WHZ Westsächsische Hochschule Zwickau
Hochschule für Mobilität

Research Report

KFT07000 Research Project Participation

Dipl.-Ing. Benjamin Gabber

Topic 1

Development of a Virtual Signal Controller in Python

Submitted by

Abdul Hadi Mohammed (57882)

Rahil Asit Malviya (56730)

Seminar Group number: **242100**

Table of Content

List of Figures

Abstract

Traffic signal control systems are typically designed using professional planning tools such as LISA+, which allow engineers to define complex signal logic, safety constraints, and operational plans. While these tools are highly effective for configuration and validation, the actual runtime behavior of traffic controllers often remains hidden within closed systems. This project addresses this gap by developing a Python-based Virtual Signal Controller (VSC) that executes real-world traffic signal logic defined in LISA+ XML files within an open and transparent simulation environment.

The proposed system acts as a software-based execution layer that reproduces the behavior of a real German traffic signal controller in real time. It strictly follows RiLSA guidelines, including standard phase transitions, conflict management, and intergreen clearance times. The controller supports multiple Signal Time Plans (STPs) and enables safe plan switching during runtime using a controlled "Safe Switching Window," ensuring stable and collision-free operation at all times.

By externalizing LISA-defined logic into Python, the project provides a flexible digital twin of a signalized intersection. This approach enables detailed observation, safe experimentation, and future extensions such as adaptive or AI-based control strategies, making the system valuable for education, research, and advanced traffic engineering studies.

1. Introduction and Objectives

Traffic signal control systems are usually developed using professional planning tools and strict safety rules. One such widely used tool is **LISA+**, which allows traffic engineers to design complex signal logic, define stages, intergreen times, detector behavior, and multiple Signal Time Plans (STPs). LISA also provides a test environment where this logic can be checked for correctness. However, even with these capabilities, a gap still exists between **planning and testing logic inside LISA** and **understanding and executing controller behavior in an open, programmable environment**.

The **Virtual Signal Controller (VSC)** project was developed to clearly understand and demonstrate how professional traffic signal logic behaves when it is executed step by step, in real time, similar to a real traffic controller. The main goal of this project is to build a **Python-based simulation environment** that reads **LISA+ XML files** and executes the defined signal logic outside the LISA software.

By doing this, the project does not aim to replace LISA. Instead, it complements LISA by providing an **open and transparent execution layer**, where every signal change, timing decision, and safety check can be observed, logged, and modified. This makes the system especially useful for learning, research, and controlled experimentation without any risk to real traffic.

1.1. Need for a Python-Based Signal Controller

Although LISA allows the creation and testing of complex traffic signal logic, its execution takes place within a **closed and tool-specific environment**. The internal working of the controller during runtime, such as how time is counted, when exactly a signal changes, or how switching decisions are enforced, and its largely hidden from the user. This is sufficient for professional deployment, but it limits deeper understanding, analysis, and experimentation.

A **Python-based Virtual Signal Controller** is needed to address these limitations. It acts as a **software representation of a real traffic controller**, executing the same LISA defined logic but in an open programming environment. This provides several important advantages:

- **Clear Visibility:** The complete controller behavior can be observed second by second, making it easier to understand how signal plans are executed in practice.
- **Safe Experimentation:** New ideas, alternative switching logic, or extreme scenarios can be tested safely without affecting a real intersection.
- **Educational Value:** Students and researchers can study how traffic controllers actually work, rather than only how they are configured.

- **Extensibility:** Python allows future extensions such as adaptive control, optimization methods, or AI-based decision-making, which go beyond the intended scope of LISA.

In this sense, the Python-based controller functions as a **digital twin** of a real traffic signal controller, positioned between planning tools like LISA and traffic simulation environments.

