Asynchronous Multi-Tape Automata Intersection: Undecidability and Approximation

Carlo A. Furia
ETH Zurich, Switzerland
caf@inf.ethz.ch

February 13, 2014

Abstract

When their reading heads are allowed to move completely asynchronously, finite-state automata with multiple tapes achieve a significant expressive power, but also lose useful closure properties—closure under intersection, in particular. This paper investigates to what extent it is still feasible to use multi-tape automata as recognizers of polyadic predicates on words. On the negative side, determining whether the intersection of asynchronous multi-tape automata is expressible is not even semidecidable. On the positive side, we present an algorithm that computes under-approximations of the intersection; and discuss simple conditions under which it can construct complete intersections. A prototype implementation and a few non-trivial examples demonstrate the algorithm in practice.

1 Automata As Decision Procedures

Standard finite-state automata are simple computing devices widely used in computer science. They define a robust class of language acceptors, as each automaton instance A identifies a set $\mathcal{L}(A)$ of words that it accepts as input. The connection between finite-state automata and predicate logic has been well-known since the work of Büchi [4, 5] and others [33, 10], and is widely used in applications such as model-checking: each automaton A_P can be seen as *implementing* a monadic (that is, unary) predicate $P(x)$, in the sense that the set $\mathcal{L}(A_P)$ of words accepted by the automaton corresponds to the set $\{x \mid x \models P(x)\}$ of models of the predicate. Logic connectives (negation $\neg$, conjunction $\wedge$, etc.) translate into composition operations on automata (complement, intersection $\cap$, etc.), so that finite-state automata can capture the semantics of arbitrary first-order monadic formulas whose interpreted atomic predicates are implementable. This gives a very efficient way to decide the satisfiability of monadic logic formulas representable by finite-state automata: unsatisfiability of a formula corresponds to emptiness of its automaton, which is testable efficiently in time linear in the automaton size.

It is natural to extend this framework [1, 32] to represent n-ary predicates, for $n > 1$, by means of *multi-tape* finite-state automata. An n-tape automaton A_R is a device that accepts n-tuples of words, corresponding to the set of models of a predicate $R(x_1, \ldots, x_n)$ over n variables. Section 2 defines multi-tape automata and summarizes some of their fundamental properties. It turns out that the class of multi-tape automata (in their most expressive *asynchronous* variant) is not as robust as one-tape automata.

In particular, multi-tape automata[1] are not closed under intersection [14], and hence the conjunction of n-ary predicates is not implementable in general.

This paper investigates the magnitude of this hurdle in practice. On the negative side, we prove that determining whether the intersection of two multi-tape automata A, B is expressible as an automaton is neither decidable nor semi-decidable. On the positive side, we provide an algorithm $\mathfrak{I}(A, B, d)$ that computes an under-approximation of the intersection $A \cap B$ of A and B, bounded by a given maximum delay d between heads on different tapes. We also detail a simple sufficient syntactic condition on A and B for the algorithm to return complete intersections. Based on these results, we implemented the algorithm and tried it on a number of natural examples inspired by the verification conditions of programs operating on sequences.

2 Preliminaries

$\mathbb{Z}$ is the set of integer numbers, and $\mathbb{N}$ is the set of natural numbers $0, 1, \ldots$. For a (finite) set S, $\wp(S)$ denotes its powerset. For a finite nonempty *alphabet* Σ, Σ^* denotes the set of all finite sequences $\sigma_1 \cdots \sigma_n$, with $n \geq 0$, of symbols from Σ called *words* over Σ; when $n = 0$, $\epsilon \in \Sigma^*$ is the *empty* word. $|s| \in \mathbb{N}$ denotes the length n of a word $s = \sigma_1 \ldots \sigma_n$. An n-*word* is an n-tuple $\langle s_1, \ldots, s_n \rangle \in (\Sigma^*)^n$ of words over Σ.

Given a sequence $s = x_1 \cdots x_n$ of objects, a *permutation* $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a bijection that rearranges s into $\pi(s) = \pi_1 \cdots \pi_n$ with $\pi_i = x_{\pi(i)}$ for $i = 1, \ldots, n$. An *inversion* of a permutation π of s is a pair (i, j) of indices such that $i < j$ and $\pi(i) > \pi(j)$. For example, the permutation that turns $a_4 b_1 b_2 a_5 a_6 a_7 b_3$ into $b_1 b_2 b_3 a_4 a_5 a_6 a_7$ has 6 inversions.

2.1 Multi-Tape Finite Automata

A finite-state automaton with $n \geq 1$ tapes scans n read-only input tapes, each with an independent head. At every step, the current state determines the tape to be read, and the transition function defines the possible next states based on the current state and the symbols under the reading head. A special symbol \$ marks the right end of each input tape; $\Sigma_\$$ denotes the extended alphabet $\Sigma \cup \{\$\}$.

Definition 1 (n-tape automaton)**.** An *n-tape finite-state automaton* A is a tuple $\langle \Sigma, Q, \delta, Q_0, F, T, \tau \rangle$ where: Σ is the input alphabet, with $\$ \notin \Sigma$; $T = \{t_1, \ldots, t_n\}$ is the set of tapes; Q is the finite set of states; $\tau : Q \to T$ assigns a tape to each state; $\delta : Q \times \Sigma_\$ \to \wp(Q)$ is the (nondeterministic) transition function; $Q_0 \subseteq Q$ are the initial states; $F \subseteq Q$ are the accepting (final) states.

We write $A(t_1, \ldots, t_n)$ when we want to emphasize that A operates on the n tapes $t_1, \ldots, t_n$; $A(t'_1, \ldots, t'_n)$ denotes an instance of A with each tape t_i renamed to t'_i. Without loss of generality, assume that the accepting states have no outgoing edges: $\delta(q_F, \sigma) = \emptyset$ for all $q_F \in F$. Also, whenever convenient we represent the transition function δ as a relation, that is the set of triples (q, σ, q') such that $q' \in \delta(q, \sigma)$.

A *configuration* of an n-tape automaton A is an $(n+1)$-tuple $\langle q, y_1, \ldots, y_n \rangle \in Q \times (\Sigma^*_\$)^n$, where $q \in Q$ is the current state and, for $1 \leq k \leq n$, y_k is the input on the k-th tape still to be read. A *run* ρ of A on input $x = \langle x_1, \ldots, x_n \rangle \in (\Sigma^*)^n$ is

[1] We do not consider more powerful classes of multi-tape automata, such as pushdown automata, as they typically possess even fewer closure or decidability properties [20] unless they are significantly restricted to specific classes of languages [11].

a sequence of configurations $\rho = \rho_0 \cdots \rho_m$ such that: (1) $\rho_0 = \langle q_0, x_1\,\$, \ldots, x_n\,\$\rangle$ for some initial state $q_0 \in Q_0$; and (2) for $0 \leq k < m$, if $\rho_k = \langle q, y_1, \ldots, y_n\rangle$ is the k-th configuration—with $t_h = \tau(q)$ the tape read in state q, and $y_h = \sigma\, y'_h$ with $\sigma \in \Sigma_\$$ and $y'_h \in \Sigma^*_\$$ on the h-th tape—then $\rho_{k+1} = \langle q', y'_1, \ldots, y'_n\rangle$ with $q' \in \delta(q, \sigma)$ and $y'_i = y_i$ for all $i \neq h$. A run ρ is *accepting* if $\rho_m = \langle q_F, y_1, \ldots, y_n\rangle$ for some accepting state $q_F \in Q_F$. A accepts an n-word x if there exists an accepting run of A on x. The *language* accepted (or recognized) by A is the set $\mathcal{L}(A)$ of all n-words that A accepts. The *n-rational languages* are the class of languages accepted by some n-tape automaton. Whenever n is clear from the context, we will simply write "words" and "automata" to mean "n-words" and "n-tape automata".

Definition 2. An n-tape automaton A is: **deterministic** if $|Q_0| \leq 1$ and $|\delta(q, \sigma)| \leq 1$ for all q, σ; **synchronous** for $s \in \mathbb{N}$ if every run of A is such that any two heads that have not scanned their whole input are no more than s positions apart; **asynchronous** if it is not synchronous for any s.

Example 3. Figure 1 shows a synchronous deterministic automaton $\mathcal{A}_=$ with two tapes X, Y that recognizes pairs of equal words over $\{a, b\}$. Each state is labeled with the tape read and with a number for identification (the final state's tape label is immaterial, and hence omitted). $\mathcal{A}_=$ reads one letter on tape Y immediately after reading one letter on tape X; hence it is synchronous for $s = 1$. Automaton $\mathcal{A}_\circ$ in Figure 2 recognizes triples of words such that the word on tape Z equals the concatenation of the words on tapes X and Y (ignoring the end-markers). It is asynchronous because the length of X is not bounded: when the reading on tape Y starts, the head on Z is at a distance equal to the length of the input on X.

