Modelling of Software Intensive Systems

Assignment 1: Requirements Checking

October 18, 2023

Thomas Gueutal
s0195095@ad.ua.ac.be

Felix Vernieuwe
s0196084@ad.ua.ac.be

1 Use Cases

1.1 General *System* Overview

The tables below give an overview of the different actors, descriptions and goals within the Ramp Metering *System* They are provided for the sake of completeness, hopefully making this report understandable without the assignment file.

Actor	Role	Brief Description
System	Primary	The Ramp Metering *System* Controller that manages the rate at which *Car*s merge into the freeway through the ramp. The *System* manages a single ramp. We may refer to *System* as the "Controller" in relation to UML diagrams. It governs the rate at which *Car*s merge into the freeway. Possible flowthrough rates, and thus system states, are fast and slow rate.
Car	Primary	A car that: 1. is currently driving on the freeway OR 2. wants to merge into the freeway through the ramp managed by the *System* A *Car* can trigger sensors, resulting in notifications to the *System*.
Freeway Sensor (*FSensor*)	Facilitator	The sensor that detects and notifies the *System* of *Car* traffic on the freeway. It is either **ON**, traffic is detected on the freeway, or **OFF**, no traffic detected.
End of Queue Sensor (*QSensor*)	Facilitator	The sensor that detects and notifies the *System* of intense ramp *Car* traffic, meaning traffic extends past the entrance of the on-ramp. It is either **ON**, traffic is detected, or **OFF**, no traffic detected. However, every car that enters the ramp triggers this sensor, meaning only a persistent **ON** state is indicative of intense traffic.
Demand Sensor (*DSensor*)	Facilitator	The sensor that detects and notifies the *System* of a *Car* being detected in front of the traffic light/ramp's stop bar, waiting to merge into the freeway. It is either **ON**, traffic is detected, or **OFF**, no traffic detected.
Passage Sensor (*PSensor*)	Facilitator	The sensor that detects and notifies the *System* of a *Car* that just passed the traffic light and is ready to effectively merge into the freeway. It is either **ON**, traffic is detected, or **OFF**, no traffic detected.

Traffic Light	Facilitator	The light that either allows or disallows a *Car* to transition from waiting to merge to being ready to effectively merge. It is either **RED**: *Car*s may not pass; or **GREEN**, *Car*s may pass and get ready to merge into the freeway.
Warning Light	Facilitator	The light that warns the *Car*s approaching the ramp whether ramp metering is active, indicating that there is currently traffic on the freeway.

1.2 Actor Overview

Actor	**Goals**
System	Manage ramp *Car* traffic to optimize freeway traffic flow.
Car	1. Pass by the ramp on the freeway efficiently 2. Merge into the freeway from the ramp
Freeway Sensor (*FSensor*)	Warn *System* of a *Car* approaching the ramp.
End of Queue Sensor (*QSensor*)	Warn *System* of heavy ramp *Car* traffic.
Demand Sensor (*DSensor*)	Warn *System* of a *Car* on the ramp waiting to merge into the freeway.
Passage Sensor (*PSensor*)	Warn *System* of a *Car* on the ramp leaving the *Demand Sensor*, passing the *Traffic Light* and thus transitioning from waiting to merge to being ready to effectively merge.
Traffic Light	Signal to a *Car* on the ramp waiting to merge whether they may transition to being ready to effectively merge.
Warning Light	Signal to *Car*s that ramp metering is active.

1.3 Use-Case Design Considerations

The system contains a singular primary actor: *Car*. Given the limited goals of *Car*, we only need to construct a single use-case on the user-goal level, namely for merging into the freeway. The reasoning for this is that a use-case by definition should pertain to a singular, complete interaction between user and system. It results in the meaningful fulfillment of one or more of the user's goals.

This use-case will satisfy requirement 5 and requirement 6. As they cover very specific sub-functions, we will represent these requirements as extensions to the use-case.

As a brief refresher, the full textual requirements can be found below.

> *system requirement 5 – When the freeway sensor is **ON** and the end of queue sensor is **ON**, if demand sensor is turned **ON**, it indicates intense traffic over the ramp. In this scenario, two cars must cross the traffic light before it turns **RED**. Then, set traffic light to **GREEN**. After demand sensor is **OFF**, **ON** and **OFF** (two cars), turn traffic light to red.*

> *system requirement 6 – The passage sensor can only be turned **ON** once a car leaves the demand sensor shifting from **ON** to **OFF**. Another car can only turn the passage sensor **ON** once it has been turned **OFF** (the previous car joined the mainline).*

1.4 Merging *Car* use-case

The user, a *Car* actor, only has a single goal for interacting with the *System*: merging into the freeway. Thus, a single use-case encompassing this entire interaction is defined below. It starts with the *Car* entering the *QSensor* and ends when it exits the *PSensor* in order to merge.

It should come as no surprise that all of the other actors described previously appear as facilitator actors here.

Not all requirements are incorporated into the use-case; only requirements 5 and 6 had to be considered for this assignment. We also assume that the *Car* going through the main success scenario is at the head of the queue of all *Car*s on the ramp, meaning there is not un-merged *Car* ahead of the *Car* under scrutiny. In practice, this means that the sensors the *Car* encounters are typically in an **OFF** state before the *Car* interacts with them.

Extension 6a represents the triggering of requirement 5 while 8a governs its resolution and the *Traffic Light* turning **RED** appropriately.

Extension 9a together with step 9 represent the section in requirement 6 that says *PSensor* only has space for a single *Car* at a time. Step 7 should ensure that only if a *Car* has passed by the traffic light, will *DSensor* **OFF** occur and will the *Car* have access to *PSensor* at all.

USE CASE: Merge onto freeway (USER-GOAL)

ENVIRONMENT

Scope: Ramp Metering *System*
Level: User-goal
Intention in context: A *Car* intends to merge onto the freeway using the ramp.
Multiplicity: Multiple *Car*s may take the ramp simultaneously in the form of a single file queue.

