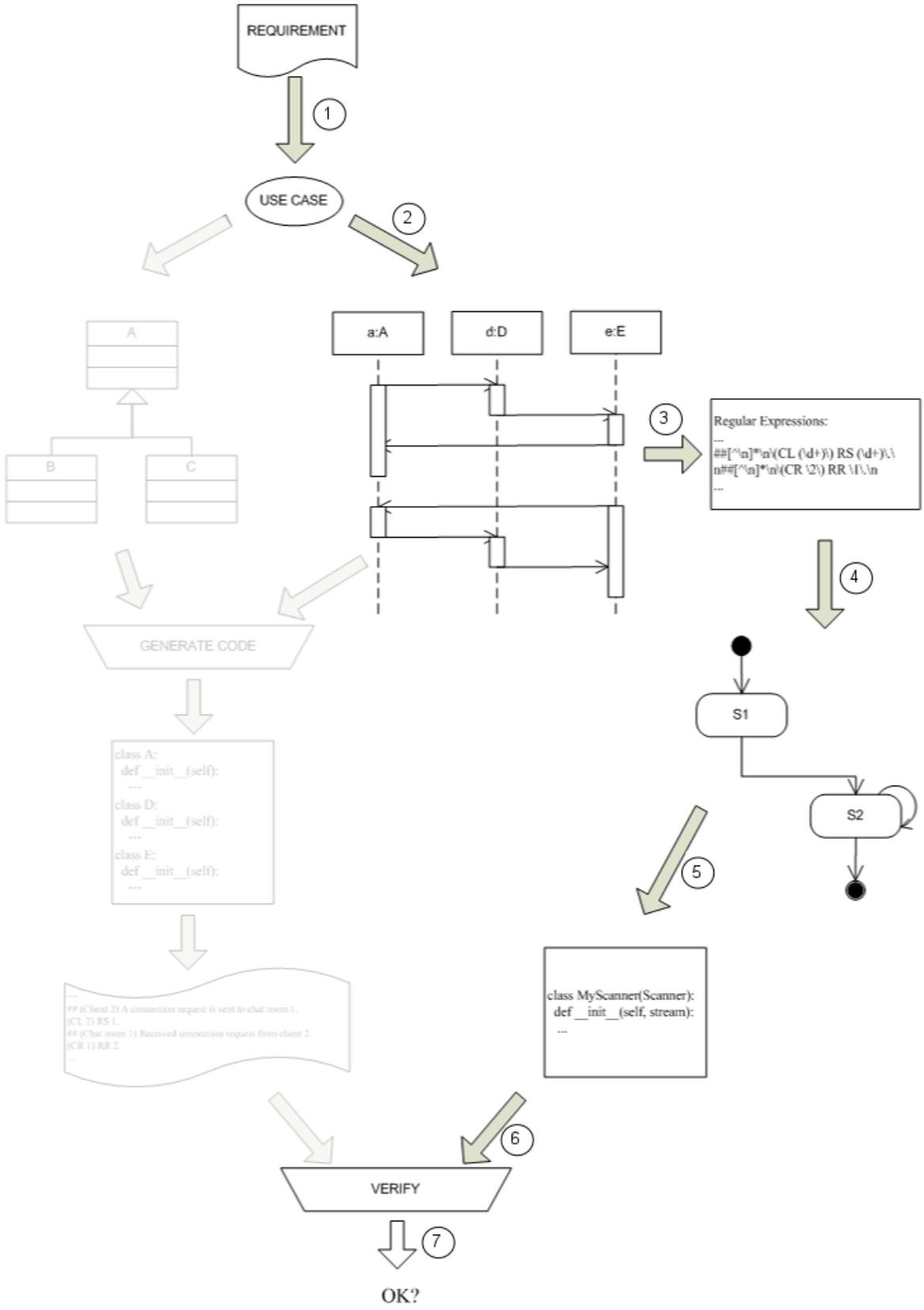
General information

- The due date is 18 October, before 23:55.
- Submissions must be done via BlackBoard. Beware that BlackBoard's clock may differ slightly from yours. *All* results must be uploaded to BlackBoard and accessible from links in the main (index.html) file.
- The assignment must be made in groups of maximum 2 people. It is understood that all partners will understand the complete assignment (and will be able to answer questions about it). Clearly identify who did what. Groups should be made up, whenever possible, of one international and one local student.
- Grading will be done based on correctness and completeness of the solution. Do not forget to document your requirements, assumptions, design, implementation and modelling and simulation results in detail!
- To get feedback about the assignment workload, provide the number of hours you spent on this assignment.
- Contact: Hans Vangheluwe.

