Learning the State Machine Behind a Modal Text Editor: The (Neo)Vim Case Study

Pierre Ganty[0000−0002−3625−6003]

IMDEA Software Institute, Madrid, Spain
pierre.ganty@imdea.org

Abstract. We use active automata-based learning to extract the state machine underlying the modal text editor Vim. We expose the various challenges to interface an active learning library with the text editor. Furthermore, we report on how we uncovered several issues and how they were dealt with by the (Neo)Vim developers. Finally, we reflect on the possible uses of automatically extracted finite-state machines beyond bug reports.

Keywords: Active Learning, Moore machine, Modal text editor

1 Introduction

Modal text editors such as Vim support multiple editing modes. Depending on the mode, typed characters are interpreted either as sequences of commands or are inserted as text. Fig. 1 depicts part of the modes (`normal`, `visual`, `insert`, . . .) and how the keystrokes (esc, ctrl+v, i, . . .) transition between them.

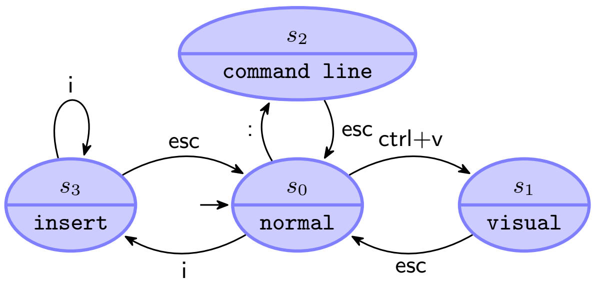


Fig. 1. Four Vim modes and examples of keystrokes to move between them. In `insert` mode, the text you type is inserted into the buffer (i.e. the in-memory text of a file). `visual` mode enables the selection of a piece of text via keystrokes. In `command line` mode, you can enter one line of text, typically commands, at the bottom of the window. In `normal` mode, the typed keys are interpreted as editor commands to be applied to the text in the buffer.

We report on our experience of using a library for automata-based active learning where the system under learning is Vim. Our main hypothesis is that modal text editors implement a finite-state machine (such as Fig. 1) that can be extracted automatically using active learning automata-based techniques.

Our motivation is twofold: we want to expose (1) active learning automata-based techniques to real-world systems like the Vim text editor and; (2) software developers to automata-like artifacts produced by active learning techniques.

Incidentally, we want to understand how active learning techniques can contribute to software development.

In our endeavor we faced several challenges. A first technical challenge asks which active learning library to interface with Vim and how. There exists several well-maintained publicly available libraries such as learnLib and LibAlf. (See the Automata Wiki [13] for an exhaustive list.) A second challenge is the scalability of the active learning techniques. What to do if the finite-state machine to be computed is too large or even infinite? We took for granted that the state machine underlying the Vim editor had finitely many states, but we quickly realized that in general the finiteness assumption is not true.

In this paper, we solve the interface between an active automata learning library (AALpy) and the Vim text editor (actually, we used the Vim fork called Neovim). We explain how to control the scalability of the active learning process, both via AALpy and the Neovim editor itself. We submitted bugs to the Vim and Neovim developers and report on our interactions, including whether the bugs were acknowledged or not and whether they were fixed.

Based on our experience, we will discuss the lessons learned and possible future work going beyond modal text editors.

