Requirements Checking Assignment

Practical Information

- **Due Date:** Friday 27 October 2023, before 23:59.
- **Team Size:** 2 (pair design/programming)!
  Note that as of the 2017-2018 Academic Year, each International student should team up with "local" (i.e., whose Bachelor degree was obtained at the University of Antwerp).
- **Submission Information:** *Only one* member of each team submits a full solution. This must be a compressed archive (`ZIP`, `RAR`, `TAR.GZ`...) that includes your report and all models, images, code and other sources that you have used to crate your solution. **Do not submit any executables.** *The report may be either HTML or PDF and must be accompagnied by all images. When an image is unreadable in your report* ***and*** *missing from your submission archive, you will not receive any points for that task.* Make sure to mention the names and student IDs of both team members. The other team member *must* submit a single `HTML` file containing *only* the coordinates of both team members. You may use this template. This will allow us to put in grades for both team members in BlackBoard.
- **Submission Medium:** BlackBoard Ultra. Beware that BlackBoard's clock may differ slightly from yours. If BlackBoard is not reachable due to an (unpredicted) maintenance, you submit your solution via e-mail to the TA. Make sure all group members are in CC!
- **Contact / TA:** Lucas Lima.
- BlackBoard recording of the assignment explanation session

Goals

In this assignment, you will use the *Use Cases*, *UML Sequence Diagrams*, *Regular Expressions*, and *State Automata* modelling languages to design and verify the communication in a ramp metering system. You will also make the link to an output trace an actual implementation in Python to perform the checking. In the end, you will be able to automatically determine, based on the obtained trace file, whether all requirements are satisfied or not. Figure 1 shows an overview of the parts in this assignment.

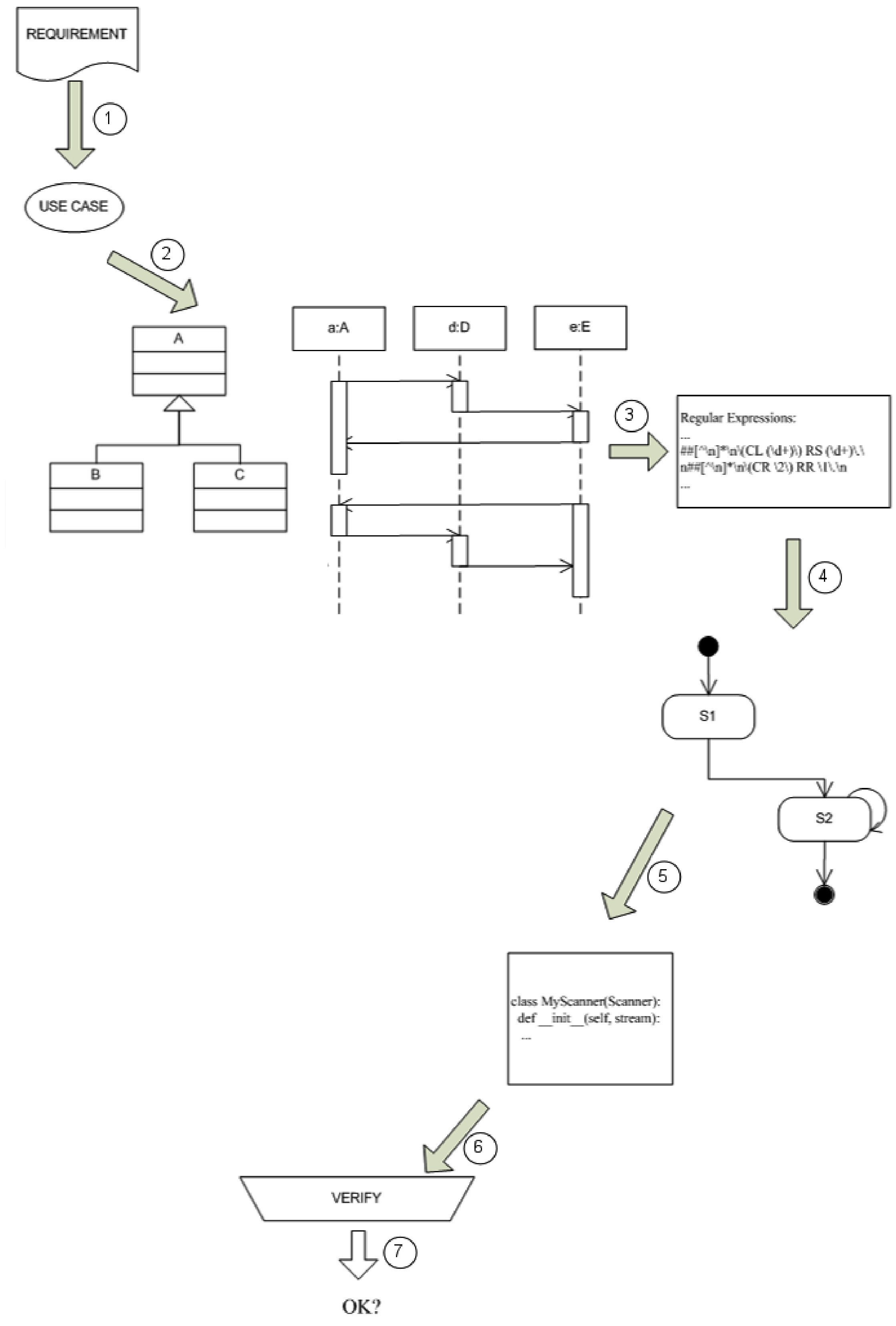
REQUIREMENT
1
USE CASE
2
A
B
C
a:A
d:D
e:E
3
Regular Expressions:
...
##[^\n]*\n\(CL (\d+)\) RS (\d+)\.\
n##[^\n]*\n\(CR \2\) RR \1\.\n
...
4
S1
S2
5
class MyScanner(Scanner):
def __init__(self, stream):
...
6
VERIFY
7
OK?

