BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2022
http://excel.fit.vutbr.cz

Register Set Automata

Sabína Gulčíková*

Abstract
Register automaton (RA) operating over an infinite alphabet is one of the great tools for pattern matching with backreferences, runtime verification, or modelling of parallel computation. In case of pattern matching with backreferences, the state-of-the-art matchers make use of backtracking algorithms, whose application causes significant slowdown in case of nondeterministic regular expressions. The RA's property of non-determinisability makes it an unsuitable model for solution to problems related to inefficient usage of backtracking algorithms. On the other hand, the RA's quality of being equipped by a finite memory serves as a good basis for storing the so-called capture groups used in this application.
In this work, a formal model called register set automaton is proposed. A large class of RAs can be transformed into this deterministic model, which, among other things, allows for fast pattern matching with backreferences. In this paper, we explore its properties including determinisability, expressive power, and closure under Boolean operations. In addition, algorithms for other problems, such as emptiness testing, are introduced.

Keywords: finite memory automata — register automata — regular expression matching

Supplementary Material: *N/A*

*xgulci00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Register automaton (RA) is an automaton model operating over an infinite alphabet. Such models have found applications in many fields of computer science including modelling of parallel computation, pattern matching with backreferences, runtime verification, or even testing satisfiability in the SMT theory of strings.

RA, also known as finite memory automaton, was first introduced in [1] as an extension of finite automaton equipped with a finite set of registers, each of which can store a single value copied from the input tape. It was introduced with the goal of preserving closure under Boolean operations. The authors succeeded in doing so, with the exception of closure under complementation. This property could only be accomplished by constraining to a more restricted model. The emptiness problem, however, remained decidable.

The RA's non-determinisability causes it to be an unsuitable model for some practical applications. One of them is matching regexes with back-references, which is done ubiquitously in validating user input on web pages, processing text using the `grep` and `sed` tools, transforming XML documents, or detecting network incidents. Consider, for instance, the (extended) regular expression (regex)

$$R_{\backslash 3\backslash 2\backslash 1} = \texttt{/(.).*;.*(.).*;.*(.).*\3\2\1/},$$

which matches strings of the form

$$\texttt{...a...;...b...;...c...cba...}$$

for any symbols `a`, `b`, `c`, and any number of arbitrary symbols (except '`;`') at the positions of '`.`'.

Using the state-of-the-art PCRE2 regex matcher, it takes 10,416 steps before reporting no match in the 42-character-long string

```
"ah;jk2367ash;la5akv45lwkjb9f.dj5fqkbxsfyrf".
```

If we were to remove the delimiters (semicolons in $R_{\backslash 3\backslash 2\backslash 1}$), the process would take 179,372 steps in the same text. Ideally, this process should only take 42

steps, one for each character in the string. This slowdown is caused by the so-called *catastrophic backtracking*. The PCRE2 matcher is based on backtracking, and since the regex is nondeterministic, the backtracking algorithm needs to try all possibilities of placing the three capture groups before concluding that there is no match. Such scenario causes the systems making use of pattern matching with backreferences to perform poorly. In the worst case, this leads to an undesirable behavior, with systems being prone to attacks such as *regular expression denial of service* (ReDoS) [2].

Even though it is impossible to determinise a significant class of register automata, the presence of its finite memory lays out a convenient basis for storing of capture groups in pattern matching with backreferences. The main goal of this paper is to introduce a formal model called register set automaton (RsA), which is based on the theory of finite state automata, whose deterministic subclass can capture a large fragment of nondeterministic register automata (NRAs). For example, Figure 1 depicts a comparison of non-deterministic register automaton and deterministic register set automaton, both accepting a language of strings containing at least two occurrences of some symbol. In case of the standard finite automaton operating over an infinite alphabet, there is no automaton that could accept such language.

This work is directed towards the development of formal theory for RsAs, focusing on Boolean closure properties, the power of nondeterminism and size reduction. It explores the decidability of problems such as emptiness testing in case of RsA and its extensions.

