I F I G

Research

Report

Institut für Informatik

Self-Assembling Finite Automata

Andreas Klein Martin Kutrib

IFIG Research Report 0201

January 2002

Institut für Informatik
JLU Gießen
Arndtstraße 2
D-35392 Giessen, Germany
Tel: +49-641-99-32141
Fax: +49-641-99-32149
mail@informatik.uni-giessen.de
www.informatik.uni-giessen.de

Justus-Liebig-Universität Giessen

IFIG Research Report
IFIG Research Report 0201, January 2002

Self-Assembling Finite Automata

Andreas Klein[1]
Institut für Mathematik, Universität Kassel
Heinrich Plett Straße 40, D-34132 Kassel, Germany

Martin Kutrib[2]
Institute für Informatik, Universität Giessen
Arndtstr. 2, D-35392 Giessen, Germany

Abstract. We investigate a model of self-assembling finite automata. An automaton is assembled on demand during its computation from copies out of a finite set of items. The items are pieces of a finite automaton which are connected to the already existing automaton by overlaying states. Depending on the allowed number of such interface states, the degree, infinite hierarchies of properly included language families are shown. The presented model is a natural and unified generalization of regular and context-free languages since degrees one and two are characterizing the finite and pushdown automata, respectively. Moreover, by means of different closure properties nondeterministic and deterministic language families are separated.

CR Subject Classification (1998): F.1, F.4.3

[1]E-mail: klein@mathematik.uni-kassel.de
[2]E-mail: kutrib@informatik.uni-giessen.de

1 Introduction

Self-assembly appears in nature in several ways. One of the simplest mechanisms is the merging of drops of water when placed close together. The process is directed by minimization of potential energy and, thus, an example for uncoded self-assembly. On the other extreme in complexity protein molecules inside biological cells self-assemble to reproduce cells each time they divide. In this example the assembly instructions are built in the components and, therefore, it is coded self-assembly [3]. Originally, the study of self-assembly was motivated by biologists. A well-studied example is the assembly of bacteriophages, a type of virus which infects bacterial cells [1]. Formal investigations in this field are accompanied by the development of corresponding computational models which are also of great interest from an engineering point of view. An introduction can be found in [7] where an automaton model of self-assembling systems is presented. The model operates on one-dimensional strings that are assembled from a given multiset of smaller strings. The research was motivated by the evolutionary design of mechanical conformational switches. Classes of automata are defined depending on classes of subassembly sequences. Then the minimal number of conformations necessary to encode subassembly sequences in that class is studied.

Here, in some sense, we adapt self-assembly to the theory of automata and formal languages. Basically, the idea is to assemble an automaton during its computation. Therefore, we provide a finite set of items, the so-called *modules*. The automata are assembled from module copies on demand. The assembling rules are encoded by the state transition function. Starting with one piece of a finite automaton during the computation so-called *assembling transitions* are traversed that direct the assembling of another copy of some item in a well-specified manner.

Each of the modules has a set of entry and a set of return states which together are called *interface states*. An assembling transition specifies how the new copy of the module fits to the already existing part of the automaton. The connection is made by overlaying the interface states by existing states. So the result of the self-assembly is a finite automaton, but the number of its states may depend on the input. It will turn out that the generative capacity of such models depend on their degree, i.e. the number of interface states of the modules.

Related to the work of the present paper are the so-called self-modifying finite automata [4, 5, 6, 8]. In this model modifications of the automaton are allowed during transitions. The modifications include adding and deleting states and transitions. A weak form of self-modifying automata has been shown to accept the metalinear languages as well as some other families of context-free and non-context-free languages. Less restricted variants can accept arbitrarily hard languages, even non-recursive ones.

Since assembling modules sounds like calling subroutines another related paper is [2], where finite automata are considered that have a stack for storing return addresses (states). Every time a final state is entered the computation continues in the state at the top of the stack. Depending on the number of states which may be stored during one transition an infinite hierarchy in between the regular and context-free languages is shown.

Here by means of self-assembling finite automata of degree k we obtain a natural and unified generalization of finite automata and pushdown automata. In particular, infinite hierarchies depending on the degree are shown. For degree one and two the regular and context-free languages are characterized, respectively. Moreover, some closure properties are proved which lead to a separation result between nondeterministic and deterministic computations.

2 Self-Assembling Finite Automata

We denote the positive integers $\{1, 2, ...\}$ by $\mathbb{N}$ and the set $\mathbb{N} \cup \{0\}$ by $\mathbb{N}_0$. The empty word is denoted by λ. For the length of w we write $|w|$. We use $\subseteq$ for inclusions and $\subset$ if the inclusion is strict.

Basically, the idea is to assemble the automaton during its computation. For this purpose we assume a finite set of basic items which can be used in the assembling process. These items are called *modules*. To each module further copies of modules (from the finite set) can be connected. A module is very similar to a (piece of a) finite automaton. The difference is the presence of so-called *assembling transitions* which direct the assembling process. Every time the computation tries to change a state by an assembling transition a new copy of a module is connected. These connections are done by overlaying some states. Each module has an interface consisting of entry and exit states. The assembling rule specifies which of the states (of the already existing part) are to be identified by the interface states.

In order to introduce the model under consideration in more detail at first we define modules more formally:

Definition 1 *Let $u, v \in \mathbb{N}_0$ be constants. A* (nondeterministic) module with u entries and v exits *(u:v-module) is a system $\langle Q, I, O, A, \delta, F \rangle$, where*

1. *Q is the finite set of* inner states,
2. *$I = \{r_1, \ldots, r_u\}$ is the ordered set of u* entry states *such that $I \cap Q = \emptyset$,*
3. *$O = \{r_{u+1}, \ldots, r_{u+v}\}$ is the ordered set of v* return states *such that $O \cap (Q \cup I) = \emptyset$,*
4. *A is the finite set of* input symbols,
5. *The* module transition function *δ maps $Q \times A$ to the finite subsets of $Q \cup O \cup (\mathbb{N} \times Q^+ \times (Q \cup O)^+)$ and $I \times A$ to the subsets of Q,*
6. *$F \subseteq Q \cup I \cup O$ is the set of* accepting (or final) states.

So, the nondeterministic transition function may map states to states in which case we have state changes without assembling new items as usual.

