From Two-Way to One-Way Finite Automata—Three Regular Expression-Based Methods

Mans Hulden(✉)

University of Colorado Boulder, Boulder, USA
mans.hulden@colorado.edu

Abstract. We describe three regular expression-based methods to characterize as a regular language the language defined by a two-way automaton. The construction methods yield relatively simple techniques to directly construct one-way automata that simulate the behavior of two-way automata. The approaches also offer conceptually uncomplicated alternative equivalence proofs of two-way automata and one-way automata, particularly in the deterministic case.

1 Introduction

An early result in automata theory is that of the equivalence of two-way and one-way finite automata. Rabin and Scott [13] outlined a proof of this equivalence by analyzing the so-called *crossing sequences* that occur during the acceptance of a string by a two-way automaton. This proof was slightly simplified by Shepherdson [15]. Later, Vardi [16] has shown equivalence through a subset construction that is used to characterize the complement of the language accepted by a two-way nondeterministic automaton. The *crossing sequences* proof and construction is more involved if one wants to include non-deterministic two-way automata (2NFA) in addition to deterministic ones (2DFA). While these methods in principle allow for the construction of the equivalent one-way automaton, the calculations involved are rather complex. In the crossing sequences approach, this calls for the analysis of the possible crossing sequences for possible prefixes of strings, and the complement construction requires laborious bookkeeping. While two-way automata have been analyzed intensely, especially as regards theoretical size bounds in conversions from two-way to one-way automata and state complexity of operations [3,4,7,8,10–12], practical conversion algorithms have received less attention. The intricacies involved in previous conversion methods may also be reflected in the paucity of actual implementations for converting arbitrary two-way to one-way automata.

In this paper, we describe a new method to characterize the language accepted by some two-way automata (2DFA/2NFA). Our approach is very direct: we model the set of accepting computation sequences of a two-way automaton as strings in a regular language that includes annotations about the behavior of a two-way automaton. Following this, a homomorphism is applied to delete the

F. Drewes (Ed.): CIAA 2015, LNCS 9223, pp. 176–187, 2015.
DOI: 10.1007/978-3-319-22360-5_15

annotations, yielding the set of actual strings accepted by a two-way automaton. Central to the modeling is a compact simulation of the accepting sequences of a given 2DFA/2NFA. Apart from providing a construction method, the approach also gives an alternative to the equivalence proofs of one-way and two-way automata customarily provided in most textbooks on automata (e.g. [5,9,14]).

2 Notation and Definitions

We define a 2NFA M the standard way as a 5-tuple $(\Sigma, Q, Q_0, \delta, F)$, where Σ denotes the alphabet, Q the finite set of states, a set of initial states $Q_0 \subseteq Q$. The transition function is denoted by $\delta: \ Q \times \Sigma \to 2^{Q \times \{L,S,R\}}$, and the set of final states by $F \subseteq Q$. If $|\delta(q,a)| \leq 1$ for all $q \in Q$ and all $a \in \Sigma$ and $|Q_0| = 1$, the automaton is deterministic. We say M accepts a string w whenever there is a transition path in M from an initial state to a final state such that M at each state moves its read head in the direction specified by the transition function (to the left (L), right (R), or staying (S)) and ends up at the right edge of w. Formally, we can define acceptance as a specific series of configurations using strings of the format $\Sigma^* Q \Sigma^*$. A string wqx describes the circumstance where the input string is wx and q is the current state when M is scanning the first symbol of x. We say $wpax \vdash waqx$ is a permitted change of configuration if $(q,R) \in \delta(p,a)$, as is $wbpax \vdash wpbax$ if $(q,L) \in \delta(p,a)$ and $|b| = 1$, and $wpax \vdash wqax$ if $(q,S) \in \delta(p,a)$. We say M accepts w if there exists some choice of a sequence of configuration changes such that $pw \vdash \ldots \vdash wq$, where $p \in Q_0$ and $q \in F$.

In the following, we also make use of extended regular expressions to denote operations on regular languages. We will make use of the following standard notational devices in regular expressions: a is a single symbol drawn from an alphabet, ϵ is the empty string, and $\emptyset$ the empty language. Additionally we use the following operators: $L_1 L_2$ (denoting concatenation), $L_1 \cup L_2$ (union), $L_1 \cap L_2$ (intersection), $\neg\ L_1$ (complement), $L_1 - L_2$ (subtraction), L_1^* (Kleene closure) and L_1^+ (Kleene plus). We also make use of the fact that regular languages are closed under homomorphisms $h\colon \Gamma^* \to \Sigma^*$.

