Smart Street Lightning System

Embedded Software Engineering

1st Furkan Ali Yurdakul
Fachhochschule Dortmund
Dortmund, Germany
furkan.yurdakul001@stud.fh-dortmund.de
Matriculation number: 7215046
Contribution to project: 25%

2 nd Bruno Hyska
Fachhochschule Dortmund
Dortmund, Germany
bruno.hyska001@stud.fh-dortmund.de
Matriculation number: 7216772
Contribution to project: 25%

3rd Ian Murnane
Fachhochschule Dortmund
Dortmund, Germany
ian.murnane001@stud.fh-dortmund.de
Matriculation number: 7216662
Contribution to project: 25%

4th Elbek Bakiev
Fachhochschule Dortmund
Dortmund, Germany
elbek.bakiev001@stud.fh-dortmund.de
Matriculation number: 7216249
Contribution to project: 25%

***Abstract*— This report presents an in-depth analysis of the smart street lighting system as a component of the network traffic control in a smart city. The report examines the different stages of the project, including hardware setup and testing, to provide a comprehensive overview of the smart street lighting system. The report considers a set of high-level requirements, including traffic management, energy efficiency, and safety, to ensure that the smart street lighting system meets the needs of a modern and sustainable smart city. The goal of this report is to provide a comprehensive understanding of the smart street lighting system and its role in improving the quality of life for residents and enhancing the overall operations of the city.**

Keywords*— *Smart city, street lighting, network traffic control, functionality, requirements.

I. INTRODUCTION

Figure 1: Mind map highlighting Smart Street Lighting

The implementation of smart cities is gaining traction as a solution to the increasing challenges faced by urban areas, such as traffic congestion, environmental pollution, and limited resources. A smart city incorporates various interconnected systems to improve the quality of life for its residents and enhance the overall functioning of the city. One of the crucial components of a smart city is the smart street lighting system, which is the focus of this report.

This report provides an in-depth analysis of the smart street lighting system as a part of network traffic control in a smart city. The report covers all stages of the project, from analysis to design and implementation, including the hardware setup and testing. The aim of this report is to provide a comprehensive overview of the smart street light system and its role in a smart city.

A mind map is presented to showcase the various elements that make up a smart city, including smart buildings, city planning, urban environmental monitoring, surveillance and safety, and network traffic control. All these elements work together to create a smart city that is efficient, sustainable, and safe. Additionally, a set of high-level requirements have been identified to guide the development of the smart street light system, ensuring that the design and implementation of the system meet the needs of a modern and sustainable smart city. These requirements, outlined in a table, include key functionalities such as traffic management, energy efficiency, and safety.

TABLE I. SMART CITY REQUIREMENTS

ID	*Priority*	*Description*
R01	Must have	Vehicles within the transportation network should transmit a location feed to enable smart traffic management
R02	Should have	Smart streetlights and traffic lights should enter low-power mode when no vehicles are nearby
R03	Could have	Parking should report availability to enable more efficient utilization and less time required to find a park
R04	Should have	Public transport services should make available their data using the Realtime GTFS data specification (Google, 2022).
R05	Could have	Rubbish services can increase efficiency by providing smart bins that sense their fill level. This will allow maintenance services to be managed more efficiently and ensure a cleaner city
R06	Could have	Surveillance should be provided in areas as required and monitored by AI algorithms to increase safety, security and help manage incidents
R07	Could have	Buildings should aim to incorporate smart lighting and heating to increase energy efficiency. This could include various sensors to only enable sectors currently occupied by workers, while switching off areas not occupied
R08	Could have	Smart environmental monitoring should be implemented by collating various sources of data into a central database. This could be used to ensure high quality environmental characteristics, such as air, noise, water, soil, and waste monitoring
R09	Must have	Urban planning should be mindful of, and incorporate as much as possible, disability aids to ensure everyone can use infrastructure equally
R10	Should have	Distances between housing and workplaces should be reduced as much as possible. This can reduce transportation network usage and vehicle pollution, while increasing the free time and work-life balance of the population

II. Application Domain Overview

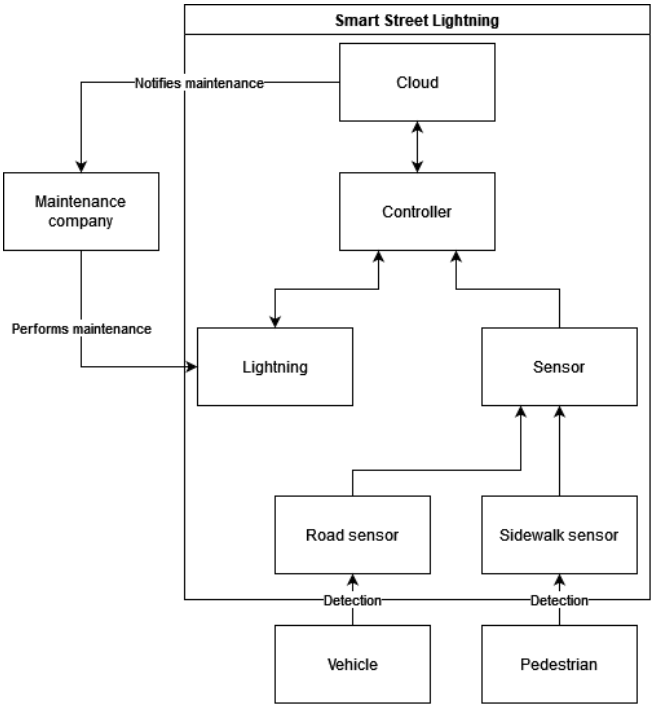

Figure 2: Context diagram

The smart street lighting system is a crucial component of the smart city's network traffic control system. Its purpose is to improve the quality of life for residents by optimizing the functionality of the city, managing traffic congestion, and reducing environmental pollution. The context diagram of the smart street lighting system provides a visual representation of its main components and how they interact with each other, including key external entities such as Pedestrian, Vehicle, and the Maintenance Company.

The system consists of four main components: the Cloud, the Controller, the Sensors, and the Lightning. The Cloud acts as a central repository for data and allows remote monitoring and control of the system. The Controller serves as the central hub, communicating with other components and managing the flow of information between the Cloud, the Sensors, and the Lightning. The Sensors gather data from various sources, including road sensors and sidewalk sensors, and send it to the Controller. The Lightning component is responsible for implementing the lighting control decisions made by the Controller.

