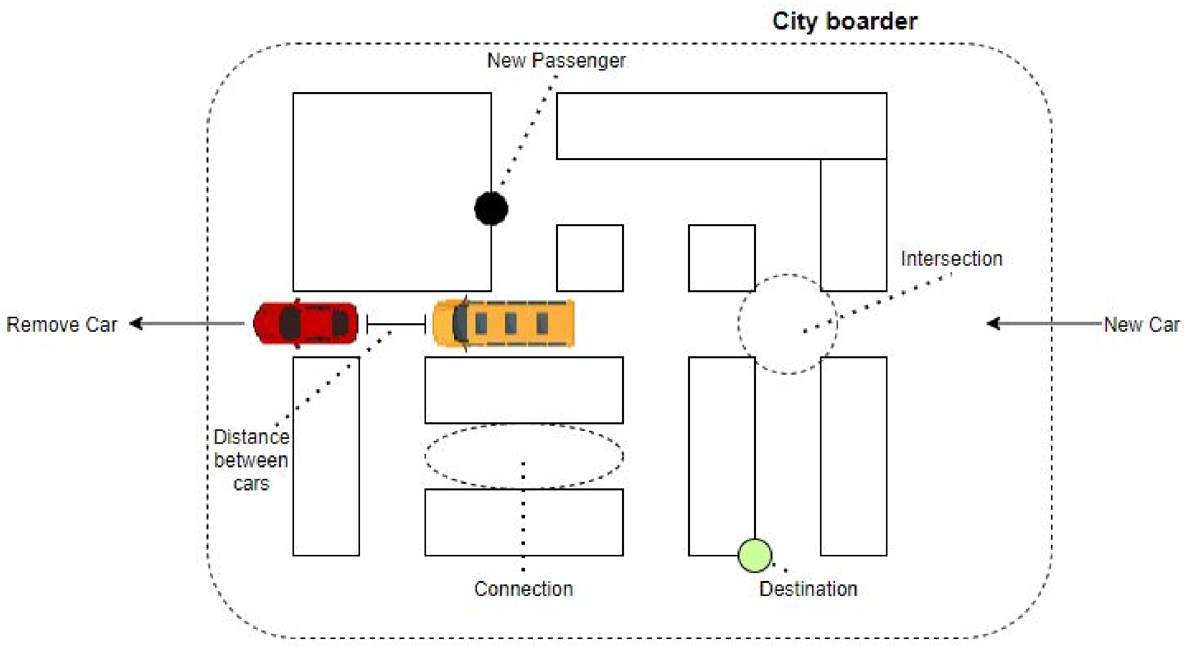
The self-driving traffic control system

Modeling of Critical Systems

Mathias Nyberg Grønne, 201606144

08. December, 2020



Abstract:
The modelling of a critical system is explored through this project. The exploration uses VDM and a structured process to slowly but surely work its way through the creation of a critical system. The process goes from a user defined system description all the way to a technical implementation of the system model. Each step of the way is explained and motivated to understand why each is important. In the technical implementation of the model are VDM specific elements and model concepts explained. The explanations are used to understand how a model is created and how to begin on this approach in future development projects. The project also extends the model to the sequential level in the VDM-RT process[5] to showcase how time easily can be represented. Finally is unit tests used to verify the model.

Table of content

Introduction

This project is a part of the course “Modeling of Critical System”[1] offered by Aarhus University Department of Engineering in the fall semester of 2020. The goal is to give the student hands-on experience with the theory taught in class and for the student to show capabilities in the programming language Overture[2]. Overture is a formal language building on the Vienna Development Method(VDM)[3], using the VDM++[3] extension to add object-oriented functionality to the language. These tools are used to model a critical system which functionality must be secured before deployment. In order to ensure mission safety a structured process must be followed. The process provides an in-depth understanding of the problem space which creates a better foundation to build on top of.

To show these capabilities have the “The self-driving traffic control system” been chosen. The system is a mission-critical system, where an error can lead to misdirection in transportation or even cars crashing together. This system allows for the student to show both a structured process in identifying important classes and models, but also in using major parts of the Overture tools.

The project is divided into three major parts:

1. Analyze the problem and formalize the system
2. Implement the system
3. Test the system to ensure safety

1 Is the major part of the report, 2 and 3 have less of a presence but can be found in full length in the appended code[4]. The result section focuses more on the final product than on every detail from the code and tests. The development process follows [5] starting with the system boundaries and moves toward a real-time system, improving the system in iterations. The development process can be found in figure 1.

The project is made by Mathias Nyberg Grønne, 201606144, and guided by Peter Gorm Larsen [6], course instructor.

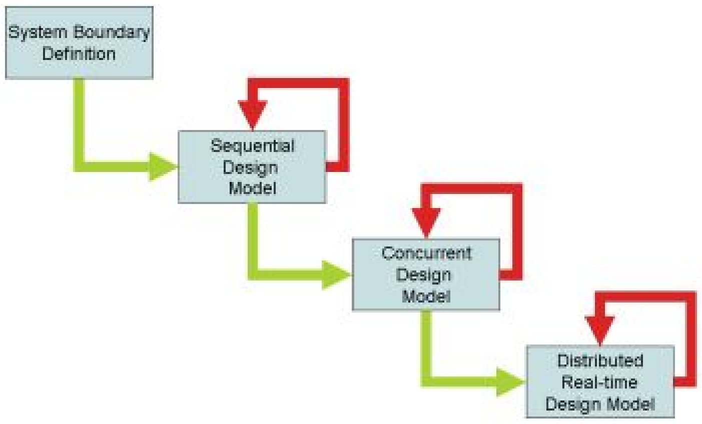

System Boundary Definition
Sequential Design Model
Concurrent Design Model
Distributed Real-time Design Model

Figure 1 - The VDM real time process, going step by step towards a more embedded model. Image from [5]

Motivation

A clear projection for the future is visible when looking at the technological innovations of cars. The public interest and the functionality for cars have over the past decade had more and more focused on self-driven capabilities[7][8]. Cruise control came 40 years ago, adaptive cruise control was introduced 20 years ago [9], and automatic parking was implemented just 10 years ago [10]. Every new innovation has pointed in the direction of more autonomous driving and a less straining job for the driver. Just over the past 5 years cars have gone from level 0 to level 3 autonomy [12]. This progression in autonomy has shown no sign of slowing down[13].

It is important for the political system to soon consider their stance in this future. Do they want cars to act independently, letting the driving be up to the individual company, or law enforce a system that collectively controls all autonomy? Each of the approaches brings with them pros and cons and it can be hard to say which approach is better.

The independent cars need to be stronger in autonomy in order to navigate by themself, this makes them better and safer to use in a chaotic traffic environment. on the other hand, each company will be a risk factor because no standard exists to ensure the safety of passengers and pedestrians. The independent approach will also have the danger of not knowing the intent of other cars and thus in general be in a more chaotic system where only one car needs to be in the wrong to create critical scenarios.

The central control system will be more efficient in its navigation and more secure from knowing the intent of every car in the system. But problems can occur because the cars are less intelligent, making for more error in their independent skills to handle sudden problems in the environment. A city is a chaotic system where everything can happen, a pedestrian can walk out on the road or a car without autonomy can crash into the system.

This project looks into the capabilities of a centrally controlled system where the autonomous cars are navigated by a central unit. The project is not to advocate for a centrally controlled approach but instead to explore the capabilities of such a system. This can hopefully shed light on alternative ways for the future, contributing to a more nuanced discussion.

Description

A concept image of the system can be found in figure 2. The concept is of a city where autonomous cars drive around. The goal is to transport passengers from their current location to their destination. The cars come in three different sizes: 1-, 4-, and 8-passenger cars. The more passengers the car can transport the less it costs for the passenger but the longer it can take as a consequence. The system plans each car's route and both minimizes the transport time for each passenger and the distance driven by each car to deliver those passengers.

The city consists of intersections and roads(connections) between each intersection. Each intersection can hold one car at a time and each connection has a set length and speed limit. The passengers will appear by an intersection and will be dropped off by their designated intersection. Because a city is a chaotic environment, intersections can shut down and roads can be blocked. The system must be able to handle these errors and reroute in case the intersection or road was a part of the plan. The closest intersection will be used as a drop off location in case a current passenger's destination is shut down.

Both autonomous and non-autonomous (manual) cars can be a part of the city. Both types of cars can enter and leave the city through the city borders. The autonomous cars will enter and leave the city based on how many requests there are for transportation. The non-autonomous car is only there to explore the city and will leave the city again after driving around for a bit; their routes are unknown.

There must be a set distance between the cars In order to ensure safety, the distance changes with speed. The distance must not only be kept to other autonomous cars but to other objects on the road as well (e.g. non-autonomous cars or pedestrians).

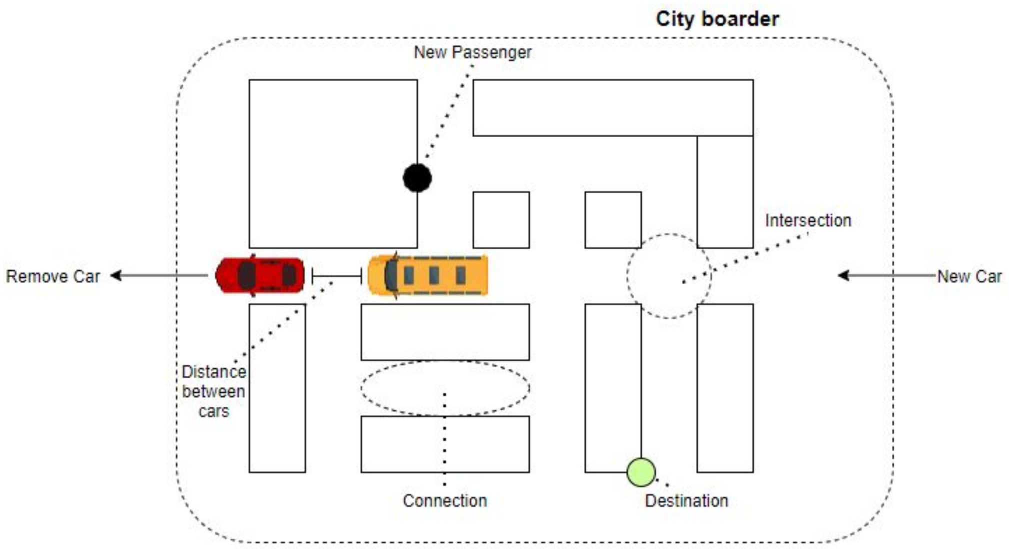


Figure 2 - Concept image of the Central controlled traffic system

System Requirements

The system requirements are based on the system description from *Description*. The requirements are split into two types: Functional and nonfunctional. The functional requirements describe the system behaviour and functionality. The nonfunctional requirement describes the system qualities; everything that can be measured.

System requirements are used to define which specifications that the system must agree to. This creates a written agreement of what must be in the system and creates a common ground to discuss the system from. This helps in understanding whether the agreement is in accordance with the system description and satisfies the intended purpose of the stakeholders. The requirements will be prioritized in accordance to MoSCoW [14] to allow for an iterative process. Each functional requirement will be further prioritized with a priority value starting from 1 as the highest priority.

Functional requirements

The functional requirements can be found in table 1.

Table 1 - Functional Requirements

ID	Description	Priority
R.F.1	The system must transport a passenger from a starting location to their designated location.	1
R.F.2	The system must be able to increase and lower the number of active cars.	1
R.F.3	The system must transport passengers with different tiers of premium[1].	1
R.F.4	The system must be able to pick up multiple passengers if room for it	2
R.F.5	The system must be able to pick up and drop off passengers in different order	2
R.F.6	The system should be able to redirect its route if an intersection of road shuts down.	3
R.F.7	The system should be able to find the best place to put off a passenger in case the passenger's designated intersection is shut down.	3
R.F.8	The system should minimize each passenger's travel time.	4
R.F.9	The system should minimize each car's travel distance	4
R.F.10	The system could change cars on the road accounting to	4

[1] A higher premium is equal to a smaller car size.

	current demand	
R.F.11	The system should take the length of the roads and the cars currently on the roads into consideration when planning its route	5
R.F.12	The system should take the road speed limit and the speed of cars currently on the road into consideration when planning its route	5
R.F.13	The system should plan based on other cars in the systems plan	6
R.F.14	The system could allow for non-autonomous cars drive around the city	7
R.F.15	The system could keep a distance to other cars	7
R.F.16	The system could react to pedestrians or other objects entering the traffic system and stop before hitting them	8
R.F.17	The distance to other object could change with speed	8

Nonfunctional requirements

The nonfunctional requirements can be found in table 2.

Table 2 - Nonfunctional Requirements

ID	Description
R.NF.1	The system must have cars with room for 1, 4, and 8 passengers
R.NF.2	The system must pick up a passenger within 300 seconds.
R.NF.3	The system must work in a city with a minimum size of 10 Intersections and and 15 Roads
R.NF.4	The system could work between 0 and 100 passengers
R.NF.5	The system could work between 0 and 10 of each car
R.NF.6	The system must be able to send only one type of car out if demand is only for it
R.NF.7	When entering a car a unique passenger should not experience that more than the number passengers equal to the number of seats in the car will be dropped off before themself
R.NF.8	A car could not wait in an intersection for more than 10 cars
R.NF.9	A car should alway have its remaining time be minimized before the

	next destination when arriving at a new intersection
R.NF.10	The system should choose the intersection with the lowest walking distance to the original intersection in case a passengers original destination intersection is not reachable
R.NF.11	The number of car seats on the road should not fall below the number of passengers waiting to be picked up
R.NF.12	A car must not exceed its total number of passenger seats
R.NF.13	A car must be able to transport the number of passengers equal to its number of seats
R.NF.14	There should not be more cars on a road than there is room
R.NF.15	A 1 person car must be 3 meters +- 1cm.
R.NF.16	A 4 person car must be 5 meters +- 1 cm.
R.NF.17	A 8 person car must be 7 meters +- 1 cm.
R.NF.18	A car could make a u-turn on a road
R.NF.19	A car must not exceed the speed limit on it current road
R.NF.20	A car could keep a distance to all objects in front of it to a minimum of 1 meter unless that object enters the cars front within 1 meter or within a distance that makes the car unable to stop within 1 meter
R.NF.21	A car could not hit a pedestrian
R.NF.22	There must be no car driving around without a destination. That destination can be to head out of the city, pick up a passenger, or deliver a passenger
R.NF.23	An intersection could have a maximum of 1 car in it at the time
R.NF.24	A road must have a driving speed limit
R.NF.25	A road could have a average walking speed
R.NF.26	A car must access and leave the city via the city border

System Overview

The system overview is based on the Description from *Description* and requirements from *System Requirements*. The goal of the system overview is to translate the requirements into a model that can be implemented and tested by determining the basic building blocks of the system. These building blocks are independent from the actual implementations and are used to discuss the system structure. The system overview is split into three phases in order to make the translation in a structured process:

1. Classes, Data types, and Operations
2. Domain model
3. Logical view

Classes, Data types, and Operations are identified based on the *Description* and *System Requirements*. This ensures that every element is included before going forward and that the role of each element is known. It also helps to convert the requirements to a domain model as an intermediate step. Each of the identified elements will be defined to ensure that the readers agree to their definition and know how they relate to previous sections.