Figure 1: Main overview of the assignment.

Problem Statement

Due to the increase in urban areas, practical freeway management tools are crucial. Ramp metering is a potential tool for addressing congestion and safety issues. Despite initial opposition and scepticism from various stakeholders, ramp metering has been deployed, sustained, and expanded in many regions.

What is a Ramp Metering?

A ramp metering system is a traffic management strategy used on highways/freeways to regulate the flow of traffic entering the mainline from entrance ramps. The primary goal of ramp metering is to optimize traffic flow, reduce congestion, and enhance overall road efficiency. Ramp metering smooths out the flow of traffic and enhances safety by balancing conflicting traffic demands. Smoother traffic flow increases mobility and reduces the potential for accidents.

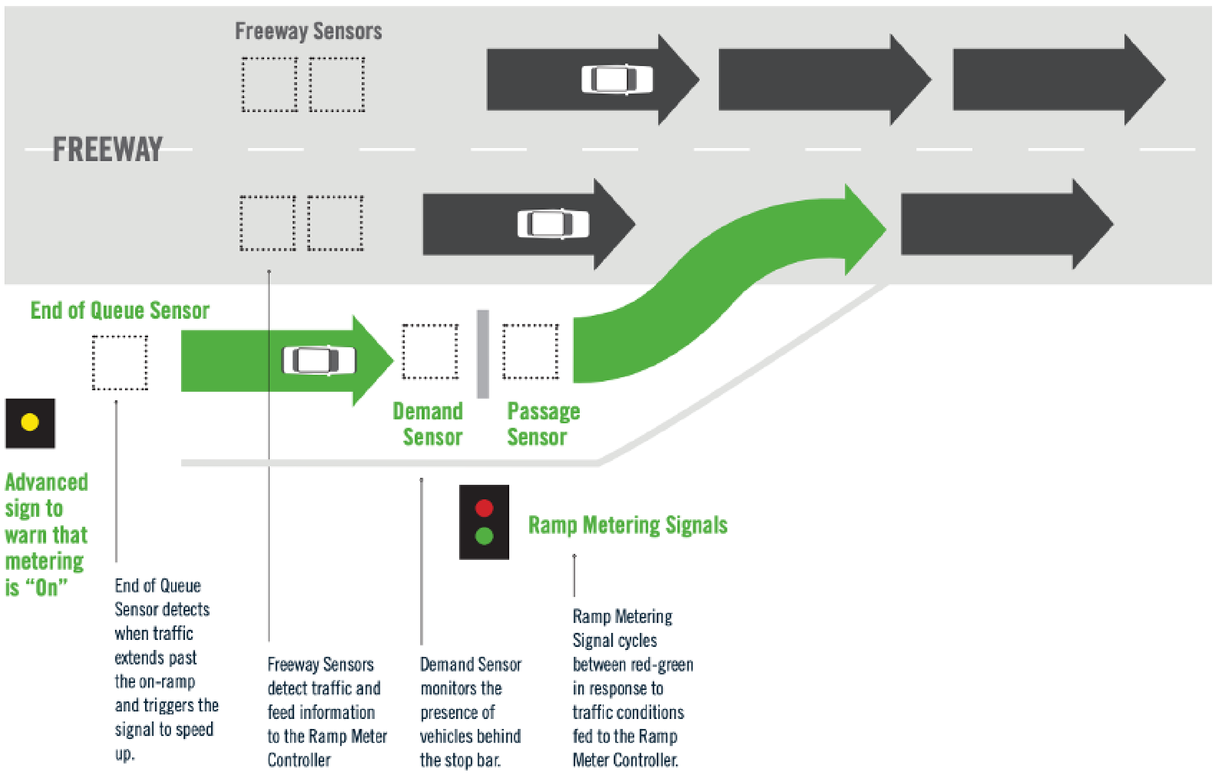
Freeway Sensors
FREEWAY
End of Queue Sensor
Demand Sensor
Passage Sensor
Advanced sign to warn that metering is "On"
Ramp Metering Signals
End of Queue Sensor detects when traffic extends past the on-ramp and triggers the signal to speed up.
Freeway Sensors detect traffic and feed information to the Ramp Meter Controller
Demand Sensor monitors the presence of vehicles behind the stop bar.
Ramp Metering Signal cycles between red-green in response to traffic conditions fed to the Ramp Meter Controller.

Figure 2: An example of the ramp metering system. Source: MTC

How does Ramp Metering work?

A ramp metering system is composed of the following components:

1. **Traffic Sensors**: The system relies on various sensors, such as loop detectors embedded in the roadway or cameras, to monitor the traffic conditions on both the entrance ramps and the mainline.
2. **Ramp Meter Controller**: It is a controller installed in a cabinet beside the road that receives data from the different sensors and uses an algorithm to process the real-time traffic data. It considers factors like traffic volume, speed, and density to determine the optimal rate at which vehicles should be allowed to enter the mainline. The metering rate is the frequency at which vehicles are allowed to merge onto the mainline. The control algorithm adjusts this rate dynamically based on current traffic conditions. During periods of high traffic, the metering rate may be reduced to prevent congestion on the mainline.
