UNIVERSITY OF AMSTERDAM

Authors: Robin Visser & Tristan Laan
Based on the work by:
Bas van den Heuvel
Daan de Graaf

Automaten en Formele Talen
Teacher: R. Reiffenhäuser
Supervisor: Tristan Laan
9th April 2024

Assignment 1

Finite automata for runtime verification

Due: 24th April 2024

Background – Runtime verification

In this series of exercises we are going to produce a framework capable of performing *runtime verification* on *Turing machines*. Runtime verification typically consists of a *system* being observed by a *monitor* to verify compliance to a *specification*. In practice, 'system', generally refers to a closed source (but runnable) binary executable, 'specification' refers to a verifiable property written in a specification language, and 'monitor' to a third party application that either inspects the system as it runs (online monitoring) or which analyzes the system's execution traces, such as log files, after it has halted (offline monitoring).

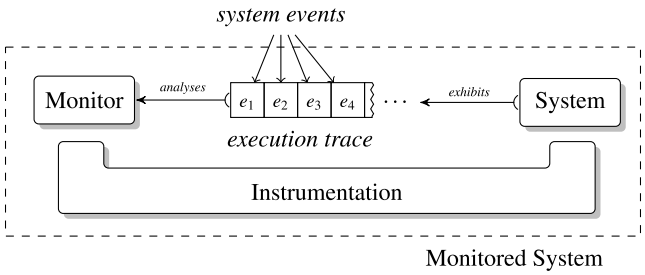

Figure 1: Overview of runtime verification[1]

For the specification language the monitor normally employs one of the following: linear temporal logic, finite state machines (such as **finite automata**) or **regular expressions**. Some runtime *monitors* that you may have encountered in other courses include *strace* and *valgrind*. The latter of which also performs some verification by checking whether a program 'leaked' any memory. As mentioned above, in these exercises we will build our own framework to perform runtime verification. We won't be testing against actual Linux binaries due to the sheer complexity associated with them.[2] Instead, we will focus on runtime verification of a theoretical models of 'real' systems known as Turing machines.

Background – Turing machine

A Turing machine can be considered a mathematical abstraction of a 'real' system. Turing machines will be explained in detail later in the course. for now, we are only interested in the steps that it takes during operation.

[1] Bartocci, E., Falcone, Y., Francalanza, A. & Reger, G. (2018). *Introduction to runtime verification*. Springer
[2] To illustrate the complexity run `$ strace ls` in a terminal to print the *syscalls* necessary to display some files.

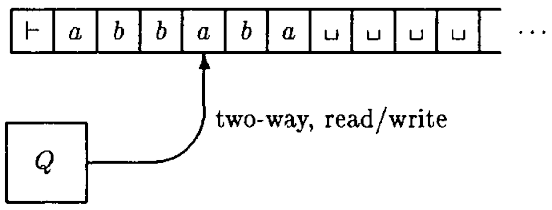

Figure 2: Kozen's 'off-the-shelf' Turing machine[3]

The Turing machine we are going to monitor is the deterministic one-tape model displayed in Figure 2. It's basic components are:

- The *tape* (whose *cells* are currently filled with "⊢ a b b a b a ␣ ␣ ␣ ␣ ..." in Figure 2). The tape is analogous to the memory of a 'real' process. It is delimited on the left by the *left endmarker* (⊢) and infinite on the right.
- The *head*. A Turing machine examines one cell (piece of tape) at the time. The head (represented by the arrow in Figure 2) keeps track of which cell this is. One could argue that the head is similar to the stack pointer of a 'real' process.
- The finite control. Like a finite automaton, a Turing machine has both states and a transition function (typically defined in a state transition table). It is analogous to the programming/instructions of a 'real' process. Other than the name of the current state in the state register, 'Q', no table is shown in Figure 2. Since we have no access to its internal logic one could say this Turing machine is 'closed source'.

Next, we look at how these components are used. Before the Turing machine starts, the input-string (similar to that of a finite automaton), is written on the tape from left to right. In addition the Turing machine's head is placed on the leftmost cell (⊢). When operating it repeats the following steps:

1. Read the symbol from the cell that the head points to.
2. Write a symbol back to the current cell.
3. Move the head either left or right (one cell).

Which symbols the Turing machine decides to write back (and which direction it moves) depends on its *finite control*. For now we are only interested in what the machine does, the *why* comes later.

Assignment 1.1: Lexing a Turing machine trace (55 points)

The first step towards runtime verification of a Turing machine is identifying bits of the execution trace (i.e. tokenization). The trace is handed to us as a string (which may include spaces). The events, which form the execution trace, correspond to the visible actions the Turing machine performs during its steps and are defined in Table 1.

[3]Kozen, D. C. (1997). *Automata and computability*. Springer.