2. Preliminaries

We use $\mathbb{N}$ to denote the set of natural numbers without 0, $\mathbb{N}_0$ to denote $\mathbb{N} \cup \{0\}$, and $[n]$ for $n \in \mathbb{N}$ to denote the set $\{1, \ldots, n\}$ (we note that $[0] = \emptyset$). We use $f\colon X \rightharpoonup Y$ to denote a partial function f from X to Y. If the value of f for $x \in X$ is undefined, we write $f(x) = \bot$. We use $|s|$ to denote the *cardinality* of set s. The notation $P \leq S$ is used to denote the *reduction* from problem P to problem S.

Data Word. Let us fix a finite nonempty *alphabet* Σ and an infinite *data domain* $\mathbb{D}$ (in examples, we will often work with $\mathbb{D} = \mathbb{N}_0$). A (finite) *data word* of *length* n is a function $w\colon [n] \to (\Sigma \times \mathbb{D})$; we use $|w| = n$ to denote its length. The *empty word* of length 0 is denoted ε. We use $\Sigma[w]$ and $\mathbb{D}[w]$ to denote the *projection* of w onto the respective domain (e.g., if $w = \langle a,1\rangle\langle b,2\rangle\langle b,3\rangle$, then $\Sigma[w] = abc$ and $\mathbb{D}[w] = 123$).

Register Automata on Data Words. A (nondeterministic one-way) *register automaton* (on data words), abbreviated as (N)RA, is a tuple $\mathscr{A} = (Q, \mathbf{R}, \Delta, I, F)$ where Q is a finite set of *states*, $\mathbf{R}$ is a finite set of *registers*, $I \subseteq Q$ is a set of *initial states*, $F \subseteq Q$ is a set of *final states*, and $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to \mathbf{R} \cup \{in, \bot\}) \times Q$ is a *transition relation* such that if $t = (q, a, g^{=}, g^{\neq}, up, s) \in \Delta$, then $g^{=} \cap g^{\neq} = \emptyset$. We use $q \xrightarrow{a \mid g^{=}, g^{\neq}, up} s$ to denote t (we often drop from up mappings $r \mapsto r$ for $r \in \mathbf{R}$, which are implicit). The $g^{=}$ and $g^{\neq}$ are used to denote the *guard* of a transition, and up to denote the *update* of a transition. Intuitively, the semantics of t is that $\mathscr{A}$ can move from state q to state s if the Σ-symbol at the current position of the input word is a and the $\mathbb{D}$-symbol at the current position is equal to all registers from $g^{=}$ and not equal to any register from $g^{\neq}$; the content of the registers is updated so that $r_i \leftarrow up(r_i)$ (i.e., r_i can be assigned the value of some other register, the current $\mathbb{D}$-symbol, denoted by in, or it can be cleared by being assigned $\bot$).

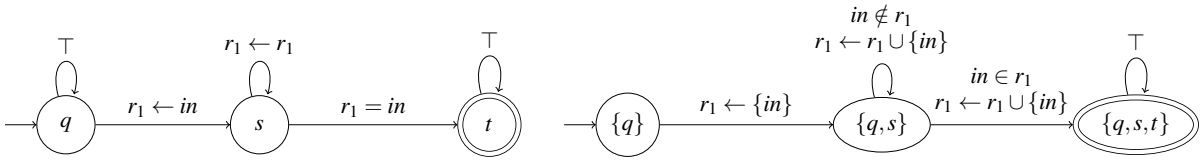
A *configuration* of $\mathscr{A}$ is a pair $c \in Q \times (\mathbf{R} \rightharpoonup \mathbb{D})$, i.e., it consists of a state and an assignment of data values to registers. An *initial configuration* of $\mathscr{A}$ is a pair $c_{init} \in I \times \{\{r_i \mapsto \bot \mid r_i \in \mathbf{R}\}\}$. Suppose $c_1 = (q_1, f_1)$ and $c_2 = (q_2, f_2)$ are two configurations of $\mathscr{A}$. We say that c_1 can make a *step* to c_2 over $\langle a, d\rangle \in \Sigma \times \mathbb{D}$ using transition $t\colon q \xrightarrow{a \mid g^{=}, g^{\neq}, up} s \in \Delta$, denoted as $c_1 \vdash_t^{\langle a,d\rangle} c_2$, iff

