# Concolic and learning assignment

Luigi Coniglio (Group 4)

April 25, 2018

## 1 Introduction

For this assignment we were asked to apply KLEE [1] and LearnLib [2] to the RERS reachability problems (http://rers-challenge.org/) and report our reults. In section 2 we present the tools and introduce the RERS reachability problems, in section 3 the method and goals of the experiment are described, finally in section 4 we illustrate and comment the results. In addition section 5, which was intended as a bonus for this assignment, shows how to combine the two approaches (concolic execution and model learning) and the results obtained from this combination.

## 2 Challenges and Tools

### 2.1 RERS challenges

The Rigorous Examination of Reactive Systems (RERS) challenges are a series of program analysis challenges which objective consist in estimating the effectiveness of different software verification techniques.[1] The RERS challenges include problems from different categories but for this assignment we will only focus on the reachability problems.

The RERS reachability problems consist of a set of heavily obfuscated programs which accept a series of numbers as input and behave accordingly to the input provided. Each program contains a certain amount of error statements which can be reached only providing the correct input sequence.

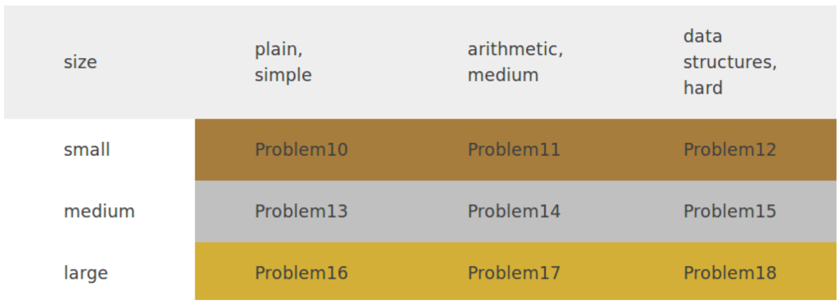
Listing 1 shows an example: in this case we can see that the input sequence ”6 5 2 4” produces the error *error_86* for Program12. Not all errors are reachable and our goal is to determine, for each problem, which errors can be reached.

Listing 1: Program12 reaches the error statement 86 when given the input sequence 6 5 2 4

```
$ echo 6 5 2 4 | ./Problem12
20
20
22
16
error_86
```

Figure 1 shows an overview of the all set of RERS 2017 reachability problems. The problems are classified by size (top to bottom) and complexity (left to right). The first column contains challenges using only plain variable assignments, the challenges in the second column consist of more complex arithmetic operations, finally the problems in the last column make use of complex data structures.

Figure 1: Overview of the RERS 2017 reachability challenges, set up and difficulty of each problem (source: www.rers-challenge.org).



| size | plain, simple | arithmetic, medium | data structures, hard |
|---|---|---|---|
| small | Problem10 | Problem11 | Problem12 |
| medium | Problem13 | Problem14 | Problem15 |
| large | Problem16 | Problem17 | Problem18 |

### 2.2 KLEE

KLEE is concolic execution tool capable of automatically generating test cases that produce high code coverage in a program. KLEE operates as a VM for Low Level Virtual Machine [3] (LLVM) assembly language, therefore the target

[1]The RERS challenges are also the object of a competition which is held annually.

program needs to be compiled to LLVM before KLEE could run it. When launched, KLEE tries to explore every possible path. This is done by keeping track of the constraints on the input each time a condition is encountered and using a constraint solver to determine which direction will be taken. Each condition is then negated in order to explore new paths. Upon execution KLEE produces a set of files, each of them containing the input values used to unlock a certain execution path.

KLEE uses several search heuristics to decide which execution path to explore. The search algorithm can be selected passing a command line option, moreover multiple search heuristics can be combined (in this case KLEE will use them alternately). For example when using the *random-path* search algorithm KLEE maintains a binary tree of the possible execution, where each internal node represent a condition and the leaves represent the current processes. KLEE then select a process by traversing the tree from the root and randomly selecting which branch to follow at each internal node. When using Non Uniform Random Search (NURS) KLEE randomly selects a state according to a given distribution, as for example one based on the minimum distance to an uncovered instruction.

In this assignment we will apply KLEE to the RERS reachability problems to try to determine which error statements are reachable.