The domain model is used to ensure that the concept from *Description* is represented correctly. The domain model is of a higher abstraction level than the logical view and is easier to talk with the stakeholder about. The goal is to adjust the system based on feedback from stakeholders to ensure that the right system is being created. The domain model is also the first phase in formalizing the system and makes the transition to a system easier and more structured for the developers.

The logical view begins when an agreement on the domain model has been achieved. The logical view uses the actors and domains to formalize the system into logical blocks that can be adopted by a system designer. This phase defined models, relationships, and how data flows through the system. The phase helps the designers to discuss how to represent the system.

Classes, Data types, and Operations

Classes and data types are often identified from nouns and operations are identified from actions. Classes perform functionality, Data types consist of data without functionality, and Operations are the functionality that classes perform. Each is identified in the following tables.

Classes

The classes can be found in table 3.

Table 3 - Identified Classes from *Description* and *System Requirements*

Name	Definition
Self-driving Control System	The entire System
Route Planner	Plans the Car routes
City	The city that the system control cars in
Car (Autonomous)	The cars that the system controls
Car (Manual)	The cars that the system have no control over
Citizen (Passenger)	The Citizens that the system transports. Have a current location, destination, and a time they have waited.
Citizen (Pedestrian)	The citizens that walk around the city without needing transportation

Data Types

The Data types can be found in table 4.

Table 4 - Identified Data Types from *Description* and *System Requirements*

Name	Definition
City Map	Map of the City's intersections and Roads
Intersection (City Center)	The place where roads meet and where Passengers can be picked up and dropped off
Intersection (City Border)	The same as Intersection (City Center) but one of its Roads lead out of the City.
Road	Connects Intersections. Have a length and speed limit
CarType	Size and price of Autonomous Car
Route	The route that the Car (Autonomous) takes to get to its destination. A sequence of Roads.
Location	The location where a citizen or car is or want to go.

	Defined from Intersections or Roads
Speed	The speed of the object

Operations

The Operations can be found in table 5.

Table 5 - Identified Operations from *Description* and *System Requirements*

Name	Definition
Transport Passengers	Transport passengers in the relevant tier from their current location to their designated location
Plan routes	Plan the routes that the autonomous cars are going to travel. Taking queues, speed, and other cars into consideration
Minimize time	Minimize the transport for each passenger
Minimize distance	Minimize the distance traveled by each car
Pick up multiple passengers	Pick up passengers up to equal the cars tier.
Drop off unordered	Drop off passengers in optimal order not in pick up order
Block Intersection or Road	Intersections can shut down and roads can be blocked so cars cannot drive there
Reroute if blocked	Reroute if an Intersection or Road have been blocked
Drop off if blocked	Drop off passengers at the closest intersection if the designated intersection cannot be reached.
Regulate cars	The system can decrease and increase number of cars on the road
Explore the city	Manual cars drive around an explores the city
Keep distance	To Cars and Citizens in front of the car. More speed will require more distance.
Leave City	A car must leave the city if it does not have a destination
Do not overflow roads	There cannot be more cars on a road than there is room for
Remember the passengers pick up order	R.NF.7

Domain model

The domain model can be found in figure 3 (full scale in *Appendix A*). The domain model is based on the requirements from *System Requirements* and the description from *Description*.

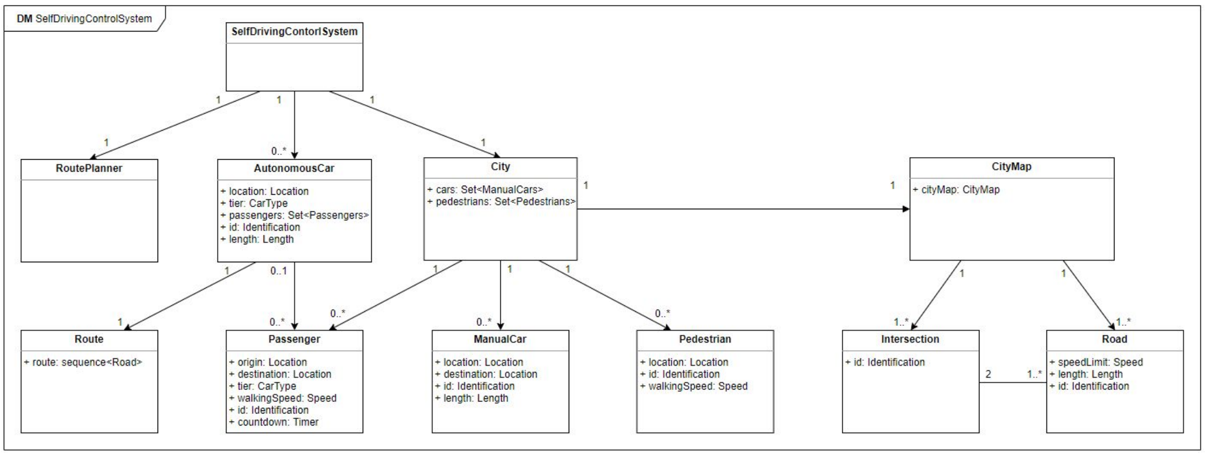


Figure 3 - Domain model of the “Self-driving Control System”

Logical view

The logical view shows a class diagram that is based on the *Domain model* and *Operations* defined in *Classes, Data types, and Operations*. The class diagram can be seen in figure 4. The structure has been changed to better reflect the implementation and names have been changed to better reflect their given purpose. The diagram is simplified and only shows the outlines, a fully detailed diagram can be found in Appendix B. The “SimulationLoader” and “CityMapLoader” are added in order to make testing easier. The architectural considerations are further explained in the next section *VDM Model Structure and descriptions*.

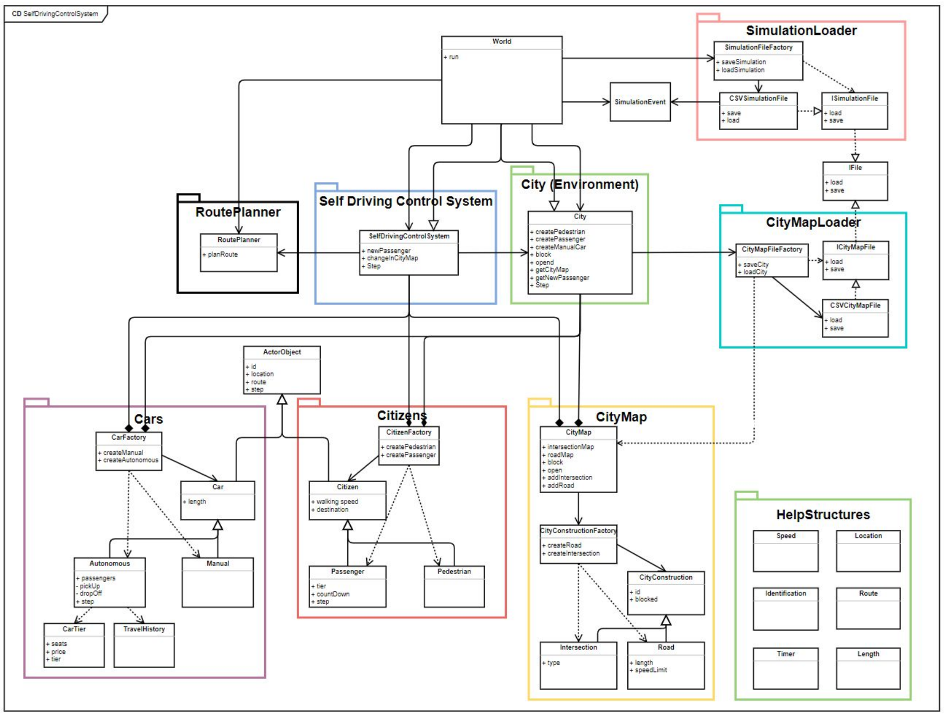


Figure 4 - Class diagram without real time elements

VDM Model structure and descriptions

This section explains how the logical view form figure 4 helps to realise the model. The logical view tries to capture concepts that make the model better and easier to create, both in its modelling capabilities and for further expansion. A system is implemented based on the architecture and concepts from figure 4, this system is shown and explained in *Results*. The following points will be explained in this section:

1. Independent Actors
2. Actor creator
3. Functionality through inheritance
4. Step functions for time simulation
5. Type definitions
6. Loaders
7. Environment vs. World

1. Independent Actors

The system is reflecting a city, in the city there are pedestrians walking around, passengers waiting to be transported, autonomous cars transporting those passengers, and manual cars sightseeing the city. It is the city that has the infrastructure that makes this traffic and life possible by supplying the roads and intersections for actors to use. But it is not only the actors that need to use the infrastructure but also the Self-Driving-Control-system. The system uses knowledge about the city and its actors to plan the routes for its fleet of autonomous cars.

These independent elements (actors, city, and system) need to be represented when modelling the system. The model should thrive to represent the real system in a preferred detail level, the representation in this case is to focus on individuality and separate these through relationships. The different elements can be seen in the system's package diagram in figure 5.

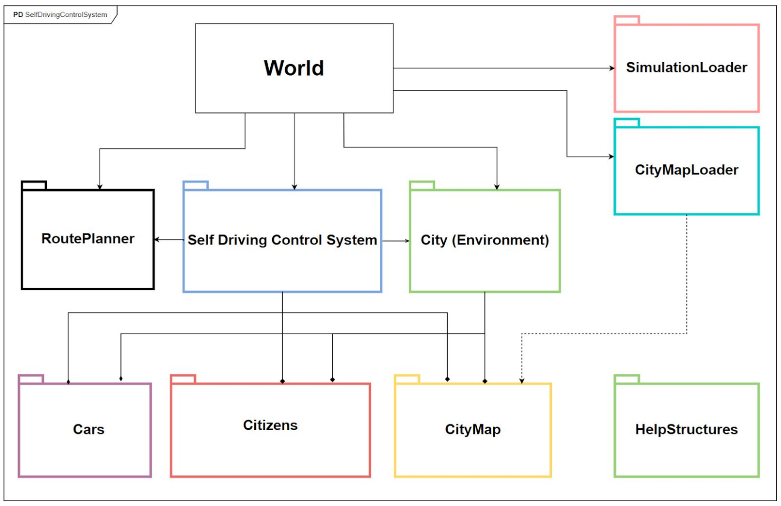


Figure 5 - Package diagram for the self-driving control system

Figure 5 shows the CityMap represented as an individual element, the city has an infrastructure and is not in itself infrastructure. This helps to isolate the map's functionality and increase modelling and modulation capabilities.

2. Actor Creators

Both Cars and Citizens are actors in the city and wish to act independently within the boundaries of the City. The simplified diagram in figure 6 shows that actors are being created with factories in order to reflect the independent behaviour. The factories abstracts the actor implementation and creation away from the caller and modification can be changed without impacting the remaining system. This architecture allows for the model to start as a sequential model and later change to a concurrent implementation by simply changing how the factories create the actors. At the same time, the abstraction of the actor types makes it possible to easier populate the city with different types of cars, citizens or even new types of actors.

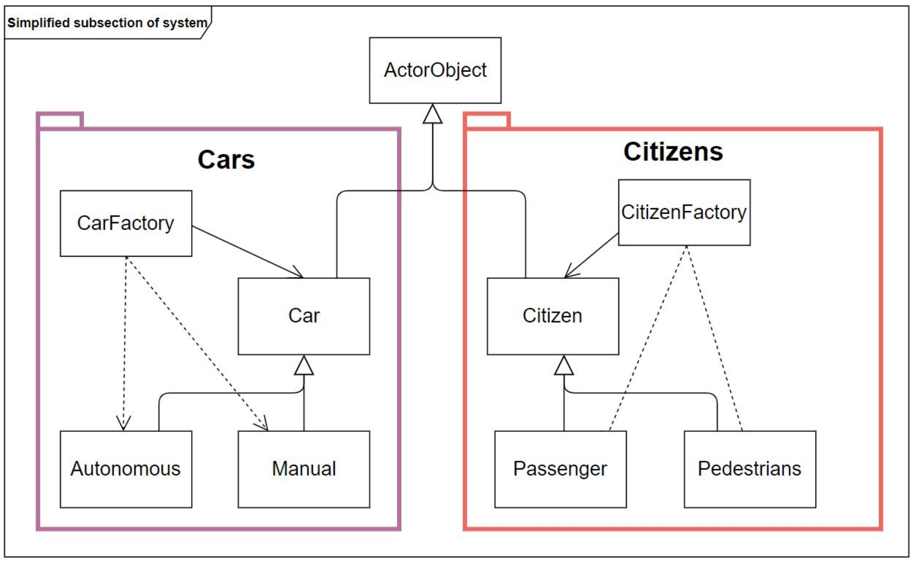


Figure 6 - Simplified version of the Car and Citizen modules

3. Functionality through inheritance

Figure 6 is used to explain how inheritance and polymorphism is used to make modelling and further extensions easier. A general behaviour of actors can be expressed by having the ActorObject as the base class. This functionality can be to specify that every actor must be uniquely identified, have a location, and a route to follow. Remember this is a model and the route is used to specify how the actor should perform a specified behaviour.
The ActorObject has the base functionality, the ones inheriting from the ActorObject will automatically have this behaviour as a default behaviour without any other code. This helps to populate the model with different kinds of actors and create a rich environment
Cars and Citizens can also specify their own behaviour by overloading the functionality defined in the ActorObject. This makes it possible for The City and System to call all actors the same but still expect different behaviour. If something is wrong with a general behaviour can the ActorObject be changed, if something is wrong with a specialization can that specialization be changed. The specializations can also be extended with new functions, for each level of inheritance can new core functionality be added and a new part of the system can be dedicated to handle those types of Actors. An example is the Car that must have a Length or the Autonomous Car that must have Passengers and of a specific tier.

