RWTH Aachen University
Software Engineering Group

Development of Web Playground for Component and Connector Models

Master Thesis

presented by

Strepkov, Ievgen

1st Examiner: Prof. Dr. B. Rumpe

2nd Examiner: Prof. Dr. H. Lichter

Advisor: Dipl.-Math. Michael von Wenckstern

The present work was submitted to the Chair of Software Engineering

Aachen, October 2, 2018

Abstract

Cyber-physical systems become highly popular in people's everyday life. One of these systems are self-driving vehicles in automotive industry. To design and develop that kind of systems, component and connector (C&C) models are widely used. This thesis focuses on creating a tool supporting the learning process of C&C modeling principles. EmbeddedMontiArc has been chosen as one of the most advanced C&C modelling languages. Furthermore, it is not sufficient to have an environment for the development, it is important to have enthralling tutorials which reveal all advantages of the C&C modeling. The presented tutorials apply the gamification principles, which become popular nowadays through a good learning outcome. To build a comprehensive teaching tool, we analysed existing tutorials and playgrounds, to derive important requirements for the future product. Based on the derived requirements, an architecture is created and implementation of components is given. Moreover, technologies that were used during the development process are explained in details. Later, it is clarified how to use the environment to achieve high productivity during the learning process. Finally, this thesis presents a group of tutorials, with real examples and step-by-step solutions. To try the tutorials online, the following link should be used [Onl18]. The article, which describes the toolchain with tutorials [KRSvW18b], was published on the fifth International Workshop on Interplay of Model-Driven and Component-Base Software Engineering.

Contents

Chapter 1

Introduction and Motivation

Autonomous vehicles are very important part of our future life. To design and develop that kind of cyber-physical systems, component and connector (C&C) models are widely used. The models are applied in automotive, aerospace or even image processing domains. Using C&C modeling, it is easier to represent different feature layers and their logical interactions. The most important feature of C&C modeling is an opportunity to decompose complex models into sub-components and develop and manage these, less complex components, by domain experts.
To inspire students to be involved in the future technology, a web-playground has been developed, which facilitates controllers creation for a simulator and almost instantly observe the result in the 3D environment. The visualization motivates students and makes the studying process more attractive due to gamification. This kind of education becomes more popular recent years due to good learning outcome [Gam18].
To teach students how to develop C&C models, must be used a C&C modeling language which has all features and tools to satisfy our requirements. EmbeddedMontiArc [HR17] language was chosen for this purpose, to achieve the best results in short terms.
This thesis is focused on building the ultimate teaching playground with real tutorial examples. Chapter 2 analyzes existing tutorials and playgrounds. After the analysis, the important requirements for the future product will be derived. Chapter 3 describes the technologies and products, which are used in the thesis. After that, in the chapter 4, the architecture and implementation are presented. Next chapter, which is number 5, describes how to use the implemented system. Chapter 6 presents a group of tutorials with solutions. In the end, the possible improvements are presented in the future work chapter and then the conclusion follows.

Chapter 2

Related work

This section analyzes different tutorials with the aim of extracting the most convenient and important features. The results are used to increase the productivity and efficiency of the teaching process. Different tutorials and playgrounds have been taken into consideration from various domains: programming languages, numerical computing environment and so on. The main goal is to find the most useful features and integrate them. Section 2.1 introduces tutorials for the very powerful tool, which is used in different areas for building systems with any complexity level - Simulink. Section 2.2 presents tutorials for the Rust programming language. In the section 2.3, is shown the theorem prover Z3 Solver from Microsoft. Section 2.4 considers the Octave online, web-playground for numerical computations. The next one is the section 2.5, where is Wolfram Alpha is reviewed, which can solve various of mathematical tasks. In the section 2.6, is presented a playground for the Typescript language, which is widespread due to type checking, in contrast to its predecessor JavaScript. Section 2.7, analyzes the Swift Playground, which has the most advanced 3D visualization but limited platform support. After reviewing the different tutorials and playgrounds, the requirements for the future tool will be derived and subsequently used in the development process. Using this approach, it is possible to combine advantages from the reviewed applications in one the most advanced tool.

2.1 Simulink

Mathworks Simulink [Mat18a] is a block diagram environment for various domain simulation and Model-Based Design [LMT+18]. It also involves C&C models into the modeling process. Simulink is a graphical modeling environment. Figure 2.1 shows an example of a fuel calculation subsystem. There are depicted the incoming and outgoing ports of the schema, like `est_airflow` or `fuel_rate` and the connections between the components. The schema is very useful for a visual perception. You can easily see connections between elements and understand the logic behind that. When there is just a textual representation of elements and connections, it is much harder to imagine the whole schema, e.g., find some issue with an accidentally connected port or disconnected one. Simulink has plenty of tutorials in many different areas with detailed description and videos on solving it, which describe it step-by-step. It is very helpful to present step-by-step solutions with detailed description for understanding of important details. Based on this understanding of essentials is build the future knowledge and depends future success. But there is some weakness in an education process. The issue is that they do not have methods for validation the correctness of the user's solution. They do not encourage users to try it out by

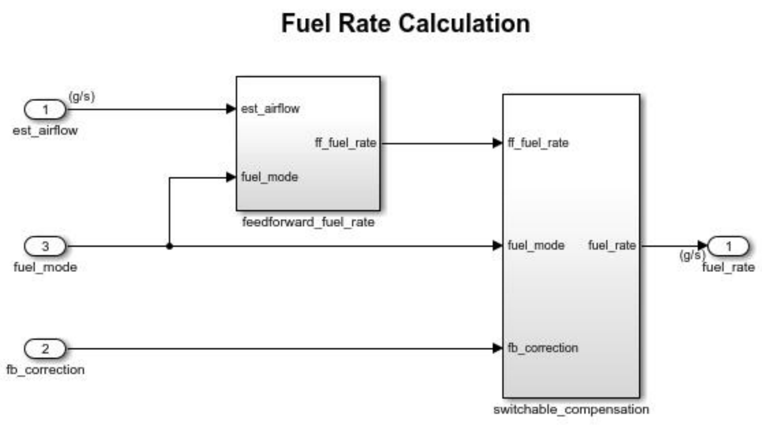
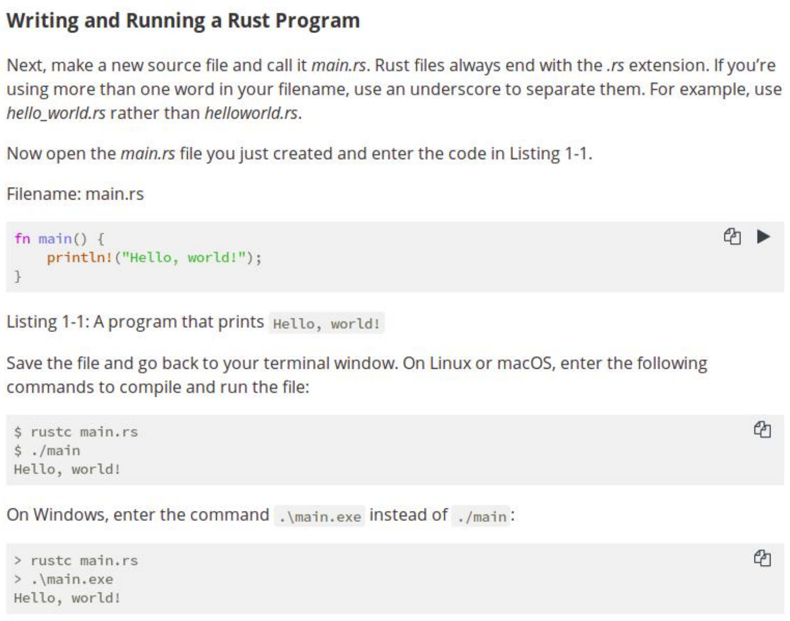


Figure 2.1: Simulink schema example.

themselves, just copy a sample solution, that can not guaranty the solution understanding.

2.2 Rust

Rust [Rus18, KN18] is a very popular programming language, the prevalence of which is growing every day. Firefox is written in Rust. It has a consistent tutorial which describes language constructs with gradually increasing complexity. It has the informative and structured index, where users can easily jump from one topic to another almost instantly and then just go back to the place where he was reading before. They use highlighted areas to show some code examples, that facilitate understanding of presented materials. It is possible to copy some parts of the code and even directly execute it in a browser (see figure 2.2).



Writing and Running a Rust Program

Next, make a new source file and call it *main.rs*. Rust files always end with the *.rs* extension. If you're using more than one word in your filename, use an underscore to separate them. For example, use *hello_world.rs* rather than *helloworld.rs*.

Now open the *main.rs* file you just created and enter the code in Listing 1-1.

Filename: main.rs

```
fn main() {
    println!("Hello, world!");
}
```

Listing 1-1: A program that prints `Hello, world!`

Save the file and go back to your terminal window. On Linux or macOS, enter the following commands to compile and run the file:

```
$ rustc main.rs
$ ./main
Hello, world!
```

On Windows, enter the command `.\main.exe` instead of `./main`:

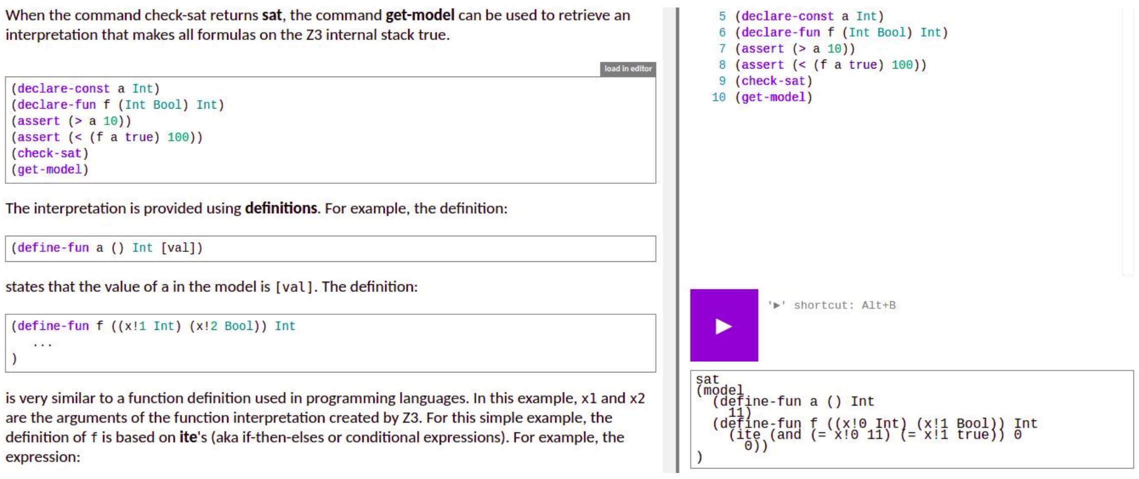
```
> rustc main.rs
> .\main.exe
Hello, world!
```

Figure 2.2: Rust tutorial example.

To execute the given code, you just should click on the play button in the right upper corner and in several seconds a result will be displayed. This is very important and useful feature, because students can directly see what the code does. At the given example, it is pretty straightforward, but if you have some computations, not even complex ones, it is already not so easy to imagine the output. Nevertheless, even in this short example, there are some issues related to the compilation process. Namely, a different compilation and execution process for diverse platforms (Windows, Linux, macOS). It means, that we have to install the compiler to use it during the teaching process. It would be very convenient to have all these features directly in the web-browser, without an installation.

2.3 Microsoft Z3 Prover

Z3 is a state-of-the art theorem prover from Microsoft [Z3P18]. It provides similar experience compare to Rust tutorial, but with some improvements, which simplify the studying process. There is a possibility, not only to execute the given code from the current example directly in a browser, but edit the code, and still see results almost instantly, due to in-browser execution. It facilitates seeing the direct binding between written commands and real results, that improve understanding of given material.


When the command check-sat returns **sat**, the command **get-model** can be used to retrieve an interpretation that makes all formulas on the Z3 internal stack true.

```
(declare-const a Int)
(declare-fun f (Int Bool) Int)
(assert (> a 10))
(assert (< (f a true) 100))
(check-sat)
(get-model)
```

The interpretation is provided using **definitions**. For example, the definition:

```
(define-fun a () Int [val])
```

states that the value of a in the model is [val]. The definition:

```
(define-fun f ((x!1 Int) (x!2 Bool)) Int
   ...
)
```

is very similar to a function definition used in programming languages. In this example, x1 and x2 are the arguments of the function interpretation created by Z3. For this simple example, the definition of f is based on **ite**'s (aka if-then-elses or conditional expressions). For example, the expression:

```
(declare-const a Int)
(declare-fun f (Int Bool) Int)
(assert (> a 10))
(assert (< (f a true) 100))
(check-sat)
(get-model)
```

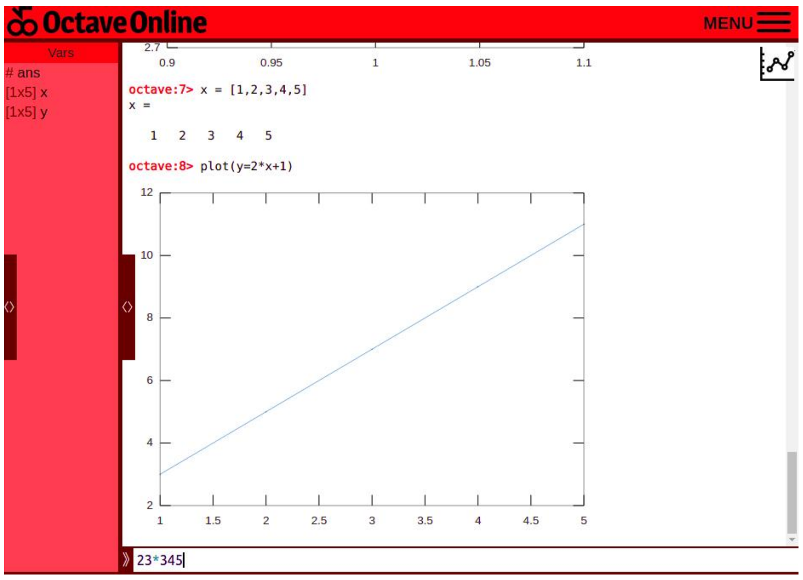
```
sat
(model
  (define-fun a () Int
    11)
  (define-fun f ((x!0 Int) (x!1 Bool)) Int
    (ite (and (= x!0 11) (= x!1 true)) 0
      0))
)
```

Figure 2.3: Z3 tutorial example.

Figure 2.3 shows the code, which is given inside the tutorial on the left hand side. On the right hand side, there is the code which has been executed and the result appeared below. Then, it is possible to adjust the code to check some hypothesis and instantly see results. By doing this, students can check their understanding of an explained material. Another advantage in this tutorial is a in-browser execution. Students do not have to install anything and can directly work in browsers. It means, the operating system does not have any influence on the execution and compilation process. Any operating system can be used, it is necessary only to have a web-browser. If this tutorial is used directly on a lecture, then students just enter the URL in a browser and can instantly try to execute some examples. Microsoft also allows to share code via URL links. This is convenient if a student has a question, he only should send the URL to the supervisor.

2.4 Octave Online

Octave Online [Oct18, Nag18] is a web-playground for a high-level language Octave, which is primarily intended for numerical computations. Octave is an open-source alternative to the commercial MathWorks Matlab [Mat18b]. It has a simple and intuitive interface despite a complexity of an internal implementation. It provides directly in a browser fast execution with errors handling. Octave Online supports even complex calculations, it is not needed to install any software on the PC. Everything works out of the box in a browser. This web-playground is oriented on students, who are using the tool directly



```
octave:7> x = [1,2,3,4,5]
x =

   1   2   3   4   5

octave:8> plot(y=2*x+1)
```

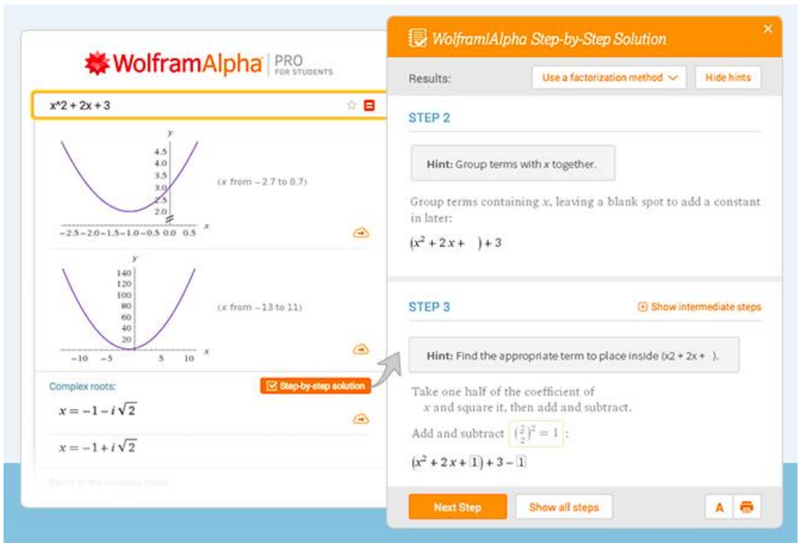
```
23*345
```

Figure 2.4: Octave web-playground example.

on a lecture. They do computations in a browser and run previously created scrips, as a sequence of commands. To visualize given data, there is a possibility to build plots for better understanding of mathematical abstractions. Despite of a browser execution, it has solid computation performance.

2.5 Wolfram Alpha

Wolfram Alpha [Wol18] is a very powerful tool, which works by using expert-level knowledge and algorithms to automatically answer questions, do analysis and generate reports. It has matrix operations and calculations as the EmbeddedMontiArc language has, to describe atomic components. Wolfram Alpha can do even more as solving linear equations, it allows to specify the behavior of controllers, and after, by solving the equations, they synthesize the controller. Furthermore, it has one very interesting and useful feature, which provides an interactive visualization of a given data. The idea behind it is that students can perceive, how one or another parameter influences on the final result. It promises a better understanding of the dependencies between the components or elements of the observed system. It offers tutorials, which have step-by-step solutions with detailed explanation of each step and additional hints for students (see figure 2.5). Futhermore, it has a high quality visualization, which encourage students to do some experiments and see the changes on the displayed figure almost in real time.



STEP 2

Hint: Group terms with x together.

Group terms containing x, leaving a blank spot to add a constant in later:

$(x^2 + 2x + \quad) + 3$

STEP 3

Hint: Find the appropriate term to place inside (x2 + 2x +).

Take one half of the coefficient of x and square it, then add and subtract.