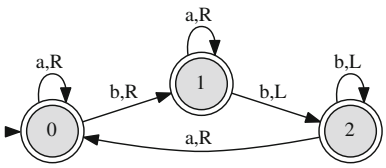

Fig. 1. Example (deterministic) 2DFA M with initial state 0. The language described is $(a|ba)^*(b|\epsilon)$.

3 Overview

The general idea behind the three methods given below is to simulate accepting or nonaccepting move sequences of a two-way automaton by a specific string representation, the correctness of which is locally checkable, that is, verifiable by a one-way automaton. The verification can be modeled through regular expressions.

We model the set of accepting computations of a two-way automaton M operating over the alphabet Σ as a regular set of strings over an auxiliary alphabet Γ and the original alphabet Σ. In particular, strings in this language consist of symbols from Σ (the alphabet of M), interspersed with auxiliary substrings that characterize relevant movements of the 2DFA/2NFA. The auxiliary alphabet Γ consists of symbols representing states in Q as well as symbols corresponding to possible moves $\{L, S, R, C\}$, i.e. $\Gamma = \{q_0, \ldots, q_n, L, S, R, C\}$. The symbol C is a move that models a crash (only used in the second construction method). We enforce that these auxiliary substrings always come in triplet-size chunks where the three-symbol sequence encodes a move by M in the order (1) the source state, (2) the target state, and (3) the direction of movement (left, right, stay).

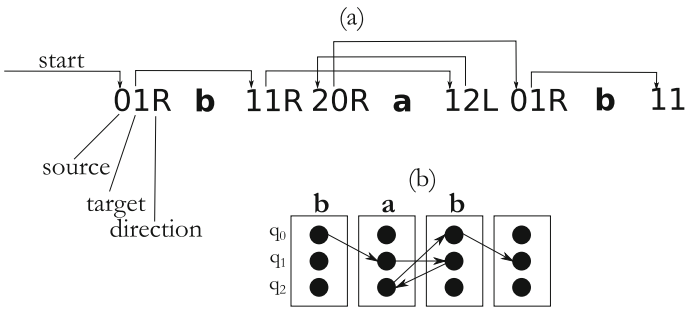

Fig. 2. Example of an accepting sequence for the 2DFA in Fig. 1 in our string encoding (a). The arrows show which source-target-direction triplets license the presence of others in the first construction method. The figure (b) illustrates the computation of the 2DFA the the string effectively encodes.

Figure 2 shows such a sequence in our encoding. The intuition behind the constructions is that, out of all possible sequences over $(\Gamma \cup \Sigma)^*$, we want to characterize all and only those that fulfill criteria that correspond to acceptance or non-acceptance of a string over Σ by some two-way automaton M. We do so by providing additional local constraints on these strings to simulate the legal movements in a 2DFA/2NFA.

In such an string, we say that any subsequence of symbols from Γ is in the same *position* as another subsequence from Γ so long as there are no intervening symbols from Σ between the two. For example, in Fig. 2, `11R` and `20R` are in the same *position*, while `20R` is in the position preceding `12L`.

4 Method 1: 2DFA to 1NFA/1DFA

In this construction method, which applies to deterministic two-way automata only, the idea is to declare a language over $(\Gamma \cup \Sigma)^*$ in such a way that the first Γ-triplet at the left edge corresponds to movement from an initial state, and that any other two-symbol sequences representing state pairs present at any position need to all be 'licensed' by some previous move from a previous position. We also require that all strings end in a qq sequence, reflecting a halt in a final state at the right edge of a string.

More formally, the conditions for well-formedness of a string w in our encoding for an accepting sequence by a 2DFA M can be specified as follows:

(1) The string w is of the form $(T^+a)^*\ T_{end}$, where T is any three-symbol sequence pqD representing transitions of M where p and q correspond to a valid transition $p \rightarrow q$ in M between states in Q using symbol a, and $D \in \{L, S, R\}$ denotes the corresponding direction of movement in M. T_{end} is a two-symbol string qq, where q is any final state in M.
(2) Additionally, when w contains a two-symbol sequence pq, then at least one the following holds:
  (i) p corresponds to the initial state Q_0 in the definition of M and is at the left edge of w.
  (ii) there is a substring pS in the same *position* in w.
  (iii) there is a substring pL in the following *position* in w.
  (iv) there is a substring pR in the preceding *position* in w.