ACTORS

Primary Actor: *Car*
Secondary Actor: None
Facilitator Actor: *System*, *FSensor*, *QSensor*, DSensor, *Traffic Light*, *PSensor*

SCENARIO

Main Success Scenario:

1. *Car* triggers the *QSensor* (*QSensor* turns **ON**), notifying *System* of possibly intense ramp traffic.
2. *System* takes the continuous **ON** signal into account, and if deemed necessary, switch to a faster flowthrough rate.
3. *Car* leaves *QSensor* and keeps queueing (*QSensor* turns **OFF**), notifying *System* of possible resolution of intense on-ramp traffic.
4. *System* takes the continuous **OFF** signal into account, and if necessary, enables a slower flowthrough rate.
5. *Car* triggers DSensor (DSensor turns **ON**), notifying *System* of demand to merge into freeway.
6. If *FSensor* is **OFF**, THEN *Traffic Light* is **GREEN** and nothing needs to be done.
7. WHEN *Traffic Light* is **GREEN** THEN *Car* leaves DSensor (DSensor turns **OFF**), notifying *System* of the car passing *Traffic Light* and queueing to merge into the freeway. ELSE retry step 7.
8. *System* takes a continuous *DSensor* **OFF** signal into account. IF the *System* determines that there should be a slow flowthrough rate, THEN turn *Traffic Light* **RED**.
9. *Car* moves onto *PSensor* (*PSensor* turns **ON**), blocking *PSensor* from being turned **ON** by another *Car*, and later moves past *PSensor* (*PSensor* turns **OFF**), notifying the *System* that it is effectively merging into freeway. *PSensor* may now be turned **ON** again.

Extensions:

6a. DSensor is **ON**. If both *FSensor* and *QSensor* are **ON**, then *System* detects freeway traffic and heavy on-ramp traffic.
  6a-1. *System* sets *Traffic Light* to **GREEN** and sets *System* state to fast flowthrough rate. The use-case continues at step 7.

8a. If *System* state is in a fast flowthrough rate, then two cars need to pass the *Traffic Light* before its state can be changed.
  8a-1a. IF *System* detects that DSensor has been set to **OFF** twice since *Traffic Light* turned **GREEN**, THEN turn *Traffic Light* **RED**.
  8a-1b. ELSE do nothing.
  8a-2. The use-case continues at step 9.

9a. If *PSensor* is **ON**, then another *Car* is already merging.
  9a-1. The use-case continues at step 9. (retry)

2 UML Class Diagrams

We first discuss the features shared by both diagrams:

- A single **Controller** instance
  - The ***flow_through_rate* member** represents the metering rate and is initialized to SLOW. It is manipulated through its private getter and setter.
  - The **Rate enum** represents possible metering rates: SLOW (req. 4) and FAST (req. 5).
  - The ***demand_passed_ctr* member** keeps track of the number of cars that have passed *Traffic Light* during FAST metering.
  - The ***detected(Sensor,* Car*)* and *departed(Sensor,* Car*)* methods** allow a Sensor to report to Controller when a *Car* respectively enters or leaves that Sensor.
- A single ***Traffic Light*** instance
  - The ***state* member** represents the light's colour. It is initialized to **GREEN**. It is manipulated through its getter and setter.
  - The **Color enum** represents possible *Traffic Light* colors.
  - The **Controller** calls both getter and setter.
  - A ***Car*** only views the *Traffic Light* state, albeit indirectly.
- ***Warning Light*** class structure is completely analogous to that of *Traffic Light*
  - Its ***state* member** makes use of the State enum instead of the Color enum.
- Zero or more ***Car*** instances
  - The ***id* member** represents the identifier of the ***Car*** instance, for use by the *DSensor* and the *PSensor*. It is accessed through its getter.

Figure 1 features the more general diagram of the two. It combines all sensor functionality into a single Sensor class. Sensor is structured as follows:

1. Differentiate between Sensor instances through the *name* member
2. The Controller interacts with the *state* member through the getter and setter
3. The *enter(*Car*)* and *exit(*Car*)* methods are called when a *Car* triggers the Sensor.
   These methods should call the Controller's *detected(Sensor,* Car*)* and *departed(Sensor, Sensor)* methods respectively.

The idea is that all Sensor instances are able to identify any *Car* instance that either enters or exits it. Every Sensor can then pass this *Car* along to the Controller through its *detected(Sensor,* Car*)* and *departed(Sensor,* Car*)* methods. But, the Controller will only actually use the passed *Car* in the case of *DSensor* and *PSensor*, while the parameter is ignored for *QSensor* and *FSensor*. Additionally, the multiplicity on the arcs connecting Sensor and *Car* are not entirely correct; a *DSensor* can only be occupied by one *Car* instance for example. These inaccuracies are a trade-off for the diagram imposing very few implementation restrictions by making the Sensor implementation quite open.

Figure 2 depicts a more explicit variation, that imposes structural implementation restrictions on the Sensor functionality. In return, the diagram more accurately describes the system requirements.

The *state* member, its getter and setter methods and the *name* member are all abstracted into a Sensor interface as common functionality. The distinction between a Car*Sensor* and *TrafficSensor* gets made. The former is capable of identifying which specific *Car* instance enters or exits it. Which is why its *enter(*Car*)* and *exit(*Car*)* method signatures include a *Car* parameter. The latter only knows that some *Car* entered or exited, but is uninterested in which *Car*. Thus, its *enter()* and *exit()* method signatures include no parameters. The latter type of Sensor would call the Controller's *detected(Sensor,* Car*)* and *departed(Sensor,* Car*)* methods with some kind of NULL value for the *Car* parameter.

The distinction between Car*Sensor* and *TrafficSensor* is derived from the assignment. It describes *DSensor* and *PSensor* as being capable of handling a single, specific car at a time. While *FSensor* and *QSensor* instead take note of the presence of any traffic at all.

This difference is reflected in the different multiplicities on the *Car* side of the associations between *Car* and the derived Sensor classes. A Car*Sensor* may interact with 0…1 *Cars*, as opposed to the 0… ∗ *Cars* that interact with a *TrafficSensor*. Also take note of the associations between *Car* and the Sensor interface. Their multiplicities ensure that a *Car* only has a single, unique Sensor that it is associated with at a time, and not with both a Car*Sensor* and *TrafficSensor* at the same moment.

