Checking Requirements: Use Cases, Sequence Diagrams, Regular Expressions and State Automata

Ken Bauwens 20143225
and Baturay Ofluoglu 20174797

26/10/17

Contents

1 Use Cases Description

1.1 Use Case 3 Description

Use case: Adjusting traffic lights while a train is on the junction
Scope: Junction Control system
Level: Sub-function
Intention in context: The train intends to cross the junction safely.
Multiplicity: Only one train can be on the junction at any time
Primary actor: Trains
Secondary actor: Controller
Facilitator actor: Traffic Lights
Scenarios:
Main success:

- Train has arrived on the junction
- All Traffic Lights remain red
- Train leaves junction

1.1.1 Use Case 3 Explanation

In use case 3, the level is sub-function because this use case is one of the sub-branch of junction control system. The primary actors has been chosen as trains because they intend to cross the junction safely. In this model train also uses controller to reach its target. Thus, controller is the secondary actor. In addition, lights used by secondary actor, controller, to communicate with the primary actor which are trains. Therefore, traffic light is considered to be a facilitator.

1.2 Use Case 4 Description

Use case: Priority for multiple trains waiting at the junction
Scope: Junction control system for multiple trains
Level: Sub-function
Intention in context: Trains want to cross the junction fairly
Multiplicity: There must be two trains to have priority
Primary actor: Trains
Secondary actor: Controller
Facilitator actor: Traffic Lights
Scenarios:

- **Main success:**
    1. Two trains are waiting at the junction
    2. Traffic light for earliers arriving train turns green
    3. Traffic light for latest arriving train stays red
    4. Earlier Train enters junction
- **Extensions & exceptions:**
    1. Two trains are waiting at the junction
    2. They arrived at the exact same time (Possibly because of low time presicion)
    3. Assign earliest status to random train
    4. Execute main scenario

1.2.1 Use Case 4 Explanation

This use case is a sub-function for the entire function of crossing the junction. Therefore we decided that the level is sub-function.
The primary actors are trains. They want to reach a goal (Crossing the junction fairly). They use the controller to achieve this goal, which is why the controller is a secondary actor. The lights facilitate the communication to ensure the use case runs correctly.

The main scenario is fairly short, because we only consider the steps that actually pertain to the use case. For example, the second train should get a green light as soon as the first one leaves the junction. This however is not part of this use case (it is more part of the first use case), so we didn't include it.

2 UML Sequence Diagrams for the use cases

2.1 UML Sequence Diagram for Use Case 3

For the entire sequence diagram, refer to Appendix A
In Use Case 3, we only consider the out track and the controller labeled as "TrackOut" and "Controller" in the UML sequence diagram.
At this use case, we assume that a train is on the junction. Therefore, it occupies the out track. If the out track is occupied, then the controller detects and turn all traffic lights to red. Note that we did not include setLight() method to the sequence diagram because the lights are not changed if a train is on the junction. By default, they are all set red and for this case we do not need to change it.

2.2 UML Sequence Diagram for Use Case 4

For the entire sequence diagram, refer to Appendix B
We use Light1 and TrackIn1 to denote the track on which the first train arrives.
The flow for this diagram is fairly straightforward, despite of the size of the diagram. We assume that we start with a train on each track. As soon as the TrackOut detects that the train leaves that track, it sends a signal to the controller, who will set the light for the first arrival to green. This train can then proceed to the outputtrack which will result in the TrackIn1 detecting that the train left. TrackIn1 will thus send a signal to the controller and the controller will set Light1 back to red. Then this entire transaction will happen again when this first train leaves TrackOut, but this time for the train that arrived secondly.

3 Regular Expressions to verify use cases

3.1 Use Case 3

Our regular expression for use case 3 is:
`^(([^E]|(E [12])).*\n)*((E 3\n([^XG].*\n)*X 3\n)(([^E]|(E [12])).*\n)*)*$`

For this regular expression, we use positive match. If the regex match with all trace file then it means that use case 3 is verified.
The first group of regular expression `^(([^E]|(E [12])).*\n)*` tries to match with every line that does not start with E except E 1 or E 2. Therefore, when it capture E 3 it will stop. We target to find E 3 because it indicates that there is a train at the junction. After we find out that there is E 3 then all the traffic lights must be red until train leaves from the junction. The leaving train from out track is symbolized as X 3. So between the lines E 3 and X 3 there should not be any G 1 or G 2. To satisfy this constraint we used `((E 3\n([^XG].*\n)*X 3\n)(([^E]|(E [12])).*\n)*)*$` this expression. Basically, it starts with E 3 and match with every line that does not start with X or G until there is X 3. After it finds X 3 then it will look for another E 3 to repeat this pattern if there is more.