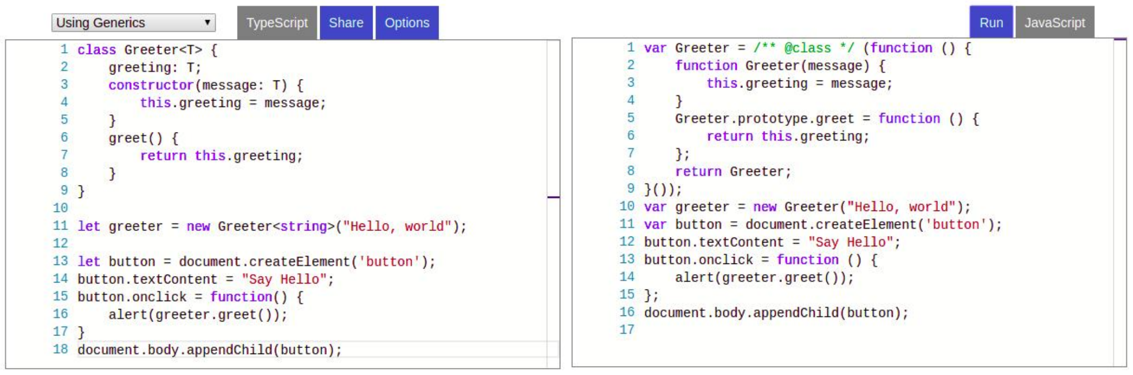
Add and subtract $\left(\frac{2}{2}\right)^2 = 1$:

$(x^2 + 2x + \boxed{1}) + 3 - \boxed{1}$

Figure 2.5: Wolfram Alpha example.

2.6 TypeScript Playground

TypeScript [Typ18, Jan18] is a typed superset of JavaScript language. It has a clean and simple playground which shows the difference and benefits of TypeScript over JavaScript language. It has preloaded examples which actually show this difference and a user can see distinction in the direct comparison (see figure 2.6). The given direct comparison gives a better understanding and facilitates further analysis of the features.



```
class Greeter<T> {
    greeting: T;
    constructor(message: T) {
        this.greeting = message;
    }
    greet() {
        return this.greeting;
    }
}

let greeter = new Greeter<string>("Hello, world");

let button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function() {
    alert(greeter.greet());
}
document.body.appendChild(button);
```

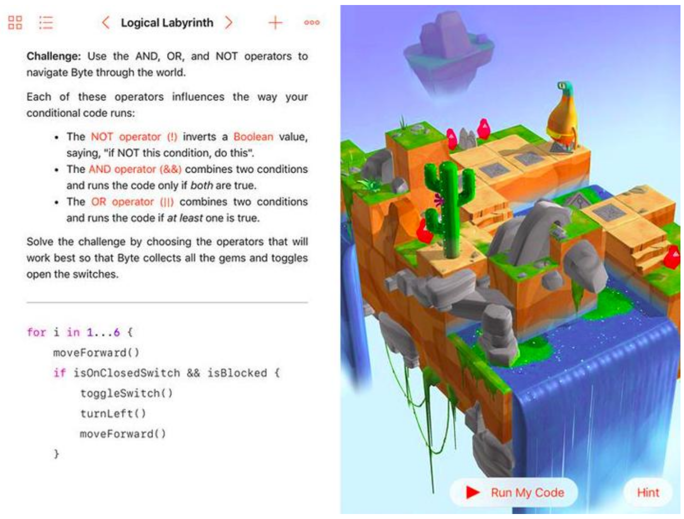
```
var Greeter = /** @class */ (function () {
    function Greeter(message) {
        this.greeting = message;
    }
    Greeter.prototype.greet = function () {
        return this.greeting;
    };
    return Greeter;
}());
var greeter = new Greeter("Hello, world");
var button = document.createElement('button');
button.textContent = "Say Hello";
button.onclick = function () {
    alert(greeter.greet());
};
document.body.appendChild(button);
```

Figure 2.6: Typescript playground example.

Figure 2.6 emphasizes the difference in classes architecture. It helps to find architecture advantages of using the new language (means Typescript). Moreover, you can run the given example and see the result directly in a browser, but in a different tab, which is not really convenient. If you have some complex code, it is really helpful to see the source code and the result simultaneously, to be able to match them.

2.7 Swift Playground

Swift Playground [Swi18, KLB18] has been made for teaching the Swift language in a game form. The playground was developed with a focus on high school students. For this category of users it is very important to attract their attention. Students can create small programs that instantly show the results of the code that you have written. From the right side of the screen is shown a 3D world where an action is happen (see figure 2.7). The tutorials are pretty simple but the concept is very interesting. Tutorials have automatic verification of an implemented correctness in the 3D environment. To produce many diverse game oriented tutorials, it would be convenient to have simple 3D models importing, that can use different models from various 3D editors. Apple gives into trends



Challenge: Use the AND, OR, and NOT operators to navigate Byte through the world.

Each of these operators influences the way your conditional code runs:

- The NOT operator (!) inverts a Boolean value, saying, "if NOT this condition, do this".
- The AND operator (&&) combines two conditions and runs the code only if *both* are true.
- The OR operator (||) combines two conditions and runs the code if *at least* one is true.

Solve the challenge by choosing the operators that will work best so that Byte collects all the gems and toggles open the switches.

```
for i in 1...6 {
    moveForward()
    if isOnClosedSwitch && isBlocked {
        toggleSwitch()
        turnLeft()
        moveForward()
    }
```

Figure 2.7: Swift playground example.

and have used the gamification in the tutorials, to attract learners to their product. The biggest limitation is that the tutorials are only available on iPads.

2.8 Derived Requirements

Thoroughly analyzed the projects described above, the following list of requirements has been derived:

- (R1) 3D visualization for demonstration purposes.
- (R2) Simple, clean, and intuitive interface.
- (R3) Work on any operating system and without installation.
- (R4) Fast compilation.
- (R5) Import and use existing 3D models for the simulation.
- (R6) Displaying the object's trajectory.
- (R7) Integrated testing support.
- (R8) Automatic verification of obtained results.

The table 2.1 summarizes the comparison between the all considered tutorials.
Take a closer look at the differences between the tutorials regarding to the derived re-

Table 2.1: summarized comparison between tutorials (+ support, P partially support)

	Z3 Solver	Octave	Wolfram	TypeScript	Swift	Rust	Simulink	EMAM PG
R1	-	+	+	-	+	-	+	+
R2	+	+	+	+	+	+	+	+
R3	+	+	+	+	-	+	-	+
R4	+	+	+	+	+	+	+	-
R5	-	-	-	-	-	-	+	+
R6	-	-	P	-	P	-	P	+
R7	-	-	-	-	-	-	+	+
R8	-	-	-	-	+	-	+	+

quirements above.
(R1) 3D visualization for demonstration purposes: Four of considered tutorials have a 3D visualization. Octave online has an ability to generate plots and graphics for given data. Wolfram Alpha has a very powerful tool which can generate 3D models and user can even interact with them and see the changes in real time. Whereas, Swift Playground possesses the most advanced 3D world, which is a part of a tutorial and results presentation. Simulink has an opportunity to build beautiful 3D models, which are involved in the simulation process, but the difference is that a user has to build everything himself. It means, it is not as a part of a tutorial, but an additional feature.
(R2) Simple, clean, and intuitive interface: This is the only requirement, which all tutorials are satisfied. It is very important to have an understandable and clear interface, which does not distract from an educational process. Moreover, the interface should be intuitive, so that students can use it directly at the lecture without detailed explanation of its features.
(R3) Work on any operating system and without installation: Almost all investigated tutorial satisfy this requirement, except the Swift tutorial and Simulink. The Swift tutorial has only iOS implementation and only is used at the iPad. Simulink does not work on the Web, only MatLab, which has partial web-implementation. Due to this inconvenience, these two tools can be used without preliminary preparation.
(R4) Fast compilation: All presented tutorials have sufficiently fast compilation process. The EmbeddedMontiArc generator has quite complex compiler with amount of features. Due to this reason and the using as a back end not the best hardware, the compilation process takes time. Later the architecture will be described in details.
(R5) Import and use existing 3D models for the simulation: Only Simulink has feasibility to import 3D models. It helps to create tutorials quickly and efficiently, by using the previously created models and configurations. An example of reusing a 3D object can be a cone that is used in many exercises. This feature simplifies the process of creating new tutorials and decreases the time which has to be invested into the building process. Even better to have possibility to load the models, which can have different format, like Babylon [Bab18] or Blender [Ble18].
(R6) Displaying the object's trajectory: Wolfram Alpha, Swift PG and Simulink, are partially support this feature in case, that you can see the entire process of movement of an object from the very beginning to the end. But it would be better to improve the concept and add the separate window which permanently displays a traversed route of an object, for the better visual perception and visual comparison of results. In our case the object is a car and a trajectory completely describes the passed segment of a track.

(R7) Integrated testing support: Only Simulink has integrated testing options. Due to the specificity of the tutorials, tests play an important role. Writing streaming tests for components, students can be sure that it reacts properly on incoming data. Tests make the components more reliable and robust. Because of the using the C&C language, it is great to be sure that each component of a composed model behaves correctly.
(R8) Automatic verification of obtained results: Only one among the examined tutorials, the Swift tutorial, has a gaming base verification of a solution correctness. It causes additional interest in the studying process, and can be the motivation to keep solving the tasks, by analogy with computer games. Whereas Simulink gives you possibility to do it, but it is more like extra feature which can be implemented.
Taking into account all these derived requirements we are going to start working on the architecture, which is capable to support the required features.

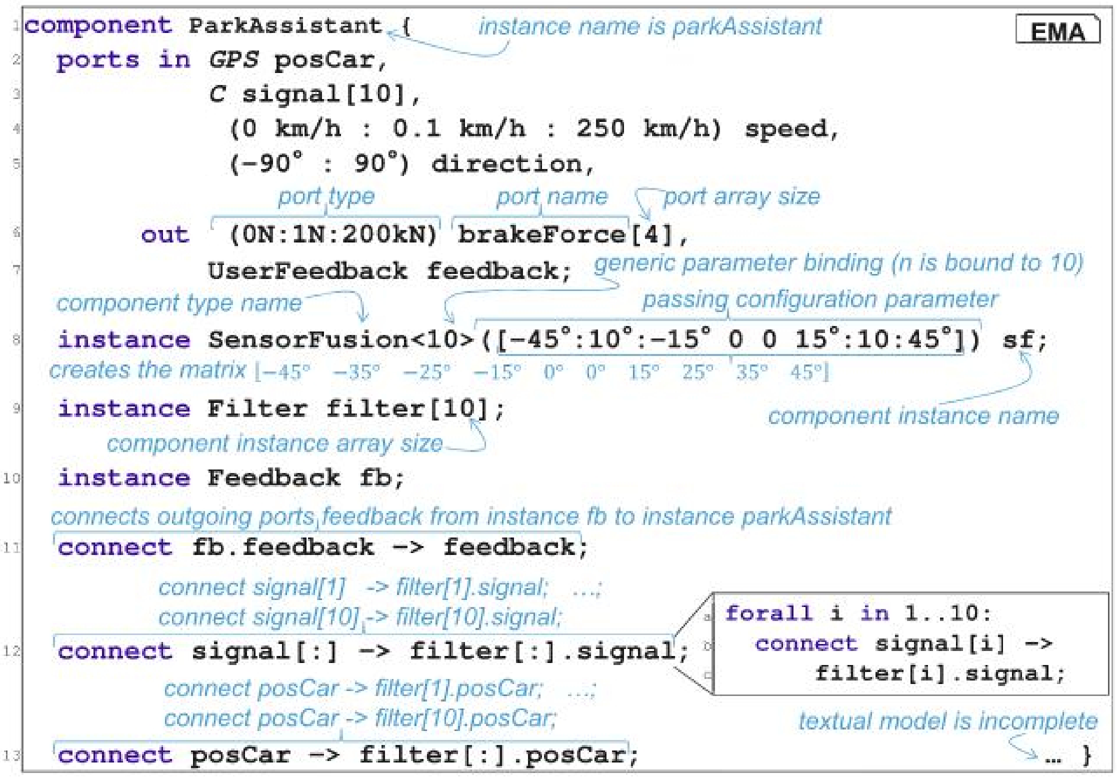
Chapter 3

Preliminaries

This chapter introduces technologies that have been used in this thesis. Section 3.1 gives an overview of the modelling C&C language - EmbeddedMontiArc. Section 3.2 presents the full stack of EmbeddedMontiArc tools that have been used in the toolchain. More detailed description of the tools will be given in the following subsections. Subsection 3.2.1 gives an overview of the Online IDE, which is used for writing EmbeddedMontiArc code in a web browser. Subsection 3.2.2 describes EmbeddedMontiArc's WebAssembly generator, which translate code from EmbeddedMontiArc to C++ and then to WebAssembly. WebAssembly is a binary format running directly in a web browser. Subsection 3.2.3 explains the features of the schema' picture generator.

3.1 EmbeddedMontiArc Language

EmbeddedMontiArc language family consists of the main language EmbeddedMontiArc [KRRvW17, KRRvW18, Hei18, Hal18], plus extra languages providing basic features to EmbeddedMontiArc, e.g., structure, test, and design languages. It was designed to model architectures of embedded and cyber-physical systems. Other popular modeling architectures of cyber-physical systems are: Simulink [Bis96, DH04], Modelica [Ass05, Fri11], SysML [OMG07, Hau06] and many others. To show main advantages of the language, it is easier to start directly with an example. Figure 3.1 shows the `ParkAssistant` component. This is not the entire component but a main part, which shows the most important features. It has incoming and outgoing ports. All ports must have a type, name and range. The type defines a kind of a signal (e.g., `Q`,`N`,`Z`,`B`). The range is very important, it helps to control the signal in boundaries, which are defined in advance. Moreover, the range defines not only the boundaries values, but the units (e.g., km/h, m/s and so on). Another option, which is available for ports it is a port size array. There is a possibility to define an array of ports, it can increase the readability of a connection schema and simplify a ports connection. When the component is created and ports have been defined, it is possible to instantiate another components inside the component. Lines 8, 9 and 10 illustrate several available features. The first one is to define the generic parameter for a component, to be able to create a more general component, which can be used for different types of incoming signal. The next one is to create an array of components, if it is required several similar components for some reason, like the filter component, which does some signal refining. The last one, it is just a normal component instantiation, without any extra parameters. When all required components have been instantiated, it is time to connect the components together. EmbeddedMontiArc offers several convenient options to do that. Firstly,


EMA
instance name is parkAssistant
port type
port name
port array size
generic parameter binding (n is bound to 10)
component type name
passing configuration parameter
creates the matrix [−45° −35° −25° −15° 0° 0° 15° 25° 35° 45°]
component instance name
component instance array size
connects outgoing ports feedback from instance fb to instance parkAssistant
connect signal[1] -> filter[1].signal; ...;
connect signal[10] -> filter[10].signal;
connect posCar -> filter[1].posCar; ...;
connect posCar -> filter[10].posCar;
textual model is incomplete

```
1  component ParkAssistant {
2    ports in GPS posCar,
3             C signal[10],
4              (0 km/h : 0.1 km/h : 250 km/h) speed,
5              (-90° : 90°) direction,
6         out  (0N:1N:200kN) brakeForce[4],
7             UserFeedback feedback;
8    instance SensorFusion<10>([-45°:10°:-15° 0 0 15°:10:45°]) sf;
9    instance Filter filter[10];
10   instance Feedback fb;
11   connect fb.feedback -> feedback;
12   connect signal[:] -> filter[:].signal;
13   connect posCar -> filter[:].posCar;
                                                    ... }
```

```
forall i in 1..10:
  connect signal[i] ->
      filter[i].signal;
```

Figure 3.1: EmbeddedMontiArc model of ParkAssistant (copied from [KRRvW17])

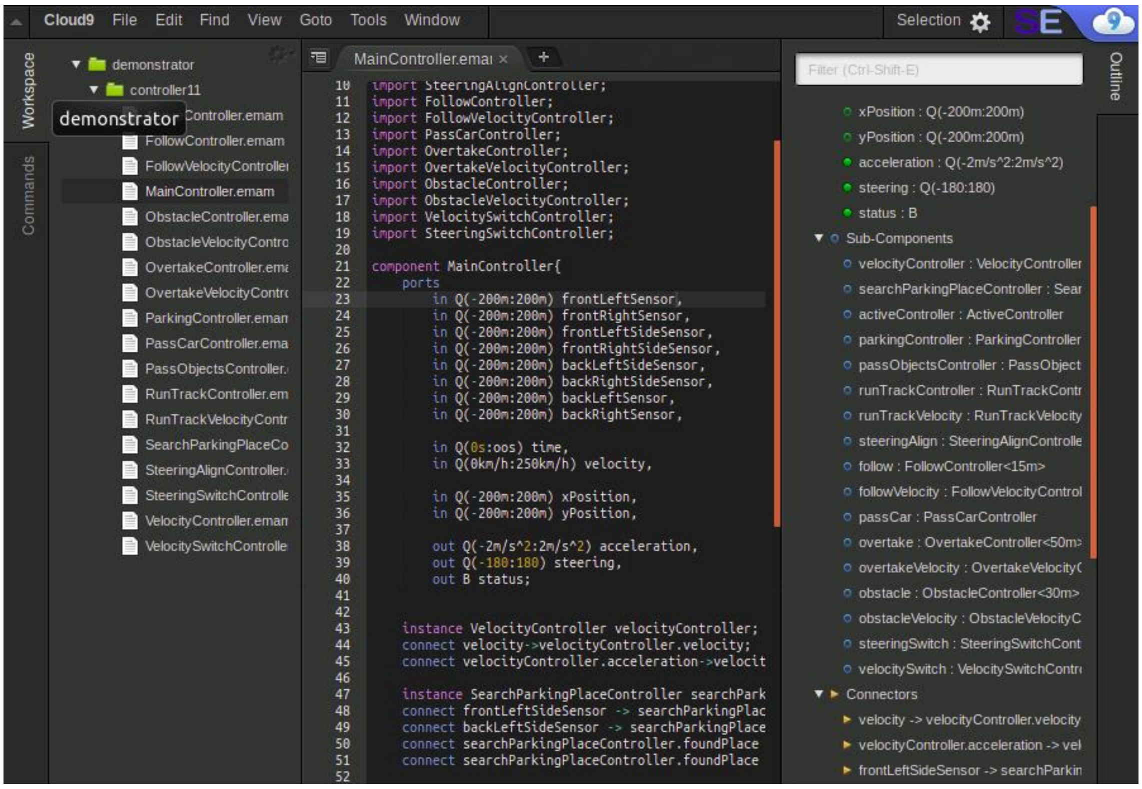
you can just connect the specific port of one component to another specific port of another component (line 11). Then it is possible to connect an array of ports to another one, to reduce the number of connections between components. Another possibility, to connect one port of some component to a specific port of other components, like one-to-many connection. There were described the most important features which influence on the building process of components. More detailed explanation of the language features can be found in the following book [HR17].

3.2 EmbeddedMontiArc Tools

This section describes a full stack of components, which have been used in the toolchain. These components have been developed by other students and in this thesis they are combined in the one chain. The first one is Online IDE, which is used in the front end, and another two are EmbeddedMontiArc's WebAssembly generator and Visualisation generator, which are used in the back end.