In the second case δ requires to assemble a new copy of a module which is identified by an index from $\mathbb{N}$. The interface states I' and O' of the new module are overlayed by the states specified by $Q^+ \times (Q \cup O)^+$. From this point of view the restrictions of δ are convenient and natural: A return state is for exit purposes and, therefore, δ is not defined for states in O. Otherwise, a return state would at the same time be an entry state. Conversely, an entry state cannot be reached from inside the module. Otherwise it also would be a return state. Finally, after assembling a new module the computation should enter the module for at least one time step without assembling further modules, i.e., $I \times A$ is mapped to subsets of Q only.

Since modules are the basic items from which k-self-assembling finite automata are assembled, for their definition we need to ensure that only pieces are connected that fit together.

Definition 2 *Let $k \in \mathbb{N}_0$ be a constant. A* nondeterministic self-assembling finite automaton $\mathcal{M}$ of degree k (k-NFA) *is an ordered set of modules $\langle M_0, \dots, M_m \rangle$ over a common input alphabet A, where for all $0 \leq i \leq m$ the module $M_i = \langle Q, I, O, A, \delta, F \rangle$*

1. *has at most k interface states, i.e. $|I| + |O| \leq k$,*
2. *for all $(s, a) \in (Q \times A)$ the* assembling transition *$\big(j, (p_1, \dots, p_u), (p_{u+1}, \dots, p_{u+v})\big) \in \delta(s, a)$ implies*
   (a) *$j \leq m$ and M_j is a u:v-module,*
   (b) *$\{p_1, \dots, p_{u+v}\}$ are different and $s \in \{p_1, \dots, p_u\}$,*
3. *M_0 is a 0:0-module with a designated* starting state *s_0.*

Condition 2*b* ensures that at most two states are overlayed and, moreover, an assembling transition transfers the computation into the new module.

The general behavior of a k-NFA is best described by configurations and their successor configurations.

A *configuration* c_t of $\mathcal{M}$ at some time $t \geq 0$ is a description of its global state which is a set of existing states S_t, transition and assembling rules given by a mapping δ_t from $S_t \times A$ to the finite subsets of $S_t \cup \big(\mathbb{N} \times S_t^+ \times S_t^+\big)$, the currently active state s_t, the current set of final states F_t and the remaining input word w_t. Thus, a configuration is a 5-tuple $c_t = (S_t, \delta_t, s_t, F_t, w_t)$.

The initial configuration $c_0 = (Q_0, \delta_0, s_0, F_0, w)$ at time 0 is defined by the input word $w = a_0 \cdots a_{n-1} \in A^*$ and the components of module $M_0 = \langle Q_0, \emptyset, \emptyset, A, \delta_0, F_0 \rangle$, where s_0 is the designated starting state from Q_0.

Successor configurations are computed according to the *global transition function* Δ:

Let $c_t = (S_t, \delta_t, s_t, F_t, a_t \cdots a_{n-1})$ be a configuration, then for each element from $\delta_t(s_t, a_t)$ successor configurations $(S_{t+1}, \delta_{t+1}, s_{t+1}, F_{t+1}, a_{t+1} \cdots a_{n-1}) \in \Delta(c_t)$ are defined as follows.

1. During an ordinary state transition, as usual for finite automata, only the active state changes:
   Let $s \in S_t$ be an element from $\delta_t(s_t, a_t)$, then $S_{t+1} = S_t$, $\delta_{t+1} = \delta_t$, $s_{t+1} = s$ and $F_{t+1} = F_t$.
2. During an assembling transition a copy of the new module has to be created, the active state has to be computed and the interface states have to be overlayed, what includes the appropriate update of the rules.
   Let $\big(j, (p_1, \dots, p_u), (p_{u+1}, \dots, p_{u+v})\big) \in (\mathbb{N} \times S_t^+ \times S_t^+)$ be an element from $\delta_t(s_t, a_t)$.
   $\bar{M}_j = \langle \bar{Q}, \{\bar{r}_1, \dots, \bar{r}_u\}, \{\bar{r}_{u+1}, \dots, \bar{r}_{u+v}\}, A, \bar{\delta}, \bar{F}\rangle$ be a copy of M_j such that all of its states are different from states in S_t.
   Set $S_{t+1} = S_t \cup \bar{Q}$ and identify the entry states of $\bar{M}_j$ by $p_1, \dots, p_u$ and the return states by $p_{u+1}, \dots, p_{u+v}$. Accordingly F_{t+1} is defined to be $F_t \cup \{p_i \mid \bar{r}_i \in \bar{F}, 1 \le i \le u+v\} \cup (\bar{F} \setminus \{\bar{r}_1, \dots, \bar{r}_{u+v}\})$.
   In order to define s_{t+1} and δ_{t+1} we first observe that by definition state s_t has to belong to $\{p_1, \dots, p_u\}$, say $s_t = p_l$. Since p_l overlays $\bar{r}_l \in \bar{I}$ and $\bar{\delta}$ maps $(\bar{r}_l, a_t)$ to subsets of $\bar{Q}$ the new active state s_{t+1} is chosen from $\bar{\delta}(\bar{r}_l, a_t)$, i.e., the computation enters the newly assembled module.
   It remains to join the mappings δ_t and $\bar{\delta}$ to δ_{t+1} as follows:
   $$\begin{aligned}&\delta'_{t+1}(s, a) = \delta_t(s, a) \text{ for all } (s, a) \in (S_t \times A) \setminus \{(s_t, a_t)\}\\ &\delta'_{t+1}(s_t, a_t) = \delta_t(s_t, a_t) \setminus \{(j, (p_1, \dots, p_u), (p_{u+1}, \dots, p_{u+v}))\}\end{aligned}$$
   So in an intermediate step δ'_{t+1} is defined to be δ_t except for the applied assembling rule which is used and so will be replaced.
   The next step is to take the mapping $\bar{\delta}$ of $\bar{M}_j$ and textually rename each occurrence of an interface state $\bar{r}_i$ from $\bar{I} \cup \bar{O}$ by its overlaying state p_i. If the result is a mapping δ''_{t+1}, then the construction is completed by
   $$\begin{aligned}&\delta_{t+1}(s, a) = \delta'_{t+1}(s, a) \cup \delta''_{t+1}(s, a) \text{ for all } (s, a) \in \{p_1, \dots, p_u\} \times A\\ &\delta_{t+1}(s, a) = \delta'_{t+1}(s, a) \text{ for all } (s, a) \in (S_t \setminus \{p_1, \dots, p_u\}) \times A\\ &\delta_{t+1}(s, a) = \delta''_{t+1}(s, a) \text{ for all } (s, a) \in \bar{Q} \times A\end{aligned}$$

An input word $a_0 \cdots a_{n-1}$ is *accepted* by a k-NFA iff the set of possible configurations at time n (i.e., after processing the whole input) is not empty and contains at least one configuration whose active state s_n belongs to the set of accepting states F_n.