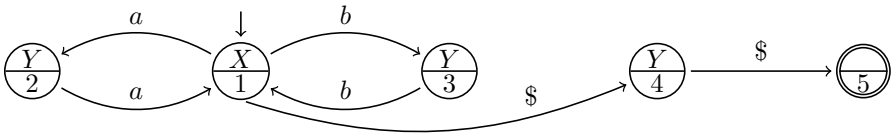


Figure 1: 2-tape deterministic synchronous automaton $\mathcal{A}_=$.

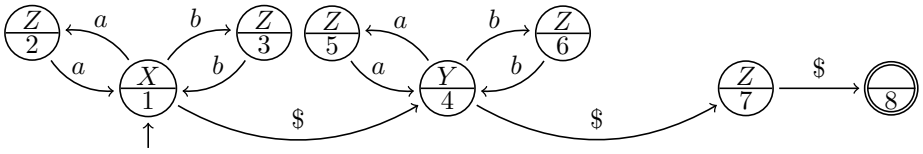


Figure 2: 3-tape deterministic asynchronous automaton $\mathcal{A}_\circ$.

2.2 Closure Properties and Decidability

Automata define languages, which are sets of words; correspondingly, we are interested in the closure properties of automata with respect to set-theoretic operations on their languages. Specifically, we consider closure under complement, intersection, and union; and the *emptiness* problem: given an automaton A, decide whether $\mathcal{L}(A) = \emptyset$,

that is whether it accepts some word. The complement of a language L over n-words over Σ is taken with respect to the set $(\Sigma^*)^n$; the intersection $L_1 \cap L_2$ is also applicable when L_1 is a language over n-words and L_2 a language over m-words, with $m > n$: define $L_1 \cap L_2$ as the set of m-tuples $\langle x_1, \ldots, x_m \rangle$ such that $\langle x_1, \ldots, x_n \rangle \in L_1$ and $\langle x_1, \ldots, x_m \rangle \in L_2$; a similar definition works for unions. We lift set-theoretic operations from languages to automata; for example, the intersection $A_3 = A_1 \cap A_2$ of two automata A_1, A_2 is an automaton A_3 such that $\mathcal{L}(A_3) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$, when it exists; we assume that intersected automata share the tapes with the same name (in the same order). The rest of this section summarizes the fundamental closure properties of multi-tape automata; see [14] for a more detailed presentation and references.

Synchronous automata [22, 23] define a very robust class of languages: they have the same expressiveness whether deterministic or nondeterministic; they are closed under complement, intersection, and union; and emptiness is decidable. In fact, computations of synchronous n-tape automata can be regarded as computations of standard single-tape automata over the n-track alphabet $(\Sigma \cup \{\Box\})^n$, where the fresh symbol $\Box$ pads some of the n input strings so that they all have the same length. Under this convention, the standard constructions for finite-state automata apply to synchronous automata as well. Most applications of multi-tape automata to have targeted synchronous automata (see Section 6), which have, however, a limited expressive power.

Asynchronous automata are strictly more expressive than synchronous ones, but are also less robust:

- Nondeterministic asynchronous automata are strictly more expressive than deterministic ones.
- *Deterministic* asynchronous automata are closed under complement, using the standard construction that complements the accepting states. They are not closed under union, although the union of two deterministic asynchronous automata always is a nondeterministic automaton. They are not closed under intersection because, intuitively, the parallel computations in the two intersected automata may require the heads on the shared tapes to diverge.
- *Nondeterministic* asynchronous automata are not closed under complement or intersection, but are closed under union using the standard construction that takes the union of the transition graphs.
- *Emptiness* is decidable for asynchronous automata (deterministic and nondeterministic): it amounts to testing reachability of accepting states from initial states on the transition graph.

3 Multi-Tape Automata: Negative Results

Since multi-tape automata are not closed under intersection, we try to characterize the class of intersections that *are* expressible as automata. A logical characterization is arduous to get, because conjunction would be inexpressible in general. Indeed, we can prove some strong undecidability results.

Rational intersection is undecidable. The *rational intersection problem* is the problem of determining whether the intersection language $\mathcal{L}(A) \cap \mathcal{L}(B)$ of two automata A and B is rational, that is whether it is accepted by some multi-tape automaton.

Theorem 4. *The rational intersection problem is not semidecidable.*

Proof. Following [18, 19], we consider valid computations of Turing machines. A single-tape Turing machine M has state set S, input alphabet I, transition relation $\delta \subseteq Q \times I \times Q \times I \times \{-1, 0, 1\}$; and $s_0, s_F \in S$ respectively are the initial state and the accepting state (unique, without loss of generality). We can write M's configurations as strings over $I \cup S$ of the form $i_1 \cdots i_k \, s \, i_{k+1} \cdots i_m$ where $i_1 \cdots i_m \in I^*$ is the sequence of symbols on the tape, s is the current state, and the read/write head is over the symbol i_{k+1}. The set $\mathrm{ACC}(M)$ of accepting computations contains all words of the form $\#w_1\# \cdots \#w_m\#$, with $\# \notin I \cup S$, such that each w_k is a configuration of M, w_1 is an initial configuration (of the form $s_0 I^*$), w_m is an accepting configuration (of the form $I^* s_F I^*$), and w_{k+1} is a valid successor of w_k according to δ, for all $1 \leq k < m$ (that is, for $w_k = i_1 \cdots i^- \, s \, i^+ \cdots i_n$ and $(s, i^+, s', i', h) \in \delta$ then: if $h = -1$ then $w_{k+1} = i_1 \cdots s' i^- i' \cdots i_n$; if $h = 0$ then $w_{k+1} = i_1 \cdots i^- s i' \cdots i_n$; if $h = +1$ then $w_{k+1} = i_1 \cdots i^- i' s \cdots i_n$). The problem of determining, for a generic M, whether $\mathrm{ACC}(M)$ is regular is not semidecidable [18].

Consider now the language L_{M^2} defined as $\{\langle x, x\rangle \mid x \in \mathrm{ACC}(M)\}$. Since the single-component projection of a rational language is always regular [30], if L_{M^2} is rational then $\mathrm{ACC}(M)$ is regular. We can express L_{M^2} as the intersection of two languages L^1_M and L^2_M. L^1_M is the set of 2-words $\langle \#u_1\# \cdots \#u_m\#, \#v_1\# \cdots \#v_m\# \rangle$ such that: u_1 is an initial configuration of M; v_m is an accepting configuration; for $1 \leq k < m$, u_k is a valid configuration and v_{k+1} is a valid successor of u_k. L^2_M is simply the set of 2-words whose first and second component are equal. It is not difficult to see that $L^1_M \cap L^2_M = L_{M^2}$ and both L^1_M and L^2_M are rational (and deterministic). An automaton for L^2_M works synchronously by alternately reading and comparing one character from each tape, generalizing the automaton in Figure 1. An automaton for L^1_M starts with the second head moving forward to v_2; it then compares each u_k and v_{k+1} one character at a time, checking that they are consistent with M's δ.

We can finally prove the theorem by contradiction: assume the rational intersection problem is semidecidable. Then, the following is a semi-decision procedure for the problem of determining whether $\mathrm{ACC}(M)$ is regular. Construct the automata for L^1_M and L^2_M. If $L^1_M \cap L^2_M = L_{M^2}$ is rational, then the semi-decision procedure for rational intersection halts with positive outcome; then we conclude that $\mathrm{ACC}(M)$ is regular; otherwise loop forever. Since regularity of $\mathrm{ACC}(M)$ is not semidecidable, we have a contradiction. □

Remark 5. The rational intersection problem belongs to Σ^0_2 in the arithmetical hierarchy. Consider an enumeration $C_1, C_2, \ldots$ of multi-tape automata. The rational intersection problem for A and B is expressible as: $\exists z \forall x : C_z(x) \Leftrightarrow A(x) \wedge B(x)$, where $C_k(x)$ is a predicate that holds iff automaton C_k accepts input x. The formula $P(x, z) = C_z(x) \Leftrightarrow A(x) \wedge B(x)$ within quantifier scope is recursive (just simulate the automata runs); hence $\forall x : P(x, z)$ is Π^0_1 and $\exists z \forall x : P(x, z)$ is Σ^0_2.

