Theoretical Computer Science 411 (2010) 702–715

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

On notions of regularity for data languages☆

Henrik Björklund*,[1], Thomas Schwentick

Technische Universität Dortmund, Computer Science Chair 1, D-44227 Dortmund, Germany

A B S T R A C T

With motivation from considerations in XML database theory and model checking, data strings have been introduced as an extension of finite alphabet strings which carry, at each position, a symbol *and* a data value from an infinite domain. Previous work has shown that it is difficult to come up with an expressive yet decidable automaton model for data languages. Recently, such a model, *data automata*, was introduced. This paper introduces a simpler but equivalent model and investigates its expressive power, algorithmic and closure properties, and some extensions.

1. Introduction

The concept of regular string languages is clearly one of the most fundamental concepts in (theoretical) computer science. It has applications in basically all branches of computer science. It can be argued that the following properties were crucial to its success: (1) Expressiveness, (2) decidability, (3) efficiency, (4) closure properties, and (5) robustness. The notion of regularity has been successfully generalized to other kinds of structures, including infinite strings and finite or infinite, ranked or unranked trees. More recent applications of regular languages (on infinite strings and finite, unranked trees, respectively) can be found in model checking and XML processing.

In model checking, a system is given as a finite state model, and properties are specified in a logic like LTL. The step from the "real" system to its finite state representation usually involves many abstractions, especially with respect to data values (variables, process numbers, etc.). Often their range is restricted to a (small) finite domain. Even though this approach has been successful and found its way into large scale industrial applications, the finite abstractions have some inherent shortcomings. As an example, n identical processes with m states each give rise to an overall model size of m^n. (Symbolic model checking partially addresses this problem by "compressing" redundant states.) If the number of processes is unbounded and/or unknown in advance, the finite state approach fails. Previous work has shown that even then, decidability can be obtained by restricting the problem in various ways [9,1].

In XML document processing, regular concepts occur in various contexts. First, most applications restrict the structure of the allowed documents to conforming to a certain schema, which can be modeled as a regular tree language. Second, navigation (XPath) and transformation (XSLT) languages have tight connections to tree automata models and other regular description mechanisms (see, e.g., [14]). All of these approaches concentrate on document structure and ignore attribute and text values. From a database point of view this is not completely satisfactory: a *schema* should not only restrict the allowed (tree) structure, but also state integrity constraints, such as key or inclusion dependencies, on the data. This problem has been addressed (see, e.g., [2]), but just as in model checking, the methods largely rely on case-to-case analysis.

☆ This work was supported by the DFG Grant SCHW678/3-1.

* Corresponding author.
E-mail addresses: henrikb@cs.umu.se (H. Björklund), Thomas.Schwentick@udo.edu (T. Schwentick).
[1] Present address: Umeå University, Department of Computing Science, S901 87 Umeå, Sweden.

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.10.009

Thus, in both settings, the finite state abstraction leads to interesting results but does not address all problems arising in applications. In both cases, it would already be a big step if each string position (or tree node) could carry a *data value* in addition to its label. As any kind of arithmetic operation on the infinite domain quickly leads to undecidability of basic processing tasks (even a linear order on the domain is harmful), we concentrate on the setting where data values can only be tested for equality. Furthermore, in this paper we only consider finite *data strings*, i.e., finite strings where each position carries a symbol from a finite alphabet and a data value from an infinite domain.

Several specification mechanisms for data languages have been suggested, such as *register automata* [12], *pebble automata* [15], *quasi-regular expressions* [13], *data automata* [3], and *LTL with a freeze quantifier* [5]. For an overview, see [18]. There are also some investigations which assume more knowledge of the data [4,19]. Two observations are immediate from this body of work: (1) the landscape of data languages is very heterogeneous, i.e., pairs of defined classes are often incomparable, and (2) one quickly obtains undecidability.

Thus, the question remains of whether there is a notion of regularity for data languages with the five desirable properties mentioned above. The results so far indicate that there might not be a single such class sharing all required properties. Rather there could be several classes fulfilling the requirements only to a certain extent. The research dedicated to this question is therefore of an exploratory nature, taking a broad variety of models into account.

Here, our requirements on expressiveness are guarded by the goal of model checking in the presence of an unbounded number of processes. In this scenario, a computation naturally gives rise to a data string w, where the data values represent process identifiers. We aim at describing *global properties* of the computation, taking the whole string into consideration, as well as *local properties* which concern the actions of individual processes. As an example, consider processes sharing a printer with three kinds of events: a print job can be requested (r), start (s), and terminate (t). A global property could be that a started job must be terminated before the next job can be started, inducing a regular (finite alphabet) constraint of the form $(r^*sr^*tr^*)^*$. A natural local property is stated by $(rst)^*$, i.e., each process has arbitrarily many request–start–terminate cycles. We want to be able to specify global and local properties by (classical) regular languages R_{glob} and R_{loc}, respectively. Formalisms will differ in their ability to coordinate the local and global properties.

Register automata [12] are a quite natural decidable model for data languages. In [12], the data strings do not have a finite alphabet component, but the generalization of the model to data languages is straightforward. They are able to deal with any regular global properties, but their ability to specify local properties is very limited. For instance, they cannot express the local property of printers stated above. Another natural approach to specification is through logics. Data strings can be modeled in a straightforward way as finite structures. Due to the limited access to data values, they can be represented by an equivalence relation. First-order logic on data strings is undecidable, but the two-variable fragment has a decidable satisfiability problem [3]. In its decidability proof, the latter paper introduced a new automaton model for data strings, *data automata (DAs)*. As they have the expressive power described above and a decidable emptiness problem, they fulfill, at least to some extent, the requirements of (1) expressiveness and (2) decidability. The other three requirements were not studied in depth in [3]. Nevertheless, the paper gave a characterization of the class $\mathcal{R}$ of data languages accepted by DAs in terms of an existential monadic second-order logic and thus established a certain robustness of the class. In this paper, we study $\mathcal{R}$ and some extensions and restrictions more thoroughly.

Contributions. First, we address the robustness of $\mathcal{R}$. We exhibit some simplifications of data automata which do not affect their expressive power (cf. 3.6 and 3.7). We arrive at the equivalent model of *class memory automata (CMAs)*. We further confirm the expressiveness of $\mathcal{R}$ by showing that it (strictly) captures all data languages accepted by register automata (4.1).

We next turn to the complexity of model checking. We first consider register automata. Even though their data complexity is polynomial in time, the combined complexity is NP-complete [17]. The number k of registers turns out to be a crucial parameter here: with respect to k, the problem is $W[1]$-complete (5.2(b)). For class memory automata, things are even worse, as the data complexity of model checking is already NP-complete (5.1(b)). The high data complexity of CMAs suggests the consideration of *deterministic* CMAs. The existence of a reasonable deterministic variant is actually an advantage of CMAs over DAs. The data and combined complexity of model checking become polynomial (5.1(a)). Even though deterministic CMAs can express regular global and local properties, they are considerably weaker than CMAs, as they capture neither register automata (4.2) nor two-variable logics (4.3). The attempt to augment the expressive power of deterministic CMAs by allowing them to operate in a two-way fashion is unsuccessful, as it results in undecidability (5.3).

We also investigate closure properties of $\mathcal{R}$. It is closed under union, intersection, product, and concatenation, but neither under complementation nor under Kleene star. The former follows already from the undecidability of universality for register automata [15], the latter is Proposition 6.1. Deterministic CMAs are closed under intersection but not under union, concatenation, or complementation. To obtain a deterministic model closed under Boolean operations, we introduce acceptance conditions which combine conditions for global properties with Presburger conditions on the numbers of data values fulfilling certain local properties (6.4). Despite its closure under negation this model is still decidable as even non-deterministic such CMAs can be effectively translated into CMAs without Presburger conditions. Non-deterministic CMAs with these conditions are, however, still not closed under complementation.

Since $\mathcal{R}$ is still unable to handle some natural properties arising in model checking, we investigate how much the expressive power of CMAs can be extended while preserving decidability. More precisely, we consider two such extensions which allow more interaction between the global and the local properties: a model with a synchronization mechanism and one with the ability to "reset" information seen for a data value. Returning to our printer example, these automata

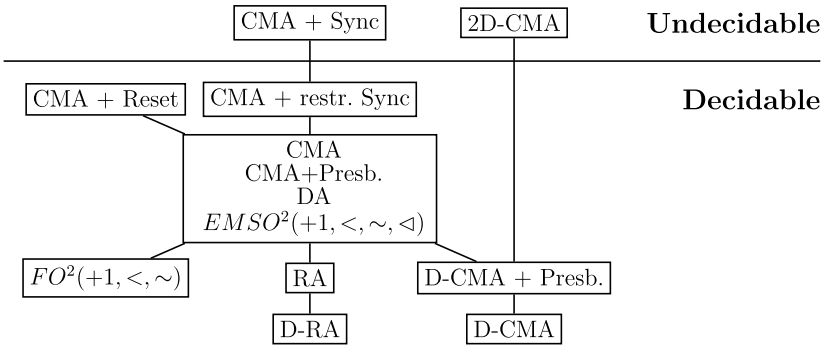

Fig. 1. A schematic picture of inclusions among classes of data languages. The lowermost three (branches of) classes are pairwise incomparable.

can handle, e.g., restarts of the system, where the content of the printer queue is lost. An overview of the classes under consideration is given by Fig. 1.

