



Deliverable D1.4

Implementation: Extension of ROLL Library

Deliverable

<i>Number and title:</i>	D1.4 – Implementation: Extension of ROLL Library		
<i>Work package:</i>	WP1 (Energy and Resources)		
<i>Lead author:</i>	Andrea Turrini (IISG)		
<i>Reviewers:</i>	Pablo Diale (ASC), Thomas Noll (RWTH)		
<i>Due date (GA):</i>	M49 (2025-10-31)	<i>Dissemination level:</i>	Public
<i>Due date (revised):</i>	2025-11-30	<i>Version:</i>	1.0
<i>Submission date:</i>	2025-11-28	<i>Pages:</i>	11

Version history

<i>Version</i>	<i>Date</i>	<i>Notes</i>
1.0	2025-11-28	First official release

Project

<i>Title:</i>	Models in Space Systems: Integration, Operation, and Networking		
<i>Acronym:</i>	MISSION	<i>Start date:</i>	01-10-2021
<i>GA no.:</i>	101008233	<i>Duration:</i>	56 months
<i>Call:</i>	H2020-MSCA-RISE-2020	<i>Website:</i>	mission-project.eu



This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 101008233.

This document reflects only the authors' view. It does not represent the opinion of the European Union, and the European Union is not responsible for any use that may be made of the information it contains.

1 Introduction

Finite state automata are a well known and elegant formalism used to model systems when studied in theoretical computer science. Depending on their semantics, they can be adopted for instance as computational models to represent the behavior of programs and reactive systems, and to identify whether such behaviors are acceptable or need to be rejected. This can be achieved by e.g. checking whether the behavior is in the language of the automaton, i.e., the set of words whose letters are from a given alphabet Σ that make the automaton reach specific states marked as accepting.

In literature there are several types of automata that have been developed for different purposes and with different interpretations, so finite state automata can be partitioned into multiple classes depending on a variety of criteria. The first, main syntactic classification criterion is whether the automaton is deterministic or it exhibits nondeterministic behaviors, that is, whether from each state there is only one successor state or there may be multiple possible successor states for a given letter, respectively. The second, main semantic classification criterion is the type of the language $\mathcal{L}(\mathcal{A})$ the automaton $\mathcal{A}$ recognizes: whether $\mathcal{L}(\mathcal{A})$ is composed by finite words ($\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$), or by infinite words ($\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$). The third, main temporal classification criterion is whether the automaton considers the passage of time as a sequence of discrete ticks or as a continuous flow.

As examples of automata classified differently according to the three criteria, deterministic finite automata (DFAs) and nondeterministic finite automata (NFAs) are discrete automata accepting regular languages, i.e., languages whose words have finite (but not bounded in advance, in general) length and are generated by a regular expression. Nondeterministic Büchi automata (NBAs) are the simplest type of automata accepting infinite words belonging to a given ω -regular language and the same type of languages is also recognized by other deterministic ω -automata such as Rabin, Street, Muller, and parity automata, while deterministic Büchi automata (DBAs) are strictly less expressive. Real-time automata (RTAs) are automata working with continuous time and the accepted words are of finite length and also include time delays between consecutive letters. This means that, while two words with the same sequence of actions lead to the same state in a DFA, different timing information can make two timed words make a deterministic RTA reach different states, with possibly different acceptance outcomes.

When working with finite state automata, for example by using them in model checking to verify the correctness of a system (see, e.g., [6, 7, 10]), the first issue one needs to tackle is how to get the automaton modelling the “good” and “bad” behaviors of the system that need to be accepted and rejected, respectively. If the “good” behaviors are exactly those satisfying a given temporal logic formula, such as a linear temporal logic (LTL) formula, then the corresponding NBA can be obtained mechanically; if nondeterminism is not suitable for the analysis, like when analyzing systems exhibiting both probabilistic and nondeterministic behaviors (see, e.g., [6]), then the NBA can be translated to an equivalent deterministic Rabin/Street/Muller/parity ω -automaton (see, e.g., [7, 10]). If the intended behaviors are not expressed by a logical formula but directly as a black-box automaton or by its language, for which we have an oracle answering membership and equivalence queries, then active learning algorithms can be used to learn an automaton equivalent to the target one. For a language $\mathcal{L} \subseteq \Sigma^*$, the notorious A^* algorithm by Angluin [4] can be applied to learn a DFA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$. For an ω -regular language $\mathcal{L} \subseteq \Sigma^\omega$, a learning algorithm for Büchi automata based on families of DFAs [13] can be used to learn a limit deterministic Büchi automaton $\mathcal{A}$ accepting the same ultimately periodic words in $\mathcal{L}$; ROLL [14] provides the implementation for multiple variants of such algorithm.

