

Assignment 1

Finite automata for runtime verification

Author: Robin Visser

Due: 30 April 2023, 23:59

Supervisor: Peter Soulard

Teacher: R. Reiffenhäuser

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).

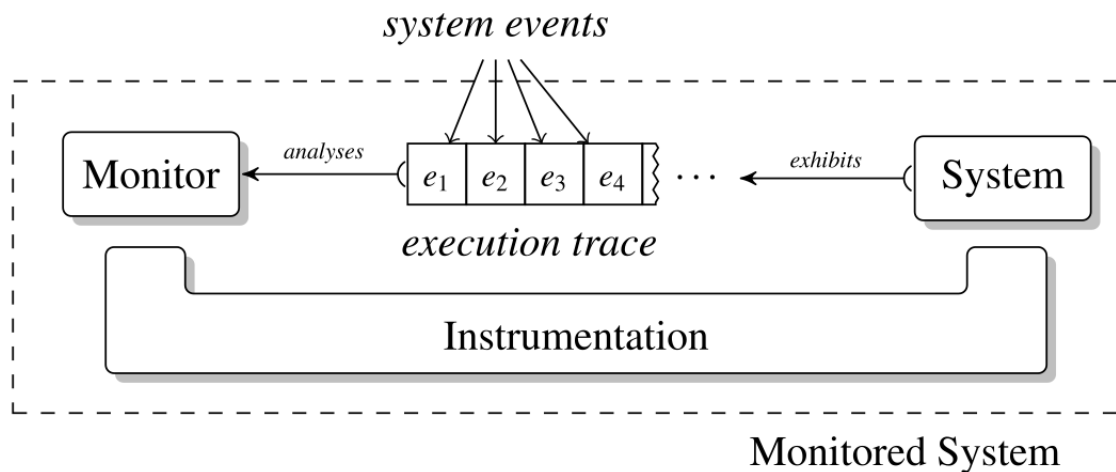


Figure 1: Overview of runtime verification¹

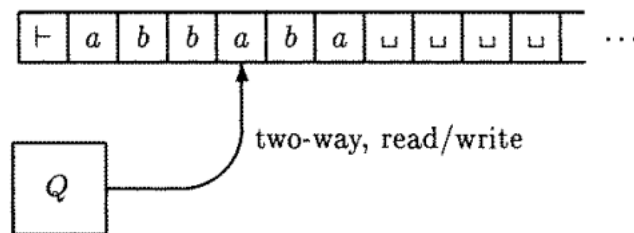
For the specification language the monitor normally employs one of the following: linear temporal logic, finite state machines (such as **finite automata**) or **regular expressions**. Some runtime *monitors* that you may have encountered in other courses include *strace* and *valgrind*. The latter of 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.

¹Bartocci, E., Falcone, Y., Francalanza, A., & Reger, G. (2018). *Introduction to runtime verification*. Springer

²To illustrate the complexity run `$ strace ls` in a terminal to print the *syscalls* necessary to display some files.

Figure 3: Kozen's 'off-the-shelf' Turing machine³

The Turing machine we are going to monitor is the deterministic one-tape model displayed in Figure 3. It's basic components are:

- The *tape* (whose *cells* are currently filled with "⊢ a b b a b a ␣ ␣ ␣ ␣ ..." in Figure 3). 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 3) 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 3. 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 step:

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 (75 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 the Table below.

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.