Rational nondeterministic complement is undecidable. Recall that deterministic automata are closed under complement and nondeterministic automata are closed under union. Therefore, the undecidability Theorem 4 carries over to the rational complement problem (defined as obvious): the languages L^1_M and L^2_M used in the proof of Theorem 4 are deterministic; hence their complements $\overline{L^1_M}$ and $\overline{L^2_M}$ are rational languages whose union $\overline{L^1_M} \cup \overline{L^2_M}$ is also rational. Thus, the complement of $\overline{L^1_M} \cup \overline{L^2_M} = \overline{L^1_M \cap L^2_M}$ is rational iff $\mathrm{ACC}(M)$ is regular.

Corollary 6. *The rational complement problem (determining whether the complement of a rational nondeterministic language is rational) is not semidecidable.*

Closure with respect to equality implies synchrony. The proof of Theorem 4 also reveals that even the intersection of an asynchronous automaton with a synchronous one is in general not rational. A natural question is then whether there exist interesting combinations of synchronous and asynchronous automata whose intersection is rational. A particularly significant case is equality of tapes: it is a relation clearly recognized by synchronous automata, and it plays an important role in the combination of decision procedures (e.g., à la Nelson-Oppen [28] or following the DPLL(T) paradigm [29]). Unfortunately, it is a corollary of standard results that the only "natural" and robust class of rational languages are those accepted by synchronous automata.

Corollary 7. *Consider an n-tape automaton A; if the language $\mathcal{L}(A) \cap \{x^n \mid x \in \Sigma^*\}$ is rational, then it is also accepted by a synchronous automaton.*

Proof. Language $L = \mathcal{L}(A) \cap \{x^n \mid x \in \Sigma^*\}$ is so-called "length-preserving": in any word in L, all components have the same length. Theorem 6.1 in [8, Chap. IX] shows that length-preserving rational languages are synchronous. □

4 Multi-Tape Automata: Positive Results

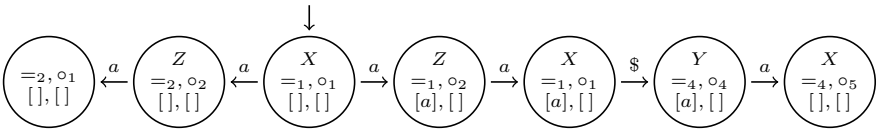
The undecidability of whether an intersection is rational does not prevent the definition of *approximate* algorithms for intersection. Section 4.1 describes one such algorithm that bounds the maximum delay between corresponding heads of the intersecting automata; the algorithm under-approximates the real intersection. We also discuss very simple syntactic conditions under which a bound of zero delay still yields a complete intersection. Section 4.3 discusses to what extent some of these results can be extended to the approximation of complement for nondeterministic automata.

4.1 An Algorithm for the Under-Approximation of Intersection

This section outlines an algorithm $\mathfrak{I}(A, B, d)$ that inputs two multi-tape automata A and B and a delay bound $d \in \mathbb{N} \cup \{\infty\}$ and returns a multi-tape automaton C that approximates the intersection $A \cap B$ to within delay d. The intersection construction extends the classic "cross-product" construction: simulate the parallel runs of the two composing automata by keeping track of what happens in each component.

Informal overview. Let us introduce the algorithm's basics through examples. Consider the intersection of $\mathcal{A}_=$ and $\mathcal{A}_\circ$ in Figures 1 and 2; the initial state is labeled $\langle =_1, \circ_1 \rangle$ to denote that it combines states $=_1$ (i.e., state 1 in $\mathcal{A}_=$) and $\circ_1$ (state 1 in $\mathcal{A}_\circ$). As the intersection develops, the composing automata synchronize on transitions on shared tapes and proceed asynchronously on non-shared tapes. In the example, there is a synchronized transition from $\langle =_1, \circ_1 \rangle$ to $\langle =_2, \circ_2 \rangle$ upon reading a on shared tape X, and an asynchronous transition from the latter state to $\langle =_2, \circ_1 \rangle$ upon reading a on $\mathcal{A}_\circ$'s non-shared tape Z. $\mathcal{A}_=$ in state $=_2$ can also read a on shared tape Y; this is a valid move in the intersection even if $\mathcal{A}_\circ$ cannot read on tape Y until it reaches state $\circ_4$. Since reading can proceed on other tapes, we just have to "delay" the transition that reads a on Y to a later point in the computation and store this delay using the states of the intersection automaton; $\mathcal{A}_\circ$ will then be able to take other transitions and will consume the delayed ones asynchronously *before* taking any other transition on Y (that is, delays

behave as a FIFO queue). For example, when a run of the intersection automaton reaches state $\langle =_4, \circ_4 \rangle$, $\mathcal{A}_\circ$ can read a on Y matching $\mathcal{A}_=$'s delayed transition (which is then consumed). Here is a picture showing these steps:



Delays may become unbounded in some cases. In the example, automaton $\mathcal{A}_=$ may accumulate arbitrary delays on tape Y while in states $=_1, =_2, =_3$; this corresponds to the intersection automaton "remembering" an arbitrary word on tape Y to compare it against Z's content later. An unbounded delay is necessary in this case, as the computations on $\mathcal{A}_=$ and $\mathcal{A}_\circ$ manage the heads on X and Y in irreconcilable ways: the intersection language of $\mathcal{A}_=$ and $\mathcal{A}_\circ$ is not rational.

The algorithm. Consider two automata $A = \langle \Sigma, Q^A, \delta^A, Q_0^A, F^A, T^A, \tau^A \rangle$ and $B = \langle \Sigma, Q^B, \delta^B, Q_0^B, F^B, T^B, \tau^B \rangle$, such that A has m tapes $T^A = \{t_1^A, \ldots, t_m^A\}$ and B has n tapes $T^B = \{t_1^B, \ldots, t_n^B\}$. We present an algorithm $\mathfrak{I}(A, B, d)$ that constructs an automaton $C = \langle \Sigma, Q, \delta, Q_0, F, T, \tau \rangle$—with C's tapes $T = T^A \cup T^B$—such that $\mathcal{L}(C) \subseteq \mathcal{L}(A) \cap \mathcal{L}(B)$. We describe the algorithm as the combination of fundamental operations, introduced as separate routines. All components of the algorithm have access to the definitions of A and B, to the definition of C being built, and to a global stack *s* where new states of the composition are pushed (when created) and popped (when processed). The complete pseudo-code of the routines is in Section B of the Appendix.

Routine *async_next* (lines 1–17 in Figure 4) takes a t-tape automaton D (i.e., A or B) and one of its states q, and returns a set of tuples $\langle q', h_1, \ldots, h_t \rangle$ of all next states reachable from q by accumulating delayed transitions $h_i \in (\delta^D)^*$ in tape t_i, for $1 \leq i \leq t$. We call *delayed states* such tuples of states with delayed transitions. The search for states reachable from q stops at the first occurrences of states associated with a certain tape. For example, *async_next* $(\mathcal{A}_\circ, \circ_1)$ consists of $\langle \circ_1, \epsilon, \epsilon, \epsilon \rangle$, $\langle \circ_2, (\circ_1, a, \circ_2), \epsilon, \epsilon \rangle$, $\langle \circ_3, (\circ_1, b, \circ_3), \epsilon, \epsilon \rangle$, $\langle \circ_4, (\circ_1, \$, \circ_4), \epsilon, \epsilon \rangle$, $\langle \circ_5, (\circ_1, \$, \circ_4), (\circ_4, a, \circ_5), \epsilon \rangle$, $\langle \circ_6, (\circ_1, \$, \circ_4), (\circ_4, b, \circ_6), \epsilon \rangle$, and $\langle \circ_7, (\circ_1, \$, \circ_4), (\circ_4, \$, \circ_7), \epsilon \rangle$.

Consider now a pair of delayed states $\langle p, h_1, \ldots, h_m \rangle$ and $\langle q, k_1, \ldots, k_n \rangle$, respectively of A and B. The two states can be composed only if the delays on the synchronized tapes are pairwise *consistent*, that is the sequence of input symbols of one is a prefix (proper or not) of the other's; otherwise, the intersection will not be able to consume the delays in the two components because they do not match. *cons*(h_i, k_i) denotes that the sequences h_i, k_i of delayed transitions are consistent. Routine *new_states* (lines 19–26 in Figure 4) takes two sets P, Q of delayed states and returns all consistent states obtained by composing them. *new_states* also pushes onto the stack *s* all composite states that have not already been added to the composition. For convenience, *new_state* also embeds the tape t of each new composite state within the state itself. (All tapes are considered: states corresponding to inconsistent choices will be dead ends.)