A k-NFA is deterministic (k-DFA) iff for any input all configurations have deterministic mappings, i.e., $\delta_t : S_t \times A \to S_t \cup (\mathbb{N} \times S_t^+ \times S_t^+)$ is a partial function.

The *family of all languages that are acceptable* by some k-NFA (k-DFA) is denoted by $\mathscr{L}(k\text{-NFA})$ ($\mathscr{L}(k\text{-DFA})$).

The following example illustrates self-assembling finite automata more figurative. It becomes important for proving hierarchies in later sections.

Example 3 For any constant $k \geq 1$ let $A_k = \{a_1, \dots, a_k\}$ be an alphabet and

$$L_k = \{a_1^n \cdots a_k^n \mid n \in \mathbb{N}\}$$

In order to show that L_k is accepted by a k-DFA we present constructions for $k = 2, 3$ which can easily be generalized to arbitrary k.

The following 2-DFA $\mathcal{M} = \langle M_0, M_1, M_2 \rangle$ accepts L_2:

$$\begin{aligned}
M_0 &= \langle \{s_0, q_1, q_2, q_3, q_4\}, \emptyset, \emptyset, A_2, \delta, \{q_3, q_4\} \rangle \text{ with starting state } q_0 \\
&\quad \delta(s_0, a_1) = q_1, \\
&\quad \delta(q_1, a_1) = q_2, \quad \delta(q_1, a_2) = q_3 \\
&\quad \delta(q_2, a_2) = (2, (q_2), (q_4)), \quad \delta(q_2, a_1) = (1, (q_2), (q_4)) \\
M_1 &= \langle \{q_1, q_2\}, \{r_1\}, \{r_2\}, A_2, \delta, \emptyset\} \rangle \\
&\quad \delta(r_1, a_1) = q_1 \\
&\quad \delta(q_1, a_1) = (1, (q_1), (q_2)), \quad \delta(q_1, a_2) = (2, (q_1), (q_2)) \\
&\quad \delta(q_2, a_2) = r_2 \\
M_2 &= \langle \{q_1\}, \{r_1\}, \{r_2\}, A_2, \delta, \emptyset \rangle \\
&\quad \delta(r_1, a_2) = q_1 \\
&\quad \delta(q_1, a_2) = r_2
\end{aligned}$$

Figure 1 shows the graphical representation of a 2-DFA $\mathcal{M} = \langle M_0, M_1, M_2 \rangle$ that accepts L_2. Assembling transitions are indicated by double arrows.

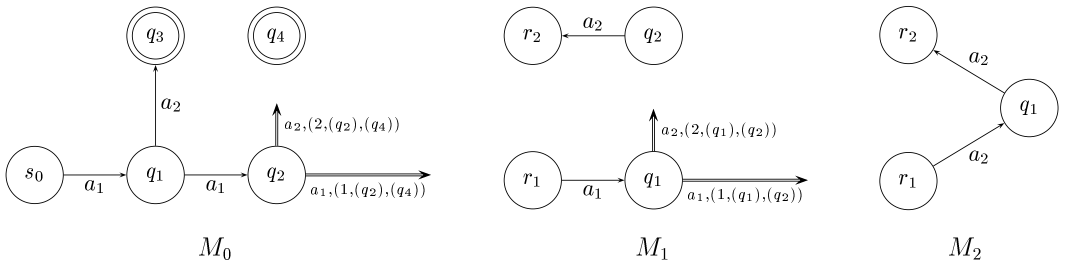

Figure 1: Modules of a 2-DFA accepting L_2.

The assembled 2-DFA after accepting the input $a_1^4 a_2^4$ is depicted in Figure 2.

Observe that there are simpler 2-DFAs for L_2. But this construction has been presented with an eye towards an easily understandable generalization.

The 3-DFA whose modules are shown in Figure 3 accepts L_3.

The assembled 3-DFA after accepting the input $a_1^4 a_2^4 a_3^4$ is depicted in Figure 4.

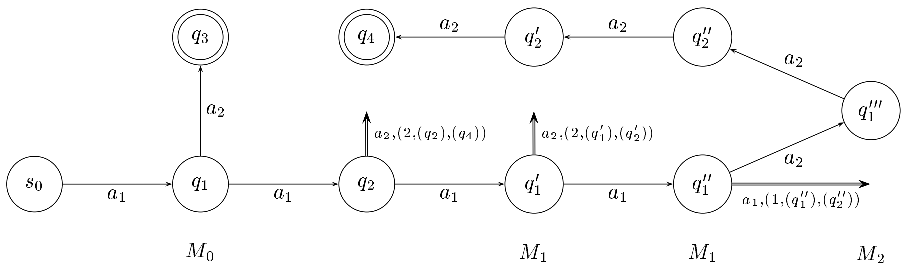

Figure 2: Structure of a 2-DFA accepting L_2 after processing $a_1^4 a_2^4$.

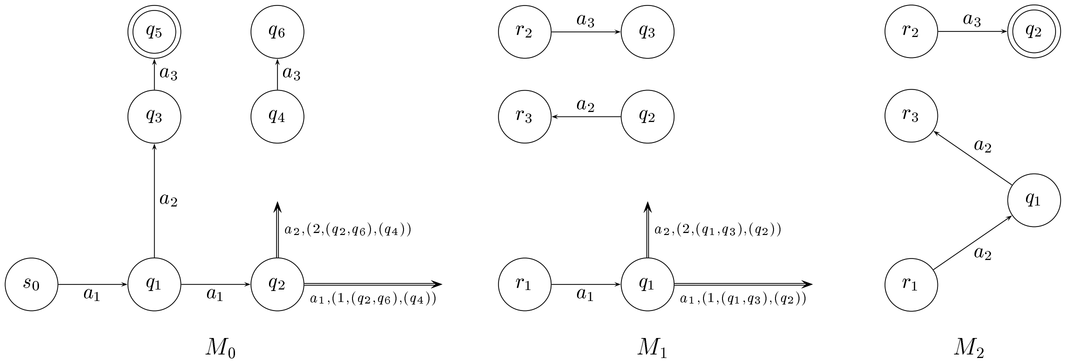

Figure 3: Modules of a 3-DFA accepting L_3.