2. Preliminaries

Data words. Let Σ be a finite alphabet and Δ an infinite set. A **data word** is a finite sequence over $\Sigma \times \Delta$. A **data language** is a set of such words. If $w = (a_1, d_1) \dots (a_n, d_n)$, then $\mathrm{str}(w) = a_1 \dots a_n$ is the **string projection** of w. The *marked string projection* $\mathrm{mstr}(w)$ is the string $(a_1, b_1) \cdots (a_n, b_n)$ over $\Sigma \times \{0, 1\}$ for which $b_i = 1$ iff $d_i = d_{i-1}$ ($b_1 = 0$). For each data value d, the set of all positions with value d is called a **class** of w. The string induced by these positions is called a **class string**. A position j is called the **class successor** of a position i (denoted by $i \lhd j$) if $i < j$, both have the same data value, and there is no other position with the same value between i and j. Unless otherwise stated, data values can only be compared with respect to equality.

In the sequel, we will assume w.l.o.g. that all data languages and automata that we investigate are defined over the same data set Δ, which contains all data values used in examples and proofs. In particular $\mathbb{N} \subseteq \Delta$. We will also talk about data languages over Σ, where Σ is a finite alphabet, implicitly assuming that the data set is Δ.

Register automata. Register automata were introduced by Kaminski and Francez [12] and were later studied in, e.g., [15,5]. They were defined for sequences of data values only, but the generalization to data words is straightforward. Register automata are equipped with a constant number of registers in which they can store data values, which can later be compared with the data value of the current position. We extend the notion of [12].

Definition 2.1 ([12,15]). A **register automaton (RA)** over finite alphabet Σ is a tuple $R = (Q, q_0, F, k, P)$, where Q is a finite set of states, q_0 is the initial state, F are the accepting states, k is the number of registers, and P is a finite set of transitions. A transition is either a **write transition** of the form $(i, p, a) \to q$ or a **read transition** of the form $(p, a) \to (q, i)$, for $i \in \{1, \dots, k\}$, $p, q \in Q$, and $a \in \Sigma$. A **configuration** of R is a pair (q, τ), where $q \in Q$ and $\tau : \{1, \dots, k\} \to \Delta \cup \{\bot\}$ is a **register assignment** ($\bot \notin \Delta$ indicates an empty assignment). The initial configuration is (q_0, τ_0), where $\tau_0(i) = \bot$ for all $i \in \{1, \dots, k\}$.

A read transition $(i, p, a) \to q$ can be applied if the current state is p, the next input symbol is a and the next input data value is already stored in register i. It takes the automaton from configuration (p, τ) to (q, τ). A write transition $(p, a) \to (q, i)$ can be applied if the current state is p, the next input symbol is a and the next input data value d is currently not stored in any register. It takes the automaton from a configuration (p, τ) to (q, τ'), where $\tau'(i) = d$, and $\tau'(j) = \tau(j)$ for all $j \neq i$. R is **deterministic** if for each state p and letter a there is exactly one transition $(p, a) \to (q, i)$, and for each register i at most one transition $(i, p, a) \to q$. A **run** on a data string w is a sequence $(q_0, \tau_0), \dots, (q_n, \tau_n)$ of configurations, defined in the obvious way. The set of data words accepted by R is denoted as $L(R)$.

It should be noted that other definitions of RAs allow a non-empty initial assignment. This makes it possible to consider a finite number of constants, an ability that isn't needed when we have a finite as well as an infinite alphabet.

The definition ensures that a data value can never occur in more than one register at the same time. In particular, this feature can be used to verify that the current data value is different from those in the registers.

In [12], languages recognized by register automata are called **quasi-regular**. We mention some of the results from [12]: The class of quasi-regular languages is closed under union, intersection, concatenation, and Kleene star, but not under complementation. The emptiness problem for register automata is decidable. If R_1 and R_2 are register automata, and R_2 has at most two registers, then it is decidable whether $L(R_1) \subseteq L(R_2)$. In [15], different versions of register automata are investigated (two-way, alternating, etc.). In particular, it is shown that deterministic RAs are strictly weaker than RAs, which are in turn strictly weaker than two-way RAs, and that RAs are strictly weaker than *MSO**.

As the following example shows, the ability of register automata to combine global and local properties is severely limited.

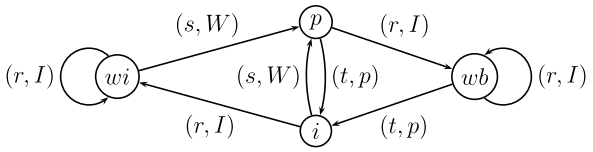

Fig. 2. A CMA for L_0. The labels are explained in Example 3.2.

Example 2.2. We take up the printer example from the introduction. Let L_0 be the set of **valid traces**, i.e., the data words whose string projection matches the expression $(r^*sr^*tr^*)^*$ and for which each class string satisfies $(rst)^*$. We claim that there is no register automaton for L_0. For a contradiction, assume that register automaton R accepts L_0. Let k be the number of registers of R. Let w be the data string $(r, 1)\cdots(r, k+1)(s, 1)(t, 1)\cdots(s, k+1)(t, k+1)$. As $w \in L_0$, R has an accepting run ρ on w. After reading the first $k+1$ positions of w, there is at least one data value $d \in \{1, \ldots, k+1\}$ that does not occur in any register of R. We can conclude that R also accepts the string $w' \notin L_0$ resulting from w by replacing (s, d), (t, d) with $(s, k+2)$, $(t, k+2)$.

3. Data and class memory automata

As seen in the previous section, register automata only have a limited ability to check local properties, e.g., in general they cannot check whether the class strings of a data word belong to some given regular language. With an automata model in mind that can check regular local and global properties it is natural to consider a combination of a finite state automaton reading the (string projection of the) whole input word and an automaton reading the different class strings.

In this section, we introduce the *class memory automaton (CMA)*, an automaton model which basically combines a global and a local automaton and allows some interaction between them. We will show then that class memory automata have exactly the same expressive power as the *data automata* that were introduced in [3] mainly as a tool for use in a decidability proof.

CMAs have two advantages over data automata: they are conceptually slightly simpler and they have a meaningful notion of determinism.

Definition 3.1. A **class memory automaton** C is a tuple $(Q, \Sigma, \delta, q_I, F_L, F_G)$, where Q is a finite set of states, Σ is a finite alphabet, q_I is the initial state,

- $\delta : (Q \times \Sigma \times (Q \cup \{\bot\})) \to \mathcal{P}(Q)$ is a **transition function**; and
- $F_G \subseteq F_L \subseteq Q$ are the sets of **globally and locally accepting states**, respectively.

The semantics of class memory automata (CMAs) is defined through the notion of class memory functions. Such a function simply assigns to every data value d the state of the automaton that was assumed after reading the last (previous) position with value d. More formally, a **class memory function** is a function $f : \Delta \to Q \cup \{\bot\}$ such that $f(d) \neq \bot$ for only finitely many d. A **configuration** of C is a pair (q, f) where $q \in Q$ and f is a class memory function. We call q the **global state** of C and $f(d)$ the **local state** of d. The initial configuration of A is (q_I, f_I), where $f_I(d) = \bot$ for all $d \in \Delta$. When reading a pair $(a, d) \in \Sigma \times \Delta$, the automaton can go from configuration (q, f) to (q', f') if (1) $q' \in \delta(q, a, f(d))$, (2) $f'(d) = q'$, and (3) for all $d' \neq d, f'(d') = f(d')$. The automaton accepts if, for the final configuration (q, f), $q \in F_G$ and $f(d) \in F_L \cup \{\bot\}$, for all $d \in \Delta$. A CMA is **deterministic** if each $\delta(p, a, q)$ is a singleton.

It should be noted that δ naturally induces a *transition relation* which is a subset of $(Q \times \Sigma \times (Q \cup \{\bot\})) \times Q$. We freely switch back and forth between these two points of view.

Example 3.2. We construct a CMA C that accepts the language L_0 from Example 2.2. The automaton C, depicted in Fig. 2, has four states:

- p (the printer is printing for the current process),
- i (the current process is neither printing nor waiting for a print),
- wi (the current process is waiting for a print and the printer is idle)
- wb (the current process is waiting for a print but the printer is busy)

Edge labels (σ, p) indicate that the transition can be taken reading symbol σ if the class memory is p. Here W abbreviates wi or wb and I stands for i or $\bot$.

To get a better understanding of the automaton let us have a look at the transitions leaving p. In state p the automaton just read some (s, d) reflecting the start of a print for the process number d. Thus, there are only two possible kinds of next data symbols: either (t, d) which ends the print of process d or (r, d') which moves process $d' \neq d$ into the waiting state. It should be noted that no (s, d') could be read next.