Links to the class diagrams are also hosted at Google Drive, in case the attached images prove to be illegible: **split sensor diagram**, **combined sensor diagram**. We include two versions of the class diagram, which only differ in the design of the Sensor class.

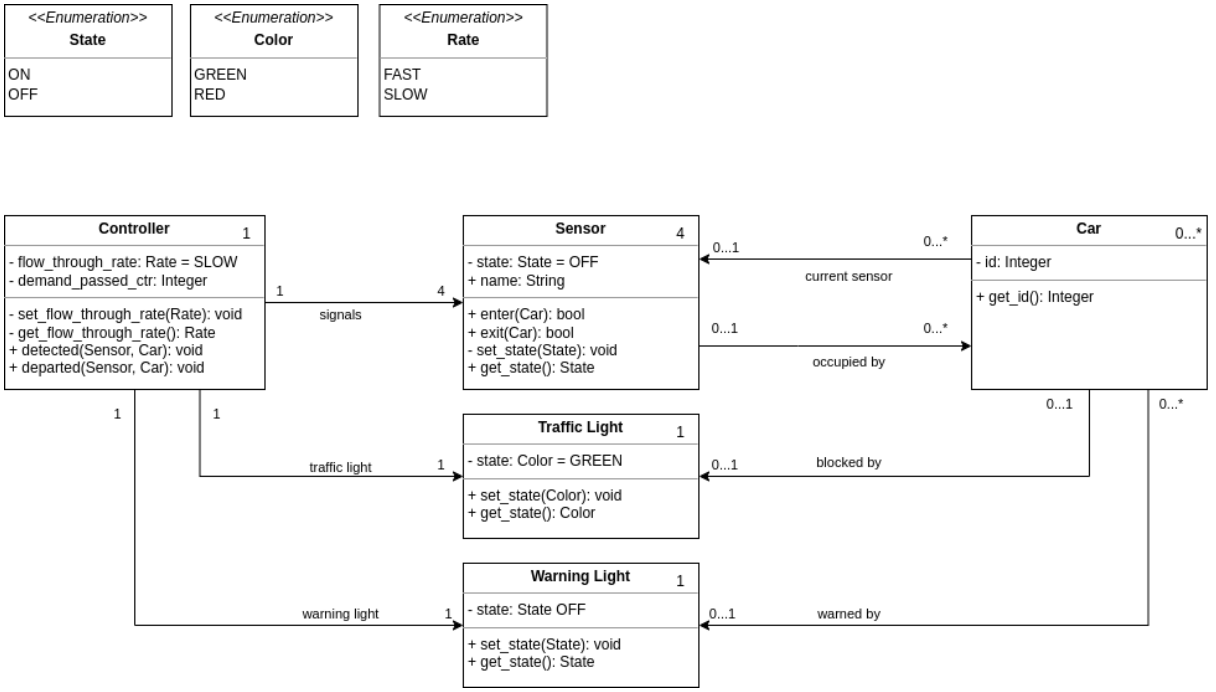

Figure 1: Combined Sensor class diagram

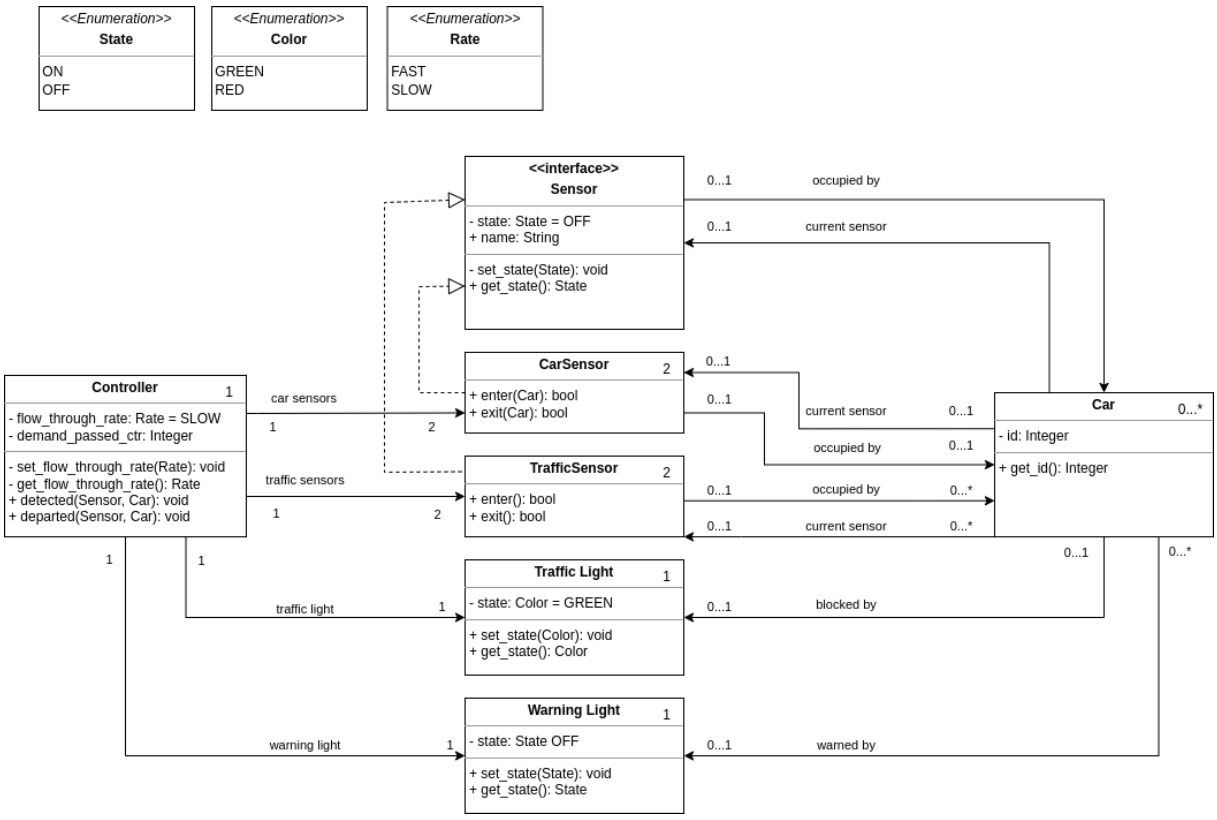

Figure 2: Split Sensor class diagram

3 UML Sequence Diagrams

We will make one impactful assumption: *Cars* are able to infer the *Traffic Light* state without exchanging any messages with it in the sequence diagram. This reflects the indirect nature in which a *Car* entity and the *Traffic Light* entity interact in a real scenario (i.e. via optical/visual communication). This results in a *Car* implicitly being able to infer the *Traffic Light* state by having the sequence diagram implement conditional arcs or self-loops, both of which require the light to be green.

3.1 Requirement 5

The assumed pre-conditions are:

1. *FSensor* **ON**
2. *QSensor* **ON**

Refer to Figure 3 for the sequence diagram. The main steps of the diagram go as follows:

1. *Car* x enters *DSensor* and immediately enters a self-loop, awaiting *Traffic Light* **GREEN**, blocking access by other *Cars* to *DSensor*
2. *DSensor* reports detection of *Car* x to Controller, which enters a *fast-flowthrough state* and requests *Traffic Light* to turn **GREEN**
3. *Car* x detects the light turning **GREEN** and leaves *DSensor*, allowing access by others to *DSensor* again
4. *DSensor* reports the departure of *Car* x to Controller, which maintains its state
5. Similar to step 1. but for *Car* y
6. Similar to step 2. but for *Car* y and Controller maintains its state
7. Similar to step 3. but for *Car* y
8. *DSensor* reports the departure of *Car* y to Controller, which then requests *Traffic Light* to turn **RED** *before* another *Car* enters *DSensor*

An optional case was depicted in the diagram, taking place during step 6 defined above. Suppose *QSensor* turns **OFF** after *Car* x enters *DSensor*. Then requirement 5 has already been triggered so that Controller will still enter the *fast-flowthrough state*. However, the pre-conditions for requirement 4 are now satisfied. This means that *Car* y entering *DSensor* provides the required trigger for Controller to transition from *fast-flowthrough* to *slow-flowthrough*. The only difference this makes, is that *Traffic Light* is redundantly turned **GREEN** again after *Car* y enters *DSensor*. *QSensor* turning **OFF** or **ON** after *Car* y entering *DSensor* does not influence this requirement.

3.2 Requirement 6

The assumed pre-conditions are:

1. *Traffic Light* **GREEN**
2. *PSensor* **OFF**
3. *Car* y is behind *Car* x in the ramp queue

Refer to Figure 4 for the sequence diagram. Note that while this diagram seems quite complex at first blush, its main series of steps are actually quite simple.

1. *Car* x leaves *DSensor* when *Traffic Light* turns **GREEN**, which we assume to be the *Traffic Light* state as a precondition, allowing access to the *DSensor* by other *Cars*