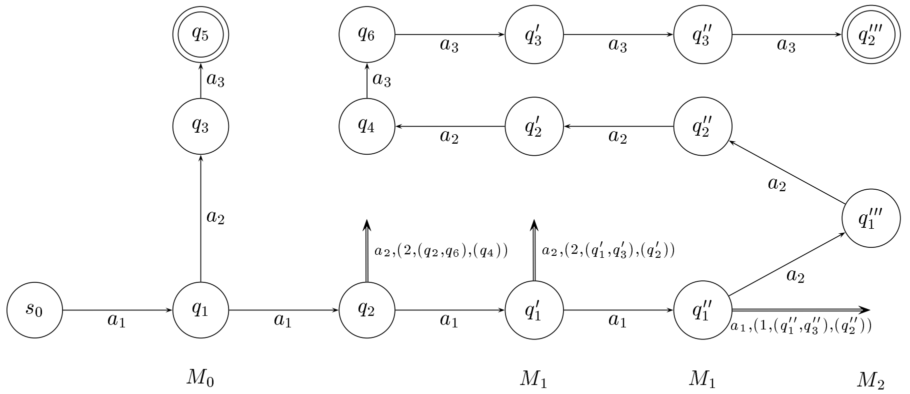

Figure 4: Structure of a 3-DFA accepting L_3 after processing $a_1^4 a_2^4 a_3^4$.

The self-assembling finite automata in Example 3 have an important and interesting property, they are *loop-free*. Exactly this restriction of the model yields a natural and unified generalization of finite and pushdown automata.

In particular, the generative capacity of self-assembling finite automata depends on their degree. Later on we are going to show infinite hierarchies of properly included language families for loop-free and unrestricted variants, but first we present the results concerning degrees 1 and 2. We prove that loop-free 1-DFAs accept exactly the regular and loop-free 2-NFAs exactly the context-free languages.

Definition 4 *Let $k \geq 1$ be a constant. A* computation *of a self-assembling finite automaton $\mathcal{M}$ of degree k is* loop-free *if $\mathcal{M}$ enters each of its existing states at most once. An* automaton *$\mathcal{M}$ is* loop-free *if all of its computations are loop-free.*

In order to distinguish loop-free languages we denote the family of all languages that are acceptable by some loop-free k-NFA (k-DFA) by $\mathscr{L}_{lf}(k\text{-NFA})$ ($\mathscr{L}_{lf}(k\text{-DFA})$).

Theorem 5 *Every context-free language is accepted by some loop-free* 2-NFA.

Proof. It is well known that for every context-free language not containing λ there exists a grammar in Greibach normal form. I.e., every production is of the form $X \to a\gamma$, where X is a variable, a a terminal and γ a possibly empty word of variables. In the following a loop-free 2-NFA is constructed that computes leftmost derivations of such a grammar $\mathcal{G}$. Subsequently, the empty word can be included simply by making the starting state final.

For each production $X \to aY_1 \cdots Y_n$ in $\mathcal{G}$ whose right-hand side has at least one variable a module $M_{Y_1 \cdots Y_n}$ is constructed as follows:

$$Q = \{Y_1, \dots, Y_n\}, \quad I = \{r_1\}, \quad O = \{r_2\}, \quad F = \emptyset,$$

$\delta(r_1, a) = \{Y_1\}$

For all $x \in A$:

$$\begin{aligned}
Y_{i+1} &\in \delta(Y_i, x) \text{ iff there exists the production } Y_i \to x \text{ in } \mathcal{G}, 1 \leq i \leq n-1,\\
r_2 &\in \delta(Y_n, x) \text{ iff there exists the production } Y_n \to x \text{ in } \mathcal{G},\\
(j, (Y_i), (Y_{i+1})) &\in \delta(Y_i, x) \text{ iff there exists a production } Y_i \to xZ_1 \cdots Z_l \text{ in } \mathcal{G} \text{ and}\\
&\quad M_j = M_{Z_1 \cdots Z_l}\\
(j, (Y_n), (r_2)) &\in \delta(Y_n, x) \text{ iff there exists a production } Y_n \to xZ_1 \cdots Z_l \text{ in } \mathcal{G} \text{ and}\\
&\quad M_j = M_{Z_1 \cdots Z_l}
\end{aligned}$$

Therefore, with input symbol x, the computation process assembles a module M_γ iff a leftmost derivation step of $\mathcal{G}$ generates $x\gamma$. The process returns from M_γ iff the variables γ have been completely replaced by terminals.

In order to complete the construction module M_0 is defined as the others, with the exception that r_1 is omitted, r_2 is now an inner state, the axiom of $\mathcal{G}$ is the second inner state, $F = \{r_2\}$ and the starting state is the axiom.

Altogether, the 2-NFA starts in a state that corresponds to the axiom of $\mathcal{G}$, simulates leftmost derivations and returns to the unique final state only if all variables that appear during the derivation of the input could be replaced. An example for the mirror language $\{w \mid w \in \{a,b\}^*, w = w^R\}$ with the productions $(S \to aSX \mid bSY \mid aX \mid bY \mid a \mid b)$, $(X \to a)$ and $(Y \to b)$ is depicted in Figure 5. □

In order to complete the characterization we now prove the converse of Theorem 5.