3.2.1 Online IDE

Online IDE [Ron17] is based on the Cloud9 IDE [GMC^{+}13, MOA17], that is an online integrated development environment. It supports various languages and was adapted to support the EmbeddedMontiArc. It has many features which simplify the development process (see figure 3.2). The IDE supports an autocompletion for the given languages and syntax highlighting. It increases speed of the development process and convenience during the code writing. From the left hand side of the figure 3.2 the directory tree is located. It simplifies the navigation inside the project. In this area, new files and folders can be created. Files, which have beed opened, organized in tabs, which can be also used for navigation. From the right hand side, there is shown the list of all ports of the currently selected component. Next to it, the list of the sub-components, which contains the selected component. It shows types of components and their names. If there are many other components inside the component, it is easier to see the entire list of components in one screen. Below the sub-components, a group of connectors is shown, which are used to connect currently selected component with others components instantiated inside it. Moreover, the autocompletion is offered to connect to ports of a component, which actually has this component. It means, that the autocompletion tracks all instantiated components with ports. Needless to say, that the IDE supports a text search and search through the ports, sub-components and connectors.



```
import SteeringAlignController;
import FollowController;
import FollowVelocityController;
import PassCarController;
import OvertakeController;
import OvertakeVelocityController;
import ObstacleController;
import ObstacleVelocityController;
import VelocitySwitchController;
import SteeringSwitchController;

component MainController{
    ports
        in Q(-200m:200m) frontLeftSensor,
        in Q(-200m:200m) frontRightSensor,
        in Q(-200m:200m) frontLeftSideSensor,
        in Q(-200m:200m) frontRightSideSensor,
        in Q(-200m:200m) backLeftSideSensor,
        in Q(-200m:200m) backRightSideSensor,
        in Q(-200m:200m) backLeftSensor,
        in Q(-200m:200m) backRightSensor,

        in Q(0s:oos) time,
        in Q(0km/h:250km/h) velocity,

        in Q(-200m:200m) xPosition,
        in Q(-200m:200m) yPosition,

        out Q(-2m/s^2:2m/s^2) acceleration,
        out Q(-180:180) steering,
        out B status;


    instance VelocityController velocityController;
    connect velocity->velocityController.velocity;
    connect velocityController.acceleration->velocit

    instance SearchParkingPlaceController searchPark
    connect frontLeftSideSensor -> searchParkingPlac
    connect backLeftSideSensor -> searchParkingPlace
    connect searchParkingPlaceController.foundPlace
    connect searchParkingPlaceController.foundPlace

```

```
xPosition : Q(-200m:200m)
yPosition : Q(-200m:200m)
acceleration : Q(-2m/s^2:2m/s^2)
steering : Q(-180:180)
status : B
Sub-Components
    velocityController : VelocityController
    searchParkingPlaceController : Sear
    activeController : ActiveController
    parkingController : ParkingController
    passObjectsController : PassObject
    runTrackController : RunTrackContr
    runTrackVelocity : RunTrackVelocity
    steeringAlign : SteeringAlignControlle
    follow : FollowController<15m>
    followVelocity : FollowVelocityControl
    passCar : PassCarController
    overtake : OvertakeController<50m>
    overtakeVelocity : OvertakeVelocity(
    obstacle : ObstacleController<30m>
    obstacleVelocity : ObstacleVelocityC
    steeringSwitch : SteeringSwitchCont
    velocitySwitch : VelocitySwitchContr
Connectors
    velocity -> velocityController.velocity
    velocityController.acceleration -> vel
    frontLeftSideSensor -> searchParkin
```

Figure 3.2: EMA Online IDE.

To store project's files, the virtual file system [VFS18] is used. It provides the filesystem API, which is used by developers to manipulate the project's files. It makes available all basic methods for working with files and folders (e.g., `mkdir`, `readFile`, `copy` and so on).

3.2.2 WebAssembly Generator

This tool generates a WebAssembly binary file [Web18b] and JavaScript file from the EmbeddedMontiArc project. The tool is based on the EmbeddedMontiArc's C++ generator [Sch17b, KRSvW18a], which firstly produces the C++ equivalent of a EmbeddedMontiArc model, afterwards a generated C++ model is used to generate a WebAssembly model. For

this purpose the Emscripten compiler [Ems18, Zak11] is used. The produced files later are used in a browser to execute compiled EmbeddedMontiArc model for a 3D visualization process.
The tool is consisted of the .jar file, which has packed with all dependencies and script files. One of these script files, does the setup procedure and another one executes the main .jar file, with all important parameters. One of the important parameters, which is given in the script file, is a path to Armadillo [Arm18, San10, SC16] library. It is a C++ library for linear algebra & scientific computing and it is used for matrix computations.

3.2.3 Visualisation Generator

The main purpose of the visualisation generator [Sch18a]is to create single or multiple graphical layers of a given controller. A generated schema helps to visualize a textual model of a EmbeddedMontiArc controller. The visualization helps to find some errors related to incorrect connections between some components. The Visualisation generator able to generate multiple layer models, that increases a readability of schemas, because firstly you see more general schema of the model, then you just need to click on some component and an internal schema of the component is appeared. Figure 3.3 depicts the

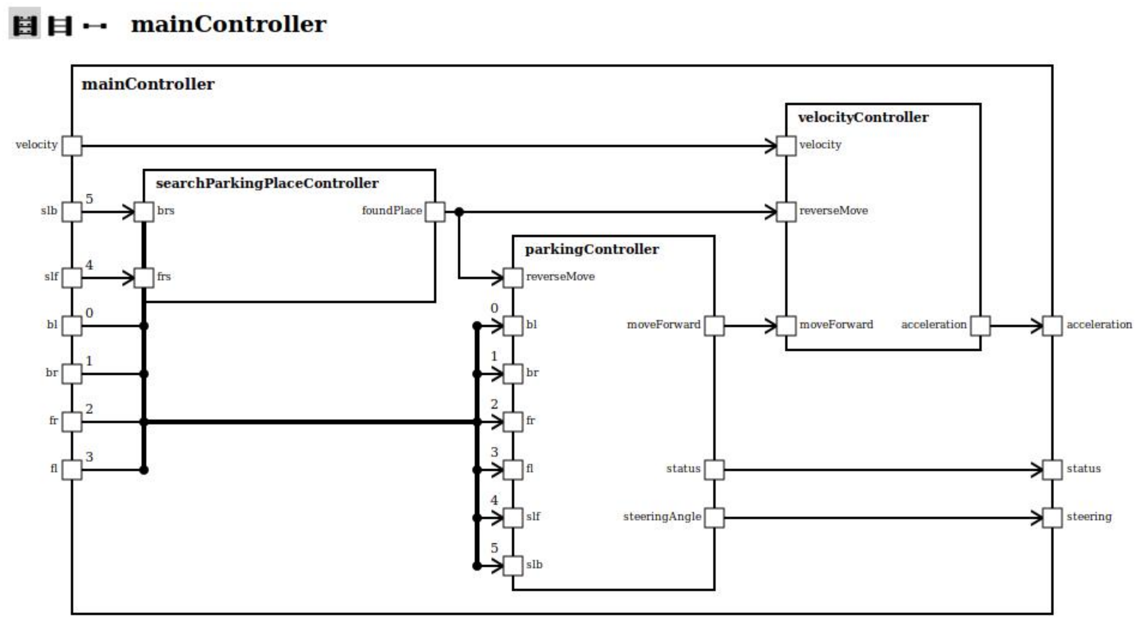


Figure 3.3: EmbeddedMontiArc Parking controller.

parking controller, which has three internal sub-components. Each of these components can have another components inside. The generator calculates the layout in the way to optimize positions of components and decrease the length of the connectors between them. One of the features, is a possibility to simplify the schema by hiding the ports' names and grouping the connections between components.

Chapter 4

Toolchain

This chapter describes the entire toolchain, which is used in the thesis, from creating a C&C model to results demonstration in the 3D simulator. The toolchain provides the functionality to satisfy the earlier derived requirements. The following sections provide the detailed explanation of an architecture and an internal implementation of components from which the toolchain consists.

4.1 Architecture

Software architecture refers to the high level structures of a software system, the discipline of creating such structures, and system [SWA18]. The selection of a suitable architecture is very important at the initial design stage. At the very begging, the idea was to create a serverless application hosted on the GitHub pages [GHP18]. And there was a solution for that, at least it looked like the solution. There is a Clang In Browser [CIB18] project to compile the generated C++ code to WebAssembly and run it directly in a browser. But then some issue, related to the solution, has been revealed. Clang In Browser does not support dynamic linking yet. Therefore, it is not possible to link against the large Armadillo mathematics library, which is used by EmbeddedMontiArc and static compilation would extend the linking process to several minutes. For this reason, the web playground is uses a stateless server to compile the EmbeddedMontiArc code to WebAssembly. User triggers sending of the C&C files to the server by JavaScript, the server translates it to WebAssembly and then the client receives compiled WebAssembly controller. The controller is used in the 3D simulator. The huge advantage is that the server is not involved in the simulation process. It means that it is much easier to handle multiple users and it does not run into critical performance issues, due to multiple requests. The simulation in the browser runs fluently, due to local execution for each user and, in our opinion, it has a massive impact on the user experience. Also, the maintenance of a stateless server is much simpler as the maintenance of stateful one. The disadvantage is that users have to wait for the compilation on the server side and it can take longer due to multiple requests. Nevertheless, the user can observe a fluent driving of the car during the simulation process. EmbeddedMontiArc has variety of developed tools as self-containing services based on .jar files. Due to the derived requirement R3, it has to be developed a web-based application. Therefore on the server-side, some services, from EmbeddedMontiArcStudio, can be selected and integrated. But still, one server-side component is missing. The server must handle multiple users at the same time, and the server must do the messaging from the back-end to the front-end. Students should get a response from the server, to receive com-

piled controller or just fix errors which can appear during the compilation process.
The EmbeddedMontiArcStudio has it own 3D simulator [DDE+17, Lor17, Sch17a, Fro17, Sem17], which has a lot of powerful features [Pav18, Ric18, Ilo18, Sch18b]. However, it has some weaknesses, which do not allow to use it in this case. It cannot handle multiple users, requires a powerful computer and has to be installed. It contradicts the requirements. There was another attempt [Ho17] to create a server-base simulator, but it was not successful, due to inability to run simulation fluently. The issue was related to using a stateful server. Then the decision was made, to develop a new simulator, which satisfies our requirements. For implementation, TypeScript was picked, which is the better version of JavaScript, because it is typed. The simulator which is working on the front-end gives much smoother and fluent experience for a user. Therefore, a fully autonomous front-end is used for the simulator and a back-end during the preparation phase of the controller for the simulator. For the controller implementation of the simulator, has been decided to use WebAssembly. WebAssembly is a new type of code, that runs in modern web browsers. It is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages such as C/C++, so that they can run on the web. It is very well suited to this task due to the fact, that it is used EmbeddedMontiArc generator which converts the code to C++. Especially for this purpose, the EmbeddedMontiArc to WebAssembly converter was developed. The architecture more clearly and fully is shown in the figure 4.1.

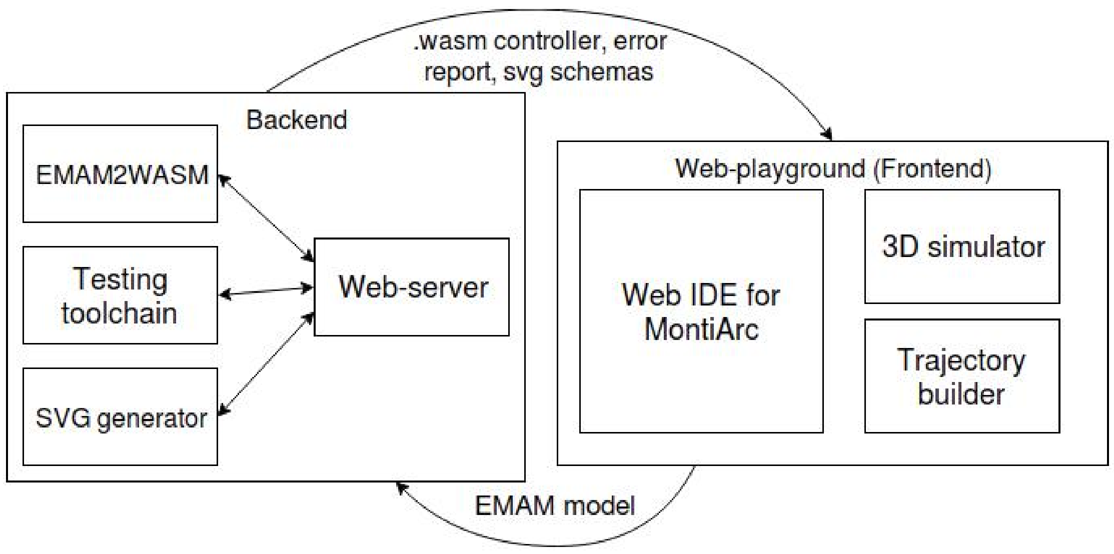


Figure 4.1: The toolchain architecture.

To clarify the goal of each component which is shown in the figure, it will be considered the seven most important components that are linked together:

1. IDE for EmbeddedMontiArc language, it helps to write components easier, reveals the errors and shows incoming and outgoing ports of the components.

2. Web server, it receives the requests for compiling the EmbeddedMontiArc models and sends back a compiled controller, packs and extracts models, controls the compilation process, providing an error handling for users. The server provides the handling of multiple users simultaneously.

3. EmbeddedMontiArc's WebAssembly generator, it gets a model from the web server and compiles it, generating the WebAssembly file, which is the "brain" of the 3D simulation.

4. Testing toolchain, which provides stream testing for incoming models. The toolchain is consist of EmbeddedMontiArc's C++ generator, which generates tests. Then the tests are compiled and executed. The output of the stream testing phase could be used to be shown to the user or be the condition for generating the .wasm file.

5. Visualisation generator, it generates a schema of the components and connections for better readability. Users can easier find errors using a schema of components.

6. Simulator, it receives a compiled model from the server and instantiates it directly in the browser. Then, the controller is used to process data from sensors, which are located on the car.

7. Trajectory builder and comparator. It builds in real time a trajectory of the car's movements and does a comparison between a sample trajectory and generated one. The comparator allows having some deviation from the sample trajectory.

In this composition of components, some of these were reused from the previous successful development. The missing components will be implemented, in the way to accomplish the goal in the most efficient and optimal way.

4.2 Front End

The front end part of the toolchain consists of the Online IDE, 3D simulator and trajectory builder. The Online IDE has been already implemented and successful used in production. In the following subsections will be described the main page, which manages a communication with the server, loading of tutorials and files' operations. Then the 3D simulator will be presented. It demonstrates the car's behaviour, which is controlled by the compiled controller.

4.2.1 Main Page

The main page (see Figure 1 and Figure 2 in Appendix) consists of four `iframe` tags, which have inside independent web-pages: online IDE, trajectory comparator, 3D web-simulator and description of tutorials. Moreover, it ensures the communication between these pages and the Back-End. The functionality which is provided by the main page:

- Select and load tutorials. One of the given tutorials can be selected. After that the specified description of the tutorial will be loaded from the server. Due to lack of space on the screen, there is the simulator with the trajectory builder or the tutorial description can be shown, not all together.

- Read the data from the Online IDE. It means, read files, which were created by the user inside the IDE. All files, that were created, are stored in the Virtual File System. If you need to read the data from these files, the special API [Fil18] has to be used. In this case, an arbitrary number of files and folders can be created. Due to this reason the recursive algorithm has to be implemented for a model reading. Futhermore, JavaScript is an asynchronous programming language and the promises [JSP18] have to be used, to implement the sequential execution of operations.

- Pack files, which have been read, to decrease the size of the blob and amount of transferred files. To pack files directly in the browser, the special library is used. The file reading functionality, which was described above produces an array of the files, which is used by the JSZip library [JSZ18] to create a blob.

- Send and receive the data to/from the server. When a model has been read and packed, it should be sent to the server for further processing. For this purposes `XMLHttpRequest` [XML18] is used. It uses `POST` request to send the blob, which was created by the JSZip library, to the server. The response is sent asynchronously. It means, we can continue to interact with the page during the processing the data by the server. When the data was processed on the server and sent back to the browser, the request object has changed the state to the readyState and it starts to receive the binary data and stores it.

- Unpack the incoming data. The browser receives a binary large object, which is actually just a binary data. Then it has to be saved as a ZIP archive in a memory. After that, the archive is being extracted. The archive consists of two files. One of them is a WebAssembly file, which has `uint8array` encoding. Another one is a JavaScript file, which has string data inside. It is important to differentiate the encoding of these files, due to data consistency inside. During the unpacking process the correctness of received data is checked.

- Transfer the compiled model data to the simulator. After the extraction of the received from the server data, the WebAssembly binary data is validated, by build in WebAssembly validation function. Only after successful validation, the data is being transferred to the 3D web-simulator, where the WebAssembly controller will be instantiated for further usage.

- Load a completed text model of the controller into the IDE. There is possible to load the model, which was implemented as a solution for a tutorial. To load the data to the IDE, the Filesystem API is used, besides, extra functions are required. One of them is cleaning the online IDE environment. It means, delete all files, which were loaded or created earlier. And there is the same issue like we had before with reading. There is no an implemented recursive function, which can do it. And it has to be implemented.

- Do not send loaded solution controller to the server if there is no changes in the code. After loading the solution controller, the content of the files can be changed in the IDE. But if the files were not changed, we do not want to spend time waiting the compilation process. Due to this reason, there is pre-compiled controllers, which can be loaded directly to the simulator, to increase usability and convenience of using the product.

All described functions have an error handling. If users have some problem with a EmbeddedMontiArc model, an error message from the server will be received. It is related to all functions, which is implemented. It could be an error during packing/extracting process, file reading or WebAssembly file validation.

4.2.2 Tutorial's Internal Structure

Each tutorial has a textual description, solution and configuration file. The tutorial description has a detailed explanation of the task, which includes some advices and recommendations. The solution gives the step-by-step instructions how to solve the task, and describes why we use particular methods. Listing 4.1 presents an example of the configuration file for tutorials.

```
{
    "car" :
    {
        "start" : "0:0",
        "end" : "z43"
    },
    "trackObject0" :
    {
        "position" : "100:10",
        "width" : "4",
        "height" : "5",
        "type" : "car"
    },
    "trackObject1" :
    {
        "position" : "z24b",
        "width" : "1",
        "height" : "1",
        "type" : "cone"
    }
}
```

Listing 4.1: Configuration file example.

