \mathbb{D} are referred to as *data values*. We use letters a, b etc to denote elements of Σ and use d, d' to denote elements of \mathbb{D} .

A **data word** w is an element of $(\Sigma \times \mathbb{D})^*$. A collection of data words $L \subseteq (\Sigma \times \mathbb{D})^*$ is called a *data language*. In this article, by default, we refer to data words simply as words and data languages as languages. As usual, by $|w|$ we denote the length of w .

Let $w = (a_1, d_1)(a_2, d_2) \dots (a_n, d_n)$ be a data word. The *string projection* of w , denoted as $str(w) = a_1 a_2 \dots a_n$, the projection of w to its Σ components. Let $i \in [n]$. The **data class** of d_i in w is the set $\{j \in [n] \mid d_i = d_j\}$. A subset of $[n]$ is called a data class of w if it is the data class of some d_i , $i \in [n]$. Note that the set of data classes of w form a partition of $[n]$.

A **constraint** is a pair $c = (\text{op}, e)$, where $\text{op} \in \{<, =, \neq, >\}$ and $e \in \mathbb{N}$. When $v \in \mathbb{N}$, we say $v \models c$ if $v \text{ op } e$ holds. Let \mathcal{C} denote the set of all constraints. Define a *bag* to be a finite set $h \subseteq (\mathbb{D} \times \mathbb{N})$ such that whenever $(d, n_1) \in h$ and $(d, n_2) \in h$,

we have: $n_1 = n_2$. Thus h defines a partial function from \mathbb{D} to \mathbb{N} which is defined on a finite subset of \mathbb{D} . By convention, we implicitly extend it to a total function on \mathbb{D} by considering h to represent the set $h' = h \cup \{(d, 0) \mid \text{there is no } n \in \mathbb{N} \text{ such that } (d, n) \in h\}$. Hence we (ab)use the notation $h(d) = n$ for a bag h . Let \mathbb{B} denote the set of bags. Note that the notation $h \oplus (d, n)$ now stands for the bag $h' = (h - (\{d\} \times \mathbb{N})) \cup \{(d, n)\}$.

The automaton we present below includes a bag of infinitely many monotone counters, one for each possible data value. When it encounters a letter - data pair, say (a, d) , the multiplicity of d is checked against a given constraint, and accordingly updated, the transition causing a change of state, as well as possible updates for other data as well. We can think of the bag as a hash table, with elements of \mathbb{D} as keys, and counters as hash values. Transitions depend only on hash values (subject to constraints) and not keys.

Below, let $Inst = \{\uparrow^+, \downarrow\}$ stand for the set of *instructions*. Each instruction takes a natural number as an argument. The \uparrow^+ instruction with argument k tells the automaton to increment the counter by k , whereas \downarrow with argument k asks for a reset to the value k . Note that the instruction $(\uparrow^+, 0)$ says that we do not wish to make any update, and $(\uparrow^+, 1)$ causes a unit increment; we use the notation $[0]$ and $[+1]$ for these instructions below.

Definition 1. A *class counting automaton*, abbreviated as CCA, is a tuple $CCA = (Q, \Delta, I, F)$, where Q is a finite set of states, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states. The transition relation is given by: $\Delta \subseteq (Q \times \Sigma \times C \times Inst \times U \times Q)$, where C is a finite subset of \mathbb{C} and U is a finite subset of \mathbb{N} .

Representation of constants : We note here that the constants in the definition of the automata are represented in unary. The mode of representation of numbers turns out to be crucial for the upper bound of the emptiness problem.

Let A be a CCA. A **configuration** of A is a pair (q, h) , where $q \in Q$ and $h \in \mathbb{B}$. The initial configuration of A is given by (q_0, h_0) , where h_0 is the empty bag; that is, $\forall d \in \mathbb{D}, h_0(d) = 0$ and $q_0 \in I$.

Given a data word $w = (a_1, d_1), \dots, (a_n, d_n)$, a **run** of A on w is a sequence $\gamma = (q_0, h_0)(q_1, h_1) \dots (q_n, h_n)$ such that $q_0 \in I$ and for all $i, 0 \leq i < n$, there exists a transition $t_i = (q, a, c, \pi, m, q') \in \Delta$ such that $q = q_i, q' = q_{i+1}, a = a_{i+1}$ and:

- $h_i(d_{i+1}) \models c$.
- h_{i+1} is given by:

$$h_{i+1} = \begin{cases} h_i \oplus (d_{i+1}, m') & \text{if } \pi = \uparrow^+, m' = h_i(d_{i+1}) + m \\ h_i \oplus (d_{i+1}, m) & \text{if } \pi = \downarrow \end{cases}$$

γ is an **accepting run** above if $q_n \in F$. The language accepted by A is given by $L(A) = \{w \in (\Sigma \times \mathbb{D})^* \mid A \text{ has an accepting run on } w\}$. $L \subseteq (\Sigma \times \mathbb{D})^*$ is said to be recognizable if there exists a CCA A such that $L = L(A)$. Note that the counters are either incremented or reset to fixed values.

We first observe that CCA runs have some useful properties. To see this, consider a bag h and $d_1, d_2 \in \mathbb{D}$, $d_1 \neq d_2$ such that at a configuration (q, h) , we have two transitions enabled on inputs (a_1, d_1) and (a_2, d_2) leading to configurations (q_1, h_1) and (q_2, h_2) respectively. Notice that for any condition c , if $h(d_2) \models c$ then so also $h_1(d_2) \models c$. Similarly, for any condition c' , if $h(d_1) \models c'$ then so also $h_2(d_1) \models c'$. Thus when we have distinct data values, tests on them do not “interfere” with each other. We can extend this observation further: given data words u and v such that the data values in u are pairwise disjoint from those in v , if we have a run from (q, h) on u to (q, h_1) and on v from (q, h_1) to (q', h_2) , then there is a configuration (q', h') and a run from (q, h) on v to (q', h') . This will be useful in the following.

Example 2. *The language $L_1 =$ “Data values under a are all distinct” is accepted by a CCA. The CCA accepting this language is the automaton $A = (Q, \Delta, q_0, F)$ where $Q = \{q_0, q_1\}$, q_0 is the only initial state and $F = \{q_0\}$. Δ consists of:*

- $(q_0, a, (=, 0), q_0, [+1]); (q_0, a, (=, 1), q_1, [0]);$
- $(q_0, b, (\geq, 0), q_0, [0]); (q_1, \Sigma, (\geq, 0), q_1, [0]).$

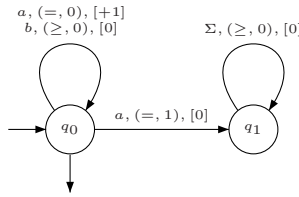


Fig. 1. Automaton in the Example 2.