3.2 Use Case 4

Our regular expression for use case 4 is:
`^(E 1\n(([^G].*|G 2)\n)*?E 2\n([^G].*\n)*G 2)$` (For track one arriving first)
and
`^(E 2\n(([^G].*|G 1)\n)*?E 1\n([^G].*\n)*G 1)$` (For track two arriving first)

We decided to combine these in a singular regex using the OR operator.
We also decided to use negative matching, because this would be easier to implement. The goal is then to have zero matches when trying to match the trace to the regex.

The first regex will first look for a E 1. It will then allow anything except for a G 1. If it already discovers a G 1 before discovering a E 2, the firstly arrived train can already move on and it does not adhere to the use case. As soon as it finds a E 2, it looks for a G 2 before finding a G 1. This is obviously a violation of the usecase.

For the second regex, the roles of E 1 and E 2, and of G 1 and G 2 are switched.

4 Designing a FSA to verify regular expressions

4.1 Use Case 3

For the FSA for use case 3, refer to C
We used a positive match for the use case 3. Therefore, all states except Error can be accepted as verified. If it reaches "Error" state then it means that the Use Case 3 is violated.
In this FSA, our alphabet includes all English characters and "#". Also, note that " " symbolize space character in our model. FSA stars from "Init" states. If the given Input would be "E 3\n G 1\n X 3" then the FSA would follow this path: Init - 2 - 3 - 4 - 5 - ERROR. After reaching error state, it indicates that the case is violated.

4.2 Use Case 4

For the FSA for use case 4, refer to D
As indicated in the section about regular expressions, we decided to use negative matching for use case 4. This greatly reduces the complexity of the FSA.
We also decided to change our alphabet. Instead of using single characters, we used the set containing al valid operations in the trace without any of the comments. This results in an alphabet {E 1, E 2, E 3, X 1, X 2, X 3, R 1, R 2, G 1, G 2}
If at any point during the execution the fsa reaches a final state, it is an indication of failure and an incorrect trace.
The FSA starts in the INIT state. The FSA is symmetric across a horizontal line. This is because each part describes one of the two subsections of the regex. The upper part describes when a train arrives on the second track, another train arrives on the first track, and this second train gets a green light first. The lower part is the opposite.
An illegal entry would be E 1\n E 2\n G 2, with maybe some comments or irrelevant text in between each one of these statements. On processing an entry like this, the automaton will reach an end state. If it discovers a G 1 before the G 2 however, the entry obviously is correct, so it returns to the init state.
Keep in mind that this automata only checks for use case 4.

5 Implementing FSA for verification

5.1 Use Case 3

To implement FSA for use case 3, we used the provided scanner.py code to scan each character of trace file and we wrote UseCase3Scanner.py to implement our FSA. This implementation verifies that trace.txt file does not violate use case 3.

5.2 Use Case 4

As said in the "designing a FSA"-section, we used a different alphabet for use case 4. This reduced the amount of unnessecary states.
This also meant that we had to adjust the scanner to get first 3 letters of each line. The modified implementation looks for \n and then it scans the first three chars such as "E 1". This new scanner

can be found in "scannerUseCase4.py".
The implementation is fairly straightforward. We simply implemented the fsa in the "UseCase4Scanner.py"-file.
We included two example files. One of them is the given tracefile, which fails because it contains the bug. The other file is "traceCorrect.txt" which will succeed.
To switch between these files, the correct lines of code have to be commented out or in. Refer to the README.

6 Revealing the bug at the trace file

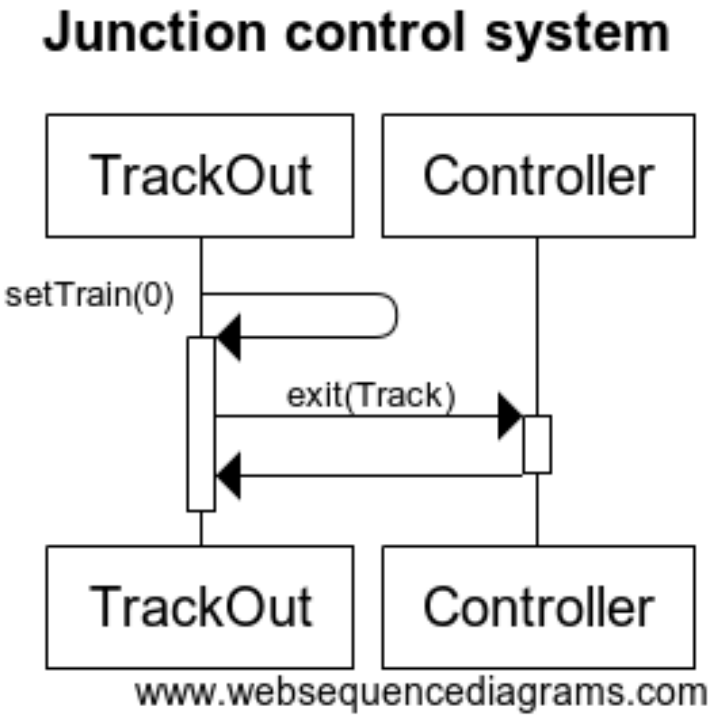
We discovered that the bug is a violation of use case 4. Consider the following:

- Train1 arrives on track 1
- Train2 arrives on track 2
- Train1 gets permission to enter junction
- Train3 arrives on track 1
- Train1 leaves junction

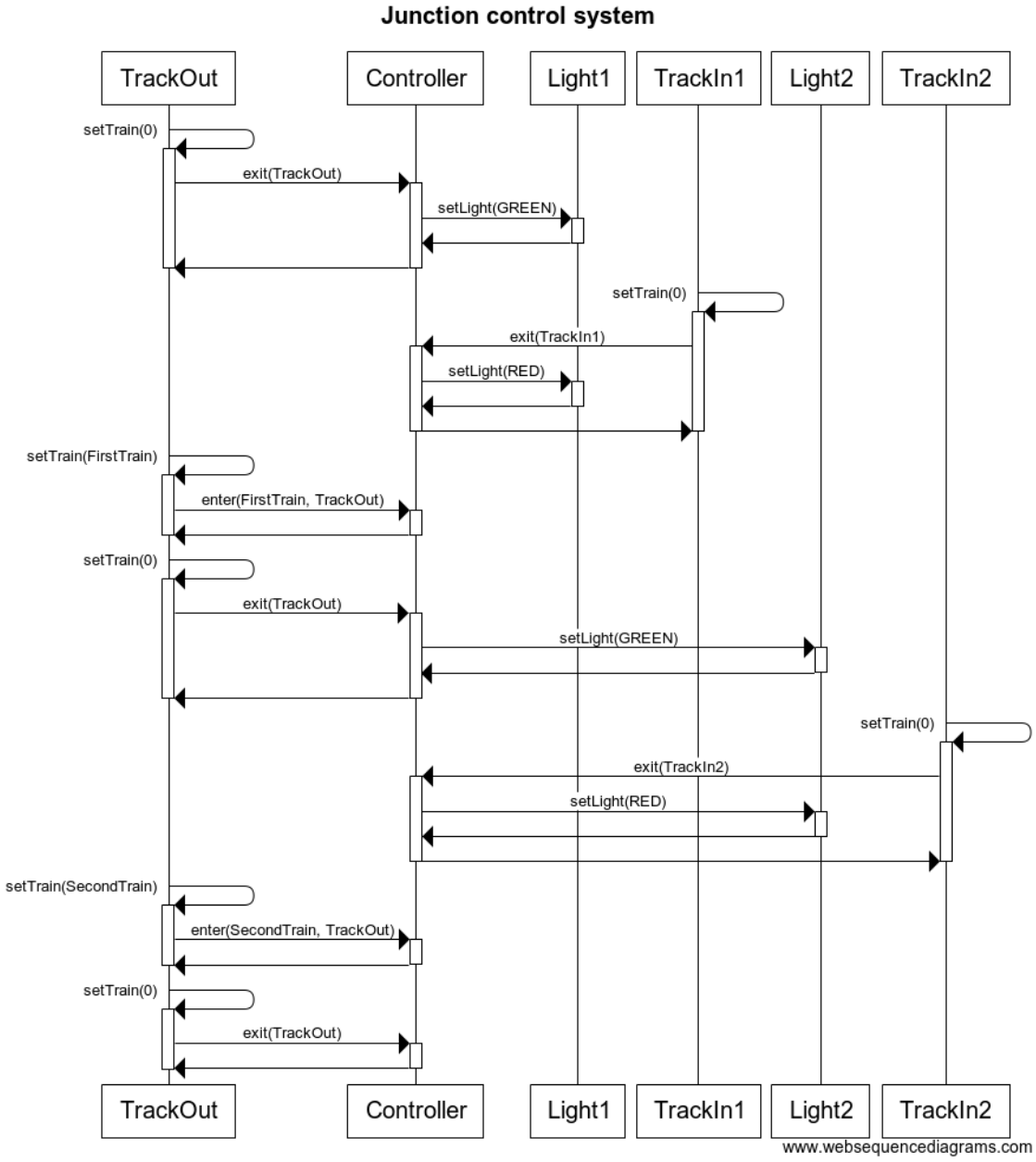
At this point, Train2 should get a green light, because he arrived before Train3. In the tracefile this is not the case, and Train3 will get a green light.
This bug can be seen by executing the FSA for Use Case 4. It will result in a violation.