Sometimes, however, the whole language $\mathcal{L}$ is not available, but only two finite, disjoint sets of words that provide positive samples that must be accepted by the DFA $\mathcal{A}$ and negative samples that must be rejected by $\mathcal{A}$ are available; whether to accept or reject each of the remaining words not in the two sets is not specified. This scenario happens for instance when simulating or interacting with a system, so the positive and negative sample sets depend on the finite number of executions obtained from the system, for instance in computational biology [8], inference of network invariants [11], regular model

checking [18], and reinforcement learning [12]. In this situation, active learning is not suitable (as membership and equivalence queries cannot be answered reliably for words not sampled); instead, passive learning is tailored to such a scenario, where one looks to infer a minimum-size separating DFA $\mathcal{A}$ consistent with the given sets of positive and negative samples. Finding such a minimal separating DFA can be done in two steps: one can first learn a concise representation of an almost deterministic finite automaton with *three* values: *accept*, *reject*, and *don't-care*, that are used to match the given positive and negative samples and the other remaining words, respectively. That is, one first represents them as a 3-valued almost deterministic finite automaton (3ADFA), and then, as second step, the problem of finding a minimal DFA that agrees with the 3ADFA outside of don't-care entries is reduced to a Boolean Satisfiability (SAT) problem. This approach has been proposed in [9] and implemented in DFAMiner.

One main drawback of inferring a minimum-size separating DFA consistent with a given sets of positive and negative samples is that the obtained automaton can be quite large, and this affects its simplicity, interpretability, and interoperability, as a single DFA representing complex system behaviors can have a very intricate structure. To address this issue, recent work [12] has moved toward the DFA *decomposition identification problem* (DIP), i.e., inferring a group of DFAs from the samples, where their conjunction language includes all positive samples and excludes all negative samples. This approach allows for capturing sub-tasks performed by the system, with the overall system behavior being described as the intersection of the languages from several DFAs, one for each sub-task. Since the resulting DFAs are likely to be small and easy to interpret, the overall behavior is also interpretable, as language intersection is easy to conceive. In a recent work [16], we have shown how the approach by Lauffer et al. [12] can be considerably improved by using again DFAs with 3 values and an appropriate SAT encoding when looking for the Pareto optimal decomposition with a given number of DFAs in the decomposed DFA. We have also proposed in [16] a new optimization objective for the DFA decomposition identification problem: the states optimal DIP. In this optimization problem, the aim is to find the minimum total number of states in a decomposition consistent with the given sets of samples and, among the decompositions having the same minimum number of states, one that has the highest entropy. The experimental evaluation provided in [16], and briefly presented in Section 3, shows that our approach scales much better in terms of running time than the one in [12]; also solving the minimum states optimal DIP can be done efficiently.

Inspired by recent work on learning real time automata [3], we considered in [17] to apply DFAs with 3-valued states to the synthesis of deterministic real time automata from sets of positive and negative timed samples, together with the corresponding SAT encoding. Compared with previous work (see, e.g., [19, 20]), our solution advances the RTA learning from samples with respect to multiple aspects: it introduces 3DRTAs, the real-time counterparts of 3DFAs; it provides a procedure to construct them from the input set of samples; it gives the means to synthesize minimal DRTAs with theoretical guarantees on correctness and optimality; it specifies an efficient SAT encoding for DRTA synthesis taking advantage of the 3DRTA structure; and it indicates how to merge regions so to reduce the excessive number of transitions that may arise in coarse DRTAs. The experimental evaluation presented in [17] and summarized in Section 3 shows that compared with the existing FLEXFRINGE tool¹ implementing the technique developed in [20], our approach generates deterministic RTAs with considerably fewer states, with cases where FLEXFRINGE produced more than 300 times the number of states generated by our tool.

In this deliverable, we provide a unified automata learning tool supporting the learning of Büchi automata through ROLL, of minimal separating DFAs via DFAMiner, and of separating decomposed DFAs and RTAs by means of our specific implementations; the remainder of this document is devoted to present our automata learning tool.