Since the automaton above is deterministic, by complementing it, that is, setting $F = \{q_1\}$, we can accept the language $\overline{L_1} =$ “There exists a data value appearing at least twice under a ”.

Example 3. *Fix Σ to be $\{a\}$. Let the language L_2 be “There exists a data value whose multiplicity is not two.” is accepted by a CCA. The CCA accepting this language is the automaton $A = (Q, \Delta, q_0, F)$ where $Q = \{q_0, q_1, q_2, q_3\}$, q_0 is the only initial state and $F = \{q_1, q_3\}$. Δ consists of:*

- $(q_0, a, (=, 0), q_1, [+1]); (q_0, a, (=, 0), q_0, [0]);$
- $(q_1, a, (=, 1), q_2, [+1]); (q_1, a, (=, 0), q_1, [0]);$
- $(q_2, a, (=, 2), q_3, [+1]); (q_2, a, (=, 0), q_2, [0]);$
- $(q_3, a, (\geq, 3), q_3, [+1]); (q_3, a, (=, 0), q_3, [0]).$

The idea is that the automaton chooses non-deterministically a data value and faithfully counts its multiplicity, while keeping the counters of other data values zero. Finally the automaton accepts the word, if the current count is not two.

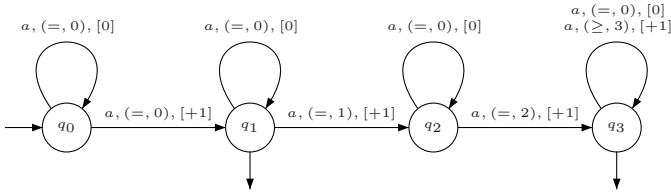


Fig. 2. Automaton in the Example 3.

But as we show below, its complement language, $\overline{L_2} = \text{“All data values occur exactly twice”}$ is not recognizable. Thus, CCA- recognizable data languages are not closed under complementation.

Proposition 4. *The language $\overline{L_2} = \text{“All data values occur exactly twice”}$ is not recognizable.*

Proof. Suppose there is a CCA A with m states accepting this language. Consider the data word

$$w = (a, d_1)(a, d_2)..(a, d_{m+1})(a, d_1)(a, d_2)..(a, d_{m+1})$$

Clearly, $w \in \overline{L_2}$. Therefore, there is a successful run of A on w . Then there is a state q repeating in the suffix of length $m + 1$. Let us say this splits w as $u \cdot v \cdot v'$, such that the configuration after u is (q, h) and after v it is (q, h_1) . Then by the remarks we made earlier, we can find an accepting run for $u \cdot v'$ as well. But then $u \cdot v'$ is not in $\overline{L_2}$. □

Proposition 5. *CCA-recognizable data languages are closed under union and intersection but not under complementation.*

Proof. Closure under union and intersection is easily obtained by product construction. □

The following observation will be useful for decision questions that follow. Given a CCA $A = (Q, \Delta, q_0, F)$ let m be the maximum constant used in Δ . We define the following equivalence relation on \mathbb{N} , $c \simeq_{m+1} c'$ iff $c < (m+1) \vee c' < (m+1) \Rightarrow c = c'$. Note that if $c \simeq_{m+1} c'$ then a transition is enabled at c if and only if it is enabled at c' . We can extend this equivalence to configurations of the CCA as follows. Let $(q_1, h_1) \simeq_{m+1} (q_2, h_2)$ iff $q_1 = q_2$ and $\forall d \in \mathbb{D}, h_1(d) \simeq_{m+1} h_2(d)$.

Lemma 6. *If C_1, C_2 are two configurations of the CCA such that $C_1 \simeq_{m+1} C_2$, then $\forall w \in (\Sigma \times \mathbb{D})^*, C_1 \vdash_w^* C'_1 \implies \exists C'_2, C_2 \vdash_w^* C'_2$ and $C'_1 \simeq_{m+1} C'_2$.*

Proof. Proof by induction on the length of w . For the base case observe that any transition enabled at C_1 is enabled at C_2 and the counter updates respects the equivalence. For the inductive case consider the word $w.a$. By induction hypothesis

$C_1 \vdash_w^* C'_1 \implies \exists C'_2, C_2 \vdash_w^* C'_2$ and $C'_1 \simeq_{m+1} C'_2$. If $C'_1 \vdash_a C''_1$ then using the above argument there exists C''_2 such that $C'_2 \vdash_a C''_2$ and $C''_1 \simeq_{m+1} C''_2$. \square

In fact the lemma holds for any $N \geq m + 1$, where m is the maximum constant used in Δ . This observation paves the way for proving the decidability of the emptiness problem.

3. Decision Problems

Since the space of configurations of a CCA is infinite, reachability is in general non-trivial to decide. We now show that the emptiness problem is elementarily decidable.

Theorem 7. *The non-emptiness problem for CCA is EXPSpace-complete.*

3.1. Upper bound

We reduce the emptiness problem of CCA to the covering problem on Petri nets ([11]). For checking emptiness, we can omit the $\Sigma \times \mathbb{D}$ labels from the configuration graph; we are then left only with counter behaviour. However since we have unboundedly many counters, we are led to the realm of multi-counter automata, or vector addition systems.

Definition 8. *An ω -counter machine B is a tuple (Q, Δ, q_0) where Q is a finite set of states, $q_0 \in Q$ is the initial state and $\Delta \subseteq (Q \times C \times Inst \times U \times Q)$, where C is a finite subset of \mathcal{C} and U is a finite subset of \mathbb{N} .*

A configuration of B is a pair (q, h) , where $q \in Q$ and $h : \mathbb{N} \rightarrow \mathbb{N}$. The initial configuration of B is (q_0, h_0) where $h_0(i) = 0$ for all i in \mathbb{N} . A run of B is a sequence $\gamma = (q_0, h_0)(q_1, h_1) \dots (q_n, h_n)$ such that for all i such that $0 \leq i < n$, there exists a transition $t_i = (p, c, \pi, m, q) \in \Delta$ such that $p = q_i$, $q = q_{i+1}$ and there exists j such that $h(j) \models c$, and the counters are updated in a similar fashion to that of CCA.

The reachability problem for B asks, given $q \in Q$, whether there exists a run of B from (q_0, h_0) ending in (q, h) for some h (“Can B reach q ?”).