III. System Design

In this chapter, we will be discussing the design of the smart street lighting system, including the allocation diagrams, constraint diagrams, state machine, and task scheduling. These diagrams and models provide a comprehensive overview of the system's architecture and functionality, serving as the basis for the implementation of the system. Through careful analysis and consideration of the high-level requirements outlined in the previous chapter, the system design has been developed to meet the needs of a modern and sustainable smart city.

A. Allocation diagram

The allocation diagrams play a crucial role in the system design process of the Smart Street Lighting project. These diagrams provide a clear and visual representation of the mapping of software components onto the hardware platform, enabling us to determine the physical distribution of functions across hardware and software components.

This chapter will provide a detailed description of the allocation diagrams for the Controller, Sensors, and Lightning components, outlining the mapping of software components onto the hardware platform and the associated implications for the system design.

1) Controller subsystem

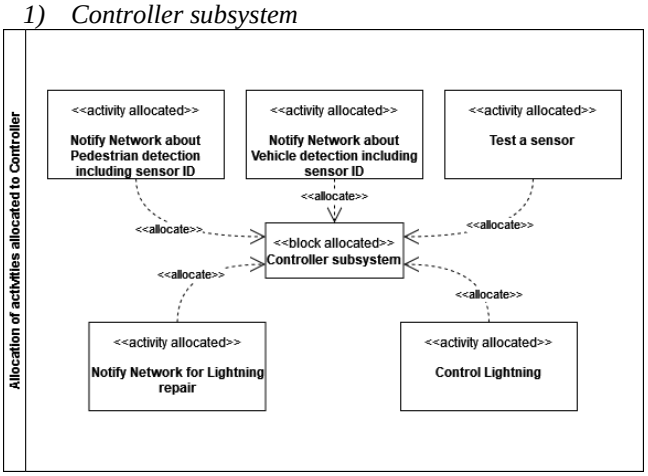

Figure 3: Allocation diagram for the Controller subsystem

The Controller subsystem is the central component of the Smart Street Lighting system, responsible for managing and controlling various functions such as detecting and notifying the network of pedestrian and vehicle detections, testing sensors, and controlling the lighting system. The allocation diagram for the Controller subsystem represents the mapping of the various functions onto the hardware platform.

In the allocation diagram, four main functions can be seen: Notify Network about pedestrian detection including sensor ID, Notify Network about Vehicle detection including sensor ID, Test a sensor, and Notify Network for Lightning repair. Each of these functions has been allocated to the Controller subsystem, indicating that the hardware component responsible for carrying out these functions is the Controller.

Additionally, the allocation diagram also shows the Control Lightning function being allocated to the Controller subsystem. This means that the Controller is responsible for controlling the lighting system and adjusting as necessary.

2) Lightning subsystem

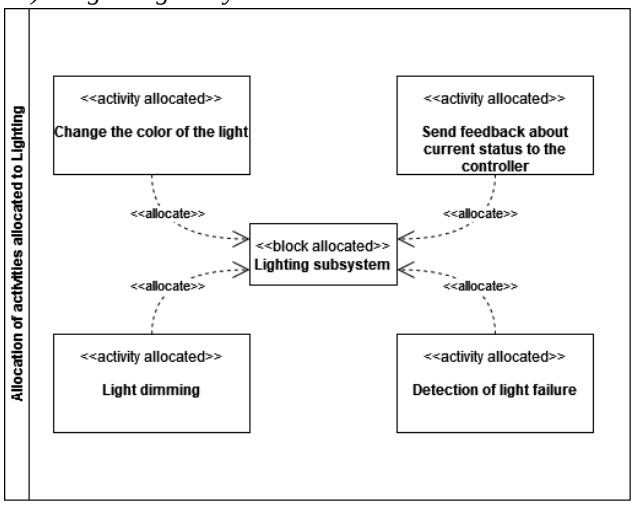

Figure 4: Allocation diagram for the Lightning

As can be seen from the diagram, the Lightning Subsystem is responsible for several important functions, including the ability to change the color of the light, send feedback about the current status to the controller, perform light dimming, and detect light failures. Each of these functions is critical to the overall operation of the Smart Street Lighting System and the proper functioning of the Lightning Subsystem.

It's important to note that the Lightning Subsystem communicates with the Controller Subsystem, allowing it to receive commands and feedback from the overall system, while also sending status updates back to the Controller. This communication ensures that the system is always aware of the current status of the lighting and can respond accordingly.

3) Sensor subsystem

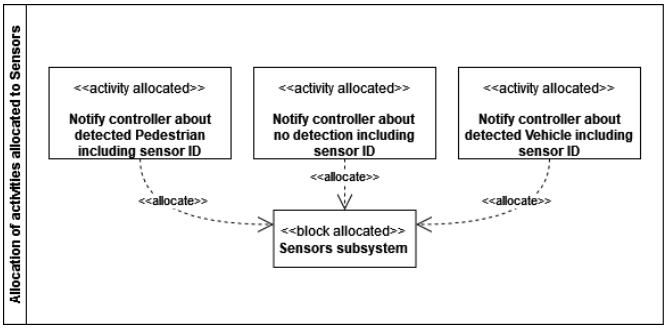

Figure 5: Allocation diagram for the Sensor

The Sensor Subsystem is an important component of the Smart Street Lighting System, responsible for detecting pedestrians, vehicles, and other important events. As depicted in the allocation diagram, the Sensor Subsystem is responsible for notifying the Controller Subsystem about any detected pedestrians or vehicles, including the sensor ID. It's also responsible for sending notifications about no detection, providing valuable information to the system and ensuring that the lighting is operating effectively.

It's important to note that the Sensor Subsystem is constantly monitoring the environment and sending updates to the Controller Subsystem. This communication allows the overall system to respond quickly to changes in the environment, providing a safe and efficient lighting experience. The allocation of these functions onto the Sensor Subsystem ensures that this critical component is fully equipped to perform its tasks and contribute to the overall success of the Smart Street Lighting System.

In conclusion, the allocation diagrams provide a clear understanding of the mapping of software components onto the hardware platform and their physical distribution. By outlining the functions of the Controller Subsystem, Lightning Subsystem, and Sensor Subsystem, we can see how these components interact with one another and play a crucial role in the overall operation of the Smart Street Lighting System. The allocation diagrams help in determining the physical distribution of functions across hardware and software components and play a significant role in ensuring that the system is designed and implemented efficiently. With this information in hand, we can move forward with confidence in our system design, knowing that we have taken into account all of the necessary components and their interactions.