The configuration is given in JavaScript Object Notation format. The first object `car` is the main car, which controlled by a controller. It is possible to define `start` and `end` positions. When the car reaches the `end` point, the simulation is stopped. If the simulation should run until a controller terminates the execution, `end` point must not be given. The position of objects can be given as coordinates (x,y) or labels, which are written in the track. The next objects are defined like `trackObjectX`, `X` means a number of an object, and have the list of parameters:

- Position of the object, which can be given in the same way like `start` point of the car.
- Width and height define the size of the object, values given in meters.
- Type defines which kind of 3D Objects will be shown in the 3D environment.

Usage of configuration files is simplify the creation process for new tutorials and managing of objects on the track. To create a new tutorial, we shouldn't change anything in the simulator, just specify new configuration for the tutorial.

4.2.3 3D Web Simulator

The simulator uses Three.js [Thr18], that is a lightweight cross-browser JavaScript library/API used to create and display animated 3D computer graphics on a Web-browser. This library uses WebGL [Web18a] to increase performance and smoothness of 3D graphics and reduce the intensity of CPU usage, by using the hardware acceleration of the GPU. Let us consider the basics of three.js library. To be able to display anything using the

library, we need three the most important things: a scene, camera and renderer, so that we can render the scene with camera. Then we can add objects to the scene. The most important objects for our tutorials are: track, car and other objects, depends on the task(e.g., cones, cars).
The simulator environment consists of the visual and mathematical model. The visual means a graphical representation, that can be seen in a browser. The mathematical one, it is a model, base on which the graphical representation is built. Then, there is another important part of the chain - a controller. The controller is built in advance, before the simulation has been started. In the figure 4.2, an interaction between the components is depicted.

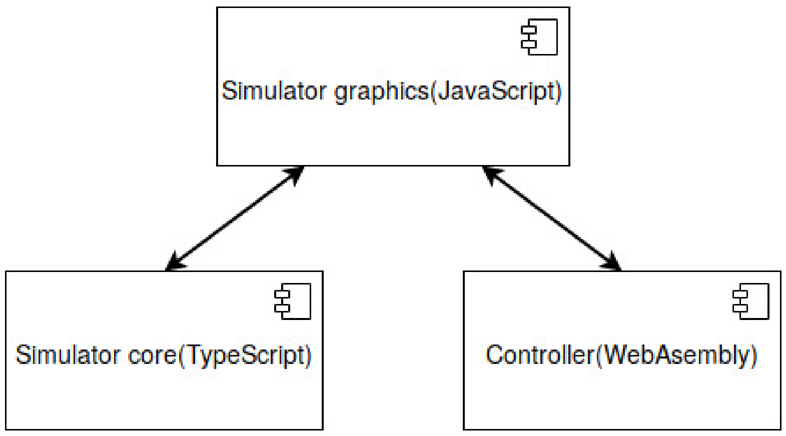


Figure 4.2: The simulator components' interaction.

It is very important, that we separate the implementation of the simulator from the visualisation and controller. It benefits by being able to replace one of the components. We can have a different visualisation e.g., using another graphical library, or change the simulator core. It can be written using another language or implement more advanced mathematical model.

Simulator Graphical Part

The graphical part of the simulator is in charge of displaying the simulation process in the 3D environment. The most important part is a main loop. The loop defines how often the `move` function in the scene is called. It changes the position of the car in the track. Moreover, inside the loop, the simulator core interacts with the controller. On every cycle, the following sequence of actions is taking place:

1. Read environment data. It contains information, from the car's sensors, velocity, position and execution time.

2. Transfer the data from the environment to the controller, which calculates the next move.

3. Use the calculated data in the controller to change the car's position in the mathematical environment of the simulator's core.

4. Use the processed data from the simulator's core to change the car's position in the 3D environment.

The sequence of the actions given above is a loop, which is repeated during the execution in the 3D simulator. There are lots of different functions inside the graphical part of the simulator, which do some service tasks. There is shortly described the most important ones:

- Initializing the simulator's core and controllers.
- Resetting the simulation environment after execution, if it is needed to execute the simulation again. All variables, like execution time or sensor's noise have to be set to initial value.
- Reading tutorials' configurations. During loading one of the tutorials, the configuration file is used for instantiation of the track environment. It means, adding the objects to the track and setting up the initial position of the car.

The graphical part of the simulator is tightly coupled with the core part. It maps all actions which occurs in the mathematical model to the 3D virtual model, which we can observe during the simulation.

Simulator Core

The next paragraphs consider the the simulator's core in more details. The simulator has mathematical model of the track. It means, there is an array of walls, which constantly defined. The track has two different types of the walls: linear and curved. A linear one is defined by two points and a curved one has more parameters: start, middle, end points and radius. When the distances to the track's walls have been calculating, in fact, the following occurs:

- An intersection of a ray and line is calculated. The initial point of the ray is a sensor position and direction. The line is a wall of the track.
- If the intersection was found, then we calculate the distance between the sensor and intersection line. If it was not found, we assume that intersection point exist but with the value `Number.MAX_VALUE`.

After the calculation of the distances to the track's walls, this data transferred to the controller, which uses it to consider the subsequent behavior of the car. Then the controller passes the result again to the simulator's core, which uses this data for further processing. The simulator's core has the function - calculate, which uses the data from the controller and executes the next simulation step in the mathematical model. This function controls the main simulation parameters:

- Time:
  $$time = time + samplingTime \tag{4.1}$$
  where time is simulation time and samplingTime is an increment on each step.
- Velocity:
  $$velocity = velocity + acceleration \cdot samplingTime \tag{4.2}$$
  where velocity and acceleration are related to the car.

- Rotation:
$$rotation = rotation + steering \tag{4.3}$$
where rotation and steering angle are related to the car.
- Position:
$$x = x_0 + velocity \cdot samplingTime \cdot cos(rotation) \tag{4.4}$$
$$y = y_0 + velocity \cdot samplingTime \cdot sin(rotation) \tag{4.5}$$
the x_0 and y_0 are defined the previous position of the car.

After the execution of this function, the outgoing data is ready to be used by the graphical part of the simulator and during the next execution step. To better understand how the core simulator works, consider a figure 4.3. It depicts classes' dependencies and interaction between them.

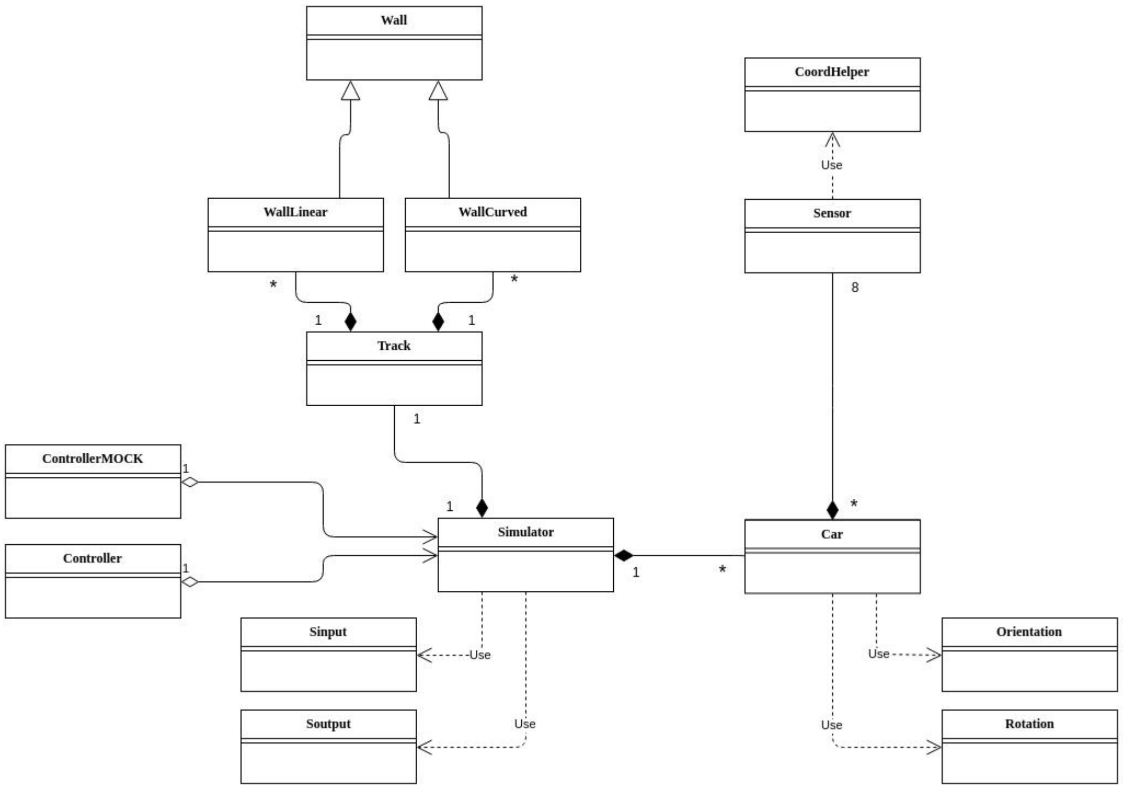


Figure 4.3: The simulator's core class diagram.

The main simulator class has a class `Track`, which has `Walls` and they can be linear or curved. Moreover, the simulator class has a class `Car`, which has eight embedded sensors. The sensors are using the auxiliary class `CoordHelper`, which helps to calculate the distances. The `Orientation` and `Rotation` classes are used by the `Car` class, to calculate the positions of the sensors when the car changes the angle on the flat. The `Simulator` class uses the structures `Sinput` and `Soutput` to fill the input and output data respectively, every execution loop. The simulator works in cooperation with the controller, but for the testing purposes, the MOCK [Dog18] controller - `ControllerMOCK` has been developed. It helps to test the simulator functionality and body of the MOCK controller can be written in TypeScript.

4.2.4 Trajectory Builder

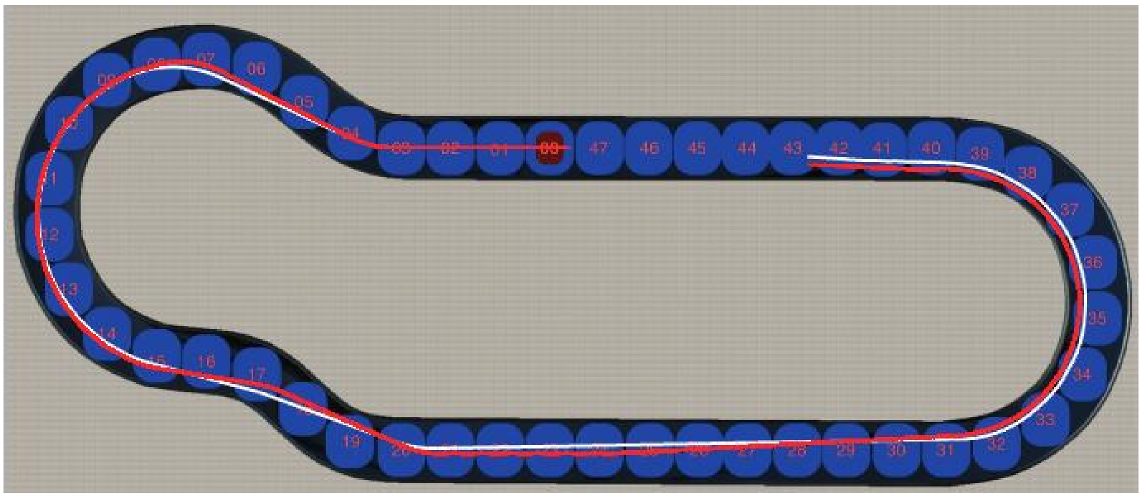


Figure 4.4: Trajectory builder.

Trajectory builder gives a better understanding and a simplicity of analysis the passed way. It is hard to track the whole path of the car and see the controller's errors in real time during the execution. The trajectory helps to solve this problem. In the figure 4.4 is shown the track with the trajectories. The white one is the sample trajectory while the red one is being drawn in real time, simultaneously with the passing the part of the track by the car. To be able to draw the trajectory a log file is used. It contains all important information about the car in each simulation moment. Information from the log file can be used to analyse the behaviour of the car and reveal problems with the model.

```
logObject.telemetry.push({
  "Status": simulator.output.triggerStatus,
  "Time": simulator.output.ti.value,
  "Position": [simulator.output.xi.value, simulator.output.yi.value],
  "Rotation": simulator.output.degree.value * 180 / Math.PI,
  "Velocity": simulator.output.velocity.value,
  "Sensors": distances
});
```

Listing 4.2: Simulation log data.

The listing 4.2 shows the information which is stored in the log. `Time` shows the simulation time. `Position`, `Rotation` and `Velocity` related to the car. `Sensors` is an array of sensors with measurements in the current simulation moment. To build the trajectory, only positioning data is used. The length of the trajectory is calculated by summing up the lengths of all segments between position points. The length between points is calculated using the distance formula [Dis18]:

$$distBetweenPoints \quad += \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2} \tag{4.6}$$

After the summing up all the segments, the *distBetweenPoints* has the entire distance of the track, which is used later to compare the trajectories. To draw the trajectory, all points from the log file are mapped to the picture of the track.
Futhermore, we can analyse the movement trajectory, using the the acceleration vectors in each point. It reveals issues related to twisting motion of the car. It can be caused by incorrect computation of a car's angle. Figure 4.5 shows an example of this kind trajectory.

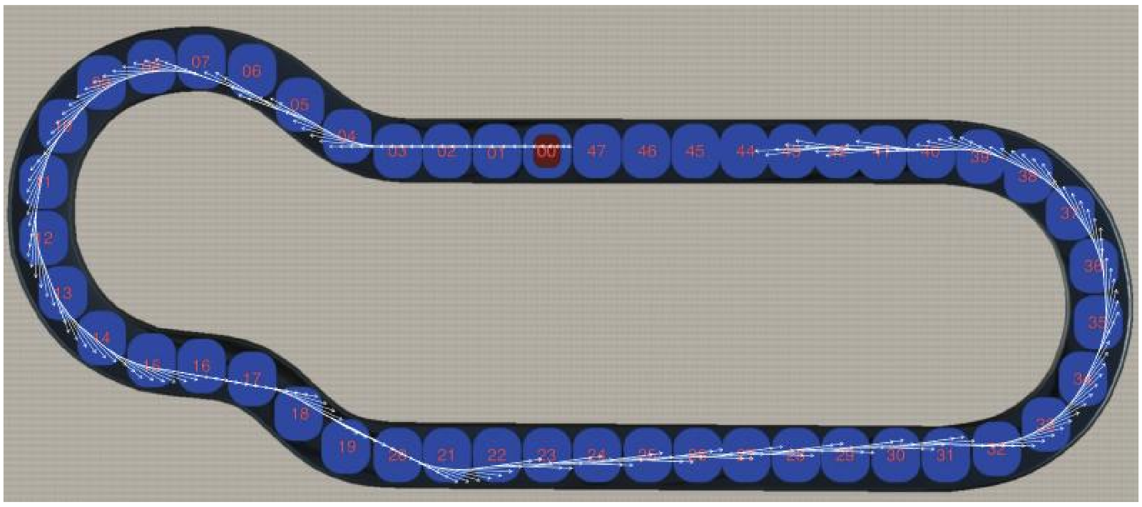


Figure 4.5: Trace analyser.

A vector's angle is calculated using the following formula [Ata18]:

$$angle \quad = \quad atan2(y_1 - y_0, x_1 - x_0) \tag{4.7}$$

Where the (x_0, y_0) related to the point, which is a start point for the acceleration vector and (x_1, y_1) the next point. Then the length of the vector is calculated using the standard distance formula, which is shown before. Figure 4.5 illustrates an example of the incorrect controller behaviour and ability to see this. The constantly changing the steering angle during passing the sectors 21-31 on the track. This analysis method is not computationally expensive but very efficient for visual analysis purposes.
Finally we have to analyse a trajectory after passing the required part of the track to decide whether the task is solved correctly or not. The formulas shown in 4.8 to 4.11 are used.

$$D_1 = \sqrt{(C1_x - S1_x)^2 + (C1_y - S1_y)^2} \tag{4.8}$$

$$D_2 = \sqrt{(C2_x - S2_x)^2 + (C2_y - S2_y)^2} \tag{4.9}$$

$$sectorArea = \sqrt{((C2_x - C1_x)^2 + (C2_y - C1_y)^2) * 0.5 * (D_1 + D_2)} \tag{4.10}$$

$$deviation = \frac{\sum_{n=1}^{sectors} sectorArea_n}{drivenDistance} \tag{4.11}$$

Where `C1`, `C2` are point which belong to current trajectory. `S1` and `S2` belong to sample trajectory. Using the given formulas, the square between four points is calculated. Depends on the deviation from the sample trajectory, the task is counted as solved or failed. Figure 4.6 shows the graphical representation of the calculation.

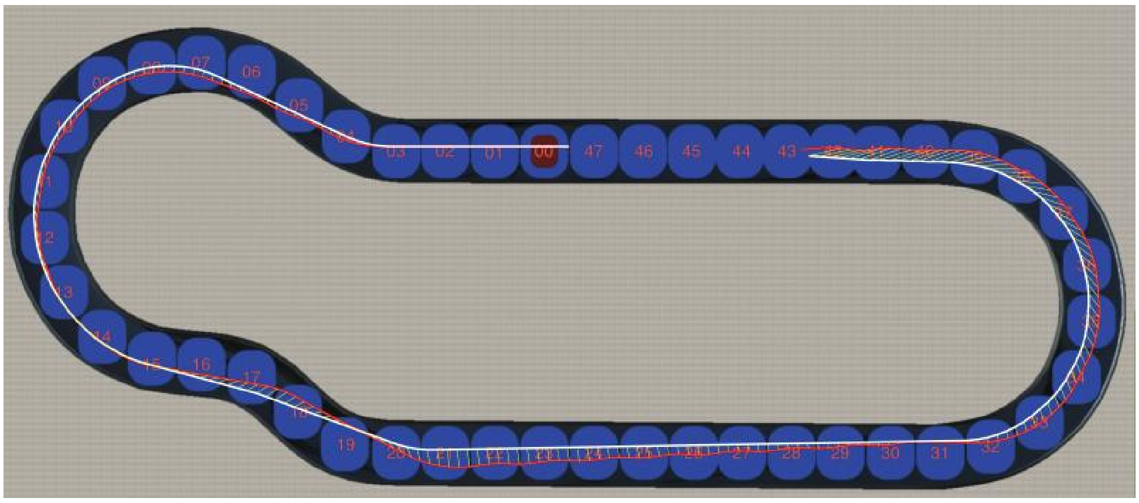


Figure 4.6: Trajectories comparator.

The area, which is marked between the current trajectory and the sample one is a deviation. This square is influence on the tutorial's result, whether it will be passed of failed.

4.3 Back End

The back end part of the toolchain is consist of a web server, EmbeddedMontiArc's WebAssembly generator, testing toolchain and visualisation generator. All described component, except the web server, have been implemented and successful used in previous projects. The main goal to connect them all together and implement the web server, which controls the execution of all services and communicates with the front end. Moreover, in EmbeddedMontiArc language there is a lack of some context conditions checking, which have to be implemented, to obtain better error handling during compilation process.