1. $d = f_1(r_i)$ for all $r_i \in g^{=}$,
2. $d \neq f_1(r_i)$ for all $r_i \in g^{\neq}$, and
3. for all $r_i \in \mathbf{R}$, we have

$$f_2(r_i) = \begin{cases} f_1(r_j) & \text{if } up(r_i) = r_j \in \mathbf{R} \\ d & \text{if } up(r_i) = in \\ \bot & \text{if } up(r_i) = \bot. \end{cases}$$

A *run* ρ of $\mathscr{A}$ over the word $w = \langle a_1, d_1\rangle \ldots \langle a_n, d_n\rangle$ from the configuration c is a sequence of alternating configurations and transitions $\rho = c_0 t_1 c_1 t_2 \ldots t_n c_n$ such that $\forall 1 \leq i \leq n\colon c_{i-1} \vdash_{t_i}^{\langle a_i,d_i\rangle} c_i$ and $c_0 = c$. We say that ρ is accepting if c is an initial configuration, $c_n = (s, f)$, and $s \in F$. The language accepted by $\mathscr{A}$, denoted as $\mathscr{L}(\mathscr{A})$ is defined as $\mathscr{L}(\mathscr{A}) = \{w \in (\Sigma \times \mathbb{D})^* \mid \mathscr{A} \text{ has an accepting run over } w\}$.

We say that $\mathscr{A}$ is a *deterministic RA* (DRA) if for all states $q \in Q$ and all $a \in \Sigma$, it holds that for any two distinct transitions $q \xrightarrow{a \mid g_1^{=}, g_1^{\neq}, up_1} s_1 \in \Delta$, and $q \xrightarrow{a \mid g_2^{=}, g_2^{\neq}, up_2} s_2 \in \Delta$ we have that $g_1^{=} \cap g_2^{\neq} \neq \emptyset$ or $g_2^{=} \cap g_1^{\neq} \neq \emptyset$.



(1) *Non-deterministic Register Automaton* **(2)** *Deterministic Register Set Automaton*

Figure 1. Comparison of register models equivalent to regex `/(.).*\1/`

Transfer Petri Nets Intuitively, transfer Petri net is an extension of Petri nets where transitions can *transfer* all tokens from one place to another place at once. They are closely related to *broadcast protocols* [3].

Formally, a *transfer Petri net* (TPN) is a triple $\mathcal{N} = (P, T, M_0)$, s.t. P is a finite set of *places*, T is a finite set of *transitions*, and $M_0 \colon P \to \mathbb{N}$ is an *initial marking*. The set of transitions T is such that $P \cap T = \emptyset$ and every transition $t \in T$ is of the form $t = \langle In, Out, Transfer \rangle$ where $In, Out \colon P \to \mathbb{N}$ define t's *input* and *output places* respectively and $Transfer \colon P \to P$ is a (total) *transfer function*.

A *marking* of $\mathcal{N}$ is a function $M \colon P \to \mathbb{N}$ assigning a particular number of tokens to each state. Given a pair of markings M and M', we use $M \leq M'$ to denote that for all $p \in P$ it holds that $M(p) \leq M'(p)$.

Given a marking M, we say that a transition $t = \langle In, Out, Transfer \rangle$ is *enabled* if $In \leq M$, i.e., there is a sufficient number of tokens in each of its input places. We use $M[t\rangle M'$ to denote that

1. t is enabled in M and
2. M' is the marking such that for every $p \in P$:

$$M'(p) = \sum \{M_{aux}(p') \mid Transfer(p') = p\} + Out(p), \text{ where } M_{aux} = M - In.$$

That is, the successor marking M' is obtained by (i) removing In tokens from inputs of t, (ii) transferring tokens according to $Transfer$, and (iii) adding Out tokens to t's outputs.

We say that a marking M is *reachable* if there is a (possibly empty) sequence $t_1, t_2, \ldots, t_n$ of transitions such that it holds that $M_0[t_1\rangle M_1[t_2\rangle \ldots [t_n\rangle M$, where M_0 is the initial marking.

A marking M is *coverable*, if there exists a reachable marking M', such that $M \leq M'$. The *Coverability* problem for TPNs asks, given a TPN $\mathcal{N}$ and a marking M, whether M is coverable in $\mathcal{N}$.