Lemma 9. *Checking emptiness for CCA can be reduced to checking reachability for ω -counter machines.*

Proof. It suffices to show, given a CCA, $A = (Q, \Delta, q_0, F)$, where $F = \{q\}$, that there exists a counter machine $B_A = (Q, \Delta', q_0)$ such that A has an accepting run on some data word exactly when B_A can reach q . (When F is not a singleton, we simply repeat the construction.) Δ' is obtained from Δ by converting every transition (p, a, c, π, m, q) to (p, c, π, m, q) . Now, let $L(A) \neq \emptyset$. Then there exists a data word w and an accepting run $\gamma = (q_0, h_0)(q_1, h_1) \dots (q_n, h_n)$ of A on w , with $q_n = q$. Let $g : \mathbb{N} \rightarrow \mathbb{D}$ be an enumeration of data values. It is easy to see that $\gamma' = (q_1, h_0 \circ g)(q_1, h_1 \circ g) \dots (q_n, h_n \circ g)$ is a run of B_A reaching q .

(\Leftarrow) Suppose that B_A has a run $\eta = (q_0, h_0)(q_1, h_1) \dots (q_n, h_n)$, $q_n = q$. It can be seen that $\eta' = (q_0, h_0 \circ g^{-1})(q_1, h_1 \circ g^{-1}) \dots (q_n, h_n \circ g^{-1})$ is an accepting run of A on $w = (a_1, d_1) \dots (a_n, d_n)$ where w satisfies the following. Let (p, c, π, m, q) be the transition of B_A taken in the configuration (q_i, h_i) , and d_k such that $h_i(d_k) \models c$. Then by the definition of B_A there exists a transition (p, a, c, π, m, q) in Δ . Then it should be the case that $a_{i+1} = a$ and $d_{i+1} = g(d_k)$. \square

Proposition 10. *Checking non-emptiness of ω -counter machines is decidable.*

Let $s \subseteq \mathbb{N}$, and c a constraint. We say $s \models c$, if for all $n \in s$, $n \models c$.

We define the following partial function Bnd on all finite and co-finite subsets of \mathbb{N} . Given $s \subseteq_{fin} \mathbb{N}$, $Bnd(s)$ is defined to be the least number greater than all the elements in s . Given $s \subseteq_{co-finite} \mathbb{N}$, $Bnd(s)$ is defined to be $Bnd(\mathbb{N} \setminus s)$. Given an ω -counter machine $B = (Q, \Delta, q_0)$ let $m_B = \max\{Bnd(s) \mid s \models c, c \text{ is used in } \Delta\}$. It is worth noting that m_B is $\mathcal{O}(|A|)$.

We construct a Petri net $N_B = (S, T, F, M_0)$ where,

- $S = Q \cup \{i \mid i \in \mathbb{N}, 1 \leq i \leq m_B\}$.
- T is defined according to Δ as follows. Let $(p, c, \pi, n, q) \in \Delta$ and let i be such that $0 \leq i \leq m_B$ and $i \models c$. Then we add a transition t such that $\bullet t = \{p, i\}$ and $t^\bullet = \{q, i'\}$, where (i) if π is \uparrow^+ then $i' = \min\{m_B, i + n\}$, and (ii) if π is \downarrow then $i' = \min\{m_B, n\}$. Note that i can be zero, in which case we add edges only for the places in $[m_B]$.
- The flow relation F is defined according to $\bullet t$ and t^\bullet for each $t \in T$.
- The initial marking is defined as follows. $M_0(q_0) = 1$ and for all p in S , if $p \neq q_0$ then $M_0(p) = 0$.

Let M be any marking of N_B . We say that M is a *state marking* if there exists $q \in Q$ such that $M(q) = 1$ and $\forall p \in Q$ such that $p \neq q$, $M(p) = 0$. When M is a state marking, and $M(q) = 1$, we speak of q as the state marked by M . For $q \in Q$, define $M_f(q)$ to be set of state markings that mark q . It can be shown, from the construction of N_B , that in any reachable marking M of N_B , if there exists $q \in Q$ such that $M(q) > 0$, then M is a state marking, and q is the state marked by M .

We now show that the counter machine B can reach a state q iff N_B has a reachable marking which covers a marking in $M_f(q)$. We define the following equivalence relation on \mathbb{N} , $m \simeq_{m_B} n$ iff $(m < m_B) \vee (n < m_B) \Rightarrow m = n$. We can lift this to the bags (in ω -counters) in the natural way: $h \simeq_{m_B} h'$ iff $\forall i (h(i) < m_B) \vee (h'(i) < m_B) \Rightarrow h(i) = h'(i)$. It can be easily shown that if $h \simeq_{m_B} h'$ then a transition is enabled at h if and only if it is enabled at h' .

Let μ be a mapping of B -configurations to N_B -configurations as follows: given $\chi = (q, h)$, define $\mu(\chi) = M_\chi$, where

$$M_\chi(p) = \begin{cases} 1 & \text{iff } p = q \\ 0 & \text{iff } p \in Q \setminus \{q\} \\ |[p]| & \text{iff } p \in P \setminus Q, p \neq 0 \end{cases}$$

Above $[p]$ denotes the equivalence class of p under \simeq_{m_B} on \mathbb{N} in h . Now suppose that B reaches q . Let the resulting configuration be $\chi = (q, h)$. We claim that the marking $\mu(\chi)$ of N_B is reachable (from M_0) and covers $M_f(q)$. Conversely if a reachable marking M of N_B covers $M_f(q)$, for some $q \in Q$, then there exists a reachable configuration $\chi = (q, h)$ of B such that $\mu(\chi) = M$. This is proved by a simple induction on the length of the run.

Since the covering problem for Petri nets is decidable, so is reachability for ω -counter machines and hence emptiness checking for CCA is decidable.

Complexity of Emptiness checking : The decision procedure discussed above runs in EXPSPACE[11], and thus we have elementary decidability. Note that the representation of constants in unary is a crucial assumption about the EXPSPACE upper bound. When the constants are represented in binary, we do not know whether the upper bound still holds.

3.2. Lower bound

We now show that the emptiness problem is also EXPSPACE-hard. Effectively this is a reduction of the covering problem again, but for technical convenience, we use multicounter automata.

A k -multicounter automaton with weak acceptance is a tuple $A = (Q, \Sigma, \Delta, q_0, F)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is a set of final states. The transition relation is of the form $\Delta \subseteq_{fin} (Q \times \Sigma \times \mathbb{N}^k \times \mathbb{N}^k \times Q)$. The two vectors in the transition specify decrements and increments of the counters.

The automaton works as follows: it has k -counters, denoted by $\bar{v} = (v_1, \dots, v_k)$ which hold non-negative counter values. A configuration of the machine is of the form (q, \bar{v}) where $q \in Q$ and $\bar{v} \in \mathbb{N}^k$. The initial configuration is $(q_0, \bar{0})$. Given a configuration (q, \bar{v}) the automaton can go to a configuration (q', \bar{v}') on letter a if there is a transition $(q, a, v_{dec}^-, v_{inc}^-, q')$ such that $\bar{v} - v_{dec}^- \geq \bar{0}$ (pointwise) and $\bar{v}' = \bar{v} - v_{dec}^- + v_{inc}^-$. A final configuration is one in which the state is final.