3. **Ramp Meter Signals**: At the end of the ramp, there is a traffic signal or meter that controls the release of vehicles onto the mainline. This signal is often located just before the merge point.

Some of the benefits of applying ramp metering are the help in preventing congestion and maintaining a more consistent flow of traffic on the mainline, and the system can prevent the mainline from becoming oversaturated, which could lead to stop-and-go traffic, by controlling the rate at which vehicles merge.

Many modern ramp metering systems use adaptive control strategies that take into account not only current conditions but also predicted traffic patterns. This allows for a more proactive approach to traffic management. Ramp metering systems are often part of larger Intelligent Transportation Systems that use advanced technologies to manage and optimize traffic flow across a network of roads.

For the sake of this assignment, we will consider a version of a ramp metering system composed of:

- Four sensors that can be either *ON* or *OFF*. For instance, if the freeway sensor is *ON*, then the system should consider there is traffic on the freeway, and when it is *OFF*, there is no traffic on the freeway. For the ramp sensors, *ON x* means that a camera detected a car *x*, while *OFF x* means that the car *x* has no longer been detected.
  - **Freeway sensor**: detects if there is traffic in the freeway;
  - **Demand sensor**: indicates that there is a car on the ramp wanting to join the freeway;
  - **Passage sensor**: detects that a car that was previously over the demand sensor has passed the ramp metering light and is ready to join the mainline;
  - **End of queue sensor**: detects that traffic on the ramp is intense and the system should increase the rate cars join the freeway;
- **Ramp traffic light**: a traffic signal with two possible states, *RED* and *GREEN*. *RED* indicates that a car on the ramp must not join the mainline, and green indicates the opposite.