B. Constraint diagram

Parametric Constraint Diagrams are powerful tools that help to represent the mathematical relationships between different parameters in a system, and to understand the impact of these relationships on the desired outcome. They can be used to model a wide range of physical, engineering, and design problems, and are especially useful in optimization and control systems.

One of the key features of parametric constraint diagrams is their ability to specify a network of constraints that represent mathematical expressions. They can also be used to identify critical performance parameters and their relationships to other parameters. This can be particularly useful in the design and optimization of complex systems, where the interplay between different parameters can have a significant impact on the desired outcome. By tracking these relationships throughout the system life cycle, engineers and designers can better understand the constraints and limitations that affect the system and identify opportunities for improvement.

The Parametric Constraint Diagram for the Smart Street Lighting Control System consists of three main components: **Vehicles**, **Light Control System**, and **Pedestrians**. These components are characterized by a set of parameters and constraints as shown in the picture below.

1) Vehicles and Pedestrians Parameters

Vehicles and Pedestrians are represented by the following parameters: Velocity (v), Position (x), Speed (v), and State (S).

2) Light Control System Parameters

The Light Control System is represented by Power Level (PL), System ON (SO), Stay Activated (SA), Sunlight (SL), Lights ON (LON), Input Data from Sensor (IDS), Sensor Triggered (ST), and Sensor Data to Center (SDC).

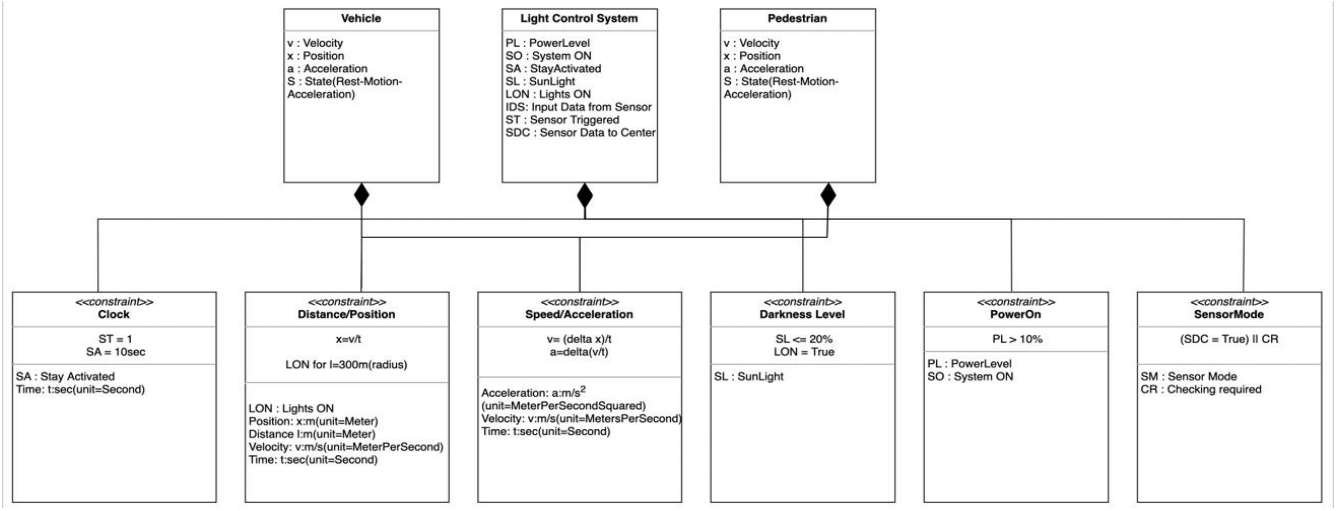

Figure 6: Constraint diagram

3) Vehicles and Pedestrians Constraints

Vehicles and Pedestrians are subject to two constraints. The first constraint, position, ensures that the lights are turned on within a radius of 300 meters based on the position of the vehicles and pedestrians, which is determined using the formula x = v/t. The second constraint, speed, ensures that the system is predicting what lights to turn ON next based on the speed of the vehicles and pedestrians, which is determined using the formula v = x/t.

4) Light Control System Constraints

The Light Control System is subject to four constraints. The first constraint, clock, ensures that the system would stay activated for 10 seconds(SA=10s) if the sensor is triggered(ST=1). The second constraint, powerON, ensures that the System will be ON(SO) as long as the power level of it is more than 10%. The third constraint, Darkness Level, ensures that the Lights will be ON (LON) as long as the Sunlight is less than 20%. The last one is SensorMode which states that the sensors would send data to Center (SDC) if they were fully functional. When they are not, then checking is required (CR).

In conclusion, the Parametric Constraint Diagram provides a comprehensive framework for monitoring and controlling the lighting system in a smart city. It ensures efficient use of city resources and promotes energy efficiency by adjusting the brightness and turning off lights when necessary.

C. State Machine diagram

The Smart Lighting system consists of four main components: vehicles, the smart lighting system, the cloud, and sensors (Fakhroutdinov, n.d.).

1) Sensor

The sensor is responsible for scanning the road for pedestrians and vehicles, and they send the collected information, including coordinates, to the cloud.

A state machine diagram for a sensor of a smart lighting system in a smart city can be described as follows:

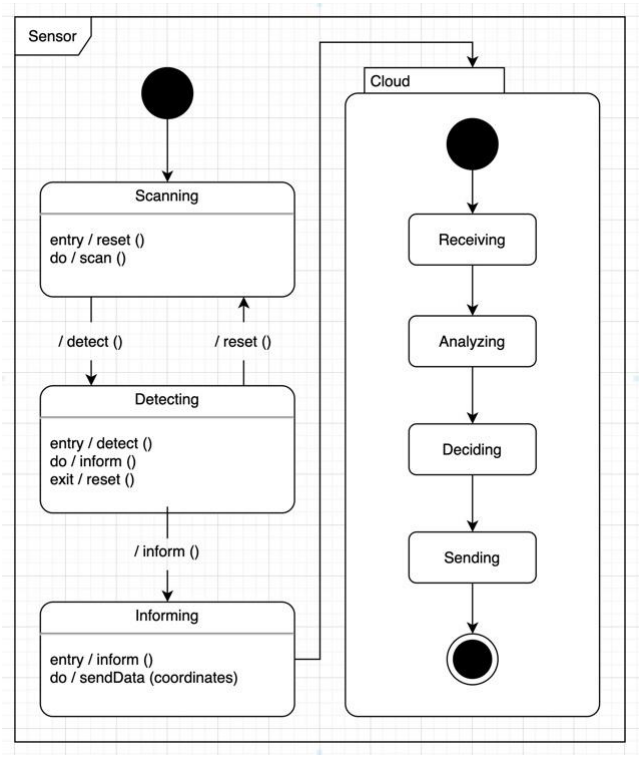

Figure 7: State machine diagram of a Sensor

States:

Scanning: The sensor is in this state most of the time and it constantly scans the environment for any movement or activity.