For example, in Fig. 2, the first occurrence of `01` is permitted since it occurs at the left edge of the word and 0 is an initial state in M (by condition (2i)), while the second occurrence is permitted because the preceding position contains `0R` (by condition (2iv)). Likewise, `20` is permitted because it is followed by `2L` in the following position (condition (2iii)), etc.

Note that the constraints above say nothing about the order in which the triplets themselves are permitted. There may also be arbitrary repetitions of the same triplets within a position, so long as their presence is allowed by conditions (1) and (2i-iv); these two questions are irrelevant for purposes of the encoding. By the same token, the encoding says nothing about the specific order in which the moves actually occur when M accepts a word w—only that each substring representing a move or halting be *licensed* by some other substring representing another move, save for the base case of the initial state symbol, which is always allowed as the first symbol in the string.

The sets of strings that satisfy (1) and (2i-iv) is regular, and an automaton that accepts the sets is easily constructed (see below for precise regular expressions).

If we call the language where all strings fulfill condition (1) L_{base} and the language where all strings fulfill conditions (2i-iv) $L_{license}$, we have, for a homomorphism that deletes symbols in Γ, $h(a) = \epsilon$ for all a in Γ:

$$L_1 = h(L_{base} \cap L_{license}) \tag{1}$$

Theorem 1. *M accepts a word w iff $w \in L_1$.*

Proof. First, consider the case where M accepts w. By induction on the number of steps in the computation of M, we see that $(L_{base} \cap L_{license})$ then contains a string ending in qq, for some final state q, and hence that $w \in L_1$. In the other direction: all strings in $(L_{base} \cap L_{license})$ end in the sequence qq (by definition). Now, such a string qq is only permitted by the presence of some other move encoding at some position which in turn is permitted by some previous move, etc. (by 2ii-iv), forming a sequence of position-state pairs $(p_1, q_1) \leftarrow (p_2, q_2) \ldots \leftarrow (p_k, q_k)$, tracing the computation backward. In such a sequence no position-state pair may repeat, since—by assumption—the two-way automaton is deterministic. In other words, repetition of a state-position pair (p_i, q_j) with symbol a at position p_i would imply that $|\delta(q_i, a)| > 1$. Since there are only $|Q|n$ possible unique position-state pairs for a string of length n, this sequence must terminate, which is only possible by (2i). Hence, the final substring qq must ultimately be licensed by the initial state and the sequence describes a legitimate accepting path in M. □

5 Construction Details

The construction described above can be immediately implemented for an arbitrary 2DFA M.

We use the alphabets:

- Σ (of M)
- $\Gamma = \{q_0, \ldots, q_n, L, S, R\}$
- $\Delta = \Sigma \cup \Gamma$ (as shorthand)

The language L_{base}, which enforces the general structure of the string, can be defined as:

$$L_{base} = \left(T^+_{a_1} a_1 \cup \ldots \cup T^+_{a_n} a_n\right)^* L_{end} \tag{2}$$

for symbols $a_1, \ldots, a_n \in Q$. Here, T_{a_i} contains all three-symbol strings pqD corresponding to M's transitions $p \rightarrow q$ reading symbol a_i and moving in the direction D. Formally, $pqD \in T_{a_i}$ iff $(q, D) \in \delta(p, a_i)$.

L_{end} is the set of two-symbol strings qq, where q is a halting state in M. That is, $qq \in L_{end}$ iff $q \in F$.

