RERS 2020:
Learning, Testing, Fuzzing and Slicing

Joshua Moerman
RWTH Aachen University
joshua@cs.rwth-aachen.de

Jana Berger
RWTH Aachen University
berger@cs.rwth-aachen.de

October 16, 2020

1 Overview

The RERS challenge consists of analysis problems for reactive software. We participate in the following two tracks.

Sequential LTL problems These consist of source code (in both C and Java) and we have to evaluate whether certain LTL properties hold for the program.

Sequential reachability problems These consist of source code (again both C and Java) and we have to decide which error codes are reachable.

Our main technique is that of *active automata learning*. For the LTL problems, we hope that the learning finds the right model, in which case we can *model-check* the LTL properties, since this logic is decidable for finite state systems. For the reachability problems, we can only find reachable errors through learning. For non-reachability, we have to rely on the fact that the learned models are complete. We have observed that our models were not complete, and so we used another technique: *program slicing*. This was able to prove some non-reachability results. For both tracks we did additional *testing* and *fuzzing*.

1.1 Tools

Here we give a list of used tools with a short description.

LearnLib [IHS15] (All problems)
This is a Java library for automata learning and testing. We have used this to learn all the provided problems.

NuSMV [CCGR99] (LTL problems)
This is an explicit state model checker, which we use after the models have been learned.

hybrid-ads [Moe19] (LTL problems)
We did additional testing with our own tool, which is based on distinguishing sequences.

afl-fuzz [Zal] (Reachability problems)
We used fuzzing to do additional testing of the reachability problems. This is closely related to random testing, but it also instruments the code, in order to discover different execution paths.

FramaC [CKK+12] (Reachability problems)
This is a library for many types of analysis for C code. We used slicing to prove that certain errors are unreachable.

All commands, used parameters, and results can be found in our repository: git.rwth-aachen.de/joshua/rers2020

1.2 Timeline

Since learning, testing, and fuzzing may run indefinitely, we have set fixed time outs. We did the following. The actual running times can be found in the subsequent sections.

- 24 hours of learning, with max 2 000 000 tests (per hypothesis).
- 90 hours of learning for known incomplete problems (12, 15, 16).
- 112 hours of additional fuzzing.
- 112 hours of additional testing.

After that, we did the model checking for the LTL problems. And we extracted the reachable errors for the reachability problems. Additionally we did slicing for the reachability problems. All these later steps did not take a considerable amount of time.

Remark 1. For slicing, we had time issues for the biggest problem (Problem 19). It needed roughly a full day *per error code* to slice the problem. For this reason, we did not slice Problem 19.

Problem	*States*	*Time*	*EQs*	*Remark*
1	35	5s	15	
2	55	8s	30	
3	107	5m 44s	70	
4	88	31s	59	
5	151	23s	72	
6	96	1m 4s	51	
7	107	10m 29s	60	
8	42	1m 59s	23	
9	77	20m 19s	35	
11	20	1s	7	
12	9 724	90h 0m 0s	3770	Timeout
13	77	40s	41	
14	162	27s	77	
15	11 238	90h 0m 0s	6152	Timeout
16	13 895	90h 0m 0s	7914	Timeout
17	714	3m 0s	372	
18	658	2m 53s	339	
19	884	32m 23s	485	

Table 1: Runtime to the final hypothesis and number of equivalence queries. Note this time does not include testing *after* the last hypothesis. Some of the reachability problems timed out.

2 Learning

In this section we report more details on the learning. We have used the TTT algorithm [IHS14] for learning and the randomised Wp method [FvBK+91, Moe19] for testing the hypotheses. Otherwise, the LearnLib code was straightforward. The number of tests was bound by 2 million per hypothesis.

The running times for learning can be found in Table 1. Note that some problems ran until the timeout, meaning that these models are known to be incomplete. (For the other models, we do not know whether they are complete.) Unfortunately, we did not count the number of membership queries.

Remark 2. We observe that the problems 17 and 18 are quite big, but are learned very fast. This is in contrast to, for example, problem 9. We conjecture that the running time is mostly determined by the time to find counterexamples (based on luck) and that the learning algorithm is efficient.