1.2. Key Objectives

- **Data Integration:** To design and implement a reliable XML parsing framework that reads complex LISA+ configuration files and converts them into structured Python data models. This includes extracting signal groups, stages, cycle times, switching points, conflict relations, and intergreen times, ensuring that no planning information is lost during the transformation process.
- **Behavioral Accuracy:** To implement a deterministic, time-based state machine that accurately reflects real German traffic signal behavior as defined in RiLSA. This includes correct execution of standard signal sequences such as **Green** → **Amber** → **Red** during termination and **Red** → **Red-Amber** → **Green** during initiation, as well as precise timing control at the second level.
- **Safety Enforcement:** To ensure collision-free operation by strictly enforcing the Conflict Matrix and Intergreen (Clearance) Matrix during all stage transitions. The controller must guarantee that incompatible signal groups never receive green simultaneously and that sufficient clearance time is always provided between conflicting movements.
- **Real-Time Interactivity:** To develop a multithreaded runtime environment that allows users to interact with the controller during operation. This includes switching between different Signal Time Plans (STPs) via keyboard input, while ensuring that all changes occur only at safe and stable switching points without violating safety constraints.
- **Modularity and Extensibility:** To structure the controller architecture in a modular manner so that individual components—such as XML parsing, control logic, visualization, and communication—can be easily modified or extended. This objective ensures that the system can serve as a foundation for future research topics such as adaptive control, detector-based logic, or AI-assisted traffic signal optimization.

2. Intersection Specifications and Planning in LISA

For the foundation of this research, we selected **Intersection 311** in Zwickau, Germany. The physical layout and traffic requirements of this location provided a realistic environment for testing complex signal transitions and safety protocols.

2.1. Intersection Modelling

Fig 2.1 : Google Satellite Image of Intersection 311

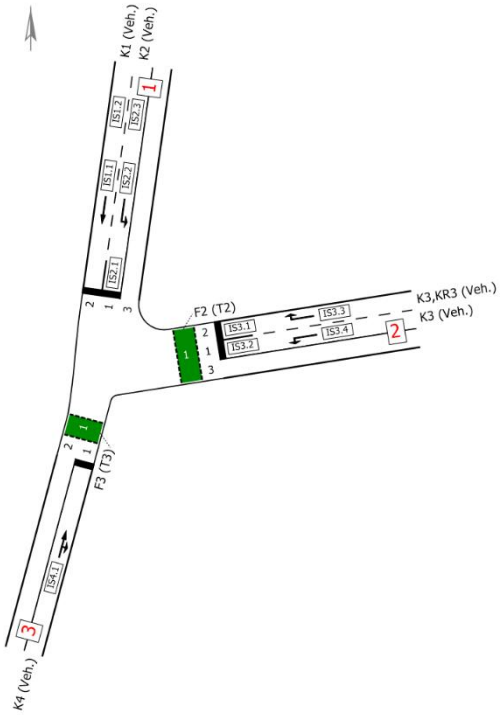

Fig 2.2: Site Plan of Intersection 311 (created in LISA)

The intersection was meticulously modeled using **LISA 8.2** software. This planning phase involved:

- **Stage Design:** We defined five distinct stages that allow for safe vehicle and pedestrian movements.
- **Intergreen Matrix:** Following the **RiLSA** safety guidelines, we calculated the mandatory "Intergreen Times" (Zwischenzeiten). These are the clearance periods required to ensure that once a signal group turns red, the intersection is clear before a conflicting group receives a green light.

2.2 Creation of Signal Time Plans (STPs)

To demonstrate the adaptability of our controller, we designed three unique **Signal Time Plans (STPs)** within the LISA environment. Each plan features a different stage sequence and timing logic to accommodate varying traffic scenarios:

- **STP 1 (Standard):** Follows a sequence of stages 1-2-3, representing a balanced traffic flow.
- **STP 2 (Optimized):** Follows a sequence of 2-1-3, prioritizing secondary road movements.
- **STP 3 (Alternative):** Follows a sequence of 3-1-2, providing an alternative routing strategy.

These three plans were specifically engineered to fulfill the interactive requirements of the project. By creating three distinct plans, we enabled the "Communicative Interface" in our Python script, where pressing keys **1, 2, or 3** triggers a seamless transition between these professionally designed traffic patterns.