To describe $L_{license}$, we make use of the regular expression idiom

$$\neg(\neg S \ T \ \neg U)$$

to convey the idea that strings drawn from the set T must either be preceded by some string from the set S or followed by some string from U. This allows us to express the relevant parts of (2i-iv) above concisely. We assume that we have a set of single symbols $\mathcal{Q} \subset \Gamma$, representing the states of M. Now conditions (2i-iv) for some state q are expressed as:

$$L_{licence_q} = \neg(\neg(\mathcal{Z} \cup \underbrace{\Delta^* q S \Gamma^*}_{\text{'stay' move in same position to the left}} \cup \underbrace{\Delta^* q R \Gamma^* \Sigma \Gamma^*)}_{\text{'right' move in previous position}} q\mathcal{Q} \neg(\underbrace{\Gamma^* \Sigma \Gamma^* q L \Delta^*}_{\text{'left' move in following position}} \cup \underbrace{\Gamma^* q S \Delta^*)}_{\text{'stay' move in same position to the right}}) \tag{3}$$

Here,

$$\mathcal{Z} = \begin{cases} \epsilon & \text{if } q \in Q_0 \\ \emptyset & \text{otherwise} \end{cases} \tag{4}$$

In other words, a sequence $q\mathcal{Q}$ (the symbol for state q followed by any other state symbol), must be preceded by right move in the previous position or a stay move in the same position with target state q, or followed by a stay move in the same position or a left move in the following position with target state q. Note that the 'stay' move is brought up twice in the expression because within a position, the moves listed are in arbitrary order, and we must therefore account for the possibility that an S-move can occur either to the left or the right of the relevant state symbol q. Additionally, symbols representing an initial state may always occur initially in the string (modeled by $\mathcal{Z}$).

For a 2DFA M with states $Q = q_0, \ldots, q_n$, the language L_1 is then

$$h(L_{base} \cap L_{license0} \cap \ldots \cap L_{licensen}) \tag{5}$$

6 Method 2: 2NFA to 1DFA by Complement Construction

The previous method cannot be used to convert a nondeterministic two-way automaton to a 1DFA as is seen from the correctness argument which hinges on the two-way automaton being deterministic. However, we can use the same string encoding to create a similar setup where we model as a regular set all and only the words the are rejected by some 2NFA, i.e. the complement of acceptance.

The only minor change to the encoding used previously is in the auxiliary alphabet, which now becomes $\Gamma = \{q_0, \ldots, q_n, L, S, R, C\}$. the symbols L, S, R are as before, and the symbol C is an extra arbitrary symbol we use to denote a 'crash' configuration—either a state that has no outgoing transitions with some symbol, or a nonfinal state at the right edge.

In this construction, the idea is to capture all possible failing paths as a regular language by (1) insisting that all initial states be present as source states in the first position of the string encoding, and (2) requiring that each move encoding be *followed* by another move encoding or a crash—note that this enforcement is different from method 1 where we *permitted* a state-pair in the string if it resulted from a legitimate previous move; here we *require* a subsequent move for any state-pair. In other words, the presence of each transition triplet pqD

requires the presence of all legal transition triplets qrD in the following, preceding, or the current positions (for moves right, left, stay). For any state q without an outgoing transition with the symbol in that position, this requirement will also be satisfied by a triplet qqC. Such crash triplets do not themselves require a follow-up move. This encoding ensures that if a valid path through a 2NFA M exists, that path cannot be encoded in our string representation, since we lack a halting configuration. Conversely, the encoding contains all invalid paths.

Here, we have the following requirements on the well-formedness of a string w in the encoding:

(1) The string w is of the form $(T_1 \dots T_k a_i)^*\ T_{end}$, where the Ts are three-symbol transition sequences of the form $p_1 q_1 D_1, \dots, p_k q_k D_k$ corresponding to all transitions from p in M using symbol a, and moving to q, and $D \in \{L, S, R, C\}$ denotes the corresponding direction of movement in M. In case a state has no transition with a, ppC may be present. T_{end} is a three-symbol string qqC, where q is any nonfinal state in M. Also, the first position in string w contains all sequences pq where $p \in Q_0$ and some $q \in Q$.
(2) Additionally, when w contains a two-symbol sequence pD where $p \in Q$ (representing a state in Q) and $D \in \{L, S, R\}$, then the following holds:
   (i) if D is L there is a substring pq in the preceding *position* in w, where $q \in Q$, or pD is in the leftmost position.
   (ii) if D is R there is a substring pq in the following *position* in w, where $q \in Q$.
   (iii) if D is S there is a substring pq in the current *position* in w, where $q \in Q$.

Condition (1) enforces the general well-formedness of the strings, assuring that each symbol in Σ is surrounded by sequences of triplets corresponding to valid transitions in M. Also, if one triplet pqD is present, all other possible outgoing transitions from p also need to be listed. It also sets up the base case that all initial states are represented at the first position. Conditions (1) and (2) also ensure that any transition modeled is followed by all possible outgoing transitions from the target state, or, in the case that the target state has no valid outgoing transitions and is nonfinal, that fact is marked by a ppC, where p is the state with no outgoing transitions with the symbol at hand.