2. *Car* x enters *PSensor*, blocking access of other *Cars* to *PSensor*. It then leaves *PSensor* at a later moment and merges into the freeway, allowing access to the *PSensor* again by other *Cars*
3. *Car* y enters *DSensor*, blocking access of other *Cars* to *DSensor*. It then leaves *DSensor* when *Traffic Light* **GREEN** at a later moment, allowing access to the *DSensor* again by other *Cars*
4. *Car* y enters *PSensor*, blocking access of other *Cars* to *PSensor*. It then leaves *PSensor* at a later moment and merges into the freeway, allowing access to the *PSensor* again by other *Cars*

Notice that any of the steps of *Car* x interacting with *PSensor* are independent of any of the steps of *Car* y interacting with *DSensor*. This is the main source of the diagram's complexity; all of the different message order permutations of the individual messages for steps 3 and 4 are depicted in the *alt* blocks of the diagram. The options in the outer *alt* block dictate when *Car* y enters *DSensor* and the options in the inner *alt* blocks specify when *Car* y leaves *DSensor*, in relation to the messages concerning *Car* x.

To simplify the diagram, we left out possible extensions of these scenarios. Their descriptions are given below.

First, we assumed *Traffic Light* **GREEN** as a precondition. This is because a *Car* can only cross from *DSensor* to *PSensor* if the light is **GREEN**. At any point in the sequence diagram the light could turn **RED** due to, for example, *FSensor* **OFF** occurring. This would only delay the *Car* currently on *DSensor* from leaving it, until *Traffic Light* **GREEN** happened again. So we could extend the diagram by:

1. Turning *Traffic Light* **RED** and later **GREEN** again before *Car* x enters *DSensor*, idem for *Car* y. In other cases, turning the light **RED** has no effect on this requirement.
2. Additionally, each *Car* would need to await *Traffic Light* **GREEN** when it is **RED**. Two implementations of waiting behavior come to mind: a self-loop from and to *Car* until a green light occurs OR a conditional arc from *Car* to *DSensor* that requires the light to be **GREEN**.
3. A *Car* entering *DSensor* may trigger requirements 4 or 5, so the parts of the diagram for reqs. 4 and 5 that serve to showcase the interactions between *DSensor* and *PSensor* could be merged into the diagram for req. 6; iff the preconditions for reqs. 4 or 5 were satisfied accordingly

Second, we assumed that the *PSensor* is **OFF**. This allows *Car* x to enter *PSensor* immediately.
The diagram could include:

1. Some *Car* z leaving *PSensor*, and only after which that happens, could *Car* x enter *PSensor*. Optionally added is *Car* z entering *PSensor*, which would also have to occur before *Car* x entered *PSensor* in order for the requirements to hold.
2. A conditional on the arcs from *Car* to *PSensor* that check whether *PSensor* is empty when the *Car* is intending to enter, or a self-loop to await *PSensor* **OFF**. Both of these cases are similar to the ones described for the *Traffic Light* options.