4.3.1 Web Server

Overwhelming majority of tools related to EmbeddedMontiArc have been written using Java language and use the .jar containers. Using JAR container, the applications can be executed on different operating systems, it means they are cross-platform. This is a big advantage, since applications can be run on the server, which does not use Windows, instead some Linux-based system. The web server is based on Eclipse Jetty [Jet18]. Eclipse Jetty provides a web server and Java Servlet container [HC01, CY03]. In our case, we create a server, which bind to the specific port. The servlets' container holds servlets inside, which react on incoming requests from the front end (c.f. figure 4.7).
What is important in our case, that when the request is coming, for each request, a new thread is being created. It means, that we can maintain multiple users, and do not have to create manually a multiple user management system. Let us consider the steps, which occur on the back end part of the toolchain:

- The servler receives the request from the front end. The request contains an archive, which has to be unpacked and a model has to be copied for the further processing. For this purpose, a class has been created, which in charge of unpacking ZIP archives and

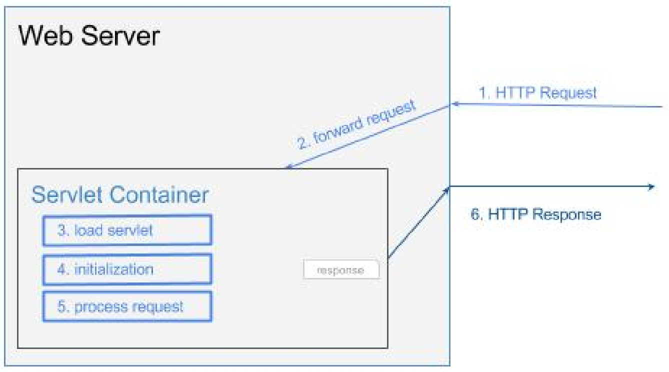


Figure 4.7: Servlet container.

can handle any structure of directories/folders. To decrease a number of input/output operations, the incoming archive is being stored in RAM.

- Then, after the unpacking, the incoming model is being saved to the folder, where models for EmbeddedMontiArc's WebAssembly compiler are located. The compilation process is initiated by executing the particular batch file with parameter, which is a name of the model, that has to be compiled. The batch file initiates the execution of the `emam2wasm.jar` package. Which has a bunch of parameters e.g., which libraries have to be included, the path to the compiled result and other service parameters, that described in the documentation for EmbeddedMontiArc's WebAssembly compiler [EMA18]. The compilation process runs as a separate system process and the web server receives an output form the process. It means, entire console output is processed by the server and if there is an error, it will be sent to the front end, to notify the user about the error during the compilation process.
- If there is no errors and the model has been successful compiled, it has to be packed to decrease the generated traffic. One of these generated files is a JavaScript file, which actually contains textual data inside and has high compression rate. Another class, which is in charge of the packing process, archives the data and return the `zipStream`.
- The given `zipStream` is being written to the response body. The response is marked like a successful performed and is being sent to the front end.
- After sending the response, all temporary data on the server will be deleted, to be sure that one day the server will not run out the disc space, due to the large number of compiled models.

The web server, like other EmbeddedMontiArc tools is a JAR package and can be installed on different operating systems. Currently, the back end part of the toolchain runs on Windows base OS, but in the future the back end could easily migrate to linux-base environment, due to using cross platform java implementation.

4.3.2 Context Conditions

The EmbeddedMontiArc language is based on a context-free grammar [BW84]. It means, the language does not support the context-sensitive restrictions by default. Context conditions implement such context-sensitive restrictions. In our case, it would be very helpful to check the correctness of written models. Better error handling by the compiler, speedup a process of finding errors during the development. New context conditions were implemented to simplify the development process for a user. It was decided to implement the following context conditions:

1. **Range port check assignment :** the assigned value for a port, has to be given in the specified range.
2. **Range value check assignment :** the value assigned to a variable, has to be given in the specified range.
3. **Range value check declaration :** if an assignment of a value done during a declaration (e.g., type range = value), the value has to be given in the specified range.
4. **Units comparison equal, greater, smaller, smallerEqual, greaterEqual, notEqual :** when values are compared using the given operations, the units of co compared variables/ports must be equal.
5. **Units must be equal assignment :** if a value with units has been assigned to some variable/port, the units have to be equal.

To be able to implement given above context condition, the knowledges of the architecture of the language are required. The detailed description of the language architecture and grammar is given in the following book [HR17]. Namely in chapters 6, 9 and 10. Let us consider the most important statements for the implementation, which are defined in the book. In the chapter 9 is written that, the context conditions have to be checked after the creation of the symbol table from an abstract syntax. Then we should check, whether the abstract syntax has been already created or not. The symbol table provides helpful information to implement context conditions in the efficient way. In our case, during the comparison check, it is important to find a left and right hand side of an expression. Then consider this parts separately, and only when the desired data as a value or units have been found, compare them. Searching the data like range of units, it must be checked whether the rage or units have been defined or not, to does not end up with an runtime error during the comparison process. Another important statement, that constants and variables have difference in type, which is obvious from the names. But still, it is important to check all possible combinations during the comparison(e.g., var == const, var == var, const == var, const == const). To increase the code reusability and decrease amount of written code, for the units comparison, was decided to use additional level of abstraction. The helper class, which does all comparison operations, and receives the type of the node(e.g., `ASTMathCompareNotEqualExpression`) like a parameter.
To be able to use the created context conditions, they all have to be added to the `EmbeddedMontiArcMathCoCoChecker`. Only after registering the context conditions, they will be taken into account during the checking process. But before using them, the context conditions must be tested. As was advised in the book, tests for valid and invalid models should be created. It makes sure, that valid models do not violate the context condition, whereas invalid models do violate them. Due to these reasons, for each context condition was created two tests and the global test, which involves all context conditions at the same time. Listing 4.3 presents the invalid global test, which checks different conditions. The valid global test is given in Appendix, Listing 1.

```
 1 package test.coco.invalid.units;
 2 
 3 component MyComponentMix {
 4     ports
 5         out Q(0kg: 100kg) out1;
 6 
 7     implementation Math {
 8 
 9         Q(0kg: 10kg) var1 = 100kg;
10         if (var1 <= 10m)
11             out1 = 10kg;
12         end
13         if (10m < var1)
14             out1 = 10kg;
15         end
16         Q(0m: 100m) var2 = 10m;
17         if (var1 > var2)
18             out1 = 10kg;
19         end
20         if (10kg == 10m)
21             out1 = 10kg;
22         end
23         Q(0kg: 10kg) var3;
24         var3 = 100m;
25     }
26 }
```

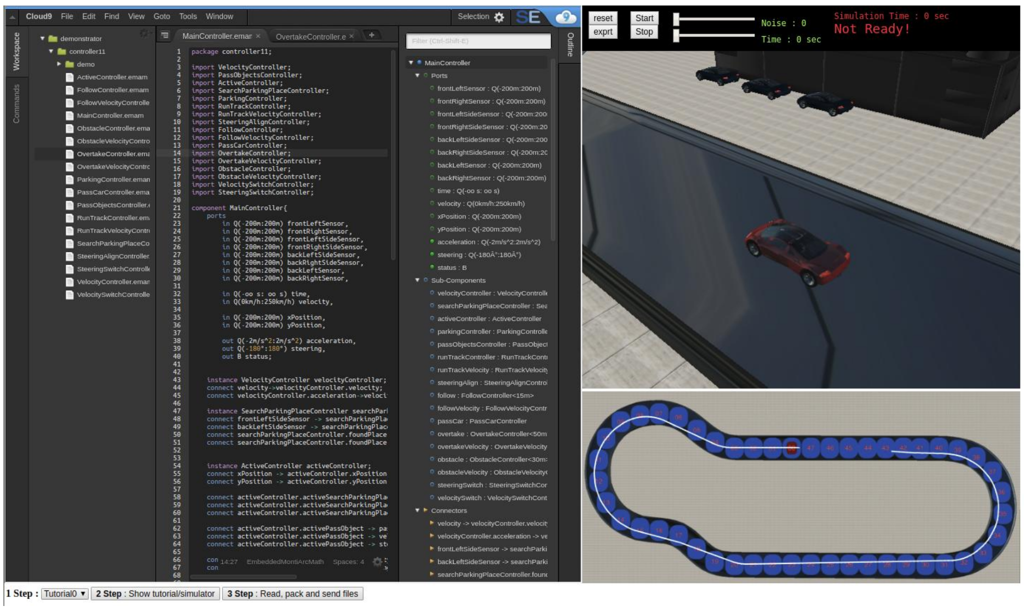
Listing 4.3: Example of an invalid test.

Consider the example of an invalid test. Line 9 has a range error, the value 100 exceeds the values of given range. Lines 10, 14, 19 and 23 have units errors. Each of these statements use different types of comparison. And finally the line 28 has errors related to different units and values simultaneously. The test involves checking not only variables but ports and constants. To be sure that correct comparison have not caused any errors, the similar valid global test is executed. After implementing and integrating the context conditions, the chain of projects has to be recompiled, to be able to use implemented innovations.

Chapter 5

Usage

Students use the web-playground to understand how to work with C&C modeling languages, like EmbeddedMontiArc. The main idea of the playground to increase interest in the learning process using a gamification of the tutorials. There are several simple steps in the learning process. The first tutorial is a task, which already has a solution but the idea behind that to show the main constructions and principles of the language and the playground. Next tutorials have tasks with increasing complexity and have some hints, which motivate students to use particular constructions. The visualization of the process gives the feeling of the language and understanding of the binding between writing the code and real actions which are caused by the written code. The process of writing tests shows the benefits of test-driven development and understanding an importance of independent testing of components. Usage of the web-playground is very simple. Students do not have to install any applications on computers and it is possible to use it from any platform, whether it is Mac, Windows or Linux. Only one important condition has to be satisfied - to have a "fresh" version of a web browser, which supports the WebAssembly. IDE, tutorial, visualization are located in one window and has a very intuitive interface. Figure 2 presents the web interface.



```
package controller11;

import VelocityController;
import PassObjectsController;
import ActiveController;
import SearchParkingPlaceController;
import ParkingController;
import RunTrackController;
import RunTrackVelocityController;
import SteeringAlignController;
import FollowController;
import FollowVelocityController;
import PassCarController;
import OvertakeController;
import OvertakeVelocityController;
import ObstacleController;
import ObstacleVelocityController;
import VelocitySwitchController;
import SteeringSwitchController;

component MainController{
    ports
        in Q(-200m:200m) frontLeftSensor,
        in Q(-200m:200m) frontRightSensor,
        in Q(-200m:200m) frontLeftSideSensor,
        in Q(-200m:200m) frontRightSideSensor,
        in Q(-200m:200m) backLeftSideSensor,
        in Q(-200m:200m) backRightSideSensor,
        in Q(-200m:200m) backLeftSensor,
        in Q(-200m:200m) backRightSensor,

        in Q(-oo s: oo s) time,
        in Q(0km/h:250km/h) velocity,

        in Q(-200m:200m) xPosition,
        in Q(-200m:200m) yPosition,

        out Q(-2m/s^2:2m/s^2) acceleration,
        out Q(-180°:180°) steering,
        out B status;


    instance VelocityController velocityController;
    connect velocity->velocityController.velocity;
    connect velocityController.acceleration->veloci

    instance SearchParkingPlaceController searchParl
    connect frontLeftSideSensor -> searchParkingPla
    connect backLeftSideSensor -> searchParkingPlac
    connect searchParkingPlaceController.foundPlace
    connect searchParkingPlaceController.foundPlace


    instance ActiveController activeController;
    connect xPosition -> activeController.xPosition
    connect yPosition -> activeController.yPosition

    connect activeController.activeSearchParkingPla
    connect activeController.activeSearchParkingPla
    connect activeController.activeSearchParkingPla

    connect activeController.activePassObject -> pa
    connect activeController.activePassObject -> ve
    connect activeController.activePassObject -> st

    con
    con
```

14:27 EmbeddedMontiArcMath Spaces: 4

Figure 5.1: Web interface view.

Then let us consider the standard sequence of steps, which have to be made, to develop a model and see the result. Suppose we have already running operating system and know the URL [Onl18] of the playground. Then the following steps should be performed:

1. Open the web-playground in a browser, enter the given URL.
2. Select the tutorial from the list, which is shown near the label Step 1.
3. Read the tutorial description, using the button - Step 2 : Show tutorial/simulator.
4. Write code with tests, using the Online IDE from the left hand side.
5. Send a model controller to the server, using the Step 3 button, to execute tests and compile the controller.
6. When the simulator displays the ready state, it means that you can run a visualization execution. If the solution contains errors, a student receives an error message with a description.
7. It is possible to restart the simulation process, add some noise to the sensors to emulate more realistic measurements, or specify the period of the simulation process. The noise helps students to feel, the influence of the imperfect environment (through a boiling weather or rain) or just a hardware component which has only a certain accuracy. Students learn the difference between plain software engineering (which mostly has no uncertainties) and software systems engineering (which works by interaction with the environment). Also, a controller for embedded systems must be robust, it is another finding that students can learn. A car must continue driving even when it has some inaccuracies in measurements or not critical errors.
8. After the execution, the current trajectory is compared with the sample solution and the student is notified whether he passed the test or not.

The studying process is built on a concept from simple to complex. Doing the tutorials one by one, students get closer to the main goal, which is a creation of a controller for a self-driving car. The final tutorial summarizes all knowledges which were obtained during the education and uses it for implementing the controller with highest complexity.

Chapter 6

Tutorials

This chapter presents a group of tutorials, which can be solved inside the created environment. The goal of the tutorials to teach students the basics of the EmbeddedMontiArc language and give a general idea how to build the autopilot's models. The tutorials are given with respect of an increasing the complexity level. Examples are explained step-by-step, to give the best studying experience. The tutorials' descriptions are presented in the same way, like they were written for students. Beginner's tutorial shows the basics of the language and simulator. Then, Stream tutorial describes the way of writing test for controllers. The next two tutorials, Parallel parking and Maneuverability test, show the real examples of basic tutorials. The last one, the most complex tutorial, illustrates only the task due to the size of the solution. This chapter shows only a part of the created tutorials, because of the amount of space, which is required for all tutorials.

6.1 Beginner's Tutorial

Task: Accelerate to the given speed.
Implement the model that continuously accelerates to 10 m/s and then stops the simulation.

Explanation of the simulator basics :
The main object, which are controlled by the controller, is a car. The car has 8 sensors to measure distances to obstacles. The figure 6.1 show the sensors location.

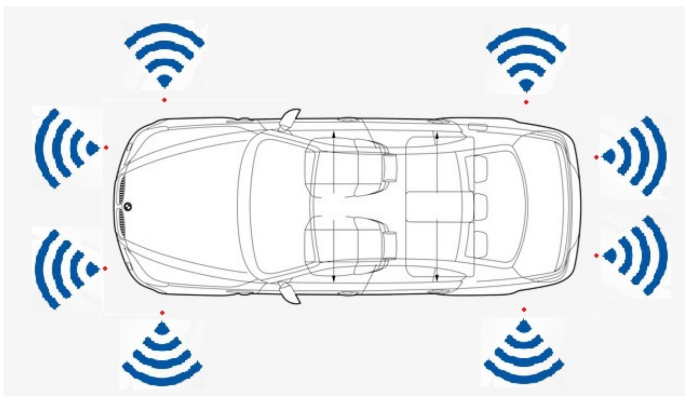


Figure 6.1: Car with sensors.

To solve this task, it is needed to control only an acceleration of the car. Changing the acceleration, you may control the behavior of the car. To be able to reach 10 m/s speed, the car must accelerate continuously until it reaches the desired speed. Let us start with a `MainController`, which defines an interface to the simulator. A new file should be created, which has the same name like component has, with .emam extension.

```
1      package controller;
2
3      component MainController{
4          ports
5              in (-200m:200m) frontLeftSensor,
6              in (-200m:200m) frontRightSensor,
7              in (-200m:200m) frontLeftSideSensor,
8              in (-200m:200m) frontRightSideSensor,
9              in (-200m:200m) backLeftSideSensor,
10             in (-200m:200m) backRightSideSensor,
11             in (-200m:200m) backLeftSensor,
12             in (-200m:200m) backRightSensor,
13
14             in (0s:oo s) time,
15             in (0m/s:25m/s) velocity,
16             in (-200m:200m) xPosition,
17             in (-200m:200m) yPosition,
18
19             out (-2m/s^2:2m/s^2) acceleration,
20             out (-180°:180°) steering,
21             out  status;
22     ...
23     }
```

Listing 6.1: MainController interface.

The first eight incoming ports (cf. ll. 5-12) receive the data from the sensors which are shown in the figure 6.1. The next port is a simulation `time`, which starts from zero second to infinity. The textttvelocity incoming port shows a current velocity of the car. The next two ports give a position of the car on a track. After the incoming ports, the outgoing are following. Outgoing ports control a behaviour of the car using the data which is coming to the incoming ports. The `acceleration` port controls the velocity, it is like you have two pedals in a car. If you push a brake pedal, you have a negative acceleration. But if you push an accelerator pedal, the car accelerates and velocity has been increased. Then following the `steering` port, which controls the rotation of the car. And finally the `status` port, which signals the end of a simulation process. The reason to finish the simulation can be a successful completion of an assignment or violating by the car the track or a map borders. After the examination of the example, we should notice:

- Component has incoming and outgoing ports.
- For each port must be specified a type (like `Q`, `B`) with a valid range.
- After each port's name has to be a comma and the last one must have a semicolon.
- The most common units are:
  - Distance: meters (`m`), kilometers (`km`), miles(`ml`) and so on.

  – Time: seconds(`s`), minutes(`m`), hours(`h`).
  – Velocity: `km/h`, `m/s`, `mi/h` and so on.
  – Acceleration: m/s^2
  – Rotation: degrees(°)

It was the default interface for the simulator. It has to be define for all feature controllers. Then you should create your own components which will be connected to the `MainController`. Let us create a simple component and connect it to the main one. To do that, we have to create a new file .emam with the following content:

```
    package controller;

    component ExampleController {
      ports
        in  (0m/s:25m/s) velocity,
        out (-2m/s^2:2m/s^2) acceleration,
        out B status;

      implementation Math {

        if (velocity <= 10 m/s)
              acceleration = 1m/s^2;
          else
            status = true;
            end
      }
    }
```

Listing 6.2: ExampleController.