¹<https://github.com/tudelft-cda-lab/FlexFringe>

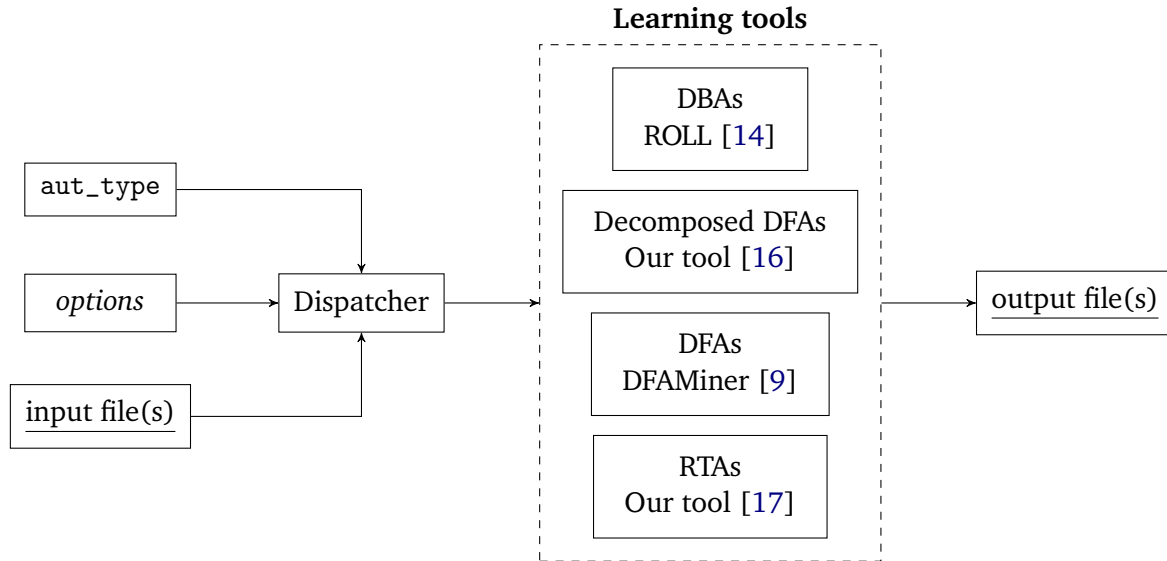


Figure 1: The architecture of our automata learning tool

2 The Automata Learning Tool

In this section we describe our automata learning tool in detail.

2.1 Architecture

Our unified automata learning tool is written in Python and is available publicly at https://github.com/andreat/automata_learning under the MIT license. Its architecture is shown in Figure 1 and comprises two main components: the dispatcher and the specific learning tools.

The dispatcher: this component is responsible for invoking the appropriate learning tool according to the type of automaton the user wants to obtain and for providing the specified options based on the format the invoked learning tool expects. To ensure maximal flexibility in supporting the addition of new learning tools and to adapt to new versions of the ones already included in our unified automata learning framework, the dispatcher interacts with the underlying learning tools by calling each of them as an independent subprocess. In this way the dispatcher, and so our unified tool, is not bound by the choices made by the developers of the single learning tools and their dependencies and requirements. Moreover, in case the underlying tools have conflicting requirements, the user of our unified software can just create multiple virtual environments according to the different dependencies of the specific program; then the user can just call our unified learning tool in each of them accordingly, in a uniform way and without having to modify our code.

The learning tools: currently we support four different learning tools that are specifically developed for four types of finite state automata: ROLL [14] for DBAs, DFAMiner [9] for DFAs, our tool from [16] for decomposed DFAs, and the tool we developed for DRTAs in [17]. Except for ROLL that is currently implemented in Java, all other tools are implemented in Python.