The problem of checking non-emptiness of a multicounter automaton with weak acceptance is known to be EXPSPACE-hard [12].

Any multicounter automaton $M = (Q, \Sigma, \Delta, q_0, F)$ can be converted to another (in a "normal form"): $M' = (Q', \Sigma, \Delta', q_0, F)$ such that $L(M)$ is non-empty if and only if $L(M')$ is non-empty and M' uses only unit vectors or zero vectors in its transitions. A unit vector is of the form (b_1, b_2, \dots, b_k) where there is a unique $i \in [k]$ such that $b_i = 1$ and for $j \neq i$, $b_j = 0$. That is M' decrements or increments at most one counter in each transition.

Δ' is obtained as follows. Let $t = (q, a, v_{dec}^-, v_{inc}^-, q')$. Let $\bar{u}_1, \bar{u}_2, \dots, \bar{u}_n$ be a sequence of unit vectors such that $v_{dec}^- = \sum_i \bar{u}_i$ and $\bar{u}'_1, \bar{u}'_2, \dots, \bar{u}'_m$ be a sequence of unit vectors such that $v_{inc}^- = \sum_i \bar{u}'_i$. We add intermediate states to rewrite t by

the following sequence of transitions,

$$(q, a, \bar{u}_1, \bar{0}, q_{(t, \bar{u}_1)}), (q_{(t, \bar{u}_1)}, a, \bar{u}_2, \bar{0}, q_{(t, \bar{u}_2)}), \dots,$$

$$(q_{(t, \bar{u}_n)}, a, \bar{0}, \bar{u}'_1, q_{(t, \bar{u}'_1)}), (q_{(t, \bar{u}'_1)}, a, \bar{0}, \bar{u}'_2, q_{(t, \bar{u}'_2)}), \dots,$$

$$(q_{(t, \bar{u}'_{m-1})}, a, \bar{0}, \bar{u}'_m, q')$$

Lemma 11. *$L(M)$ is non-empty if and only if $L(M')$ is non-empty.*

Proof. By an easy induction on the length of the run. It is easy to see that for every accepting run ρ of M we have an accepting run ρ' of M' , this is achieved by replacing every transition t in the run ρ by the corresponding sequence of transitions. For the reverse direction, we need to show that every run accepting run ρ' of M' can be translated to an accepting run ρ of M . This is possible since the intermediate states added to obtain the transitions in M' are unique for each transition t in M . Hence for every sequence of transitions taking M' from q_1 to q_2 where $q_1, q_2 \in Q$ there is a unique transition t which takes M from q_1 to q_2 . By doing an induction on the number of states occurring in ρ' which are from Q we can show that there is a valid run ρ which is accepting. □

Next we convert M' to a CCA thus establishing a lower bound of EXPSPACE for the emptiness problem. Let $M' = (Q, \Sigma, \Delta, q_0, F)$ be a k -multicounter automaton in normal form. We construct the automaton $A = (Q, \Sigma, \Delta_A, q_0, F)$. Let $t = (q, a, \bar{u}, \bar{u}', q')$ where \bar{u}, \bar{u}' are either unit or zero vectors. If \bar{u} is a i -th unit vector and \bar{u}' is a zero vector, we add a transition $t_A = (q, a, (x = i), (\downarrow, 0), q')$ to Δ_A . If \bar{u} is a i -th unit vector and \bar{u}' is j -th unit vector, we add a transition $t_A = (q, a, (x = i), (\downarrow, j), q')$ to Δ_A . If \bar{u} is a zero vector and \bar{u}' is a j -th unit vector, we add a transition $t_A = (q, a, (x = 0), (\downarrow, j), q')$ to Δ_A .

Lemma 12. *$L(M')$ is non-empty if and only if $L(A)$ is non-empty.*

Proof. The proof is by induction on the length of the run. First we define a mapping from configurations of A to configurations of M' in the following manner, $\mu((q, \bar{h})) = (q, \bar{v})$ where $v_i = |\{j \mid \bar{h}(j) = i\}|$. We show, by induction on the length of the run, that for every configuration χ reachable by A there is a configuration ψ of M' such that $\mu(\chi) = \psi$ and conversely for every configuration ψ reachable by M' there is a configuration χ reachable by A such that $\mu(\chi) = \psi$.

For the base case, it is evident that $\mu((q_0, \bar{h}_0)) = (q_0, \bar{0})$.

Suppose that $\chi = (q, \bar{h})$ is a configuration reachable in l steps, and that the transition $t = (q, a, x = j, (\downarrow, i), q')$ is enabled at χ . Therefore there is a counter holding the value j . By induction hypothesis there exists a configuration ψ such that $\mu(\chi) = \psi = (q, \bar{v})$ such that $v_j > 0$. After the transition t , the number of counters holding the value j decreases by one and the number of counters holding the value

i increases by one (if $i \neq 0$). This is achieved by the transition $(q, a, \bar{u}_j, \bar{u}_i, q')$ in Δ' , preserving the map μ .

Conversely, suppose a configuration $\psi = (q, \bar{v})$ is reachable by M' in l steps. Then by induction hypothesis we have a configuration χ reachable by the automaton A such that $\mu(\chi) = \psi$. Suppose a transition $t' = (q, a, \bar{u}_i, \bar{u}_j, q')$ is enabled in ψ resulting in ψ' .

Consider the case where $\bar{u}_i \neq \bar{0}$ and $\bar{u}_j \neq \bar{0}$. By construction t' is obtained from a transition $t = (q, a, (x = i), \downarrow, j, q')$. We choose the smallest counter holding the value zero and apply the transition t , resulting in ξ' such that $\mu(\xi') = \psi'$. The remaining cases are similar. □

The reduction from M to M' is not in polynomial time when the constants in the transitions of the Multicounter automata are encoded in binary. However, we observe that the EXPSPACE-hardness for covering problem from [11, 12] can be obtained with updates restricted to the values $-1, 0$ and 1 . Hence, the lower bound extends to the scenario where the constants are represented in binary.

3.3. Word problem

Since emptiness checking is of such high complexity, one may wonder whether the model is complex enough to render even the word problem to be hard: the simplest algorithmic question of how one can check whether a given word is accepted or not. The important thing to note is that during a run, the size of the configuration is bounded by the length of the input data word. Therefore a non-deterministic Turing machine can easily guess a path in polynomial time and check for acceptance. Hence the word problem is easily seen to be in NP. Interestingly, it turns out to be NP-hard as well.