Detecting: When the sensor detects motion or activity, it transitions to this state and converts the detected input into an electrical signal.

Informing: In this state, the sensor sends the detected data to the cloud for analysis and processing.

Transitions:

Waiting to Light On: The transition from Waiting to Light On occurs when the sensor receives a trigger from the cloud to turn on the light.

Light On to Checking Signal: The transition from Light On to Checking Signal occurs after the light has been turned on, and the sensor checks if it is still being triggered by the cloud.

Checking Signal to Light On or Waiting: If the sensor detects that it is still being triggered by the cloud, it remains in the Light On state. If the trigger is no longer active, the sensor transitions back to the Waiting state.

This state machine diagram represents the basic functionality of a smart lighting sensor in a smart city, showing how it operates in a continuous loop between scanning, detecting, and informing.

2) Streetlight

This state machine diagram represents the basic functionality of a smart lighting system in a smart city, showing how it waits for a trigger from the cloud, turns on the light, and checks if it is still being triggered before turning off the light.

The smart lighting system, in turn, waits for a trigger from the cloud to turn on or off.

A state machine diagram for Streetlight in a smart city can be described as follows:

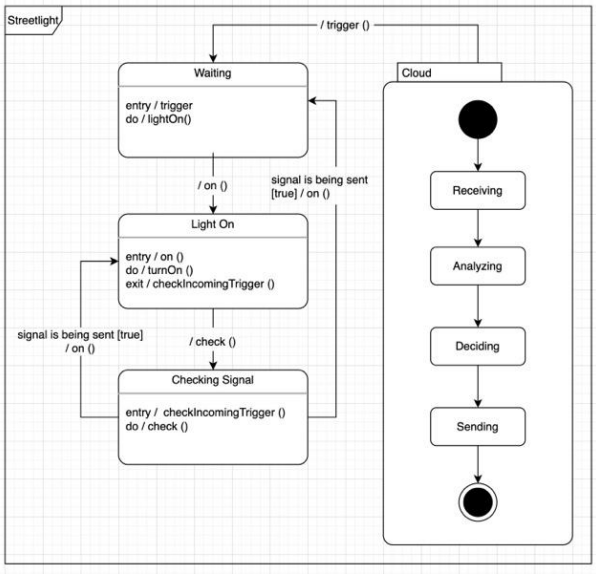

Figure 8: State machine diagram of a Streetlight

States:

Waiting: In this state, the system waits for a trigger from the cloud.

Light On: The system transitions to this state when it receives a trigger from the cloud to turn on the light.

Checking Signal: In this state, the system checks if it is still being triggered by the cloud. If the trigger is still active, the system remains in the Light On state, otherwise, it turns off the light.

Transitions:

Waiting to Light On: The transition from Waiting to Light On occurs when the system receives a trigger from the cloud to turn on the light.

Light On to Checking Signal: The transition from Light On to Checking Signal occurs after the light has been turned on, and the system checks if it is still being triggered by the cloud.

Checking Signal to Light On or Waiting: If the system detects that it is still being triggered by the cloud, it remains in the Light On state. If the trigger is no longer active, the system transitions back to the Waiting state.

3) Vehicle

Vehicles at intersections send requests to the cloud for information about the clearance of the intersection.

The cloud receives signals and coordinates from the sensors and analyzes the information before sending it back to the relevant entities in the system. The system operates in a continuous loop, with each component working in tandem to ensure the efficient functioning of the smart city.

A state machine diagram for a vehicle in a smart city can be described as follows:

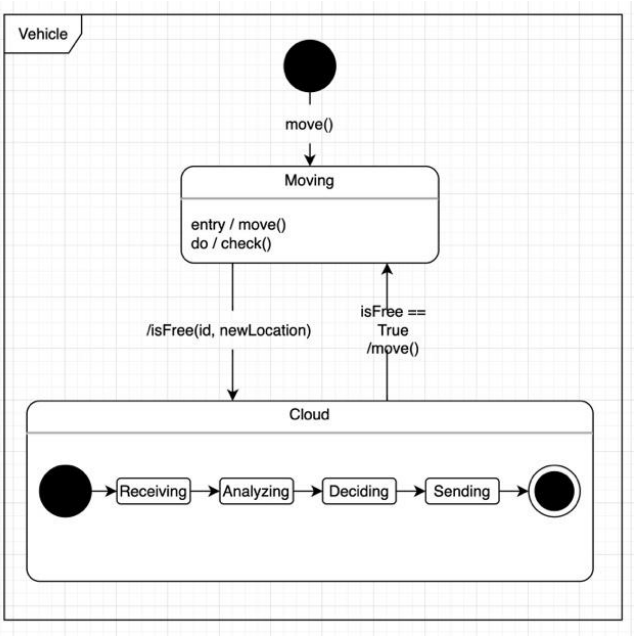

Figure 9: State machine diagram of a Vehicle

States:

Driving: In this state, the car is driven on the road.

Waiting for Clearance: The vehicle transitions to this state when it reaches an intersection and sends a signal to the cloud for clearance to continue driving.

Transitions:

Driving to Waiting for Clearance: The transition from Driving to Waiting for Clearance occurs when the vehicle reaches an intersection.

Waiting for Clearance to Driving: The transition from Waiting for Clearance to Driving occurs when the vehicle receives clearance from the cloud to continue driving.

IV. IMPLEMENTATION

The Implementation section of this project report outlines the steps taken to bring the design of the Smart Street Lighting System to life. This section covers the hardware setup used for the system and the software implementation in the form of C++ code. The hardware setup details the components used to build the physical system and how they are connected. The C++ code, on the other hand, showcases how the system functions were implemented and how they interact with each other. In this section, we aim to provide a comprehensive overview of the steps taken to bring the Smart Street Lighting System to life.