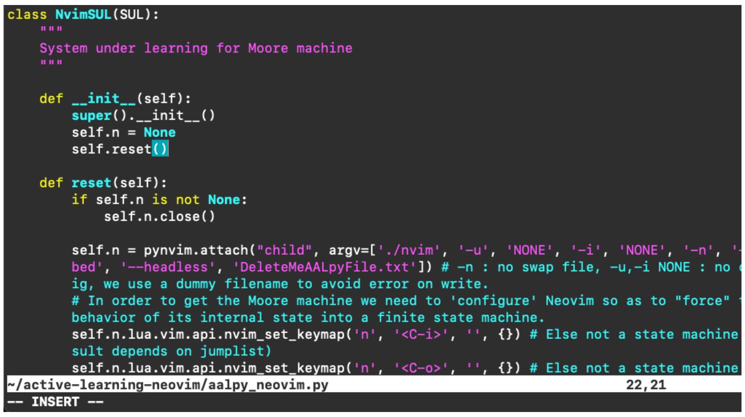
2 Preliminaries

(Neo)Vim. We used the Vim[1] text editor for this case study. First and foremost, we used Vim because there are several reports of a state machine underlying its modal capabilities (see, for instance, Vim's Wikipedia page[2] or Darcy Parker's Vim Modes Transition Diagram [19]). Second, Vim is a rather large piece of software, mostly written in C. It implements a highly configurable text editor with a rich set of features. At least a dozen books[3] about Vim are available. Vim has a large user base (see for instance this survey,[4] where Vim came fifth as the most popular "Integrated development environment"). Vim also has a long history of over 40 years of development [20]. For the above reasons, we claim that it is a battle-tested software. It is also worth pointing that its documentation[5] is well-maintained.

[1] https://www.vim.org/
[2] https://en.wikipedia.org/wiki/Vim_(text_editor)#Modes
[3] https://iccf-holland.org/vim_books.html
[4] https://survey.stackoverflow.co/2022/#section-most-popular-technologies-integrated-development-environment
[5] Available via the `:help` command of Vim or via dedicated websites on the Internet.

We use a fork of Vim called Neovim[6] because of its remote Python API (pynvim[7]). Neovim shares a lot of its source code with Vim, in particular the core functionality of the editor. Consequently, it is often the case that bugs reported to Neovim are deferred to the Vim developers, who publish patches fixing the issue which are then ported to the Neovim code base by the Neovim developers. The two projects have developers in common.

AALpy. AALpy [17] is a light-weight active automata learning library written in Python. AALpy supports a wide range of modeling formalisms, including Moore machine, which we use as the formal model for the modes transition diagrams of the Vim editor. Indeed, Moore machines are finite-state machines like that of Fig. 1 where transitions between states are labeled by the so-called "input" symbols while states carry "output" symbols. In relationship to Vim, we have that the input symbols corresponds to the typed characters (that is, the keystrokes) while the output symbols corresponds to the modes reported by Vim (e.g. `INSERT` mode in Fig. 2).



```
class NvimSUL(SUL):
    """
    System under learning for Moore machine
    """

    def __init__(self):
        super().__init__()
        self.n = None
        self.reset()

    def reset(self):
        if self.n is not None:
            self.n.close()

        self.n = pynvim.attach("child", argv=['./nvim', '-u', 'NONE', '-i', 'NONE', '-n', '-
        bed', '--headless', 'DeleteMeAALpyFile.txt']) # -n : no swap file, -u,-i NONE : no c
        ig, we use a dummy filename to avoid error on write.
        # In order to get the Moore machine we need to 'configure' Neovim so as to "force" t
        behavior of its internal state into a finite state machine.
        self.n.lua.vim.api.nvim_set_keymap('n', '<C-i>', '', {}) # Else not a state machine
        sult depends on jumplist)
        self.n.lua.vim.api.nvim_set_keymap('n', '<C-o>', '', {}) # Else not a state machine
~/active-learning-neovim/aalpy_neovim.py                                          22,21
-- INSERT --
```

Fig. 2. Screenshot of Neovim running inside a terminal.

A (deterministic) *Moore machine* is a 6-tuple $(S, s_0, \Sigma, O, \delta, G)$, where: S is a finite set of *states* including an *initial state* s_0; Σ is a nonempty finite set called the *input alphabet*, O is a nonempty finite set called the *output alphabet*; $\delta \colon S \times \Sigma \to S$ is a *transition function* mapping a state and the input alphabet to the next state; and $G \colon S \to O$ is an *output function* mapping each state to the output alphabet.

Fig. 1 gives an example of Moore machine with states $\{s_0, s_1, s_2, s_3\}$; input alphabet given by $\boxed{:}$, $\boxed{\text{esc}}$, $\boxed{\text{ctrl}}+\boxed{\text{v}}$, $\boxed{\text{i}}$; and output alphabet comprising `command line`, `normal`, `visual` and `insert`.