Theorem 13. *The word problem for CCA is NP-complete.*

The proof is by reduction of the satisfiability problem for 3-CNF formulas to the word problem for CCAs. Given the 3-CNF formula, we code it up as a data word, where data values are used to remember the identity of literals in clauses. We use a two letter alphabet with $+$, $-$ indicating whether a propositional variable occurs positively or negatively. Data values stand for the propositional variables themselves. Thus a pair $(+, d_1)$ asserts that the first boolean variable occurs positively.

We show the coding by an example, let $\varphi \equiv (p_1 \vee \neg p_3 \vee p_4) \wedge (\neg p_2 \vee p_5 \vee p_1) \wedge (\neg p_3 \vee \neg p_4 \vee p_5)$, we construct the corresponding word $w = (+, d_1)(-, d_3)(+, d_4)(\#, d)(-, d_2)(+, d_5)(+, d_1)(\#, d)(-, d_3)(-, d_4)(+, d_5)(\#, d) \in (\{+, -, \#\} \times \mathbb{D})^*$.

The non-deterministic automaton checks satisfiability in the following way. Every time the automaton encounters a new data value (representing a propositional variable), the automaton non-deterministically assigns a boolean value and stores

it in the counter (1 for \perp and 2 for \top) corresponding to the data value, in the future whenever the same data value occurs the counter is consulted to obtain the assigned value to the propositional variable. The automaton evaluates each clause and carries the partial evaluation in its state. Finally the automaton accepts the word if the formula evaluates to \top .

4. Extensions

We observe that the model admits many extensions, without substantially affecting the main decidability result.

4.1. Many bags

Instead of working with one bag of counters, the automaton can use several bags of counters, much as multiple registers are used in the register automaton. It is easy to formally define CCA with k -bags, using k -tuples of constraints on guards. An interesting fact is that a CCA with k -bags can be converted to a CCA with one bag. This can be achieved because of the following:

- Any CCA, no matter how many bags it has, can be converted to a CCA whose counter values are bounded (We take the maximum constant used in Δ and rewrite the transitions in such a way that we never increment a counter once it reaches that value).
- A k -bag CCA, whose counters are bounded can be simulated by a CCA with one bag, by using a bit representation. Since the counters are bounded, we know a priori how many bits are needed to represent each bag.

4.2. Checking any counter

Another strengthening involves checking for the presence of *any* counter satisfying a given constraint and updating it. The idea is to extend the transitions to the following form, $t = (q, a, \tau_0, \tau_1, \dots, \tau_n, q')$ where each $\tau_i \in C \times Inst \times U$ is of the form (c_i, π_i, m_i) . The intended semantics of the transition is as follows. Suppose that the current letter is a and data value is d_0 . The transition t is enabled if there exist distinct data values d_1, \dots, d_n such that, for every $i \in [n]_0$, d_i satisfies τ_i . On the occurrence of t each d_i is updated with respect to τ_i . Note that in this way we can modify the counter of a data value which is not the current data value.

Formally a CCA with context check, denoted CCAC, is a tuple (Q, Δ, I, F) , where the transition relation is modified to be $\Delta \subseteq (Q \times \Sigma \times (C \times Inst \times U)^n \times Q)$, where C is a finite subset of \mathcal{C} , U is a finite subset of \mathbb{N} , while everything else remain the same.

Let A be a CCAC. A configuration of A is a pair (q, h) , where $q \in Q$ and $h \in \mathbb{B}$. The initial configuration of A is given by (q_0, h_0) , where h_0 is the empty bag; that is, $\forall d \in \mathbb{D}, h_0(d) = 0$ and $q_0 \in I$.

Given a data word $w = (a_1, d_1), \dots, (a_m, d_m)$, a run of A on w is a sequence $\gamma = (q_0, h_0)(q_1, h_1) \dots (q_m, h_m)$ such that $q_0 \in I$ and for all $i, 0 \leq i < m$, there exists a transition $t_i = (q, a, \tau_0, \tau_1, \dots, \tau_n, q') \in \Delta$ where $\tau_j = (c_j, \pi_j, m_j)$ such that $q = q_i, q' = q_{i+1}, a = a_{i+1}$ and:

- $h_i(d_{i+1}) \models c_0$ and there exist distinct e_1, \dots, e_n in \mathbb{D} such that for all $j \in \{1, \dots, n\}, e_j \neq d_{i+1}$ and $h_i(e_j) \models c_j$.
- h_{i+1} is given by:

$$h_{i+1} = \begin{cases} h_i \oplus (d_{i+1}, m') & \text{if } \pi_0 = \uparrow^+, m' = h_i(d_{i+1}) + m_0 \\ h_i \oplus (d_{i+1}, m_0) & \text{if } \pi_0 = \downarrow \\ h_i \oplus (e_j, m') & \text{if } \pi_j = \uparrow^+, m' = h_i(e_j) + m_j \\ h_i \oplus (e_j, m_j) & \text{if } \pi_j = \downarrow \end{cases}$$

We define ω -counter machines with context in a similar way: such a machine is a tuple (Q, Δ, q_0) where Q is finite set of states, q_0 is the initial state and $\Delta \subseteq (Q \times (C \times Inst \times U)^n \times Q)$, where C is a finite subset of \mathcal{C} , U is a finite subset of \mathbb{N} . A run of an ω -counter machine with context is defined analogously to that of CCA with context. We can then easily show that checking emptiness for CCA with context can be reduced to checking reachability for ω -counter machines with context.

Finally, the following proposition shows that checking emptiness of CCA with context is decidable in EXPSPACE.

Proposition 14. *Checking non-emptiness of ω -counter machines with context is decidable.*

Proof. Given an ω -counter machine $B = (Q, \Delta, q_0)$, we define m_B as in the proof of Proposition 10.

We construct a Petri net $N_B = (S, T, F, M_0)$ where,

- $S = Q \cup \{i \mid i \in \mathbb{N}, 1 \leq i \leq m_B\}$.
- T is defined according to Δ as follows. Let $t = (q, a, \tau_0, \tau_1, \dots, \tau_n, q')$ be a transition in Δ where $\tau_j = (c_j, \pi_j, m_j)$ and let i_0, i_1, \dots, i_n be such that $0 \leq i_j \leq m_B$ and $i_j \models c_j$. Then we add a transition t such that $\bullet t = \{p, i_0, i_1, \dots, i_n\}$ and $t^\bullet = \{q, i'_0, i'_1, \dots, i'_n\}$ (take note of the fact that $\bullet t$ and t^\bullet are multisets), where (i) if π_j is \uparrow^+ then $i'_j = \min\{m_B, i_j + n_j\}$, and (ii) if π_j is \downarrow then $i'_j = \min\{m_B, n_j\}$. Note that i_j can be zero, in which case we add edges only for the places in $[m_B]$.
- The flow relation F is defined according to $\bullet t$ and t^\bullet for each $t \in T$.
- The initial marking is defined as follows. $M_0(q_0) = 1$ and for all p in S , if $p \neq q_0$ then $M_0(p) = 0$.