2.2 Tool usage

To use the automata learning tool, it suffices to call it at command line as follows:

```
python3 learning.py aut_type options
```

```
usage: learning.py [-h] {dba,ddfa,dfa,drta} ...

Learn different types of automata

options:
  -h, --help            show this help message and exit

automaton type:
  type of automata to work with

  {dba,ddfa,dfa,drta}
    dba                  use ROLL to work with Buchi automata
    ddfa                 synthesize decomposed deterministic finite automata
    dfa                  use DFAMiner to synthesize deterministic finite automata
    drta                 synthesize deterministic real time automata
```

Figure 2: Automata learning tool main help

```
usage: learning.py ddfa [-h] -i file [-m {pareto-apta,pareto-3dfa,states-optimal}]
      [--n-min int] [--n-max int] [-s {Cadical195,Glucose4}] [--optimized]

options:
  -h, --help            show this help message and exit
  -i file, --in file    the input file with the positive and negative samples
  -m {pareto-apta,pareto-3dfa,states-optimal}, --method
                        {pareto-apta,pareto-3dfa,states-optimal}
                        the solution method to use (default: pareto-3dfa)
  --n-min int, --n-dfas-min int
                        minimum number of DFAs to use (default: 2)
  --n-max int, --n-dfas-max int
                        maximum number of DFAs to use (default: unlimited)
  -s {Cadical195,Glucose4}, --solver {Cadical195,Glucose4}
                        SAT solver to use (default: Cadical195)
  --optimized           whether to optimize the construction
```

Figure 3: Automata learning tool help for synthesizing decomposed DFAs

where `aut_type` is the type of the automaton to be learned and *options* are the specific options available for the associated tool. The list of available types can be obtained by using the `-h` option, which prints the usage help. Thus, by calling

```
python3 learning.py -h
```

we get the help listed in Figure 2, providing the four automata types `dba`, `ddfa`, `dfa`, and `drta`, associated with the corresponding tools as shown in Figure 1.

Each tool offers different options to control its behavior and usually expects to receive an input file with the automaton/samples to be used in the analysis. As before, how to call the tool and the list of available options can be obtained again with the `-h` option; for instance, by calling

```
python3 learning.py ddfa -h
```

we get the usage details for learning decomposed DFAs, shown in Figure 3. From this help, we can see that we must provide an input file with the positive and negative samples through the `-i/--in` option; in addition, we can specify what kind of optimization problem we want the solution method for (option `-m`), what kind of SAT solver to use, and so on and so forth.

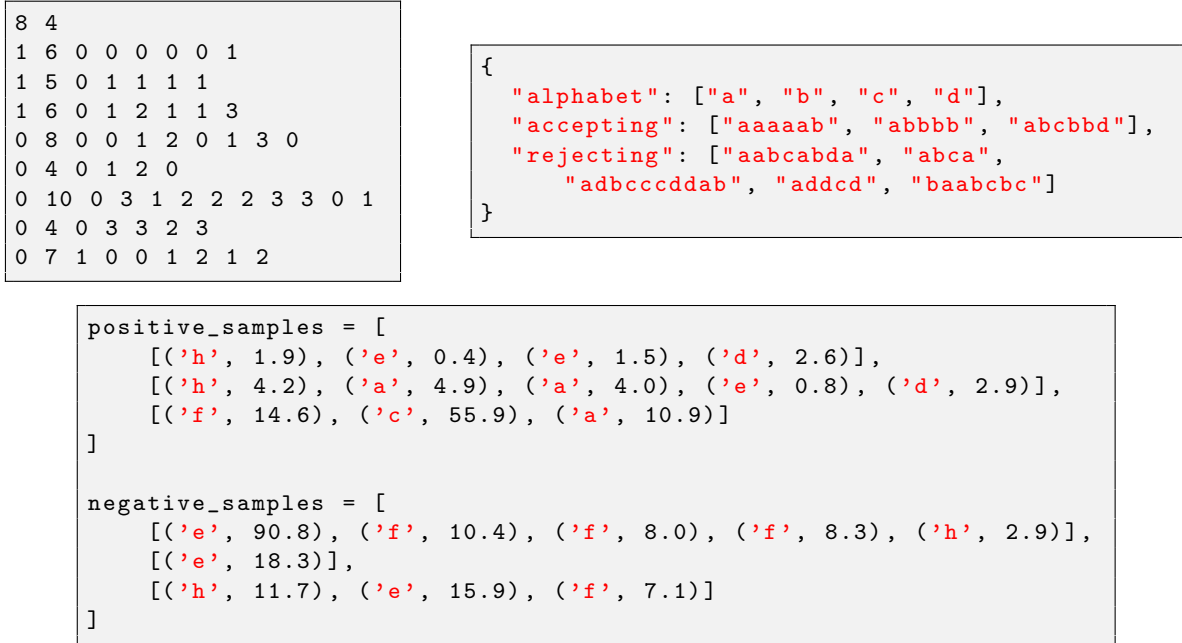


Figure 4: Input file format: Abbadingo (top left), JSON (top right), and Python (bottom)