A. Concept Solutions

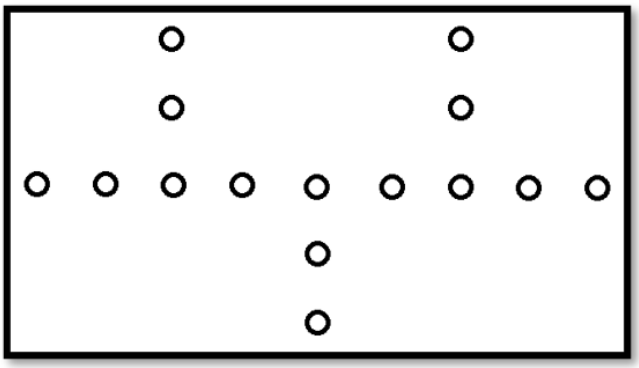

Figure 10: Solution 1

This solution involves the installation of smart streetlights in a city with 3 intersections. The lights can detect the presence of a passing vehicle and become bright as the vehicle passes. Additionally, as the vehicle passes the intersection, the surrounding lights turn on, providing improved visibility and safety for drivers.

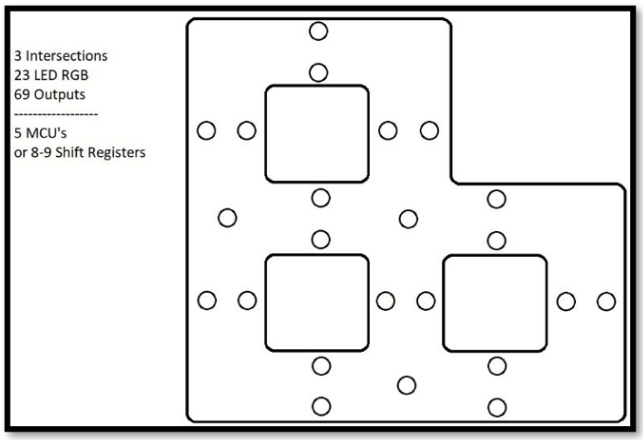

Figure 11: Solution 2

This solution involves the implementation of a smart traffic control system with two lanes and the ability to control 3-4 vehicles. The system allows only one vehicle to be in an intersection at a time, preventing congestion and reducing the risk of accidents. The vehicles can be programmed to follow a track or do figure eights, allowing for various scenarios to be tested and analyzed.

In terms of ease of implementation, the first solution is relatively straightforward and can be quickly implemented. It only involves installing smart streetlights at three intersections and programming them to detect the presence of vehicles and turn them on when needed. This solution can be a good starting point for a city looking to implement a smart lighting system as it requires minimal resources.

The first solution provides a demonstration of how a smart lighting system can improve visibility and safety for drivers. By turning on lights as a vehicle passes, it provides a clear example of how technology can be used to enhance the driving experience and reduce the risk of accidents.

It should be noted that while the first solution is a good starting point, the second solution "Smart Traffic Control" provides a more comprehensive way of optimizing traffic flow and reducing congestion. This solution can be more beneficial in the long run but requires more time and resources to implement.

B. Hardware

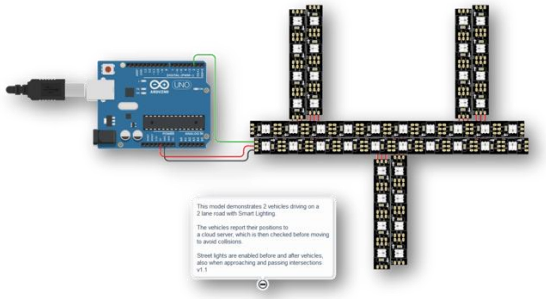

Figure 12: Smart streetlights representation with 3 intersections.

The figure above shows a representation of the Smart Street Lighting System in Tinkercad, which consists of three intersections equipped with six NeoPixel strips with four LEDs and two NeoPixel strips with ten LEDs. These lights are connected to an Arduino Uno microcontroller board, which is equipped with an Atmega328p (1 core/ 1 thread) processor. The Tinkercad model was used to visually demonstrate the functioning of the lighting system and to ensure that all hardware components were properly connected and functioning as expected (Murnane, 2023).

The hardware setup allowed for a hands-on demonstration of the Smart Street Lighting System, providing a better understanding of how the system works and its interactions with other components.

C. Software

The code is organized to show the simulation of two vehicles driving on a two-lane road with Smart Lighting. The vehicles share their positions with a Cloud which analyzes provided data and gives an appropriate response to vehicles to avoid collisions (FH-Dortmunders, 2023).

Code constants and their necessities are described in the comments of the code:

```
#include <Adafruit_NeoPixel.h>

#define PIN 2  // Input pin Neopixel is attached to
#define NUMPIXELS 44  // Total LEDs in all pixels
#define NUMVEHICLES 2  // Number of vehicles, also update the vehicle array

#define YELLOW pixels.Color(255, 255, 0) // Street light color
#define RED pixels.Color(255, 0, 0)  // Vehicle color
#define BLUE pixels.Color(0, 0, 255)  // Vehicle color
#define OFF pixels.Color(0, 0, 0)   // Empty road zone color

Adafruit_NeoPixel pixels = Adafruit_NeoPixel(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);   //  'pixels' object of Adafruit_NeoPixel class
```

The Adafruit NeoPixel library is used in the code to provide an easy-to-use interface for controlling individual LEDs (known as pixels) on a NeoPixel strip. By including the header file in your program, you have access to the functionality provided by the Adafruit NeoPixel library, allowing you to control NeoPixel strips in your projects.

In the Cloud class, the ***update*** method updates the street and address of a vehicle, given its id. The ***getLocationFromId*** method returns the location of a vehicle given its id. The ***isFree*** method checks if a location is occupied by any vehicle except the one given by id. The ***getStreet*** and ***getAddress*** methods return the street and address of a vehicle, given its id.

```
class Cloud {
private:
    int streets[NUMVEHICLES];
    int addresses[NUMVEHICLES];
public:
    void update(int id, int street, int address) {
        streets[id] = street;
        addresses[id] = address;
    }

    int getLocationFromId(int id) {
        return getLocation(streets[id], addresses[id]);
    }

    bool isFree(int id, int location) {
        for (int i = 0; i < NUMVEHICLES; i++) {
            if (i == id) continue;
            if (getLocationFromId(i) == location) return false;
        }
        return true;
    }

    int getStreet(int id) {
        return streets[id];
    }
    int getAddress(int id) {
        return addresses[id];
    }
};
```