### 2.3 LearnLib

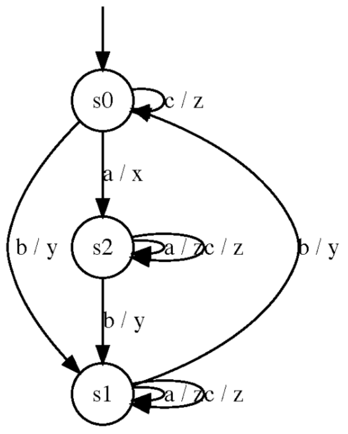
LearnLib is a Java framework for active automata learning. It provides a set of functions which can be used to model a finite state automata from a target system (e.g. a program). This is achieved using the so-called membership and equivalence queries.

Membership queries are meant to investigate on the system and are performed by sending inputs to the target and recording the outputs which will then be used to build a model based on the correlation between input sent and answers received.
Equivalence queries are used to test the model learned in order to verify if it reflects the target system, this is usually done trying find a counterexample, i.e. a sequence of inputs producing different outputs for the model and the system under learning. Once a counterexample is found the model is adjusted to include the counterexample, for this LearnLib goes trough a new round of membership queries. This process is repeated until no counterexamples are found.

LearnLib can make use of several learning and testing algorithms. Among the wide set of available learning algorithm we can find for example the well known **L*** algorithm [4] or the more recent **TTT** automata learning algorithm [5].

Figure 2 shows an example of model produced by LearnLib for a small program accepting "a b c" as input alphabet and retrieving "x y z" as outputs. In this assignment we will use LearnLib to model the RERS reachability problems and try to determine how many error statements are reachable.

Figure 2: Example of model produced by LearnLib



## 3 Methodology and Configuration

For this experiment we applied LearnLib and KLEE to the whole set of RERS reachability problems from 2017 (Problem10 to Problem18).

We used LearnLib configured with four different combinations of learning and testing algorithms as shows table 1 and compared the results . Listing 2 shows how the learning engine of LearnLib has been used for this experiment.
KLEE was used with two different combinations of search heuristics as illustrated in table 2 (those heuristics are the same described in the previous section). Listing 3 shows the command line options used for KLEE.

For each LearnLib and KLEE configuration all problems have been analyzed for five hours on an Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz. In the case of LearnLib the analysis have been performed under an Oracle Java 8 JVM configured with a 4GB heap.

Table 1: Combinations of LearnLib's learning and testing algorithms used for the experiment

| Learning Algorithm | Testing Algorithm |
|---|---|
| L* | Random Walk |
| L* | WMethod |
| TTT | Random Walk |
| TTT | WMethod |

Table 2: Combinations of KLEE's search algorithms used for the experiment

| Search Heuristics |
|---|
| Random Path Selection |
| Random Path Selection + Non Uniform Random Search (NURS) with Coverage-New heuristic |

Listing 2: Usage of the BasicLearner interface to LearnLib

```
BasicLearner.runControlledExperiment(SystemUnderLearning, LearningAlgorithm,
TestingAlgorithm, InputAlphabet);
```

Listing 3: KLEE command line options used for the experiment

```
klee --search=random-path [--search=nurs:covnew] --allow-external-sym-calls
-only-output-states-covering-new -max-time=1800 ProgramN.bc
```

## 4 Results

### 4.1 Individual result for LearnLib

Table 3 shows the results obtained with LearnLib using different configurations after a five hours run on each of the RERS 2017 reachability problems. As we can see there are clear patterns emerging. First, certain algorithms (or combination of algorithms) performs better than others in terms of number of states found. For instance the random walk testing algorithm performs poorly if compared to a more sophisticated algorithm such as W-Method.

Second, LearnLib was more effective in finding errors on smaller programs. This because bigger programs have a larger number of states if compared to smaller programs. In the context of the RERS reachability challenges this means that for smaller programs the number of states to traverse before reaching an error statement is lower. As a result LearnLib is able to explore smaller program more effectively than larger program. Furthermore as the size of the program increase, the number of queries needed rises.

Finally we can observe that there is no big difference in effectiveness between programs having the same size but a different degree of complexity (e.g. 10, 13, 16). This is because LearnLib treats the tested program as a black box, i.e. the program is tested only by observing its output and not by analyzing its inner working. As a consequence LearnLib is not influenced by the complexity of the operations performed inside the program tested. This is a big advantage of LearnLib if compared to white box testing tools.