2.3 Input file formats

Each tool supports input of different types, all provided as plain-text files: ROLL needs automata in the BA [1, 2, 15] or HOA [5] format; the three tools working with sets of samples expect them to be provided as Python variables or in JSON format, with the Abbadingo² format also available for the DFA-based tools. Examples of the input Abbadingo and semantically equivalent JSON formats are shown in the top row in Figure 4, while a Python input with timed samples is shown in the bottom row.

The Abbadingo format follows a strict syntactical structure: the first line contains the number of samples and the size of the alphabet, whose letters are consecutive natural numbers starting from 0. Each of the following lines provides a sample: it starts with a 0/1-Boolean flag indicating whether the sample is accepted, followed by the number of letters in the sample, and then the single letters (that are actually natural numbers) forming the sample word.

The JSON format instead allows to use characters instead of natural numbers, so it contains the alphabet and the vectors of accepting and rejecting samples, given as strings. The JSON data shown in the right hand side of Figure 4 is semantically equivalent to the Abbadingo data shown on the left hand side, once the characters are mapped to numbers as follows: $\{a \mapsto 0, b \mapsto 1, c \mapsto 2, d \mapsto 3\}$.

The Python file instead directly provides two list variables with the positive and negative samples, where each sample is given as a list of letters; in the example shown in the bottom row of Figure 4, the samples also carry time information, so this input is suitable for the DRTA synthesis tool.

2.4 Output file formats

Each tool provides its output in a format suitable for the learned automaton. ROLL produces as output a DBA in the HOA format, unless the `translate` functionality is chosen to translate two input HOA automata into the BA format. The three DFA-based automata learning tools produce output files in the DOT format, suitable to be visualized by Graphviz; DFAMiner can also encode the synthesized DFA in a

²<https://abbadingo.cs.nuim.ie/>

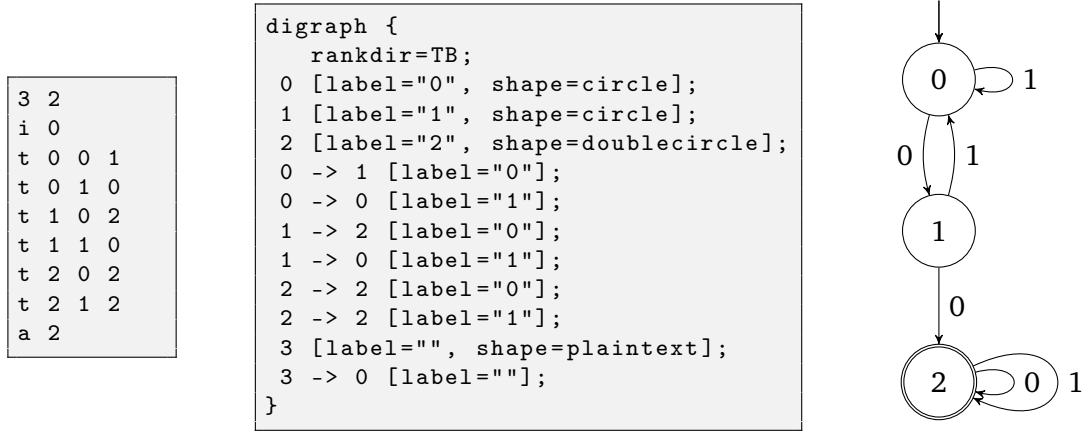


Figure 5: Learned DFA as “textual” (left) and “dot” (center) output format, with its pictorial visualization on the right

textual format á la Abbadingo: the first line contains the number of states and the size of the alphabet while each of the following lines is either representing a state by a letter in $\{i, a, r\}$, denoting that the state is initial, accepting, or rejecting, respectively, followed by the numerical identifier of the state, or representing a transition by the letter t followed by the numerical identifiers of the source state, letter, and destination state. Note that the output is an ordinary DFA, not a 3DFA, so no output state needs to be marked as don’t-care. An example of the three output formats, as generated by the tool on the `dfa` automaton type, is shown in Figure 5, where each format represents the same DFA.