The rest of the proof is similar to the proof of Proposition 10 with obvious modifications. \square

4.3. The language of constraints

The language of constraints can be strengthened. Previously, the constraints were of the form $c = (\text{op}, e)$. Consider the following language, the language of Presburger arithmetic. The terms in this language are given by the grammar,

$$t ::= 0 \mid 1 \mid t_1 + t_2 \mid x, x \in V$$

where V is a countably infinite set of variables. The formulas of this language are given by:

$$\varphi ::= t_1 \leq t_2 \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \exists x.\varphi.$$

The semantics is given as follows. The variables takes natural numbers as their values and $+$ is interpreted as addition. We call a formula $\varphi(x)$ with one free variable, a Presburger constraint. We say that $k \in \mathbb{N}$ satisfies $\varphi(x)$ if $k \models \varphi(x)$. Note that the set of numbers satisfying a constraint may be neither finite nor co-finite. For example, the formula $\exists y.y + y = x$ defines the set of even numbers.

Let \mathcal{C}_p be the set of all Presburger constraints. We define CCA with Presburger constraints, abbreviated as CCA + Presburger, as a tuple $\text{CCA} = (Q, \Delta, I, F)$, where the transition relation is modified to be $\Delta \subseteq (Q \times \Sigma \times C_p \times \text{Inst} \times U \times Q)$, where C is a finite subset of \mathcal{C}_p , while everything else remain the same. The definitions of run and acceptance condition is defined in the obvious way.

A set of natural numbers D is *eventually periodic* iff there exists positive numbers m and p such that for all n greater than m , $n \in D$ iff $n + p \in D$. From the following theorem we know that the set of numbers satisfying a Presburger constraint is eventually periodic.

Theorem 15 ([13]) *A set of natural numbers is representable in Presburger arithmetic iff it is eventually periodic.*

Using this, the decision procedure in Section 3 can be modified to check the emptiness of CCA with Presburger constraints. As above, we define ω -counter machines with Presburger constraints: such a machine is a tuple (Q, Δ, q_0) where Q is a finite set of states, $q_0 \in Q$ is the initial state and $\Delta \subseteq (Q \times C_p \times \text{Inst} \times U \times Q)$, where C_p is a finite subset of \mathcal{C}_p and U is a finite subset of \mathbb{N} . Runs are defined in the natural way.

We can then easily show that checking emptiness for CCA with Presburger constraints can be reduced to checking reachability for ω -counter machines with Presburger constraints. Then the following proposition shows that checking emptiness of CCA with Presburger constraints is decidable in EXPSPACE.

Proposition 16. *Checking non-emptiness of ω -counter machines with Presburger constraints is decidable.*

Proof. Given an ω -counter machine $B = (Q, \Delta, q_0)$, let c_1, \dots, c_n be the constraints used in Δ . From the theorem above, we know that c_1, \dots, c_n are eventually periodic

with the pairs $(m_1, p_1), \dots, (m_n, p_n)$. We take $m = m_1 + \dots + m_n$ and p as the least common multiple of p_1, \dots, p_n .

We construct a Petri net $N_B = (S, T, F, M_0)$ where,

- $S = Q \cup \{i \mid i \in \mathbb{N}, 1 \leq i \leq m + p\}$.
- T is defined according to Δ as follows. Let $(p, c, \pi, n, q) \in \Delta$ and let i be such that $0 \leq i \leq m + p$ and $i \models c$. Then we add a transition t such that $\bullet t = \{p, i\}$ and $t^\bullet = \{q, i'\}$, where (i) if π is \uparrow^+ then $i' = \min\{i + n, m + (i + n - m) \bmod p\}$, and (ii) if π is \downarrow then $i' = \min\{n, m + (n - m) \bmod p\}$. Note that i can be zero, in which case we add edges only for the places in $[m_B]$.
- The flow relation F is defined according to $\bullet t$ and t^\bullet for each $t \in T$.
- The initial marking is defined as follows. $M_0(q_0) = 1$ and for all p in S , if $p \neq q_0$ then $M_0(p) = 0$.

The rest of the proof is similar to the proof of Proposition 10 with obvious modifications. □

4.4. Two-way CCA

A two-way CCA is system (Q, Δ, I, F) , where Q, I, F are as usual, the transition relation is $\Delta \subseteq (Q \times \Sigma \times C \times Inst \times U \times Q \times \{L, R, S\})$. A configuration of A is a triple (q, i, h) , where $q \in Q$, $i \in \mathbb{N}$ and $h \in \mathbb{B}$, where the variable i denotes the position of the head. The initial configuration of A is given by $(q_0, 1, h_0)$, where h_0 is the empty bag; that is, $\forall d \in \mathbb{D}, h_0(d) = 0$ and $q_0 \in I$.

Given a data word $w = (a_1, d_1), \dots, (a_n, d_n)$, a run of A on w is a sequence $\gamma = (q_0, i_0, h_0)(q_1, i_1, h_1) \dots (q_l, i_l, h_l)$ such that $q_0 \in I$ and for all $j, 0 \leq j < l$, there exists a transition $t_j = (q, a, c, \pi, m, q', \mu) \in \Delta$ such that $q = q_j, q' = q_{j+1}, a = a_{i_j}$ and $h_j(d_{i_j}) \models c$. The resulting counter configuration h_{j+1} is defined as in the case of CCA. Finally, the updated position of the head is determined in the following way,

$$i_{j+1} = \begin{cases} i_j - 1 & \text{if } \mu = L \\ i_j + 1 & \text{if } \mu = R \\ i_j & \text{if } \mu = S \end{cases}$$

We assume that the input word is wrapped with end markers so that if the machine tries to go off the boundary of the word it halts erroneously. We say a run is accepting if the machine halts in a final state.

As we will see below, the emptiness problem is undecidable for the two-way extension of CCAs.

4.5. Alternating CCA

An alternating CCA is system $(Q = Q_\forall \uplus Q_\exists, \Delta, I)$, where Q, I, Δ are as usual. Note that there is no designated set of final states; instead, the state set is partitioned

into a set of universal states Q_{\forall} and a set of existential states Q_{\exists} . A configuration of A is a tuple (q, h) , where $q \in Q$ and $h \in \mathbb{B}$. The initial configuration of A is given by (q_0, h_0) , $q_0 \in I$ and h_0 is the empty bag; that is, $\forall d \in \mathbb{D}, h_0(d) = 0$ and $q_0 \in I$.