Table 3: Results given by LearnLib after a five hours run. The results are expressed in terms of error statements reached, states found, number of membership queries (MQ) and equivalence queries (EQ)

| | **Problem:** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** |
|---|---|---|---|---|---|---|---|---|---|---|
| **L* & Random walk** | # statements | 11 of 32 | 11 of 15 | 24 of 38 | 24 of 29 | 43 of 55 | 11 of 20 | 8 of 8 | 12 of 25 | 29 of 52 |
| | # states | 39 | 36 | 58 | 237 | 233 | 232 | 545 | 568 | 768 |
| | # MQ | 2797 | 3970 | 13935 | 26080 | 55935 | 55695 | 59960 | 136335 | 322580 |
| | # EQ | 2819 | 3999 | 13964 | 26109 | 55964 | 55724 | 59989 | 136364 | 322609 |
| **L* & WMethod** | # statements | 32 of 32 | 12 of 15 | 25 of 38 | 25 of 29 | 43 of 55 | 11 of 20 | 8 of 8 | 13 of 25 | 30 of 52 |
| | # states | 118 | 70 | 61 | 267 | 237 | 239 | 637 | 610 | 795 |
| | # MQ | 239806 | 5020735 | 559643 | 1779374 | 1067266 | 1473039 | 1377099 | 1568504 | 869108 |
| | # EQ | 629866 | 5012822 | 557752 | 1775413 | 1062735 | 1468373 | 1369788 | 1553713 | 851487 |
| **TTT & Random walk** | # statements | 5 of 32 | 8 of 15 | 18 of 38 | 22 of 29 | 15 of 55 | 3 of 20 | 4 of 8 | 1 of 25 | 0 of 52 |
| | # states | 23 | 27 | 23 | 89 | 88 | 61 | 54 | 93 | 2 |
| | # MQ | 708 | 1433 | 3360 | 5178 | 9273 | 6740 | 2766 | 12779 | 138 |
| | # EQ | 736 | 960 | 2788 | 5114 | 8178 | 5894 | 2629 | 12664 | 27 |
| **TTT & WMethod** | # statements | 32 of 32 | 12 of 15 | 25 of 38 | 25 of 29 | 43 of 55 | 10 of 20 | 5 of 8 | 6 of 25 | 7 of 52 |
| | # states | 118 | 52 | 62 | 265 | 237 | 216 | 509 | 412 | 350 |
| | # MQ | 441807 | 1615161 | 1631822 | 1733070 | 1448258 | 1818542 | 1991698 | 2232353 | 1773312 |
| | # EQ | 885057 | 1615065 | 1631742 | 1732950 | 1448047 | 1818390 | 1991608 | 2232210 | 1772879 |

### 4.2 Individual result for KLEE

Table 4 shows the results obtained applying KLEE to the RERS reachability problems. We can immediately see that in many cases the number of reachable error statement discovered by KLEE is larger than the one reported by LearnLib, in the next section we will discuss the reasons behind this and compare the two approaches.

There are not big differences in terms of error found between the two configuration we used. Only for the bigger and more complex problems the first configuration seems to perform slightly better. This is because it alternates two different methods and thus is more likely to unlock new paths.

As with LearnLib, we can see that the proportion of errors found decreases as the size of the program analyzed increases. This is because the bigger is the program analyzed the longer and more difficult will be the constraints that KLEE will need to solve in order to unlock new execution paths. However we can also see that KLEE's performance is influenced by the complexity of the program analyzed. While KLEE performs very well with programs 10,13 and 16 which operations consist of simple variables assignments, KLEE is less successful on more complex program having similar size. This happens for example when analyzing programs using arithmetic operations (11,14,17) or complex data structures (12,15,18).

To summarize, KLEE reaches more statements when analyzing smaller and/or simpler programs (e.g. problem 10), and reaches less statements in programs having bigger size and/or complex logic (e.g. problem 18). This is respectively because a bigger size implies longer constraints and complex operations generate more complex constraints which are harder to solve.