To add arbitrary prefixes to the delays of delayed states generated by *new_states*, routine *compose_transition* (lines 28–33 in Figure 4) takes two sets P, Q of delayed states and an $(m + n)$-tuple of delays, and calls *new_states* on the modified states obtained by orderly adding the delays to the states in P and Q. It also adds all transitions reaching the newly generated states to C's transition function δ.

We are ready to describe the main routine *intersect* which builds C from A and

B; see Figure 5 for the pseudo-code (some symmetric cases are omitted for brevity). Routine *intersect* takes as arguments a bound on the maximum number of states and on the maximum delay *max_delay* (measured in number of transitions) accumulated in the states. After building the initial states of the compound (lines 4–5), *intersect* enters a loop until either no more states are generated (i.e., the stack *s* is empty) or it has reached the bound *max_states* on the number of states. Each iteration of the loop begins by popping a state r from the top of the stack (line 7). r is normally added to the set Q of C's states, unless some of its sequences of delayed transitions are longer than the delay bound *max_delay*; in this case, the algorithm discards r and proceeds to the next iteration of the loop (line 8). If r is not discarded, *intersect* builds all composite states reachable from r. These depend on the tape t read when in r: if it is shared between A and B we have synchronized transitions (lines 10–30), otherwise we have an asynchronous transition of A (lines 32–41) or one of B (line 43).

Consider the case of a synchronized transition on some shared tape $t \in T^A \cap T^B$. While both A and B must read the same symbol on the same tape, they may do so by consuming some transition that has been delayed. For example, if A has a non-empty delay $h_t \neq \epsilon$ for tape t, it will consume the first transition (u_a, σ, u'_a) in h_t; since the transition is delayed, A's next state in the compound is not determined by the delayed transition (which only reads the input σ at a delayed instant) but by A's current state q_a in the compound (line 12 and line 17). The reached states are the composition of those reached within A and B, with the delays updated so as to remove the delayed transitions consumed. For example, lines 12–14 correspond to both A and B taking a delayed transition, whereas lines 17–20 correspond to A taking a delayed transition and B taking a "normal" transition determined by its transition function δ^B on symbol σ. If neither A nor B have delayed transitions for tape t, they can only perform normal transitions according to their transitions functions, without consuming the delays stored in the state; this is shown in lines 26–30.

The final portion of *intersect* (from line 32) handles the case of transitions on some non-shared tape t. In these cases, the component of the state r corresponding to the automaton that does not have tape t does not change at all, whereas the other component is updated as usual—either by taking a delayed transition (lines 33–35) or by following its transition function (lines 37–41).

The output of $\mathfrak{I}(A, B, d)$ coincides with the main routine *intersect* called on A and B with no bound on the number of states and *max_delay* = d; the final states F in C coincide with those whose components are both final in A and B and have no delayed transitions.

4.2 Correctness and Completeness

In the proofs of this section, we make the simplifying assumption that all tapes are shared: $T^A = T^B = T$; handling non-shared tapes is straightforward. Let us show that $\mathfrak{I}(A, B, d)$ is correct, that is it constructs an under-approximation of the intersection.

Theorem 8 (Correctness)**.** *For every finite delay $d \in \mathbb{N}$, $\mathfrak{I}(A, B, d)$ returns a C such that $\mathcal{L}(C) \subseteq \mathcal{L}(A) \cap \mathcal{L}(B)$.*

Proof. Let us show that $x \in \mathcal{L}(C)$ implies $x \in \mathcal{L}(A) \cap \mathcal{L}(B)$. The basic idea is that, given an accepting run $\rho = \rho_0\rho_1 \cdots \rho_n$ of C on x, one can construct two permutations π^A, π^B such that $\pi^A(\rho)$ is an accepting run of A and $\pi^B(\rho)$ is an accepting run of B on x. The permutation π^A is constructed as follows (constructing π^B works in the same way): each element ρ_k in ρ, for $0 \leq k < n$, corresponds to either a synchronous

or a delayed transition of A; in the former case, π^A does not change the position of ρ_k, otherwise it moves it to where the transition was delayed (i.e., consumed asynchronously). For accepting runs, it is always possible to construct such permutations, since accepting states in C have no delays, and hence delayed transitions must have been consumed somewhere *before* reaching the accepting state. □

Remark 9 (Termination). $\mathfrak{I}(A, B, d)$ only expands states encoding a maximum delay of d, and terminates after the given maximum number of states have been generated or when all states have been explored—whatever comes first. Upon termination, in general we do not know if the generated intersection automaton accepts $\mathcal{L}(C)$ or only a subset of it—consistently with the undecidability degree of deciding the intersection (Remark 5).

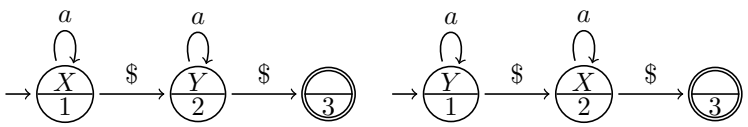
Remark 10 (Complexity). Since $\mathfrak{I}(A, B, d)$ may have to enumerate all d-delayed states, its worst-case time complexity is exponential in d and $|T|$—which determine the combinatorial explosion in the number of compound states—as we now illustrate.

To get an upper bound on the time complexity of $\mathfrak{I}(A, B, d)$, let $q = \max(|Q^A|, |Q^B|)$, $\delta = \max(|\delta^A|, |\delta^B|) = \mathrm{O}(q^2)$, and $t = |T^A \cap T^B| = |T^A| = |T^B|$. Consider the case in which *intersect* expands *all* d-delay compound states determined by A and B, and ignore constant multiplicative factors. The main loops executes once per compound state, that is $q^2\delta^{2dt}$ times. Each iteration: (1) calls *async_next* on A and B, taking time tq^3 using an algorithm such as Floyd-Warshall for the all-pairs shortest path (but whose results can be cached); and (2) composes the sets of states by calling *compose_transition* , taking time dtq^4 assuming states in Q are hashed. The dominant time-complexity factor is then dtq^6q^{4dt}, exponential in d and t.

There is a simple condition for $\mathfrak{I}(A, B, d)$ to return complete intersections. If A and B share only one tape, $\mathfrak{I}(A, B, 0)$ returns a C that reads the input on non-shared tapes asynchronously whenever possible; otherwise, C reads synchronously the input on the single shared tape without need to accumulate delays.

Lemma 11 (Sufficient condition for completeness)**.** *If A and B share at most one tape, then $\mathfrak{I}(A, B, 0)$ returns a C such that $\mathcal{L}(C) = \mathcal{L}(A) \cap \mathcal{L}(B)$.*

Example 12. Consider the intersection of $A_1 = \mathcal{A}_\circ(X, Y, Z)$ and $A_2 = \mathcal{A}_=(Z, W)$ (the latter is $\mathcal{A}_=$ in Figure 2 with tapes renamed to Z and W). Since A_1 and A_2 only share tape Z, they can be ready to read synchronously on Z whenever necessary without having to delay such transitions, since asynchronous transitions can be interleaved ad lib. Therefore, bounding the construction to have no delays gives an automaton that accepts precisely the intersection of A_1's and A_2's languages.



(a) Automaton $\mathcal{A}_X$. (b) Automaton $\mathcal{A}_Y$.

Figure 3: Two automata accepting the same language $\{\langle a^m, a^n\rangle \mid m, n \in \mathbb{N}\}$.

Remark 13. Even when called without bound on the delays, $\mathfrak{I}(A, B, \infty)$ may terminate; in this case, there is not guarantee on the completeness of the returned C.

Consider, for example,[2] automata $\mathcal{A}_X$ and $\mathcal{A}_Y$ in Figure 3: they both accept the language $L = \{\langle a^m, a^n\rangle \mid m, n \in \mathbb{N}\}$ but by reading on the tapes in different order. $\mathfrak{I}(\mathcal{A}_X, \mathcal{A}_Y, \infty)$ terminates and returns a C accepting the language

$$L_C = \{\langle a^n, \epsilon\rangle, \langle \epsilon, a^n\rangle \mid n \in \mathbb{N}\}.$$

Clearly, $L_C \subset \mathcal{L}(\mathcal{A}_X) \cap \mathcal{L}(\mathcal{A}_Y) = L$.