In the `ExampleController` component, there are one incoming port and two outgoing ones. Firstly, we should reach the speed 10 m/s, then stop the simulation. The logic of the controller is implemented inside the `Math` scope. Inside the scope, `if-else-end` construction is given and an example how to use it. Moreover, in the `Math` scope can be used various constructions, which you will see later in the next tutorials. The logic of this controller is straightforward: if the velocity of the car is less then 10 m/s, then speed up the car with the acceleration 1 m/s. When the given velocity is reached, stop the simulation process by setting up the status port to `true`.
After the creation of the `ExampleController`, we should import it inside the main one and then instantiate it, to have possibility to use it, like a part of the `MainController`:

```
    package controller;

    import ExampleController;

    component MainController {
    ...
    instance ExampleController exampleController;
    ...
    }
```

Listing 6.3: MainController import.

Listing 6.3 shows the importing of the `ExampleController` (cf. l. 3). Later, after specifying the ports of the `MainController`, we have instantiated the controller with the type `ExampleController` and the name exampleController. Now we should connect this controller to the main controller using specified ports.

```
    ...
    connect velocity->exampleController.velocity;
    connect exampleController.acceleration->acceleration;
    connect exampleController.status->status;
}
```

Listing 6.4: MainController connection.

We have connected the incoming port, `velocity` of the `MainController`, to our instantiated controller and it's corresponding incoming port `velocity`. Then, we have connected the outgoing port of `velocityController.acceleration` to the outgoing port of our `MainController`. Lastly, the `status` port of the `ExampleController` to `status` of the `MainController`.
Finally the connections' scheme should look like depicted in figure 6.2.

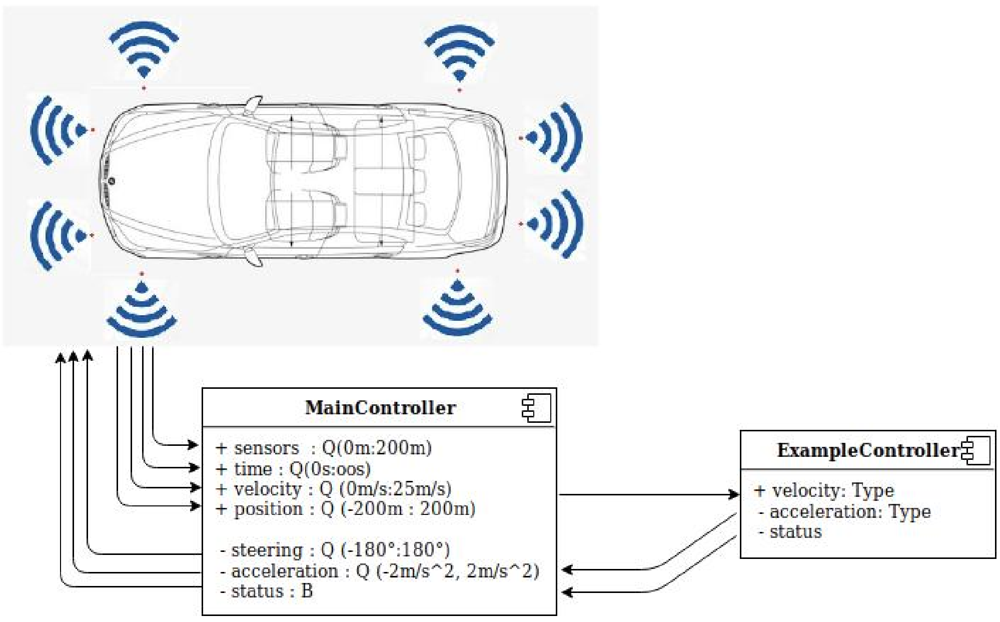


Figure 6.2: Car with controller.

Eventually, these files have to be sent to the server to process them and later execute in the simulator. During the execution, we can observe how the car behaves using our first created controller.

6.2 Stream Testing Tutorial

EmbeddedMontiArc generator has a built-in support of stream tests. The stream tests allow to define a sequence of input values for C&C input ports and the expected output sequences for output ports. It helps to carry out unit and integration testing of C&C models [KRB96]. That gives the big advantage in the development process, test driven development [TDD18, Bec03] can be applied, to create models for the simulator. It means, that we should start from an implementation of tests firstly, even before we created the controller. After that, a controller,which satisfy the tests, has to be implemented. This is the main principle of test driven development. Let us start with simple example:

```
package simulatorModel;

    stream VelocityControllerTest for VelocityController {
        velocity: 0m/s tick 10m/s tick 11m/s;
        time: 0s tick 10s tick 11s;

        acceleration: 1m/s^2 tick 1m/s^2 tick 0m/s^2;
}
```

Listing 6.5: Stream test example.

Now, we are going to analyse the listing 6.5. From the very beginning, we have to declare the package name. If we have a chain of folders, then they have to be separated by points e.g., `folder1.folder2.folderWithTests`. Then, the reserved word `stream` is written, which is followed by the name of the test. It has to coincide with the file name, where the test is written. After the reserved word `for` must be written the actual controller's name, which is tested. Inside the test scope, incoming and outgoing ports of the controller should be given. The controller has three ports: two incoming `velocity` and `time` and one outgoing - `acceleration`. The reserved word `tick` divides discrete data on chunks. Then, the first value for `velocity` is 0 m/s and 0 seconds for `time`. After, we should have the respective value on the `acceleration` port. It has to be 1 m/s^2 in this case.
When the test for the VelocityController has been created, it is time to implement the controller. To satisfy the test, the controller could be like is shown in the listing 6.6.

```
package simulatorModel;

component VelocityController {
    port
        in  (0m/s : 25m/s) velocity,
        in  (0s:oo s) time,
        out (-2m/s^2:2m/s^2) acceleration;

    implementation Math{

        if (velocity > 10 m/s)
            acceleration = 0 m/s^2;
        else
            acceleration = 1 m/s^2;
        end
    }
}
```

Listing 6.6: Velocity controller for stream testing.

Currently, if we pass 0 m/s into `velocity` port, then there is 1 m/s^2 for the outgoing `acceleration` port. If it is true, the controller passed the test.

6.3 Parallel Parking Tutorial

Task: To carry out a parallel parking between two cars.
Implement a model that manages a parking between two cars. The model has to have several modules which are responsible for different actions. One of the examples could be:

1. Module which controls the speed of the car, depends on the current action(e.g. parking, searching a parking place).
2. Module which looking for a gap between cars for the parking.
3. Module which controls a steering of the car during the parking process.

Each module should be as simple as possible.
To solve this task you should use at least 6 sensors, which measure the distance to objects which are located in front, left-side and back of the car. The speed has to be between 0.5 m/s and 1.0 m/s. The figure 6.3 shows the main idea how the parking has to be done:

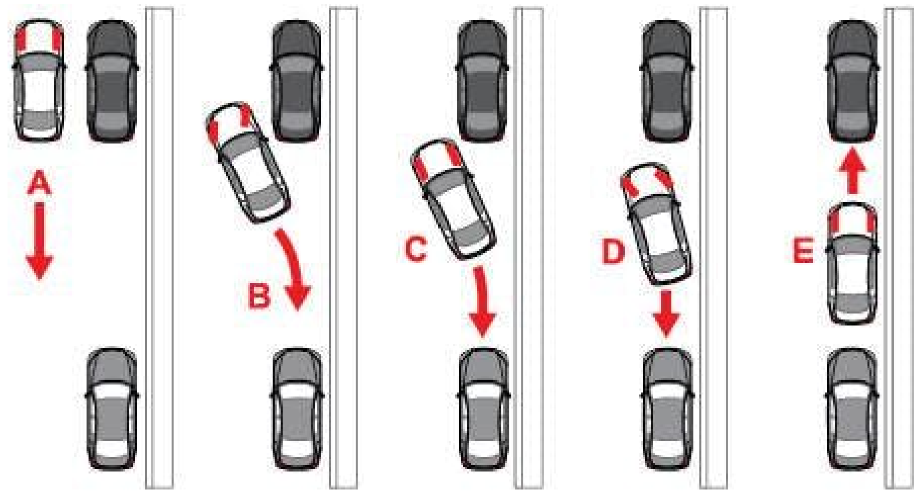


Figure 6.3: Parking process.

The parking process can be divided into four steps:

1. Go straight until a place have been found.
2. When the place is found, go back and rotate the car.
3. When the car has reached appropriate point, rotate car in other direction to fit into given gap between the cars.
4. When the car is close to the car behind, stop and go forward until it reaches certain distance to the front car.

Important to understand that the wed-simulator is a simplified version of the more powerful simulator [GKR^{+}17]. When the car is turning, it does not change an angle of the front wheels but the angle of the car entirely. It has to be taken into account during controllers' development process.

Solution

To solve the task, the `MainController` has to be used, like we did in the previous tutorials. But let us start from an implementation of a controller which searches a place for the car, between two other cars, to carry out the parallel parking. The idea is to pass the first car and find the "end" of the second car to start parking process form a right point, like it is shown in the figure 6.3, step A. Consider one of possible implementations of the controller, listing 6.7 illustrates it.

```
 1 package controller03;
 2 
 3 component SearchParkingPlaceController {
 4     port
 5         in (-200m:200m) frontLeftSide,
 6         in (-200m:200m) backLeftSide,
 7         out B foundPlace;
 8 
 9     implementation Math{
10 
11         static B passed0 = false;
12         static B passed1 = false;
13         static B passed2 = false;
14         static B passed3 = false;
15 
16         if ((backLeftSide - frontLeftSide) > 3m)
17             passed0 = true;
18         end
19         if (((frontLeftSide - backLeftSide) > 3m) && passed0)
20             passed1 = true;
21         end
22         if (((backLeftSide - frontLeftSide) > 3m) && passed1)
23             passed2 = true;
24         end
25         if (((frontLeftSide - backLeftSide) > 3m) && passed2)
26             passed3 = true;
27         end
28 
29         foundPlace = passed3;
30     }
31 }
```

Listing 6.7: Searching parking place controller.

The `SearchParkingPlaceController` has two incoming and one outgoing port. The incoming ones are receive the data from the `frontLeftSide` and `backLeftSide` sensors. The out going port `foundPlace` indicates that the place for parking has been found. The side sensors are used to measure the distances to the cars and the side of the road. Inside the `Math` scope, the group of `static` variables defined. Using `static` variables the states can be saved. Static variables are initialized only once. It means, `false` value (cf. ll. 11-14) initializes the variables `passedX` only once. The following executions, `false` does not influence on the variables. Using these static variable, the states, from the analysing the environment from the left hand side of the car, will be saved. In our case we are measuring the difference between the `frontLeftSide` and `backLeftSide` sensors. If the value on the `backLeftSide` sensor greater then the value on the `frontLeftSide`, then the front part of the car passed the back of the parked car by, and the state will be saved. Then the same technique is applied, measuring the differences in sensors' values to find the parking lot and reach the state A, on the figure 6.3. Then, from this position the parking process can be started. But this controller only searching for a parking lot and we should control the car driving. We should implement a `VelocityController`, which controls the speed of the car, during the searching of parking place and carrying out the parking of the car. Listing 6.8 demonstrates the controller.

```
1   package controller03;
2
3   component VelocityController {
4       port
5           in  (0 m/s: 25 m/s) velocity,
6           in  B reverseMove,
7           in  B moveForward,
8           out (-2m/s^2: 2m/s^2) acceleration;
9
10      implementation Math{
11
12          if (velocity > 1 m/s)
13              acceleration = 0 m/s^2;
14          else
15              acceleration = 1 m/s^2;
16          end
17          if reverseMove
18              acceleration = -0.5 m/s^2;
19          end
20          if (velocity < -0.5 m/s)
21              acceleration = 0 m/s^2;
22          end
23          if (reverseMove && moveForward)
24              acceleration = 0.5 m/s^2;
25          end
26          if (reverseMove && moveForward && (velocity > 0.5 m/s))
27              acceleration = 0 m/s^2;
28          end
29      }
30  }
```

Listing 6.8: Velocity controller.

The controller has three incoming ports and only one outgoing. The `velocity` port helps to control the speed of the car. In general, the velocity of the car is controlled by acceleration, it is like an acceleration and brake pedals. Measuring the current velocity we can change the acceleration and doing that control the speed. Other two incoming ports have influence on the direction and speed of the car, during the different stages of the parking process. Consider the `Math` scope. There is implemented three different phases of the parking. The first one is a limit for the velocity during searching parking place (cf. ll. 12-16). The next one is the velocity limit during the reverse move, when the car moves back during the parking (cf. ll. 17-22). The last one, when the car moves forward during the parking process, to be closer to the front car (cf. ll. 23-28), state E on the figure 6.3. When we can control the velocity of the car, continue with the most important controller, which manages the parking process. The listing 6.9 shows it.

```
 1 package controller03;
 2
 3 component ParkingController {
 4     port
 5         in  (-200m:200m) frontLeft,
 6         in  (-200m:200m) frontRight,
 7         in  (-200m:200m) frontLeftSide,
 8         in  (-200m:200m) backLeftSide,
 9         in  (-200m:200m) backLeft,
10         in  (-200m:200m) backRight,
11         in  B reverseMove,
12         out (-35°:35°) steering,
13         out B moveForward,
14         out B status;
15
16     implementation Math{
17
18         static B forwardState = false;
19
20         if reverseMove
21             steering = 1°;
22         else
23             steering = 0°;
24         end
25         if (reverseMove && (backLeft < 2m))
26             steering = -1°;
27         end
28         if (reverseMove && ((backRight == backLeft) ||
29             ((backRight < 3m) && (backLeft < 3m))))
30                 forwardState = true;
31         end
32         if(forwardState &&(frontLeftSide > backLeftSide) &&
33             frontLeftSide != backLeftSide)
34                 steering = -0.5°;
35         else
36                 steering = 0.5°;
37         end
38         if (((frontRight < 3m) || (frontLeft < 3m)) &&
39             forwardState)
40             status = true;
41         else
42             status = false;
43         end
44         moveForward = forwardState;
45     }
46 }
```

Listing 6.9: Parking controller.

The controller receives data from the six sensors, which is used to control the phases of the parking. The implementation inside the `Math` scope we can divide into three parts. The first one (cf. ll. 20-31), the car is going back and when the distance from the back left sensor to the road border, is less then 2m, it starts to rotation in the opposite direction, until it is in parallel with the road or too close to the car behind. Then the car stops if the limit of the distance from the back is reached and changes the state of the `forwardState` to `true`. After that, the second part (cf. ll. 32-37) of the controller involved into parking process. The car is getting closer to the front car and at the same time aligns the body to the road. The last part (cf. ll. 38-43) is in charge of finishing the simulation in time, when the car is enough close to the front car. The stop of the simulation happens when the `status` set to `true`. All these steps are nicely illustrated in the picture 6.3. Finally, we should instantiate all these controllers in the MainController and then connect to the interface.

```
 1  package controller03;
 2
 3  import VelocityController;
 4  import SearchParkingPlaceController;
 5  import ParkingController;
 6
 7  component MainController{
 8      ports
 9          in (-200m:200m) frontLeftSensor,
10          in (-200m:200m) frontRightSensor,
11          in (-200m:200m) frontLeftSideSensor,
12          in (-200m:200m) frontRightSideSensor,
13          in (-200m:200m) backLeftSideSensor,
14          in (-200m:200m) backRightSideSensor,
15          in (-200m:200m) backLeftSensor,
16          in (-200m:200m) backRightSensor,
17
18          in (0s:oo s) time,
19          in (0km/h:250km/h) velocity,
20          in (-200m:200m) xPosition,
21          in (-200m:200m) yPosition,
22
23          out (-2m/s^2:2m/s^2) acceleration,
24          out (-180°:180°) steering,
25          out B status;
26
27      instance VelocityController velocityController;
28      connect velocity->velocityController.velocity;
29      connect velocityController.acceleration->acceleration;
30
31      instance SearchParkingPlaceController searchParkingPlace;
32      connect this.*->searchParkingPlace.*;
33
34      instance ParkingController parkingController;
35      connect this.*->parkingController.*;
36      connect searchParkingPlace.foundPlace-> [
37          velocityController.reverseMove,
38          parkingController.reverseMove];
39      connect parkingController.*->*;
40      connect parkingController.*->velocityController.*;
41  }
```

Listing 6.10: MainController.

Listing 6.10 shows the `MainController`. To be able to use the created controllers, they have to be instantiated (cf. ll. 3-5). Then the controllers have to be instantiated and connected to the corresponding ports(cf. ll. 27-40). The star symbol(cf. l. 33) means that the ports, which have similar name are connected to each other. It simplifies the connection schema, decreases the size and readability. The square brackets(cf. ll. 36-38) denote an array of ports. In this particular case, it means, `foundPlace` port is connected to the `reverseMove` ports of the `velocityController` and `parkingController` simultaneously. To simplify the visualisation of the connected controllers, the entire connection scheme was depicted in the figure 6.4.

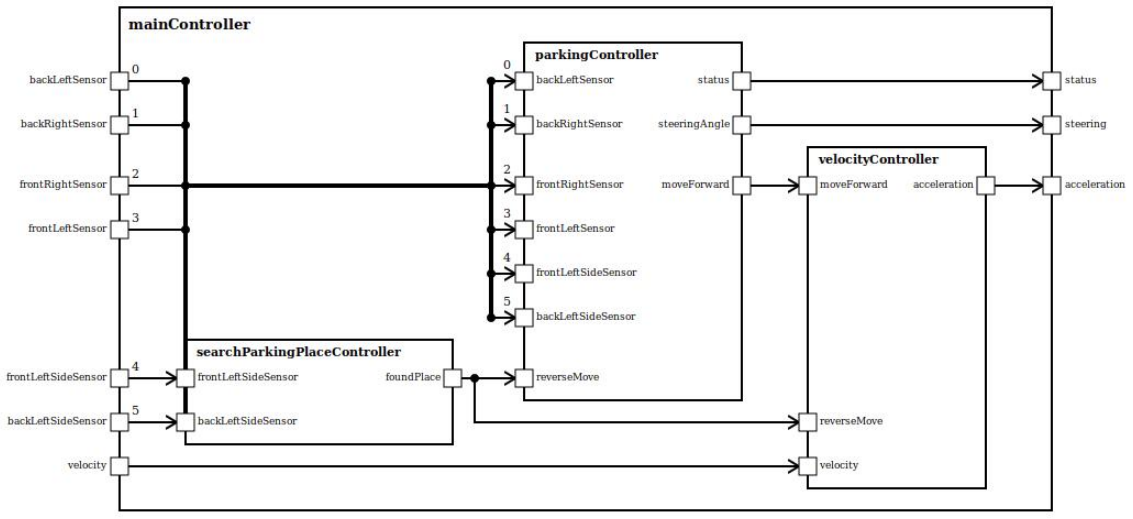


Figure 6.4: Parking controller scheme.

6.4 Maneuverability Test Tutorial

Task: Implement a model that manages driving between cones, which emulates the maneuverability test. The model should have several modules which are responsible for different actions. One of the examples could be:

1. Module controls the velocity of the car.
2. Module is responsible for a steering angle of the car, depends on current position.