As before, links to the sequence diagrams are also available at websequencediagrams, in case the attached images are illegible: **requirement 5** and **requirement 6**.

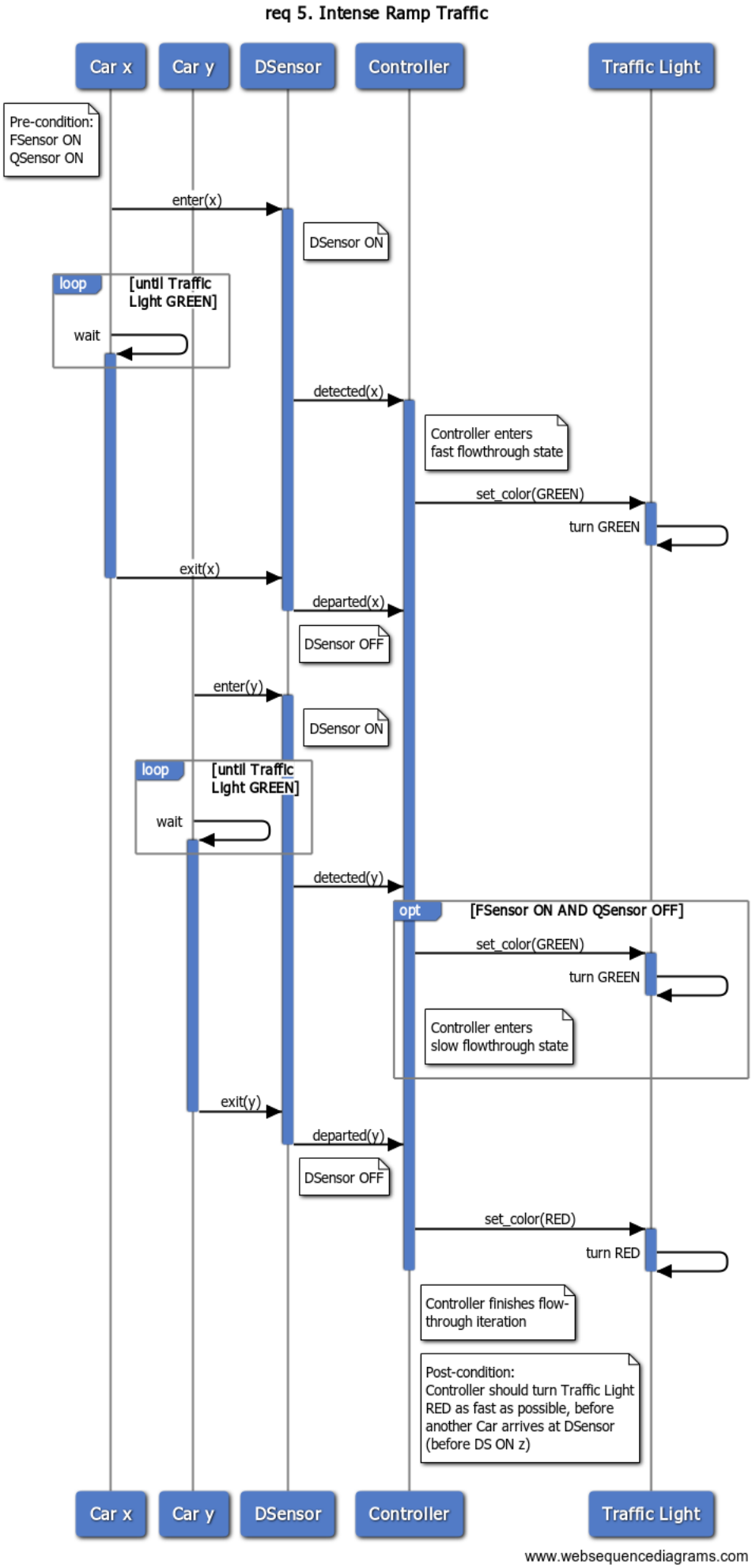

Figure 3: requirement 5. full sequence diagram

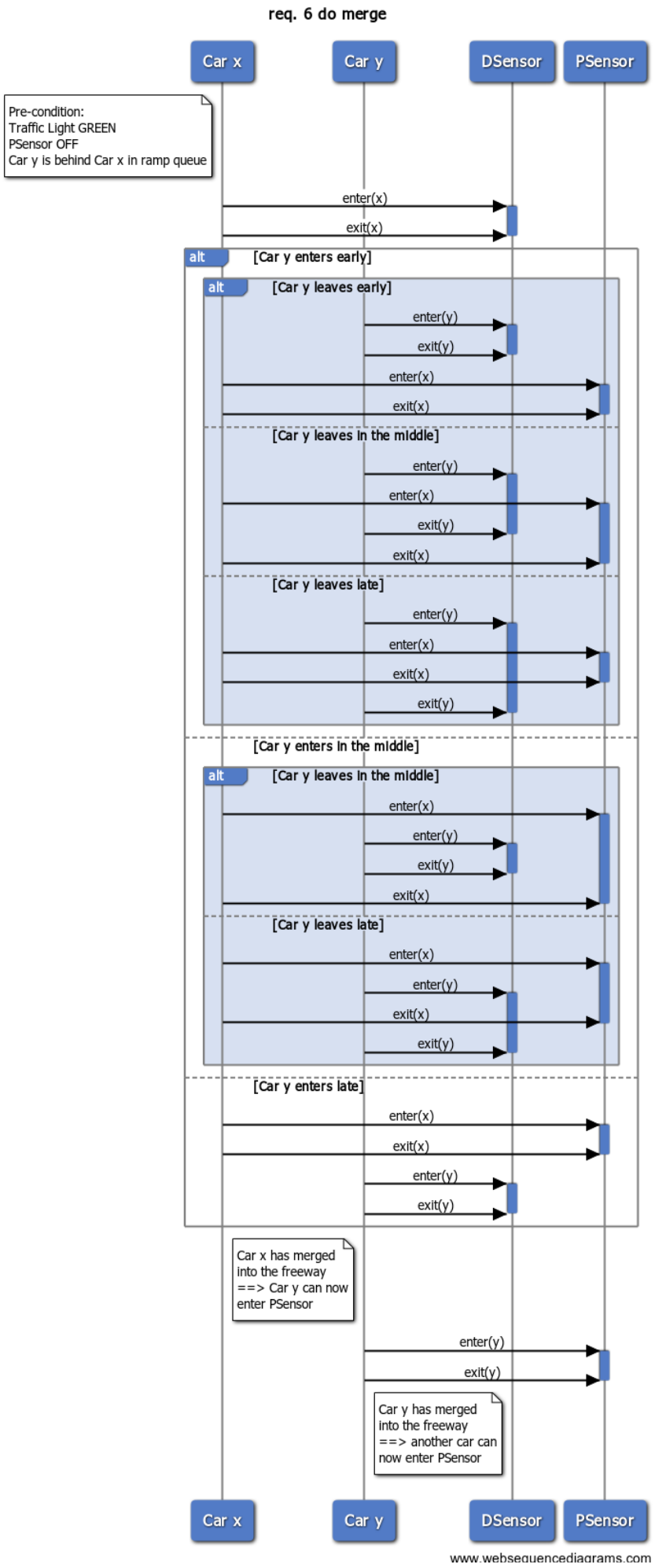

Figure 4: requirement 6. full sequence diagram

4 Regular Expressions

4.1 Assumptions, Observations and Construction

First, we will state two important assumptions about requirement five's wording. It states:

> "*When* FSensor *is* **ON** *and the* QSensor *is* **ON**, *if* DSensor *is turned* **ON** ..."

We will interpret this requirement to be composed of a precondition, a trigger event and then the subsequent actions to be performed. The precondition consists of the *FSensor* and *QSensor* both being in the **ON** state. The trigger event can be considered as the *DSensor* being set to **ON**. The actions are understood as the behavior related to faster flowthrough rate. Grossly stated – as soon as the preconditions are fulfilled and the trigger gets executed – no matter how the precondition's values change, the action of letting two cars pass will be executed regardless.

To elaborate on above statement, turning **OFF** the *QSensor* before a second car triggers the *DSensor* will trigger the execution of the logic associated with requirement 4. Since *FSensor* **ON** and *QSensor* **OFF** constitute the preconditions for requirement 4, and *DSensor* **ON** the trigger, the requirement gets executed. Note that this neither violates nor contradicts the requirements stated in requirement 5, as exactly two *Cars* will still pass before the *Traffic Light* is set back to **RED** upon the passage of the second car.

On the other hand, if the *FSensor* turns off during the execution of the requirement, we fall back to requirement 2. As requirement 2 can be understood as having a higher priority than 5, we will not consider this as a breach of the requirement.

For both regexes, following constructs will be utilised for codeblocks 1 and 2:

- Flag `\s` (Single Line/Match All): The 'dot' metacharacter also matches newline characters `\n`, this simplifies the expression ever so slightly
- Subpattern `(?:xyz)` (Non-capturing Groups): Ensuring that the matched groups do not get memorized, which keeps the back-references order clean
- Construct `(?:(?!xyz).)*` (Negative Lookahead): A pattern that matches any character `.*`, except if it will match `xyz`

4.2 Requirement 5

For this requirement we will make use of Positive Matching, more specifically: a match is only given iff *exactly* two cars pass whenever all preconditions are fulfilled for ramp passthrough under intensive traffic conditions.