3. Register Set Automaton

A (nondeterministic) *register set automaton* (operating on data words), abbreviated as (N)RsA is a tuple $\mathscr{A}_S = (Q, \mathbf{R}, \Delta, I, F)$ where $Q, \mathbf{R}, I, F$ are the same as for RAs and $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to 2^{\mathbf{R} \cup \{in\}}) \times Q$ such that if $q \xrightarrow{a \mid g^{\in}, g^{\notin}, up} s \in \Delta$, then $g^{\in} \cap g^{\notin} = \emptyset$ (as with NRAs, we often do not write mappings $r \mapsto \{r\}$ for $r \in \mathbf{R}$ when giving up). Intuitively, the semantics of a transition $q \xrightarrow{a \mid g^{\in}, g^{\notin}, up} s$ is that $\mathscr{A}_S$ can move from state q to state s if the Σ-symbol at the current position of the input word is a and the $\mathbb{D}$-symbol at the current position is in all registers from $g^{\in}$ and in no register from $g^{\notin}$; the content of the registers is updated so that $r_i \leftarrow \bigcup \{x \mid x \in up(r_i)\}$ (i.e., r_i can be assigned the union of values of several registers, possibly including the current $\mathbb{D}$-symbol denoted by in).

A *configuration* of $\mathscr{A}_S$ is a pair $c \in Q \times (\mathbf{R} \to 2^{\mathbb{D}})$, i.e., it consists of a state and an assignment of data values to registers. An *initial configuration* of $\mathscr{A}_S$ is a pair $c_{init} \in I \times \{\{r_i \mapsto \emptyset \mid r_i \in \mathbf{R}\}\}$. Suppose $c_1 = (q_1, f_1)$ and $c_2 = (q_2, f_2)$ are two configurations of $\mathscr{A}_S$. We say that c_1 can make a *step* to c_2 over $\langle a, d \rangle \in \Sigma \times \mathbb{D}$ using transition $t \colon q \xrightarrow{a \mid g^{\in}, g^{\notin}, up} s \in \Delta$, denoted as $c_1 \vdash_t^{\langle a,d \rangle} c_2$, iff

1. $d \in f_1(r_i)$ for all $r_i \in g^{\in}$,
2. $d \notin f_1(r_i)$ for all $r_i \in g^{\notin}$, and
3. for all $r_i \in \mathbf{R}$, we have $f_2(r_i) = \bigcup \{f_1(r_j) \mid r_j \in \mathbf{R}, r_j \in up(r_i)\} \cup \begin{cases} \{d\} & \text{if } in \in up(r_i) \\ \emptyset & \text{otherwise.} \end{cases}$

A *run* ρ of $\mathscr{A}_S$ over the word $w = \langle a_1, d_1 \rangle \ldots \langle a_n, d_n \rangle$ from the configuration c is an alternating sequence of configurations and transitions $\rho = c_0 t_1 c_1 t_1 \ldots t_n c_n$ such that $\forall 1 \leq i \leq n \colon c_{i-1} \vdash_{t_i}^{\langle a_i, d_i \rangle} c_i$ and $c_0 = c$. We say that ρ is accepting if c is an initial configuration, $c_n = (s, f)$, and $s \in F$. The language accepted by $\mathscr{A}_S$, denoted as $\mathscr{L}(\mathscr{A}_S)$ is defined as $\mathscr{L}(\mathscr{A}_S) = \{w \in (\Sigma \times \mathbb{D})^* \mid \mathscr{A}_S \text{ has an accepting run over } w\}$.

We say that the RsA $\mathscr{A}_S$ is *deterministic* (DRsA) if for all states $q \in Q$ and all $a \in \Sigma$, it holds that for any two distinct transitions $q \xrightarrow{a \mid g_1^{\in}, g_1^{\notin}, up_1} s_1$, $q \xrightarrow{a \mid g_2^{\in}, g_2^{\notin}, up_2} s_2 \in \Delta$ we have that $g_1^{\in} \cap g_2^{\notin} \neq \emptyset$ or $g_2^{\in} \cap g_1^{\notin} \neq \emptyset$.