Each module should be as simple as possible. To solve this tutorial, you may use from 4 to 6 sensors, which measure the distance to objects, which are located in front/side of the car. The velocity has to be from 1 m/s to 2 m/s. The figure 6.5 shows the main idea how the driving has to be done:

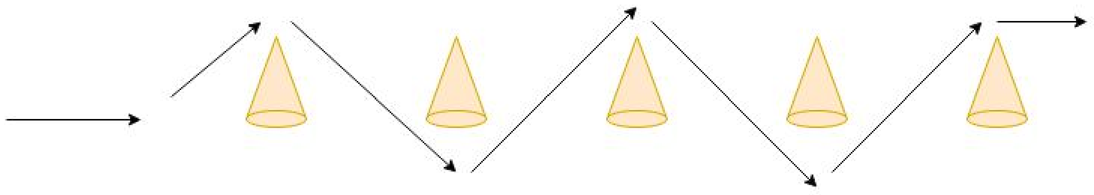


Figure 6.5: Maneuverability test.

Parking process can be divided in several steps:

1. Go straight, until a cone is closer then 10m then start to rotate the car to round the cone.
2. When a front part of the car is passed the cone, start rotation in opposite direction.
3. When the car is rotated enough to pass the next cone, stop rotation and continue the straight movement.
4. Repeat the second step until the car passed all cones.

Hint: In this task will be helpful to use static variables like a counters, to control a rotation angle of the car.

Solution

To solve the given task we should use the `MainController`, which provides an interface for a communication with the car. Nevertheless, let us start from the `VelocityController`, which controls the velocity of the car. To drive trough the road with cones, we do not have to change the velocity of the car, just use some constant value. Listing 6.11 shows an example of the `VelocityController`.

```
 1 package controller04;
 2
 3 component VelocityController {
 4     ports
 5         in  (0km/h: 250km/h) velocity,
 6         out (-2m/s^2: 2m/s^2) acceleration;
 7
 8     implementation Math {
 9
10         if (velocity > 1 m/s)
11             acceleration = 0m/s^2;
12         else
13             acceleration = 1m/s^2;
14         end
15     }
16 }
```

Listing 6.11: VelocityController.

The controller has one incoming and one outgoing port. The incoming one, `velocity` port, serves to control the velocity of the car by changing the value of the `acceleration` port. Lines 10-14, set the limit of the car's velocity. If the car has not reached the desired velocity (1 m/s), then the car accelerates. When the velocity is reached, the `acceleration` is equal zero. This simple controller provides constant speed of the car. Then we should implement the most important controller for this task, that controls a steering of the car to maneuver between cones. Listing 6.12 shows a possible solution for this task.

```
1     package controller04;
2
3     component ConeRunner {
4         port
5             in  (-200m:200m) frontLeftSensor,
6             in  (-200m:200m) frontRightSensor,
7             in  (-200m:200m) frontLeftSideSensor,
8             in  (-200m:200m) frontRightSideSensor,
9             in  (-200m:200m) backLeftSideSensor,
10            in  (-200m:200m) backRightSideSensor,
11            in  (-200m:200m) xPosition,
12            out (-180°:180°) steering,
13            out B status;
14
15        implementation Math {
16
17            static B passRight = false;
18            static B passLeft = false;
19            static (0°:180°) count = 0°;
20            status = false;
21
22            if ((frontRightSensor < 10m) && !passLeft && !passRight)
23                steering = -1°;
24            else
25                steering = 0°;
26            end
27            if (frontRightSideSensor < 1m)
28                passRight = true;
29                passLeft = false;
30                count = 0°;
31            end
32            if passRight && (count < 55°)
33                steering = 1°;
34                count +=1°;
35            end
36            if passRight && (frontLeftSideSensor < 1m)
37                passLeft = true;
38                passRight = false;
39                count = 0°;
40            end
41            if passLeft && (count < 55°)
42                steering = -1°;
43                count +=1°;
44            end
45            if (xPosition > 30m)
46                status = true;
47            end
48        }
49    }
```

Listing 6.12: ConeRunner.

The controller has seven incoming ports and two outgoing ones. The first six incoming ports are the measuring sensors. The last one, `xPosition`, the position of the car in the 3D environment. The outgoing ports control the steering of the car and a simulation status. Let us consider the `Math` scope. Lines 22-26 are in charge of passing the first cone, because it is located in the center, and the car should behave a bit differently. The car is turning to the left(cf. l. 23), when it is 10 meters left to the first cone. When it reaches the cone, it changes the rotation to opposite direction. Then, lines 27-44 describes the main actions, which helps to the car drive around cones. When the car has passed the cone from the right side, it start a rotation to the right, using the counter (cf. l. 34), which is counting the rotation angle to be able to pass a next cone from the left side. When the car reaches the next cone to pass it from the left side, the counter is set to zero, to start calculation to the opposite direction (cf. l. 39). Then the car is passing cones one by one, passing cones from different sides. After passing the field of cones, the simulation process has being stopped, by using the current position of the car. It happens in the last condition check (cf. ll. 45-47). There may be other ways of solving this tutorial, without using counters. The approximate trajectory of the car is shown in the figure 6.5. When the controllers, which deliver the main functionality have been developed, we should connect them to the car's interface, which is given in the `MainController`. Listing 6.13 shows an example of it.

```
1      package controller04;
2
3      import VelocityController;
4      import ConeRunner;
5
6      component MainController{
7          ports
8              in (-200m:200m) frontLeftSensor,
9              in (-200m:200m) frontRightSensor,
10             in (-200m:200m) frontLeftSideSensor,
11             in (-200m:200m) frontRightSideSensor,
12             in (-200m:200m) backLeftSideSensor,
13             in (-200m:200m) backRightSideSensor,
14             in (-200m:200m) backLeftSensor,
15             in (-200m:200m) backRightSensor,
16
17             in (0s:oo s) time,
18             in (0km/h:250km/h) velocity,
19             in (-200m:200m) xPosition,
20             in (-200m:200m) yPosition,
21
22             out (-2m/s^2:2m/s^2) acceleration,
23             out (-180°:180°) steering,
24             out B status;
25
26         instance VelocityController velocityController;
27         connect velocity->velocityController.velocity;
28         connect velocityController.acceleration->acceleration;
29
30         instance ConeRunner coneRunner;
31         connect this.* -> coneRunner.*;
32         connect coneRunner.*->this.*;
33     }
```

Listing 6.13: Main Controller.

`MainController` imports the implemented controllers: `VelocityController` and `ConeRunner` (cf. ll. 3-4). After that follows the `Math` scope, which begins from the interface (cf. ll. 8-24). Later the imported controllers have been instantiated and connected respectively. Usage of the dot-star(.*) notation (cf. ll. 31-32) is one of the convenient and important feature of the EmbeddedMontiArc, we can connect all ports with equal names between components. It is simplifies the text model of the controller. If we need to connect only one port, it is better to specify it directly, to facilitate connection's understanding (cf. ll. 27-28). The entire connection scheme is depicted in the figure 6.6.

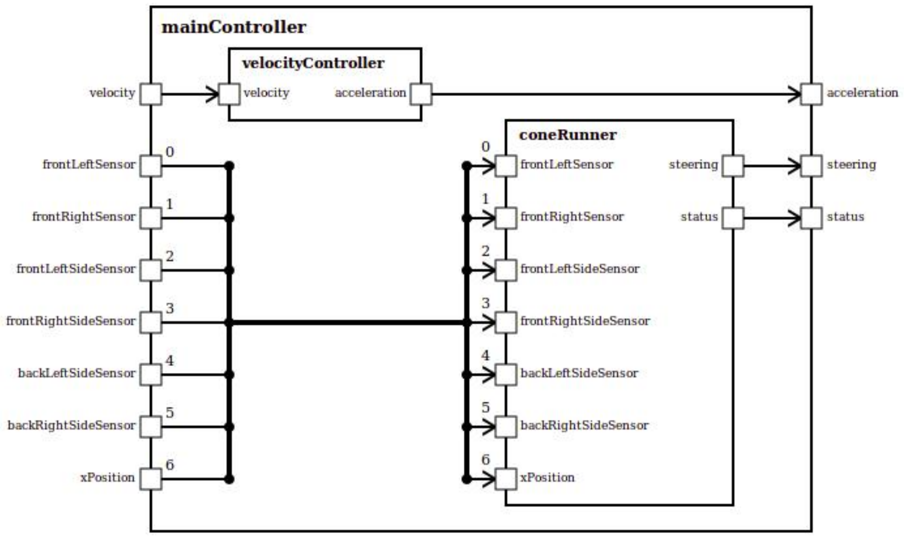


Figure 6.6: Maneuverability controller scheme.

From the left hand side of the connection scheme and the right one (see figure 6.6), the ports of the interface are depicted. Inside the `MainController` are shown the controllers, which were instantiated in the listing above(see listing 6.13).

6.5 Complete Track Tutorial

Task: Run through the entire track.

In this tutorial we are going to use previously created controllers to run trough the track with different obstacles and sub-tasks. Figure 6.7 depicts the entire track, which already have some loaded objects on it. We can divide the track into several zones, where different actions are going to happen:

1. First one is the curved area which we have to drive trough, using our previously created controller for running the whole track with optimal trajectory.

2. Then, on the straight part of the track, a car waiting to start movement, gives the possibility firstly to use the follow controller and then the overtaking one.

3. After the overtaking, at the end of the strait part of the track, the obstacle form the cones is waiting for the car. The car should brake before the obstacle and then continue movement after the obstacle will be gone.

4. At the curve part of the track we have to use again, the controller which we used in the first part, just drive through the track.

5. Next task is to drive across the field of cones.
6. The last one is the parallel parking. We should use the controller which measure the gap between the cars.
7. When the car was successful parked - stop the simulation.

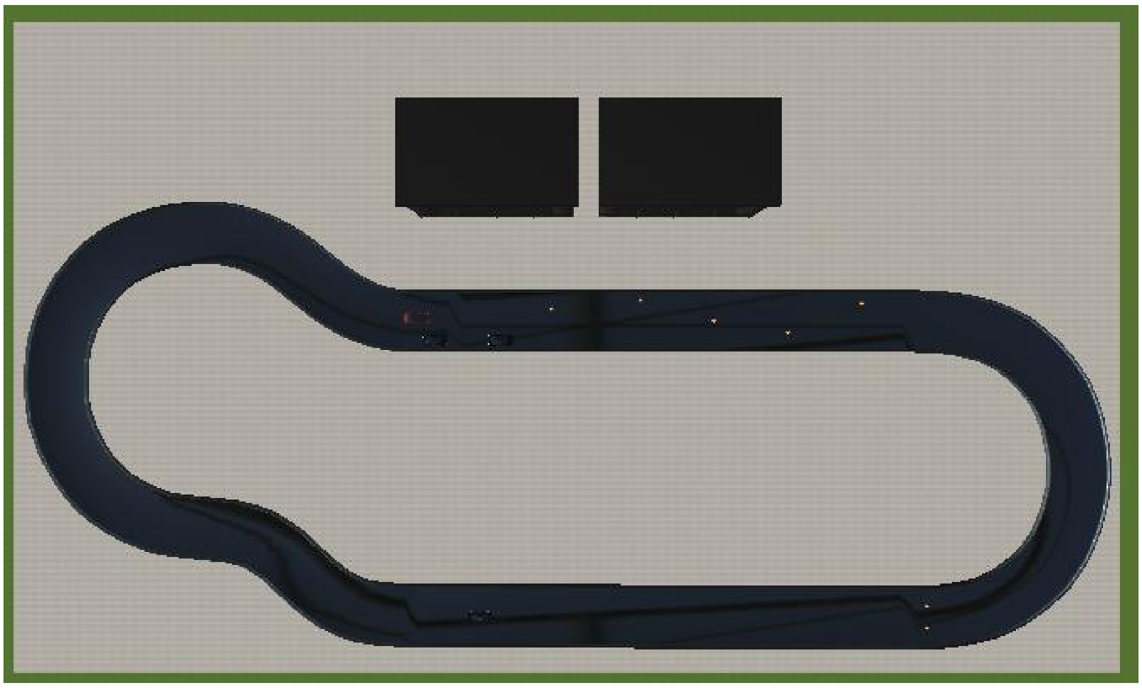


Figure 6.7: Complete track tutorial.

All these controllers were created before in the first ten tutorials. This tutorial combines them all in one controller which can manage all these actions. Previously, we have done some tutorials which used several controllers simultaneously. The main idea, that we have an actions controller, which manages some currently active action and changes the active controller, depends on conditions. This tutorial has to be done in the similar way, but a number of the controllers is much greater.

Solution:

The solution won't be given here, due to large size of the text model. The controller combines all previously created controllers in eleven tutorials.

Chapter 7

Future Work and Conclusion

This chapter introduces some features which were not implemented, due to the amount of time given for a thesis implementation, or third part applications, that currently do not support required features. Let us consider some possible improvements:

- **Compilation in a web browser:** currently, the toolchain uses the dedicated server for compilation of models. It would be better to simplify the compilation process and carry out it directly on a client side. Clang in browser can not manage the liking of the required mathematical library, details provided in the Architecture chapter.
- **Using different tracks:** in the simulator's core, the integrated track is used, which consists of different kinds of walls, like linear or curved. It would be an improvement to separate the track from the core part, to be able to load different tracks. Even better to have external track constructor, which can build new tracks, like a configuration file for these tracks. In this configuration file, the data related to walls' positions should be stored. Then, it will be possible to construct not only the new tutorials, but new tutorials with various tracks. It increases variability of possible tasks, e.g., we can create a track with some intersections and develop an absolutely new controller. It will be able to manage the crossroad passing.
- **Several controllers in one 3D simulator:** the simulator uses only one controller, which is in charge of the car driving. In the future versions, the simulator could manage several controllers, which being executed simultaneously. It let us to create more complex tutorials with much wider variety of tasks.
- **Standalone tutorials builder:** create tutorials directly in 3D environment, then the tutorial builder will generate the configuration file, which will be used in the simulator. Currently, a configuration file for a tutorial has to be created manually. Moreover, the creation process of new tutorials takes longer, because all data related to objects should be entered manually in a text form. The 3D tutorials' builder could increase the building time significantly.
- **Better objects detection:** currently, the car is uses eight sensors, which measure the distance to objects that located only at the line which begins from the sensor's position and directed strictly like a ray with a given angle. Using this type of measuring, we can not detect types of objects. Moreover, an object can be located just between the measuring sensors and consequently it can not be detected. It would be

an significant improvement, if we could detects the shapes of object or some characteristics of objects, which give us an opportunity to determine a type of objects.

All offered changes could greatly improve the functionally of the simulator and the education toolchaing in general. A possible disadvantage might be the performance requirements for client side. Currently the car in the simulator runs smoothly even on the PC which has integrated graphical adapter.

In this thesis, the new education environment for learning C&C modeling language EmbeddedMontiArc has been developed. Moreover, it has already integrated eleven tutorials, which teach student C&C modeling step-by-step.From the very beginning, we have analyzed the existing tutorials in various fields. The purpose was to find the most important features, which have an influence on the studying process, and even discover the weaknesses and try to overcome them. Then we have figured out the most suitable architecture, which is partially using the already implemented solutions, what decreases the amount of work, and at the same time, does not influence negatively on an user experience. After that, the implementation of missing components in the toolchain was given, which describes the main algorithms and describes which were made during development process. Lately, was shown how to use the tutorials' tool to be productive during development new models. Then, the group of the tutorials was presented. They present: two tutorial for beginners with explanation of basics; two interesting real-world examples, which present the main concepts and reveal important integrated features; one large final tutorial, which summarize the experience derived from all previous tutorials. At the very end, the future possibles enhancements are proposed, which were not implemented due to some technical restrictions, which currently exist, or lack of time, that was given for the thesis implementation.

Bibliography

[Arm18] Armadillo C++ library for linear algebra & scientific computing. http://arma.sourceforge.net/, 2018.

[Ass05] Modelica Association. The modelica language specification version 2.2, 2005.

[Ata18] Explanation of usage the function atan2., Website https://en.wikipedia.org/wiki/Atan2, 2018.

[Bab18] A complete JavaScript framework for building 3D games and experiences with HTML5, WebGL, WebVR and Web Audio., Website https://www.babylonjs.com/, 2018.

[Bec03] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[Bis96] Robert H Bishop. *Modern control systems analysis and design using MATLAB and SIMULINK.* Addison-Wesley Longman Publishing Co., Inc., 1996.

[Ble18] Blender is the free and open source 3D creation suite., Website https://www.blender.org/, 2018.

[BW84] Alan Bundy and Lincoln Wallen. Context-free grammar. In *Catalogue of Artificial Intelligence Tools*, pages 22–23. Springer, 1984.

[CIB18] Compile clang to wasm and have it generate code within the browser., Website https://github.com/tbfleming/cib, 2018.

[CY03] Danny Coward and Yutaka Yoshida. Java servlet specification version 2.3. *Sun Microsystems, Nov*, 2003.

[DDE+17] Jens Dankert, Christian Dernehl, Lutz Eckstein, Stefan Kowalewski, Evgeny Kusmenko, and Bernhard Rumpe. RapidCoop - Robuste Architektur durch geeignete Paradigmen für Kooperativ Interagierende Automobile. In *Automatisiertes und Vernetztes Fahren (AAET'17)*, February 2017.

[DH04] James B Dabney and Thomas L Harman. *Mastering simulink.* Pearson, 2004.

[Dis18] The distance between two points formula description, wikipedia, Website https://en.wikipedia.org/wiki/Distance, 2018.

[Dog18] Fernando Doglio. Testing your api. In *REST API Development with Node. js*, pages 261–282. Springer, 2018.

[EMA18] EmbeddedMontiArc to WebAssembly compiler's description. `https://git.rwth-aachen.de/monticore/EmbeddedMontiArc/generators/EMAM2Wasm`, 2018.

[Ems18] Emscripten, toolchain for compiling to WebAssembly, website `https://kripken.github.io/emscripten-site/index.html`, 2018.

[Fil18] Cloud9 SDK, the filesystem API, website `https://cloud9-sdk.readme.io/docs/the-filesystem-api`, 2018.

[Fri11] Peter Fritzson. Modelica—a cyber-physical modeling language and the openmodelica environment. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, pages 1648–1653. IEEE, 2011.

[Fro17] Christian Frohn. Simulation and modeling of vehicle-to-vehicle communication for autonomously driving vehicles, 2017.

[Gam18] The gamification of learning, wikipedia article. `https://en.wikipedia.org/wiki/Gamification_of_learning`, 2018.

[GHP18] Websites for you and your projects. Hosted directly from your GitHub repository., Website `https://pages.github.com/`, 2018.

[GKR$^+$17] Filippo Grazioli, Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. In *Proceedings of MODELS 2017. Workshop EXE*, CEUR 2019, September 2017.