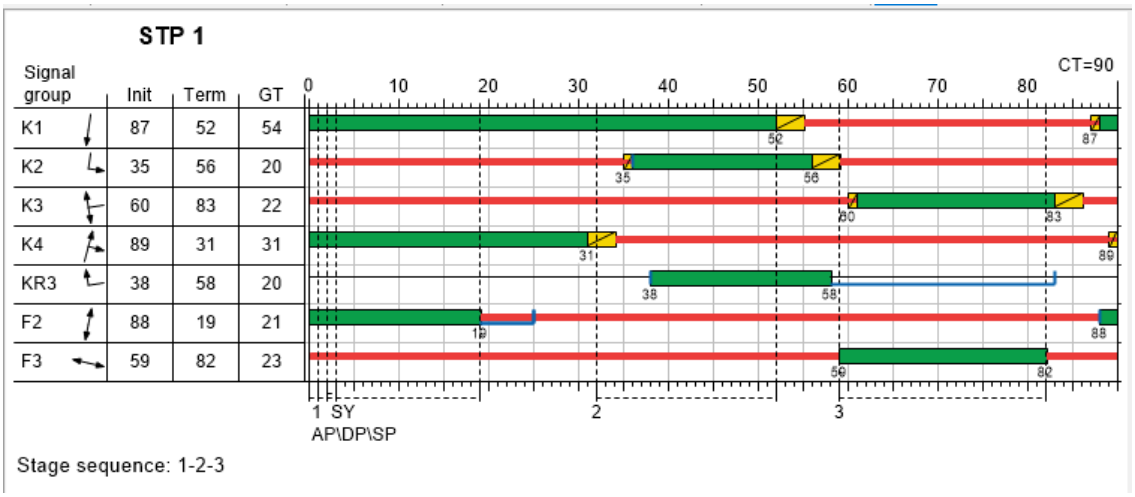

Fig 2.3 : STP 1 (Created in LISA)

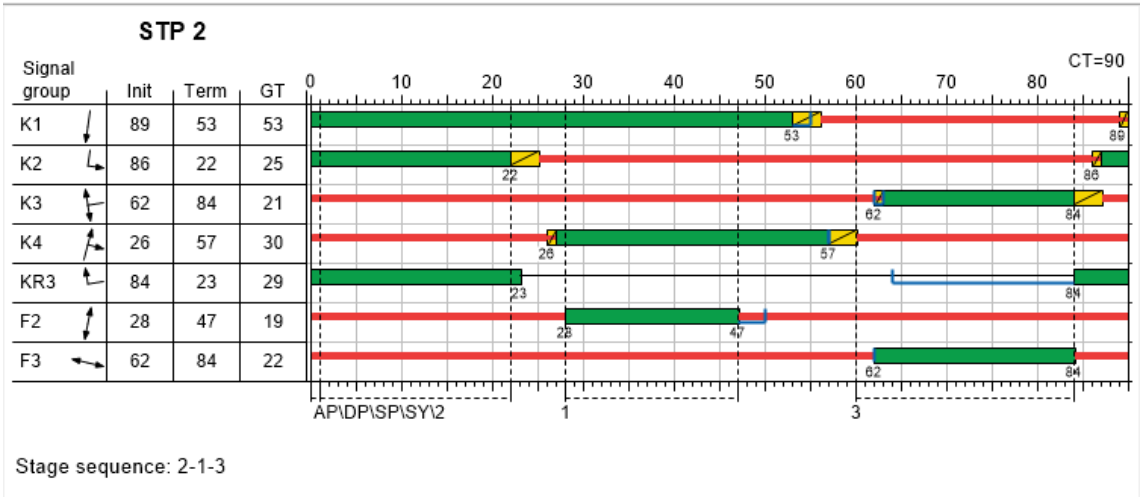

Fig 2.4 : STP 2 (Created in LISA)

