AI for Software Testing and Reverse Engineering Lab 4

INTRODUCTION

Model/machine learning is a key technology for reverse engineering. The basic task in machine learning is to uncover a model from data that explains the observed behavior, which is indeed very similar to reverse engineering. The key difference is the presence of data, but we have an executable that can be used to generate it! This is what active state machine learning does. Moreover, it uses the current hypothesized model to minimize the amount of required data.

Finite-state machines are mathematical models that describe the inner workings of software. The models can be used to understand how a system changes its "state" based on an arbitrary input. In earlier labs, we couldd keep track of a software's state state by assuming that reaching new lines of code triggers new states. In state machines, a new state is only reached if a software's subsequent behavior is different as determined using input-output tests. Different lines of code can thus result in the same state if their behavior is different. Activations of the same line of code can result in different states if their behavior is similar. In machine learning, statistical tests are used to measure this similarity. In active learning, we use techniques from model-based testing.

In this lab, you will implement the key algorithm for learning mealy machines (a variant of a finite-state machine) and apply it to the RERS problems. This includes the learning process (L), the input-output tests (membership queries), and the model-based testing technique (equivalence queries using the W-method).*

LEARNING OUTCOMES

After completing this assignment, you will be able to:

- *Implement the L*(L-star) algorithm*
- *Learn mealy machines for a black-box system*
- *Analyse a black-box system using mealy machines*

INSTRUCTIONS

For this lab, **you will implement the L* (L-star) algorithm** for learning mealy machines of the RERS Problems. Before you start with this assignment, it is **highly recommended** to read the papers listed in the ""Resources" section (specifically the one writted by **Angluin** and the one written by **Chow**). In previous lab assignments, you have implemented methods to find errors in the reachability problems of the RERS challenge (problems 11-19). For this lab **we will be using the Linear Temporal Logic (LTL)** problems, these are also in the RERS folder and correspond **to problems 1-9**. Specifically, you should learn mealy machines from problems **1, 2, 4 and 7**. Our ProblemPin is also still available in the custom_problems folder, and can be useful for debugging your implementation on a smaller example.

We have again provided the setup and instrumentation. To instrument the files for this lab, you can use the learning option e.g.:

```
java -cp target/aistr.jar nl.tudelft.instrumentation.Main --type=learning --file=Problem1.java > instrumented/Problem1.java
```

The instrumentation is minimal and only captures the output of the program.

You will be **working on three different files** located in

src/main/java/nl/tudelft/instrumentation/learning:

- ***LearningLab.java*** for the general learning loop: updating the observation table to make it closed and consistent, generating a hypothesis model, checking for equivalence and possibly incorporating the counterexample into the observation table.
- ***ObservationTable.java*** for checking whether the observation table is closed and consistent.
- ***WMethodEquivalenceChecker.java*** for implementing the W-method equivalence checker.

Task 1: updating the observation table, make closed & consistent

The first step in learning a model using L* is to update the observation table to make it closed and consistent. The methods that **need to be implemented** can be found in the *ObservationTable.java*.

In *LearningLab.java,* we print the current observation table using *ObservationTable.print()*. An example of the observation table is shown below. The row starting with *E* contains all the suffixes in *ObservationTable.E*. The rows labeled with *S* contain all the prefixes in *ObservationTable.S*. Note that for the table below, the suffixes are 0 and 1 and *S* only contains the empty string. The rows with *S·A* are all the suffixes concatenated with each symbol in the input alphabet. In the example below, there are only two input symbols: 0 and 1. For each sequence *S·E* and *S·A·E*, the last output of running that word/trace is shown.

E		0	1
S		ODD	ODD
SA	0	EVEN	ODD
	1	ODD	EVEN

To get a visual representation of an hypothesis, you can save the hypothesis as a dot file using:

```
MealyMachine hypothesis = observationTable.generateHypothesis();
hypothesis.writeToDot("hypothesis.dot");
```

To convert this to a PDF, you need you to have graphviz installed (*apt install graphviz*). You can then use *dot -Tpdf -O hypothesis.dot* to create a PDF. For any inconsistencies, the hypothesis will create dummy states or edges ending with a *?*. The dummy states and edges are also colored red in the PDF.

Your task is to implement the two methods *ObservationTable.checkForClosed* and *ObservationTable.checkForConsistent*. To get a row of the observation table for a specific word from *S* and compare it to another row, you can use the following example:

```
// Get the first word from S.
Word<String> s = S.get(0);
// Get the row corresponding to s
ArrayList<String> row1 = table.get(s);
// Get a symbol from the input alphabet
String a = inputSymbols[0];

Word<String> sa = s.append(a);
// Get the row corresponding to s·a
ArrayList<String> row2 = table.get(sa);
// Check if the rows are equal
boolean areRowsEqual = row1.equals(row2);
```

If you find an inconsistency, you can return something useful to add to either *S* or *E*, for example: (*return Optional.of(sa)*). In *LearningLab.java*, you can use the methods for consistency and closedness. To create a observation table that is closed an consistent by adding to *S* or *E*, for example:

```
Word<String> newPrefix = ...;
observationTable.addToS(newPrefix);
```

By using the methods addToS and addToE, the observation table will automatically be populated with the output of the system under learn.

- What is wrong with an inconsistent observation table? Can you explain it in terms of the mealy machine that can be created from the observation table.
- What is wrong with an observation table that is not closed? Can you explain it in terms of the mealy machine that can be created from the observation table.