4. Step Functions for time simulation

The system is not a static but instead a dynamic system with interactions and movements. It is therefore important that the model can represent such changes. The system changes with time, time makes the cars move, pedestrians walk, and passengers' patience run out. Each independent element will ideally run on individual threads and be controlled by a shared time indicator. Threads come with the problem that it makes models harder to correct and improve. This is also the reason why the RT-method in figure 1 recommends starting from a more sequential approach.

The step operations in the class diagram from figure 8 is what makes the sequential model possible. The step takes a time difference as an input argument and uses that difference to calculate what would happen in the given period. The smaller the time gets the closer the model will be to simulating a real time system. The larger it is the less processing power is required to execute the model. The model can this way be changed according to model user's needs

The idea behind the architecture is to have the two high level classes "SelfDrivingControlSystem" and "City" step functions be called by the World with a specified time difference. Between each step is it possible for the world to change the environment by e.g. adding new passengers or manual cars to the City. Next time City takes a step will the new changes be used. The City runs through its sets of pedestrians and manual cars and calls their step operations. If not anything else has been specified will the ActorObject's step operation be used, it is possible for each actor to specify their own behaviour. This is also seen in figure 8 where Autonomous, Passenger, and Pedestrian either changes the behaviour or extends it. A change can be seen in Passenger, Passengers do not move but instead count down their patience. An extension is found in Autonomous cars which uses the ActorObject step operation but extends it by picking up or dropping off passengers. Each Passenger and Autonomous car is called by the SelfDrivingConstrolSystem.

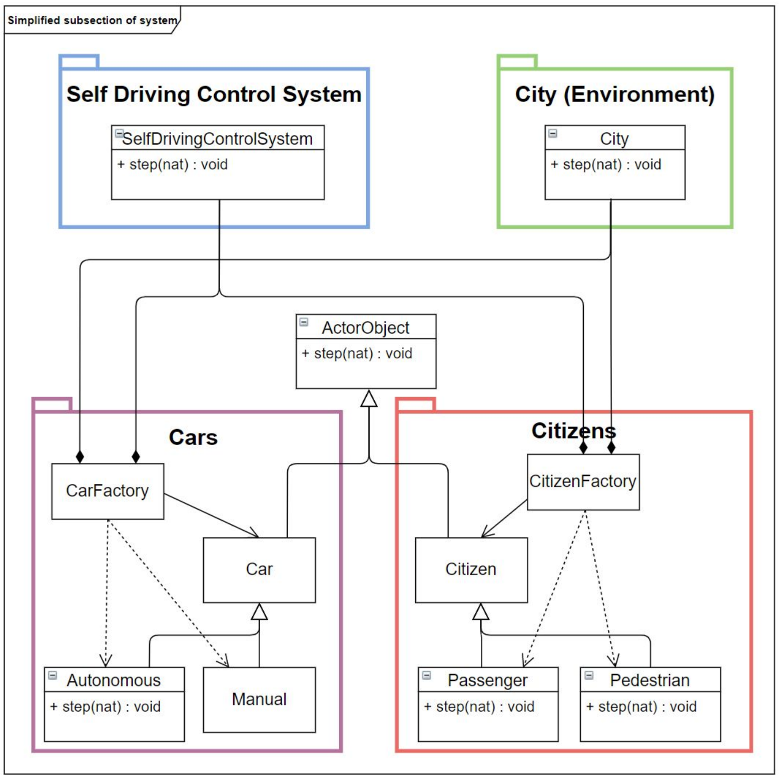


Figure 8 - Simplified diagram with step functions specified

5. Type definitions

Types are used to make change easier across the system. It is not known whether a type may change in the future and it can therefore be a good idea to prepare for it. The preparation is done by type definition. The type definitions can either be defined in a global context or in the representative class that it origins from. The simplified diagram on figure 9 Shows all the types within the classes they origin from. It is recommended to keep types in the original class if they are specially relevant for that class, if it is a more general type as "string" can it be a better idea to define it Globally. A good idea can be to keep it in the class in order to make the type explicit and easier to understand. Change will also be easier because the origin place is known and the direct effects of the change can be more clearly understood. A full diagram can be found in Appendix C.

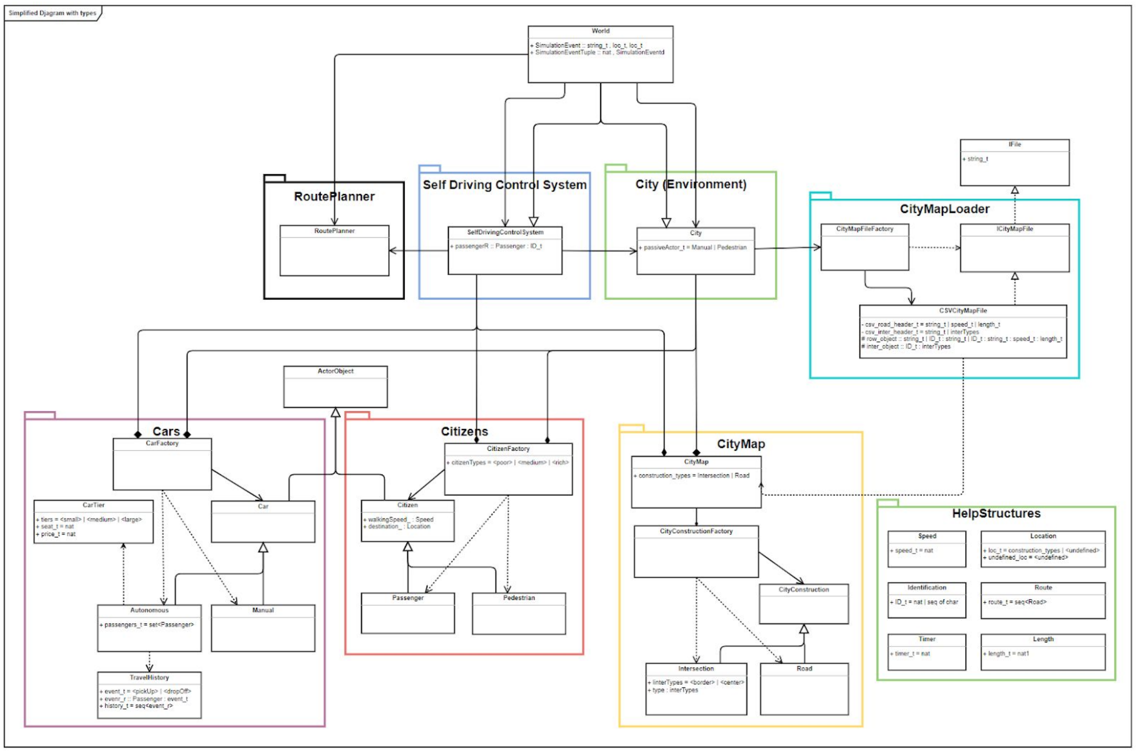

Simplified Djagram with types
World
+ SimulationEvent :: string_t , loc_t, loc_t
+ SimulationEventTuple :: nat , SimulationEventd
RoutePlanner
RoutePlanner
Self Driving Control System
SelfDrivingControlSystem
+ passengerR :: Passenger : ID_t
City (Environment)
City
+ passiveActor_t = Manual | Pedestrian
IFile
+ string_t
CityMapLoader
CityMapFileFactory
ICityMapFile
CSVCityMapFile
- csv_road_header_t = string_t | speed_t | length_t
- csv_inter_header_t = string_t | interTypes
# row_object :: string_t | ID_t : string_t | ID_t : string_t : speed_t : length_t
# inter_object :: ID_t : interTypes
ActorObject
Cars
CarFactory
CarTier
+ tiers = <small> | <medium> | <large>
+ seat_t = nat
+ price_t = nat
Car
Autonomous
+ passengers_t = set<Passenger>
Manual
TravelHistory
+ event_t = <pickUp> | <dropOff>
+ evenr_r :: Passenger : event_t
+ history_t = seq<event_r>
Citizens
CitizenFactory
+ citizenTypes = <poor> | <medium> | <rich>
Citizen
+ walkingSpeed_ : Speed
+ destination_ : Location
Passenger
Pedestrian
CityMap
CityMap
+ construction_types = Intersection | Road
CityConstructionFactory
CityConstruction
Intersection
+ linterTypes = <border> | <center>
+ type : interTypes
Road
HelpStructures
Speed
+ speed_t = nat
Location
+ loc_t = construction_types | <undefined>
+ undefined_loc = <undefined>
Identification
+ ID_t = nat | seq of char
Route
+ route_t = seq<Road>
Timer
+ timer_t = nat
Length
+ length_t = nat1

Figure 9 - Simplified diagram with types defined

6. Loaders

There are two ways that a CityMap can be defined in the system. One way is to define it directly in the code, where the code will be loaded and used. This approach makes each version of the CityMap highly coupled with the CityMap class and thus makes it hard to make changes and extensions to the class. This will make modelling of different CityMaps harder and the exploration of different environments will be limited. The second approach tackles this problem by decoupling the definition of a CityMap from the implementation of it. By making another representation of the CityMap (in this case CSV rows) can an adapter be used between the two representations and thus make it possible to change the CityMap class without more extensive intervention. CSV files are used in order to not only decouple the code but also to make the generation of new maps easier. To make a new CityMap a CSV file is created and the address is defined in the code. The Loader will then load the file and an adapter will convert it to CityMap instructions.

The two loaders planned for the system can be seen in figure 10 and are the “SimulationLoader” and the “CityMapLoader”. “SimulationLoader” uses the same ideas as for the “CityMapLoader” by loading the events and adapting it to a format that the world can use to model a specific behaviour. The purpose of the SimulationLoader is to define what should happen through the simulation. The simulation in this case is when new Citizens, Cars, or events should happen in the City, to observe how the system handles the cases. The event should be defined by a start time for its introduction so that the World knows what to set up in between each of the step operations.

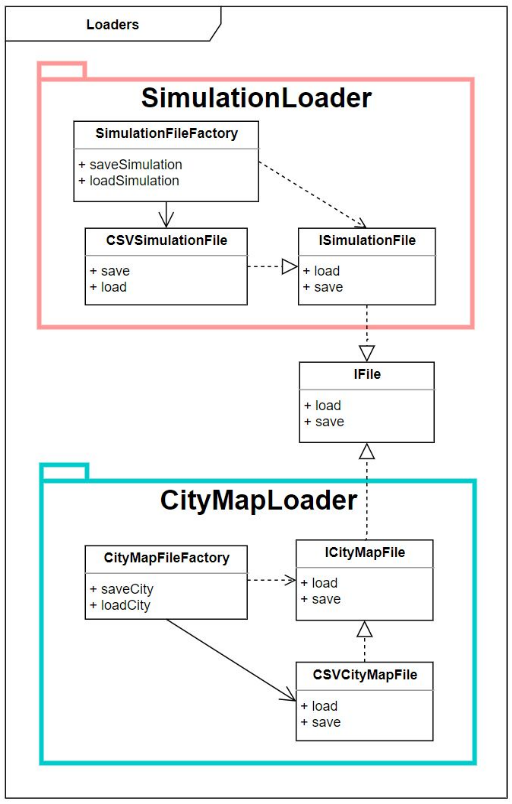


Figure 10 - Loaders for the system

7. Environment Vs. World

It is important to separate the idea of an environment and a world. The environment is dependent on time but does not in itself control it. The environment emulates everything that is not the system and provides an environment for the system to interact with. The system can be dependent on the environment and the environment can be dependent on the system but none of them knows of the World. The world is used to control the simulation by setting up the system and environment, the world controls time and determines what should be set up between each time step. The reason why the world can be seen to inherit from the two classes in figure 4 and 9 is so it can access protected operations. Protected operations are the ones that set up the system or represent concurrent calls (e.g. notifier), this allows for the world to set up the environment and system in a way that protects them from other modules in the model.

Results

The previous section "*VDM model structure and descriptions"* focuses on presenting and explaining the architecture. The architecture is decoupled from the actual implementation and simply ensures that the concepts are well thought out before the implementation begins. This section goes into details about the implementation and shows how the different concepts are realised. The realisation will both focus on the implementation of concepts presented in the previous section and how the VDM specific syntax is used to model the system. This section is split up into the following subsections:

- Implementation
- Tests
- Execution

The tests are used to show which parts of the system works. The tests both help to prove the system and to make the programmer trust their model. By trusting the model will changes be more likely and thus a more ideal model can arise.
The execution shows if the model can run and if the system can handle the task given to it.
All implementation can be found in the appended source code.