4. Closure Properties of Register Set Automata

Theorem 4.1. *The following closure properties hold for* RsA*:*

1. RSA *is closed under union and intersection.*

2. RSA *is not closed under complement.*

Proof. The proofs for closure under union and intersection are standard: for two RSAs $\mathscr{A}_1$, and $\mathscr{A}_2$ with disjoint sets of states and registers, where $\mathscr{A}_1 = (Q_1, \mathbf{R}_1, \Delta_1, I_1, F_1)$ and $\mathscr{A}_2 = (Q_2, \mathbf{R}_2, \Delta_2, I_2, F_2)$, the RSA $\mathscr{A}_\cup$ accepting the union of their languages is obtained as $\mathscr{A}_\cup = (Q_1 \cup Q_2, \mathbf{R}_1 \cup \mathbf{R}_2, \Delta_1 \cup \Delta_2, I_1 \cup I_2, F_1 \cup F_2)$. Similarly, assuming that $\mathbf{R}_1 \cap \mathbf{R}_2 = \emptyset$, $\mathscr{A}_\cap$ accepting their intersection is constructed as the product $\mathscr{A}_\cap = (Q_1 \times Q_2, \mathbf{R}_1 \cup \mathbf{R}_2, \Delta', I_1 \times I_2, F_1 \times F_2)$, where

$$(s_1, s_2) \xrightarrow{a \mid g_1^{\in} \cup g_2^{\in}, g_1^{\notin} \cup g_2^{\notin}, up_1 \cup up_2} (s_1', s_2') \in \Delta'$$

iff

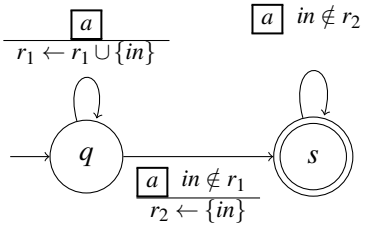
$$s_1 \xrightarrow{a \mid g_1^{\in}, g_1^{\notin}, up_1} s_1' \in \Delta_1, \text{ and}$$
$$s_2 \xrightarrow{a \mid g_2^{\in}, g_2^{\notin}, up_2} s_2' \in \Delta_2.$$

Correctness of the constructions is clear.

For showing the non-closure under complement, consider the language

$$L_{\neg\forall repeat} = \{w \mid \exists i \forall j \colon i \neq j \implies \mathbb{D}[w_i] \neq \mathbb{D}[w_j]\}$$

Intuitively, $L_{\neg\forall repeat}$ is the language of all data words that contain a data value with exactly one occurrence. This language is accepted, e.g., by the RSA in the following figure:



For the complement of the language, i.e., the language

$$L_{\forall repeat} = \{w \mid \forall i \exists j \colon i \neq j \wedge \mathbb{D}[w_i] = \mathbb{D}[w_j]\}$$

where all data values appear at least twice, there is no RSA that can accept it.

Intuitively, the equivalent RSA would need to store each seen data value into register r_1, which would semantically hold all data values that appear at least once. Then, each symbol already present in r_1 would need to be stored into register r_2, which would hold all values appearing at least twice. However, there would be no possibility to check if the difference of these two registers is empty, and, therefore, if it really holds that each data value appears at least twice. □

5. Determinising RAs into RSAs

An important property of RSAs is that a large class of NRA languages can be determinised into DRSAs. One of the properties that need to be satisfied in order for NRA to be determinisable, is that each of its registers is active only in one state. Any NRA can be transformed to satisfy this property, by creating a new copy of a register for every state that uses it, potentially increasing the number of registers to $|Q| \cdot |\mathbf{R}|$. The class of determinisable NRAs involves the ones, whose language does not require manipulation with dependencies between individual symbols of the input word (e.g. language of strings in which some tuple of symbols appears at least twice, such that the order of the symbols within the tuple is preserved). By collecting all possible values that can occur in registers, the algorithm for determinisation is performing the so-called *Cartesian abstraction* (i.e., it is losing information about dependencies between components in tuples). This can lead to a scenario where, for some set-register assignment f_i' of $\mathscr{A}'$, we would have $d_1 \in f_i'(r_1)$ and $d_2 \in f_i'(r_2)$, but there would be no corresponding configuration of $\mathscr{A}$ with register assignment f_i such that $d_1 = f_i(r_1)$ and $d_2 = f_i(r_2)$. The algorithm is complete for $\text{NRA}_1^{=}$, which contains NRAs with at most one register and no disequality tests ($g^{\neq}$ in every transition is $\emptyset$). In this section, we give the determinisation semi-algorithm.