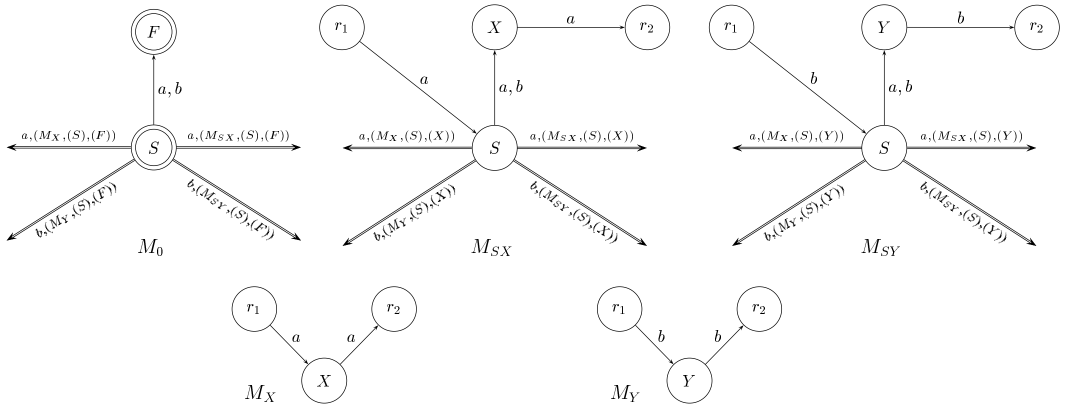

Figure 5: Modules of a loop-free 2-NFA accepting the mirror language.

Theorem 6 *Every loop-free* 2-NFA *language is context-free.*

Proof. For a given loop-free 2-NFA $\mathcal{M}$ we may assume without loss of generality that each module (except M_0) has at least one entry state. Otherwise, the module could not be assembled and, thus, could be omitted.

A nondeterministic pushdown automaton $\mathcal{A}$ can be designed in such a way that the finite control can simulate each of the modules of $\mathcal{M}$. Every time step a new module is assembled, $\mathcal{A}$ pushes the parameters of the assembling transition together with the index of the predecessor module onto its stack and starts the simulation of the new module in the appropriate entry state.

If the new module is a 2:0-module, this is all since the computation cannot return. In fact, in this case the stack content up to that time is not accessed any more.

If, on the other hand, the new module is a 1:1-module, then the process may return from it. In this case the information for the predecessor module is popped from the stack and the simulation continues appropriately.

The basic fact why this simulation works fine is the limitation of the degree and the loop-freeness. Every time a module is popped from the stack the instance of the module from which the process returns is forgotten. But this causes no restrictions since both interface states have been passed through and, thus, this instance of the module will never be entered again. $\square$

Corollary 7 *The family* $\mathscr{L}_{lf}(\text{2-NFA})$ *is equivalent to the context-free languages.*

Now we climb down the Chomsky-hierarchy and consider regular languages.

Theorem 8 *Every* 1-NFA *language is regular.*

Proof. Since when assembling a new module the computation process has to enter it, 0:1- or 0:0-modules are unreachable. So without loss of generality we may assume a 1-NFA has only 1:0-modules.

Since once a 1:0-module has been entered, the computation can never return and the sequence of assembled modules need not to be remembered. So in order to construct a nondeterministic finite automaton $\mathcal{A}$ with λ-moves from a given 1-NFA $\mathcal{M}$ one copy of each module is sufficient. These copies can be reused when a certain module is assembled more than once.

The state set of $\mathcal{A}$ is the disjoint union of all states of all modules of $\mathcal{M}$. Correspondingly, the state transition is the union of all module transition functions, where each assembling transition $\delta(s,a) = (j,(s),\lambda)$ is replaced by a λ-transition from state s to the unique entry state of M_j.

It is easy to verify that $\mathcal{A}$ and $\mathcal{M}$ are equivalent. $\square$

Conversely, every regular language can be accepted by some 1-DFA, since every deterministic finite automaton is a 0:0-module without assembling transitions. But using the idea of Theorem 5 a stronger result can be shown:

Theorem 9 *Every regular language is accepted by some loop-free* 1-DFA.

Proof. Let the regular language be given by a right-linear grammar. So the productions are of the form $X \to aY$, where a is a terminal and Y is empty or a variable. It is well known that we may assume that this grammar is deterministic, i.e, at every derivation step at most one production is applicable.

Now we can adapt the construction of Theorem 5. The difference is that in the regular case no return state is available. But here we do not need to return from an assembled module since at any time there exists at most one variable. Therefore, it suffices to provide and to connect to an inner final state instead of the return state.

Again, the empty word can be accepted in addition simply by making the starting state final. $\square$

Corollary 10 *The families* $\mathscr{L}$(1-NFA), $\mathscr{L}_{lf}$(1-NFA), $\mathscr{L}$(1-DFA) *and* $\mathscr{L}_{lf}$(1-DFA) *are equivalent to the regular languages.*

These results immediately raise the question for the power and limitations of loop-freeness in connection with self-assembling finite automata. There is no difference for automata with degree one. Are all loop-free automata with a given degree equivalent to unrestricted automata with the same degree? Quite the contrary, there is a difference for all $k \geq 2$. The next example proves the claim for $k = 2$. In the next section it is extended to arbitrary degrees.

Example 11 The 2-DFA whose modules are depicted in Figure 6 accepts the language

$$L = \{a^{n_1}b^{n_1} \cdots a^{n_j}b^{n_j}a^m c \mid j \in \mathbb{N}, n_i \geq 2 \text{ for } 1 \leq i \leq j \text{ and } m < \max\{n_1, \ldots, n_j\}\}$$

Since L is not context-free it does not belong to $\mathscr{L}_{lf}$(2-NFA).

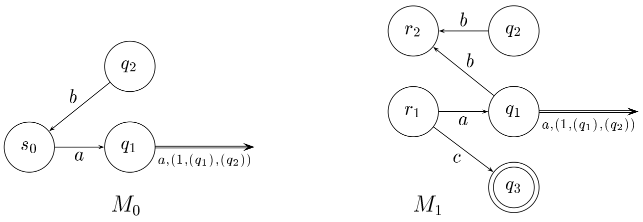

Figure 6: Modules of a 2-DFA accepting L of Example 11.

The example yields the following proper inclusions.

Corollary 12 $\mathscr{L}_{lf}$*(2-DFA)* $\subset \mathscr{L}$*(2-DFA) and* $\mathscr{L}_{lf}$*(2-NFA)* $\subset \mathscr{L}$*(2-NFA).*

In order to determine where the generative power of k-NFAs ends up here we state that for any $k \in \mathbb{N}$ the family $\mathscr{L}(k\text{-NFA})$ is a proper subfamily of the context-sensitive languages.