[6] https://neovim.io/

[7] https://github.com/neovim/pynvim

In AALpy, active learning of Moore machines is a fully automated process following a paradigm called *minimally adequate teacher* (MAT) initially put forward by Angluin [14]. In our setting, the MAT interacts in rounds with a learner whose task is to compute a Moore machine. The learner asks the teacher Neovim related queries that fall into two categories: (1) *membership queries* asking the teacher to return the output of a sequence of keystrokes applied to a newly spawned Neovim process; and (2) *equivalence queries* asking the teacher whether a Moore machine coincides with the state machine underlying Neovim. In AALpy, equivalence queries can be approximated via *conformance testing strategies*, which performs multiple membership-like queries comparing the output of the Moore machine with that of Neovim when applied the same sequence of keystrokes. For more details, we refer the interested reader to the survey of Vaandrager [24] who also provides an exhaustive list of references on the subject as well as the AALpy website[8] for implementation specific details. In recent years, active learning automata-based techniques have been used successfully on real-world system ranging from virtual private network servers to recurrent neural networks and Bluetooth Low Energy protocols [23,18,21,22].

3 Active Learning of Neovim Moore Machine

Interfacing Neovim and AALpy. AALpy interacts with Neovim via its remote API pynvim[9]. One design goal of pynvim is to provide a library for connecting to and scripting Neovim processes, which is the feature we use in this paper. AALpy requires implementing a `step` function as well as a `pre` and a `post` function. The `pre` and `post` functions have to do with initialization/startup of a Neovim process and graceful shutdown of the Neovim process/memory cleanup. For the shutdown, we close the Neovim sub-process and for the initialization we spawn a new "child" process as shown at the top of Fig. 2. The `step` function submits to the Neovim process the next keystroke and returns the updated mode resulting from the keystroke by invoking `nvim_get_mode()`[10]. The set of keystrokes to be used by AALpy is configurable and, in our latest version of the interface [3], we set it to: $\{\boxed{\text{l}}, \boxed{\text{ctrl}}+\boxed{\text{g}}, \boxed{\text{0}}, \boxed{\text{ctrl}}+\boxed{\text{v}}, \boxed{\text{c}}, \boxed{\text{:}}, \boxed{\text{v}}, \boxed{\text{g}}, \boxed{\text{ctrl}}+\boxed{\text{o}}, \boxed{\text{r}}, \boxed{\text{esc}}, \boxed{\hookleftarrow}, \boxed{\text{ctrl}}+\boxed{\text{c}}, \boxed{\text{ctrl}}+\boxed{\backslash}+\boxed{\text{ctrl}}+\boxed{\text{n}}\}$.

Scaling up. Generally the larger the set of keystrokes the more time AALpy needs before returning a Moore machine, and therefore we have to select the keystrokes carefully. Indeed, when AALpy adds a state to the Moore machine, it also adds transitions (one per keystroke) leaving that state. Hence, when AALpy returns a Moore machine with 100 states for a set of, say, 15 keystrokes, we know immediately the machine has 1,500 transitions. We settled on the previous set of keystrokes because it allows visiting many modes: $\boxed{\text{:}}$ enters the command line

[8] https://github.com/DES-Lab/AALpy
[9] https://github.com/neovim/pynvim
[10] https://neovim.io/doc/user/api.html#nvim_get_mode()

mode, `l` followed by `c` the insert mode while also covering operator pending and motions, `ctrl`+`v` the visual mode,...

Apart from the keystrokes, an independent way to control scalability of the learning process is to customize the Neovim process. Roughly speaking, different settings of Neovim yield different Moore machines. In general, the machine is not even finite, since the result of some keystrokes depends on the previously inserted text inside Neovim. For instance, an editor command in normal mode, like `c` `f` `z`, deletes the text between the cursor position and the next occurrence of the symbol `z` on the same line of text, after which it completes by entering the `insert` mode. (`c` is for change, `f` is for forward and `z` is the symbol looked for). However, if `z` does not occur between the cursor and the end of the line, then no text is deleted and the editor command completes by staying in normal mode.[11] Because the size of the text being edited is virtually unbounded, the learner is therefore facing the task to compute a Moore machine with infinitely many states. But there is more, assuming we can make the machine finite it can still be very large: the result of the keystrokes `ctrl`+`i` and `ctrl`+`o` depends on the so-called 'jumplist' which counts up to 100 entries.

To remedy this problem, we configure Neovim to prevent the behaviors described above to happen. More precisely, we use options controlling Neovim's behavior as well as key mapping which we set after spawning the Neovim subprocess. As shown in Fig. 2, we map the keystrokes `ctrl`+`i` and `ctrl`+`o` to do "nothing" when we are in normal mode to avoid the pitfall of the jumplist mentioned above.