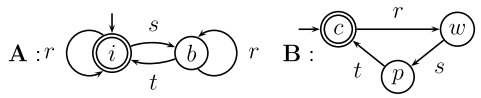

Fig. 3. A data automaton for the language L_0. A has states i (idle) and b (busy). B has states c (computing), w (waiting) and p (printing).

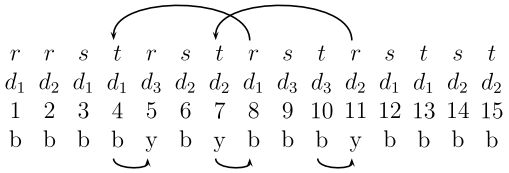

Fig. 4. A data string w with its graph $G(w)$ and the induced coloring.

We next show that CMAs are not "yet another automata model for data strings". Indeed, as mentioned before, they are equivalent to the automata model of [3] which in turn has a robust characterization by existential monadic second-order formulas with two first-order variables.

Data automata implement the idea of combining a global with a local automaton in a different way: the global automaton is a string transducer; the local automaton reads the class strings induced by its output.

Definition 3.3 (*[3]*). A **data automaton (DA)** D is a pair (A, B), where A is a non-deterministic letter-to-letter transducer (the **base automaton**) with a finite **output alphabet** Γ and B an NFA (the **class automaton**). A data word $w = w_1 \dots w_n$ over Σ is accepted by D if there is an accepting run of A on the marked string projection $\mathrm{mstr}(w)$, yielding an output string $g_1 \dots g_n$, such that B accepts each string $g_{i_1} \dots g_{i_k}$ induced by a class of w.

Example 3.4. We construct a data automaton $D = (A, B)$ for the language L_0 from Examples 2.2 and 3.4 (cf. Fig. 3). The transducer A makes sure that the string projection matches $(r^*sr^*tr^*)^*$, and copies its input to the class strings. The class automaton B verifies that each class string matches $(rst)^*$.

We aim to show next that data automata and CMAs are expressively equivalent. As a technical preparation for that proof we first show that it is not necessary that a DA A reads the marked string projection $\mathrm{mstr}(w)$ (indicating where the data value changes): we define **unmarked data automata** in the same way as data automata but the base automaton reads $\mathrm{str}(w)$ instead of $\mathrm{mstr}(w)$ and show that the expressive power of unmarked data automata is the same as that of data automata. It should be stressed that thanks to Proposition 3.6 we could define CMAs to read $\mathrm{str}(w)$ as opposed to $\mathrm{mstr}(w)$.

We illustrate the coloring technique used in the proof of this result with an example.

Example 3.5. We consider the language L_1 of traces in which the pattern $(t, d)(r, d)$ does not occur, i.e., after a print job of a process terminates it can request the next one only after some other event has occurred. We note that a DA whose base automaton reads $\mathrm{mstr}(w)$ can easily accept this language by simply avoiding the pattern $(t, b)(r, 1)$, for $b \in \{0, 1\}$. It is, however, less obvious how a data automaton can do this if its base automaton only sees $\mathrm{str}(w)$. Intuitively, the class automaton has no clue whether between a t and the subsequent r of some class some other event occurred. The base automaton, on the other hand, does not know anything about data values whatsoever. Nevertheless, by working together, the base and class automaton can accept L_1 by using the coloring technique explained next. The idea is that the base automaton guesses a color (black or yellow), for each t-position and each r-position, such that the following two conditions hold.

(1) Every r-position shares the color of the previous t-position in the same class (if it exists).
(2) If an r-position immediately follows a t-position they have different colors.

Obviously, if colors can be assigned such that (1) and (2) hold, then $w \in L_1$. Furthermore, condition (1) can be checked by the class automaton, condition (2) by the base automaton.

It remains to show that for each $w \in L_1$ a coloring fulfilling (1) and (2) can be found. To this end, we associate with w a directed graph $G(w)$ whose vertices are the positions of w carrying t or r and which has an edge from i to j if

(i) $j < i, j$ carries t and i is the next r-position of w in the class of j, or
(ii) $j = i + 1$, i carries t and j carries r.

The intuitive meaning of an edge (i, j) is that the color of i determines the color of j. Observe that each node in $G(w)$ has in-degree at most 1. Furthermore, there are no cycles. Thus, we can assign colors as follows: (1) Each node of in-degree 0 gets the color black. (2) Whenever there is an edge (i, j) and i is a t-position which is already colored then j gets a color different than i. (3) Whenever there is an edge (i, j) and i is an r-position which is already colored then j gets the same color as i. Clearly, this leads to a coloring respecting conditions (1) and (2). Fig. 4 gives an illustration.

Proposition 3.6. *For every data automaton, there is an equivalent unmarked data automaton.*

Proof. Let $D = (A, B)$ be a data automaton. We construct an equivalent data automaton $D' = (A', B')$ with the claimed restriction. Intuitively, on input of $\mathrm{str}(w)$, A' guesses a marked string v and simulates the behavior of A on v. Simultaneously, together with B', it checks that $v = \mathrm{mstr}(w)$ using the coloring technique of Example 3.5.

Thus, first of all, A' guesses, for each position i, whether it has the same data value as position $i-1$. Furthermore, for each i, it guesses a subset S_i of the set $S = \{\mathrm{open(black)}, \mathrm{open(yellow)}, \mathrm{close(black)}, \mathrm{close(yellow)}\}$. More precisely, S_i contains exactly one opening element, open(black) or open(yellow), if position i is (guessed to be) in a different class than $i-1$. Likewise, S_i contains either close(black) or close(yellow) if position i is in a different class than $i+1$. At position i, the output of A' is just (g_i, S_i), where g_i is the output of A when reading position i of $\mathrm{mstr}(w)$. The automaton (A', B') has to make sure that the positions with a data value change are guessed correctly, i.e., that the open-position after each close-position has a different data value. To this end, (A', B') accepts if the following two conditions hold, for every position i.

(1) If $\mathrm{open}(c) \in S_i$, for some color c then the previous position $j < i$ with the same data value fulfills $\mathrm{close}(c) \in S_j$ (if such a position exists).
(2) If $i > 1$ and $\mathrm{open}(c) \in S_i$, for some color c, then $\mathrm{close}(c') \in S_{i-1}$, where $c' \neq c$.

Again, (1) can be checked by the class automaton B' and (2) by the base automaton A'. Clearly, if (1) and (2) are satisfied, A' guessed the marked string correctly and thus (A', B') accept if and only if (A, B) would have accepted. On the other hand, if the marking is guessed, there is a coloring fulfilling (1) and (2) which is shown in complete analogy to Example 3.5. □

Now we turn to the main result of this section.

Proposition 3.7. *DAs and CMAs are expressively equivalent.*

Proof (*Sketch*). We show that, for every unmarked DA, there is an equivalent CMA. The simulation of an unmarked DA $D = (A, B)$ by a CMA C is just a generalization of Example 3.2. The states of C simply combine the states of A and B; the state set Q_C of C is $Q_A \times Q_B$. A configuration $((q, q'), f)$ of C thus represents the current state q of A and, for each data value d, the current state of B is just the second component of $f(d)$. More precisely: if A has a transition from p to p' that reads an a and outputs a t, and B has a transition from q to q' that reads a t, then C has a transition $((p, x), a, (y, q), (p', q'))$ for each state x of A and y of B. If q is an initial state of B, then C also has a transition $((p, x), a, \bot, (p', q'))$ for each state x of A. The locally accepting states F_L of C are those whose second components are accepting in B, and the globally accepting states F_G are those where the first component is also accepting in A.

We next show how a CMA C can be simulated by a DA $D = (A, B)$. The idea is simple: for each position with a data value d, A guesses $f(d)$, and simulates C on the basis of this guess. B checks that the guesses were correct. More precisely, A has the same state set Q as C. If C has a transition (p, a, q, p') then A can go from state p to p', reading a and outputting (q, p') (where $q = \bot$ is also allowed). B checks, for each data value, that the guesses of A are consistent. For this purpose, B has state set $Q \cup \{q_I, q_-\}$, where the locally accepting states of C are accepting in B. From the initial state q_I it enters state p if it reads a symbol of the form $(\bot, p)$ and q_-, otherwise. Likewise, from a state p it enters p' when it reads a symbol (p, p') and q_-, otherwise. It is straightforward to see that D accepts the same language as C. □

Note that in both directions, the translation can be computed in polynomial time.

4. Expressiveness

In this section, we compare the expressive power of CMAs with that of RAs. The main result is that CMAs are strictly stronger than register automata. Remarkably, this result does not carry over to the deterministic counterparts.

Theorem 4.1. *CMAs are strictly more expressive than RAs.*

Proof. The set L_0 of valid traces is recognized not by any RA (Example 2.2) but by a CMA (Example 3.2). It remains to show that for every RA, we can construct an equivalent CMA. Let $R = (Q, q_0, F, k, P)$ be a fixed RA with k registers. Without loss of generality, we can assume that each state q determines whether it is reached by a read or a write transition. Let $\rho = (q_0, \tau_0), \ldots, (q_n, \tau_n)$ be a run of R on input $w = (a_1, d_1) \cdots (a_n, d_n)$. Note that, after each step i, d_i is stored in some register of R, i.e., $\tau_i(j) = d_i$, for some j. We say that a transition using register j *closes* the register if either it is the last transition involving j in the run or there is no transition reading from j before the next write to j.