- **Warning signal**: a signal that blinks if the ramp metering is active, otherwise it is turned off.
- **Ramp meter controller**: The equipment responsible for receiving data from the sensors and controlling the behaviour of the ramp traffic light and warning signal.

You are given a simple implementation of this system that handles the cars on the freeway and the ramp. It is your task to check whether or not the specified requirements are validated. However, you are only given access to trace files of the behaviour of this system (due to the execution of code). A trace file contains debugging information about the state of system components, such as the state of a sensor (ON or OFF), the traffic light current sign, and if the warning sign is blinking or not. Due to this verbosity and the total runtime of the execution, the file can be quite long, and therefore, validating the requirements is not to be done manually. Thus, the validation will 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 requirements for the system are the following:

1. The **freeway sensor**, the **end of queue sensor** and the **warning signal** start in the *OFF* mode and **traffic light** is GREEN;
2. When the **freeway sensor** is turned *OFF*, **traffic light** must be set to *GREEN* and **warning signal** set to *OFF* in any order before any other change in the system state;
3. If the **freeway sensor** is *ON*, **traffic light** must be set to *RED* and **warning signal** set to *ON* (blinking) in any order.
4. When the **freeway sensor** is *ON* and the **end of queue sensor** is *OFF*, if **demand sensor** is turned *ON*, it indicates that there is traffic on the mainline and there is a car on the ramp trying to join it, but the ramp traffic is not critical. Then, set **traffic light** to *GREEN*. When the car is crossing the traffic light (*GREEN*), the **demand sensor** must be set to *OFF*, and the **traffic light** must turn *RED* before **demand sensor** is turned *ON* again (another car arrives). When the car crosses the **traffic light** but has not joined the mainline yet, **passage sensor** is *ON*. When the car joins the mainline, the **passage sensor** is *OFF*.
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*.
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).
7. Whenever the **freeway sensor** is *ON*, the **warning signal** must be blinking, which indicates that the ramp meter control is active. It must be turned *OFF*, otherwise.

For simplicity of this assignment, you will **only need to check use cases 5 and 6**.

Note that the trace will only terminate as soon as all cars have arrived at the mainline. For example, if a car arrives at **demand sensor**, the trace will always contain the departure of the car in **passage sensor**. Furthermore, in each iteration, all cars will be handled in order. This means that if car 0 demands to join the mainline, the dispatch will happen before car 1 is handled. You can use this information to make some of the rules a little simpler. Finally, two cars cannot be in the same state. For instance, cars 4 and 5 either over or leaving the **demand sensor** (*ON* or *OFF*) is a situation that is not allowed.

Tasks

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

1. Write full *Use Cases* using the use case template (for ONLY requirements 5 and 6).
2. Design a class diagram for the system considering its relevant entities, relationships, attributes and operations.
3. Design the dynamic interaction behaviour in *UML Sequence Diagrams* for the above use cases (ONLY requirements 4 and 5), using the textual rendering tool PlantUML, or using the online tool WebSequenceDiagrams. You can start designing the class diagram without relationships and attributes, then design the sequence diagrams to realize the necessary operations and relationships between classes, and finally update the class diagram with the identified operations. **Describe all possible options.** Use variable names to reduce complexity. Clearly indicate the pre- and postconditions for your *Sequence Diagrams*.
4. Write *Regular Expressions* (*RegEx*) (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:

```
FS x    := Freeway sensor is set to x, which can be ON or OFF.
DS x id := Demand sensor is set to x, which can be ON or OFF, by car id.
PS x id := Passage sensor is set to x, which can be ON or OFF, by car id.
QS x    := End of queue sensor is set to x, which can be ON or OFF.
TL y    := Traffic Light is set to y, which can be GREEN and RED.
WS x    := Warning signal is set to x, which can be ON (blinking) or OFF.
```