3 Testing and Fuzzing

After the learning process we did additional testing. For the LTL problems, we did this with our FSM-based conformance checking *hybrid-ads* [Moe19].

Problem	*Common findings*	*Unique to learning*	*Unique to fuzzing*
11	18		
12	16		
13	23		8
14	15		
15	41		
16	14		1
17	30		
18	29	1	
19	13	1	

Table 2: Number of reachable errors found by the different techniques.

This tool randomly generates tests according to the HSI-method (using adaptive distinguishing sequences when possible). We tested for 112 hours, slowly increasing the length of the length of the tests. *No counterexamples were found.* To give an idea of the number of tests, for problem 1: We performed roughly 67 million tests, of an average length of 1030 actions. Meaning that we roughly performed 69 billion tests, which, admittedly, is a bit overkill.

For the reachability problems, we used fuzzing instead. The reason for this is that we expected counterexamples to be found (because problems 12, 15, and 16 timed out). We wanted to only look for reachable errors, not just any counterexample, and fuzzing finds traces for which the program crashes. We did provide the fuzzer an initial state cover, so that it is already able to find the known errors quickly. The fuzzer found additional error codes, which we summarise in Table 2. Note that for the known incomplete models it only found 1 additional error. But it also found many more errors for problem 13.

Remark 3. The fuzzing did find additional errors to be reachable. However, it also didn't find all the errors found through learning. This is surprising, as we provided the fuzzer a state cover of the learned models. The fuzzer outputted that there were too many initial traces, and that it removed redundant ones. This means that, although our state cover reaches different states, these are seen as equivalent by the fuzzer.

4 LTL Model Checking

Since the learning algorithm constructs finite state machines, we can directly apply LTL model checking. We used the NuSMV tool, for no particular reason. The above mentioned repository contains code for converting LearnLib models into NuSMV models, by using the alternating I/O semantics.

It is interesting to see how the valuation of the LTL properties change over time. We have computed all the properties on the intermediate hypotheses

Figure 1: Evaluating all the LTL properties on the intermediate hypotheses. The problems are in order (first left-to-right, then top-to-bottom).

Problem	*Reachable*	*Unreachable*	*Unknown*	*Avg. Time per Slice*
11	18	64	18	2.15s
12	16	49	35	7.82s
13	31	22	47	9.93s
14	15	43	42	9.02s
15	41	16	43	248s
16	15	2	83	1488s
17	30	26	44	54.6s
18	30	18	52	1954s
19	14	0	86	>86400s

Table 3: Number of reachable errors (see previous section) and unreachable errors as proven by program slicing. We have added a column which indicates for how many error codes we do not have an answer.

as well. We observe that the valuation doesn't change very often, and that the "final" result is obtained before the final hypothesis. The first hypothesis is always a single-state machine. See Figure 1 for these timelines.

We note that some properties don't change at all, they probably are tautologies, or have little to do with the actual program. Some properties only change once, these could be safety properties, as they can be violated with a single finite word. We do not know whether there is any connection between the ease of learning and the complexity of the property.

5 Slicing

We select a specific error code to be sliced and modify the program to only contain the error locations of this error code. FramaC then removes all the code which it deems irrelevant for that particular error. We achieve this by slicing for calls to `__VERIFIER_error()`. In some cases, no code was left (i.e. just an empty `main()` function), meaning that those error codes are unreachable. In Table 3 we report the number of unreachable error codes proven this way.

Unfortunately, this approach is really separate from the learning, and we see no way of using the slicing information for the learning, or vice versa.

References

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499. Springer, 1999.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c - A software analysis

perspective. In *SEFM*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012.

[FvBK+91] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Software Eng.*, 17(6):591–603, 1991.

[IHS14] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In *RV*, volume 8734 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2014.

[IHS15] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source LearnLib - A framework for active automata learning. In *CAV (1)*, volume 9206 of *Lecture Notes in Computer Science*, pages 487–495. Springer, 2015.

[Moe19] Joshua Moerman. *Nominal Techniques and Black Box Testing for Automata Learning*. PhD thesis, Radboud University, Nijmegen, The Netherlands, 2019.

[Zal] Michal Zalewski. American fuzzy lop. `https://lcamtuf.coredump.cx/afl/`. Online; accessed 26-August-2020.