Again, the conditions (1)–(3) are all local and easily testable by DFA(s) and hence the set of strings that fulfill all conditions is a regular set.

We now claim, using the same pattern as before, that the language where all strings fulfill condition (1) L_{base} and the language where all strings fulfill conditions (2i-iii) $L_{license}$, we have for a homomorphism $h(a) = \epsilon$ for all a in Γ:

$$L_2 = \Sigma^* - h(L_{base} \cap L_{license}) \tag{6}$$

Theorem 2. *A 2NFA M accepts a word w iff $w \in L_2$.*

Proof. Suppose M accepts w. Then the accepting path through M will be modeled by (1) and (2i-iii) in $L_{base} \cap L_{license}$ with the exception of the accepting move which is never permitted, and so w is not in $h(L_{base} \cap L_{license})$. M can

reject a word w if all paths in the computation eventually lack a transition for the symbol being read, end up at the right edge of a word in a nonfinal state, or try to transition left at the left edge. All such configurations are accepted by $L_{base} \cap L_{license}$, and hence w is in the language $h(L_{base} \cap L_{license})$. □

Details of the actual construction are very similar to that of the first method and are omitted here.

6.1 A Note on the Construction

This approach bears similarities to the method suggested by Vardi [16]. In that work, a type of subset construction is used that directly constructs the states in the complement language accepted by a 2NFA. That construction relies on the following lemma:

Lemma 1 (Vardi, 1989). *Let $M = (\Sigma, Q, Q_0, \delta, F)$ be a two-way automaton, and $w = a_0, \ldots, a_n$ be a word in Σ^*. M does not accept w if and only if there exists a sequence $T_0, \ldots, T_{n+1}$ of subsets of Q such that the following conditions hold:*

1. $Q_0 \subseteq T_0$
2. $T_{n+1} \cap F = \emptyset$
3. *for $0 \leq i \leq n$, if $q \in T_i$, $(q', k) \in \delta(q, a)$, and $i + k > 0$, then $q' \in T_{i+k}$*

It is assumed here that k is an integer $\{-1, 0, 1\}$ corresponding to the directions of movement in the transition function ($\{L, S, R\}$ in our notation).

One of the consequences of this more abstract construction is that it cannot directly be used to model the set of strings *accepted* by a 2NFA, and requires the complement construction.[1] Our regular language 2NFA-1DFA construction, however, can be modified to do precisely that which is alluded to in [16]; we present the details of this additional construction method below.

7 Method 3: 2NFA to 1DFA Directly

With the 2NFA-1DFA construction above, is it not possible to directly model the set of accepting sequences by a 2NFA M, instead of modeling the complement? That is, can one not combine the techniques in method 1 and method 2 and construct a language that contains the same triplets that mark transitions in such a way as to only contain valid computation sequences of M that end in a final state. This would mean, in addition to enforcing the overall format of the strings,

[1] "It may be tempting to think that it is easy to get a similar condition to acceptance of w by A. It seems that all we have to do is to change the second clause in [the lemma] to $T_{n+1} \cap F \neq \emptyset$. Unfortunately, this is not the case; to characterize acceptance we also have to demand that the T_i's be minimal. While the conditions in the lemma are local, and therefore checkable by a finite-state automaton, minimality is a global condition." [16], p. 3.

requiring that all well-formed strings have initial states represented at the left edge, and that each transition pqD 'require' that a subsequent transition from q be present in the appropriate position, except for a final qq, at the right edge. Additionally, any 'crash' configuration would of course not be in the language. The problem with such an idea is exactly what is touched upon in [16]—that one must also require that any accepting path be minimal. This is illustrated in Fig. 3. Here (a) exemplifies a nonminimal path that leads to acceptance since the path of computation from the left edge fulfills the criteria by ending in a loop. Additionally, there is a spurious transition triplet before y leading to acceptance. In (b) we see the corresponding minimal path induced by the same string, showing a case of non-acceptance.

Fig. 3. Illustration of a nonminimal path starting from the initial state with a spurious path which causes nonminimality, together with the corresponding minimal path.