4.3 Approximating Complement

Since deterministic automata are closed under complement, we can use a construction to approximate determinization to build approximate complement automata. A straightforward under-approximation algorithm for determinization works as follows. Consider a generic nondeterministic automaton A, and let b be a bound on delays; $\widetilde{A}$ is the approximate deterministic version of A which we construct. Whenever A has a nondeterministic choice between going from state q to states q_1 or q_2 upon reading some σ, $\widetilde{A}$ goes to q_1 and continues the computation corresponding to that choice for up to b steps; while performing these b steps, $\widetilde{A}$ stores the symbols read in its finite memory. If the computation terminates with acceptance within b steps, then $\widetilde{A}$ accepts; otherwise, it continues with the computation that chose to go to q_2, using the stored finite input for b steps and then continuing as normal. It is clear that if such an automaton $\widetilde{A}$ accepts, A accepts as well; the converse is in general not true. Since $\widetilde{A}$ is deterministic, it can be complemented with the usual construction that switches accepting and non-accepting states.

Remark 14. Note that, while deterministic automata are closed under complement, the converse is not true: there exist rational languages whose complement is also rational that are strictly nondeterministic. For example, consider $L = \{\langle a^x, a^y\rangle \mid x \neq y \text{ or } x \neq 2y\}$. It is clear that L is rational; it also requires nondeterminism to "guess" whether to check $x \neq y$ (pair each a on the first tape with one a on the second tape) or $x \neq 2y$ (pair each a on the first tape with two a's on the second tape). L's complement $\overline{L}$ is the singleton set with $\langle \epsilon, \epsilon\rangle$, and hence also rational.

5 Implementation and Experiments

To demonstrate the constructions for multi-tape automata in practice, we implemented the algorithm of Section 4.1 in Python with the IGraph library to represent automata transition graphs; the prototype implementation is about 900 lines long, and includes other basic operations on asynchronous automata such as union, complement (for deterministic), and emptiness test. Using this prototype, we constructed eight composite automata corresponding to language-theoretic examples and simple verification conditions expressible as the composition of rational predicates, and tested them for emptiness. Table 1 lists the results of the experiments; the examples themselves are described below, and all the formulas are listed in Section E of the Appendix. All the experiments ran on a Ubuntu GNU/Linux box with Intel Quad Core2 CPU at 2.40 GHz, 4 GB of RAM, Python 2.7.3, and IGraph 0.6. Each experiment consists of two parts: computing the intersection until (possibly bounded) termination (INTERSECTION) and testing the emptiness on the simplified intersection (EMPTINESS). For each part of each experiment, Table 1 reports the time taken to complete it (t, in seconds); for the first part, it

[2] Thanks to the anonymous Reviewer 1 of CSR 2014 for suggesting this example.

	INTERSECTION			EMPTINESS	
	t	$\|Q\|$	$\|\delta\|$	t	?
$L_{1,2}$	0	14	6	0	N
$L_{3,4}$	0	56	48	0	N
tail : $\widetilde{\mathsf{vc}}_0$	0	32	32	0	Y
tail : $\widetilde{\mathsf{vc}}_1$	0	248	387	0	Y
tail : $\widetilde{\mathsf{vc}}_2$	119	1907	11061	4	Y
tail : ice_1	0	224	564	0	N
tail : ice_2	222	1644	21048	28	N
cat_0	2	595	1009	0	N

Table 1: Checking languages and verification conditions with multi-tape automata.

also shows the number of states $|Q|$ and transitions $|\delta|$ of the generated automaton; the EMPTINESS column also shows the outcome (?: Y for empty, N for non-empty), which is, of course, checked to be correct. Note that the prototype is only a proof-of-concept: there is plenty of room for performance improvement.

Language-theoretic examples. Examples $L_{1,2}$ and $L_{3,4}$ (taken from [24]) are 2-word languages whose intersection is finite. The structure of the automata recognizing the intersected components is such that the algorithm *intersect* can only unroll their loops finitely many times, hence terminates without a given bound. $L_{1,2}$ is the intersection $L_{1,2} = L_1 \cap L_2 = \langle abcabc, abcabca \rangle$ of $L_1 = \{\langle ab(cab)^n c, a(bc)^n abca \rangle \mid n \in \mathbb{N}\}$ and $L_2 = \{\langle (abc)^n, a(bca)^n \rangle \mid n \in \mathbb{N}\}$. $L_{3,4}$ is the intersection $L_{3,4} = L_3 \cap L_4 = \langle ab, xyz \rangle$ of $L_3 = \{\langle ab^n, xy^n z \rangle \mid n \in \mathbb{N}\}$ and $L_4 = \{\langle a^n b, xy^n z \rangle \mid n \in \mathbb{N}\}$. It is trivial to build the automata for L_1, L_2, L_3, L_4; the experiments reported in Table 1 composed them and determined their finite intersection languages, which happens to be complete for $L_{1,2}$ and $L_{3,4}$.

Program verification examples. Consider a routine *tail* that takes a nonnegative integer n and a sequence x and returns the sequence obtained by dropping the first n elements of x (where *rest* (x) returns x without its first element):

```
tail (n: ℕ, x: SEQUENCE): SEQUENCE is
    if n = 0 or x = ϵ then Result := x else Result := tail (n−1, rest (x)) end
```

If $|y|$ denotes the length of y, a (partial) postcondition for *tail* is:

$$(n = 0 \wedge \mathbf{Result} = x) \vee (n > 0 \wedge |x| \geq n \wedge |\mathbf{Result}| = |x| - n)\,. \tag{1}$$

The bulk of proving *tail* against this specification is showing that the postcondition established by the recursive call in the **else** branch (assumed by inductive hypothesis) implies the postcondition (1). Discharging this verification condition is equivalent to proving that three simpler implications, denoted vc_0, vc_1, and vc_2, are valid. For example: $\mathsf{vc}_1 \equiv |y| \geq m \wedge y = rest(x) \Rightarrow |x| \geq n \wedge m = n - 1$ states that if sequence $rest(x)$ has length $\geq n - 1$, then the sequence x has length $\geq n$.

We discharged the verification conditions $\mathsf{vc}_0, \mathsf{vc}_1, \mathsf{vc}_2$ using multi-tape automata constructions as follows. vc_k is valid if and only if $\widetilde{\mathsf{vc}}_k = \neg\mathsf{vc}_k$ is unsatisfiable. Hence, we have:

$$\widetilde{\mathsf{vc}}_1 = \neg\mathsf{vc}_1 \;\equiv\; |y| \geq m \wedge y = rest(x) \wedge (|x| < n \vee m \neq n - 1)\;.$$

Assume that sequence elements are encoded with a binary alphabet $\{a, b\}$ and elements of the sequence are separated by a symbol $\#$; this is without loss of generality as a binary alphabet can succinctly encode arbitrary sequence elements.

Then, define multi-tape automata that implement the atomic predicates appearing in the formulas; in all cases, these are very simple and small *deterministic* automata. For example, define 3 automata $\mathcal{A}_{len}(X, N)$, $\mathcal{A}_{rest}(X, Y)$, $\mathcal{A}_{dec}(M, N)$ for $\widetilde{\mathsf{vc}}_1$. In $\mathcal{A}_{len}(X, N)$, tape X stores arbitrary sequences encoded as described above, and tape N encodes a nonnegative integer in unary form (as many a's as the integer); $\mathcal{A}_{len}(X, N)$ accepts on X sequences whose length (i.e., number of $\#$'s) is no smaller than the number encoded on N. $\mathcal{A}_{rest}(X, Y)$ accepts if the sequence on tape Y equals the sequence on tape X with the first element (until the first $\#$) removed. $\mathcal{A}_{dec}(M, N)$ inputs two nonnegative integers encoded in unary on its tapes M, N and accepts iff M has exactly one less a than N.

Finally, compose an overall automaton according to the propositional structure of the formula $\widetilde{\mathsf{vc}}_k$ (using intersection, union, and complement as described in Section D of the Appendix) that is equivalent to it, and test if for emptiness. For example, $\mathcal{A}_{\widetilde{\mathsf{vc}}_1}$ is equivalent to $\widetilde{\mathsf{vc}}_1$:

$$\mathcal{A}_{\widetilde{\mathsf{vc}}_1} \quad \equiv \quad (\mathcal{A}_{len}(Y, M) \cap \mathcal{A}_{rest}(X, Y)) \cap \left(\overline{\mathcal{A}_{len}(X, N)} \cup \overline{\mathcal{A}_{dec}(M, N)}\right) , \quad (2)$$

where $\mathcal{A}_{len}(Y, M)$ denotes an instance of $\mathcal{A}_{len}$ with tapes X, N renamed to Y, M. In all cases $\widetilde{\mathsf{vc}}_0, \widetilde{\mathsf{vc}}_1, \widetilde{\mathsf{vc}}_2$, the overall automaton is effectively constructible from the basic automata and each intersection shares only one tape; hence constructing intersections with a zero bound on delays is complete (Lemma 11). For example, $\mathcal{A}_{\widetilde{\mathsf{vc}}_1}$ build with zero delays is complete, because each element of the disjunction (1) is treated separately, as every run of the disjunction automaton is either in $\overline{\mathcal{A}_{len}(X, N)}$ (that only shares X) or in $\overline{\mathcal{A}_{dec}(M, N)}$ (that only shares M).