The determinisation (semi-)algorithm for a nondeterministic *RA* $\mathscr{A}$ is shown in Algorithm 1. On the high level, it is similar to a worklist algorithm for determinising NFAs with additional treatment of registers superimposed onto it. Intuitively, we start from the macrostate I (representing all initial states of $\mathscr{A}$) and generate all reachable macrostates (subsets of Q) while determinising the transition function of $\mathscr{A}$, saving the macrostates into Q'. The main part of the algorithm is the loop starting at line 5, in which we are creating a set of transitions from the macrostate S such that any two transitions have incompatible guards (and the constructed RSA is therefore deterministic). Intuitively, for every symbol $a \in \Sigma$ and every partition of $\mathbf{R}$ into two subsets g and $\overline{g}$, we collect into T all transitions $q \xrightarrow{a \mid g^{=}, g^{\neq}, up} q'$ of $\mathscr{A}$ from states in S over a such that they are compatible with the partition, i.e., $g^{=} \subseteq g$ and $g^{\neq} \subseteq \overline{g}$. The sets g and $\overline{g}$ then serve as the guards of the new transition (line 21).

Next, for each r in $g^{\neq}$, we check whether it contains at most one value. This is done in order to ensure that the new transition preserves the semantics of $\mathscr{A}$. The check is done on line 8. The information about the number of values in register r is denoted by $c(r)$,

Algorithm 1: Determinisation of NRAs

Input : NRA $\mathscr{A} = (Q, \mathbf{R}, \Delta, I, F)$ s.t. each register is active in only one state
Output : DRsA $\mathscr{A}'$ with $\mathscr{L}(\mathscr{A}') = \mathscr{L}(\mathscr{A})$ or $\bot$

1 $Q' \leftarrow$ worklist $\leftarrow \{(I, c_0 = \{r_i \mapsto 0 \mid r_i \in \mathbf{R}\})\}$;
2 $\Delta' \leftarrow \emptyset$;
3 **while** worklist $\neq \emptyset$ **do**
4 $\quad (S, c) \leftarrow$ worklist.*pop*();
5 $\quad$ **foreach** $a \in \Sigma, g \subseteq \mathbf{R}$ **do**
6 $\quad\quad T \leftarrow \{q \xrightarrow{a \mid g^{=}, g^{\neq}, \cdot} q' \in \Delta \mid q \in S, g^{=} \subseteq g, g^{\neq} \cap g = \emptyset\}$;
7 $\quad\quad S' \leftarrow \{q' \mid \cdot \xrightarrow{\cdot \mid \cdot, \cdot, \cdot} q' \in T\}$;
8 $\quad\quad$ **if** $\exists q \xrightarrow{\cdot \mid \cdot, g^{\neq}, \cdot} q' \in T, \exists r_i \in g^{\neq} : c(r_i) = \omega$ **then**
9 $\quad\quad\quad$ **return** $\bot$;
10 $\quad\quad$ **foreach** $r_i \in \mathbf{R}$ **do**
11 $\quad\quad\quad op_{r_i} \leftarrow \{x \in \mathbf{R} \cup \{in\} \mid \cdot \xrightarrow{\cdot \mid \cdot, \cdot, up} \cdot \in T, up(r_i) = x\}$;
12 $\quad\quad P \leftarrow op_{r_1} \times \ldots \times op_{r_n}$ for $\mathbf{R} = \{r_1, \ldots, r_n\}$;
13 $\quad\quad$ **foreach** $(x_1, \ldots, x_n) \in P$ **do**
14 $\quad\quad\quad$ **if** $\nexists (\cdot \xrightarrow{\cdot \mid \cdot, \cdot, up} \cdot) \in T$ *s.t.* $\bigwedge_{1 \le i \le n} up(r_i) = x_i$ **then**
15 $\quad\quad\quad\quad$ **return** $\bot$;
16 $\quad\quad up' \leftarrow \{r_i \mapsto op_{r_i} \mid r_i \in \mathbf{R}\}$;
17 $\quad\quad c' \leftarrow \{r_i \mapsto \sum_{x_j \in up'} c(x_j) \mid r_i \in \mathbf{R} \wedge up' \notin g\}$;
18 $\quad\quad$ **if** $(S', c') \notin Q'$ **then**
19 $\quad\quad\quad$ worklist.*push*$((S', c'))$;
20 $\quad\quad\quad Q' \leftarrow Q' \cup \{(S', c')\}$;
21 $\quad\quad \Delta' \leftarrow \Delta' \cup \{(S, c) \xrightarrow{a \mid g, \mathbf{R} \setminus g, up'} (S', c')\}$;
22 **return** $\mathscr{A}' = (Q', \mathbf{R}, \Delta', \{I\}, \{(S, c) \in Q' \mid S \cap F \neq \emptyset\})$;