Implementation

A selected number of code sections has been chosen for the implementation review. These sections are chosen to give an insight into some of the different concepts used, not all concepts are shown and it is recommended to look into the code to gain a clearer understanding of the system. The following code sections will be looking into:

1. World setup
2. Car factory
3. CityMap
4. CityMap Loader
5. Step operations
6. Pre- and postconditions

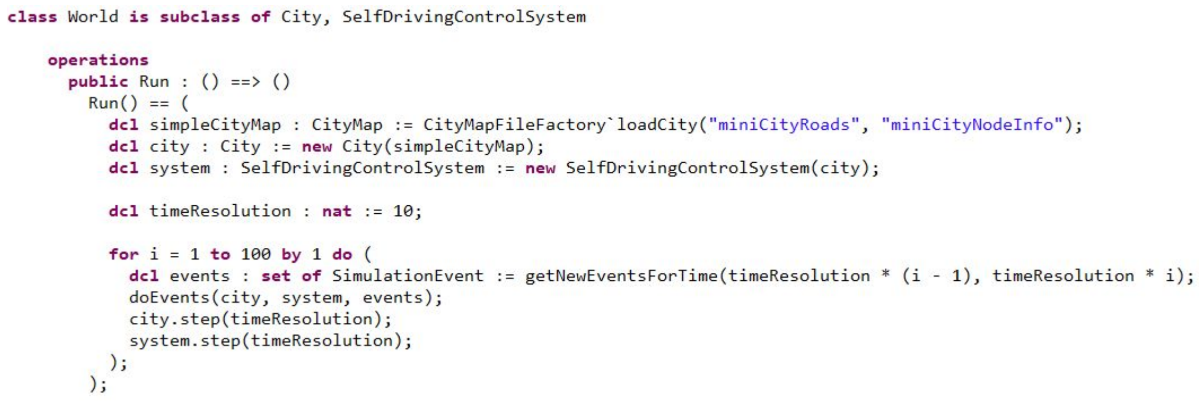
1. World Setup

The top part of world is shown in figure 11. The figure shows that the World inheritage from both the environment (City) and the system (SelfDrivingControlSystem). This allows it to access the two objects protected functions and provides a way to set them up between each step. The world's responsibility can be split up into 5 major parts:

1. Setup Environment
2. Setup System
3. Step through time
    a. Get new event for time interval
    b. Setup environment with new events
    c. Call Step with time

The world has a Run operation, Run is the operation executed by the program and will both handle the model setup and the step execution. In the setup process is a city map being

loaded in as two CSV files, one file for the roads and one for the intersections. These two together will define the whole CityMap. The City can now be set up with the CityMap and afterwards be passed into the system. City is required as an argument to the system so that it can fetch the CityMap or new Passengers when notified.
The model can be run after the system has been set up. The time resolution is set to 10 seconds per jump and it will continue until a 1000 second has passed in the model. Both numbers can be changed, the closer the timeResolution is to 0 the closer it represents a real system but will as a effect take a longer time.



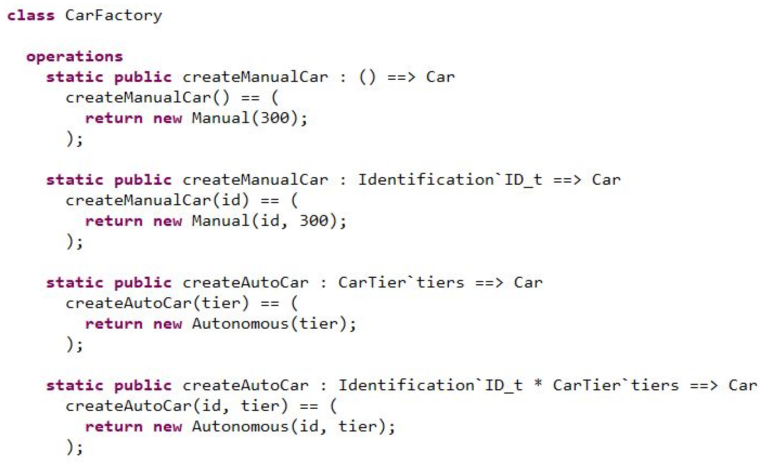
```
class World is subclass of City, SelfDrivingControlSystem

  operations
    public Run : () ==> ()
      Run() == (
        dcl simpleCityMap : CityMap := CityMapFileFactory`loadCity("miniCityRoads", "miniCityNodeInfo");
        dcl city : City := new City(simpleCityMap);
        dcl system : SelfDrivingControlSystem := new SelfDrivingControlSystem(city);

        dcl timeResolution : nat := 10;

        for i = 1 to 100 by 1 do (
          dcl events : set of SimulationEvent := getNewEventsForTime(timeResolution * (i - 1), timeResolution * i);
          doEvents(city, system, events);
          city.step(timeResolution);
          system.step(timeResolution);
        );
      );
```

Figure 11 - Code from the stop part of the world class

The system will in the stepping process look for new events to be activated. The event is defined with an activation time, the event is chosen if it is in the current step's time interval. The World changes the environment based on the chosen events. After the new events have been added to the environment will the next step be activated and the model can continue.

2. Car Factory

The goal of the car factory is to create cars. The top part of the CarFactory class can be seen in figure 12. The figure shows that all factory operations are static, a factory should not depend on a specific instance but only work as a creator. The operations shown are creating either a manual or automatic car. The Cars can be created with an ID_t but are not required to. The ID_t is for test purposes as to see if a specific car is having the correct behaviour. The ID_t is also the first sign of the types shown in figure 9.



```
class CarFactory

  operations
    static public createManualCar : () ==> Car
      createManualCar() == (
        return new Manual(300);
      );

    static public createManualCar : Identification`ID_t ==> Car
      createManualCar(id) == (
        return new Manual(id, 300);
      );

    static public createAutoCar : CarTier`tiers ==> Car
      createAutoCar(tier) == (
        return new Autonomous(tier);
      );

    static public createAutoCar : Identification`ID_t * CarTier`tiers ==> Car
      createAutoCar(id, tier) == (
        return new Autonomous(id, tier);
      );
```

Figure 12 - Code from the top part of the CarFactory class

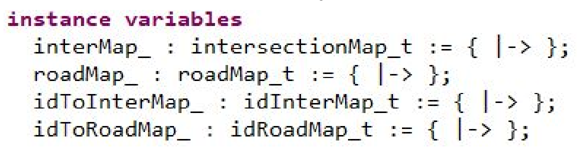
Tiers are the second sign of the types. Tier is an enum and should be specified as:

- <poor>
- <medium>
- <large>

More enums can easily be added across the system simply by changing it in the CarTier class. The automatic cars can be created as one of these tier types and an instance of Autonomous is returned to the caller. This class would be changed to a thread creator if the system were to be changed to concurrent.

3. CityMap

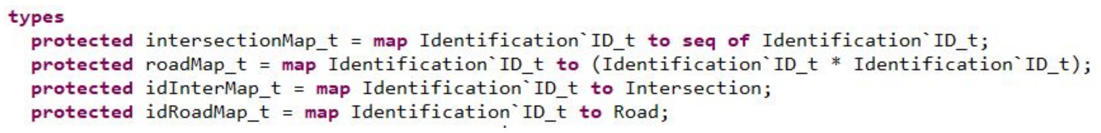
The CityMap consists of four instance variables, the variables can be found in figure 13. Each of the four is a map and their types have been abstracted away. Again, this is to make it easier to change system-wide or atleast class-wide in case it is required. The four types can be found in figure 14 and can be seen to only be relevant class-wide.



```
instance variables
  interMap_ : intersectionMap_t := { |-> };
  roadMap_ : roadMap_t := { |-> };
  idToInterMap_ : idInterMap_t := { |-> };
  idToRoadMap_ : idRoadMap_t := { |-> };
```

Figure 13 - The CityMap's four instance variables

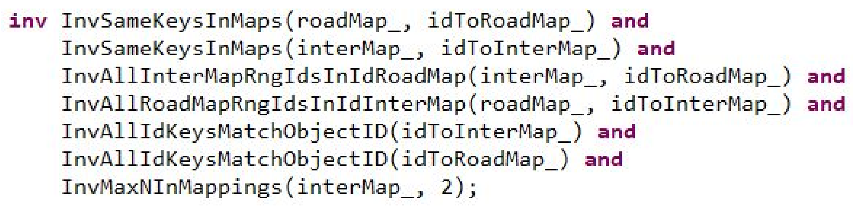
The four instance maps are made in order to create a graph[15]. A graph consists of nodes and connections, connections connect nodes and defines the length that there is between them. The intersections are nodes and the roads are connections. To express a graph will the first map "interMap_" have all Intersections in it, each intersection points to a sequence of roads that leads from it. The reason a sequence is used is because a road can be connected to the same intersection two times. The second map "roadMap_" is a map having roads that points to a tuple of Intersections, the intersections in the tuple are the ones that are at each end of the road.
As figure 14 shows is the roads and intersections not used in these two maps. The reason is that numbers are easier to use and the identification type is therefore used in their stead. The Intersections are then saved in "idToInterMap_" where an id points to the Intersection object, the same is implemented for the roads in "idToRoadMap_".



```
types
  protected intersectionMap_t = map Identification`ID_t to seq of Identification`ID_t;
  protected roadMap_t = map Identification`ID_t to (Identification`ID_t * Identification`ID_t);
  protected idInterMap_t = map Identification`ID_t to Intersection;
  protected idRoadMap_t = map Identification`ID_t to Road;
```

Figure 14 - Four of CityMap's types

The CityMap is bound by a series of Invariants, invariants are rules that must always hold true for the instance. The invariants will be checked after each line and not just after an operation or function is executed. The invariants defined for CityMap can be found in figure 15.



```
inv InvSameKeysInMaps(roadMap_, idToRoadMap_) and
    InvSameKeysInMaps(interMap_, idToInterMap_) and
    InvAllInterMapRngIdsInIdRoadMap(interMap_, idToRoadMap_) and
    InvAllRoadMapRngIdsInIdInterMap(roadMap_, idToInterMap_) and
    InvAllIdKeysMatchObjectID(idToInterMap_) and
    InvAllIdKeysMatchObjectID(idToRoadMap_) and
    InvMaxNInMappings(interMap_, 2);
```

Figure 15 - Invariants for the CityMap class

The invariants checks:

- **InvSameKeysInMaps** - Check if all ID_t's in the id graph maps are also in their representative id to object maps.
- **InvAllInterMapRngIdsInIdRoadMap** - Checks if all the roads in the range of the inter map exists
- **InvAllRoadMapRngIdsInIdInterMap** - Check if all the intersections in the range of the road map exists
- **InvAllIdKeysMatchObjectId** - Check if the key and the object it points to matches.
- **InvMaxNInMaps** - Check if there is a maximum of 2 identical roads in the intersections' road sequences. A road only has two ends and can therefore only connect two times.

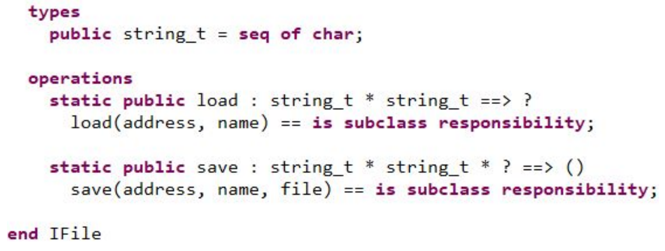
The interaction with the CityMap happens through the following operations:

+ addIntersection(Intersection) : void
+ addRoad(Road, Intersection, Intersection) : void
+ getSetOfInter() : set<Intersection>
+ getSetOfRoad() : set<Road>
+ getSeqOfRoad(Intersection) : seq<Road>
+ getSeqOfActiveRoad(Intersection) : seq<Road>
+ getSeqOfActiveRoad(ID_t) : seq<Road>
*+ getInterTupleFromRoad(Road) : (Intersection * Intersection)*
+ block(construction_types) : void
+ open(construction_types) : void
+ getRoad(ID_t) : Road
+ getRoad(Road) : Road

In short, the user can add Intersections and roads to the CityMap, get a set of either all the intersections or all the roads, and get the roads connected to an intersection or the intersections connected to a road. To accommodate R.F.6, roads/intersections can be opened/closed and it is possible to get only the active roads leading from an intersection.
This setup should make it possible to make the CityMap as a graph in an intuitive way. The graphs have been specialised to reflect the graph properties defined in the requirements. Protected help functions and operations are defined but not shown. These are used to properly abstract the operations in a way that is easier to read and extend. This extra effort makes further development of the model easier and more likely.

4. CityMap Loader

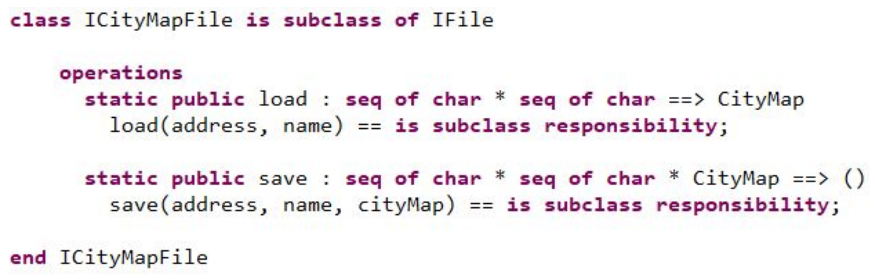
The CityMap loader is a realization of the IFile interface which can be found in figure 16. The interface ensures that realizations have implemented the load and save functions by using the "is subclass responsibility" syntax. This is also what makes it possible for the CityMapFileFactory to decouple the CSVCityMap from the remaining code. The interface also defines the string_t that address must be defined as. The interface further makes it up to the realization to specify what kind of return and save object they wish to implement.



```
class IFile
  types
    public string_t = seq of char;

  operations
    static public load : string_t * string_t ==> ?
      load(address, name) == is subclass responsibility;

    static public save : string_t * string_t * ? ==> ()
      save(address, name, file) == is subclass responsibility;

end IFile
```