Given a data word $w = (a_1, d_1), \dots, (a_n, d_n)$, assume that the automaton is at position i with configuration (q_i, h_i) . We say that (q_{i+1}, h_{i+1}) is a valid successor configuration if there exists a transition $t = (q, a, c, \pi, m, q', \mu) \in \Delta$ such that $q = q_i$, $q' = q_{i+1}$, $a = a_{i+1}$ and $h_i(d_{i+1}) \models c$. The resulting counter configuration h_{j+1} is defined as in the case of CCA.

We say that a configuration (q, h) is accepting if

- (1) $q \in Q_{\forall}$ and all of its valid successor configurations are accepting. (Note that a configuration with no valid successor configurations is accepting.)
- (2) $q \in Q_{\exists}$ and there is a valid successor configuration (q', h') which is accepting.

Finally we say that the word is accepted if the initial configuration (q_0, h_0) is accepting.

Theorem 17. *The emptiness problem is undecidable for Two-way CCAs and for Alternating CCAs.*

Proof. We do the proofs simultaneously by reducing the Post’s Correspondence Problem to the emptiness of two-way CCA and of alternating CCA. Without loss of generality, assume that we are given a PCP instance I which is a set of ordered pairs of non-empty strings over the alphabet $\Sigma = \{l_1, l_2, \dots, l_k\}$, that is $I = \{(u_i, v_i) \mid i \in [n], u_i, v_i \in \Sigma^+\}$. A solution for I is a finite sequence of integers i_0, i_1, \dots, i_m , all of them from the set $\{1, \dots, n\}$ such that $u_{i_0}u_{i_1} \dots u_{i_m} = v_{i_0}v_{i_1} \dots v_{i_m}$. We define a two-way CCA which accepts precisely all solutions of I .

For this purpose, we code the PCP solution as a dataword, in the following way. Let $\bar{\Sigma} = \{\bar{l}_1, \bar{l}_2, \dots, \bar{l}_k\}$ and $\hat{\Sigma} = \Sigma \cup \bar{\Sigma}$. Given a word $w = a_1a_2 \dots a_n$ in Σ^* , we denote by \bar{w} the word $\bar{a}_1\bar{a}_2 \dots \bar{a}_n$ in $\bar{\Sigma}^*$.

A solution of I is a dataword w over $\hat{\Sigma}$ such that,

- (I) The string projection of the word is in $(u_1\bar{v}_1 + u_2\bar{v}_2 \dots + u_n\bar{v}_n)^+$.
- (II) Every data value d occurring in w appears precisely twice, once labelled by a letter from Σ and once by a letter from $\bar{\Sigma}$. Moreover if d is labelled by $l_i \in \Sigma$ in w if and only if it is labelled by $\bar{l}_i \in \bar{\Sigma}$ in v (the second occurrence).
- (III) The ordering of data values in the positions labelled by Σ is exactly the same as the ordering of data values in positions labelled by $\bar{\Sigma}$. Formally, let d and e are data values occurring in w . Let d_{Σ} and e_{Σ} be the positions where d and e are labelled by letters from Σ . Similarly, let $d_{\bar{\Sigma}}$ and $e_{\bar{\Sigma}}$ be the positions where d and e are labelled by letters from $\bar{\Sigma}$. The condition says that $d_{\Sigma} < e_{\Sigma}$ if and only if $d_{\bar{\Sigma}} < e_{\bar{\Sigma}}$.

It is easy to see that there is data word w satisfying the above three conditions iff I has a solution. We show that two-way CCA and alternating CCA can check these three conditions.

- (1) The first condition is a regular property and can be checked by any finite state automaton. Hence it is easily checked by a CCA.
- (2) The conjunction of the following four conditions is equivalent to condition (II).
 - (a) Data values occurring in Σ -labelled positions are all distinct.
 - (b) Data values occurring in $\bar{\Sigma}$ -labelled positions are all distinct.
 - (c) All data values occurring under $\bar{\Sigma}$ -labels occur under Σ -labels as well.
 - (d) All data values occurring under Σ -labels occur under $\bar{\Sigma}$ -labels as well.

Note that each of these conditions can be checked by a CCA. Since CCAs are closed under intersection, a CCA can verify condition (II).

- (3) Condition (III) is checked by a two-way CCA in the following way. We assume that conditions (I) and (II) are verified independently. Given a position i labelled by a letter from Σ we say that the position $j > i$ is the Σ -successor of i iff j is a position labelled by a letter from Σ and all positions k , $i < k < j$ are labelled by letters from $\bar{\Sigma}$. Similarly we can define $\bar{\Sigma}$ -successor of a $\bar{\Sigma}$ -labelled position. Let i and j be Σ -successors and let d_i and d_j be the corresponding data values. We know that d_i and d_j occurs under $\bar{\Sigma}$ as well. Let those positions be \bar{i} and \bar{j} . For each Σ -successors i, j the automaton verifies that \bar{i} and \bar{j} are $\bar{\Sigma}$ successors.

To achieve this, assume that the automaton starts in a Σ position i , it resets the counter of d_i to 1 and goes to next Σ -labelled position j . It increments the counter of d_j to 2. Now, the automaton moves to left end marker and makes a left to right sweep ignoring all Σ positions. During this sweep the automaton stops when it sees the data value d_j under a $\bar{\Sigma}$ label. It resets counter of d_i to zero and then verifies that the next $\bar{\Sigma}$ position has the data value d_j with the help of the counter. After this step the automaton goes to the left end of the word and again makes a right sweep. This time it stops when it sees the data value d_j under a Σ label. Then the procedure is repeated for position j . Finally the machine halts and accepts when it reaches the last Σ position in the data word.

- (4) Condition (III) is checked by an alternating CCA in the following way. The automaton starts in state q_0 . In this state automaton records all the data values it has seen till the current position. Whenever it sees a fresh data value, it makes a universal branching, one branch continues in state q_0 and one branch goes to state q_1 . In the state q_1 the automaton verifies the following. Assume the fresh data value d occurs under a Σ label and let the data value on its Σ successor position is e . The automaton verifies that the positions where d and e are occurring under $\bar{\Sigma}$ labels are $\bar{\Sigma}$ successors. This can easily be done by incrementing the counters corresponding to d and e to specially designated values. The q_1 branching halts successfully after each verification. The q_0 branching accepts at

the end of the word. □

An important corollary of this result is that the universality problem for CCAs is undecidable, and hence language inclusion is undecidable as well.

Other interesting extensions relate to the kind of updates allowed and to acceptance conditions. While adding decrements to counters in CCA leads to undecidability of the emptiness problem, we can add **resets** to counters preserving decidability. A reset operation sets the corresponding counter value to zero. The acceptance condition we have in CCA is *global* in the sense that it relates only to the global control state rather than multiplicities encountered. We can strengthen the acceptance condition as follows: $A = (Q, \Delta, q_0, F, G)$ where (Q, q_0, Δ, F) are as before, and $G \subseteq_{fin} \mathbb{N}$. We say a final configuration (q, h) is accepting if $q \in F$ and $\forall d \in \mathbb{D}, h(d) \in G$ or $h(d) = 0$.