3 Brief Experimental Evaluation

We now briefly present the experimental results for the tools using 3-valued automata in learning decomposed DFAs and DRTAs; these programs and their underlying theory have been proposed and developed during the MISSION project. A more detailed analysis of the experiments can be found in the corresponding papers: [16] for decomposed DFAs and [17] for DRTAs.

The scatter and box plots presented in the remainder of this section adopt a logarithmic scale for their axes, when the values associated with the ticks on the axes are written in exponential notation, e.g., 10^i . In a scatter plot, each mark corresponds to one experiment, and its coordinates are derived from the results of the corresponding tools. A box plot provides the quartile distribution of the results; from bottom to top we find the horizontal lines corresponding to: minimum value, lower quartile (25%), median (50%), upper quartile (75%), and maximum value, with lower, median, and upper quartiles forming the box. Missing extreme values just coincide with the nearest quartile.

3.1 Experiments on decomposed DFAs

The scatter plots in Figure 6 show a comparison between the running time required to synthesize a decomposed DFA when using the ordinary APTA acceptor tree or our version based on 3DFAs, where an acceptor tree is a tree used to store the input samples and their accepted/rejected label in a compact way. Marks on the solid line at 600 seconds represent the fact that the corresponding tool hit the 10 minutes timeout we set in the experimental evaluation; marks on the dashed line at 1000 seconds stand for the tool exhausting the 15GB of memory we assigned to each run. As we can see from the plots, by changing the number of DFAs in the decomposition and the alphabet size, our 3DFA-based

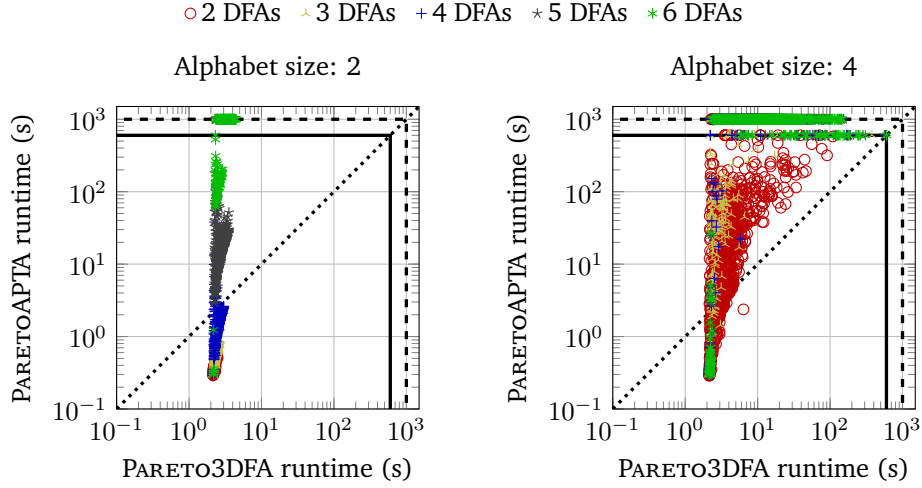


Figure 6: Running time comparison between PARETOAPTA and PARETO3DFA

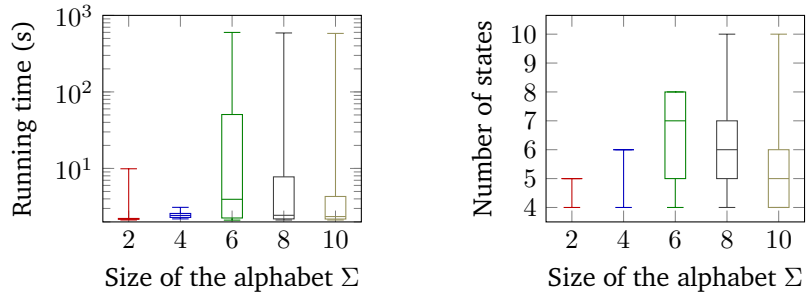


Figure 7: Box plots for the states optimal DIP experiments

version is consistently outperforming the original APTA-based version for all cases requiring more than few seconds. For shorter running times, the positive effect of our advanced techniques in simplifying the encoding is not sufficient to compensate the time spent to apply them.