Finally, let us mention that we commented out in the Neovim source code all function calls causing time delays as suggested by developers [6]. These delays are present in the source code to give the user the time needed to read some messages. However, those delays slow down the active learning process.

Current State of the Learning. Using the above set of 14 keystrokes and a customized Neovim as described above AALpy learns a Moore machine of 99 states and 1,386(=14 $\times$ 99) transitions after 58 rounds between the learner and the teacher in less than 10 minutes by performing 31,479 memberships queries 29,700 of which (or 94%) are carried out in the context of the conformance testing to approximate equivalence queries.

4 Outcomes and Impact

After successfully learning a Moore machine, we set AALpy to produce a file in the dot language of the Graphviz project [16]. The dot language allows for visual inspection of the Moore machine as well as running queries on it using some external tool supporting the dot language format. Our typical workflow has been to first manually inspect the computed Moore machine using a dot file visualization tool (xdot.py[12]) and then formulate queries (e.g. How to reach

[11] See [12] for a more detailed explanation.

[12] https://github.com/jrfonseca/xdot.py

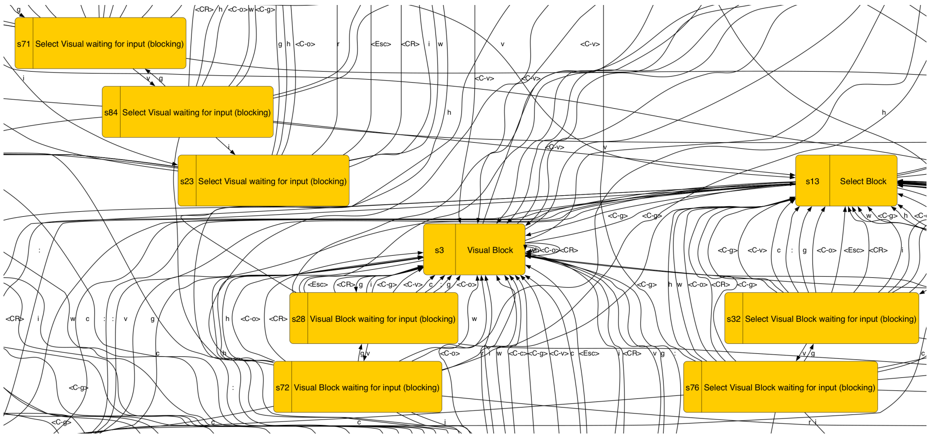

s71 Select Visual waiting for input (blocking)
s84 Select Visual waiting for input (blocking)
s23 Select Visual waiting for input (blocking)
s13 Select Block
s3 Visual Block
s28 Visual Block waiting for input (blocking)
s32 Select Visual Block waiting for input (blocking)
s72 Visual Block waiting for input (blocking)
s76 Select Visual Block waiting for input (blocking)

Fig. 3. Detail of a Moore machine learned using our approach.

that state? What differentiates these two states?...) to be answered either directly within AALpy (e.g. `compare_automata`, `find_distinguishing_seq`, ...) or using an external graph visualization and exploration tool like Gephi[13] [15]. Fig. 3 depicts part of such learned Moore machine where states are labelled with modes and transitions with keystrokes.

After inspection and querying we either modify the set of keystrokes and/or Neovim's customization and repeat the learning process; or we find a behavior in the Moore machine requiring further investigation. Such behavior typically consists of one or two short sequences of keystrokes whose resulting modes cannot be explained clearly following the Neovim's documentation. If the unexplained behavior is reproducible in Vim by manually typing the sequences of keystrokes, then we submit an issue to the Vim developers. For instance, according to the documentation, $\boxed{\text{ctrl}}+\boxed{\text{v}}$ and $\boxed{\text{ctrl}}+\boxed{\text{q}}$ behave the same in `insert` and `replace` mode. When learning a Moore machine with a set of keystrokes including $\boxed{\text{ctrl}}+\boxed{\text{v}}$ and $\boxed{\text{ctrl}}+\boxed{\text{q}}$, it turned out that $\boxed{\text{ctrl}}+\boxed{\text{v}}$ labeled transitions and $\boxed{\text{ctrl}}+\boxed{\text{q}}$ labeled transitions leaving the same state ended up in different states. After reproducing and reporting the issue in Vim, the developers fixed it promptly [2].