Intuitively, the CMA C guesses, for each transition, whether it closes the register. To ensure that the guesses are correct, C makes use of the coloring technique that was already used in Example 3.5 and Proposition 3.7. More precisely, the states of C are of the form (q, l, S, p), where $q \in Q$, $l \in \{1, \ldots, k\}$ and S is a subset of $\{\mathrm{open(black)}, \mathrm{open(yellow)}, \mathrm{close(black)}, \mathrm{close(yellow)}\}$ and p stores some more information to be specified below. Intuitively, it corresponds to a configuration of R with state q, in which the last transition affected register l, and in which this transition was a write iff $\mathrm{open}(b) \in S$ for some b and it closed the register iff $\mathrm{close}(c) \in S$ for some c. We show that C can be constructed such that the following holds.
Claim. C has a run $\rho = (q_0, l_0, S_0, p_0), \ldots, (q_n, l_n, S_n, p_n)$ on w, fulfilling conditions (1)–(3) below if and only if R has an accepting run on w, where we call a position *opening* if $\mathrm{open}(b) \in S_i$ for some b and *closing* if $\mathrm{close}(c) \in S_i$ for some c.

(1) The transitions of C are consistent with the transition relation of R, i.e., for each i, $0 < i \leq n$, R has a read transition $(l_i, q_{i-1}, a_i) \to q_i$ or a write transition $(q_{i-1}, a_i) \to (q_i, l_i)$, and, if i is opening, then the latter applies.

(2) For each position i, there is an opening position $j \leq i$ and a closing position $j' \geq i$ with (a) $l_i = l_j = l_{j'}$, (b) $d_i = d_j = d_{j'}$, and (c) for all positions $m, j < m < j'$ it holds that either $l_m \neq l_i$ and $d_m \neq d_i$ or $l_m = l_i$, $d_m = d_i$ and $S_m = \emptyset$.
(3) If open$(b) \in S_i$, for some b, then either there is no $j < i$ with $d_j = d_i$, or the following two conditions hold.
   (a) For the largest position $j < i$ with $d_j = d_i$, close$(b) \in S_j$.
   (b) If the largest position $m < i$ with $l_m = l_j$ is closing then close$(c) \in S_m$, for $c \neq b$.

We first note how these conditions can be ensured by C: (1) is straightforward. (2) can be checked by the local states. For each data value, the sequence of opening, closing and other positions must be OK. Condition (3a) can also be checked by using local states whereas (3b) uses the global state. The necessary information is stored in the p-component of the states of C.

It remains to prove the above claim. Let us first assume that there is a run ρ of C fulfilling conditions (1)–(3). We have to show that there is an accepting run ρ' of R. We construct ρ' inductively and show by induction that, for each i, the prefix of ρ' of i steps is consistent and leads to a configuration (q_i, τ_i), where for each $l \leq k$ one of the following conditions holds.

- $\tau_i(l) = \tau_0(l)$ and $l_j \neq l$, for every $j \leq i$.
- $\tau_i(l) = d_j$, for the maximal $j \leq i$ with $l_j = l$.

For $i = 0$, the conditions clearly hold. Assume now that the i-th transition of C corresponds to the read transition $(l_i, q_{i-1}, a_i) \to q_i$. By (1), i is not an opening position. Thus, (2) guarantees that the maximum $j < i$ with $l_j = l_i$ fulfills $d_i = d_j$. Therefore, $\tau_{i-1}(l_i) = d_i$, the read transition can be applied by R and the induction statement holds for i. Otherwise, the i-th transition corresponds to the write transition $(q_{i-1}, a_i) \to (q_i, l_i)$. By (1), i is an opening position and if d_i has occurred before its last occurrence was at a closing position. We have to make sure that $d_i \neq \tau_{i-1}(l_i)$, or equivalently that for the maximal position $j < i$ with $l_j = l_i$ it holds that $d_j \neq d_i$. Let us assume otherwise. Condition (1) implies that j is a closing position. Let b be its corresponding color. By (3b), there must be a position $m, j < m < i$, with $l_m = l_i$ and thus $\tau_{i-1}(l_j) \neq d_j = d_i$. As l_j is the register in which d_i was stored last time, by induction d_i is not stored in any other register either.

By an argument similar to that in Example 3.5 and Proposition 3.7 it can be shown that a run of C fulfilling (1)–(3) exists if R has an accepting run. It is sufficient to notice that, in condition (3), the closing color of m determines the opening color of i which in turn determines the closing color of $j < m$. Thus, in the underlying graph each node has in-degree at most 1. □

In the next section, we show that model checking for CMAs is expensive, due to non-determinism. Thus, it is natural to consider deterministic CMAs (D-CMAs) which turn out to be quite expressive but less powerful than CMAs.

Proposition 4.2. *Deterministic RAs and D-CMAs are expressively incomparable.*

Proof. The language L_0 of valid traces can be recognized by a deterministic CMA (Example 3.2), but not by any RA (Example 2.2). For the opposite direction, we show that L_1 defined in Example 3.5 cannot be accepted by a deterministic CMA. Note that L_1 can be easily checked by a deterministic RA. Consider data words $w_n = (t, 1)(t, 2) \dots (t, n)$, for $n \in \mathbb{N}$. For any deterministic CMA C, if n is large enough, there are $i < j$ such that the state of C after reading position i of w_n is the same as after reading position j. Thus, for the configuration (q, f) after position j we have $f(i) = f(j) = q$. We conclude that $w_j \cdot (r, i)$ is accepted by C if and only if $w_j \cdot (r, j)$ is accepted. But $w_j \cdot (r, i)$ is in L_1 while $w_j \cdot (r, j)$ is not. □

The above proof also shows that a statement corresponding to Proposition 3.6 does *not* hold for deterministic CMAs.

As discussed in the introduction, data languages can also be described in terms of logic. In particular, when using fragments of first-order logic, we let variables range over positions in words, and use predicates to talk about the labels, data values, and order of positions. It is shown in [3] that emptiness for data automata is decidable, and that they capture $FO^2(+1, <, \sim)$, that is, the two-variable fragment of FO with the usual string predicates $+1$ and $<$, and the $\sim$-predicate, which is true for two positions in the same class. Actually, marked data automata are shown to be expressively equivalent to $EMSO^2(+1, <, \sim, \lhd)$, i.e., (the two-variable fragment of) existential monadic second-order logic with the class successor $\lhd$ as additional predicate. By Propositions 3.6 and 3.7 and their constructive proofs, all these results carry over to unmarked data automata and CMAs. As L_1 is defined by the $FO^2(+1, \sim)$-formula $\forall x \forall y\, (x + 1 = y \wedge t(x) \wedge r(y)) \to x \not\sim y$ we can conclude the following.

Proposition 4.3. *D-CMAs cannot express all $FO^2(+1, \sim)$-definable properties.*

5. Algorithmic properties

The *model checking problem* for automata asks whether a data word w is in the language $L(A)$, for an automaton A. If A is fixed, we refer to the complexity of the problem as *data complexity*. If A is considered as part of the input we speak about *combined complexity*.

Proposition 5.1. (a) *For D-CMAs and deterministic RAs, data and combined complexity are polynomial.*
(b) *The data complexity (and the combined complexity) of model checking for CMAs is NP-complete.*

Proof. In (a), every new input pair (a, d) uniquely defines the next transition. Thus the unique run on a word can be constructed in polynomial time. For (b), NP-membership is easy, since a run of the automaton on a word can be guessed. We show NP-hardness by a reduction from 3-CNF-SAT. An instance $\phi = \phi_1 \wedge \cdots \wedge \phi_m$ of 3-CNF-SAT can be encoded as a data word of length $3m$ in the following way. We associate with each variable some data value, in a pairwise distinct fashion. Each clause is encoded by a data word over $\{+, -\}^3$. For example, if x_1 appears negatively in the first clause and d is its associated data value then the first sub-word contains $(-, d)$.

Now we construct a CMA C. The basic idea is that C will guess the truth values of the variables, and verify that it has guessed correctly. Each state of C is a tuple (v, p, c), where v is a *variable truth value* from $\{0, 1\}$, p is a *position* from $\{0, 1, 2\}$, indicating how many positions in the current clause C has read, and c is a *clause truth value* from $\{0, 1\}$, indicating whether the current clause has yet been proved to be true (according to the guessing of the truth values). The initial state is $(0, 0, 0)$. Let $((v, p, c), f)$ be a configuration of C and let (a, d) be the next input pair. We will describe the possible next configurations (v', p', c') of C.