Goals

In this assignment, you will use the Use Cases, UML Sequence Diagrams, Regular Expressions, and State Automata modelling languages to design and verify a railway junction controller. You will also make the link to an output trace an actual implementation in Python to perform the checking. At the end, you will be able to automatically determine, based on the obtained trace file, whether all requirements are satisfied or not.

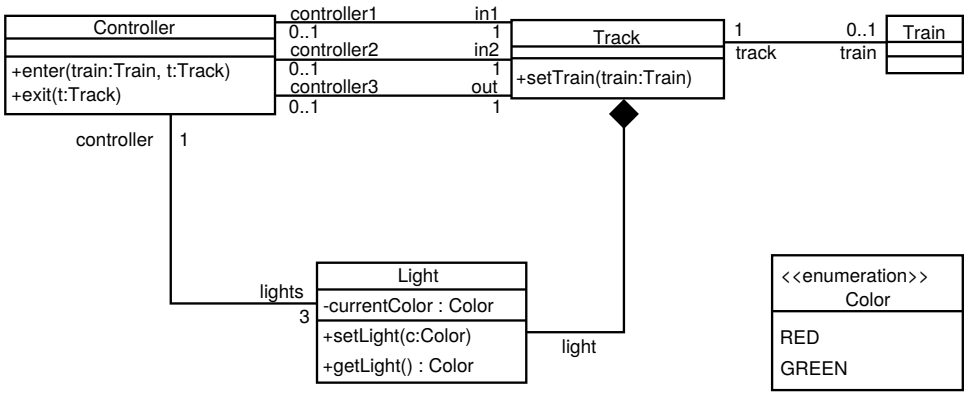


The problem

You are given an implementation of a railway junction controller, which needs to set the traffic signals of the segments coming in to the junction. It is necessary to check whether or not the specified requirements are validated. You are, however, only given access to a trace file of the execution. This trace file contains debugging information about the junction, such as all incoming signals, outgoing signals, and internal timers that get triggered. Due to this verbosity, and the total runtime of the execution, the file is quite long, and therefore validating the requirements is not to be done manually.

The validation will thus be done automatically. You will first model a set of Regular Expressions, and then a set of Finite State Automata, which you will subsequently encode, to automatically verify whether the system implementation complies with the system specification given in the requirements below (and modelled visually in Sequence Diagram form).

The structure of the junction is described in the following UML Class Diagram:



The behaviour can be described as follows:

- By default, all traffic lights that prevent the entry to the junction are set to red for safety.
- The railway segments continuously check for the presence of a train on that segment. As soon as a train is detected, a signal is sent to the controller.
- The controller will process these incoming signals, and will put the traffic light of the requesting segment to green. This on the condition that there is currently no train on the junction itself, and that the traffic light on the other entry to the junction is red too.
- As soon as a train has entered the junction, all traffic lights for entry to the junction are put to red again.
- If it is detected that the junction is clear again, new (or queued) requests are handled.
- For safety, the signal to the traffic light is sent out every second (such that traffic lights can detect if there is a problem with the connection).

The behaviour to be verified (use cases) is as follows:

1. If a train wants to enter the junction, it will eventually get a green light.
2. If a train enters the junction, all traffic lights to the junction become red.
3. If a train is on the junction, all traffic lights will remain red until that train has left the junction.
4. If two trains are waiting to enter the junction at the same time, permission will be granted in order of arrival (i.e., the first train to arrive will get a green light, and the second one has to wait).
5. The controller will send the traffic light signals out every second.

For simplicity, you will only need to check use case 3 and 4.

Note that the trace will not terminate while there are still trains on any of the tracks. That is, for every enter event, there will be an exit event in the trace. You can use this information to make some of the rules a little simpler. For example, if a train wants to enter the junction, the trace will only terminate when that train has left the junction.

You will need to perform the following tasks step by step:

1. Write full Use Cases using the use case template for ONLY use cases 3 and 4.
2. Design the dynamic interaction behaviour in UML Sequence Diagrams for the above use cases (in this assignment, you only need to verify TWO use cases: use case 3 and use case 4), using the textual rendering tool PlantUML, or using the online tool WebSequenceDiagrams. Start from the Class Diagram to know what objects and methods you can use.
3. Write regular expressions (refer to the format of the given output trace) for verifying the above use cases. To make life easier, we use abbreviations to shorten the messages that you need to recognize in your RegExp/FSA. Here are the mappings:

   ```
   E := A train enters the specified segment
   R := A red signal is sent to the specified segment
   G := A green signal is sent to the specified segment
   X := A train leaves the specified segment
   ```

   Beyond that, each segment has a simple encoding:

   ```
   1 := left incoming railway segment
   2 := right incoming railway segment
   3 := outgoing railway segment
   ```