Figure 16 - The IFile interface

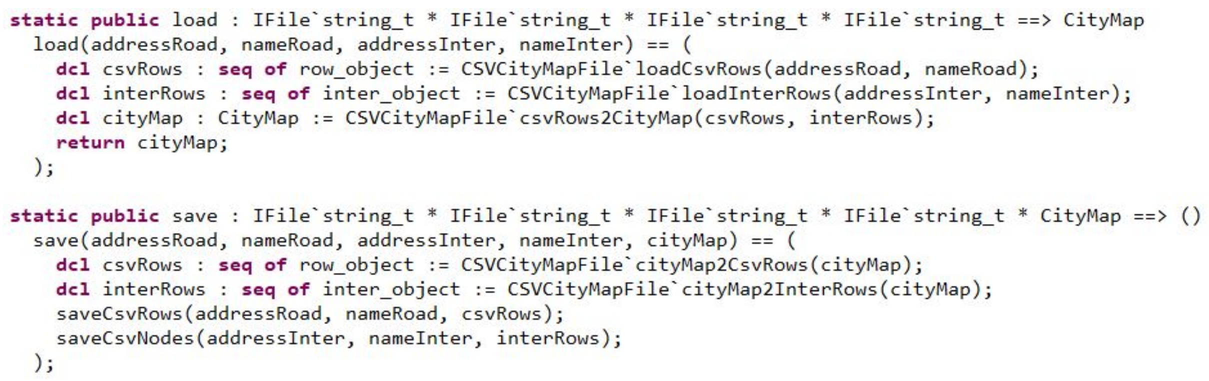
The object type is specified in the ICityMapFile interface, the interface keeps the operations as the subclasses responsibility but makes realizations dependent on returning and saving a CityMap. This makes it easy for users to simply pass on the returned object into the City's initialization. Figure 17 shows the ICityMapFile interface.



```
class ICityMapFile is subclass of IFile

  operations
    static public load : seq of char * seq of char ==> CityMap
      load(address, name) == is subclass responsibility;

    static public save : seq of char * seq of char * CityMap ==> ()
      save(address, name, cityMap) == is subclass responsibility;

end ICityMapFile
```

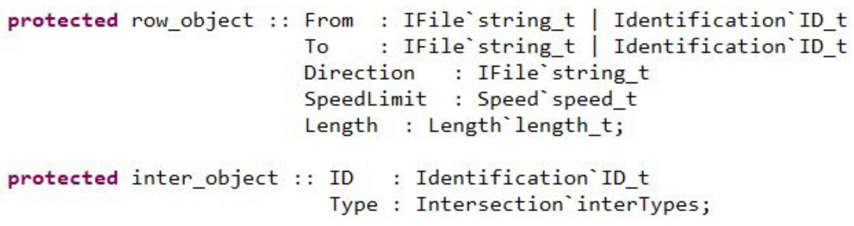
Figure 17 . The ICityMapFile interface

Next comes the CSVCityMapFile class that is the final realization of the interface. This could also have been an TXT adapter or simply be put there as code. But an CSV implementation is chosen to show how to use the CSV library.
The class is implementing the two interface operations, see figure 18. The two operations are kept simple. The loader extracts the CSV data and loads it into the row_object record and the inter_object record, these records can then be used to populate the CityMap. The records are shown in figure 19. The savers can then convert a CityMap into a row_object and inter_object and the two record types can each be saved.



```
static public load : IFile`string_t * IFile`string_t * IFile`string_t * IFile`string_t ==> CityMap
  load(addressRoad, nameRoad, addressInter, nameInter) == (
    dcl csvRows : seq of row_object := CSVCityMapFile`loadCsvRows(addressRoad, nameRoad);
    dcl interRows : seq of inter_object := CSVCityMapFile`loadInterRows(addressInter, nameInter);
    dcl cityMap : CityMap := CSVCityMapFile`csvRows2CityMap(csvRows, interRows);
    return cityMap;
  );

static public save : IFile`string_t * IFile`string_t * IFile`string_t * IFile`string_t * CityMap ==> ()
  save(addressRoad, nameRoad, addressInter, nameInter, cityMap) == (
    dcl csvRows : seq of row_object := CSVCityMapFile`cityMap2CsvRows(cityMap);
    dcl interRows : seq of inter_object := CSVCityMapFile`cityMap2InterRows(cityMap);
    saveCsvRows(addressRoad, nameRoad, csvRows);
    saveCsvNodes(addressInter, nameInter, interRows);
  );
```

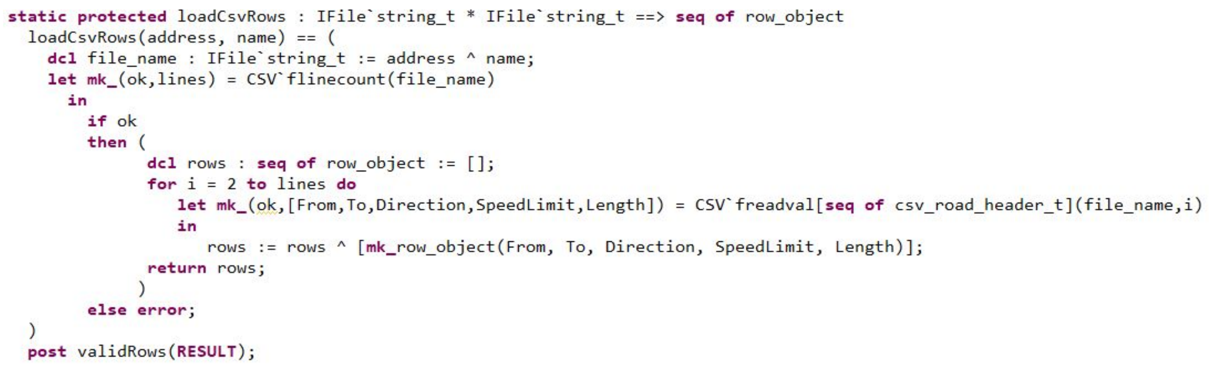
Figure 18 - CSVCityMapFile's load and save realizations



```
protected row_object :: From   : IFile`string_t | Identification`ID_t
                        To     : IFile`string_t | Identification`ID_t
                        Direction   : IFile`string_t
                        SpeedLimit  : Speed`speed_t
                        Length  : Length`length_t;

protected inter_object :: ID   : Identification`ID_t
                          Type : Intersection`interTypes;
```

Figure 19 - The two record types used to load and save CityMaps

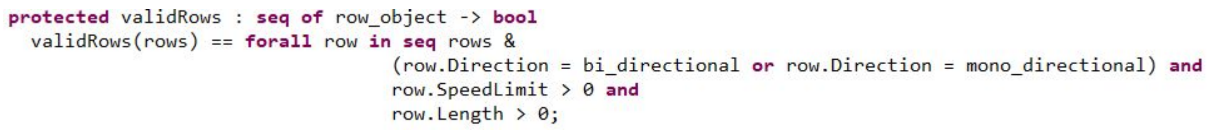
The csv-loader loadCsvRows is shown in figure 20. The loader tries to count the number of lines in the file. The count will both be used to check if the file can be read (e.g. exists) and to step through the file. A sequence of row_objects is created if the file can be read, each row is saved in this sequence and later returned. The file starts from line two because the first line is a header. Each line read by CSV`freadval is divided into columns (separated by comma) and converted into an object. Each column is given a name that afterwards can be used to add it to the row sequence.



```
static protected loadCsvRows : IFile`string_t * IFile`string_t ==> seq of row_object
  loadCsvRows(address, name) == (
    dcl file_name : IFile`string_t := address ^ name;
    let mk_(ok,lines) = CSV`flinecount(file_name)
      in
        if ok
        then (
              dcl rows : seq of row_object := [];
              for i = 2 to lines do
                 let mk_(ok,[From,To,Direction,SpeedLimit,Length]) = CSV`freadval[seq of csv_road_header_t](file_name,i)
                 in
                    rows := rows ^ [mk_row_object(From, To, Direction, SpeedLimit, Length)];
              return rows;
             )
        else error;
  )
  post validRows(RESULT);
```

Figure 20 - The csv-loader for the row_object

The post condition is a property that must hold true each time the operation finishes. This post condition is called “validRows” and goes through each row of the returned sequence to check if they are valid. The post-conditions must be defined as functions and work like static const functions from traditional C++ programming. “ValidRows” can be found in figure 21.



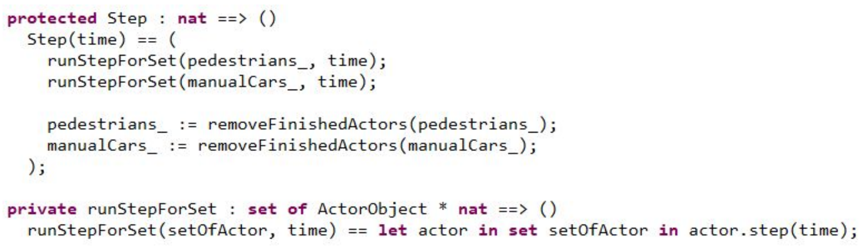
```
protected validRows : seq of row_object -> bool
  validRows(rows) == forall row in seq rows &
                            (row.Direction = bi_directional or row.Direction = mono_directional) and
                            row.SpeedLimit > 0 and
                            row.Length > 0;
```

Figure 21 - post condition for loadCsvRows and checks if the rows are valid

The function must take a sequence of row_object and thus defines the first property that must hold true (being of this type). The second property is the “forall” keyword that tells the function that all elements in the sequence must follow this scheme defined afterwards. An “exists” could be used if only a single element was required to be true. “&” defined a “such that” and the first line should be read as “For all rows in a sequence of rows it must be such that …”. The next three lines represent three statements that all must be true in order to pass the check, this is defined by the “and” keyword. The first of the checks is split up by the “or” keyword and requires that the Direction value is one of the two types (bi_directional or mono_directional). The two next lines check if both the speed limit and length is above 0.
More functions and operations are defined to help with the conversion but are not shown.

5. Step operations

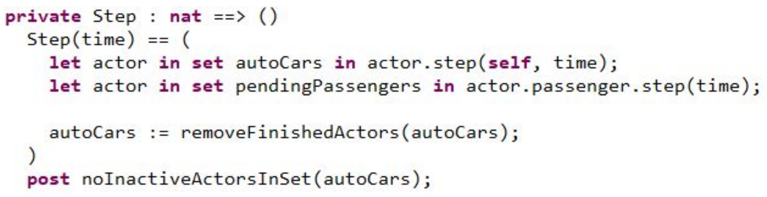
The step operations have already been shown for the World where it calls the City’s and system’s step operation. The implementation of the two can be found in figure 22 and 23. Figure 22 shows the City’s step operation which calls each of the pedestrians and manual cars step operation. The actors will afterwards be removed from the environment if they have finished their role. It is checked whether all actors that have finished also is removed from their representative sets as a post-condition . The step function for the pedestrian and manual cars is explained after SelfDrivingControlSystem is explained.



```
protected Step : nat ==> ()
  Step(time) == (
    runStepForSet(pedestrians_, time);
    runStepForSet(manualCars_, time);

    pedestrians_ := removeFinishedActors(pedestrians_);
    manualCars_ := removeFinishedActors(manualCars_);
  );

private runStepForSet : set of ActorObject * nat ==> ()
  runStepForSet(setOfActor, time) == let actor in set setOfActor in actor.step(time);
```

Figure 22 - Step operation for City

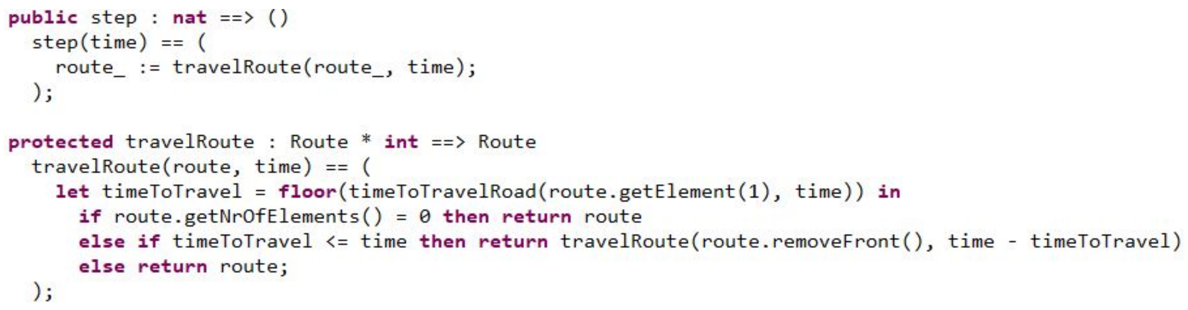
The step operation for SelfDrivingControlSystem works the same way, it runs the cars' and Passengers' step operation and removes them afterwards if they have finished. The Passengers will throw an internal invariance if their patience hits 0 and thus does not need to be checked here. They do not need to be removed either, cars will remove passengers that they have picked up and it should therefore be the fetch operation that makes this check.



```
private Step : nat ==> ()
  Step(time) == (
    let actor in set autoCars in actor.step(self, time);
    let actor in set pendingPassengers in actor.passenger.step(time);

    autoCars := removeFinishedActors(autoCars);
  )
  post noInactiveActorsInSet(autoCars);
```

Figure 23 - Step operation for SelvDrivingControlSystem

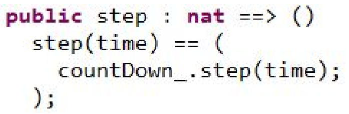
The step operation for Pedestrians, Passengers, manual cars, and autonomous cars have all roots in the ActorObject and if nothing else is defined will it be this one that is called. The ActorObject's step operation is shown in figure 24. This operation uses the actor's route and the speedLimit of the road it is currently on. It will check if it can finish the road within the given time interval and afterwards call itself without the road just passed. This continues recursively until no more time is left or no more roads have been defined.



```
public step : nat ==> ()
  step(time) == (
    route_ := travelRoute(route_, time);
  );

protected travelRoute : Route * int ==> Route
  travelRoute(route, time) == (
    let timeToTravel = floor(timeToTravelRoad(route.getElement(1), time)) in
      if route.getNrOfElements() = 0 then return route
      else if timeToTravel <= time then return travelRoute(route.removeFront(), time - timeToTravel)
      else return route;
  );
```