[GMC$^+$13] Lakshmi M Gadhikar, Lavanya Mohan, Megha Chaudhari, Pratik Sawant, and Yogesh Bhusara. Browser based ide to code in the cloud. In *New Paradigms in Internet Computing*, pages 59–69. Springer, 2013.

[Hal18] Philipp Haller. Case study on embeddedmontiarc language for supermario, 2018 (labreport). 2018.

[Hau06] Matthew Hause. The sysml modelling language. In *Fifteenth European Systems Engineering Conference*, volume 9, pages 1–12. Citeseer, 2006.

[HC01] Jason Hunter and William Crawford. *Java Servlet Programming: Help for Server Side Java Developers.* " O'Reilly Media, Inc.", 2001.

[Hei18] Malte Heithoff. Case study on embeddedmontiarc language for pacman, 2018 (labreport). 2018.

[Ho17] Dinh-An Ho. 3d visualiszation api for self-driving cars, 2017.

[HR17] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017.* Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[Ilo18] Petyo Ilov. Software architecture of distributed multi-user simulation of autonomously driving vehicles, 2018.

[Jan18] Remo H Jansen. Learning typescript 2. x: Develop and maintain captivating web applications with ease. 2018.

[Jet18] Eclipse Jetty provides a Web server and javax.servlet container, Website `https://www.eclipse.org/jetty/`, 2018.

[JSP18] JavaScript, the description of the important concept - Promises. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`, 2018.

[JSZ18] JS library for creating, reading and editing .zip files, website `https://stuk.github.io/jszip/`, 2018.

[KLB18] Stefan Kaczmarek, Brad Lees, and Gary Bennett. Becoming a great ios developer. In *Swift 4 for Absolute Beginners*, pages 1–11. Springer, 2018.

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* No Starch Press, 2018.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. Modeling Architectures of Cyber-Physical Systems. In *European Conference on Modelling Foundations and Applications (ECMFA'17)*, LNCS 10376, pages 34–50. Springer, July 2017.

[KRRvW18] Evgeny Kusmenko, Jean-Marc Ronck, Bernhard Rumpe, and Michael von Wenckstern. EmbeddedMontiArc, Textual modeling alternative to Simulink (Tool Demonstration). 2018.

[KRSvW18a] Evgeny Kusmenko, Bernhard Rumpe, Sascha Schneiders, and Michael von Wenckstern. Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc. 2018.

[KRSvW18b] Evgeny Kusmenko, Bernhard Rumpe, Ievgen Strepkov, and Michael von Wenckstern. Teaching Playground for C&C Language EmbeddedMontiArc. 2018.

[LMT$^+$18] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*, 17(1):91–113, 2018.

[Lor17] Mike Lorang. Model-based design and simulation of autonomous vehicle controllers, 2017.

[Mat18a] Mathworks, simulink, Website `https://www.mathworks.com/`, 2018.

[Mat18b] MATLAB combines a desktop environment tuned for iterative analysis and design processes with a programming language that expresses matrix and array mathematics directly., Website `https://se.mathworks.com/products/matlab.html`, 2018.

[MOA17] David J Malan, Nikolai Onken, and Dan Armendariz. A web-based ide for teaching with any language. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pages 739–739. ACM, 2017.

[Nag18] Sandeep Nagar. Interactive octave sessions. In *Introduction to Octave*, pages 17–26. Springer, 2018.

[Oct18] Octave Online is a web UI for GNU Octave, the open-source alternative to MATLAB, Website `https://octave-online.net/`, 2018.

[OMG07] OMG OMG. Systems modeling language (omg sysml) specification. *Object Management Group, OMG Available Specification (September 2007)*, 2007.

[Onl18] Teaching Playground for C&C Language EmbeddedMontiArc with examples and description. `http://www.se-rwth.de/materials/ema_tutorial/`, 2018.

[Pav18] Svetlana Pavlitskaya. Integration of deep learning components into autonomous driving architectures, 2018.

[Ric18] Christoph Richter. Model-predictive trajectory control and simulation for self-driving vehicles, 2018.

[Ron17] Jean-Marc Ronck. Creation of a multi-user online ide for domain-specific languages, 2017.

[Rus18] Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety, Website `https://www.rust-lang.org/`, 2018.

[San10] Conrad Sanderson. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. 2010.

[SC16] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based c++ library for linear algebra. *Journal of Open Source Software*, 2016.

[Sch17a] Leon Scheidt. Qualitätsmaße für clustering-performance-evaluation, 2017.

[Sch17b] Saschsa Schneiders. Development of a c++ generator for embedded modeling languages, 2017.

[Sch18a] Manuel Schrick. Visualisation of textual component and connector models, 2018.

[Sch18b] Martin Schultz. Modelling dynamic architectures for cooperative vehicles, 2018.

[Sem17] Albi Sema. Software architectures for deep learning based autonomous driving, 2017.

[SWA18] The description of Software architecture. https://en.wikipedia.org/wiki/Software_architecture, 2018.

[Swi18] Swift Playgrounds is a revolutionary app for iPad that makes learning Swift interactive and fun., Website https://www.apple.com/swift/playgrounds/, 2018.

[TDD18] The description of Test-driven development process. https://en.wikipedia.org/wiki/Test-driven_development, 2018.

[Thr18] Lightweight cross-browser JavaScript library/API used to create and display animated 3D computer graphics on a Web browser, website https://threejs.org/, 2018.

[Typ18] TypeScript is an open-source programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript, and adds optional static typing to the language, Website https://www.typescriptlang.org/, 2018.

[VFS18] The Filesystem API which interacts with vitrual file systems, Website https://cloud9-sdk.readme.io/docs/the-filesystem-api, 2018.

[Web18a] JavaScript API for rendering interactive 2D and 3D graphics, website https://en.wikipedia.org/wiki/WebGL, 2018.

[Web18b] WebAssembly is a binary instruction format for a stack-based virtual machine., Website https://webassembly.org/, 2018.

[Wol18] Wolfram Alpha defined a fundamentally new paradigm for getting knowledge and answers, Website http://www.wolframalpha.com, 2018.

[XML18] JS Objects to interact with servers. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest, 2018.

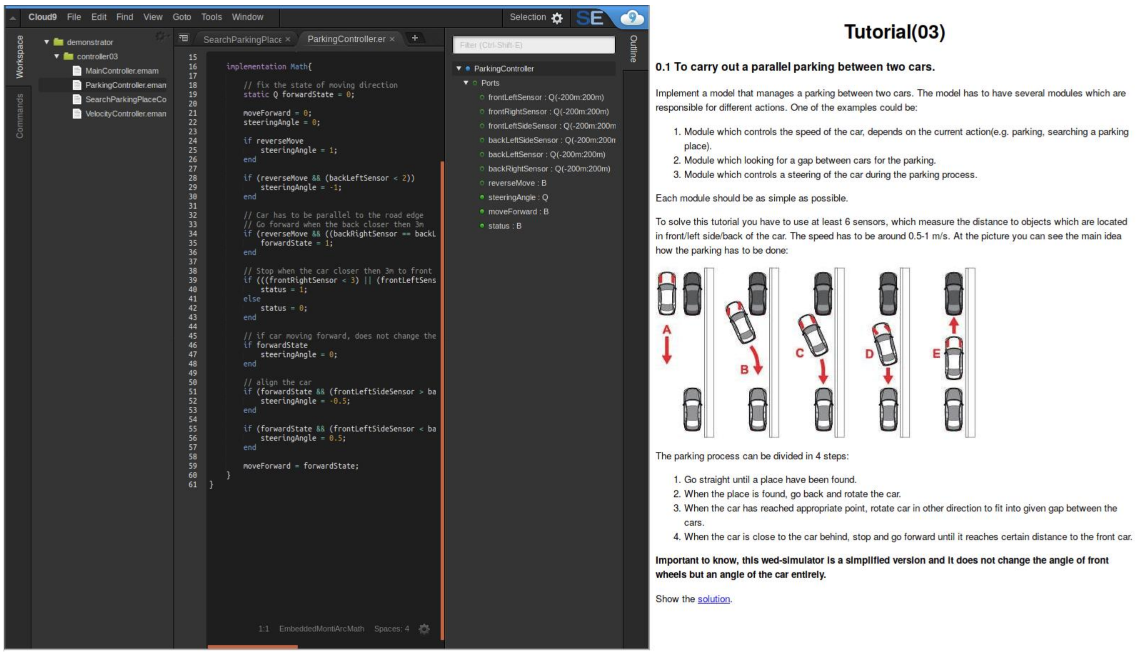
[Z3P18] Z3 is a theorem prover from Microsoft Research, Website https://github.com/Z3Prover/z3, 2018.

[Zak11] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

Appendices

```
package test.coco.valid.units;

component MyComponentMix {
    ports
        out Q(0kg: 100kg) out1;

    implementation Math {

      Q (0kg: 100kg) var1 = 10kg;
      if var1 <= 10kg
        out1 = 10kg;
      end

      if 10kg < var1
        out1 = 10kg;
      end

      Q (0kg: 100kg) var2;
      var2 = 10kg;

      if var1 > var2
        out1 = 10kg;
      end

      if 10m == 10m
        out1 = 10kg;
      end
      Q var3 = 1000m;
    }
}
```

Listing 1: Example of a valid test for Context Conditions.


Cloud9 File Edit Find View Goto Tools Window
Selection
Workspace
Commands
Outline
demonstrator
controller03
MainController.emam
ParkingController.eman
SearchParkingPlaceCo
VelocityController.eman
SearchParkingPlace
ParkingController.er
Filter (Ctrl-Shift-E)
ParkingController
Ports
frontLeftSensor : Q(-200m:200m)
frontRightSensor : Q(-200m:200m)
frontLeftSideSensor : Q(-200m:200m
frontRightSideSensor : Q(-200m:200m
backLeftSideSensor : Q(-200m:200m)
backLeftSensor : Q(-200m:200m)
backRightSensor : Q(-200m:200m)
reverseMove : B
steeringAngle : Q
moveForward : B
status : B
1:1 EmbeddedMontiArcMath Spaces: 4
A
B
C
D
E

```
    implementation Math{

        // fix the state of moving direction
        static Q forwardState = 0;

        moveForward = 0;
        steeringAngle = 0;

        if reverseMove
            steeringAngle = 1;
        end

        if (reverseMove && (backLeftSensor < 2))
            steeringAngle = -1;
        end

        // Car has to be parallel to the road edge
        // Go forward when the back closer then 3m
        if (reverseMove && ((backRightSensor == backL
            forwardState = 1;
        end

        // Stop when the car closer then 3m to front
        if (((frontRightSensor < 3) || (frontLeftSens
            status = 1;
        else
            status = 0;
        end

        // if car moving forward, does not change the
        if forwardState
            steeringAngle = 0;
        end

        // align the car
        if (forwardState && (frontLeftSideSensor > ba
            steeringAngle = -0.5;
        end

        if (forwardState && (frontLeftSideSensor < ba
            steeringAngle = 0.5;
        end

        moveForward = forwardState;
    }
}
```

Tutorial(03)

0.1 To carry out a parallel parking between two cars.

Implement a model that manages a parking between two cars. The model has to have several modules which are responsible for different actions. One of the examples could be:

1. Module which controls the speed of the car, depends on the current action(e.g. parking, searching a parking place).
2. Module which looking for a gap between cars for the parking.
3. Module which controls a steering of the car during the parking process.

Each module should be as simple as possible.

To solve this tutorial you have to use at least 6 sensors, which measure the distance to objects which are located in front/left side/back of the car. The speed has to be around 0.5-1 m/s. At the picture you can see the main idea how the parking has to be done:

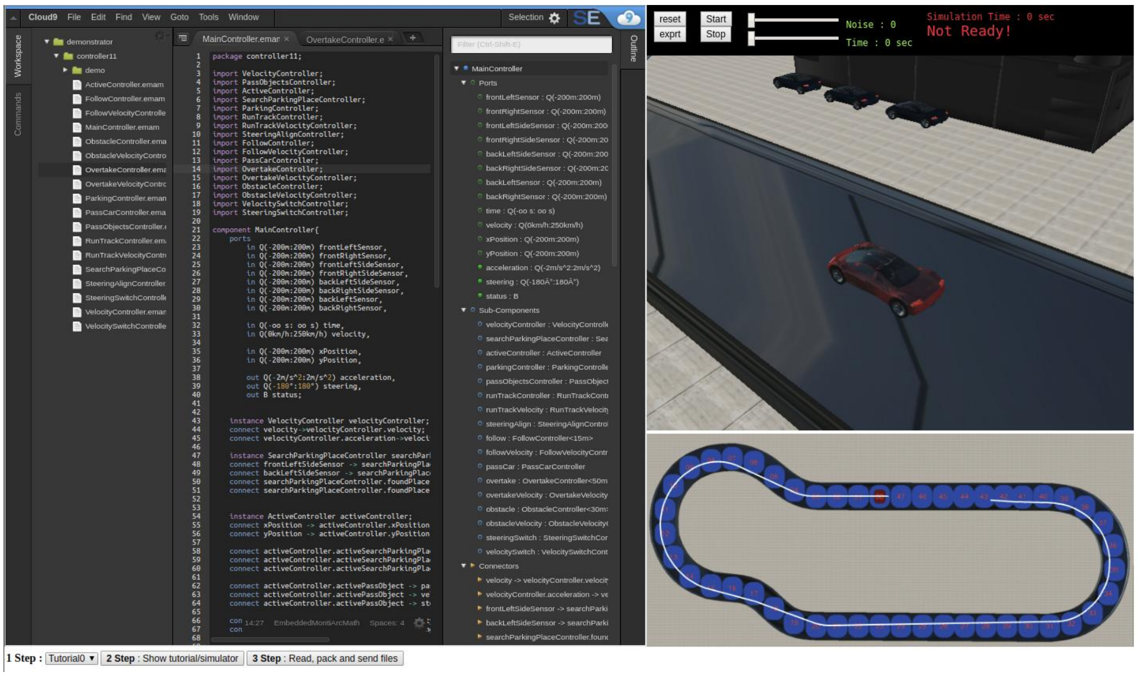
The parking process can be divided in 4 steps:

1. Go straight until a place have been found.
2. When the place is found, go back and rotate the car.
3. When the car has reached appropriate point, rotate car in other direction to fit into given gap between the cars.
4. When the car is close to the car behind, stop and go forward until it reaches certain distance to the front car.

Important to know, this wed-simulator is a simplified version and it does not change the angle of front wheels but an angle of the car entirely.

Show the solution.

Figure 1: The main page with the task tab on the right hand side.


Cloud9 File Edit Find View Goto Tools Window
Selection
Workspace
Commands
Outline
demonstrator
controller11
demo
ActiveController.emam
FollowController.emam
FollowVelocityControlle
MainController.emam
ObstacleController.ema
ObstacleVelocityContro
OvertakeController.ema
OvertakeVelocityContrc
ParkingController.eman
PassCarController.ema
PassObjectsController.
RunTrackController.em
RunTrackVelocityContr
SearchParkingPlaceCo
SteeringAlignController
SteeringSwitchControlle
VelocityController.emar
VelocitySwitchControlle
MainController.emar
OvertakeController.e
Filter (Ctrl-Shift-E)
MainController
Ports
frontLeftSensor : Q(-200m:200m)
frontRightSensor : Q(-200m:200m)
frontLeftSideSensor : Q(-200m:200
frontRightSideSensor : Q(-200m:20
backLeftSideSensor : Q(-200m:200
backRightSideSensor : Q(-200m:20
backLeftSensor : Q(-200m:200m)
backRightSensor : Q(-200m:200m)
time : Q(-oo s: oo s)
velocity : Q(0km/h:250km/h)
xPosition : Q(-200m:200m)
yPosition : Q(-200m:200m)
acceleration : Q(-2m/s^2:2m/s^2)
steering : Q(-180Â°:180Â°)
status : B
Sub-Components
velocityController : VelocityControlle
searchParkingPlaceController : Sea
activeController : ActiveController
parkingController : ParkingControlle
passObjectsController : PassObject
runTrackController : RunTrackCont
runTrackVelocity : RunTrackVelocity
steeringAlign : SteeringAlignControl
follow : FollowController<15m>
followVelocity : FollowVelocityContr
passCar : PassCarController
overtake : OvertakeController<50m
overtakeVelocity : OvertakeVelocity
obstacle : ObstacleController<30m:
obstacleVelocity : ObstacleVelocity
steeringSwitch : SteeringSwitchCor
velocitySwitch : VelocitySwitchCont
Connectors
velocity -> velocityController.velocit
velocityController.acceleration -> ve
frontLeftSideSensor -> searchParki
backLeftSideSensor -> searchParki
searchParkingPlaceController.founc
14:27 EmbeddedMontiArcMath Spaces: 4
reset
exprt
Start
Stop
Noise : 0
Time : 0 sec
Simulation Time : 0 sec
Not Ready!
1 Step : Tutorial0
2 Step : Show tutorial/simulator
3 Step : Read, pack and send files

```
package controller11;

import VelocityController;
import PassObjectsController;
import ActiveController;
import SearchParkingPlaceController;
import ParkingController;
import RunTrackController;
import RunTrackVelocityController;
import SteeringAlignController;
import FollowController;
import FollowVelocityController;
import PassCarController;
import OvertakeController;
import OvertakeVelocityController;
import ObstacleController;
import ObstacleVelocityController;
import VelocitySwitchController;
import SteeringSwitchController;

component MainController{
    ports
        in Q(-200m:200m) frontLeftSensor,
        in Q(-200m:200m) frontRightSensor,
        in Q(-200m:200m) frontLeftSideSensor,
        in Q(-200m:200m) frontRightSideSensor,
        in Q(-200m:200m) backLeftSideSensor,
        in Q(-200m:200m) backRightSideSensor,
        in Q(-200m:200m) backLeftSensor,
        in Q(-200m:200m) backRightSensor,

        in Q(-oo s: oo s) time,
        in Q(0km/h:250km/h) velocity,

        in Q(-200m:200m) xPosition,
        in Q(-200m:200m) yPosition,

        out Q(-2m/s^2:2m/s^2) acceleration,
        out Q(-180°:180°) steering,
        out B status;


    instance VelocityController velocityController;
    connect velocity->velocityController.velocity;
    connect velocityController.acceleration->veloci

    instance SearchParkingPlaceController searchParl
    connect frontLeftSideSensor -> searchParkingPla
    connect backLeftSideSensor -> searchParkingPlac
    connect searchParkingPlaceController.foundPlace
    connect searchParkingPlaceController.foundPlace


    instance ActiveController activeController;
    connect xPosition -> activeController.xPosition
    connect yPosition -> activeController.yPosition

    connect activeController.activeSearchParkingPla
    connect activeController.activeSearchParkingPla
    connect activeController.activeSearchParkingPla

    connect activeController.activePassObject -> pa
    connect activeController.activePassObject -> ve
    connect activeController.activePassObject -> st

    con
    con
```

Figure 2: The main page with the simulator and trajectory builder tab.