The following is a complete example of a regular expression to check requirement 2:

```
FS OFF\n((?!TL GREEN\n)(.|\n))*FS ON\n|FS OFF\n((?!WS OFF\n)(.|\n))*FS ON\n
```

An explanation of this regular expression is as follows: we try to find a freeway sensor being turned off (`FS OFF`), then we check that several commands may happen (`(.|\n)`) but none can be the traffic light being turned green (`?!TL GREEN\n`). Finally, we check that the freeway sensor is turned on again (`FS ON`). This means that after the freeway sensor was turned off, the traffic light did not turn green, which is a violation. The following section (after the `|`) does the same but checks for the warning signal this time.

Whenever this RegEx matches, we know there is a violation of the requirements. This is called a *negative match*. A *positive match* does the opposite: the code is deemed correct if the trace matches your description (in *Regular Expressions*, *FSA*, ...). For all parts, you are allowed to use either a 'positive match' or 'negative match'. 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!

Clarification: the above uses a *Regular Expression* notation commonly used in *UNIX Regular Expressions* (as used in the stream editor `sed`, for example). Below, you can find a short description of RegEx notation, as is required for the assignment.

- `[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 the trace".*
- `$` means "match at the end of the trace".*
- `X|Y` means "match either `X` or `Y`", with `X` and `Y` both sub-expressions.
- `[^x]` means *not* `x`, hence `^[^E].*\n` matches every line except those that start with the `E` character.
- `.` matches any single character. Depending on the used program, the `.` might either match a newline or not.* In Python's `re` module, this does not match a newline by default.
- `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. *It is, however, not required to implement this feature to be used anywhere. You may require storing the car's ID in the automaton/RegExes. It is sufficient to allow this functionality*

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.
* Some programs allow additional RegEx flags that change the behaviour of these symbols. You may use these flags in your solution if you clearly indicate this in your report.

5. Design a *FSA* which encodes the *Regular Expressions* for verification (example). You can use GraphViz, PlantUML, Diagrams.net or FSM to design your automata (note that in FSM, it is not possible to mark an initial state, so just name it "`init`").
6. Implement this *FSA* for verification in the provided code framework (see scanner.py; an example is included at the bottom of the file).
7. There are 6 output traces available for you to test your code out: trace1.txt, trace2.txt, trace3.txt, trace4.txt, trace5.txt and trace6.txt. Run your *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 traces to verify the specification.
8. There is an intentional bug in the implementation (which is visible in some of the traces) that causes the system specification not to be satisfied. You need to figure out which requirement is being violated and show how your *FSA* checks this. Identify which of the traces show the violation and which ones don't. Also, describe in your own words what the meaning is of this bug; for instance: *The traffic light never turned green after car x has been detected by the demand sensor.*.
9. Write a report that explains your solution for this assignment. Include your models and discuss them. Make sure to mention all your hypotheses, assumptions and conclusions. See also the "*submission information*" at the top of this page.

Take pride in your work. Make sure your report and all models and images are clearly readable. Additionally, produce clean and (preferably) well-documented code. If a text-to-figure transformation (such as for GraphViz and PlantUML) causes your images to be difficult to read, also include the source files from which these were generated (`*.gv` and `*.puml`). You will not lose points for doing too much, but you will lose points if you do too little.

Practical Issues

- All parts of this (and most future) assignments use Python 3.6+. *Do not use features from Python 2.7 (discontinued as of January 1, 2020).*
- There are 6 example output traces available: trace1.txt, trace2.txt, trace3.txt, trace4.txt, trace5.txt and trace6.txt.
- Program used to implement *FSA*: the scanner is in scanner.py, which includes an input stream class `CharacterStream` and a scanner/*FSA* class `Scanner`. A `NumberScanner` is included as an example *FSA*.
- 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/
  - Info on Ramp Metering: https://ops.fhwa.dot.gov/publications/fhwahop14020/sec1.htm
  - How ramp metering works: https://www.youtube.com/watch?v=4HxE3oJYUxs

Maintained by Hans Vangheluwe. Last Modified: 2023/10/13 18:34:18.