In general, the reactions to the issue largely vary: convincingly arguing the behavior is not an issue [7,12]; acknowledging a problem in the documentation [8]; acknowledging a problem in Vim's behavior but not fixing it ([9] at first); acknowledging a problem in Vim's behavior and fixing it [4,1,11,2,9,10]. On one occasion, we fixed the issue ourselves after it was acknowledged [9]. Fixes are typically one-liners and no more than 10 lines of code [11]. A comprehensive list of issues acknowledged and fixed is given on the AALpy discussion thread page New use of AALpy library[14] on their GitHub repository.

[13] https://github.com/gephi/gephi
[14] https://github.com/DES-Lab/AALpy/discussions/13

5 Discussion

Active Learning and the Software Development Cycle. Both the Vim and Neovim developers have been responsive and helpful when issues were submitted. As we stated in the introduction, the source code of Vim is battle-tested and no issue we reported involved making the text editor unresponsive, mishandle data or let alone crash. Apart from acknowledged issues, the reaction to our reporting ranged from confused[15] to nobody cares[16] and won't fix[17]. We captured the interest of some Neovim developers after showing them the Moore machines. Together with them, we envisioned two potential uses: as a source of **documentation** aimed at end users or as a **formal specification** of behaviors to be tested between (major) releases. More precisely, a conformance testing tool could become part of the software life cycle, where the Moore machine of a release is used as a specification to test subsequent releases. Recall that a conformance testing tool runs a large number of tests arising from a given Moore machine by following a strategy to order and select the tests. The above can be implemented in a few lines of Python using AALpy.

Besides testing, we also claim that a Moore machine model provides valuable information in case of **refactoring**. Case in point is the refactoring Neovim carried out starting from the Vim code base and whose logic[18] [5] leverages an "input-driven state machine". We claim that having computed some information about such input-driven state machines helps with the code refactoring.

Enhancing the Model. As we mentioned above the larger the set of keystrokes, the more time AALpy needs before returning a Moore machine. This is because AALpy has to try more sequences of keystrokes the number of which grows exponentially with the length of the sequence: with 3 keystrokes there are 3^n sequences of length n while with 10 keystrokes there are 10^n sequences.

In (Neo)vim, there are some keystrokes that are supposed to have the same effects on the mode. For instance, $\boxed{\text{i}}$ enters `insert mode` from `normal mode` and so does $\boxed{\text{I}}$, $\boxed{\text{o}}$ and $\boxed{\text{O}}$ (they differ in other aspects but not from the modal point of view). So if the set of keystrokes already contains $\boxed{\text{i}}$ adding $\boxed{\text{o}}$ should have a predictable effect on the returned Moore machine: for each edge labeled with $\boxed{\text{i}}$ from node s such that s is `normal mode` to some node t, there is another edge from s to t with label $\boxed{\text{o}}$. Therefore, leveraging (Neo)vim's documentation, one can enrich the Moore machine after the active learning process by adding edges as suggested above. The enriched Moore machine can then be used by a conformance testing tool to test whether the enriched machine still conforms with the implementation. Enriching the Moore machine is as simple as editing its dot file, since the dot format is human-readable.

[15] https://github.com/vim/vim/issues/13649#issuecomment-1847848471
[16] https://github.com/vim/vim/pull/13001#issuecomment-1703925253
[17] https://github.com/vim/vim/issues/12115#issuecomment-1483815493
[18] https://github.com/neovim/neovim/tree/master/src/nvim#nvim-lifecycle

Acknowledgments. We thank Xiao Peng Ye, Madeleine Mathieu and for their tremendous help with the interface and uncovering bugs. We also thank the Vim and Neovim developers for their time including Bram Moolenaar, the creator of Vim, who died during the summer 2023. Furthermore, we thank Edi Muskardin for his tremendous help with AALpy. Finally, we thank the SPIN reviewers for their effort and suggestions.