Event representation	Event description
<	Move head one cell to the left
>	Move head one cell to the right
−	Read (next event representation is input)
+	Write (next event representation is output)
<symbol literal>	A literal representation of a symbol-like (e.g. 'aa', '⊢', '␣')

Table 1: Events of a 'closed-source' Turing machine execution trace

An example trace could look like this:

```
- aaa + bb > - ␣ + ab <
```

The goal of the assignment is to use the automaton from Figure 3 to identify and tokenize events within a given trace. Part of this automaton is already implemented in **lexer.py**. It is up to you to:

1. Fill in the missing pieces of the automaton in the **create_fa()** function.
2. Implement the **lexer(fa, trace)** function that tokenizes the input trace.
    - The FA in Figure 3 is designed to recognize one token (consisting of one or more trace-string elements) at the time. Reset the FA in between tokens.
    - **Use the FA's methods** (see the appendix). Most of the lexer's logic should be built around the True/False output of the **transition()** method.
    - Characters that are not part of the FA (like '!', '&', etc.) are wrong input and should not be ignored (see the lexer's docstring/comment).
    - It is possible to recognize wrong input in the FA/lexer without explicitly checking if each character in the trace belongs to the input alphabet.
    - There is *no* guarantee that a 'SPACE' token will be present between two other tokes. *Remember, we are only identifying and verifying individual tokens, not the relation between them.*
    - If you don't know where to start, try to tokenize a trace consisting of only a single character.

The lexer function takes the automaton and a Turing machine execution trace and should return a list of tuples, like the following examples:

```
Trace: "a"
[('a', 'SYMBOL')]

Trace: "- aaa + bb <"
[('-', 'READ'), (' ', 'SPACE'), ('aaa', 'SYMBOL'), (' ', 'SPACE'), ('+', 'WRITE'),
 (' ', 'SPACE'), ('bb', 'SYMBOL'), (' ', 'SPACE'), ('<', 'MLEFT')]

Trace: "18Ajkh827yhhsg"
[('18Ajkh827yhhsg', 'SYMBOL')]

Trace: "j022␣39-+---+␣"
[('j022', 'SYMBOL'), ('␣', 'BLANK'), ('39', 'SYMBOL'), ('-', 'READ'), ('+', 'WRITE'),
 ('-', 'READ'), ('-', 'READ'), ('-', 'READ'), ('+', 'WRITE'), ('␣', 'BLANK')]
```

Assignment 1.2: Verifying Turing machine steps (45 points)

Now that we have a functional lexer capable of filtering tokens out of a trace, we can begin to assign/verify some meaning to them. The background section above emphasizes that a Turing machine performs it actions according to a strict order (steps). An execution trace of a valid Turing machine should reflect that order, that is, the number of events in a given trace should be a multiple of 5, and the tokens from the trace should be accepted by the FA in Figure 4.

The goal of the assignment is to use the automaton in Figure 4 to verify whether the tokens in a lexed trace comply with the behaviour expected from a Turing machine. It is up to you to:

1. Create the automaton in **verify.py**
2. Implement the **verify_steps(fa, trace)** function.
    - **verify.py** tries to use the lexer from the previous assignment by default. If it is not functional, you will have to test the **verify_steps()** function manually.
    - The verification function should only use the token portion of a lexed trace.
    - 'SPACE' tokens should be ignored (this must be done manually), alternatively the automaton in Figure 4 could be adjusted to handle them.

Submission & Grading

Submit your work to Canvas before 24th April 2024 by submitting the tarball **PO1.tar.gz** created by running `./create_submission.sh`.

If your code is not written in Python 3 you will receive no points. In addition, make sure your code:

- Complies with the **PEP8** style guide[4] (e.g. by using flake8).
- Is properly commented.

If your program fails to run (or crashes) your work will be graded by judging the code alone, but it will no longer be possible to score all the points. Your code will be tested using **Python 3.11**.

On the other hand, fully functional code does not guarantee that you will receive a perfect score. For example, we may deduct points if you do not use an FA to solve the assignment. You can see the rubric in CodeGrade for a complete overview of how your submission is graded.

After submitting, make sure the **'Verify submission'** and **'Check code style'** AutoTests have **passed** in CodeGrade. **If we have to modify your code to run the AutoTests, you will not be able to score all the points!**