Since during a computation on input of length n at most n (copies of) modules are assembled, a linearly space bounded nondeterministic Turing machine can store these modules together with the parameters of the corresponding assembling transitions on its tape and simulate the computation process of the k-NFA. It is only a technical challenge to arrange the modules on the tape such that the simulation is able to find adjacent ones. This proves the inclusion. The properness will be shown later.

Theorem 13 *Let $k \in \mathbb{N}$ be a constant, then the family $\mathscr{L}(k\text{-NFA})$ is a proper subfamily of the context-sensitive languages.*

3 Hierarchies

Now we are going to explore the relative computation power of self-assembling finite automata. In particular, we compare nondeterministic and deterministic computations and investigate the relationships between degrees k and $(k+1)$. It turns out that there are infinite hierarchies depending on k. In order to separate language families we need a tool for proving negative results. Since we consider generalizations of well-known language families it is near at hand to generalize their pumping lemmas as well. It will turn out (Lemma 17) that for this purpose we have to restrict the automata to loop-free computations. But nevertheless the lemma yields the desired results for unrestricted automata, too.

The following pumping lemma is in some sense weaker than others, since it contains no statement about the usual ordering in which the repeated subwords appear.

Lemma 14 *Let $k \in \mathbb{N}$ be a constant and $\mathcal{M}$ be a loop-free k-NFA accepting a language L. Then there exists a constant $n \in \mathbb{N}$ such that every $w \in L$ with $|w| \geq n$ may be written as $x_0 y_1 x_1 y_2 \cdots y_k x_k$, where $|y_1| + |y_2| + \cdots + |y_k| \geq 1$, and for all $i \in \mathbb{N}$ there exists a word $w' \in L$ such that w' is in some order a concatenation of the (sub)words $x_0, \ldots, x_k$ and i times y_j for each $1 \leq j \leq k$.*

Proof. $\mathcal{M}$ consists of finitely many modules each having finitely many interface states. For long enough words from L there exists an accepting computation such that a module is assembled at least twice whereby the ordering of passed through interface states is identical. (For a module there exist only finitely many such orderings.)

Obviously, the necessary input length can be calculated. It depends on $\mathcal{M}$ only and defines the constant n.

Now let w be an accepted input with $|w| \geq n$. We denote the first instance of the module by $\hat{M}$ and the second one by $\tilde{M}$. The input symbols consumed until $\hat{M}$ is assembled define the subword x_0. Thus, after processing x_0 an interface state of $\hat{M}$ and $\tilde{M}$ appears for the first time. Next we consider the sequence of input symbols until the next interface state (of $\hat{M}$ or $\tilde{M}$) is entered and define it to be y_1. We continue as follows (cf. Figure 7): Input sequences connecting return states of $\hat{M}$ with entry states of $\hat{M}$, or entry states of $\tilde{M}$ with return states of $\tilde{M}$ form the subwords x_j. Input sequences connecting each other pair of interface states from $\hat{M}$ or $\tilde{M}$ form the subwords y_j. The input sequence after entering an interface state for the last time forms the subword x_k.

Since there must exist at least one path from an entry state of $\hat{M}$ to an entry state of $\tilde{M}$ (otherwise $\tilde{M}$ would not have been assembled), at least one subword y_i is not empty. On the other hand, since $\mathcal{M}$ is loop-free and its modules have at most k interface states, there exist at most k paths defining subwords y_j.

From the given accepting computation we derive another one by using a third copy $\bar{M}$ of the module and placing it in between the paths connecting $\hat{M}$ and $\tilde{M}$. The idea is as follows:

Since the interface states appearing in the same ordering $\bar{M}$ behaves like $\hat{M}$ when the computation enters one of its entry states. Thus, the connections between $\bar{M}$ and $\tilde{M}$ are identical to the connections between $\hat{M}$ and $\tilde{M}$.

On the other hand, $\bar{M}$ behaves like $\tilde{M}$ when the computation enters one of its return states. Thus, the connections between $\hat{M}$ and $\bar{M}$ are identical to the connections between $\hat{M}$ and $\tilde{M}$.

But the connections are exactly the subwords y_j. We conclude that the new accepting computation is for an input that is a concatenation of $x_0, \ldots, x_k$ and 2 times y_j for each $1 \leq j \leq k$. Trivially, we can insert i copies of $\bar{M}$ what completes the proof. $\square$

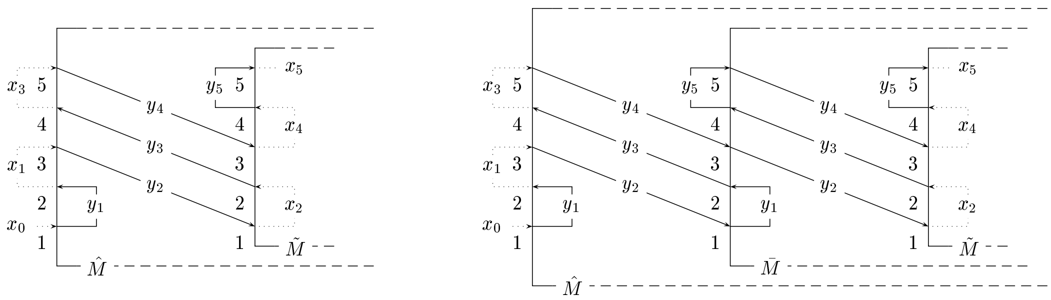

Figure 7: Accepting computations for $x_0y_1x_1y_2x_2y_3x_3y_4x_4y_5x_5$ (left) and $x_0y_1x_1y_2y_1y_3x_3y_4y_2x_2y_3y_5y_4x_4y_5x_5$ (right).

We apply the pumping lemma to the language $L_k = \{a_1^n \cdots a_k^n \mid n \in \mathbb{N}\}$ of Example 3.

Lemma 15 *Let $k \in \mathbb{N}$ be a constant, then L_{k+1} does not belong to $\mathscr{L}_{lf}(k\text{-NFA})$.*

Proof. Assume L_{k+1} belongs to $\mathscr{L}_{lf}(k\text{-NFA})$. Let n be the constant of Lemma 14 and consider the word $w = a_1^n \cdots a_{k+1}^n$. Since we may pump at most k portions of w the result would not belong to L_{k+1}. $\square$

Since the constructions in Example 3 are deterministic and loop-free, as an immediate corollary we obtain hierarchies of loop-free self-assembling automata.