In the Vehicle class, the ***move*** method updates the vehicle's location and ensures that the new location is free. If it is not free, the vehicle remains at its original location. The ***location*** method returns the current location of the vehicle.

```
class Vehicle {
private:
    Cloud &cloud;
    int direction;  // always 1 (forward) - can be removed

    bool randomBool() {
        return random(100) < 25;  // 25% chance of true
    }

public:
    int id;
    int street;  // 0-7
    int address;  // 0-9

    Vehicle(int id, Cloud &cloud, int street, int address, int direction) :
            id(id),
            cloud(cloud),
            street(street),
            address(address),
            direction(direction) {};

    void move() {
        int origStreet = street;
        int origAddress = address;
        int location = getLocation(street, address);

        // turn down side-streets
        if (street == 0 && randomBool()) {
            switch (address) {
                case 4:
                    street = 1;
                    address = -1;
                    break;
            }
        }
        if (street == 4 && randomBool()) {
            switch (address) {
                case 1:
                    street = 2;
                    address = -1;
                    break;
                case 7:
                    street = 3;
                    address = -1;
                    break;
            }
        }
        address += direction;
        // ensure the new location is free
        int newLocation = getLocation(street, address);
        if (!cloud.isFree(id, newLocation)) {
            street = origStreet;
            address = origAddress;
```

```
        } else cloud.update(id, street,
address);
    }

    int location() {
        return getLocation(street, address);
    }
};
```

The main part of the program creates two vehicles and stores them in an array called vehicles. It also creates an instance of the Cloud class called cloud.

```
Cloud cloud;
Vehicle vehicles[] = {
        Vehicle(0, cloud, 0, 0, 1),
        Vehicle(1, cloud, 4, 0, 1)
};
```

The method ***addLightingToModel*** adds yellow lights to the simulation model: 1 behind and 2 in front of each car. When a vehicle passes the point that is next to the intersection all the on the crossing are activated.

```
void addLightingToModel(Cloud cloud = cloud)
{
    int lights[] = {-1, 1, 2};  // 1 behind,
2 in front
    for (int i = 0; i < NUMVEHICLES; i++) {
        int address = cloud.getAddress(i);
        int street = cloud.getStreet(i);

        // street lighting
        for (int n: lights) {
            int addressN = address + n;
            // check light address is valid
            if (addressN < 0) continue;
            if (street == 0 || street == 4) {
                if (addressN > 9) continue;
            } else if (addressN > 3) {
                continue;
            }

            model[getLocation(street,
addressN)] = YELLOW;
        }

        // intersection lighting
        if (street == 0) {
            if (address == 3 || address == 4)
{
                model[getLocation(1, 0)] =
YELLOW;
                model[getLocation(1, 1)] =
YELLOW;
            }
        } else if (street == 4) {
            if (address == 0 || address == 1)
{
                model[getLocation(2, 0)] =
YELLOW;
                model[getLocation(2, 1)] =
YELLOW;
            }
else if (street == 7 && address > 1) {
            model[getLocation(4, 7)] =
YELLOW;
            model[getLocation(4, 8)] =
YELLOW;
            model[getLocation(4, 9)] =
YELLOW;
        }
    }
}
```

getLocation function converts street and addresses values to LED location.

```
int getLocation(int street, int address) {
    int location = address;
    if (street > 0) {
        location += 10;
        location += (street - 1) * 4;
    }
    if (street > 4) {
        location += 10;
        location -= 4;
    }
    return location;
}
```

V. TESTING

The correctness and reliability of program execution may be checked by tests. This section first covers the Unit Tests and after continues with the Defect Testing.

A. *Unit Testing*

Unit tests are used to test components and functions of the Cloud, Vehicle, and Lighting. The goal of unit testing is to verify that each unit of the program code performs as expected, both in normal conditions and in response to unexpected inputs or edge cases.

The following lines perform tests on the Cloud class methods:

`test("cloud getLocationFromId", cloud.getLocationFromId(0), getLocation(1, 2))` - tests the getLocationFromId method to ensure that it correctly retrieves the location of the vehicle with ID 0 (street 1, address 2);

`test("cloud isFree same vehicle skip", cloud.isFree(0, getLocation(1, 2)), true)` - tests the isFree method to ensure that it correctly identifies that the location of the vehicle with ID 0 is already occupied and returns true;

`test("cloud isFree check filled pos", cloud.isFree(1, getLocation(1, 2)), false)` - tests the isFree method to ensure that it correctly identifies that the location (street 1, address 2) is occupied and returns false;

`test("cloud getStreet", cloud.getStreet(0), 1)` - tests the getStreet method to ensure that it correctly retrieves the street of the vehicle with ID 0 (1);

The purpose of these tests is to verify that the Vehicle and Cloud classes are implemented correctly and function as expected.

`test("vehicle location", vehicle.location(), getLocation(0, 8))` - tests the location method of the Vehicle class to ensure that it correctly retrieves the current location of the vehicle (street 0, address 8);

`test("vehicle move", vehicle.location(), getLocation(0, 9))` - tests the location method of the Vehicle class to ensure that the vehicle has moved to the next address (street 0, address 9);

`test("cloud update 1", cloud.getLocationFromId(0), getLocation(0, 9))` - tests the getLocationFromId method of the Cloud class to ensure that the Cloud correctly updates the location of the vehicle in its database (street 0, address 9);

`test("vehicle u-turn", vehicle.location(), getLocation(4, 0))` - tests the location method of the Vehicle class to ensure that the vehicle has changed direction and is now on street 4, address 0;

`test("vehicle blocked", vehicle.location(), getLocation(4, 0))` - tests the location method of the Vehicle class to ensure that the original vehicle is blocked from moving because another vehicle is occupying the next address and the original vehicle's location has not changed;

The purpose of these tests is to verify that the Smart Lighting System classes are implemented correctly and all the lights are added to the lanes properly.

```
test("vehicle lighting 1", (int) model[getLocation(0, 1)], (int) YELLOW) ;

test("vehicle lighting 3", (int) model[getLocation(0, 3)], (int) OFF);


test("main intersection lighting 1", (int) model[getLocation(1, 0)], (int) YELLOW);

test("side street intersection lighting 1", (int) model[getLocation(0, 4)], (int) YELLOW);

test("side street intersection lighting 4", (int) model[getLocation(0, 7)], (int) OFF);
```

B. Defect Testing

TABLE II. TABLE DEFECT ID DF-1

Defect ID	Date	Reported	Assigned	Completed
DF-1	30/12/2022	Bruno	Ian	3/01/2023
Short Description			**Acceptance Criteria**	
The model should incorporate traffic control.			• Incorporate 2 or more cars • Actively avoid collisions • Negotiate right-of-way, possibly using a priority system	

Defect testing for Defect ID DF-1 involves verifying that the model has incorporated traffic control as described in the Acceptance Criteria. The following steps can be followed to test the defect:

- Verify the number of cars: The first step is to verify that the model has incorporated at least two cars. This can be done by visual inspection or by counting the number of cars present in the simulation.