- If $f(d) = \perp$ then $v' \in \{0, 1\}$ (C guesses a truth value for the variable of d).
- If $f(d) = (v'', p'', c'') \neq \perp$ then $v' = v''$ (the automaton does not change the truth value of a variable).
- If $p = i$ then $p' = i + 1 \bmod 3$.
- If $c = 0$ the next transition on input (a, d) depends on a and v''. If $a = +$ and $v'' = 1$ or if $a = -$ and $v'' = 0$ then $c' = 1$, unless $p = 2$, in which case $c' = 0$. Otherwise, $c' = 0$ or, if $p = 2$, there is no further transition: the truth assignment did not satisfy the last clause and thus C rejects.
- If $c = 1$ and $p < 2$ then $c' = 1$. If $c = 1$ and $p = 2$ then $c' = 0$ (indicating that the new clause has not been satisfied yet).

All states with $p = 0$ are accepting. Obviously, C accepts if and only if it guessed a satisfying assignment for ϕ, which implies the statement of the proposition. □

For RAs, the data and combined complexity are (probably) different.

Proposition 5.2. (a) *The data complexity of model checking for RAs is polynomial.*
(b) *The combined complexity of model checking for RAs, parameterized by the number of registers, is $W[1]$-hard.*[2]

Proof (*Sketch*). For (a), consider an RA R with k registers. The number of possible configurations of R with input w is bounded by $|Q| \cdot \binom{|w|}{k} \cdot k!$. Thus, one can check that $w \in L(R)$ in polynomial time by inductively computing the set of reachable configurations, for each position of w.

The proof of (b) is by reduction from the parameterized k-clique one. In this problem, we are given a Graph $G = (V, E)$ and a positive integer k (the parameter). The question is whether G has a clique of size k. The problem is known to be $W[1]$-complete [7]. Given G and $k \in \mathbb{N}$, we construct a word w over alphabet $\{a, b, c\}$ with data values from V and an RA R with $k + 1$ registers as follows. The word w consists of two parts. The first part is just $(a, v_1) \ldots (a, v_n)$, where $v_1, \ldots, v_n$ are the vertices in V. The second part is a concatenation of all strings $(b, u)(c, v)$ such that $u, v \in V$ but $(u, v) \notin E$. When R reads the first part, it non-deterministically selects k vertices that are stored in the first k registers. When reading the second part, it immediately rejects if it reads $(b, u)(c, v)$, for which u and v are stored in a register. If it reaches the end of w without reading such a pair, it accepts. □

No parameterized upper bound for this problem is yet known, except that it belongs to XP (as can be seen from the proof of Proposition 5.2(a)).

Two-way deterministic CMAs. Since deterministic CMAs are clearly weaker than general CMAs, it is natural to ask whether we can allow them to move both ways. A two-way CMA is a CMA in which the head of the automaton can move to the right or to the left in one step. To this end, the input string w is padded by a symbol $\triangleright$ to the left and a symbol $\triangleleft$ to the right to enable the automaton to recognize the ends of the strings. The class memory function is generalized in a straightforward way: $f(d)$ is just the state taken after reading some position (σ, d) the last time (and $\perp$ if such a position has not yet been visited). Transitions depend on the current state and the class memory function, just as for one-way CMAs. We omit a formal definition.

Unfortunately, the two-way extension does not preserve decidability, as we show next.

Theorem 5.3. *Emptiness for two-way deterministic CMAs is undecidable.*

Proof. We use a reduction from Post's Correspondence Problem (PCP) which is well-known to be undecidable [11]. An *instance* I of PCP is a sequence $(x_1, y_1), \ldots, (x_n, y_n)$ of pairs, where $x_i, y_i \in \{a, b\}^*$ for $i = 1, \ldots, n$. This instance has a *solution* if there exist $m \in \mathbb{N}$ and $i_1, \ldots, i_m \in \{1, \ldots, n\}$ such that $x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m}$. Given an instance $I = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ of PCP, we construct a two-way deterministic CMA C whose language is non-empty if and only if I has a solution.

We encode solution candidates as data words w over $\Sigma = \{a, b, \#\} \cup \{1, \ldots, n\}$. If $i_1, \ldots, i_m$ is a solution, $\mathrm{str}(w)$ will be $i_1 x_{i_1} \cdots i_m x_{i_m} \# i_1 y_{i_1} \cdots i_m y_{i_m}$. We refer to positions with a number as *index positions* and to the others (besides #) as *letter positions*. Each data value (besides the one for #) will appear exactly twice. For $1 \leq j \leq m$, the two occurrences of i_j get the

[2] For an introduction to fixed-parameter complexity and $W[1]$, see, e.g., [8,6,10].

same data values. Furthermore, two letters from $\{a, b\}$ in the first and the second half of w get the same data value if they represent the same position in $x_{i_1} \dots x_{i_m} = y_{i_1} \dots y_{i_m}$.

The automaton C intended to accept all data strings $u\#v$ encoding correct solutions to I works in four stages, each of which uses its own state space. To this end, let $u = (a_1, d_1) \cdots (a_n, d_n)$ and $v = (a'_1, d'_1) \cdots (a'_m, d'_m)$

In the first stage, the automaton checks that the word following an index character i_j really is x_{i_j} (y_{i_j}). For this it uses state set Q_1 and works as a DFA over words without data, ignoring the class memory function. When this is done, it returns to the first position of the word. After the first stage, we know that for each data value, the class memory function assigns a state in Q_1.

In the second stage, C checks that each data value appears exactly twice, once in u and once in v, and that it appears together with the same label in u and v. This is done in two passes. The first pass from left to right checks that each data value appearing in v also appears, with the same label, in u. The second pass checks the dual property with u and v switched. In the first pass, C uses one state p_a, for each $a \in \Sigma$. Inside u, it checks, for each position i, that $f(d_i) \in Q_1$ and enters state p_{a_i}. Inside v, it checks, for each position j, that $f(d_j) = p_{a_j}$. Here, f always refers to the current class memory function. The second pass is done in an analogous fashion.

After stage 2, C can make use of the fact that each data value occurs exactly once in u and v (besides the value of #). In particular it can *remember* data values as follows: If C uses a certain state p exactly once in u at a position i then it can move into v and identify the unique position i' with $d_{i'} = d_i$ by simply searching for the first position i' with $f(d_{i'}) = p$. Next, C can move to the left, "mark" $d_{i'}$ by another state q and find its way back to position i in u.

In stage 3, this technique is used to check that the sequence of data values occurring at the index positions in u is the same as in v. To this end, C goes to the first index position i of u and marks it with a state p. Then it moves to the smallest index position $j > i$ of u and marks it with a state q. Next it moves to the right until it reaches the position i' corresponding to i and verifies that it does not see the corresponding position j' of j on its way, thus verifying that $j' > i'$ as required. Next it moves back to position i (basically to "erase the memory"), and continues with the second and third index positions of u as i and j and so on.

In the last stage, C checks that the data value sequences induced by the letter positions are also identical in u and v. Clearly, C accepts a string $u\#v$ after these four stages if and only if it encodes a PCP solution. □

6. Closure properties

For the automata-theoretic approach to static analysis and verification, closure properties are of great importance, since they facilitate modular reasoning. In this section, we consider the closure properties of the classes of languages defined by non-deterministic and deterministic CMAs. We first prove a negative result; the class $\mathcal{R}$ of languages recognized by CMAs is not closed under Kleene star. We then show that by extending deterministic CMAs with Presburger conditions, we obtain a decidable model closed under Boolean operations.

Proposition 6.1. *The class $\mathcal{R}$ of languages accepted by CMAs is effectively closed under intersection, union and concatenation. It is* not *closed under complementation and Kleene star.*

Proof. Closure under union and intersection for data automata was shown in [3]. To prove closure under concatenation, let $C_1 = (Q^1, \Sigma, \delta^1, q_I^1, F_L^1, F_G^1)$ and $C_2 = (Q^2, \Sigma, \delta^2, q_I^2, F_L^2, F_G^2)$ be CMAs. An automaton C for $L(C_1) \cdot L(C_2)$ is constructed as follows. It starts by simulating C_1 and at some point, if the current state is in F_G^1, it non-deterministically switches to C_2. To this end, Q^1 and Q^2 have to be disjoint. During the second phase, C rejects whenever it encounters an input (a, d) such that $f(d) \in Q^1 - F_L^1$. On the other hand, if $f(d) \in F_L^1$ it just behaves as C_2 for $f(d) = \perp$. The globally accepting set is F_G^2; the locally accepting set is $F_L^1 \cup F_L^2$.

We next show that $\mathcal{R}$ is not closed under Kleene star. The proof is a reduction from the halting problem for two-counter machines without input. Such automata have only ε-transitions between states, which perform counter operations. A counter operation can increase, decrease, or zero-check a counter. The counters can never have negative values. The machine halts if it reaches a final state with empty counters. The halting problem is known to be undecidable. We show that if $\mathcal{R}$ were closed under Kleene star, we could effectively reduce the halting problem for two-counter machines to the emptiness problem for CMAs. This is a contradiction, since emptiness for CMAs is decidable.