Table 1 shows the results of discharging the verification conditions through this process. The most complex case is $\widetilde{\mathsf{vc}}_2$ which is the largest formula with 8 variables. The complete set of verification conditions is shown in Section E of the Appendix.

Failing verification conditions. Automata-based validity checking can also detect *invalid* verification conditions by showing concrete counter-examples (assignments of values to variables that make the condition false). Formulas ice_1 and ice_2 are invalid verification conditions obtained by dropping disjuncts or not complementing them in $\widetilde{\mathsf{vc}}_1$ and $\widetilde{\mathsf{vc}}_2$. Table 1 shows that the experiments correctly reported non-emptiness.

Even in the cases where the complete intersection is infinite, rational constructions may still be useful to search on-the-fly for accepting states, with the algorithm stopping as soon as it has established that the intersection is not empty. We did a small experiment in this line with formula cat_0, asserting an incorrect property of sequence concatenation: $x \circ y = z \wedge \mathit{last}(z) = u \wedge \mathit{last}(y) = v \Rightarrow u = v$, which does not hold if y is the empty sequence. Building the intersection with zero delays is not guaranteed to be complete because antecedent and consequent share two variables u, v; however, it is sufficient to find a counter-example where y is the empty sequence (see Table 1).

6 Related Work

The study of multi-tape automata began with the classic work of Rabin and Scott [30]. In the 1960's, Rosenberg and others contributed to the characterization of these automata [12, 9]. Recent research has targeted a few open issues, such as the properties of synchronous automata [21] and the language equivalence problem for deterministic

multi-tape automata [17]. See [14] for a detailed survey of multi-tape automata, and [7] for a historical perspective.

Khoussainov and Nerode [25] introduced a framework for the presentation of first-order structures based on multi-tape automata; while [25] also defines asynchronous automata, all its results target synchronous automata—and so did most of the research in this line (e.g., [2, 31, 22, 23]). To our knowledge, there exist only a few applications that use asynchronous multi-tape automata. Motivated by applications in computational linguistic, [6] discusses composition algorithms for weighted multi-tape automata. Our intersection algorithm (Section 4.1) shares with [6] the idea of accumulating delays in states; on the other hand, [6] expresses intersection as the combination of simpler composition operations, and targets weighted automata with *bounded* delays—a syntactic restriction that guarantees that reading heads are synchronized—suitable for the applications of [6] but not for the program verification examples of Section 5. Another application is reasoning about databases of strings (typically representing DNA sequences), for which multi-tape transducers have been used [15].

Much recent research targeted the invention of decision procedures for expressive first-order fragments useful in reasoning about functional properties of programs. Interpreted theories supporting operations on words, such as some of the examples in the present papers, include theories of arrays [3, 16], strings [26], multi-sets [27], lists [35], and sequences [13]. All these contributions (with the exception of [16]) use logic-based techniques, but automata-theoretic techniques are ubiquitous in other areas of verification—most noticeably, model-checking [34]. The present paper has suggested another domain where automata-theoretic techniques can be useful.

Acknowledgements. Thanks to Stéphane Demri for suggesting looking into automatic theories during a chat at ATVA 2010; and to Cristiano Calcagno for stimulating discussions. Many thanks to the reviewers of several conferences, and in particular to the anonymous Reviewer 1 of the 9th International Computer Science Symposium in Russia (CSR 2014), who pointed out—in an extremely detailed and insightful review of a previous version of this paper—some non-trivial errors about the completeness analysis for the algorithm of Section 4.

References

[1] Jean Berstel. *Transductions and Context-Free Languages*. Teubner-Verlag, 1979. Available at `http://goo.gl/WnDppd`.

[2] Achim Blumensath and Erich Grädel. Automatic structures. In *LICS*, pages 51–62. IEEE, 2000.

[3] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[4] J. Richard Büchi. Weak second-order arithmetic and finite automata. *Zeit. Math. Logik und Grundl. Math.*, 6:66–92, 1960.

[5] J. Richard Büchi. On a decision method in restricted second order arithmetic. In *Proceedings of the 1960 International Congress on Logic, Method, and Philosophy of Science*, pages 1–12. Stanford University Oress, 1962.

[6] Jean-Marc Champarnaud, Franck Guingne, André Kempe, and Florent Nicart. Algorithms for the join and auto-intersection of multi-tape weighted finite-state machines. *IJFCS*, 19(2):453–476, 2008.

[7] Christian Choffrut. Relations over words an logic: A chronology. *Bulletin of the EATCS*, (89):159–163, June 2006.

[8] Samuel Eilenberg. *Automata, languages, and machines*, volume 59A of *Pure and Applied Mathematics*. Academic Press, 1974.

[9] Samuel Eilenberg, C. C. Elgot, and John C. Shepherdson. Sets recognized by n-tape automata. *Journal of Algebra*, 13:447–464, 1969.

[10] C. Elgot. Decision problems of finite-automata design and related arithmetics. *Trans. Americ. Math. Soc.*, 98:21–51, 1961.

[11] Javier Esparza, Pierre Ganty, and Rupak Majumdar. A perfect model for bounded verification. In *LICS*, pages 285–294, 2012.

[12] Patrick C. Fischer and Arnold L. Rosenberg. Multitape one-way nonwriting automata. *Journal of Computer and System Sciences*, 2(1):88–101, 1968.

[13] Carlo A. Furia. What's decidable about sequences? In *ATVA*, volume 6252 of *LNCS*, pages 128–142. Springer, 2010.

[14] Carlo A. Furia. A survey of multi-tape automata. `http://arxiv.org/abs/1205.0178`, May 2012.

[15] Gösta Grahne, Matti Nykänen, and Esko Ukkonen. Reasoning about strings in databases. *JCSS*, 59(1):116–162, 1999.

[16] Peter Habermehl, Radu Iosif, and Tomás Vojnar. A logic of singly indexed arrays. In *LPAR*, volume 5330 of *LNCS*, pages 558–573. Springer, 2008.

[17] Tero Harju and Juhani Karhumäki. The equivalence problem of multitape finite automata. *Theoretical Computer Science*, 78(2):347–355, 1991.

[18] Juris Hartmanis. Context-free languages and Turing machine computations. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 42–51, 1967.

[19] Markus Holzer, Martin Kutrib, and Andreas Malcher. Multi-head finite automata: Characterizations, concepts and open problems. In *Workshop on The Complexity of Simple Programs*, volume 1 of *EPTCS*, pages 93–107, 2009.

[20] Oscar H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM*, 25(1):116–133, 1978.

[21] Oscar H. Ibarra and Nicholas Q. Tran. On synchronized multitape and multihead automata. In *DCFS*, volume 6808 of *LNCS*, pages 184–197. Springer, 2011.

[22] Oscar H. Ibarra and Nicholas Q. Trân. How to synchronize the heads of a multitape automaton. In *CIAA*, volume 7381 of *LNCS*, pages 192–204. Springer, 2012.

[23] Oscar H. Ibarra and Nicholas Q. Trân. On synchronized multi-tape and multihead automata. *Theor. Comput. Sci.*, 449:74–84, 2012.

[24] Andre Kempe, Franck Guingne, and Florent Nicart. Algorithms for weighted multi-tape automata. Technical Report 031, XRCE, 2004.

[25] Bakhadyr Khoussainov and Anil Nerode. Automatic presentations of structures. In *LCC*, volume 960 of *LNCS*, pages 367–392. Springer, 1995.

[26] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: a solver for string constraints. In *ISSTA*, pages 105–116. ACM, 2009.

[27] Viktor Kuncak, Ruzica Piskac, Philippe Suter, and Thomas Wies. Building a calculus of data structures. In *VMCAI*, volume 5944 of *LNCS*, pages 26–44. Springer, 2010.

[28] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM TOPLAS*, 1(2):245–257, 1979.

[29] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories. *JACM*, 53(6):937–977, 2006.

[30] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 39(2):114–125, 1959.

[31] Sasha Rubin. Automata presenting structures: A survey of the finite string case. *The Bulletin of Symbolic Logic*, 14(2):169–209, 2008.