This publication is part of the grant PID2022-138072OB-I00, funded by MCIN, FEDER, UE. This work has been partially supported by the PRODIGY Project (TED2021-132464B-I00) funded by MCIN and the European Union NextGeneration.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Consecutive call of nvim_get_mode() does not return the same mode · Issue #15288 · neovim/neovim — github.com. https://github.com/neovim/neovim/issues/15288, [Accessed 05-01-2024]
2. Different behavior between C-v and C-q · Issue #12684 · vim/vim — github.com. https://github.com/vim/vim/issues/12684, [Accessed 05-01-2024]
3. GitHub - pierreganty/active-learning-neovim: Active automata-based learning of the Moore machine underlying Neovim — github.com. https://github.com/pierreganty/active-learning-neovim, [Accessed 10-01-2024]
4. Non deterministic behavior when querying current mode · Issue #8323 · vim/vim — github.com. https://github.com/vim/vim/issues/8323, [Accessed 05-01-2024]
5. Nvim lifecycle · neovim/neovim — github.com. https://github.com/neovim/neovim/tree/master/src/nvim#nvim-lifecycle, [Accessed 05-01-2024]
6. nvim_get_mode waits 3∼4 secs with 'showmode' enabled or when there are error messages · Issue #19352 · neovim/neovim — github.com. https://github.com/neovim/neovim/issues/19352, [Accessed 05-01-2024]
7. Possible inconsistent behavior · Issue #8346 · vim/vim — github.com. https://github.com/vim/vim/issues/8346, [Accessed 05-01-2024]
8. Possibly inconsistent behavior · Issue #13649 · vim/vim — github.com. https://github.com/vim/vim/issues/13649, [Accessed 05-01-2024]
9. Possibly incorrect transitions between modes · Issue #12115 · vim/vim — github.com. https://github.com/vim/vim/issues/12115, [Accessed 05-01-2024]
10. Possibly undocumented behavior with replace after visual · Issue #13091 · vim/vim — github.com. https://github.com/vim/vim/issues/13091, [Accessed 05-01-2024]
11. Undocumented (possibly incorrect) behavior for Virtual Replace mode · Issue #12045 · vim/vim — github.com. https://github.com/vim/vim/issues/12045, [Accessed 05-01-2024]
12. Unexpected behavior? · Issue #10693 · vim/vim — github.com. https://github.com/vim/vim/issues/10693, [Accessed 05-01-2024]
13. Automata Wiki — automata.cs.ru.nl. https://automata.cs.ru.nl/ (2017), [Accessed 05-01-2024]
14. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (Nov 1987). https://doi.org/10.1016/0890-5401(87)90052-6

15. Bastian, M., Heymann, S., Jacomy, M.: Gephi: An open source software for exploring and manipulating networks. Proceedings of the International AAAI Conference on Web and Social Media **3**(1), 361–362 (2009). https://doi.org/10.1609/icwsm.v3i1.13937
16. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Softw. Pract. Exper. **30**(11), 1203–1233 (2000)
17. Muškardin, E., Aichernig, B.K., Pill, I., Pferscher, A., Tappler, M.: AALpy: An active automata learning library. Innovations in Systems and Software Engineering **18**(3), 417–426 (Sep 2022). https://doi.org/10.1007/s11334-022-00449-3
18. Muškardin, E., Aichernig, B.K., Pill, I., Tappler, M.: Learning finite state models from recurrent neural networks. In: Integrated Formal Methods. p. 229–248. LNCS, Springer (2022). https://doi.org/10.1007/978-3-031-07727-2_13
19. Parker, D.: Vim Modes Transition Diagram in SVG. https://gist.github.com/darcyparker/1886716 (2012), [Accessed 05-01-2024]
20. Pezzi, G.: Understanding the Origins and the Evolution of Vi & Vim. https://pikuma.com/blog/origins-of-vim-text-editor (2023), [Accessed 05-01-2024]
21. Pferscher, A., Aichernig, B.K.: Fingerprinting bluetooth low energy devices via active automata learning. In: FM 2021: Formal Methods. p. 524–542. LNCS, Springer (2021). https://doi.org/10.1007/978-3-030-90870-6_28
22. Pferscher, A., Wunderling, B., Aichernig, B.K., Muškardin, E.: Mining digital twins of a VPN server. In: FMDT 2023: Proceedings of the Workshop on Applications of Formal Methods and Digital Twins. CEUR Workshop Proceedings, vol. 3507 (2023)
23. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing iot communication via active automata learning. In: ICST 2017: IEEE International Conference on Software Testing, Verification and Validation. IEEE (2017). https://doi.org/10.1109/icst.2017.32
24. Vaandrager, F.: Model Learning. Commun. ACM **60**(2), 86–95 (Jan 2017). https://doi.org/10.1145/2967606