Mata

A Fast and Simple Finite Automata Library

github.com/VeriFIT/mata/

**David Chocholatý, Tomáš Fiedor, Vojtěch Havlena,
Lukáš Holík, Martin Hruška, Ondřej Lengál and Juraj Síč**

Faculty of Information Technology, Brno University of Technology, Czech Republic

Why another automata library

- Other libraries are **slow and/or complicated**
- **Fast**
  - bottleneck in applications: automata operations
  - **optimize** both high-level and low-level operations
- **Simple**, easy to
  - start using
  - implement complicated techniques (simulations, antichains, complex algorithms, ...)
  - **extend** with new features (registers, counters, transducers, ...)
  - quickly **prototype** new algorithms
  - **maintain** by a small teams of researchers
  - pick up by other researchers with **unpredicted ideas** to implement
  - introduce students into

Vision for Mata

- Efficient platform for automata research, in areas of
  - **string solving**
  - (abstract) regular model checking
  - deciding automata logics, Presburger, WS1S, MSO
  - analyzing regexes
- Simple data structures
- Allowing implementations close to textbook
- Handling **large alphabets**
  - bit-vectors, mintermization (string solving), BDDs-like symbolic representation (WS1S)
- Solid **infrastructure**
  - regex parsing, textual format, tests, performance tests, visualization options

Existing automata libraries

Brics (Java)

- both NFA and DFA
- transitions in a set
- character ranges

Automata.net (C#)

- symbolic NFA
- transitions in a hash map
- effective boolean algebras (implicitly BDDs)

Vata (C++)

- tree automata (i.e. NFA)
- fast simulation reduction and antichain-based inclusion checking

Awali (C/Python)

- weighted automata (i.e. NFA)
- transition in a vector,
- keeps indices to this vector

AutomataLib (Java)

- only DFA
- transitions in a 2D matrix

Automata.py (Python)

- both NFA and DFA
- transitions as a mapping

FAdo (Python)

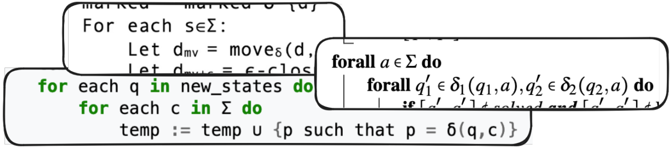
- similar to **Automata.py**

Mata today

- First step towards our vision
- Implemented in C++, both **C++ and Python interface**
- Support for **NFAs and DFAs**
  - with all basic operations implemented (and some more)
- Methods for **regex processing, textual format, parsing**
- Only **explicit alphabet**
  - enough for our use cases
  - ”everything else” symbols + mintermization
- Specific operations for string solving
  - used in **Z3-Noodler**: yesterday’s presentation, see poster

Representing transitions

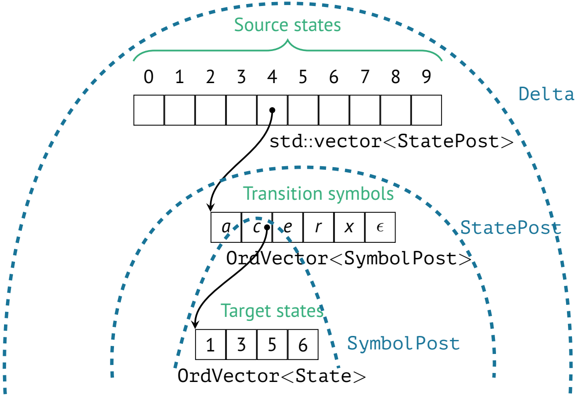
- Observation:
  - algorithms **iterate** usually over **all transitions**
  - first over **symbols**, then over the **target states**