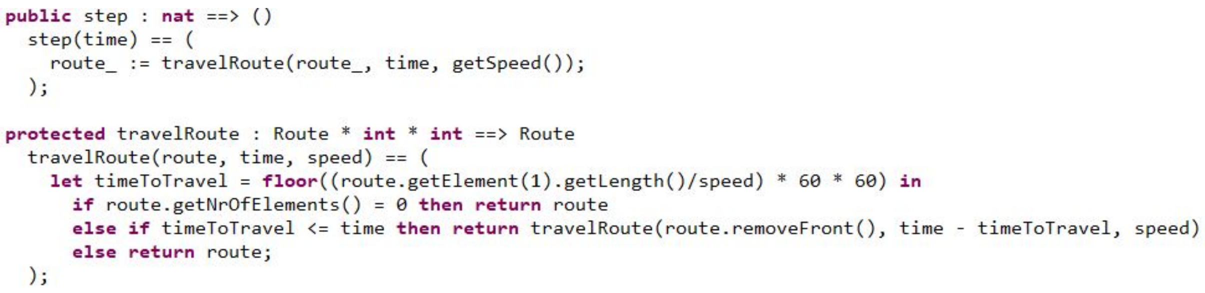
Figure 24 - Step function for the ActorObject. The step goes through the route defined for the actor

But the different actors can also define their own step operation. This is shown in figure 25 where the Passenger have changed it to counting down the patience or figure 26 where the Pedestrian are using its walking speed instead of the speed limit in the recursive operation.



```
public step : nat ==> ()
  step(time) == (
    countDown_.step(time);
  );
```

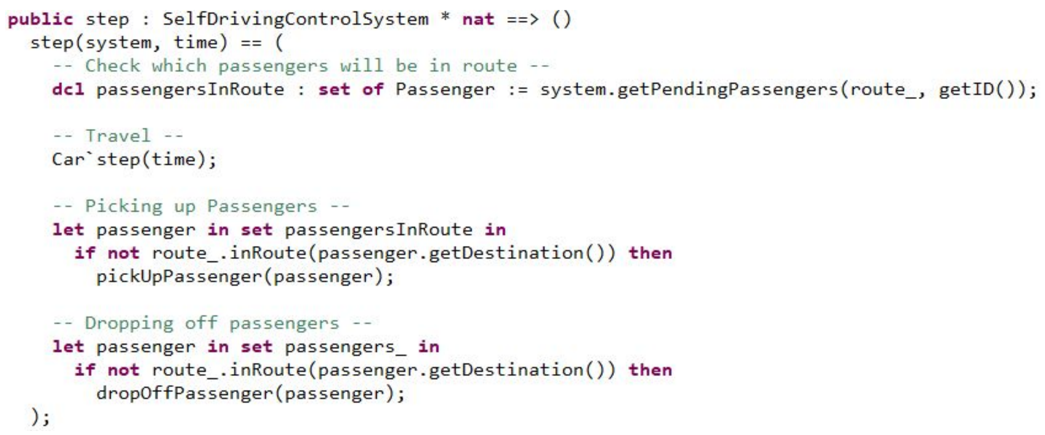
Figure 25 - Step overloading by the Passenger class



```
public step : nat ==> ()
  step(time) == (
    route_ := travelRoute(route_, time, getSpeed());
  );

protected travelRoute : Route * int * int ==> Route
  travelRoute(route, time, speed) == (
    let timeToTravel = floor((route.getElement(1).getLength()/speed) * 60 * 60) in
      if route.getNrOfElements() = 0 then return route
      else if timeToTravel <= time then return travelRoute(route.removeFront(), time - timeToTravel, speed)
      else return route;
  );
```

Figure 26 - Step overloading by the Pedestrian class

Lastly, the Autonomous car overloads the step operations but still uses the ActorObject's step operation. The Autonomous car step operation is shown in figure 27. This first check if any passenger is waiting to be picked up by the current car. The car afterwards travels by calling the Car’s step operation, the car has no overloading of the step operation and it will therefore cascade up and use the ActorObject’s step operation. The car then checks if any passenger should have been picked up at any of the Intersections just been passed. If this is the case will it fetch the passenger from the system and thus remove it from the pendingPassengers set. It then checks if any passengers should have been dropped off.

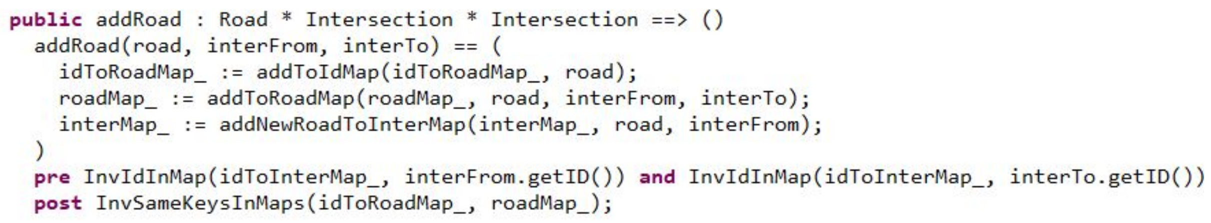


```
public step : SelfDrivingControlSystem * nat ==> ()
  step(system, time) == (
    -- Check which passengers will be in route --
    dcl passengersInRoute : set of Passenger := system.getPendingPassengers(route_, getID());

    -- Travel --
    Car`step(time);

    -- Picking up Passengers --
    let passenger in set passengersInRoute in
      if not route_.inRoute(passenger.getDestination()) then
        pickUpPassenger(passenger);

    -- Dropping off passengers --
    let passenger in set passengers_ in
      if not route_.inRoute(passenger.getDestination()) then
        dropOffPassenger(passenger);
  );
```

Figure 27 - Step function for the Autonomous car class

6. Pre- and post conditions

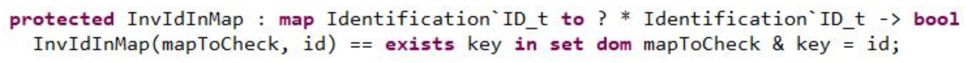
This subsection looks at what properties can be defined in the pre- and post conditions. An example can be found in figure 28 where a road is added to the CityMap. A road is always connected to two intersections so the precondition checks if these two already exist in the cityMap. It does this by using the InvIdInMap function for both of the intersections, this is to reuse already implemented code and to clearly understand which part failed. InvIdInMap can be found in figure 29.



```
public addRoad : Road * Intersection * Intersection ==> ()
  addRoad(road, interFrom, interTo) == (
    idToRoadMap_ := addToIdMap(idToRoadMap_, road);
    roadMap_ := addToRoadMap(roadMap_, road, interFrom, interTo);
    interMap_ := addNewRoadToInterMap(interMap_, road, interFrom);
  )
  pre InvIdInMap(idToInterMap_, interFrom.getID()) and InvIdInMap(idToInterMap_, interTo.getID())
  post InvSameKeysInMaps(idToRoadMap_, roadMap_);
```

Figure 28 - Check if Intersections exists in map as a precondition and check if the road have been added to the map afterwards

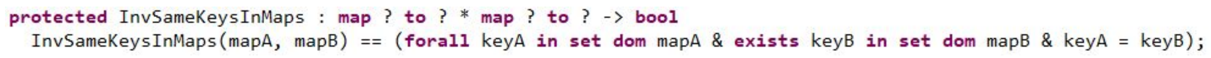
InvIdInMap from figure 29 uses ‘?’ as an input variable. This makes it possible to make any map as long as the keys will be of the type ID_t. The function can then be used for both the idToInterMap_ and idToRoadMap_. The ‘exists’ keyword is used in this function, ‘exists’ defines that only a single key in the set should match. ‘Dom’ is used as a map operation to get a set of all keys in the map (this is called a domain).



```
protected InvIdInMap : map Identification`ID_t to ? * Identification`ID_t -> bool
  InvIdInMap(mapToCheck, id) == exists key in set dom mapToCheck & key = id;
```

Figure 29 - Invariance to check if the id exists in the map

The postcondition from figure 28 checks if all the IDs in idToRoadMap_ also exist in the roadMap. This post condition is required because a road is added in two steps and would otherwise be catched by an invariance between these two. InvSameKeysInMaps is found in figure 30. Both ‘forall’ and ‘exists’ are used in this function and makes it possible to easily check the connection between the two domains.



```
protected InvSameKeysInMaps : map ? to ? * map ? to ? -> bool
  InvSameKeysInMaps(mapA, mapB) == (forall keyA in set dom mapA & exists keyB in set dom mapB & keyA = keyB);
```

Figure 30 - InvSameKeysInMaps to check if the two has the same keys

It is here important to note that both preconditions and postconditions are abstracted into functions. The reason for this is to make it easier to understand what happens and to make the condition testable through unit testing.

Tests

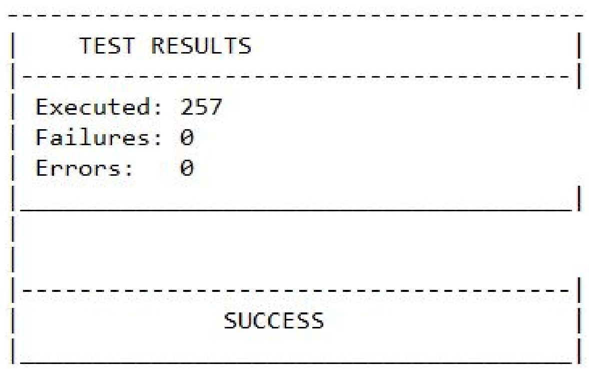
Tests are used to verify the system. The invariants, preconditions and postconditions make up a solid assurance that the system complies to set boundaries and minimizes the set of tests that should be made. But these can not show whether the system works and thus further testing is required. The tests can be splitted up into four levels of testing:

- Combinatorial Testing
- Unit Testing
- Behavioural testing
- Simulation testing

Combinatorial testing is often used to test low level properties and can be especially good to test mathematical functions. This shows that the function works in a given range of cases and thus increases the trust for that function. Combinatorial testing has not been used and low level testing is instead replaced by unit testing.
Unit Testing is used to test the units in the system. Each unit represents a piece of intent in the system which should be met by the functions used in the test case. Each unit test only tests one perspective, this makes it possible to identify exactly why and where an error has occurred. Tests should ideally be written before the code, this makes it easier to make good model abstraction and functions that capture intent instead of functionality. Most parts of the code have been covered by unit tests.
Behavioural testing test behaviour in the code. This is a step above the intent captured in the unit tests. The behaviour focuses more on use cases and flows through the system. The behavioural tests are also implemented as unit tests but the idea behind them is separated from the traditional unit testing. Few behavioural tests have been implemented and those that have is a part of the unit test suit. The system is not fully implemented and the project therefore never came to the point of behavioral testing.
Simulation testing simulates the system. These tests tried out the model with different settings to check its robustness and what combinations that can break it. No Simulation testing has been implemented because the full system is not fully implemented.

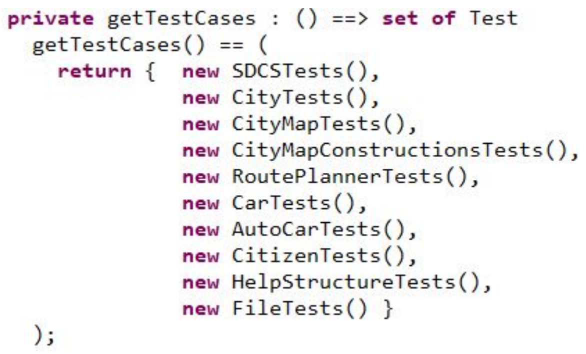
Unit Tests

There are too many unit tests to show, but all 257 tests have been passed. Figure 31 shows the tests successfully passed. The tests try to capture the intent of each class, function and operation, to test if they work on a low level of implementation. These tests do not test whether the system is performing well but simply if it can perform what it has been tasked with. The test suit can be seen in figure 32 where it shows the different parts that are being tested.



```
----------------------------------------
|    TEST RESULTS                      |
|--------------------------------------|
| Executed: 257
| Failures: 0
| Errors:   0
|______________________________________|
|
|
|--------------------------------------|
|               SUCCESS                |
|______________________________________|
```

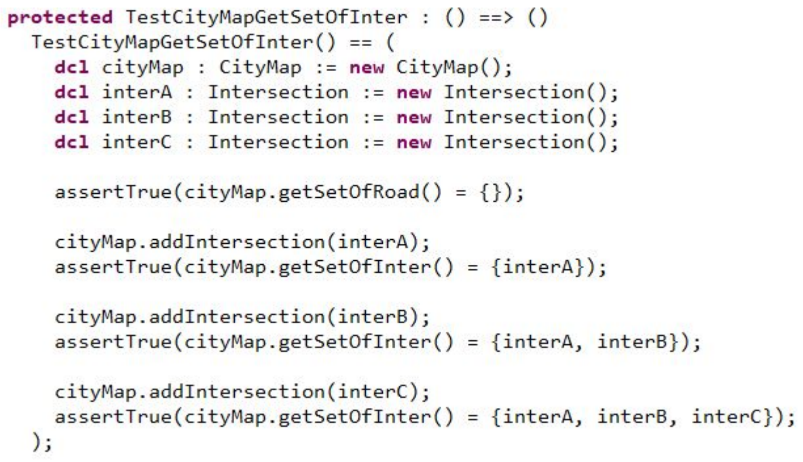
Figure 31 - All 257 unit tests for the system passed



```
private getTestCases : () ==> set of Test
  getTestCases() == (
    return {  new SDCSTests(),
              new CityTests(),
              new CityMapTests(),
              new CityMapConstructionsTests(),
              new RoutePlannerTests(),
              new CarTests(),
              new AutoCarTests(),
              new CitizenTests(),
              new HelpStructureTests(),
              new FileTests() }
  );
```

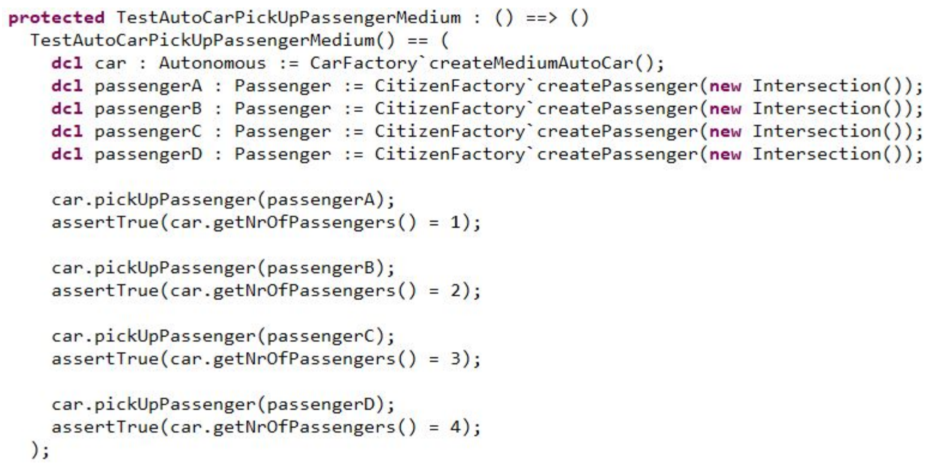
Figure 32 - Test suit for the unit tests