The regex we constructed for encoding these conditions, can be found below in Codeblock 1.

```
(?:
    QS ON\n(?:(?!QS OFF\n).)*
    FS ON\n(?:(?!FS OFF\n|QS OFF\n).)*
  |
    FS ON\n(?:(?!FS OFF\n).)*
    QS ON\n(?:(?!FS OFF\n|QS OFF\n).)*
)
DS ON \d+\n(?:(?!FS OFF\n|QS OFF\n).)*
TL GREEN
  (?:(?!TL RED|DS OFF \d+\n|FS OFF\n).)*
  DS OFF \d+\n(?:(?!TL RED|DS OFF \d+\n|FS OFF\n).)*
  DS OFF \d+\n(?:(?!TL RED|DS OFF \d+\n|FS OFF\n).)*
TL RED
```

Codeblock 1: Regular Expression for Requirement 5

We encoded following design considerations within the regex:

- `QS ON` and `FS ON` may appear in any order
- If the *QSensor* or *FSensor* get turned **OFF** at any point when checking the preconditions for requirement 6, move on to the next possible match (i.e. `QS ON\n(?:(?!QS OFF\n).)*`)
- Additionally, if the *FSensor* gets turned **OFF** while we're in the "Heavy On-Ramp Traffic" scenario, then we fall back to requirement 2 (and thus move on to the next possible match)
- As we're positively matching, we construe the requirement as:

  'iff. the preconditions were fulfilled ⟹ *exactly* two cars must pass'

  In other words: if at any point during the regex matching operation, an unexpected `TL RED\n` or `DS OFF \d+\n` was encountered, we will *not* match the string. This concept will be further clarified with the State Automata.

4.3 Requirement 6

This requirement is encoded into a regex using Negative Matching: whenever a match is found, we consider this match as a violation of the requirements. In other words, if there is ever a case of the *PSensor* turning **ON** for a specific car has driven **OFF** the *DSensor or* if the *PSensor* is turned **ON** twice – without turning **OFF** anywhere in between, then we consider this as our requirement being broken.

Thus, our regex is constructed as such:

```
(?:
  DS ON (\d+)\n
  (?:(?!DS OFF \1\n).)*
  PS ON \1\n
|
  PS ON (\d+)\n
  (?:(?!PS OFF \2\n).)*
  PS ON \d+\n
)
```

Codeblock 2: Regular Expression expressing Requirement 6

Note the usage of the OR to combine our two negative substatements. *Any* match will indicate that the requirement was violated. Let us describe the regular expression in a bit more detail:

1. If `PS ON \1\n` was matched (i.e. car 'x' moving **ON** the *PSensor*) and was not preceded by `DS OFF \1\n` (car 'x' driving **OFF** the *DSensor*), *then* match the expression
2. If `PS ON \d+\n` was matched (i.e. car 'y' moving **ON** the *PSensor*), and was not preceded by `PS OFF \2\n` (car 'x' driving **OFF** the *PSensor*), *then* match the expression

5 Finite State Automata

5.1 Construction

As we're dealing with reasonably long strings that need to be matched, it becomes necessary to find a construction that briefly describes multi-character transitions between two states.