Assignment due date: 24th April 2024

[4] https://www.python.org/dev/peps/pep-0008/

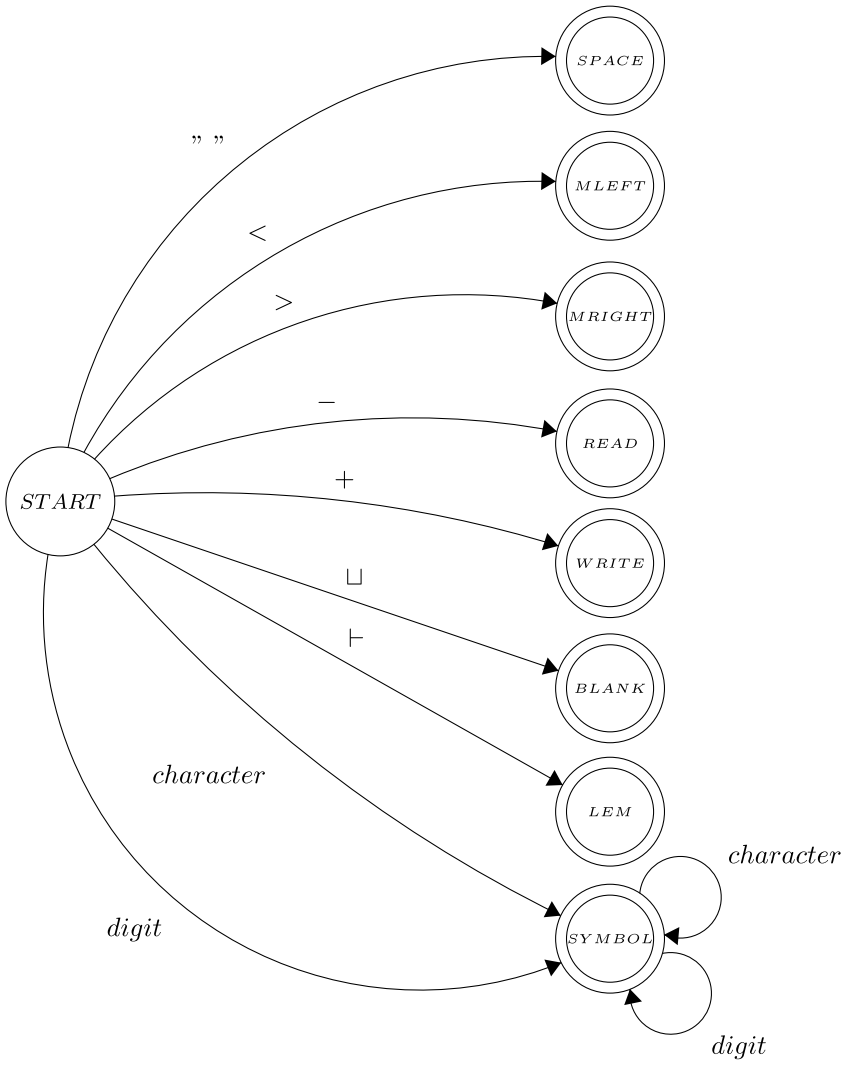

Figure 3: FA designed to tokenize execution traces

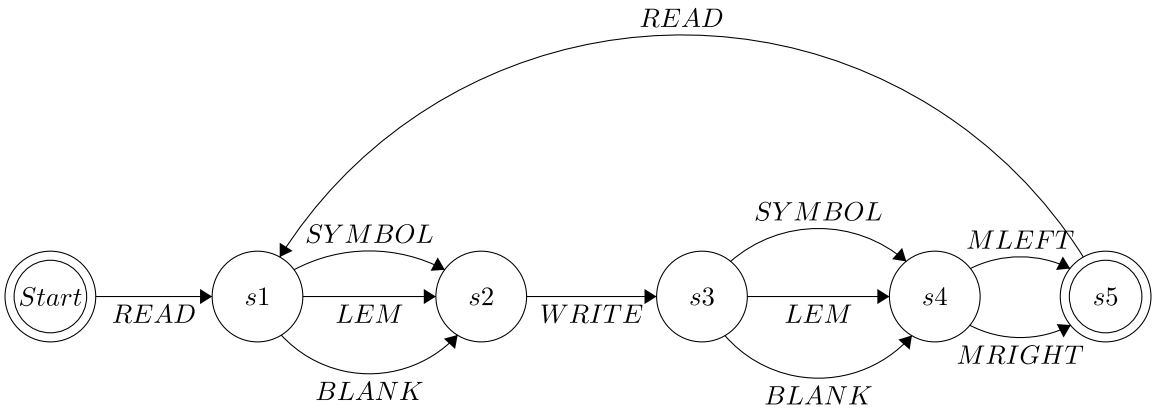

Figure 4: FA for pattern matching execution trace steps

Appendix – Finite automata and FA.py

The FA class in the assignment (**FA.py**) is similar to the FA class defined in Kozen. Its most important attributes are:

- **FA.py** is deterministic, i.e. **no epsilon transitions**.
- The FA ignores non-existent transitions. So you do *not* have to specify every possible transition.
- Warnings can be enabled by passing the `--verbose` or `-v` flag from the command line. (This can mostly be ignored for this assignment)
- The notation for the transitions is a dictionary of dictionaries, where the outer dictionary maps the state, and the inner dictionary maps the symbol.
- You can use the **reset** function to reset the FA to the starting state.
- You can use the **transition** function to feed the FA an element of Σ, and transition to a new state. Will return **False** in case the transition is not defined.
- You can use the **is_final** function to check if the current state is final.