   For example, "G 1" means that the left incoming railway segment got a green light. As you can see, any message in the output file which starts with "#" is some comments to make the output readable. You need to ignore these messages when you do verification. The following is a complete example regular expression for use case 1:

   ```
   Regular expression pattern for rule 1 (for segment 1):
   ^((([^E].*)|(E [23]))\n)*(E 1\n(.*\n)*G 1\n((([^E].*)|(E [23]))\n)*)*$

   For segment 2:
   ^((([^E].*)|(E [13]))\n)*(E 2\n(.*\n)*G 2\n((([^E].*)|(E [13]))\n)*)*$
   ```

   An explanation of rule 1 for segment 1 follows: we try to find an E 1, for which we can skip over everything that is not equal to E 1. As soon as E 1 is then found, we keep matching everything until we find a green light for segment 1. After that, we can match

anything except for E 1 again. Note that "anything except for E 1" is encoded as

```
(([^E].*)|(E [23]))\n
```

As you want this to be correct for both incoming segments, you need to run both of them seperately. If all of them match, then the output is as expected. However, if either of them doesn't match, an error is found.

Clarification: the above uses a Regular Expression notation commonly used in UNIX Regular Expressions (as used in the stream editor `sed` for example).

- `[eE]` stands for `e` *or* `E`.
- `[a-z]` stands for one of the characters in the range `a` to `z`.
- `^` means "match at the beginning of a line/string".
- `$` means "match at the end of the line/string".
- `X|Y` means "match either X or Y", with X and Y both sub-expressions.
- `[^x]` means *not* `x`, so `[^E].*\n` matches every line except those that start with the `E` character
- `.` matches any single character.
- `X?` matches 0 or 1 repetitions of `X`.
- `X*` matches 0 or more repetitions of `X`.
- `X+` matches 1 or more repetitions of `X`.
- `\` is used to escape meta-characters such as `(`. If you want to match the character `(`, you need the pattern `\(`.
- The `(` and `)` meta-characters are used to memorize a match for later use. They can be used around arbitrarily complex patterns. For example `([0-9]+)` matches any non-empty sequence of digits. The matched pattern is memorized and can be referred to later by using `\1`. Following matched bracketed patterns are referred to by `\2`, `\3`, etc. Note that you will need to encode powerful features such as this one by adding appropriate actions (side-effects) to your automaton encoding the regular expression. This can easily be done by storing a matched pattern in a variable and later referring to it again.

You are welcome to use different variant notations (such as the one used in the Python Regular Expression module) as long as you explain your notation.

4. Design a FSA which encodes the regular expressions for verification (example). You can use this tool (note that it is not possible to mark an initial state, so just name it "init").
5. Implement this FSA for verification in the provided code framework (see scanner.py, examples are included at the bottom of the file).
6. Run this FSA implementation (which in turn implements the Regular Expressions which in turn encode the checking of interaction behaviour use cases which were modelled as Sequence Diagrams) on the given output trace to verify the specification.
7. There is an intentional bug in the implementation (which is visible in the trace) which makes it fail to satisfy the system specification. You need to figure out which requirement is violated, and show your FSA which checks this.
8. Write a report that explains your solution of this assigment. Include your models and discuss them.

For all parts, you are allowed to use either a 'positive match' or 'negative match'. With a positive match, the code is deemed correct if the trace matches your description (in regular expressions, FSA, ...). With a negative match, the code is deemed wrong if the trace matches your description. Do explain why you used either kind of match, and make sure to reply "correct" or "violation" in the end, instead of saying "matched" or "not matched". Choosing the right kind of match might significantly shorten your solution!

For example, you could check requirement 1 by checking that every E 1 and E 2 is met with an E 3, thus a positive match: if your regular expression matches, the requirement is satisfied. You could also check the requirement by matching a situation in which there is an E 1 without any E 3 after it, thus a negative match: if this regular expression matches, the requirement is unsatisfied.

Practical issues

- All parts of this (and future) assignments use Python 2.7. Any version of Python 2.7.x will work, but make sure not to use the newer Python 3!
- The output trace can be found here: output
- Programs used to implement FSA: the scanner is in scanner.py which requires an input stream class CharacterStream found in charstream.py.
- Useful links:
  - Use cases: http://www.cs.mcgill.ca/~joerg/SEL/COMP-533_Handouts_files/COMP-533%204%20Use%20Cases.pdf
  - Class diagrams: http://www.uml-diagrams.org/class-diagrams-overview.html
  - Sequence diagrams: http://www.uml-diagrams.org/sequence-diagrams.html
  - Regular expressions: http://www.zytrax.com/tech/web/regex.htm
  - Finite state automata: http://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html
  - Test out regular expressions online: https://regex101.com/