[32] Jacques Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[33] B. Trakhtenbrot. The synthesis of logical nets whose operators are described in terms of one-place predicate calculus. *Doklady Akad. Nauk SSSR*, 118(4):646–649, 1958.

[34] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *LICS*, pages 332–344. IEEE, 1986.

[35] Thomas Wies, Marco Muñiz, and Viktor Kuncak. Deciding functional lists with sublist sets. In *VSTTE*, volume 7152 of *LNCS*, pages 66–81, 2012.

A Multi-Tape Automata: Negative Results (Section 3)

While Theorem 4 subsumes the undecidability of the rational intersection problem, we can give independent proofs of two variants of the problem. The first one uses a reduction from Post's correspondence problem; the second one, given later, a reduction from the disjointness problem for multi-tape automata.

Theorem 15. *The rational intersection problem is undecidable.*

Proof. We prove undecidability by reduction from Post's correspondence problem (PCP): given a finite set $\{\langle x_1, y_1\rangle, \ldots, \langle x_m, y_m\rangle\}$, of 2-words over Σ (with $|\Sigma| \geq 2$) determine if there exists a sequence $i_1, i_2, \ldots, i_k$ of indices from $1, \ldots, m$ (possibly with repetitions) such that $x_{i_1} x_{i_2} \cdots x_{i_k} = y_{i_1} y_{i_2} \cdots y_{i_k}$.

Given an instance of PCP, define $X = \{x_1, \ldots, x_m\}$ and $Y = \{y_1, \ldots, y_m\}$. Assume, without loss of generality, that the symbols $1, \ldots, m$ and a marker $\#$ are not in Σ. Consider the two languages L_1, L_2 defined as:

$$L_1 = \{\langle i_1 \cdots i_\alpha, x_{i_1} \cdots x_{i_\alpha}, y_{i_1} \cdots y_{i_\alpha} \# \widetilde{x}\rangle \mid \alpha \geq 0 \text{ and } \widetilde{x} \in X^*\},$$
$$L_2 = \{\langle j_1 \cdots j_\beta, y_{j_1} \cdots y_{j_\beta}, \widetilde{y} \# x_{j_1} \cdots x_{j_\beta}\rangle \mid \beta \geq 0 \text{ and } \widetilde{y} \in Y^*\},$$

where the i_k's and j_k's are indices from $1, \ldots, m$. It is not difficult to see that L_1 and L_2 are rational languages. An automaton accepting L_1 works as follows: for each element i_k on the first tape, it checks that the corresponding x_{i_k} and y_{i_k} respectively appear on the second and third tape; finally, it checks that only elements in X appear after the $\#$ on the third tape. An automaton for L_2 can follow a similar logic.

The intersection $L = L_1 \cap L_2$ consists of all words of the form

$$\langle (i_1 \cdots i_k)^n, x_{i_1} \cdots x_{i_k}, (y_{i_1} \cdots y_{i_k})^n \#(x_{i_1} \cdots x_{i_k})^n\rangle$$

for $n \geq 0$, such that $i_1, \ldots, i_k$ is a solution of the PCP. If the PCP has no solution, then L is the singleton $\langle \epsilon, \epsilon, \epsilon\rangle$ which is clearly rational; conversely, if the PCP has no solution, L contains infinitely many words but is not rational, because its projection onto the third component has the form $u^n \# v^n$, which is non-regular [14]. □

The proof of Theorem 15 uses 3-words, which implies that the result carries over to any number of tapes $n \geq 3$; is it possible to generalize to $n \geq 2$? It seems difficult to simultaneously express the PCP solution requirements and the non-regularity of one of the components. However, a slightly weaker (but practically as useful) undecidability result for $n \geq 2$ tapes follows easily from the undecidability [14] of the disjointness problem for rational languages (that is, determining whether the intersection $L_1 \cap L_2$ of two rational languages is empty). We can prove that the following problem P is undecidable: *constructively* determine whether the intersection $L_1 \cap L_2$ of two rational languages L_1, L_2 is rational; "constructively" refers to the fact that we require that, if $L_1 \cap L_2$ is rational, then we can build an automaton $A_{1,2}$ such that $\mathcal{L}(A_{1,2}) = L_1 \cap L_2$. Assume, to the contrary, that P is decidable. Then, we have a decision procedure for the disjointness problem: if $L_1 \cap L_2$ is rational, construct and automaton $A_{1,2}$ that accepts it, and test $A_{1,2}$ for emptiness; otherwise, $L_1 \cap L_2$ is not rational, and hence certainly $L_1 \cap L_2 \neq \emptyset$.

B Under-Approximation of Intersection (Section 4.1)

```
1  async_next (D, q): SET[⟨q', h_1, ..., h_t⟩]
2      -- q is always reachable from itself
3      Result := {⟨q, ϵ, ..., ϵ⟩}
4      -- for every tape other than q's
5      for each t_i ∈ {t_1^D, ..., t_t^D} \ τ^D(q) do
6          P := all shortest paths p from q to some q̄ such that:
7                  τ^D(q̄) = t_i and no state q̃ with τ^D(q̃) = t_i appears in p before q̄
8          -- each element in P is a sequence of transitions
9          for each e_1 ··· e_m ∈ P do
10             h_1, ..., h_t := ϵ
11             -- each transition is a triple (source, input, target)
12             for each (q_1, σ, q_2) ∈ e_1 ··· e_m do
13                 -- add the transition to the sequence corresponding
14                 -- to its source's tape
15                 h_{τ^D(q_1)} := h_{τ^D(q_1)} + (q_1, σ, q_2)
16             -- q_2(e_m) is the target state of the last transition e_m
17             Result := Result ∪⟨q_2(e_m), h_1, ..., h_t⟩
18
19 new_states (P: SET[⟨p, h_1, ..., h_m⟩], Q: SET[⟨q, k_1, ..., k_n⟩]): S
20     S := ∅
21     for each ⟨p, h_1, ..., h_m⟩ ∈ P, ⟨q, k_1, ..., k_n⟩ ∈ Q do
22         -- if delays on synchronized tapes are consistent
23         if ∀i ∈ T^A ∩ T^B : cons(h_i, k_i) then
24             for each t ∈ T do S := S ∪ {⟨p, q, t, h_1, ..., h_m, k_1, ..., k_n⟩} end
25     -- Here Q denotes C's set of states, not the input argument
26     for each r ∈ S do if r ∉ Q then s.push (r) end
27
28 compose_transition (P: SET[(p, h_1, ..., h_m)], Q: SET[(q, k_1, ..., k_n)],
29                     d: (h_1, ..., h_m, k_1, ..., k_n), σ, r)
30     J_A := {(p, h_1 h'_1, ..., h_m h'_m) | (p, h'_1, ..., h'_m) ∈ P}
31     J_B := {(q, k_1 k'_1, ..., k_n k'_n) | (q, k'_1, ..., k'_n) ∈ Q}
32     S := new_states (J_A, J_B)
33     for each r' ∈ S do δ := δ ∪ {r, σ, r'} end
```