The idea behind this third construction is to modify the construction so that only minimal paths are in the simulation. To do this, consider the language $L = L_{base} \cap L_{license}$ that contains strings over Σ (with interspersed path descriptions) if M accepts, but that also includes spurious nonminimal paths. Now, consider the homomorphism $h(a) = \epsilon$ for all a in Γ. Define an operation $insert(L)$: $\{y \mid x \in L \wedge h^{-1}(x) = y \wedge |x| < |y|\}$, i.e. the inverse homomorphism with the additional requirement that at least one symbol from Γ is inserted. If L is regular, so is obviously $insert(L)$. In practice, we model this by composition of the identity transducer for L with a transducer Ins (see Fig. 4) that inserts at least one symbol from Γ, and reconvert to an automaton by taking the output projection: $proj_2(Id(L) \circ Ins(\Gamma))$.

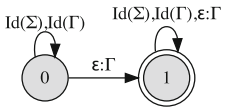

Fig. 4. Illustration of insertion transducer Ins.

The insert-operation can be used to remove the nonminimal paths in some language L that represents computations in the string encoding, and we can define the set of accepting strings by a 2NFA directly as:

$$L_3 = h(L - Insert(L)) \tag{7}$$

Taking advantage of this, we can define the conditions for any w as follows, and then use our ability to enforce minimality.

Here, we have the following requirements on the well-formedness of a string w in the encoding:

(1) The string w is of the form $(T^*a)^*\ (T_{end} \cup \epsilon)$, where T is a set of three-symbol transition sequences of the form pqD corresponding to *some* transitions p in M using symbol a, and moving to q, and $D \in \{L, S, R\}$ denotes the corresponding direction of movement in M. T_{end} is a set of two-symbol strings qq, where q is any final state in M. Also, the first position in string w contains a sequence pq for all $p \in Q_0$ and some $q \in Q$

(2) Additionally, when w contains a two-symbol sequence pD where $p \in Q$ (representing a state in Q) and $D \in \{L, S, R\}$, then the following holds:
   (i) if D is L there is a substring pq in the preceding *position* in w, where $q \in Q$.
   (ii) if D is R there is a substring pq in the following *position* in w, where $q \in Q$.
   (iii) if D is S there is a substring pq in the current *position* in w, where $q \in Q$.

In essence, we have modified method 2 to remove the possibility of including any 'crash' moves, and added the possibility of having qq substrings at the right edge to signal what would be an accepting path in M. We have also removed the requirement of follow-up states to moves carrying *all* possible transitions, i.e. we're not exploring paths in parallel with the model.

Again, call the language that conforms to (1) and (2) L, which is obviously regular, and we may construct the following language:

$$L_3 = h((L - \mathit{Insert}(L)) \cap (\Delta^* QQ)) \tag{8}$$

Theorem 3. *M accepts a word w iff $w \in L_3$.*

Proof. Suppose M accepts w. Then, by induction we see that L contains a string ending in qq and so $w \in L_3$. Conversely, if the language L contains a string u that ends in qq, then either (1) M accepts $h(u)$ or (2) running M on $h(u)$ would end in a nonterminating loop, and additional symbols are present in u that model another path ending in qq that does not start from an initial state. But then, in the latter case, L also accepts a shorter string u' that does not contain the subpath ending in qq. But this implies that u is not in $h((L - \mathit{Insert}(L)) \cap (\Delta^* QQ))$, and that M accepts w. □

8 Implementations

The methods above are practical and relatively straightforward to implement in very little space, assuming one has access to a compiler for regular expressions. We have developed a simple conversion tool that reads descriptions of 2DFAs/2NFAs and converts them into regular expressions as defined above,

which can then be compiled into one-way automata.[2] For the implementation we rely on the regular expression formalism supported by the *PARC Finite-State Tool* [1] and the finite-state toolkit *foma* [6].

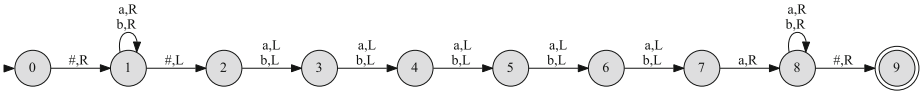

Fig. 5. Two-way automaton.