This construction can be found in Figure 5, which indicates how we might convert arbitrary-length transitions into a proper state-machine with single character transitions.

Some additional nomenclature: 'Σ' indicates the entire set of characters that the FSA is able to process, and ⟶ denotes the starting state of our FSA.

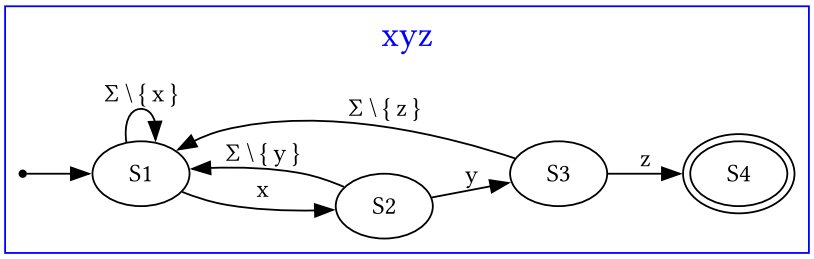

Figure 5: Multi-string transition construction

Practically speaking, each character will be matched sequentically, if there is any point where a character cannot be matched, the state machine will revert to the initial state S1. Reference FS constructions for kleene star or positive closure operators can be found below in Figures 6 and 7.

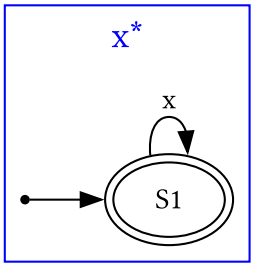

Figure 6: Kleene star operator

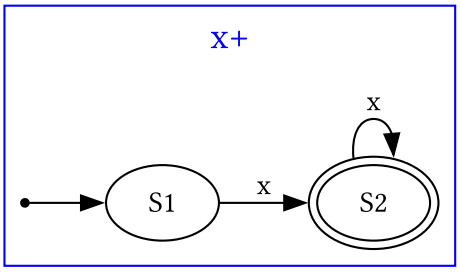

Figure 7: Positive closure operator

Further, for the sake of cleanliness, `ε` may be used as spontaneous transitions. This can be used for grouping together edges to a particular state, in order to achieve a more readable graph.

And finally, in another effort to write the graphs more cleanly, subgraphs will be used (denoted by dotted lines), where edges starting from the border of a subgraph `A` going to a node `x` indicate that it is a shorthand for every node `a` of subgraph `A` having an edge to `x` for that single character. If the edge goes from subgraph `A` to `A` (i.e. a loop), then it may be understood as each node `a` in `A` having a transition to itself given a character.

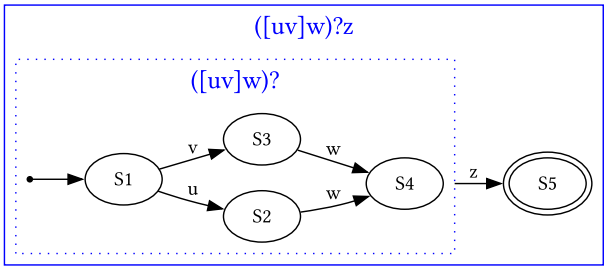

Figure 8: Abbreviated graph

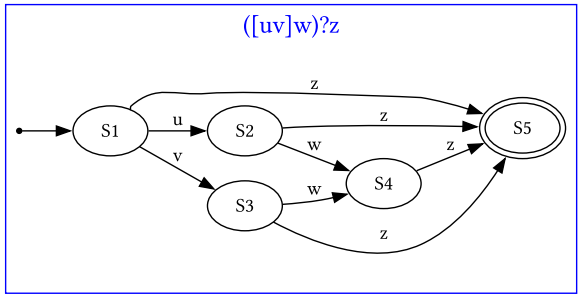

Figure 9: Full graph

5.2 Requirement 5

With these general constructions stated, we can convert the regular expression stated in Section 4.2 into the state machine found in Figure 10. Since we've written a positive matcher, the state machine will match true iff. we never landed in the Error state (every state other than Error is accepting).

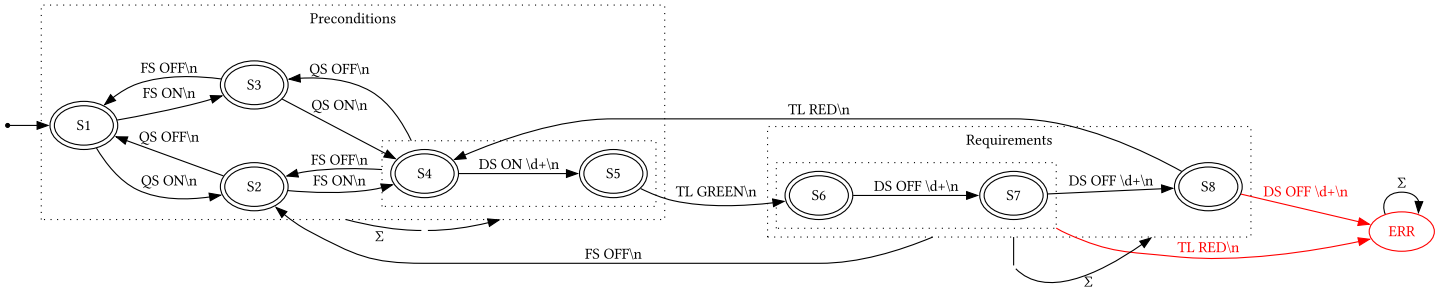

Figure 10: Intensive On-Ramp Traffic FSA (Requirement 5)

The FSA will first check if all the preconditions are **ON**, and once that is the case (meaning the FSA is in state S5), makes sure that none of the conditions turn **OFF** after that. The moment the *Traffic Light* turns **GREEN**, we only need to pay attention to the *FSensor* turning **OFF** (meaning that all restrictions for the amount of cars passing through may be ignored). If at any moment we detect an unexpected *Traffic Light* turning **RED** or *DSensor* turning **OFF**, then then we go to the error state and stay there indefinitely, these transitions are drawn in **RED**.

