Mata: A Fast and Simple Finite Automata Library

David Chocholatý

Faculty of Information Technology, Brno University of Technology

Excel@FIT 2024

What is Mata

Mata is a well-engineered, fast, and simple automata library in C++. It is maintainable and understandable. It has a simple architecture allowing a new user, a researcher, to quickly prototype new algorithms and thoroughly optimize the final implementation. Mata targets string constraint solving, reasoning about regular expressions, regular model checking, student projects, and research prototypes. It comes with a large benchmark from string constraint solving, regular model checking, and reasoning about regular expressions.

Tool

Available on GitHub.

Distinctive Features

- **Fast** and **simple**.
- **Explicit** representation of the transition relation.
- SOTA algorithms to work with **nondeterminism**.
- Modern development workflow and technologies.
- Easily **extensible** and **modifiable**.
- Well-documented, examples, testing infrastructure.
- **High-level API** with sane defaults, **low-level API** for maximal optimization.
- **Python interface**.
- A basis for a modular automata format `.mata`.

Usage

```cpp
#include "mata/parser/mintermization.hh"
#include "mata/nfa/nfa.hh"
#include "mata/nfa/builder.hh"

#include <fstream>

using namespace mata;
using namespace mata::nfa;

int main(int argc, char *argv[]) {
    std::fstream file(argv[1], std::ios::in);
    for (IntermediateAut& inter_aut:
            IntermediateAut::parse_from_mf(parser::parse_mf(
              file, true))) {
        if (inter_aut.alphabet_type == IntermediateAut::
          AlphabetType::BITVECTOR) {
            inter_aut = Mintermization{}.mintermize(inter_aut)
              ;
        }
        EnumAlphabet alphabet{ 'a', 'b', 'c' };
        Nfa aut{ nfa::builder::construct(inter_aut, &alphabet)
           };
        nfa::determinize(aut.trim());
        State new_final{ aut.add_state() };
        aut.delta.add(*aut.initial.begin(), 'a', new_final);
        aut.final.insert(new_final);
        Nfa word_aut{ nfa::builder::create_single_word_nfa(
          Word{ 'c', 'b', 'b', 'a' }) };
        if (nfa::is_included(word_aut, aut) && aut.is_in_lang(
          Word{ 'a', 'b' })) {
            aut = nfa::concatenate(aut, aut);
            aut.print_to_mata(std::cout);
        }
    }
    return EXIT_SUCCESS;
}
```

(a) An example of using Mata.

```
@NFA-explicit
%Alphabet-auto
%Initial q0
%Final q5 q6 q7
q0 97 q0
q0 97 q3
q0 98 q1
q0 99 q0
q1 97 q2
q1 97 q4
q1 97 q7
q1 98 q1
q1 98 q5
q1 99 q4
q2 97 q4
q2 97 q7
q2 98 q5
q2 99 q4
q3 97 q4
q3 97 q7
q3 98 q5
q3 99 q4
q4 97 q4
q4 97 q7
q4 98 q5
q4 99 q4
q5 97 q6
q5 98 q5
```

(b) The output in `.mata` format.

Figure 1: An example of using the C++ interface for Mata.

Architecture

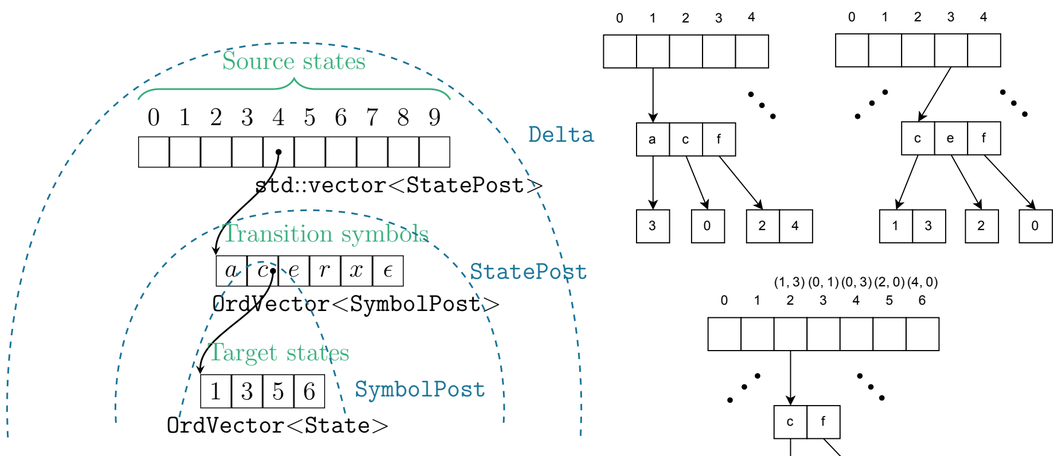


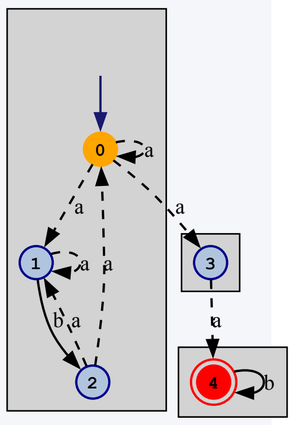
Figure 2: Delta, the transition relation.