- Test for collision avoidance: The next step is to test that the model actively avoids collisions. This can be done by manually placing the cars in a scenario where a collision is likely to occur and observing if the model takes the necessary actions to avoid the collision.

- Test for right-of-way negotiation: The final step is to verify that the model negotiates right-of-way in a realistic manner. This can be done by placing the cars in scenarios where the right-of-way is not clear and observing if the model takes the appropriate actions to negotiate the right-of-way, possibly using a priority system.

TABLE III. TABLE DEFECT ID DF-2

Defect ID	Date	Reported	Assigned	Completed
DF-2	30/12/2022	Bruno	Ian	3/01/2023
Short Description			**Acceptance Criteria**	
The model should have two lanes to demonstrate bidirectional traffic.			• Add an additional lane	

Defect testing for Defect ID DF-2 involves verifying that the model has two lanes to demonstrate bidirectional traffic, as described in the Acceptance Criteria. The following steps can be followed to test the defect:

- Verify the number of lanes: The first step is to verify that the model has two lanes. This can be done by visual inspection or by counting the number of lanes present in the simulation.

- Test for bidirectional traffic: The next step is to test that the two lanes demonstrate bidirectional traffic. This can be done by placing cars on both lanes and observing if they are able to move in opposite directions.

TABLE IV. TABLE DEFECT ID DF-3

Defect ID	Date	Reported	Assigned	Completed
DF-3	5/12/2022	Furkan	Ian	10/01/2023
Short Description			**Acceptance Criteria**	
The code and documentation should be backed up using GitHub version control.			• Create repository • Add code to GitHub • Add documentation to GitHub	

Defect testing for Defect ID DF-3 involves verifying that the code and documentation are backed up using GitHub version control, as described in the Acceptance Criteria. The following steps can be followed to test the defect:

- Verify repository creation: The first step is to verify that a repository has been created in GitHub for the code and documentation. This can be done by logging into GitHub and searching for the repository.
- Verify code in GitHub: The next step is to verify that the code has been added to the GitHub repository. This can be done by accessing the repository and checking that the code is present and accessible.
- Verify documentation in GitHub: The final step is to verify that the documentation has been added to the GitHub repository. This can be done by accessing the repository and checking that the documentation is present and accessible.

TABLE V. TABLE DEFECT ID DF-4

Defect ID	Date	Reported	Assigned	Completed
DF-4	5/01/2023	Furkan	Ian	5/01/2023
Short Description			**Acceptance Criteria**	
Traffic lights talk to the cloud not vehicles.			• Traffic lights query the cloud server to check for nearby cars	

Defect testing for Defect ID DF-4 involves verifying that the traffic lights are communicating with the cloud and not vehicles, as described in the Acceptance Criteria. The following steps can be followed to test the defect:

- Verify cloud communication: The first step is to verify that the traffic lights are communicating with the cloud server. This can be done by monitoring the network traffic to see if the traffic lights are sending requests to the cloud server.
- Verify nearby car detection: The next step is to verify that the traffic lights are using the cloud to detect nearby cars. This can be done by observing the behavior of the traffic lights in scenarios where cars are present and checking if the lights change based on the presence of the cars.

TABLE VI. TABLE DEFECT ID DF-5

Defect ID	Date	Reported	Assigned	Completed
DF-5	10/01/2023	Bruno	Ian	10/01/2023
Short Description			**Acceptance Criteria**	
Add testing			• Add at least 5 unit tests • Document defect testing	

Defect testing for Defect ID DF-5 involves adding at least 5 unit tests and documenting the testing process, as described in the Acceptance Criteria. The following steps can be followed to test the defect:

- Write unit tests: The first step is to write at least 5 unit tests that cover different scenarios and functionalities in the model. These tests should validate that the model behaves as expected and that any defects or issues are identified.
- Run unit tests: The next step is to run the unit tests and observe the results.
- Document the testing process: The final step is to document the testing process, including the tests that were written and the results of running the tests. This documentation should be detailed and comprehensive so that it can be used for future reference and to validate that the testing was performed correctly.

VI. TASK SCHEDULING

Scheduling algorithms are a crucial component of any computer system that manages the allocation of resources and coordination of tasks. In real-time systems, where tasks have deadlines that must be met, scheduling algorithms play a critical role in ensuring that tasks are completed on time and that the system's resources are used efficiently. They are divided into two groups (Rouhifar & Ravanmehr, 2015): Dynamic and Static.

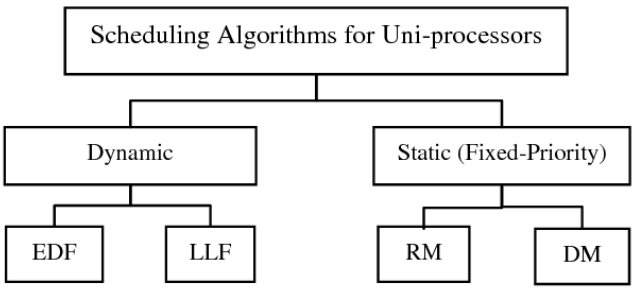

Figure 13: Scheduling Algorithms for Uniprocessors

Dynamic scheduling algorithms, such as Earliest Deadline First (EDF) and Least Laxity First (LLF), adapt to changing conditions and adjust the order in which tasks are executed. EDF schedules the task with the earliest deadline first, while LLF schedules the task with the smallest difference between its deadline and completion time first. These algorithms are based on the idea of optimally allocating resources to tasks that have the earliest deadline, ensuring that the system is able to meet its deadlines.

Static scheduling algorithms, such as Rate Monotonic Scheduling (RMS) and Deadline Monotonic Scheduling (DM), allocate resources and coordinate tasks based on a fixed set of rules. RMS prioritizes tasks based on their period, while DM prioritizes tasks based on their deadlines. These algorithms are based on the idea of allocating resources to tasks based on predefined rules, which can help reduce the computational overhead associated with dynamic scheduling algorithms.

The tasks in the smart city lighting and traffic management system are designed to meet specific objectives and requirements. These tasks are periodic and include operations such as:

1. **Clearing the model**
2. **Adding street and intersection lighting**
3. **Adding vehicles**
4. **Updating LEDs**
5. **Updating vehicle positions**

These tasks play a crucial role in ensuring the proper functioning and efficiency of the system.