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

An example trace could look like this:

```
- aaa + bb > - □ + ab <
```

The goal of the assignment is to use the automaton from Figure 4 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 4 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** (look at `FA.py`). 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 tokens. *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: "18Ajk827yhhsg"
[('18Ajk827yhhsg', 'SYMBOL')]
```

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

Assignment 1.2 - Verifying Turing machine steps (25 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 its 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 5.

The goal of the assignment is to use the automaton in Figure 5 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 5 could be adjusted to handle them.

Submission & Grading

Submit your work to Canvas before **Sunday, 1 May 2022, 23:59** by putting your files in a tarball. *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 guide⁴ (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.

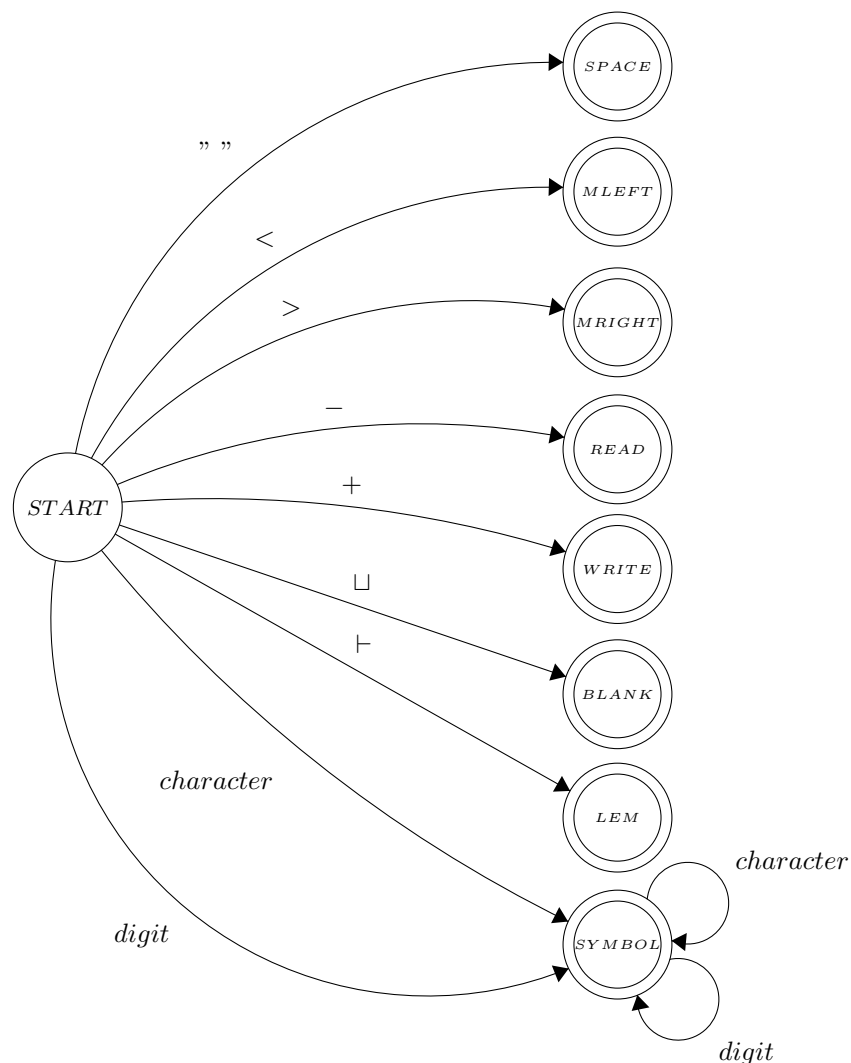


Figure 4: FA designed to tokenize execution traces

The 'character' in Figure 4 is the set of all upper-/lowercase letters in the Latin alphabet.

⁴<https://www.python.org/dev/peps/pep-0008/>

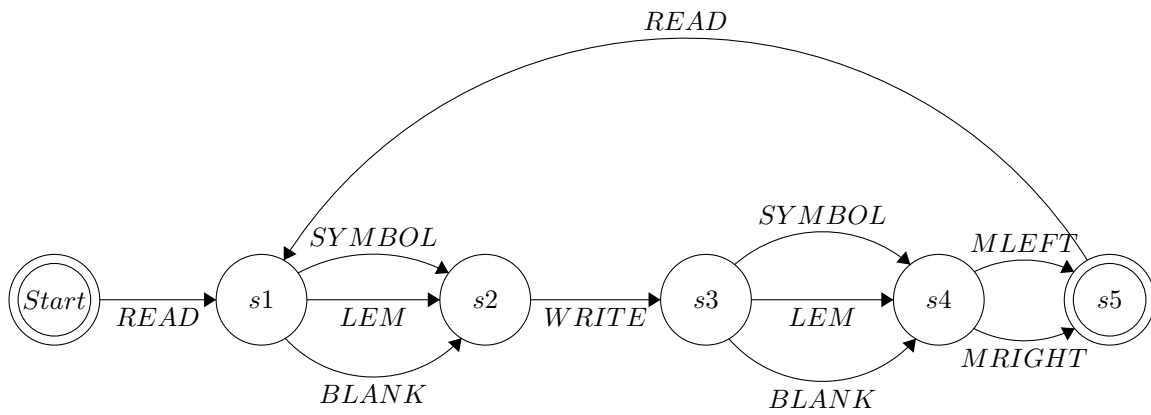


Figure 5: FA for pattern matching execution trace steps.