Figure 3: Subset construction in Delta.

Python Interface

```python
from libmata import nfa, alphabets, parser, plotting
aut1 = parser.from_regex('((a+b)*a)*')
aut2 = parser.from_regex('aab*')
con_aut = nfa.nfa.concatenate(aut1, aut2).trim()
plotting.store()['alphabet'] = \
    alphabets.OnTheFlyAlphabet.from_symbol_map({'a':97, '
      b':98})
e_h = [
    (lambda aut, e: e.symbol == 98, {'color':'black'}),
    (lambda aut, e: e.symbol == 97, {'style':'dashed','
      color':'black'})
]
n_h = [
    (lambda aut, q: q in aut.final_states,
        {'color':'red','fillcolor':'red'}),
    (lambda aut, q: q in aut.initial_states,
        {'color': 'orange', 'fillcolor': 'orange'}),
]
plotting.plot(con_aut, with_scc=True,
              node_highlight=n_h, edge_highlight=e_h)
```

(a) An example of using Mata from Python.



(b) The output.

Figure 4: An example of a Python interface for Mata.

Results per Benchmark

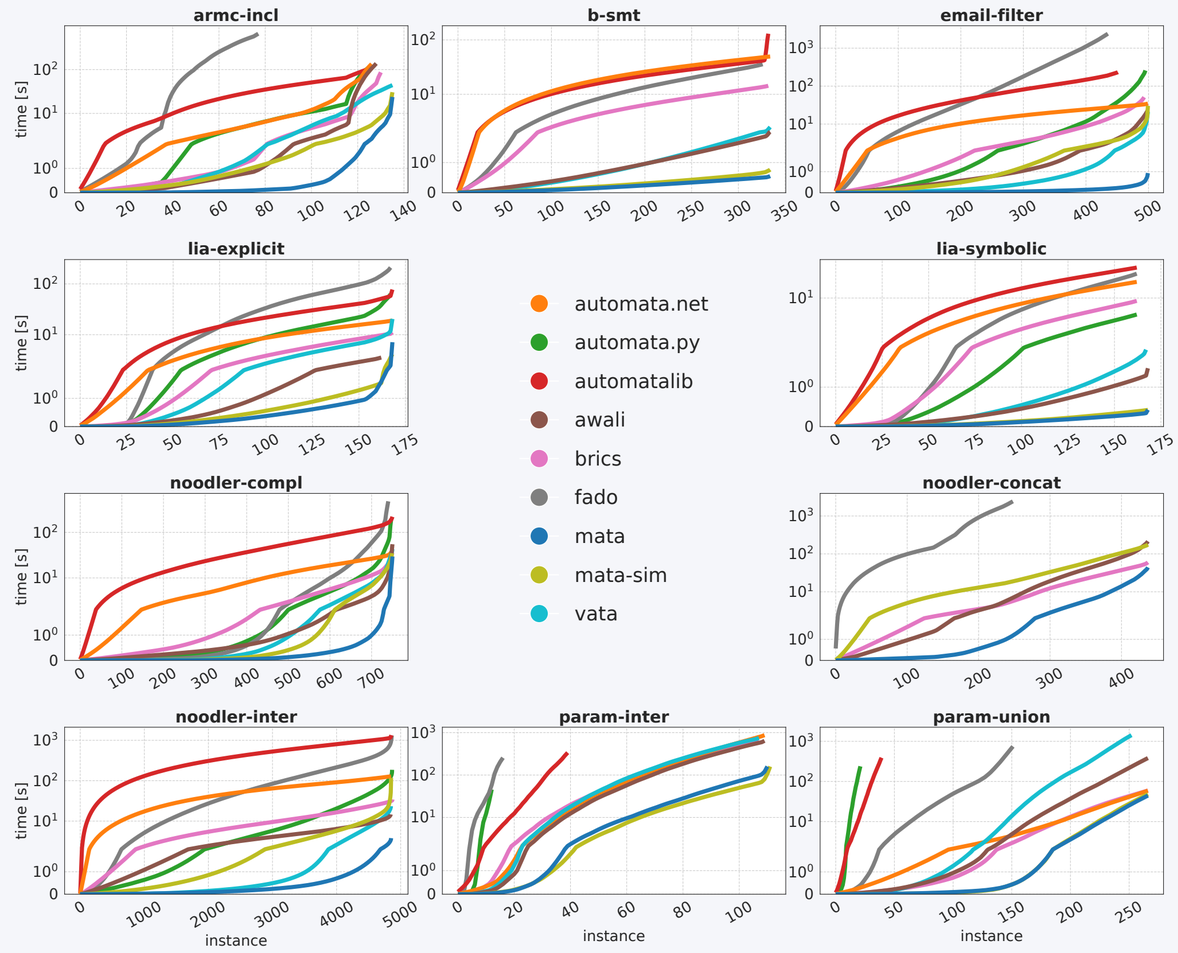


Figure 5: Cactus plot showing cumulative run time per benchmark. The time axis is logarithmic.

Table 1. Statistics for the benchmarks.

	armc-incl (136)				b-smt (384)				email-filter (500)				lia-explicit (169)				lia-symbolic (169)			
	TO	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*
Mata	**0**	**174**	**2**	1 s	**0**	**1**	**1**	1	**0**	**1**	∼**0**	9	**0**	42	**6**	356	**0**	**2**	**2**	6
Awali	7	1 s	17	3 s	**0**	6	6	4	**0**	46	4	162	6	**21**	21	16	**0**	8	7	14
Vata	**0**	324	43	577	**0**	7	7	10	**0**	42	2	322	**0**	121	51	671	1	11	10	11
Automata.net	9	1 s	125	3 s	**0**	148	153	30	**0**	69	66	30	**0**	113	117	49	6	103	107	33
Brics	5	659	34	2 s	4	43	43	19	6	103	17	280	**0**	66	62	63	6	55	60	33