To implement these tasks, an Arduino microcontroller is used as the main processing unit. This microcontroller is a single-core device, which means that only one task can be executed at a time. As a result, it is essential to properly coordinate and allocate resources to ensure that all tasks are executed in a timely and efficient manner.

TABLE VII. SCHEDULING

	Clear model	Add street/ intersection lighting to model	Add vehicles to model	Update LEDs	Update vehicle positions
Ai	**0ms**	**0.5ms**	**1ms**	**1.5ms**	**4ms**
Ci	**0.39ms**	**0.21ms**	**0.06ms**	**2.06ms**	**0.36ms**
Di	**5.5ms**	**5.5ms**	**5.5ms**	**5.5ms**	**5.5ms**
Ti	**5.5ms**	**5.5ms**	**5.5ms**	**5.5ms**	**5.5ms**

One of the key considerations in the context of these tasks is the real-time nature of the system. The system must be able to respond to events and changes in the environment in real-time, making the use of effective scheduling algorithms essential. The use of scheduling algorithms helps to prioritize tasks based on their deadlines and allocate resources in a way that maximizes system efficiency and performance.

In this study, we conducted tests using two widely used scheduling algorithms, Earliest Deadline First (EDF) and Rate Monotonic Scheduling (RMS), to assess the schedulability of the tasks in the smart city lighting and traffic management system. To conduct these tests, we used the values of *A(i)*, *C(i)*, *D(i)* and *T(i)* associated with each task, which represent the arrival time, computation time, deadline and period of each task, respectively. These values were used to calculate the utilization factor, U, for each task under both EDF and RMS algorithms (Lu, 2014).

Under the EDF algorithm, the Schedulability Test is based on the condition that U < 1, where:

$$U = \sum_{i=1}^{n} \frac{Ci}{Ti} = 0{,}56$$

If this condition is met, then the task is considered schedulable. In the case of the RMS algorithm, the schedulability test is based on the condition:

$$U \leqslant B(n) < 1$$

where B(n) is the Utilization Bound, calculated based on the number of tasks, n.

$$B(n) = n * \left(2^{\frac{1}{n}} - 1\right) = 5 * \left(2^{\frac{1}{5}} - 1\right) = 0{,}743$$

Based on the values from the tables and from the calculations, we found that the tasks were schedulable under both EDF and RMS algorithms. This result provides evidence that the smart city lighting and traffic management system is designed to meet the real-time requirements of the domain and that the use of effective scheduling algorithms helps to optimize the use of system resources and ensure that tasks are executed in a timely and efficient manner.

VII. Conclusion

The smart street lighting system is a crucial component of the smart city's network traffic control system. The project aimed to design, implement, and test a system that would improve the quality of life for residents by optimizing the functionality of the city, managing traffic congestion, and reducing environmental pollution. The result of this project is a system that can be used as a reference for future implementations of smart street lighting systems.

The system design consisted of four main components: the Cloud, the Controller, the Sensors, and the Lightning. The Cloud acts as a central repository for data, while the Controller serves as the central hub, communicating with other components and managing the flow of information between the Cloud, the Sensors, and the Lightning. The Sensors gather data from various sources, including road sensors and sidewalk sensors, and send it to the Controller. The Lightning component is responsible for implementing the lighting control decisions made by the Controller.

The implementation part consisted of Tinkercad visualization and C++ program code implementation. The hardware setup consisted of six NeoPixel strips with four LEDs and two NeoPixel strips with ten LEDs connected to an Arduino Uno microcontroller board with an Atmega328p (1 core/ 1 thread) processor. The C++ code was implemented to turn on the lights based on the location of the vehicle and when the vehicle approached an intersection, the first few lights of the intersection were also turned on.

The testing phase included unit testing and defect testing, which ensured that the system worked as intended. Task scheduling was also performed using both RMS scheduling and EDF scheduling to evaluate the best scheduling algorithm for the system.

This project demonstrates the feasibility and potential of implementing smart street lighting systems in smart cities to enhance their functionality. The system can be further improved by integrating additional components and algorithms to increase its efficiency and effectiveness.

A. Acknowledgment

The authors would like to extend their gratitude to Prof. Dr. Stefan Henkler who provided classes about Embedded Software Engineering and helpful discussions about our project.

The following is a brief overview of the contributions made by each author:

- Furkan Ali Yurdakul was responsible for the Introduction, Application Domain Overview, Allocation diagram and Conclusion (pages 1-3,12).

- Bruno contributed to the System Design section with his Constraint Diagram and Concept Solutions in the Implementation section (pages 3-4, 6, 11).

- Ian Murnane was responsible for the Hardware and Software within the Implementation section and Unit Testing (pages 6-8).

- Elbek played a key role in the System Design section with the State Machine diagram and contributed to Testing with Defect Testing (pages 4-6,9-10).

All authors contributed to the writing and reviewing of the final draft, also to the implementation of the project and approved the submitted version.

VIII. References

Fakhroutdinov, K. (n.d.). *UML State Machine Diagram Examples.* Retrieved from UML-Diagrams: https://www.uml-diagrams.org/state-machine-diagrams-examples.html

FH-Dortmunders. (2023, January 15). *Embedded Software Engineering*. Retrieved from Github: https://github.com/IanMurnane/Embedded-Software-Engineering

Google. (2022, 09 07). *Google Transit.* Retrieved from https://developers.google.com/transit/gtfs-realtime#overview_of_gtfs_realtime_feeds

Lu, C. (2014, April 9). *Real--Time Scheduling.* Retrieved from Washington University in St.Louis: https://www.cse.wustl.edu/~lu/cse467s/slides/scheduling.pdf

Murnane, I. (2023, January 5). *Copy of Copy of Copy of IcyJet44147*. Retrieved from Tinkercad: https://www.tinkercad.com/things/a4lfmYLPx9I

Puschner, P. (2016, April 1). *Techniques to Calculate the Worst-Case Execution Time.* Retrieved from TU Wien: https://ti.tuwien.ac.at/cps/teaching/courses/wcet/slides/wcet02_static_analysis.pdf

Rouhifar, M., & Ravanmehr, R. (2015, December 17). *A Survey on Scheduling Approaches for Hard Real-Time Systems.* Retrieved from Semantic Scholar: https://www.semanticscholar.org/paper/A-Survey-on-Scheduling-Approaches-for-Hard-Systems-Rouhifar-Ravanmehr/21048468c7297e1eec3e87a79c99b497f1829999