Table 1. Illustration of the growth in states when intersecting the sublanguages in suboptimal order (right) and the more efficient order that includes the L_{base} (left) with method 1, compiling the 2DFA in Fig. 5. The final 1DFA has 66 states.

k	size($L_{base} \cap L_{license_0} \cap \ldots \cap L_{license_k}$)	size($L_{license_0} \cap \ldots \cap L_{license_k}$)
0	33	58
1	77	1,394
2	112	29,634
3	166	589,570
4	204	11,271,170
5	226	NF
6	210	NF
7	131	NF
8	138	NF
9	181	NF

9 Practical Concerns

In an actual implementation it is important to calculate the intersections $L_{base} \cap L_{license_0} \cap \ldots \cap L_{license_k}$ in left-to-right order to avoid undesired exponential growth in the number of states. The $L_{license}$-languages (except for the 0-case) are symmetrical and therefore of the same size (n states) and so, in the worst case, the size of the minimal DFA result of intersection is n^k [2]. Separately constructing L_{base} and $L_{license_0} \cap \ldots \cap L_{license_n}$ is suboptimal in practice and quickly leads to unnecessary growth in the result, which would often be curbed had the general structure of L_{base} been imposed first. This is illustrated in Table 1. There we also see that the maximal partial result in the example is not substantially larger than the resulting final minimized DFA, if intersection is done in the proposed order. It is, of course, also advisable to minimize partial results through standard DFA-minimization. Additional optimizations not presented above for the sake of clarity include constraining the positions between the symbols from Σ to not contain repetitions of triplets representing transitions.

[2] Available at https://github.com/mhulden/2nfa.

10 Conclusion

We have presented three variants of a basic approach to converting 2DFA/2NFA to one-way automata. The construction methods offer a way to leverage the existence of efficient tools for compiling extended regular expressions into one-way automata, and thus makes it practicable to integrate two-way specifications into practical applications. We expect that the simulation method can be extended to cover more specific and constrained variants of two-way automata and two-way transducers.

References

1. Beesley, K.R., Karttunen, L.: Finite State Morphology. CSLI Publications, Stanford (2003)
2. Birget, J.C.: Intersection and union of regular languages and state complexity. Inf. Process. Lett. **43**(4), 185–190 (1992)
3. Birget, J.C.: State-complexity of finite-state devices, state compressibility and incompressibility. Math. Syst. Theor. **26**(3), 237–269 (1993)
4. Chrobak, M.: Finite automata and unary languages. Theoret. Comput. Sci. **47**, 149–158 (1986)
5. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
6. Hulden, M.: Foma: a finite-state compiler and library. In: Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics, pp. 29–32. Association for Computational Linguistics (2009)
7. Kapoutsis, C.A.: Removing bidirectionality from nondeterministic finite automata. In: Jedrzejowicz, J., Szepietowski, A. (eds.) MFCS 2005. LNCS, vol. 3618, pp. 544–555. Springer, Heidelberg (2005)
8. Kapoutsis, C.A.: Size complexity of two-way finite automata. In: Diekert, V., Nowotka, D. (eds.) DLT 2009. LNCS, vol. 5583, pp. 47–66. Springer, Heidelberg (2009)
9. Kozen, D.C.: Automata and Computability. Springer, New York (1997)
10. Kunc, M., Okhotin, A.: Describing periodicity in two-way deterministic finite automata using transformation semigroups. In: Mauri, G., Leporati, A. (eds.) DLT 2011. LNCS, vol. 6795, pp. 324–336. Springer, Heidelberg (2011)
11. Kunc, M., Okhotin, A.: State complexity of union and intersection for two-way nondeterministic finite automata. Fundamenta Informaticae **110**(1), 231–239 (2011)
12. Mereghetti, C., Pighizzini, G.: Optimal simulations between unary automata. SIAM J. Comput. **30**(6), 1976–1992 (2001)
13. Rabin, M., Scott, D.: Finite automata and their decision problems. IBM J. **3**(2), 114–125 (1959)
14. Shallit, J.: A Second Course in Formal Languages and Automata Theory. Cambridge University Press, Cambridge (2008)
15. Shepherdson, J.C.: The reduction of two-way automata to one-way automata. IBM J. Res. Dev. **3**, 198–200 (1959)
16. Vardi, M.Y.: A note on the reduction of two-way automata to one-way automata. Inf. Process. Lett. **30**(5), 261–264 (1989)