Table 4: Number of error statements reached and paths covered after a five hours run using KLEE

| | **Problem:** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Random Path Selection &** | # statements | 32 of 32 | 15 of 15 | 25 of 38 | 27 of 29 | 44 of 55 | 12 of 20 | 8 of 8 | 16 of 25 | 37 of 52 |
| **NURS with Coverage-New heuristic** | # paths | 17585 | 573716 | 868008 | 1243732 | 398068 | 636682 | 13034 | 407176 | 671342 |
| **Random Path Selection** | # statements | 32 of 32 | 15 of 15 | 25 of 38 | 27 of 29 | 44 of 55 | 12 of 20 | 8 of 8 | 15 of 25 | 36 of 52 |
| | # paths | 133839 | 438654 | 762085 | 1644006 | 469153 | 632926 | 27043 | 462386 | 761831 |

## 4.3 Benefits and disadvantages of using KLEE/LearnLib

We have seen that both LearnLib and KLEE are able to discover a substantial number of error statements when applied to the RERS problems. This frameworks implement two radically different techniques: automata learning and concolic execution. But, when should we use concolic execution and when should we instead prefer learning ? In order to answer this question let us analyze what are the benefits and disadvantages of each approach.

### Advantages of concolic execution (KLEE)

- Concolic execution engines such as KLEE are able to automatically discover "corner cases" which are particularly difficult to reach and would probably be overlooked by other testing techniques such as fuzzing and automata learning. This is done by crafting inputs which would specifically negate some conditions in the code
- KLEE let us test a program even without specific knowledge of the input format. The constraints on the input are automatically derived by the constraint solver on the basis of the checks performed inside the program.

### Disadvantages of concolic execution (KLEE)

- Concolic execution is a white box approach and for this reason is only applicable in situations when the source code of the program to test is available. This can be a problem for example when the program under test make use of some proprietary components for which source code is not available.
- In practice for complex programs the search is typically incomplete due to the too large number of execution paths. This issue is also known as the path explosion problem.
- One of the big challenges of concolic execution is the correct and efficient handling of the code which is used by the application under test but doesn't reside in it. System calls are a typical example: when the application under test issues a system call, the execution context is temporarily switched from the application to the operative system where the concolic execution engine cannot track the execution anymore. As a result, concolic execution engines often fail to build and solve constrains which depends on the outcome of system calls.

### Advantages of automata learning (LearnLib)

- Automata learning can be used on black box implementations where no source code is available. For this reason this approach is particularly suited for testing/reversing network protocols and smartcards.
- Automata learning is not influenced by the complexity of the operations performed by the program under test. Differently than concolic execution tools, frameworks such as LearnLib only take into account the output of a program and not how it has been generated (e.g. number and type of the operations). This techniques can therefore give good results even in case of very complex programs.

### Disadvantages of automata learning (LearnLib)

- When testing real world applications it is often very difficult to define the input alphabet. While in the case of the RERS challenges the input alphabet was just constituted by numbers (Listing 1) in the case of real applications the input alphabet could be difficult to determine. Let us take the example of a program interacting though a graphical user interface (GUI), in this case the input alphabet which causes the application to switch from a state to another would probably be constituted of user clicks. However the number and location of the buttons will change accordingly to the state of the application, hence the alphabet should be created taking into account all the different possibilities.

To summarize, both techniques can be used to find bugs in programs and/or compare existing implementations. The choice on which one of the two approaches suits best a situation depends on multiple factors as for example whether the source code of the application under test is available or not. LearnLib can be used when no source code is available to obtain a general view of the behavior of an unknown system. KLEE Concolic execution is in general more powerful since by taking advantage of the direct access to the program's logic it is able to discover corner cases otherwise difficult to reach. This is the main reason why KLEE performed slightly better than LearnLib on the RERS problems. However this technique can only be applied when the source code is accessible.

## 5 (Bonus) Complementing Model Learning with Concolic Execution

As we have seen both methods present advantages and disadvantages but one doesn't exclude the other, in fact they can be complementary. Combining black box and white box techniques is an ongoing research challenge and as showed by [6] it is possible to combine both approaches to obtain an improved results. In [6] LearnLib is combined with the American Fuzzy Lop fuzzer (AFL) by using the result obtained by this last to answer the equivalence queries (testing queries) performed by LearnLib. For this part of the assignment we used a similar technique to combine LearnLib with KLEE.