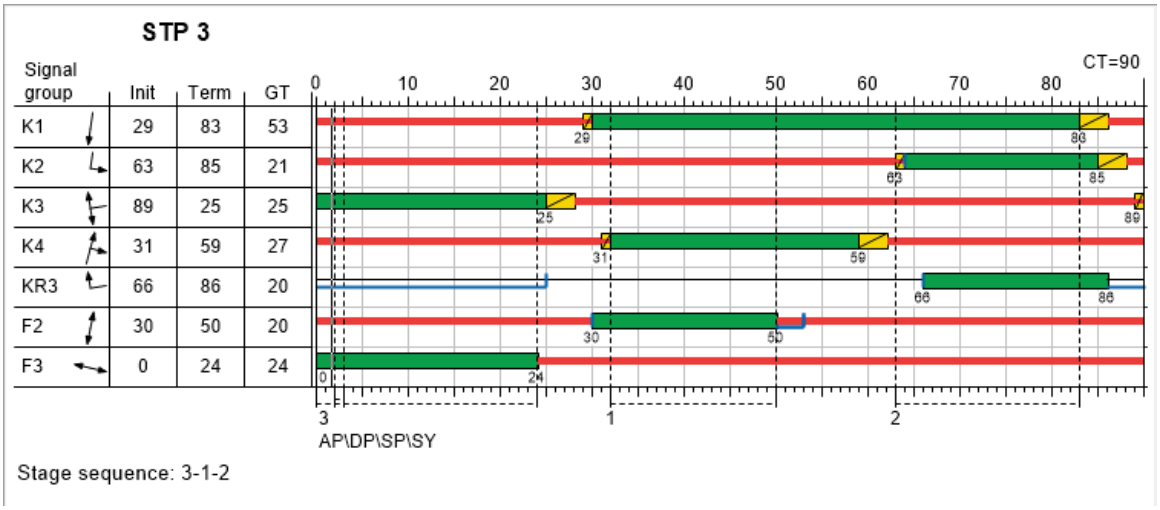

Fig 2.5 : STP 3 (Created in LISA)

3. Methodology and Approach

The project is built on a modular architecture, separating data, logic, and visualization. Below is a detailed explanation of each file used in the controller, including its working principle and origin.