such that:

$$c(r) = \begin{cases} |r| & \text{if } |r| \le 1 \\ \omega & \text{if } |r| > 1. \end{cases}$$

An essential part of the algorithm is producing the *update* function up' and detecting when is the definition of equivalent *update* impossible without the over-approximation of the content of registers. For this, on line 10, we compute for every register $r_i \in \mathbf{R}$ the aggregation op_{r_i} of update functions (w.r.t. r_i) of all transitions in T.

Now, we need to check whether the computed aggregation is precise or whether it introduced some over-approximation. This is done on lines 12–15. The Cartesian product P on line 12 defines the new update function of every RsA register for the transition. Then, on line 13, we test whether the semantics of the update is consistent with the behavior of $\mathscr{A}$. If there is a combination $(x_1, \ldots, x_n) \in P$ mixing updates from different transitions, then the algorithm aborts and returns $\bot$. Otherwise, a new transition is added (line 21).

Theorem 5.1. *If Algorithm 1 returns an* DRsA $\mathscr{A}'$, *then* $\mathscr{L}(\mathscr{A}') = \mathscr{L}(\mathscr{A})$.

6. The Language Emptiness of Register Set Automata

The next theorem shows that the emptiness problem of RsA is decidable, but for a much higher price than for NRAs, for which it is PSPACE-complete[1] [5]. For classifying the complexity of the problem, we use the hierarchy of fast-growing complexity classes of Schmitz [6], in particular the class $\mathbf{F}_\omega$, which, intuitively, corresponds to Ackermannian problems closed under primitive-recursive reductions (see Schmitz [6] for an excellent exposition).

Theorem 6.1. *The emptiness problem for* RsA *is decidable. In particular, the problem is* $\mathbf{F}_\omega$*-complete.*

Sketch of proof. The proof is done by showing the interreducibility of RsA emptiness with coverability in *transfer Petri nets* (TPNs) (often used for modelling the so-called *broadcast protocols*), which is a known $\mathbf{F}_\omega$-complete problem [7, 8, 9].

[1] Note that for an alternative definition of NRAs considered in [1, 4], where no two registers can contain the same data value, the problem is NP-complete [4].

($\text{RsA} \leq \mathit{TPN}$) Intuitively, the reduction of RsA $\mathscr{A} = (Q, \mathbf{R}, \Delta, I, F_{\mathscr{A}})$ to *TPN* $\mathscr{N}_{\mathscr{A}} = (P, T, M_0)$ is done in the following way.

The set of places contains the states of $\mathscr{A}$ (there will always be at most one token in those places), two new places *init* and *fin* that are used for the initial nondeterministic choice of some initial state of $\mathscr{A}$ and for a unique *final place* (whose coverability will be checked) respectively, and, finally, a new place for every *region* of the Venn diagram of $\mathbf{R}$, which will track the number of data values that two or more registers share (e.g., for $\mathbf{R} = \{r_1, r_2, r_3\}$, the subset $\{r_1, r_3\}$ denotes the region $r_1 \cap \overline{r_2} \cap r_3$, i.e., the data values that are stored in r_1 and r_3 but are not stored in r_2.