Figure 4: Routines *async_next*, *new_states*, *compose_transition*.

```
1  intersect ( max_states , max_delay)
2    Q := ∅ ; s := ∅
3    −− Initially reachable states
4    J_A := ⋃_{i∈Q_0^A} async_next (A, i) ; J_B := ⋃_{i∈Q_0^B} async_next (B, i)
5    S := new_states (J_A, J_B) ; Q_0 := S
6    until s = ∅ or |Q| ≥ max_states loop
7      r := (q_a, q_b, t, h_1, ..., h_m, k_1, ..., k_n) = s.pop
8      if ∀d ∈ {h_1, ..., k_n} : |d| ≤ max_delay then Q := Q ∪ {r} else continue
9      if t ∈ T^A ∩ T^B then −− event on shared tape
10       if h_t = (u_a, σ, u'_a)h̄_t and k_t = (u_b, σ, u'_b)k̄_t then
11         −− delayed transition on both A and B
12         P := async_next (A, q_a) ; Q := async_next (B, q_b)
13         d := (h_1, ..., h̄_t, ..., h_m, k_1, ..., k̄_t, ..., k_n)
14         compose_transition (P, Q, d, σ, r)
15       elseif h_t = (u_a, σ, u'_a)h̄_t and k_t = ϵ then
16         −− delayed transition on A, normal transition on B
17         P := async_next (A, q_a)
18         Q := { async_next (B, q'_b) | (q_b, σ_b, q'_b) ∈ δ^B ∧ σ = σ_b ∧ τ^B(q_b) = t}
19         d := (h_1, ..., h̄_t, ..., h_m, k_1, ..., k_n)
20         compose_transition (P, Q, d, σ, r)
21       elseif h_t = ϵ and k_t = (u_b, σ, u'_b)k̄_t then
22         −− delayed transition on B, normal transition on A
23         ...
24       elseif h_t = k_t = ϵ then
25         −− normal transition on both A and B
26         for each σ ∈ Σ do
27           P := { async_next (A, q'_a) | (q_a, σ_a, q'_a) ∈ δ^A ∧ σ_a = σ ∧ τ^A(q_a) = t}
28           Q := { async_next (B, q'_b) | (q_b, σ_b, q'_b) ∈ δ^B ∧ σ_b = σ ∧ τ^B(q_b) = t}
29           d := (h_1, ..., h_m, k_1, ..., ..., k_n)
30           compose_transition (P, Q, d, σ, r)
31     elseif t ∈ T^A \ T^B then −− event on A's non-shared tape
32       if h_t = (u_a, σ, u'_a)h̄_t then −− delayed transition on A, B stays
33         P := async_next (A, q_a) ; Q := {(q_b, ϵ, ..., ϵ)}
34         d := (h_1, ..., h̄_t, ..., h_m, k_1, ..., k_n)
35         compose_transition (P, Q, d, σ, r)
36       elseif h_t = ϵ then −− normal transition on A, B stays
37           Q := {(q_b, ϵ, ..., ϵ)}
38           for each σ ∈ Σ do
39             P := { async_next (A, q'_a) | (q_a, σ_a, q'_a) ∈ δ^A ∧ σ_a = σ ∧ τ^A(q_a) = t}
40             d := (h_1, ..., h_m, k_1, ..., k_n)
41             compose_transition (P, Q, d, σ, r)
42     elseif t ∈ T^B \ T^A then −− event on B's non-shared tape
43       ...
```

Figure 5: Routine *intersect* .

C Correctness and Completeness (Section 4.2)

Lemma 16 (Pumping lemma)**.** *Let L be an n-rational language. Then there exists an integer $N \geq 1$ such that every word $\langle x_1, \ldots, x_n \rangle \in L$ where $|x_1| + \cdots + |x_n| \geq N$ can be written as $\langle p_1 q_1 r_1, \ldots, p_n q_n r_n \rangle$, with $q_k \neq \epsilon$ for at least one $1 \leq k \leq n$, and $\langle p_1 q_1^m r_1, \ldots, p_n q_n^m r_n \rangle$ is in L for every $m \in \mathbb{N}$.*

Proof. Let A_L be an automaton accepting L; then, the number of states of A_L is the pumping length $N = M$. Consider a word $w = \langle x_1, \ldots, x_n \rangle \in L$ with length $|x_1| + \cdots + |x_n| \geq N$. A computation accepting w visits $N + 1$ states of A_L; by the pigeonhole principle, there exists a state s in the sequence which is visited twice. The sequence of symbols read in the transitions that go from the first to the second visit of s determines an n-word $\langle q_1, \ldots, q_n \rangle$ with at least one $q_k \neq \epsilon$. Looping an arbitrary number of times over the sequence that starts and ends on s determines words that are all accepted by A_L, and hence belong to L. □

D Asynchronous Rational Theories

The *signature* $S_\Theta = C \cup F \cup R$ of a first-order theory Θ is a set of constant C, function F, and predicate R symbols. A *quantifier-free formula* of Θ is built from constant, function, and predicate symbols of S_Θ, as well as variables $x, y, z, \ldots$ and logical connectives $\Rightarrow, \vee, \wedge, \neg$. An *interpretation*[3] I_Θ assigns constants, functions, and predicates over a domain D to each element of C, F, and R. It is customary that R include an equality symbol $=$ with its natural interpretation. Then, assume without loss of generality that Θ is *relational*, that is $F = \emptyset$; to this end, introduce a $(m+1)$-ary predicate R_f for every m-ary function f such that $R_f(x_1, \ldots, x_m, y)$ holds iff $f(x_1, \ldots, x_m) = y$. A *model* M of a formula F of Θ is an assignment of values to the variables in F that is consistent with I_Θ and makes the formula evaluate to true; write $M \models F$ to denote that M is a model of F. The set of all models of a formula F under an interpretation I_Θ is denoted by $[\![F]\!]_{I_\Theta}$. F is *satisfiable* in the interpretation I_Θ if $[\![F]\!]_{I_\Theta} \neq \emptyset$; it is *valid* if $[\![F]\!]_{I_\Theta}$ contains all variable assignments that are consistent with I_Θ.

Similar to automatic presentations, a *rational presentation* of a first-order theory Θ consists of:

1. A finite alphabet Σ;
2. A surjective mapping $\nu : S \to D$, with S a regular subset of Σ^*, that defines an encoding of elements of the domain D in words over Σ;
3. A 2-tape automaton $\mathcal{A}_{\mathsf{eq}}$ whose language is the set of 2-words $\langle x, y \rangle \in (\Sigma^*)^2$ such that $\nu(x) = \nu(y)$;
4. For each m-ary relation $R_m \in R$, an m-tape automaton $\mathcal{A}_{R_m}$ whose language is the set of m-words $\langle x_1, \ldots, x_m \rangle \in (\Sigma^*)^m$ such that $R_m(\nu(x_1), \ldots, \nu(x_m))$ holds.

A first-order theory with rational presentation is called *rational theory*. If the automata of the presentation are deterministic (resp. synchronous, asynchronous) the theory is also called deterministic (resp. synchronous, asynchronous).

[3] For simplicity, we do not discuss how to *axiomatize* the semantics of interpreted items.

Example 17 (Rational theory of concatenation). The theory of concatenation over $\{a, b\}^*$ is the first-order theory with constant ϵ (the empty sequence), sequence equality $=$, and concatenation predicate $R_\circ$ such that $R_\circ(x, y, z)$ holds iff z is the concatenation of x and y. This theory is asynchronous rational, with $\Sigma = \{a, b\}$, ν the identity function, $\mathcal{A}_{\mathsf{eq}}$ as in Figure 1, and $\mathcal{A}_{R_\circ}$ as in Figure 2.

Consider a quantifier-free formula F of a rational theory Θ. To decide if F is satisfiable we can proceed as follows. First, build an automaton $\mathcal{A}_F$ that recognizes exactly the models of F. This is done by composing the elementary automata of the theory according to the propositional structure of F; namely, for sub-formulas G, H, negation $\neg G$ corresponds to complement $\overline{\mathcal{A}_G}$, disjunction $G \vee H$ corresponds to union $\mathcal{A}_G \cup \mathcal{A}_H$, and conjunction $G \wedge H$ corresponds to intersection $\mathcal{A}_G \cap \mathcal{A}_H$. To verify if F is valid, test whether $\mathcal{A}_{\neg F} = \overline{\mathcal{A}_F}$ is empty: $\mathcal{L}(\mathcal{A}_{\neg F})$ is empty iff $\neg F$ is unsatisfiable iff F is valid.

We can apply this procedure only when the automaton $\mathcal{A}_F$ is effectively constructible, which is not always the case for asynchronous rational theories because asynchronous automata lack some closure properties (see Section 2.2)—intersection, in particular. The following section, however, shows some non-trivial examples of formulas whose rational presentation falls under the criterion of Corollary 11 (and whose components to be complemented are deterministic), hence we can decide their validity by means of automata constructions.

E Implementation and Experiments (Section 5)

$$
\begin{aligned}
\mathsf{vc}_0 &\equiv |y| \geq m \wedge y = \mathit{rest}(x) \;\Rightarrow\; |x| \geq n \wedge m = n - 1 \\
\mathsf{vc}_1 &\equiv |y| \geq m \wedge y = \mathit{rest}(x) \;\Rightarrow\; |x| \geq n \wedge m = n - 1 \\
\mathsf{cat}_0 &\equiv x \circ y = z \wedge \mathit{last}(z) = u \wedge \mathit{last}(y) = v \;\Rightarrow\; u = v \\
\mathsf{ice}_1 &\equiv |y| \geq m \wedge y = \mathit{rest}(x) \;\Rightarrow\; |x| < n \\
\mathsf{ice}_2 &\equiv |\mathbf{Result}| = u \wedge u = |y| - m \wedge y = \mathit{rest}(x) \;\Rightarrow\; |\mathbf{Result}| = v
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{vc}_2 \equiv\; & |\mathbf{Result}| = u \wedge u = |y| - m \wedge y = \mathit{rest}(x) \\
& \Rightarrow\; |\mathbf{Result}| = v \wedge v = |x| - n \wedge m = n - 1 \wedge |x| = n
\end{aligned}
$$