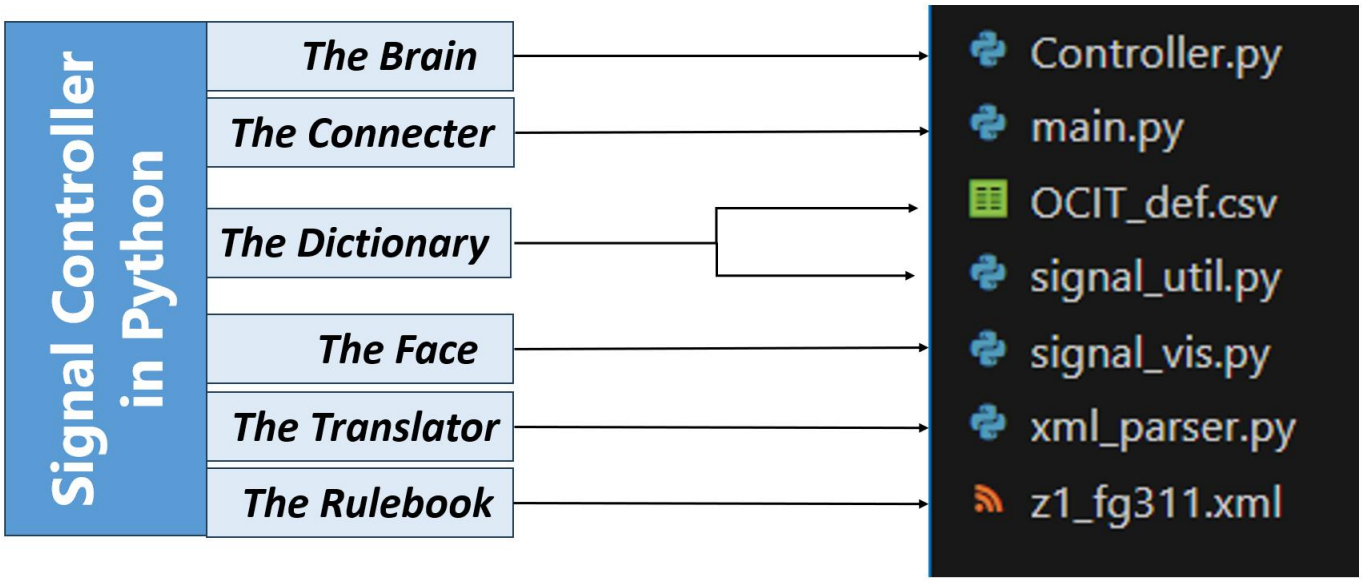

Fig 3.1 : Architecture of the Signal Controller

3.1. Configuration and Data Support Files

- **z1_fg311.xml (The Rulebook)**
  The foundation of the simulation is this XML supply file, exported from LISA+. It acts as the "Rulebook" for the specific intersection in Zwickau (Intersection 311). It contains the definitions of all signal groups (K1–K4, F2, F3), the Signal Time Plans (STP), and the critical Conflict Matrix that dictates safety parameters.

- **OCIT_def.csv (The Dictionary)**
  Provided by Dipl.-Ing. Benjamin Max Gabber.
  This file serves as a standardized "Dictionary" that maps industry-standard OCIT integer codes to human-readable signal states (e.g., Code 3 = "rot", Code 48 = "gruen"). It ensures that the software uses the same "language" as professional traffic hardware used throughout Europe.

3.2 Processing and Logic Files

- **xml_parser.py (The Translator)**
  To make the raw XML data usable, this custom ETL (Extract, Transform, Load) utility was developed. It deserializes the XML into Python-friendly dictionaries. A key feature of this parser is its dynamic adaptability. The code is designed to work perfectly even if the XML file is changed or replaced with a different intersection's data. It can still accurately extract signal groups, cycle times (TU), and stage durations without needing code modifications.

- **signal_util.py (The Dictionary Helper)**
  Provided by Dipl.-Ing. Benjamin Max Gabber.
  This script is responsible for loading the OCIT_def.csv into the system's memory. It creates lookup tables (dictionaries) that allow the controller and visualization layers to instantly translate between signal names, codes, and ASCII representations.

- **Controller.py (The Brain)**
  The logic engine acts as the "Traffic Warden." It manages the State Machine, executing predefined sets of compatible signal states. It handles the European sequence of Green → Amber → Red and Red → Red-Amber → Green. Crucially, it implements the "Safe Switching Window" at 90% of a stage's duration to ensure plan changes occur only at stable cycle boundaries.

3.3. Execution and Visualization Files

- **main.py (The Connector)**
  Core framework provided by Dipl.-Ing. Benjamin Max Gabber.
  We significantly expanded this file to include a communicative keyboard interface. Users can now interactively switch between STP 1, 2, and 3 using the '1', '2', and '3' keys. It acts as the "Conductor," using Multithreading to run the Controller logic in the background while simultaneously spawning individual GUI windows for each traffic light. It uses queue.Queue to ensure real-time communication between the logic and the visuals.

- **signal_vis.py (The Face)**
  Provided by Dipl.-Ing. Benjamin Max Gabber.
  This is the graphics engine. Using the Tkinter library, it renders the visual representation of the traffic lights. It contains the binary mapping patterns (e.g., [1, 1, 0] for Red-Amber) and handles complex visual tasks like signal flashing (Frequency/Hz) and maintenance blinking (WBL).

4. Simulation and Interactive Control

When we run the simulation, we aren't just watching lights blink; we are seeing a complex "negotiation" between safety rules and traffic flow. The simulation operates in a real-time loop, processing 1-second intervals to mimic the behavior of a physical controller cabinet sitting at an intersection.