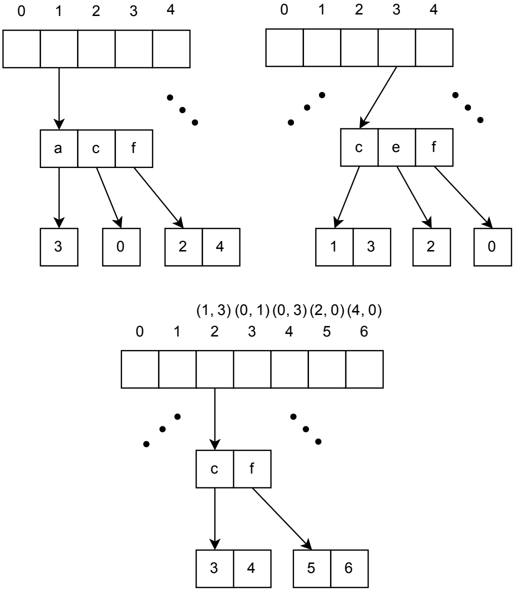
- Our approach:
  - keep only **used symbols** for each source state
  - have them **ordered**: **synchronous** iteration over transitions of multiple source states
  - for each symbol, remember the set of (ordered) targets

Delta: transition relation

- **States** and **symbols** are **integers**
- **Each** number from [0-n] is a state
- OrdVector
  - ordered vector
  - efficient synchronous iteration
  - slow general insert and erase
  - fast insert and erase at end
- StatePost
  - contains all symbols with non-empty target states
  - ordered by symbols
- SymbolPost
  - pairs: (symbol, target states)



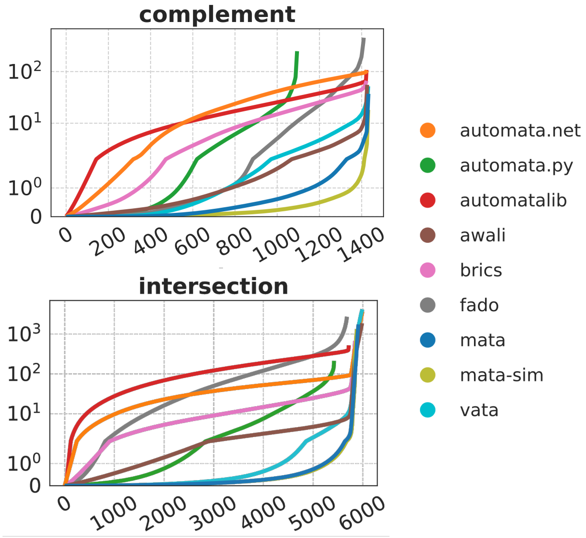
Post-image generation



- `Delta` built for computing a **post-image of a set of states**
- set of states S, compute $post(S)$
- iterate trough all $post(s)$ for $s \in S$
- they are ordered, easy to iterate **together**
- new macrostate transition inserted at the **end**

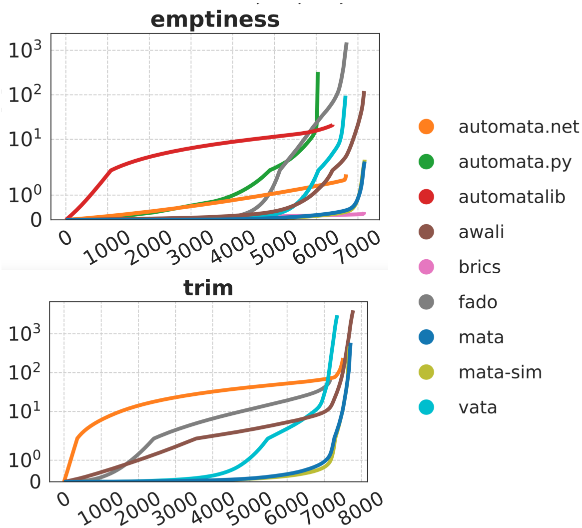
Post-image generation

- Post-image in subset construction
  - set of states S
  - used in complement (determinization)
- Post-image in intersection
  - similar, but done for a **pair of states**