We then find that the non-emptiness problem (for CCAs with reset and counter conditions) continues to be decidable but becomes as hard as Petri net reachability, which is not even known to be elementarily decidable. This is proved by relating this class to that of class memory automata discussed below.

One standard restriction is the *deterministic* subclass. In the case of CCAs, this necessitates checking that transitions are not only label-wise deterministic on Σ but also that the constraints are non-intersecting. But since requirement specifications would be non-deterministic in general, it is not clear that the deterministic subclass is interesting.

5. Other Automata Models

CCAs are situated among a family of automata models that have been proposed for data languages. The simplest form of memory is a finite random access read-write storage device, traditionally called *register*. In *finite memory* automata [9], the machine is equipped with finitely many registers, each of which can be used to store one data value. Every automaton transition includes access to the registers, reading them before the transition and writing to them after the transition. The new state after the transition depends on the current state, the input letter and whether or not the input data value is already stored in any of the registers. If the data value is not stored in any of the registers, the automaton can choose to write it in a register. The transition may also depend on which register contains the encountered data value. Because of finiteness of the number of registers, in a sufficiently long word the automaton cannot distinguish between all data values. On the other hand, register automata have the capability of keeping the “latest information”, a capability that deterministic CCA do not have.

In the previous section, we considered an extension of CCAs with context. One nice aspect of this extension is that it can recognize all data languages accepted by register automata. This is achieved by the following scheme. Suppose that we want to simulate a k -register automaton, we use k bags to represent each of these

registers. In every bag, atmost one counter has the value one, others being zero. The intended meaning is that the data value corresponding to that counter is in the particular register. Whenever a write operation happens, not only do we make the counter of the current data value one, but also we decrement an arbitrary counter as described previously. The register read operation is straightforward to simulate. In this way, we can faithfully represent the registers using the k bags. We noted above that k bags can be simulated by one bag.

In class memory automata (CMA, [4]), a function assigns to every data value d the state of the automaton that was assumed after reading the previous position with value d . We can think of this as using hash tables, with values coming from a finite set. On reading a pair (a, d) , the automaton reads the table entry corresponding to d and makes a transition dependent on the table entry, the input letter a and the current state. The transition causes a change of state as well as updating of the table entry. The non-emptiness problem for CMAs is decidable, but as hard as Petri net reachability, which is not known to be even elementarily decidable.

We can show that the class of CCA-recognizable languages is strictly contained in the class of CMA-recognizable languages, but when we add resets and counter acceptance conditions as above, the class becomes exactly as expressive as CMAs. Indeed, we see CCAs as a natural restriction of CMAs yielding elementary decidability of the non-emptiness problem. Mapping the precise relation between these classes is interesting but part of a larger exercise and hence we omit it here.

Another simple computational model, based on transducers is the data automaton model introduced in [5]. It is shown in [4] that this model is exactly as expressive as CMAs.

6. Conclusion

We defined a class of automata on datawords and showed that its emptiness problem is decidable and is elementary. We also considered many extensions this class of automata which preserve elementary decidability.

But with an NP-hard word problem, EXPSPACE-hard non-emptiness question and undecidable language inclusion, working with data languages does seem daunting. However, given the need for verifying properties of systems with unboundedly many processes, the abstraction of infinite alphabets is yet worth preserving. What we need to consider are restrictions that are meaningful for systems of unbounded processes.

Acknowledgment

We thank the anonymous reviewers for their detailed comments, which have greatly helped to improve the content and presentation of the paper.

References

- [1] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. In Oscar H. Ibarra and Zhe Dang, editors, *Developments in Language Theory*, volume 4036 of

- Lecture Notes in Computer Science*, pages 1–13. Springer, 2006.
- [2] Jean-Michel Autebert, Joffroy Beauquier, and Luc Boasson. Langages sur des alphabets infinis. *Discrete Applied Mathematics*, 2:1–20, 1980.
 - [3] Manuel Baclet. Logical characterization of aperiodic data languages. Research Report LSV-03-12, Laboratoire Spécification et Vérification, ENS Cachan, France, September 2003. 16 pages.
 - [4] Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. In Erzsébet Csuhaj-Varjú and Zoltán Ésik, editors, *FCT*, volume 4639 of *Lecture Notes in Computer Science*, pages 88–99. Springer, 2007.
 - [5] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, Luc Segoufin, and Claire David. Two-variable logic on words with data. In *LICS*, pages 7–16. IEEE Computer Society, 2006.
 - [6] Patricia Bouyer. A logical characterization of data languages. *Inf. Process. Lett.*, 84(2):75–85, 2002.
 - [7] Patricia Bouyer, Antoine Petit, and Denis Thérien. An algebraic characterization of data and timed languages. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *CONCUR*, volume 2154 of *Lecture Notes in Computer Science*, pages 248–261. Springer, 2001.
 - [8] Stephane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. In *LICS*, pages 17–26. IEEE Computer Society, 2006.
 - [9] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994.
 - [10] Michael Kaminski and Tony Tan. Regular expressions for languages over infinite alphabets. *Fundam. Inform.*, 69(3):301–318, 2006.
 - [11] Javier Esparza. Decidability and complexity of Petri net problems - an Introduction. In *Advances in Petri Nets*, volume 1491 of *Lecture Notes in Computer Science*, pages 374–428, Springer-Verlag, 1998.
 - [12] R. Lipton. The reachability problem requires exponential space. Research Report 62, Yale University, 1976.
 - [13] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Harcourt/Academic Press, second edition, 2001.
 - [14] Alexei Lisitsa and Igor Potapov. Temporal logic with predicate lambda-abstraction. In *TIME*, pages 147–155, 2005.
 - [15] Alexei Lisitsa and Igor Potapov. On the computational power of querying the history. *Fundam. Inform.*, 91(2):395–409, 2009.
 - [16] Alexei Lisitsa, Igor Potapov, and Rafiq Saleh. Automata on Gauss words. In *LATA*, volume 5457 of *Lecture Notes in Computer Science*, pages 505–517, 2009.
 - [17] Frank Neven, Thomas Schwentick, and Victor Vianu. Towards regular languages over infinite alphabets. In Jiri Sgall, Ales Pultr, and Petr Kolman, editors, *MFCS*, volume 2136 of *Lecture Notes in Computer Science*, pages 560–572. Springer, 2001.
 - [18] Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004.
 - [19] Friedrich Otto. Classes of regular and context-free languages over countably infinite alphabets. *Discrete Applied Mathematics*, 12:41–56, 1985.
 - [20] Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In Zoltán Ésik, editor, *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 41–57. Springer, 2006.