Figure 3: A diagram showing our attempt to combine concolic execution (KLEE) with model learning (LearnLib).

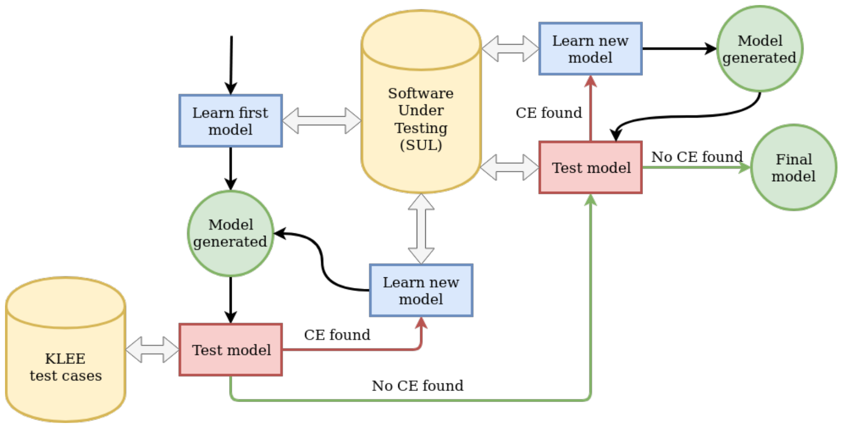


Figure 3 shows our new learning process enhanced using the test cases generated by KLEE. The combined approach that we have used works as follow:

1. A first model of the program, also called System Under Learning (SUL), is generated. This first hypothesis is needed since it constitutes the base on which we will build a more complete model and is generated by LearnLib alone without using the test cases generated by KLEE.
2. We explore the test case generated by KLEE in order to find a counterexample which contradicts the hypothesis. The test cases can be generated by KLEE running in parallel or stored from a previous run, in our case we used the second option.
3. If a counterexample is found we improve our model and go back to point 2.
4. If no counterexample are found at this point the model is consistent with the findings of KLEE. We try to improve it by obtaining a counterexample directly querying the SUL. This is the standard approach used by LearnLib.
5. If a counterexample is found we improve the model and go back to point 4.
6. If no counterexamples are found the learning is finished.

We applied the above techniques on Problem17 and Problem18 since this two problems offer much room for improvement. The learning and testing algorithm that we have used are respectively LStar and W-Method. We decided to pick this two algorithms since they performed the best in our previous experiments.

Table 5: Results of the combined approach on problems 17 and 18

| | **Problem 17** | **Problem 18** |
|---|---:|---:|
| #statements | 19 of 25 | 38 of 52 |
| #states | 712 | 910 |
| #MQ | 813027 | 1182775 |
| #EQ | 798161 | 1153954 |

Table 5 shows the result obtained after 5 hours of analysis using this combined technique. The new approach produced better results than the one obtained using the two tools individually. As we can see the size of the models has largely increased passing from 610 to 712 for Problem17 and from 795 to 910 for Problem18. Thanks to the initial bootstrapping given by KLEE's test cases, LearnLib was also able to find more statements than before.
This results show that combining the two approach is indeed a viable option which could lead to better results.

## References

[1] KLEE LLVM Execution Engine, `https://klee.github.io/`. Last access: 26-03-2018

[2] LearnLib, An open framework for automata learning `https://learnlib.de/` Last access: 26-03-2018

[3] L ATTNER , C., AND A DVE , V. Llvm: *A compilation framework for lifelong program analysis & transformation.* In CGO '04: Proceedings of the international symposium on Code generation and optimization (Washington, DC, USA, 2004), IEEE Computer Society, p. 75.

[4] Angluin, D.: *Learning Regular Sets from Queries and Counterexamples.* Inf. Com- put. 75(2), 87–106 (1987)

[5] Isberner, M., Howar, F., Steffen, B.: *The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning.* In: Bonakdarpour, B., Smolka, S. (eds.) Runtime Verification, Lecture Notes in Computer Science, vol. 8734, pp. 307– 322. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3- 319-11164-3 26

[6] Rick Smetsers, Joshua Moerman, Mark Janssen, and Sicco Verwer, *Complementing Model Learning with Mutation-Based Fuzzing*, 8 Nov 2016