Corollary 16 *Let $k \in \mathbb{N}$ be a constant, then $\mathscr{L}_{lf}(k\text{-DFA}) \subset \mathscr{L}_{lf}((k+1)\text{-DFA})$ and $\mathscr{L}_{lf}(k\text{-NFA}) \subset \mathscr{L}_{lf}((k+1)\text{-NFA})$.*

Now we return to the question concerning the limitations of loop-freeness. The answer has been given for the cases $k = 1, 2$. For $k > 2$ the question is answered by the next result. It proves also that the pumping lemma does not hold for unrestricted self-assembling finite automata.

Lemma 17 *Let $k \in \mathbb{N}$ be a constant. There exists a language $L \in \mathscr{L}(3\text{-DFA})$ which does not belong to $\mathscr{L}_{lf}(k\text{-NFA})$.*

Proof. The witness for the assertion is the language $L = \{a_1^n(a_2^n a_3^n)^+ \mid n \in \mathbb{N}\}$.

A 3-DFA accepting L is a simple modification of the 3-DFA accepting the language L_3 given in Example 3. Simply insert a transition $\delta(q_5, a_2) = q_3$ in module M_0 and a transition $\delta(q_2, a_2) = q_1$ in module M_2.

By using the pumping lemma it is easy to see that L cannot be accepted by any loop-free k-NFA for any $k \in \mathbb{N}$. $\square$

Corollary 18 *Let $k \geq 2$ be a constant, then $\mathscr{L}_{lf}(k\text{-DFA}) \subset \mathscr{L}(k\text{-DFA})$ and $\mathscr{L}_{lf}(k\text{-NFA}) \subset \mathscr{L}(k\text{-NFA})$.*

The hierarchy result for loop-free self-assembling finite automata can be adapted to the unrestricted case. Though the pumping argument requires loop-freeness, the acceptors for the languages L_k may not contain any useful loop even in the unrestricted case.

Lemma 19 *Let $k, k' \in \mathbb{N}$ be two constants and $\mathcal{M}$ be a k'-NFA accepting the language L_k. Then during all accepting computations $\mathcal{M}$ does not enter any existing state more than once.*

Proof. If some accepting computation would run through a loop, then trivially the corresponding portion of the input could be repeated. In this case either an input with forbidden ordering of the symbols $a_1, \dots, a_k$ or an input with different numbers for at least two input symbols would be accepted, too. $\square$

This observation suffices to apply the proof of the pumping lemma in order to show that L_k cannot be accepted by any k-NFA. On the other hand, the loops of rejecting computations could be replaced by non-accepting 1:0-modules and, thus, the result would be loop-free automata to which the pumping lemma can be applied directly.

Corollary 20 *Let $k \in \mathbb{N}$ be a constant, then $L_{k+1} \notin \mathscr{L}(k\text{-NFA})$ and, therefore, $\mathscr{L}(k\text{-NFA}) \subset \mathscr{L}((k+1)\text{-NFA})$ and $\mathscr{L}(k\text{-DFA}) \subset \mathscr{L}((k+1)\text{-DFA})$.*

So we have shown infinite hierarchies for the four types of self-assembling finite automata in question.

In the next section deterministic computations are compared with nondeterministic computations by means of closure properties.

4 Closure Properties under Boolean Operations

Some closure properties of language families defined by self-assembling finite automata are investigated. It turns out that the loop-free k-DFA languages are properly contained in the loop-free k-NFA languages. Thus, nondeterministic and deterministic language families are separated by different closure properties.

We start the investigation by showing that none of the families is closed under intersection.

Theorem 21 *Let $k \geq 2$ be a constant. Then exist languages $L, L' \in \mathscr{L}_{lf}(k\text{-DFA})$ such that $L \cap L' \notin \mathscr{L}(k\text{-NFA})$. In particular, none of the families $\mathscr{L}_{lf}(k\text{-DFA})$, $\mathscr{L}_{lf}(k\text{-NFA})$, $\mathscr{L}(k\text{-DFA})$ and $\mathscr{L}(k\text{-NFA})$ is closed under intersection.*

Proof. Let $L = \{a_1^n \cdots a_k^n a_{k+1}^m \mid m, n \in \mathbb{N}\}$ and $L' = \{a_1^m a_2^n \cdots a_{k+1}^n \mid m, n \in \mathbb{N}\}$. In order to construct a loop-free k-DFA for L and L' we only need minor modifications of the loop-free k-DFAs given in Example 3. But $L \cap L' = \{a_1^n \cdots a_{k+1}^n \mid n \in \mathbb{N}\} = L_{k+1}$ which by Corollary 20 cannot be accepted by any k-NFA. □

The goal is to prove different properties for nondeterministic and deterministic automata. In order to disprove the nondeterministic closure under complement we prove the closure under union. The following technical lemma prepares for the construction.

Lemma 22 *Let $k \in \mathbb{N}$ be a constant. For any (loop-free) k-NFA $\mathcal{M}$ there exists an equivalent (loop-free) k-NFA $\mathcal{M}'$ such that there is no assembling transition from the starting state.*

Proof. The first idea might be simply to resolve all assembling transitions from the starting state by assembling a copy of the required modules in advance. But this construction may fail if the modules have more than one entry state. In such case the behavior of module M may differ. For example, in the original automaton a state s may be reached which will be overlayed by a module entry state. Now the possible transitions from s depend on whether or not the module has been assembled before.

Therefore, in order to resolve an assembling transition from the starting state, we need a complete copy of module M_0 together with the already assembled module. Those copies are necessary for all assembling transitions from the starting state. In addition, a new starting state is needed. How fit all these copies together? Simply by connecting the new starting state to existing successors of the starting states of all copies. This construction yields a new module M_0 and, obviously, preserves the loop-freeness. □

Theorem 23 *Let $k \geq 2$ be a constant, then $\mathscr{L}_{lf}(k\text{-NFA})$ and $\mathscr{L}(k\text{-NFA})$ are closed under union.*

Proof. Let L and L' be two k-NFA-languages. As usual for nondeterministic devices the construction for $L \cup L'$ relies on the idea that an acceptor can initially guess whether the input belongs to L or L'.

The technical constraints are as follows. We have to provide a new initial state to deal with the case that one of the automata may reach its initial state again. Modules have to be renamed and, thus, the references in the assembling transitions.