Each run of a two-counter machine induces a word over $\Gamma = \{c_1^+, c_1^-, c_1^0, c_2^+, c_2^-, c_2^0\}$ where c_i^+, c_i^-, c_i^0 represent incrementing, decrementing, and zero-checking counter i, respectively. With each counter automaton A we associate a language $L_A \subseteq \Gamma^*$ as follows. A string $w = w_1 \cdots w_n$ is in L_A if q_n is an accepting state of A and there is a sequence $q_0, \dots, q_n$ of states of A such that, for each $i > 0$, A has a transition consistent with q_{i-1}, w_i, q_i, i.e., A can go from q_{i-1} to q_i while performing operation w_i. Note that we ignore that a transition might be applicable only if the counters allow it. Clearly, for each A a DFA accepting L_A can be effectively constructed.

Let L_{c_1} be the *data language* with string projections of the form $(c_1^+ + c_1^- + c_2^+ + c_2^- + c_2^0)^* c_1^0 (c_2^0 + \varepsilon)$ such that each class string is either $c_1^+ c_1^-$ or does not contain c_1^+ or c_1^-. It is easy to see that L_{c_1} can be recognized by a CMA C_1. Symmetrically, we define the language L_{c_2} with string projections in $(c_2^+ + c_2^- + c_1^+ + c_1^- + c_1^0)^* c_2^0 (c_1^0 + \varepsilon)$ such that every class string is either $c_2^+ c_2^-$ or doesn't contain c_2^+ or c_2^-. L_{c_2} is recognized by a CMA C_2. Towards a contradiction, let us now assume that $\mathcal{R}$ is closed under Kleene star and that C'_1 and C'_2 are CMAs for $L_{c_1}^*$ and $L_{c_2}^*$, respectively. For each two-counter automaton A we

can then effectively construct a CMA C_A for $L^*_{c_1} \cap L^*_{c_2} \cap L'_A$, where L'_A is the set of data words whose string projection is in L_A. We claim that A has an accepting computation if and only if $L(C_A) \neq \emptyset$, yielding the desired contradiction.

It should be stressed that $A \mapsto C_A$ is indeed effective even though our assumption that $\mathcal{R}$ is closed under Kleene star does not yield C'_1 and C'_2. But from the assumption we can conclude that they exist and therefore an algorithm for computing $A \mapsto C_A$ also exists. It is crucial here that C'_1 and C'_2 *do not depend on A*.

Given an accepting run ρ of A, we can construct a data word w accepted by C_A as follows: the string projection of w is obtained from the transitions of ρ in a straightforward manner. Furthermore, each position with a symbol c_i^+ (corresponding to incrementing counter i from some m to $m+1$) gets the same data value as the position (carrying c_i^-) that corresponds to the subsequent decrementation of counter i from $m+1$ to m. This data value does not occur anywhere else. All other (zero-check) positions get distinct data values. Clearly, w is accepted by C_A. On the other hand, it is straightforward to check that each data string accepted by C_A induces an accepting run of A. The proof that $\mathcal{R}$ is not closed under Kleene star is thus completed.

The proof that $\mathcal{R}$ is not closed under complementation is very similar. First it shows that there is a CMA C for the complement of $L^*_{c_1} \cap L^*_{c_2}$. The construction of C is straightforward: it is basically the union of several automata, each checking one type of violated constraint. If $\mathcal{R}$ were closed under complementation we would obtain a CMA C' for $L^*_{c_1} \cap L^*_{c_2}$. Thus we could again effectively construct an automaton for $L^*_{c_1} \cap L^*_{c_2} \cap L'_A$ from a two-counter automaton A. □

Proposition 6.2. *The class of languages recognized by* deterministic *CMAs is effectively closed under intersection. It is* not *closed under union, complementation, concatenation, or Kleene star.*

Proof. For intersection, closure is shown by a straightforward product construction. For Kleene star, non-closure follows from the proof of Proposition 6.1, since the automata involved there can be made deterministic.

Union. Let L' be the set of all data words over $\{a, b\}$ such that all data values are different, and the last letter is a b. Let L'' be all data words over $\{a\}$ such that each data value appears exactly twice. It is easy to see that both L' and L'' can be recognized by deterministic CMAs. We show that no deterministic CMA can recognize $L = L' \cup L''$. Towards a contradiction, we assume that there is such a CMA C. Let n be the number of states of C, and consider a data word w of length $n+1$ over $\{a\}$ such that all data values are different. Then there exist two positions $i < j \leq n+1$ such that the configuration (q_j, f_j) of C after reading w up to position j has $f_j(d_i) = f_j(d_j)$. Let w^i and w^j be the prefixes of w up to position i and j, respectively. The configurations of C after reading w^i or w^j must both be such that the memory for each data value is a locally accepting state, but the global state is rejecting. This is because C should accept the words obtained from w^i and w^j by appending a b-position with a fresh data value at the end.

Now consider the word $w^i w^i$. This word belongs to L, so the configuration of C after reading it must be both locally and globally accepting. Since the global state and the memories for the data values appearing in w^i are identical after reading w^i or w^j, the transitions taken when reading w^i after w^j are exactly the same as when reading w^i after w^i. But this means that C would accept $w^j w^i$, which is a contradiction, since $w^j w^i \notin L$.

As the class is closed under intersection but not under union, it can clearly not be closed under complement.

Concatenation. Let L' be the set of all data words over alphabet $\{a, b\}$ that end with letter a. Let L'' be the language over $\{a, b\}$ such that all data values are different. We claim that no deterministic CMA recognizes the concatenation language $L = L' \cdot L''$. Assume there were such a CMA C with n states. Let, for each i, u_i be the word $(b, 1)(b, 2) \cdots (b, i)$ and let w_i be the word $w_i = u_{n+1}(a, 0)u_n(a, 0) \cdots u_i$. Let q_i denote the state of C after reading w_i, for each i. We claim that the states $q_1, \ldots, q_{n+1}$ are pairwise different, yielding the desired contradiction as C only has n states. To this end, let us assume $q_i = q_j$, $i < j$. As j does not occur in the suffix of w_i after the prefix w_j, we can conclude for the configuration (q, f) of C after reading w_i that $f(i) = f(j) = q_j$. Thus, C accepts $w_i \cdot (b, j)$ if and only if it accepts $w_i \cdot (b, i)$. But since $w_i \cdot (b, j) \in L$ and $w_i \cdot (b, i) \notin L$, our assumption $q_i = q_j$ was wrong. □

Presburger conditions.

As one of our motivations for the introduction of CMAs was that they have a natural notion of determinism, Proposition 6.2 is rather bad news. This motivates the following extension of CMAs by Presburger conditions: the extension does not enlarge the power of CMAs but it yields an extension of D-CMAs that is closed under Boolean operations.

Instead of just requiring that the memory states for all data values are locally accepting, we generalize the acceptance condition as follows. Suppose that CMA C has states $Q = \{q_1, \ldots, q_m\}$. Each computation ρ of C with final configuration (p, f) induces a function $g : Q \to \mathbb{N}$, where $g(q)$ is the number of data values d with $f(d) = q$. We consider atomic formulas of two kinds: (1) q, where $q \in Q$ and (2) $(q_1 + \cdots + q_k \bmod c) = c'$, where the q_i are from Q and c, c' are constant numbers. A configuration (p, f) fulfills q iff $p = q$. It fulfills $(q_1 + \cdots + q_k \bmod c) = c'$ iff $(g(q_1) + \cdots + g(q_k) \bmod c) = c'$. Any Boolean combination of such formulas is a **limited Presburger formula**. A **Presburger CMA** C is a CMA with a limited Presburger formula Φ. A run of C is accepting if its final configuration satisfies Φ.

Proposition 6.3. *For each Presburger CMA there is an equivalent CMA.*

Proof (*Sketch*). Let C be a Presburger CMA with formula Φ. We assume, w.l.o.g., that in Φ, negation occurs only before atomic formulas. There is a CMA for each formula q and $\neg q$. The global accepting set simply has to be set to $\{q\}$ or $Q - \{q\}$, respectively. The other kinds of atomic formulas can be handled by a modulo counter in a straightforward manner. For example, for a formula $(q_1 - q_2 \bmod 3) = 0$, a modulo counter is added to the global state of the automaton which is

incremented (modulo 3) whenever the automaton enters a state q_1 or leaves (locally) state q_2. The Boolean combinations can be taken care of by the globally accepting state. □

Proposition 6.4. *The class of languages accepted by deterministic Presburger CMAs is closed under Boolean operations.*

Proof. Let $A = (Q, \Sigma, \Delta, \delta, q_I, \Phi)$ and $B = (P, \Sigma, \Delta, \gamma, p_I, \Psi)$ be deterministic CMAs. To construct an automaton that accepts the complement of $L(A)$, we only need to negate Φ. For the intersection or the union of $L(A)$ and $L(B)$, we construct the product automaton $A \times B$ and use the conjunction (or the disjunction) of Φ and Ψ, where, e.g., in the atomic modulo formulas each q of A is replaced by the sum of those q' of $A \times B$ with q in their A-component. □

The following result completes Fig. 1.