Not all code has been tested with unit tests. City, selfDrivingControlSystem, RoutePlanner and the step operations are not yet tested. Examples of unit tests can be seen in figure 33 where the addition of intersections is checked and in figure 34 where the capacity of a car is checked.



```
protected TestCityMapGetSetOfInter : () ==> ()
  TestCityMapGetSetOfInter() == (
    dcl cityMap : CityMap := new CityMap();
    dcl interA : Intersection := new Intersection();
    dcl interB : Intersection := new Intersection();
    dcl interC : Intersection := new Intersection();

    assertTrue(cityMap.getSetOfRoad() = {});

    cityMap.addIntersection(interA);
    assertTrue(cityMap.getSetOfInter() = {interA});

    cityMap.addIntersection(interB);
    assertTrue(cityMap.getSetOfInter() = {interA, interB});

    cityMap.addIntersection(interC);
    assertTrue(cityMap.getSetOfInter() = {interA, interB, interC});
  );
```

Figure 33 - Unit test that checks if an intersection is added to the map



```
protected TestAutoCarPickUpPassengerMedium : () ==> ()
  TestAutoCarPickUpPassengerMedium() == (
    dcl car : Autonomous := CarFactory`createMediumAutoCar();
    dcl passengerA : Passenger := CitizenFactory`createPassenger(new Intersection());
    dcl passengerB : Passenger := CitizenFactory`createPassenger(new Intersection());
    dcl passengerC : Passenger := CitizenFactory`createPassenger(new Intersection());
    dcl passengerD : Passenger := CitizenFactory`createPassenger(new Intersection());

    car.pickUpPassenger(passengerA);
    assertTrue(car.getNrOfPassengers() = 1);

    car.pickUpPassenger(passengerB);
    assertTrue(car.getNrOfPassengers() = 2);

    car.pickUpPassenger(passengerC);
    assertTrue(car.getNrOfPassengers() = 3);

    car.pickUpPassenger(passengerD);
    assertTrue(car.getNrOfPassengers() = 4);
  );
```

Figure 34 - Unit test that checks if the cars can pick up passengers equal to its capacity.

The code covered by the tests can be found in “SelfDrivingControlSystem -> Generated -> 2020_12_08_??_??_??” In the appended code project.

It should be mentioned that most functions and operations are either defined as public or protected in the different classes. Protected is used to make these accessible from the test cases testing them. The test case can inherit from the class and thus gain access to internal properties.

Execution

The CityMap used for the execution can be found in figure 35. The map consists of 10 border Intersections, 22 center Intersections, and 37 roads. It is possible to load the city map from a csv file, this capability is shown in figure 36 by a test case. This test also shows that the map can be added to the City and that the city can be added to the System.

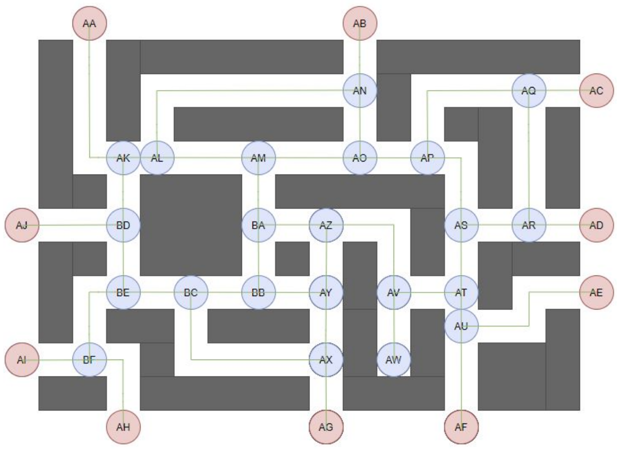
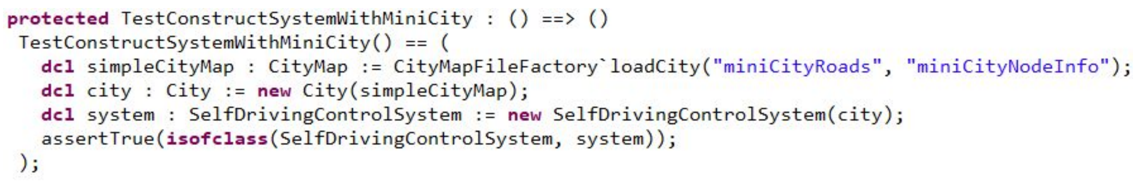


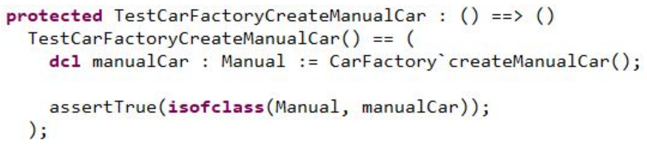
Figure 35 - The city used for the execution of the program



```
protected TestConstructSystemWithMiniCity : () ==> ()
 TestConstructSystemWithMiniCity() == (
   dcl simpleCityMap : CityMap := CityMapFileFactory`loadCity("miniCityRoads", "miniCityNodeInfo");
   dcl city : City := new City(simpleCityMap);
   dcl system : SelfDrivingControlSystem := new SelfDrivingControlSystem(city);
   assertTrue(isofclass(SelfDrivingControlSystem, system));
 );
```

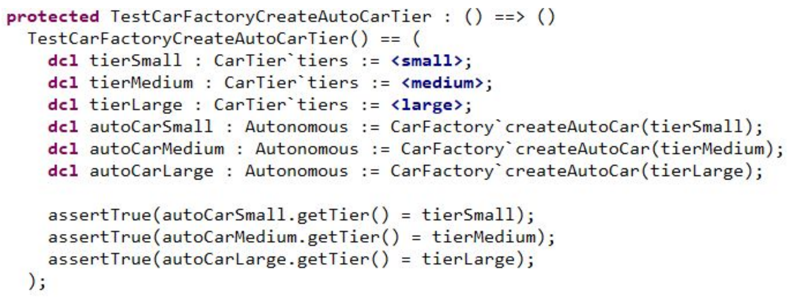
Figure 36 - Load the city map from figure 35 from a csv file

Actors of both the cars and citizens can be created by a factory, the creation tests are shown in figure 37, 38, 39, and 40.



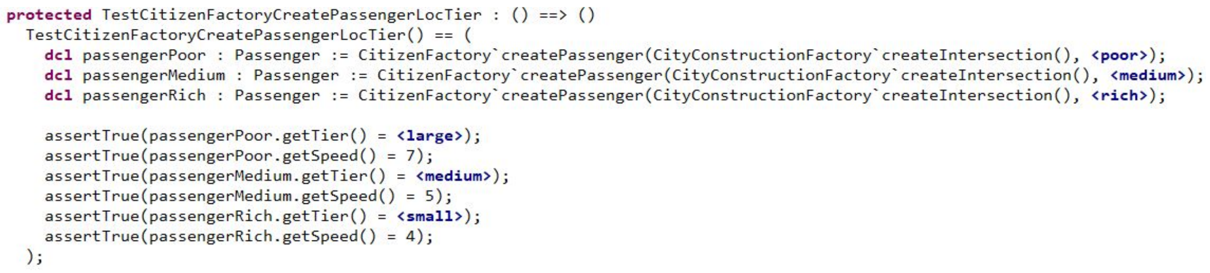
```
protected TestCarFactoryCreateManualCar : () ==> ()
  TestCarFactoryCreateManualCar() == (
    dcl manualCar : Manual := CarFactory`createManualCar();

    assertTrue(isofclass(Manual, manualCar));
  );
```

Figure 37 - Tests if a factory can create a manual car



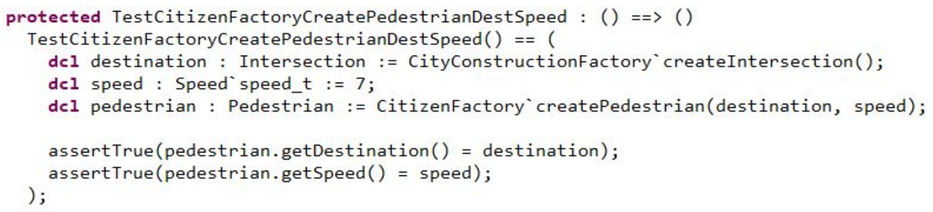
```
protected TestCarFactoryCreateAutoCarTier : () ==> ()
  TestCarFactoryCreateAutoCarTier() == (
    dcl tierSmall : CarTier`tiers := <small>;
    dcl tierMedium : CarTier`tiers := <medium>;
    dcl tierLarge : CarTier`tiers := <large>;
    dcl autoCarSmall : Autonomous := CarFactory`createAutoCar(tierSmall);
    dcl autoCarMedium : Autonomous := CarFactory`createAutoCar(tierMedium);
    dcl autoCarLarge : Autonomous := CarFactory`createAutoCar(tierLarge);

    assertTrue(autoCarSmall.getTier() = tierSmall);
    assertTrue(autoCarMedium.getTier() = tierMedium);
    assertTrue(autoCarLarge.getTier() = tierLarge);
  );
```

Figure 38 - Tests if a factory can create autonomous cars of different tiers



```
protected TestCitizenFactoryCreatePassengerLocTier : () ==> ()
  TestCitizenFactoryCreatePassengerLocTier() == (
    dcl passengerPoor : Passenger := CitizenFactory`createPassenger(CityConstructionFactory`createIntersection(), <poor>);
    dcl passengerMedium : Passenger := CitizenFactory`createPassenger(CityConstructionFactory`createIntersection(), <medium>);
    dcl passengerRich : Passenger := CitizenFactory`createPassenger(CityConstructionFactory`createIntersection(), <rich>);

    assertTrue(passengerPoor.getTier() = <large>);
    assertTrue(passengerPoor.getSpeed() = 7);
    assertTrue(passengerMedium.getTier() = <medium>);
    assertTrue(passengerMedium.getSpeed() = 5);
    assertTrue(passengerRich.getTier() = <small>);
    assertTrue(passengerRich.getSpeed() = 4);
  );
```

Figure 39 - Tests if a factory can create Passengers of different tiers



```
protected TestCitizenFactoryCreatePedestrianDestSpeed : () ==> ()
  TestCitizenFactoryCreatePedestrianDestSpeed() == (
    dcl destination : Intersection := CityConstructionFactory`createIntersection();
    dcl speed : Speed`speed_t := 7;
    dcl pedestrian : Pedestrian := CitizenFactory`createPedestrian(destination, speed);

    assertTrue(pedestrian.getDestination() = destination);
    assertTrue(pedestrian.getSpeed() = speed);
  );
```

Figure 40 - Tests if a factory can create Pedestrians with a given destination and walking speed.

Execution of the full system

There will be no execution of a fully implemented system that is orchestrated by World. The system is not fully implemented and can therefore not be executed. The following elements needs implemented:

- Route planning
- Tests of Step operations
- Tests of SelfDrivingControlSystem
- Tests of City

Tasks for future work will therefore be to get these parts to work and get the World class to run the system in steps.

Conclusion

The critical system “The self-driving control system” has been developed, modelled, and tested through this project. The project uses a structured work process defined for VDM model development and further extends it with the VDM-RT process [5]. A system description has been formulated for a critical system. Based on the description has system specifications that captures the functional and non-functional properties of the system been identified. From the description and specification has classes, data types, and operations been identified from nouns and actions. These elements have led to a domain model that represents the relationships between the identified elements. The domain model is used as the basis for the logical view, the logical view captures the technical architecture and represents ideas and concepts for a proper model of the system.

The model is implemented based on the analytic done prior to the logical view and tries to capture the ideas and concepts from the architecture. This has resulted in a model where the Car actors, Citizen actors, cityMap, and CityMapLoader is properly implemented and tested. On the contrary is the SelfDrivingControlSystem, RoutePlanner, World, and step operations not tests and route planning is yet to be implemented.

The state of the final product is based on a goal of getting a large critical system to work that is properly tested. This resulted in a TDD process that focused too much time on the implementation of low level functionality and to have these fully tested. This resulted in a process that was not iterative but instead worked as a waterfall process, this led to the system not working as a whole.