AutomataLib	10	843	669	1 s	7	390	126	3 s	48	516	390	521	**0**	458	285	1 s	6	164	173	52
FAdo	58	8 s	22 s	10 s	9	109	112	67	64	6 s	1 s	11 s	1	1 s	727	2 s	6	135	149	105
Automata.py	10	913	133	3 s	334	24	TO	15	4	520	19	2 s	1	372	167	894	6	35	35	25
	noodler-compl (751)				noodler-conc (438)				noodler-inter (4872)				param-inter (267)				param-union (267)			
	TO	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*	*TO*	*Avg*	*Med*	*Std*
Mata	**0**	**39**	∼**0**	401	**0**	**100**	**10**	286	**0**	∼**0**	∼**0**	3	**156**	**1 s**	TO	4 s	**0**	**166**	**7**	326
Awali	**0**	73	2	638	**0**	490	55	1 s	6	3	1	7	157	6 s	TO	7 s	**0**	1 s	81	3 s
Vata	**0**	57	2	296		-			2	4	∼**0**	22	159	7 s	TO	8 s	14	6 s	270	12 s
Automata.net	**0**	53	39	110		-			**0**	26	24	9	157	8 s	TO	10 s	**0**	220	47	314
Brics	**0**	47	8	190	**0**	136	35	204	**0**	7	3	21	159	6 s	TO	6 s	**0**	223	50	307
AutomataLib	**0**	293	143	793		-			17	276	216	675	227	8 s	TO	13 s	227	10 s	TO	15 s
FAdo	10	646	5	4 s	189	10 s	25 s	13 s	10	271	52	2 s	250	15 s	TO	20 s	115	5 s	12 s	11 s
Automata.py	3	263	5	2 s		-			5	38	3	353	254	4 s	TO	6 s	245	11 s	TO	16 s

Table 2. Relative speedup of Mata on instances where both libraries finished.

	Awali	Vata	Automata.net	Brics	AutomataLib	FAdo	Automata.py
armc-incl	27.52	1.86	29.73	16.98	21.44	4839.55	23.22
b-smt	3.7	4.52	89.64	26.13	236.36	70.16	24.47
email-filter	25.07	22.59	37.19	55.3	273.35	9999.29	282.41
lia-explicit	2.22	2.88	2.69	1.57	10.89	85.17	25.38
lia-symbolic	3.46	4.65	51.82	27.99	82.47	67.54	17.97
noodler-compl	1.85	1.45	1.37	1.22	7.44	137.53	15.58
noodler-conc	4.87	-	-	1.36	-	1979.56	-
noodler-inter	4.02	6.42	33.98	9.04	371.23	363.49	51.51
param-inter	5.36	7.3	7.27	6.49	1.43	2148.64	58.85
param-union	8.61	51.77	1.33	1.34	833.69	1618.04	5860.62