Each transition of $\mathscr{A}$ is simulated by one or more transitions between places representing its source and target states. The number of respective transitions depends on how specific the guard is in the original automaton, since we need to distinguish every possible option of *in* being in some region $\mathit{rgn} \in 2^{\mathbf{R}}$.

Next, for each transition representing position of *in* in some *rgn*, a set of arches transferring values between regions is calculated, in order to preserve the semantics and position of values defined by the *guard* and *update* formulae.

Finally, the marking M_F to be covered is constructed as $\forall p \in P\colon M_F(p) = 1$ iff $p \in F_{\mathscr{A}}$, else $M_F(p) = 0$.

($\mathit{TPN} \leq \text{RsA}$) Intuitively, given a TPN $\mathscr{N}$, we will construct the RsA $\mathscr{A}_{\mathscr{N}}$ simulating $\mathscr{N}$, which will have the following structure:

- There will be the state q_{main}, which will be active before and after simulating the firing of TPN transitions.
- Each place of $\mathscr{N}$ will be simulated by a register of $\mathscr{A}_{\mathscr{N}}$; every token of $\mathscr{N}$ will be simulated by a unique data value.
- For every TPN transition, $\mathscr{A}_{\mathscr{N}}$ will contain a *gadget* that transfers data values between the registers representing the places active in the TPN transition. The gadget will start in q_{main} and end also in q_{main}.
- Coverability of a marking will be simulated by another gadget connected to q_{main} that will try to remove the number of tokens given in the marking from the respective places and arrive at the single final state q_{fin}. □

7. Extensions of Register Set Automata

Even small extensions of the RsA model may lead to undecidability. For instance, if we allow testing registers for equality, we can show interreducibility with the reachability problem of Petri nets with inhibitor arcs, which is undecidable [10]. The same undecidability result holds for an extension of RsAs allowing removing values from registers and testing register emptiness.

8. Conclusion

In this paper, we have presented a register model called register set automaton, whose determinisability can, in some cases, provide a great basis for many practical applications, such as pattern matching with backreferences. In case of the original register automaton, this is not possible because of its non-determinisability, which leads to problems related to inefficient usage of backtracking algorithms.

In addition to the formal definition of RsA, we introduced basic closure properties, possible extensions of this model, (semi-)algorithm for determinisation of RAs into RsAs, and provided reduction for classifying the complexity of the language emptiness problem.

In the future, the goal of ours is to experiment with different structure of registers, which would allow us to store capture groups of greater length, identify corresponding logic fragment for (D)RsA, and introduce effective algorithms for language inclusion testing.

Acknowledgements

I would like to extend my gratitude to Ondřej Lengál under whose supervision this work was created. I am thankful for his exceptionally helpful comments on the quality of this paper and his relentless guidance throughout its creation.

References

[1] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, November 1994.

[2] Adar Weidman. Regular expression denial of service — redos. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS, 2021. [Online; accessed 1-February-2021].

[3] Javier Esparza, Alain Finkel, and Richard Mayr. On the verification of broadcast protocols. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 352–359. IEEE Computer Society, 1999.

[4] Hiroshi Sakamoto and Daisuke Ikeda. Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci.*, 231(2):297–308, 2000.

[5] Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic*, 10(3):1–30, April 2009.

[6] Sylvain Schmitz. Complexity Hierarchies Beyond Elementary. *ACM Transactions on Computation Theory*, 8(1):1–36, February 2016.

[7] Sylvain Schmitz and Philippe Schnoebelen. The power of well-structured systems. In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8052 of *Lecture Notes in Computer Science*, pages 5–24. Springer, 2013.

[8] Sylvain Schmitz. *Algorithmic Complexity of Well-Quasi-Orders*. Habilitation à diriger des recherches, École normale supérieure Paris-Saclay, November 2017.

[9] Sylvain Schmitz and Philippe Schnoebelen. Algorithmic Aspects of WQO Theory. August 2012.

[10] Marvin Lee Minsky. *Computation*. Prentice-Hall Englewood Cliffs, 1967.