4.1. The Life of a Traffic Cycle

Every time the simulation moves from one "Stage" to another, it follows a strict sequence to ensure no car or pedestrian is ever in danger. This is handled by a three-phase transition logic:

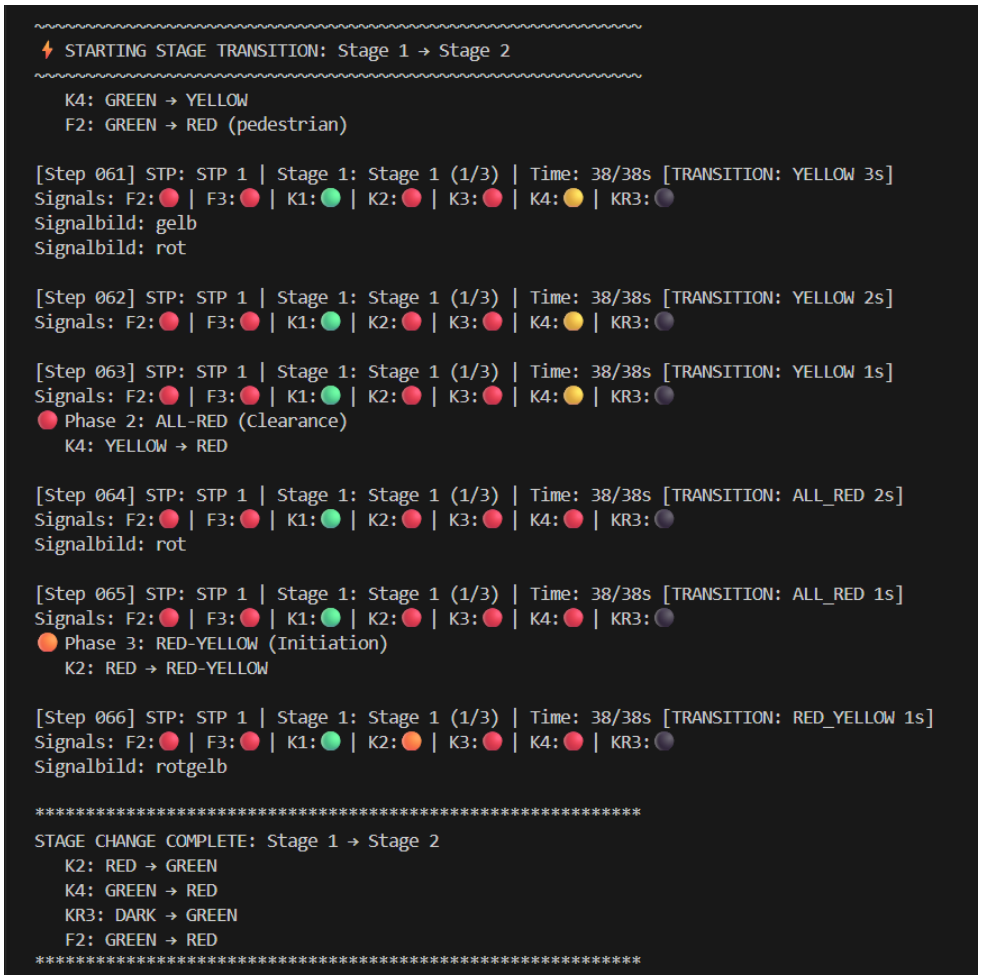

Fig 4.1 : Simulation of Stage Transition from 1 to 2 in STP 1

1. **Termination Phase (Amber)**: The current green lights transition to amber (for 3 seconds). However, our system is smart enough to know that pedestrians don't have a amber light. Their signals jump directly to red for safety.
2. **Clearance Phase (All-Red):** For a brief moment, all lights are red. This is the "Safety Buffer" that allows vehicles currently in the middle of the intersection to clear the area before anyone else starts moving.

3. **Initiation Phase (Red-Amber):** Following the German standard, vehicular signals show red and amber simultaneously for 1 second. This serves as a "get ready" signal for drivers, ensuring a smoother start to the traffic flow once the light turns green.

4.2. Detailed STP Change Request Logic

The most critical challenge in designing a professional controller is managing the transition between different Signal Time Plans (STPs). In a real-world scenario, changing timings abruptly can cause drivers to brake suddenly or lead to gridlock. To prevent this, our system follows a precise, multi-step transition process.