Appendices

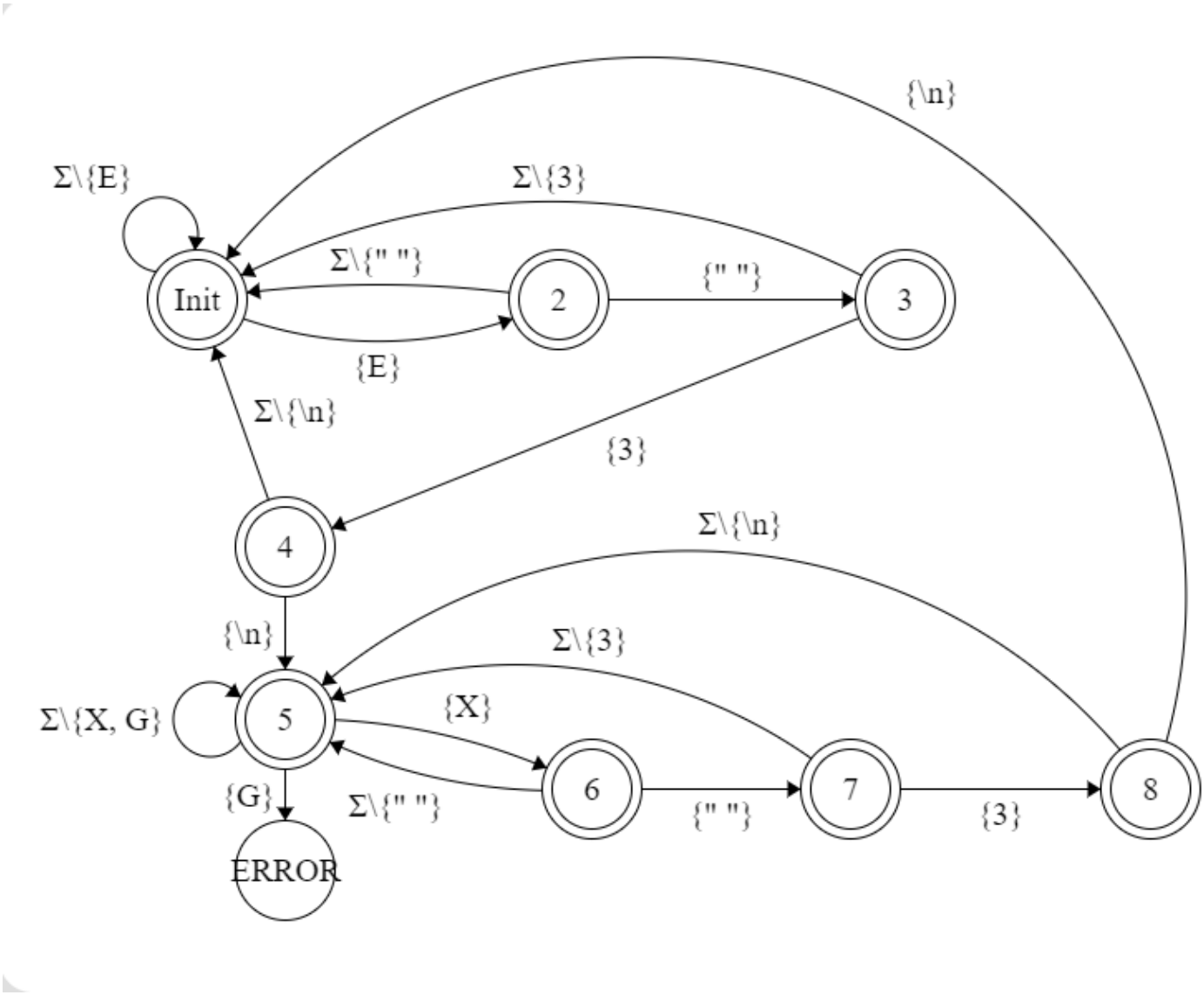
A Sequence diagram Use Case 3



B Sequence diagram Use Case 4



C FSA Use Case 3



D FSA Use Case 4