Proposition 6.5. *For each deterministic Presburger CMA there is an equivalent two-way deterministic CMA.*

Proof (*Sketch*). We only sketch the proof idea. Let A be a deterministic Presburger CMA. The two-way deterministic CMA A' first simulates A. When A reaches the end of the input, A' traverses the string backwards and counts, for each state p of A, for how many classes the computation of A ended in p. Of course, A' cannot literally compute this number but it is able to compute it modulo N, the product of all numbers c occurring in some atomic formula of A's Presburger formula. When A' reaches the left end of the string it can decide whether A would have accepted. □

7. CMAs with synchronization and reset

The expressive power of CMAs is sufficient to handle a large number of properties relevant in parameterized verification. Still, there are many natural properties that cannot be expressed. In this section we investigate ways of strengthening the expressive power, while maintaining decidability of the emptiness problem. We begin with an example of a relatively simple verification property that cannot be expressed by CMAs.

Example 7.1. Consider a variation L_s of the language L_0 from Examples 2.2, 3.2 and 3.4 where we use an additional symbol n (network failure). When a network failure occurs, all printer jobs that have been requested but not yet started are lost and thus the requests have to be repeated. The network failure notifications are sent by a special network process. In other words, L_s is the set of data words w such that

(1) if a class string contains a symbol n, then it contains only n symbols (i.e., it matches n^*),
(2) each other class string of w matches $(rst + r)^*$,
(3) if $i \lhd j$ and both i and j carry label r, there is a position k with $i < k < j$ that has label n,
(4) if $i \lhd j$, i has label r, and j has label s, there is no position k with $i < k < j$ that has label n, and
(5) str(w) matches $((r + n)^*s(r + n)^*t(r + n)^*)^*$.

Proposition 7.2. *There is no CMA that recognizes the language L_s from Example 7.1.*

Proof. We consider a sublanguage L'_s of L_s from Example 7.1 in which no request ever gets handled. The string projections of words in L'_s match the expression $(r^*n)^*$. The data values are such that if two r-positions belong to the same class, then there is an n-position between them (a process cannot send a new request until a network error has occurred). It is clear that if no CMA accepts L'_s, then no CMA accepts L_s.

Suppose there were a CMA C with m states, that accepted L'_s. We define a consistency property that C must have. Let $w \in L'_s$ and let $i < j$ be r-positions of w, with data values d_i and d_j, and with no n-position between them (thus $d_i \neq d_j$). Let (q, f) be a configuration of C after reading w up to position i in an accepting run ρ on w. Then we must have $f(d_i) \neq f(d_j)$. Indeed, if $f(d_i) = f(d_j)$, the automaton could not tell the difference between d_i and d_j when reading the rest of w. Thus ρ would also be an accepting run on the word $w' \notin L'_s$ obtained from w by replacing all occurrences of d_i with d_j, and vice versa, in the suffix starting at position $i + 1$. For words in L'_s we use n_i to denote the ith position with label n and B_i for the ith *block*, that is, the positions between n_{i-1} and n_i (we interpret n_0 as the beginning of the word).

We now construct a word w such that no CMA can satisfy the above property when reading w. As data values we use natural numbers. In block B_i, all data values have a 1 in the ith bit of their binary representation. Suppose that we can construct w so that the following is fulfilled. For each $i < j$ there is a data value $m_{i,j}$ with ones in bits i and j and zeros in all bits between i and j such that $m_{i,j}$ appears in the first half of B_i and in the second half of B_j. Let ρ be an accepting run of C on w, and let Q_i, for each i, be the set of states used in ρ when reading the first half of B_i. Now suppose that the number of blocks is 2^m. Then there are $i < j$ with $Q_i = Q_j$. Since $m_{i,j}$ appears in the first half of B_i, and doesn't appear between B_i and B_j, the memory state for $m_{i,j}$ when C starts reading B_j will be some state $q_{i,j} \in Q_i$. When reading the first half of B_j, all states in $Q_j = Q_i$ are used. Thus, after reading half of B_j, some data value appearing there will have memory state $q_{i,j}$. But since $m_{i,j}$ is yet to appear in B_j, this violates the consistency property defined above.

It remains to show that we can construct w so that for each pair i, j, there is a data value $m_{i,j}$ with the above property. We use 2^m blocks, and data values with binary representations of length $2m \cdot 2^m$. Thus, for each $i \leq 2^m$ there are at least 2^{2m} data values with ones in bit i. For $m_{i,j}$ we can choose the value that has ones *only* in bits i and j. After placing $m_{i,j}$, for each pair i, j, in the first half of B_i and the second of B_j, we can pad the two halves of B_k, for each k, with unrelated data values (with ones in bit k), until each half has length exactly 2^m. □

As mentioned in the introduction, CMAs can combine global regular properties with local regular properties (of the class strings). The "communication" between the global and the local properties is limited: the global automaton can "send information" to a class d only when a symbol (σ, d) occurs in the input. In particular, it is not possible to broadcast a global event to all classes simultaneously.

It is exactly this limitation that prohibits CMAs from recognizing L_s from Example 7.1: all processes have to be *simultaneously* informed that print requests have been lost.

We now study a stronger class of automata. It is equipped with transitions that model a *synchronous* failure broadcast to all processes, and can therefore recognize L_s.

Definition 7.3. A ***CMA with synchronization*** is a CMA $C = (Q, \Sigma, \delta, q_I, g, F_L, F_G)$ equipped with a *synchronization function* $g : Q \to \mathcal{P}(Q \cup \{\bot\})$. Some of the transitions **apply** g. When such a transition is taken from a configuration the automaton first changes state and updates the memory function for the current data value as usual, assuming a configuration (q, f). Then, it updates the class memory function by setting $f(d)$ to some state in $g(f(d))$, unless $f(d) = \bot$.

Constructing a CMA with synchronization that recognizes L_s from Example 7.1 is fairly straightforward. Unfortunately, with the full power of this extension, we overstep the border of decidability.

Theorem 7.4. *Emptiness for CMAs with synchronization is undecidable.*

Proof. The proof resembles the undecidability proof from Proposition 6.1. In particular, it is a reduction from the halting problem for two-counter machines without input. We again use the symbols $\Gamma = \{c_1^+, c_1^-, c_1^0, c_2^+, c_2^-, c_2^0\}$ with the same intended meaning. We further use additional symbols $\#_1$ and $\#_2$: after each c_i^0 an arbitrary number of $\#_i$ symbols can occur. The proof can be more easily stated in terms of data automata. In a nutshell, CMAs with synchronization correspond to data automata with the ability to push a fixed symbol \$ to all class strings in one step. More precisely, \$ is inserted into each class string at the current position of the class automaton. Thus, from a two-counter automaton A, we construct a data automaton D such that A has an accepting run if and only if $L(D) \neq \emptyset$. In principle, runs of A correspond to strings over Γ in a similar fashion to in Proposition 6.1. The treatment of zero-tests is the crucial point. Whenever a zero-check for counter i occurs, and a class has already seen c_i^+ but not c_i^- then the counter i is not zero, and thus the data string should not be accepted. On the other hand, having seen c_j^+ for $j \neq i$ does no harm as the value of the other counter is arbitrary.

The idea of the reduction is to use synchronization to handle zero-checks. More precisely, at a symbol c_i^0, the automaton pushes the symbol \$ to all class strings. For classes having seen c_i^+ but not c_i^- this will lead to rejection. However, classes having seen c_j^+, for $j \neq i$, should not be affected by the zero-test. To this end, we use the additional symbols: After c_i^0, an arbitrary number of $\#_j$ can occur with the intention that each of them has the data value of some class that has seen c_j^+ but not yet c_j^-.

Thus, a class string is rejected by D if it contains a symbol \$ between c_i^+ and c_i^- and there is no symbol $\#_i$ right after the \$. More precisely, each class string must be of the form $c_i^+(\$\#_i)^* c_i^-(\$ + \#_i)^*$ or should not contain any c_i^+ or c_i^-. The computation of the base automaton checks that the input string corresponds to a sequence of states of A which is locally consistent, and that blocks of $\#_j, j \neq i$, only occur after c_i^0. It is not hard to show that if A has an accepting run ρ, a data string w_ρ accepted by D can be constructed.

For the reverse part of the proof, let w be a data string accepted by D. Clearly, there is a sequence ρ of states of A corresponding to w which is locally consistent. It remains to show that there is such a ρ which is actually a run of A (i.e., where the transitions are consistent with the counters). For a contradiction, let us assume that some zero-test corresponding to a position labeled c_i^0 is inconsistent. That is, there is a class with some data value d, in which c_i^+ has occurred but no subsequent c_i^- has been seen. Thus, in the class string of d the symbol \$ is inserted but the subsequent block of symbols does not contain $\#_i$. We can conclude that the class string w_d of d is rejected by D, a contradiction. This is simply because there is no way to get the symbol $\#_i$ behind \$ in w_d: another \$ is pushed to w_d before the next block of $\#_i$ symbols. □

Since full synchronization is too powerful, we next suggest a limited version of synchronization, which allows the automaton to *forget* all information computed so far for the classes. This ability, which we call *reset*, is enough to capture L_s from Example 7.1.

Definition 7.5. A class memory automaton **with reset** is a CMA with synchronization function g such that for all states q, either $g(q) = \{q\}$ or $g(q) = \{\bot\}$.