Step 1: User Request (Triggering the Switch)

The process begins when a user interacts with the "Communicative Interface" by pressing keys **1, 2, or 3** on the keyboard. As shown in **Figure 4.2**, this input is immediately captured by the main.py listener. The system acknowledges the request by flagging a "Pending Switch" in the terminal. At this stage, the controller continues its current cycle without interruption, ensuring that no sudden changes occur in the middle of an active green phase.

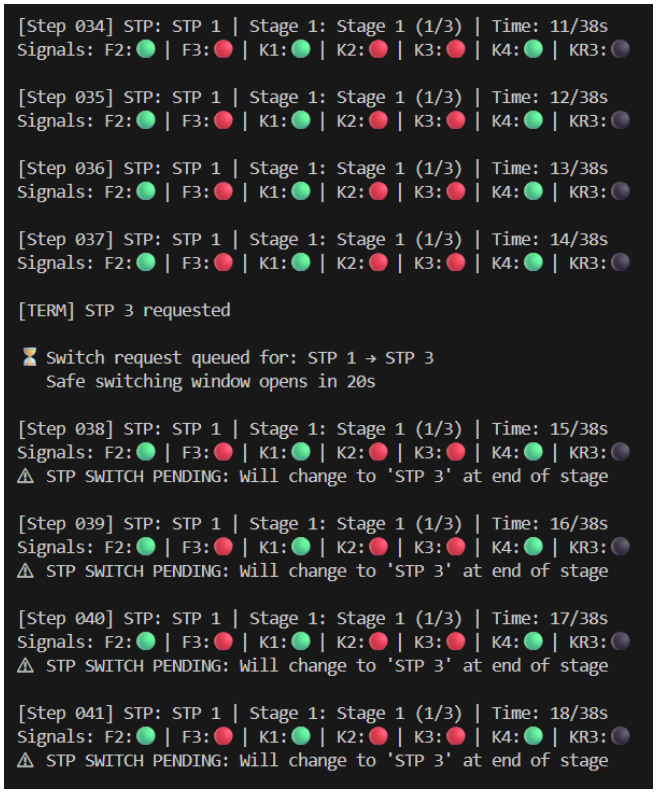

Figure 4.2: Requesting for STP 3 while STP 1 is running.

Step 2: The Safety Validation (The 90% Rule)

Before the switch can proceed, the controller performs an internal safety check. We have implemented a **Safe Switching Window** logic. As illustrated in **Figure 4.3**, the system calculates a threshold at 90% of the current stage's duration (e.g., if a stage lasts 60 seconds, the window opens at 54 seconds).

- If the request is made early, the system waits.
- Once the "Safe Window" opens, the terminal provides visual feedback to the operator (as seen in the highlighted section of **Figure 4.3**), indicating that the switch is now safely queued for the next stage boundary.