The box plots in Figure 7 depict the running time and the total number of states required to get a decomposed DFA that is minimal with respect to the state optimal DIP. We evaluated our tool on 460 benchmarks for the alphabet of size 2 and over 1000 benchmarks for each of the other alphabet sizes. As we can see from the plots, the running time of our tool is usually rather short, except for few very challenging cases requiring a rather large number of states. Usually, however, no more than 10 states in total are needed in the decomposition.

3.2 Experiments on DRTAs

The plots in Figure 8 provide two different representations of the distribution of the states generated by our RTA tool developed in [17] and FLEXFRINGE from [20]. It is immediate to recognize that RTA always generates much fewer states than FLEXFRINGE, with cases where FLEXFRINGE produced around 300 times the number of states generated by RTA.

The plots in Figure 9 show the distribution of the number of states in the learned DRTA when varying the number of samples it has to be consistent with. As we can see from the plots, RTA uses at most 5 states even when dealing with 1000 samples, while FLEXFRINGE can even need around 1000 states in the learned DRTA. Even with just 5 samples, FLEXFRINGE requires up to 30 states to deal with them, while RTA just uses 3 states at most. This clearly shows that our novel approach based on 3-valued acceptors and region merging outperforms existing techniques.

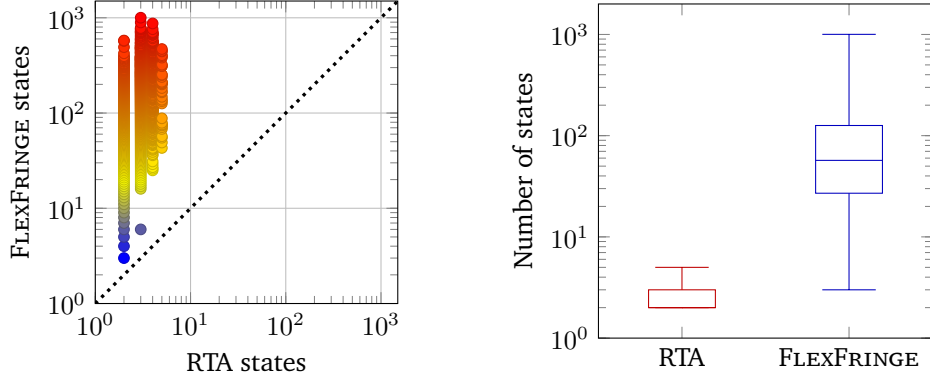


Figure 8: State comparison between our tool RTA and FLEXFRINGE

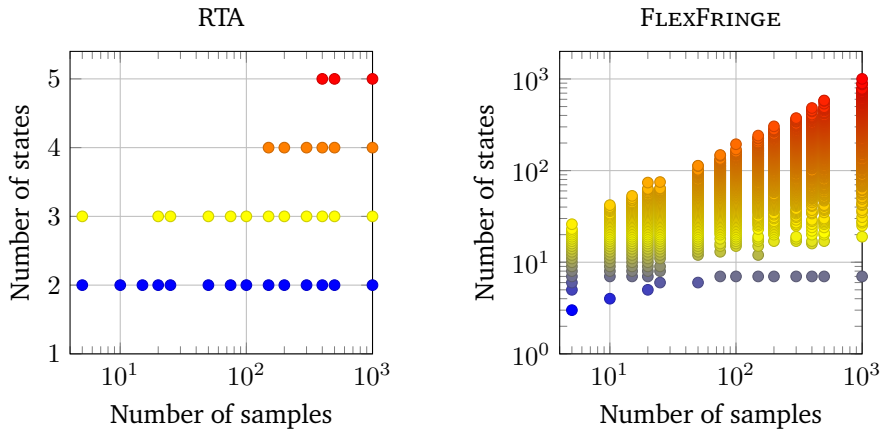


Figure 9: State distribution vs. number of samples by RTA and FLEXFRINGE

4 Conclusion

In this deliverable, we have presented the tool we developed to learn a variety of finite automata: from the well known deterministic finite automata consistent with the given set of samples to their decomposed version and their real time counterpart, as well as deterministic Büchi automata recognizing ω -regular languages. Our automata learning tool provides a unified and consistent access to the implementation of state-of-the-art algorithms, masking the differences in the way the single implementations expect their input. A Docker file is provided to make the automata learning tool portable and to provide details about the specific libraries and external tools that need to be installed in the system in case the user wants to run our automata learning tool natively.