Example 7.6. We construct a CMA with reset that recognizes L_s from Example 7.1. This automaton is very similar to the one in Example 3.2. We only need to add transitions for the network failure symbol n and a synchronization function that they apply. When reading an n with data value d in a configuration (γ, f), the automaton always requires that $f(d) = \bot$. If $\gamma \in \{p, wb\}$ the automaton goes to state wb and if $\gamma \in \{i, wi\}$ it goes to i.

The synchronization function g is defined by $g(p) = p$ and $g(wi) = g(wb) = g(i) = \bot$. All transitions that read an n apply g while none of the others does.

The intuition behind this definition is the following. When a network failure occurs, the only process that is remembered is the one currently printing (if there is one). This process has class memory p. If the automaton was in state p or wb, it goes to wb, signifying that the printer is busy. If the automaton was in i or wi, it goes to i, signifying that the printer is idle.

Corollary 7.7. *CMAs with reset are strictly stronger than ordinary CMAs.*

Proposition 7.8. *Emptiness for CMAs with reset is decidable.*

Proof. The proof is similar to the one for data automata in [3]. We briefly sketch the idea of that proof. It shows that for each data automaton $D = (A, B)$ a multicounter automaton M (on non-data strings) can be constructed which accepts the set $\mathrm{str}(L(D))$. Thus, given a finite alphabet string v, M checks whether v can be extended to a data word w that is accepted by D. To this end, M uses one counter c_q, for each state q of B. When reading w, M guesses when the classes change, simulates what A would do, and, for each class, what B would do with the parts of the class strings produced so far. The counter c_q is used to keep track of for how many classes M guesses that the part of the class string seen so far would lead B to state q. At the end, for each state q of B which is not final it is required that $c_q = 0$. Some more details are needed to ensure $L(M) = \mathrm{str}(L(D))$. Modifying the proof to work for CMAs instead of data automata is straightforward; we use one counter per state of the CMAs.

The only thing we need to add is how the multicounter automaton M should handle reset transitions. What it needs to do is to set all its counters that represent states q for which $g(q) = \perp$ to zero. Ordinary multicounter automata cannot do this. It is shown in [16], however, that emptiness for multicounter automata with this ability is still decidable. [3] □

Even though we have seen that the addition of a reset capability was enough to capture L_s from Example 7.1, it is of course not the solution to every verification problem. As mentioned in the introduction, the landscape of modeling tools for data languages is quite heterogeneous, and in many cases, it seems that we will have to select the model that we use carefully, after analyzing the problem we actually want to solve. Sometimes we may have to engineer new models in order to capture a particular problem. Below, we give another example of how CMAs may be taken as the basis for designing a slightly stronger model, which is still decidable.

Example 7.9. Consider again a printer system, but one in which a *partial* network failure may occur. Computations of this system may be modeled as a slight modification of the data language L_s from Example 7.1. Thus, let L_t be the language obtained by removing rule (4) from the definition of L_s.

We now define an automaton model which, like CMAs with reset, is a special case of CMAs with synchronization, and which, among other things, can recognize L_t.

Definition 7.10. Let C be a CMA with synchronization, Q the states of C, and g its synchronization function. Consider the graph $G_g = (Q, E)$ of g defined as follows. There is an edge from p to q if and only if $q \in g(p)$. A subset E' of E defines a permutation on Q if it is functional and bijective (each state in Q has exactly one incoming and one outgoing edge in E'). We say that C has **restricted synchronization** if there is a subset of E that induces a permutation on Q.

Example 7.11. We construct a CMA C_t with restricted synchronization that accepts L_t. This automaton is very similar to the one in Example 7.6, and also extends automaton C for L_0 from Example 3.2. Automaton C_t uses the same states as C, plus two additional states ni and nb, which are used for the classes of the network processes. The synchronization function g is defined by

- $g(i) = \{i\}$,
- $g(wi) = \{i, wi\}$,
- $g(p) = \{p\}$,
- $g(wb) = \{i, wb\}$,
- $g(ni) = \{ni\}$,
- $g(nb) = \{nb\}$.

Since $p \in g(p)$ for all states p, g clearly provides restricted synchronization.

For each state in $\{p, wb, nb\}$ there is an additional transition to nb that reads and n and checks that the memory for the data value is in $\{\perp, ni, nb\}$. For each state in $\{i, wi, ni\}$ there is a corresponding transition to ni. The outgoing transitions from ni and nb are identical to those of wi and wb, respectively. When C_s reads an n, the synchronization function is applied. Intuitively, the automaton guesses which jobs have to resend a print request and sets their state to i.

Corollary 7.12. *CMAs with restricted synchronization are strictly stronger than CMAs.*

Proof. The language L'_s from the proof of Proposition 7.2 is also a sublanguage of L_t. □

Proposition 7.13. *Emptiness for CMAs with restricted synchronization is decidable.*

[3] The automaton model in [16] is called the priority multicounter automaton, and can actually do more than what we need here.

Proof. As in the proof of Proposition 7.8 we only have to extend the argument for DAs from [3]. For an outline of this argument, see the proof of Proposition 7.8.

To extend this construction to a CMA C with restricted synchronization we do the following. Let g be the synchronization function of C, and $G_g = (Q, E)$. Assume that $E' \in E$ induces a permutation on Q. What should M do to simulate a synchronizing transition of C? If, for example, $g(p) = \{q\}$ and $g(q) = \{p\}$, then the counters c_p and c_q should switch their values. Instead of performing this switch by incrementing and decrementing counters, M can simply change the interpretation of the affected counters. That is, the counter representing p becomes that for q and vice versa. Since there are only a finite number of assignments of counters to states, the correspondence between counters and states of C can be maintained in the state of M. When simulating a synchronizing transition, M updates this assignment according to the edges in E'.

So far, we haven't explained what happens if $g(q)$ for some state q is not a singleton, i.e., in G_g, q has at least one outgoing edge in $E \setminus E'$. Assume that $g(q) = \{p_1, \ldots, p_k\}$, and that $(q, p_1) \in E'$. When M simulates a synchronizing transition, it first non-deterministically transfers counter content from c_q to $c_{p_2}, \ldots, c_{p_k}$, and only then updates the counter assignment, assigning c_q to p_1. In this way, all synchronizing actions of C can be simulated. □

Acknowledgements

We thank Mikołaj Bojańczyk, Anca Muscholl and Luc Segoufin for many valuable discussions.

References

[1] P. Abdulla, B. Jonsson, M. Nilsson, M. Saksena, A survey of regular model checking, in: CONCUR'04, in: LNCS, vol. 3170, 2004, pp. 35–48.
[2] M. Arenas, W. Fan, L. Libkin, Consistency of XML specifications, in: Inconsistency Tolerance, in: LNCS, vol. 3300, 2005, pp. 15–41.
[3] M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, C. David, Two-variable logic on words with data, in: LICS'06, 2006, pp. 7–16.
[4] P. Bouyer, A. Petit, D. Thérien, An algebraic approach to data languages and timed languages, Information and Computation 182 (2) (2003) 137–162.
[5] S. Demri, R. Lazić, LTL with the freeze quantifier and register automata, in: LICS'06, 2006, pp. 17–26.
[6] R.G. Downey, Parameterized complexity for the skeptic, in: CCC'03, 2003, pp. 147–169.
[7] R.G. Downey, M.R. Fellows, Fixed-parameter tractability and completeness II: On completeness for W[1], Theoretical Computer Science 141 (1–2) (1995) 109–131.
[8] R.G. Downey, M.R. Fellows, Parameterized Complexity, Springer, 1999.
[9] E. Emerson, K. Namjoshi, Reasoning about rings, in: POPL'95, 1995, pp. 85–94.
[10] J. Flum, M. Grohe, Parameterized Complexity Theory, Springer, 2006.
[11] J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.
[12] M. Kaminski, N. Francez, Finite-memory automata, Theoretical Computer Science 132 (2) (1994) 329–363.
[13] M. Kaminski, T. Tan, Regular expressions for languages over infinite alphabets, in: K. Chwa, J. Munro (Eds.), COCOON 04, in: LNCS, vol. 3106, 2004, pp. 171–178.
[14] F. Neven, Automata, logic, and XML, in: CSL '02, in: LNCS, vol. 2471, 2002, pp. 2–26.
[15] F. Neven, T. Schwentick, V. Vianu, Finite state machines for strings over infinite alphabets, ACM Transactions on Computational Logic 15 (3) (2004) 403–435.
[16] K. Reinhardt, Counting as method, model and task in theoretical computer science, Habilitation Thesis, University of Tübingen, 2005.
[17] H. Sakamoto, D. Ikeda, Intractability of decision problems for finite-memory automata, Theoretical Computer Science 231 (2) (2000) 297–308.
[18] L. Segoufin, Automata and logics for words and trees over an infinite alphabet, in: Computer Science Logic (CSL), in: LNCS, vol. 4207, 2006, pp. 41–57.
[19] T. Wilke, Automaten und Logiken zur Beschreibung zeitabhängiger Systeme, Ph.D. Thesis, University of Kiel, 1994.