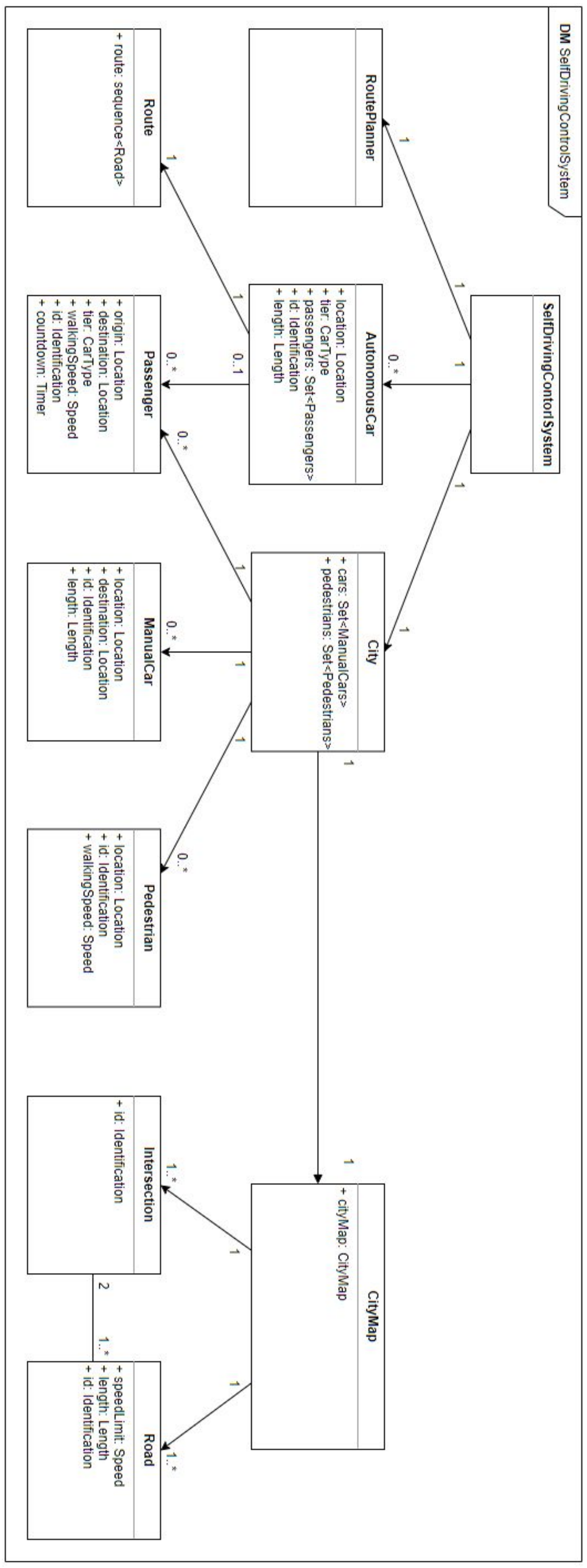
But even without the system fully working have good model structures been shown and important nuances explored. The separation of concern, the decoupling of actors, the inheritance and polymorphism to abstract and reuse, the loaders to make modelling more modular, and the use of step to simulate time are all elements that show how a model should work in the given case. This also led to the second stage of the VDM-RT process (sequential).

Bibliography

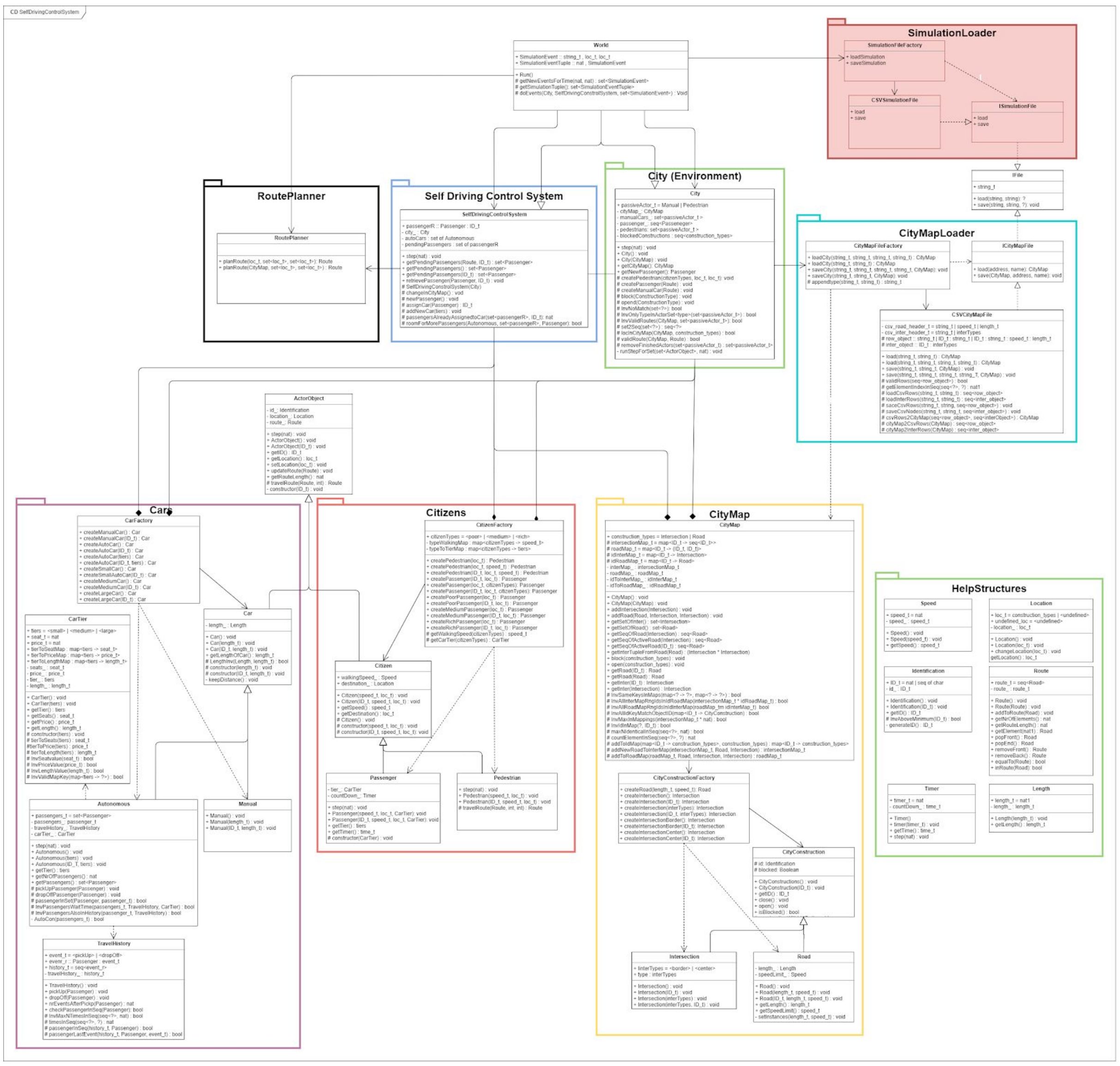
[1] Peter Gorm Larsen, 2020, Modelling of Critical Systems, Aarhus University Department Of Engineering, 06. December 2020, <https://kursuskatalog.au.dk/en/course/100190/Modelling-of-Critical-Systems>
[2] Overture, 2020, Overture Tool, Overture, 6. December, 2020, <https://www.overturetool.org/>
[3] Wikipedia, 6. September 2020, Vienna Development Method, Wikipedia.org, 06. December 2020, <https://en.wikipedia.org/wiki/Vienna_Development_Method>
[4] Mathias Grønne, 06. December 2020, The Self driving control system, Github, 07. December 2020, <https://github.com/Gronne/The-Self-Driving-Traffic-Control-System>
[5] Peter Gorm Larsen & John Fitzgeral & Sune Wolff, January 2009, Methods for the Development of Distributed Real-Time Embedded Systems Using VDM, Newcastle University, 16. November 2020
[6]Aarhus University, Unknown, Peter Gorm Larsen Processor, Aarhus University, 07. December 2020, <https://pure.au.dk/portal/da/persons/peter-gorm-larsen(318d787b-3528-4a49-88be-60491de69409).html>
[7] Google, 07. December 2020, Google Trends Autonomous cars, Google, 07. December 2020, <https://trends.google.com/trends/explore?date=all&geo=US&q=autonomous%20cars>
[8] Wikipedia, 26. November 2020, Tesla Autopilor, Wikipedia.org, 07. December 2020, <https://en.wikipedia.org/wiki/Tesla_Autopilot>
[9] Wikipedia, 11. November 2020, Cruise Control, Wikipedia.org, 07. December 2020, <https://en.wikipedia.org/wiki/Cruise_control>
[10] Wikipedia, 8. November 2020, Automatic parking, Wikipedia.org, 07. December 2020, <https://en.wikipedia.org/wiki/Automatic_parking>
[11] Wikipedia, 29. November 2020, Car, Wikipedia.org, 07. December 2020, https://en.wikipedia.org/wiki/Car
[12] TrueCarAdviser.org, 24. January 2020, The 5 levels of Autonomous vehicles, TheCarAdviser.org, 07. December 2020, https://www.truecar.com/blog/5-levels-autonomous-vehicles/
[13] Justin Hughes, 7. November 2020, Waymo is already Running Self-Driving Cars With No One Behind the Wheel, TheDrive.com, 07. December 2020, <https://www.thedrive.com/tech/15848/waymo-is-already-running-cars-with-no-one-behind-the-wheel>
[14] Wikipedia, 3. December 2020, MoSCoW method, Wikipedia.org, 07. December 2020, <https://en.wikipedia.org/wiki/MoSCoW_method>
[15] Wikipedia, 6. December 2020, Graph theory, Wikipedia.org, 07. December 2020, <https://en.wikipedia.org/wiki/Graph_theory>

Appendix

Appendix A - Domain model

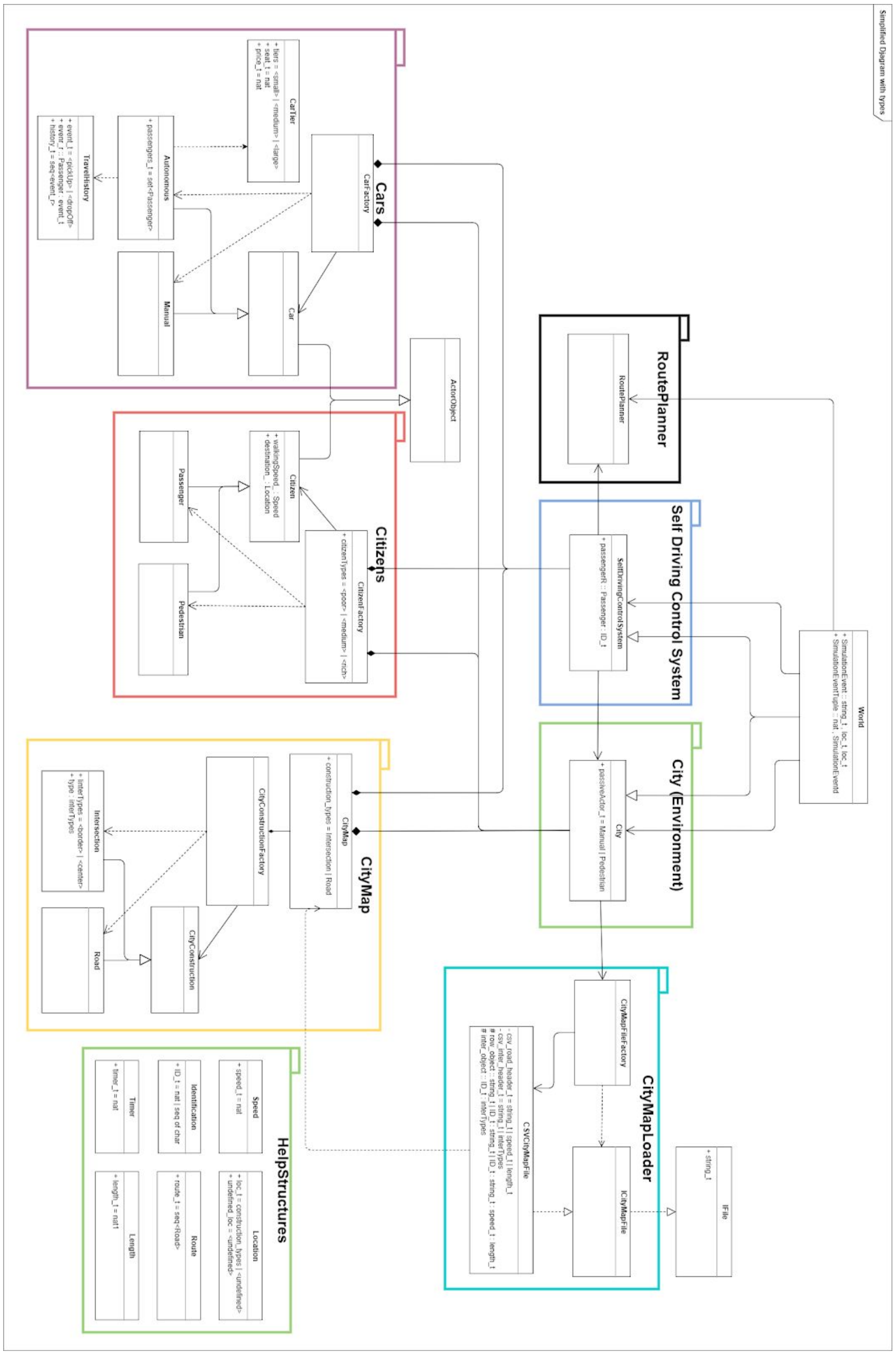


Appendix B - Class diagram



SimulationLoader has not been implemented

Appendix C - Class Diagram (Types only)


Simplified Diagram with types
Cars
CarFactory
CarTier
+ tiers = <small> | <medium> | <large>
+ seat_t = nat
+ price_t = nat
Autonomous
+ passengers_t = set<Passenger>
TravelHistory
+ event_t = <pickUp> | <dropOff>
+ event_r :: Passenger : event_t
+ history_t = seq<event_r>
Manual
Car
ActorObject
RoutePlanner
RoutePlanner
Citizens
CitizenFactory
+ citizenTypes = <poor> | <medium> | <rich>
Citizen
+ walkingSpeed_ :: Speed
+ destination_ :: Location
Passenger
Pedestrian
Self Driving Control System
SelfDrivingControlSystem
+ passengerR :: Passenger : ID_t
World
+ SimulationEvent :: string_t , loc_t, loc_t
+ SimulationEventTuple :: nat , SimulationEventd
City (Environment)
City
+ passiveActor_t = Manual | Pedestrian
CityMap
CityMap
+ construction_types = Intersection | Road
CityConstructionFactory
Intersection
+ interTypes = <border> | <center>
+ type : interTypes
Road
CityConstruction
CityMapLoader
CityMapFileFactory
CSVCityMapFile
- csv_road_header_t = string_t | speed_t | length_t
- csv_inter_header_t = string_t | interTypes
# row_object :: string_t | ID_t : string_t | ID_t : string_t : speed_t : length_t
# inter_object :: ID_t : interTypes
ICityMapFile
IFile
+ string_t
HelpStructures
Speed
+ speed_t = nat
Identification
+ ID_t = nat | seq of char
Timer
+ timer_t = nat
Location
+ loc_t = construction_types | <undefined>
+ undefined_loc = <undefined>
Route
+ route_t = seq<Road>
Length
+ length_t = nat1