Due to Lemma 22 we may assume that the acceptors $\mathcal{M}$ and $\mathcal{M}'$ for L and L' have no assembling transitions from their initial states. The acceptor $\mathcal{M}''$ for $L \cup L'$ consists of all modules $M_1, \dots, M_{m_1}$ of $\mathcal{M}$ and all modules $M'_1, \dots, M'_{m_2}$ of $\mathcal{M}'$ with appropriately renamed assembling transitions.

The modules M_0 and M'_0 are joined as follows: The new module M''_0 of $\mathcal{M}''$ contains disjoint copies of M_0 and M'_0 and a new initial state s_0. The transition functions of M_0 and M'_0 are joined unchanged (except eventually renaming). In addition, s_0 is appropriately connected to the successors of the starting states of M_0 and M'_0. If either L or L' contains the empty word the new initial state is made final.

Since the presented construction preserves loop-freeness the theorem follows. □

Corollary 24 *Let $k \geq 2$ be a constant, then $\mathscr{L}_{lf}(k\text{-NFA})$ and $\mathscr{L}(k\text{-NFA})$ are not closed under complement.*

Proof. Since the families are closed under union by $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$ the closure under complement would imply the closure under intersection what contradicts Theorem 21. □

Now we consider deterministic devices. One might expect their closure under complement by exchanging final and non-final states. But to this end the transition functions need to be total mappings.

In order to make them total we have to cope with the situation that states will eventually be overlayed by entry states of successor modules. Therefore, in general more than one transition function must be considered. On the other hand, some computations may enter these states without assembling the successor modules which implies that only one transition function must be total. The problem can be solved for loop-free k-DFAs.

Lemma 25 *Let $k \in \mathbb{N}$ be a constant and $\mathcal{M}$ be a loop-free k-DFA. Then there exists an equivalent loop-free k-DFA $\mathcal{M}'$ such that for any state in any reachable configuration the local transition function is totally defined.*

Proof. For states that neither have assembling transitions nor will overlay entry states of successor modules the transition function δ can be extended. If δ is not defined for some (s, a), we provide a non-accepting 1:0-module which is assembled for (s, a) and which recursively assembles itself for any further input.

Now we consider states with assembling transitions. If state s with input a assembles a module M', then the entry state s' of M' which overlays s may have transitions for several input symbols. But since the k-DFA is loop-free, state s (and thus s') is passed through at most once. Since the assembling consumes input symbol a, the other transitions from s' can safely be omitted. Subsequently, δ is made total for state s by assembling the non-accepting 1:0-module for all input symbols having no transition from s or s'.

For states which eventually overlay input states of successor modules without assembling them (for modules with more than one input state) the problem is more complicated to solve. Suppose module M' is assembled by a transition from state s in module M and one of its entry states r is overlayed by state p in M. Then the computation in state p can depend on whether M' was assembled or not. Therefore, we take two copies of module M and join them to form a new module M as follows. The first copy behaves like the old module M. From the second copy state s and all transitions to state s are removed. Finally, the assembling transition in question is modified such that all interface states of M' are overlayed by the corresponding states of the second copy.

Now it is obvious that states of the second copy are only reachable via module M'. So M' must have been assembled before and it is known which transitions exist for the state p.

On the other hand, states of the first copy cannot be reached after assembling M' since state s is the single connection state between the copies. But s has been passed through before. This is also the reason why s can safely be deleted from the second copy.

This construction step has successively to be performed for all assembling transitions in M. Subsequently, it is known for all states s and input symbols a whether the transition is defined when the computation enters s. If not, then the non-accepting 1:0-module is assembled in order to make the transition function total. □

Theorem 26 *Let $k \in \mathbb{N}$ be a constant, then $\mathscr{L}_{lf}(k\text{-DFA})$ is closed under complement.*

Proof. By Lemma 25 we may assume that the transition function is total. It suffices to change final and non-final states to prove the theorem. □

Since $\mathscr{L}_{lf}(k\text{-DFA})$ has been shown not to be closed under intersection but to be closed under complement we obtain immediately:

Corollary 27 *Let $k \geq 2$ be a constant, then $\mathscr{L}_{lf}(k\text{-DFA})$ is not closed under union.*

From the different closure properties and for structural reasons the separation of nondeterministic and deterministic loop-free languages follows.

Theorem 28 *Let $k \geq 2$ be a constant, then $\mathscr{L}_{lf}(k\text{-DFA}) \subset \mathscr{L}_{lf}(k\text{-NFA})$.*

For deterministic unrestricted k-DFAs, $k \geq 2$, the closures under complement and union remain open. From the shown non-closure under complement follows that the language cannot be closed under both operations. Since for deterministic devices the closure under complement is most likely, our conjecture is that they are not closed under union. The positive closure under complement would also imply a separation of unrestricted nondeterministic and deterministic language families.

Another partially open question concerns the comparison between unrestricted deterministic computations and loop-free nondeterministic computations, thus, loop-freeness versus nondeterminism.

References

[1] Casjens, S. and King, J. *Virus assembly*. Annual Review of Biochemistry 44 (1975), 555–604.

[2] Nebel, M. E. *On the power of subroutines for finite state machines*. J. Aut., Lang. and Comb. 6 (2001), 51–74.

[3] Penrose, L. S. *Self-reproducing machines*. Scientific American 200 (1959), 105–113.

[4] Rubinstein, R. S. and Shutt, J. N. *Self-modifying finite automata*. Proc. of the IFIP 13th World Computer Congress. Vol. 1 : Technology and Foundations, 1994, pp. 493–498.

[5] Rubinstein, R. S. and Shutt, J. N. *Self-modifying finite automata – power and limitations*. Technical Report WPI-CS-TR-95-4, Computer Science Department, Worcester Polytechnic Institute, 1995.

[6] Rubinstein, R. S. and Shutt, J. N. *Self-modifying finite automata: An introduction*. Inform. Process. Lett. 56 (1995), 185–190.

[7] Saitou, K. and Jakiela, M. J. *On classes of one-dimensional self-assembling automata*. Complex Systems 10 (1996), 391–416.

[8] Wang, Y., Inoue, K., Ito, A., and Okazaki, T. *A note on self-modifying finite automata*. Inform. Process. Lett. 72 (1999), 19–24.