Task 2: learning your first state machine, test for equivalence

Once you have a closed and consistent hypothesis model, you can check whether the model matches the system by using an equivalence checker. You can initially use the *RandomWalkEquivalenceChecker*, but in a later task, you will implement the *WmethodEquivalenceChecker.* To verify a method and possibly get a counterexample, use the *verify* method.

```
MealyMachine hypothesis = observationTable.generateHypothesis();
Optional<Word<String>> counterexample =
equivalenceChecker.verify(hypothesis);
```

- When the equivalence checker does not return a counterexample, you are done learning, but is your model guaranteed to be correct? Why (not)?

Task 3: processing a counter example

When you receive a counterexample from the equivalence checker, you then need to process the counterexample such that the observation table is updated to include the behavior for that example. For the processing, you might need to get the output of the system for a specific input. To get the final output after running a trace/word, you can use *sul.getLastOutput(...).*

After processing a counterexample, you can complete the learning loop by continuously making the observation table consistent, checking for equivalence, and when a counterexample is found, incorporate it into the observation table.

Task 4: implementing the W-method

Up to now, you have tested your method using a random walk equivalence checker. Although the random walk equivalence checker can be powerful, it has no guarantees on finding counterexamples.

For this task, you will implement the W-method for equivalence checking. The W-method checks all strings composed of an access sequence (A), a word of length *w* over the input symbols (X), and a distinguishing sequence (W). The set of distinguishing sequences is sometimes also referred to as the characterization set.

The W-method should be implemented in WMethodEquivalenceChecker.java. The verify method should return a counterexample if it finds one. To get the access sequences and the distinguishing sequences, you can use the following methods:

```
accessSequenceGenerator.getAccessSequences(); // For the access sequences
distinguishingSequenceGenerator.getDistinguishingSequences(); // For the
distinguishing sequences
```

To get the output of the system for a specific input you can again use *sul.getLastOutput(...).*

- When the w-method equivalence checker does not return a counterexample, you are done learning, but is your model guaranteed to be equivalent to the system under learn? Why (not)?

Task 5: run on RERS problems

Learn models from the RERS challenges 1, 2, 4, 7 and the ProblemPin in the custom_problems folder using your W-method.

- Plot the number of states over time for each RERS challenge and the ProblemPin. Include the final renders of the problems in your report.
- The ProblemPin represents a keypad where someone must enter a passcode. View the mealy machine of this problem and explain if the model matches your expectation. Do you think this is 'secure'? Why/ why not?

The task below is served as a <u>preparation for the final assignment</u> and will <u>not be graded</u> for this assignment.

Task 6: Comparing to LearnLib

LearnLib is considered to be the state-of-the-art tool for learning state machine models using active learning. For this task, you are asked to compare your implementation of your active learning algorithm to LearnLib.

To run the state-of-the-art Learning with L*, you can use the following code:

```
public static void runLearnLib() {
    LearnLibRunner llr = new LearnLibRunner();
    llr.start(1);
}
```

You can specify the w parameter as the argument to the `start(int w)` method. This corresponds to the *"w"* parameter in your own W-method. In the file `LearnLibRunner.java` you can look at the code for setting up LearnLib. We have chosen the L* algorithm by default, but LearnLib includes more efficient methods for learning models from software. One of these methods is the TTT algorithm. You can change the line `MealyLearner<String, String> learner = lstar;` to `MealyLearner<String, String> learner = ttt;` Compare the state of the art TTT algorithm with your own implementation of L* and answer the following:

- Is your method able to find the same models as TTT? Why, or why not?
- Compare the runtime, as well as the number of membership queries of your method to the state-of-the-art. How does your implementation compare to the state-of-the-art? To keep track of the number of queries for your own implementation, you can modify the LearningTracker to increment a counter for every time the runNextTrace method is called.

RESOURCES

Slides from Lectures ?

Study:

(1) Angluin Dana. "Learning regular sets from queries and counterexamples", Information and Computation, 1987.
(2) Tsun S. Chow. "Testing Software Design Modeled by Finite-State Machines", IEEE Transaction on Software Engineering, 1978.

PRODUCTS

A small report of max two A4 pages (excluding visualizations) answering the questions from all tasks. The report can be extended with at most three A4 pages if visualizations are used. Also provide an archive (tar/zip) containing the code that you have modified and the "DOT" files generated by your own implementation for each problem (so that we can view the models if they are difficult to read in the report).

Make sure you have also provided some instructions on how to run your code.

ASSESSMENT CRITERIA

You can either pass or fail this assignment. We expect everyone to pass. You have to complete all lab assignments. Submissions of which the report text does not fit into two A4s (excluding visualizations) will not be evaluated; the submission will automatically result in a fail.

Code that does not compile/run will not be evaluated; the submission will automatically result in a fail.

Your work will be evaluated based on its completeness (having done all tasks), the correctness of the implementation, and demonstration that you understand the results in the analysis. When deemed insufficient, you will receive feedback and will be given a one-week grace period to fix any shortcomings.

SUPERVISION AND HELP

There will be lab sessions every Wednesday, where the teachers and TAs will be available to answer any questions you may have. The preferred way to ask questions is through Mattermost.

SUBMISSION AND FEEDBACK
The submission is through Brightspace. You will receive feedback within one week after the deadline.