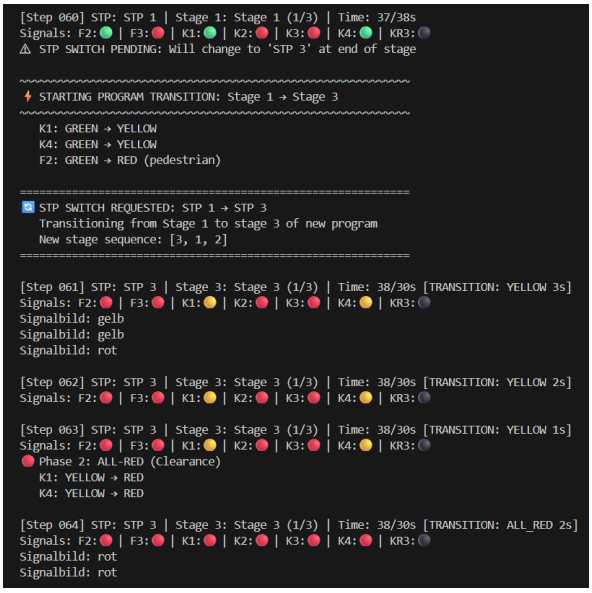

Fig 4.3 : Stage Transition before giving the requested STP

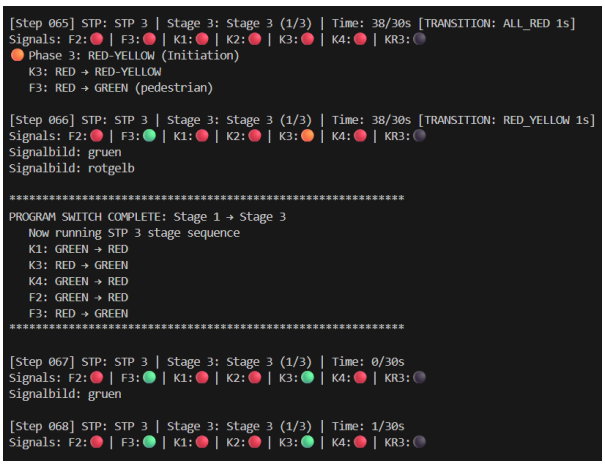

Fig 4.4: Execution the requested STP

Step 3: Execution and Cycle Boundary Synchronization

The actual execution happens the moment the current stage reaches its full duration. Instead of starting the next stage in the *old* sequence, the controller "pivots" to the first stage of the *newly requested* STP.

As shown in **Figure 4.4**, once the switch is successful:

- The terminal confirms: STP SWITCHED: STP X → STP Y.
- The new stage sequence (e.g., 2-1-3) is loaded and displayed.
- The controller forces a "Cycle Boundary Rule" (must_complete_cycle_until_next_boundary = True), meaning the simulation will not stop or allow another switch until it has completed at least one full, uninterrupted cycle under the new plan to stabilize traffic flow.
- This logical flow ensures that the virtual controller behaves with the same predictability and safety-first mindset as a high-end industrial traffic computer.

5. Conclusion and Key Takeaways

The successful development of the Virtual Signal Controller demonstrates the immense potential of using Python as a flexible platform for traffic engineering simulation. By integrating real-world datasets from LISA+ with a custom logic engine, this project has achieved a functional "Digital Twin" of an active urban intersection.

- **Bridging the Engineering Gap:** The project highlights that modern software development can effectively emulate high-level traffic hardware logic. We have moved from static XML planning to a dynamic, interactive execution model that respects the constraints of professional equipment.
- **Safety as a Core Metric:** By strictly adhering to the RiLSA guidelines and implementing a robust Conflict Matrix, we ensured that safety is never compromised for performance. The "All-Red" clearance and "90% Rule" for switching illustrate a safety-first philosophy that is mandatory in civil engineering.
- **Standardization and Interoperability:** Utilizing the OCIT standard and LISA+ XML formats ensures that our system is not an isolated experiment but a tool that aligns with existing European infrastructure. This proves that open-source tools like Python can coexist with and enhance proprietary engineering workflows.
- **Scalability and Educational Value:** The modular architecture—separating the "Brain" from the "Visuals"—allows this tool to be used for educational purposes, allowing students to visualize the immediate impact of timing changes, and for researchers to test adaptive algorithms in a risk-free virtual environment.

References

- **Forschungsgesellschaft für Straßen- und Verkehrswesen (FGSV)**: Richtlinien für Lichtsignalanlagen (RILSA). Ausgabe 2015, Köln, 2015. (https://www.fgsv-verlag.de/rilsa)
- **LISA 8.2:** Intersection configuration and XML export tool.
- **Dipl.-Ing. Benjamin Max Gabber:** Technical mentorship and provision of the core visualization framework.
- **Development Tools:** Visual Studio Code with support from Microsoft AI Co-Pilot.
- **Brilon, W., Wu, N., & Koenig, R. (2022).** Delays and queue lengths at traffic signals with two greens in one cycle. *Transportation Research Record: Journal of the Transportation Research Board*, *2677*(11).
- **OCIT-O Standard (2021).** *Open Communication Interface for Road Traffic Control - Outstations*. Industry standard documentation defining the communication protocols and status codes implemented in our OCIT_def.csv and signal_util.py.