References

- [1] P. A. Abdulla, Y. Chen, L. Clemente, L. Holík, C. Hong, R. Mayr, and T. Vojnar. Simulation subsumption in Ramsey-based Büchi automata universality and inclusion testing. In T. Touili, B. Cook, and P. B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2010.

- [2] P. A. Abdulla, Y. Chen, L. Clemente, L. Holík, C. Hong, R. Mayr, and T. Vojnar. Advanced Ramsey-based Büchi automata inclusion testing. In J. Katoen and B. König, editors, *CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings*, volume 6901 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2011.
- [3] J. An, B. Zhan, N. Zhan, and M. Zhang. Learning nondeterministic real-time automata. *ACM Trans. Embed. Comput. Syst.*, 20(5s):99:1–99:26, 2021.
- [4] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
- [5] T. Babiak, F. Blahoudek, A. Duret-Lutz, J. Klein, J. Kretínský, D. Müller, D. Parker, and J. Strejcek. The Hanoi omega-automata format. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 479–486. Springer, 2015.
- [6] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.
- [7] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors. *Handbook of Model Checking*. Springer, 2018.
- [8] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognit.*, 38(9):1332–1348, 2005.
- [9] D. Dell’Erba, Y. Li, and S. Schewe. DFAMiner: Mining minimal separating DFAs from labelled samples. In A. Platzer, K. Y. Rozier, M. Pradella, and M. Rossi, editors, *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II*, volume 14934 of *Lecture Notes in Computer Science*, pages 48–66. Springer, 2024.
- [10] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*, volume 2500 of *Lecture Notes in Computer Science*. Springer, 2002.
- [11] O. Grinchtein, M. Leucker, and N. Piterman. Inferring network invariants automatically. In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 483–497. Springer, 2006.
- [12] N. Lauffer, B. Yalcinkaya, M. Vazquez-Chanlatte, A. Shah, and S. A. Seshia. Learning deterministic finite automata decompositions from examples and demonstrations. In A. Griggio and N. Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 1–6. IEEE, 2022.
- [13] Y. Li, Y. Chen, L. Zhang, and D. Liu. A novel learning algorithm for Büchi automata based on family of DFAs and classification trees. *Inf. Comput.*, 281:104678, 2021.
- [14] Y. Li, X. Sun, A. Turrini, Y. Chen, and J. Xu. ROLL 1.0: ω -regular language learning library. In T. Vojnar and L. Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, volume 11427 of *Lecture Notes in Computer Science*, pages 365–371. Springer, 2019.

- [15] R. Mayr and L. Clemente. Advanced automata minimization. In R. Giacobazzi and R. Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 63–74. ACM, 2013.
- [16] J. Meng, J. An, Y. Li, A. Turrini, F. Xu, N. Zhan, and M. Zhang. Efficient decomposition identification of deterministic finite automata from examples. In A. Goharshady and C. Haase, editors, *Dependable Software Engineering. Theories, Tools, and Applications - 11th International Symposium, SETTA 2025, Oxford, UK, December 1-3, 2025, Proceedings*, Lecture Notes in Computer Science. Springer, 2025. To appear.
- [17] J. Meng, J. An, Y. Li, A. Turrini, and M. Zhang. In Y.-F. Chen, T. Jensen, , and O. Lengál, editors, *Proceedings of the 27th International Conference on Verification, Model Checking, and Abstract Interpretation – VMCAI 2026, Rennes, France, January 11-12, 2026, Proceedings*, Lecture Notes in Computer Science. Springer, 2026. To appear.
- [18] D. Neider. Computing minimal separating DFAs and regular invariants using SAT and SMT solvers. In S. Chakraborty and M. Mukund, editors, *Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012, Thiruvananthapuram, India, October 3-6, 2012. Proceedings*, volume 7561 of *Lecture Notes in Computer Science*, pages 354–369. Springer, 2012.
- [19] M. Tappier, B. K. Aichernig, and F. Lorber. Timed automata learning via SMT solving. In J. V. Deshmukh, K. Havelund, and I. Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 489–507. Springer, 2022.
- [20] S. Verwer, M. de Weerd, and C. Witteveen. Efficiently identifying deterministic real-time automata from labeled data. *Mach. Learn.*, 86(3):295–333, 2012.