5.3 Requirement 6

Figure 11 shows how the reasonably simple statement posited in Section 4.3 may be converted. As we are not able to match the two different conditions concurrently without resorting to non-determinism, it was necessary to expand the automaton to consider all possible car states (for pairs of cars), such that we can examine in detail which specific combinations can result in error states.

As this graph only checks for four possible stack trace values, and the graph is already quite large as-is, we make use of an abbreviated notation to describe our transitions:

	`DS && \d+\n`	`PS && \d+\n`
ON	DON	PON
OFF	DOF	POF

Another change that was made is the usage of the `----` line to the Error state. This denotates the fact that *each* state in the subgraph has a connection to the Error state for any of the provided values, iff. the state does not already have a transition for said value(s). For example: `DOF TON PON` has transitions for both `DON` and `POF` in the graph, but none for `DOF` and `PON` — if the FSA encounters these strings in the trace while in the state `DOF TON PON`, then it will transition to the Error state.

And two last notes: (1) we make use of the concept of 'Transit' (abbr. `T`) to indicate that a car is in-between the *DSensor* and *PSensor*; (2) due to the fact that we're using a FSA, we cannot trivially memorize the amount of cars in transit between the two sensors. If the front car in transit arrives at the *PSensor*, then Transit is considered to be **OFF** (TOF).

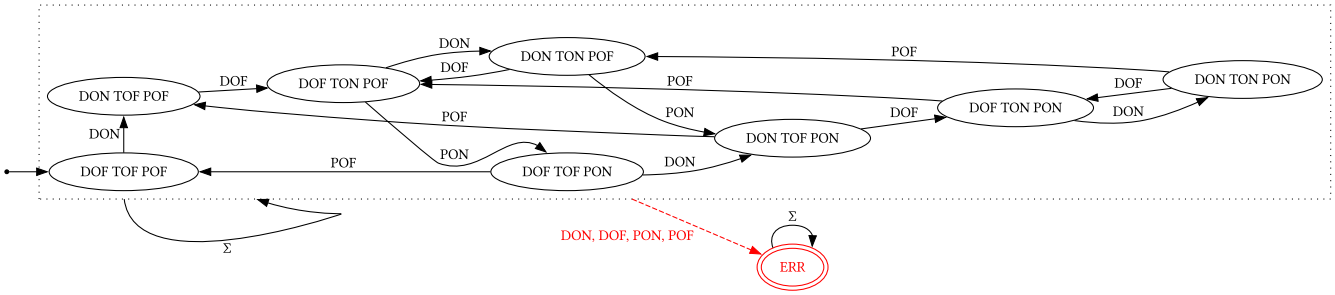

Figure 11: Demand/Passage Sensor Sequence Enforcement (Requirement 6)

6 FSA Implementation

The translation from the automata specified above to Python code was luckily rather straight-forward. The only issue that had to be tackled was implementing a generic way for the automaton to perform multi-character transitions, without having unreasonably long code files.

This problem is solved by making use of 'substates', where the states act as memory for how much of the multi-character input has been matched, and which character is expected next. A special encoding had to be used for `\d+` to simplify the character matching algorithm.

Once the automaton arrives at the final substate and the correct character is matched, we follow the transition to the originally intended destination (i.e. one that is not a substate).

7 Violation of Specification

With the scanners implemented, we could quickly verify that the conditions stated in requirement 6 were frequently being broken within the provided traces. Requirement 5 was always fulfilled for all of the traces, and no easily identifiable bugs were discovered upon manual inspection. The only potential bug for this requirement came in the form of the double `TL GREEN\n`, but as it does not violate any of the stipulated provisions, we made the assumption that **it is not the bug that we're looking for**.

After inspecting the dynamics of requirement 6's FSA, it seems to encounter an unmatched `PS ON \d+\n`, which results in a transition into the Error state. Upon further inspection, it occurs in the following traces:

- Trace 3: lines 11 - 20, lines 58 - 63
- Trace 5: lines 59 - 62
- Trace 6: lines 11 - 20

```
11 PS ON 2
12 FS ON
13 TL RED
14 WS ON
15 DS ON 3
16 TL GREEN
17 DS OFF 3
18 TL RED
19 DS ON 4
20 PS ON 3
```

Codeblock 3: Trace 3/6
`PS OFF 2` *signal not received*

```
58 PS ON 10
59 DS ON 11
60 TL GREEN
61 DS OFF 11
62 TL RED
63 PS ON 11
64 PS OFF 10
```

Codeblock 4: Trace 3
`PS OFF 10` *received out of order*

```
58 TL GREEN
59 PS ON 9
60 DS OFF 10
61 DS ON 11
62 PS ON 10
63 DS OFF 11
64 TL RED
65 PS OFF 10
66 PS ON 11
67 PS OFF 11
```

Codeblock 5: Trace 5
`PS OFF 9` *signal not received*

As the circumstances for the bug differ between the different traces, it proves to be difficult to state with definitive certainty *what* precisely is going wrong. In Codeblock 4, for instance, we see that the *PSensor* is turned ON with car 11 before previous car 10 has gotten the chance to drive off the sensor.

On the other hand, Codeblocks 3 and 5 show how occasionally, the line `PS OFF \d+` is outright missing. There doesn't seem to be any rhyme or reason for this odd behaviour, as all three circumstances are entirely different. Sometimes the *FSensor* is OFF when these bugs occur, other times, it happens in the midst of the scenario stated by requirement 4, etc.

If anything, one might conclude that the *PSensor* is faulty in some way. Either having a significant delay in processing and sending its signals, or failing to send the signal altogether.