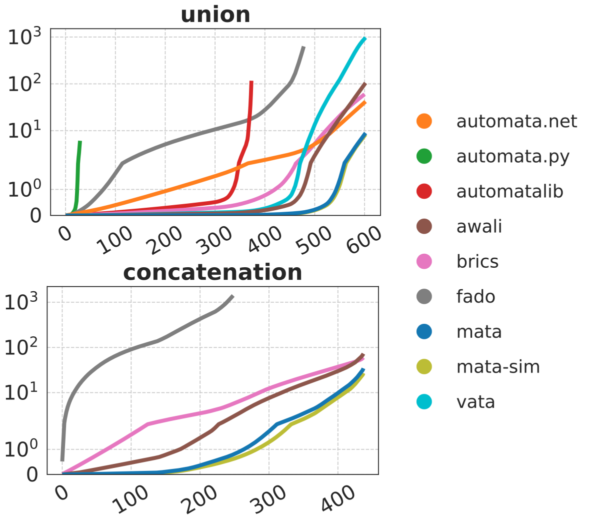
Emptiness and trimming

- Both used **frequently**, must be fast
- Simplified **Tarjan's algorithm for discovering SCCs**
  - single DFS pass
  - emptiness: find **reachable final state**
  - trim: find **useful states**
- Removing useless states in trim
  - in a `Delta`-friendly way
  - **single** pass
  - **in-place**



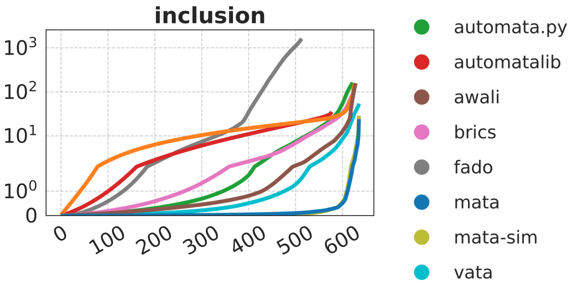
Union and concatenation

- **Copying is slow**
  - we are not sure why
  - imperfect memory locality?
- **Union** and **concatenation** use copying
- Solution: do them **in-place**
- Drawback: loss of the original automaton
  - often not a problem: **inductive regex construction**



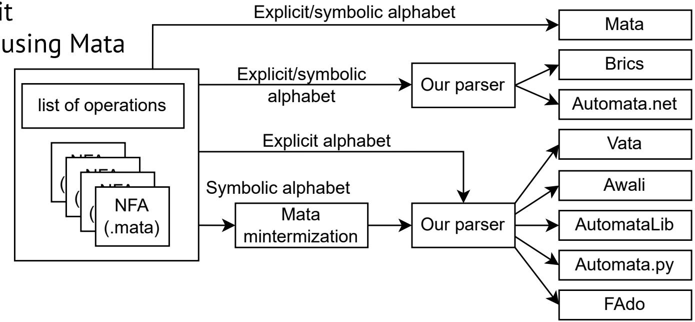
Simulation and antichain-based inclusion

- Computing **simulation relation**
  - implementation from Vata
  - used for reducing NFAs
- Antichain-based inclusion
  - uses efficient subset construction
  - optimized by subsumption pruning



Experiments

- Input: automata (in our **.mata format**) with list of operations from various sources
- Some benchmarks have **symbolic alphabets**
  - Brics and Automata.net can handle
  - other tools need explicit
  - automata **mintermized** using Mata
- **Parsers/Conversions** created by us
  - not counted to the runtime of tools



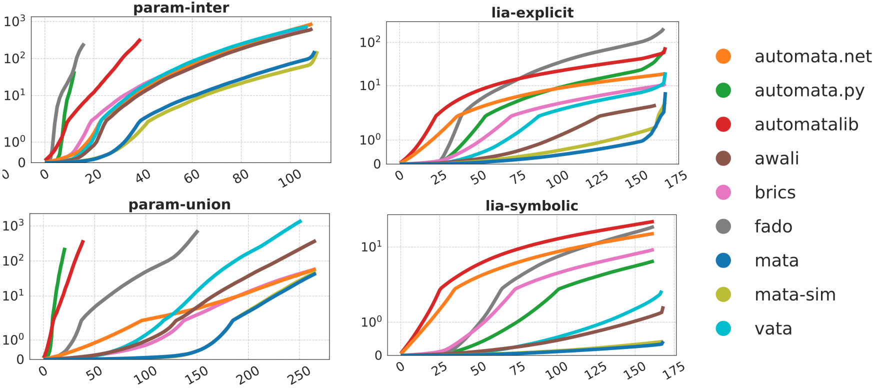
Experiments: parametric regexes and LIA

param:

- union/inters. of parametric regexes

lia:

- complement from LIA solver Amaya
- symbolic or explicit alphabet



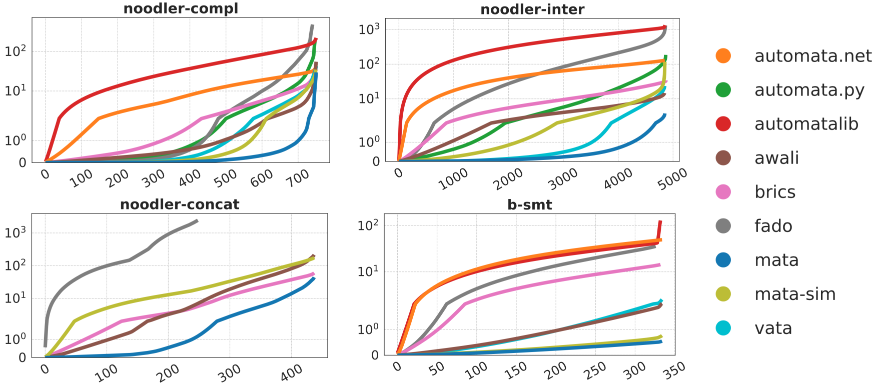
Experiments: string solving

noodler:

- automata from the string solver **Z3-Noodler**

b-smt:

- simple SMT string solving benchmarks



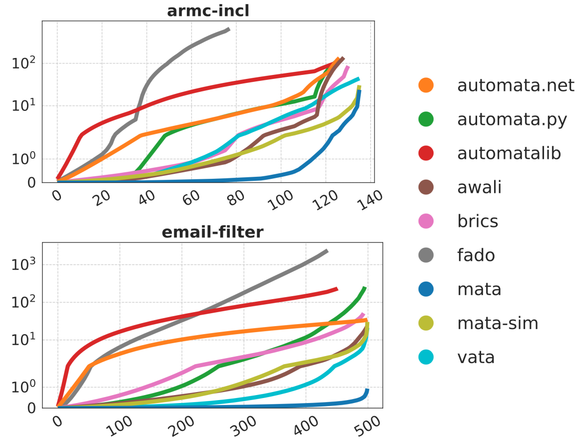
Experiments: inclusions

armc-incl:

- inclusion problems from abstract regular model checker

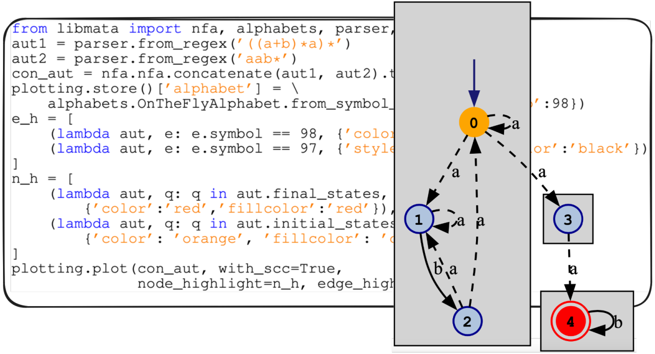
email-filter:

- inclusion problems inspired by spam filtering



Python interface

- **Efficient**
  - Cython wrapping C++ API
- Easy to install:
  `pip install libmata`
- Kept up to date with C++ code
- Useful for **prototyping**
- Nice **visualizations**
- Easy to use in jupyter notebooks



Future (and current) work

- **Transducers** (WIP, important for string solving), BDDs
  - utilizing `Delta` with **levels** for states
    - transition as a **sequence of NFA transitions**
  - **BDD**-like operations
- Alternation (empowered by IC3)
- Registers, counters

Mata: A Fast and Simple Finite Automata Library

github.com/VeriFIT/mata/

Table 1: Statistics for the benchmarks. We list the number of timeouts (TO), average time on solved instances (*Avg*), median time over all instances (*Med*), and standard deviation over solved instances (*Std*), with the best values in **bold**. The times are in milliseconds unless seconds are explicitly stated. We use ~0 to denote a value close to zero.

	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

Table 3: Statistics for the operations on solved instances. We list the average time (*Avg*), median time (*Med*), and standard deviation (*Std*), with the best values in **bold**. The times are in milliseconds. Note that only the operations that the given library finished within the timeout are counted, hence the numbers are significantly biased in favour of libraries that timeouted more (the harder benchmarks are no counted in), and should be red in the context of Table 1 and the cactus plots. We use ~0 to denote a value close to zero.

	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