Results per Operation

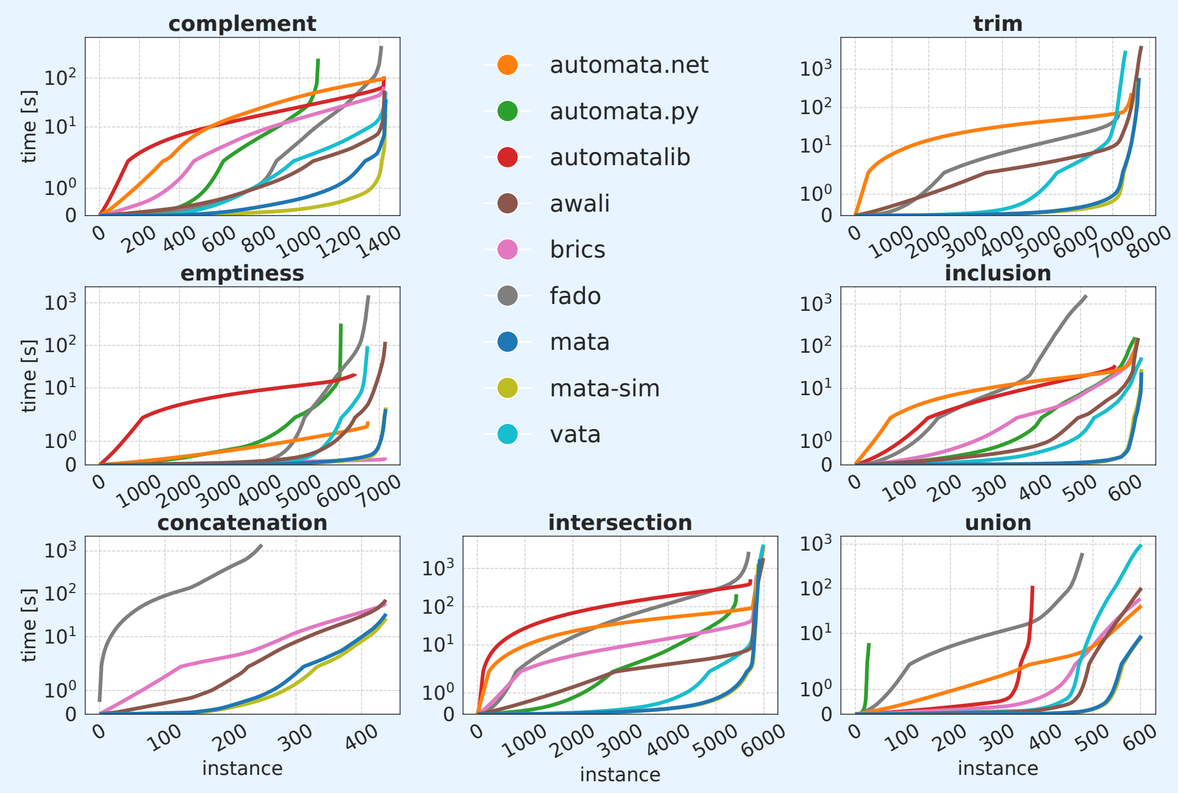


Figure 6: Cactus plot showing cumulative run time per operation. The time axis is logarithmic.

Table 3. Statistics for the operations on solved instances.

	complement			concatenation			emptiness			inclusion			intersection			trim			union		
	Avg	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*	*Avg*	*Med*	*Std*
Mata	**25**	**1**	315	**78**	**8**	235	∼**0**	∼**0**	2	**37**	∼**0**	576	295	∼**0**	3 s	76	∼**0**	828	**14**	∼**0**	45
Awali	38	2	462	166	22	402	17	∼**0**	138	250	2	2 s	312	∼**0**	2 s	516	∼**0**	4 s	173	∼**0**	527
Vata	36	3	294		-		14	∼**0**	130	85	1	374	699	∼**0**	4 s	408	∼**0**	3 s	2 s	∼**0**	5 s
Automata.net	73	59	89		-		∼**0**	∼**0**	∼0	245	43	1 s	621	14	4 s	31	9	165	69	6	163
Brics	46	24	140	136	35	204	∼**0**	∼**0**	∼0	204	10	1 s	115	4	1 s		-		99	2	232
AutomataLib	75	31	657		-		3	2	5	60	42	102	91	59	748		-		311	2	3 s
FAdo	320	3	2 s	6 s	10 s	10 s	223	∼**0**	2 s	3 s	84	8 s	479	48	3 s	**10**	3	70	1 s	84	6 s
Automata.py	226	25	2 s		-		53	∼**0**	1 s	263	6	1 s	**39**	2	479		-		203	TO	377