RWTH Aachen University
Software Engineering Group

MontiCAR
3D Modeling Using EmbeddedMontiArcMath

Seminar Paper

presented by
Mokhtarian, Armin

1st Examiner: Prof. Dr. B. Rumpe

Advisor: Michael von Wenckstern

The present work was submitted to the Chair of Software Engineering

Aachen, 26. Januar 2018

The present translation is for your convenience only.
Only the German version is legally binding.

Statutory Declaration in Lieu of an Oath

Last Name, First Name _____ Matriculation No. (optional) _____

I hereby declare in lieu of an oath that I have completed the present Bachelor's thesis/Master's thesis* entitled

_____

_____

_____

independently and without illegitimate assistance from third parties. I have use no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Location/City, Date _____ The German version has to be signed. _____ Signature

*Please delete as appropriate

Official Notification:

Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whosoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.

(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

I have read and understood the above official notification:

City, Date _____ The German version has to be signed. _____ Signature

Kurzfassung

Ein essentielles Problem der heutigen Zeit in Hinblick auf die automobile Industrie betrifft das Thema Automatisierung. Um autonomes Fahren zu ermöglichen müssen Cyber-physische Systeme (CPS) entwickelt werden, welche die Modellierung und vor allem das Testen solcher Szenarien erlauben. Um beispielsweise Routenplanung, Spurkorrektur oder eine Geschwindigkeitsregulierung zu modellieren, ist eine gute Schnittstelle zur Umgebung (Sensoren und Aktuatoren) unverzichtbar. Zudem ist eine Herausforderung an die Entwicklungsumgebung, dass sie möglichst übersichtlich ist und in einzelne Bereiche aufgeteilt werden kann. Dies schafft eine Grundlage, sodass einzelne Entwickler mit verschiedenen Expertisen an alleinstehende Komponenten arbeiten, die zum Gesamtsystem beitragen. Daher sind Component und Connector Views eine weit verbreitete Methode um die Architektur von domänenspezifischen Sprache zu beschreiben. Während sie das Zusammenspiel der Komponenten gut beschreiben, haben die meisten Systeme Schwierigkeiten die Implementierung zu ermöglichen. In diesem Paper wird die MontiCAR Sprachfamilie vorgestellt, welche dieses unter anderem berücksichtigt und die Simulierung von autonomen Fahrszenarios realisiert. Anhand eines exemplarischen, aber dennoch realistischen Szenario wird das System erklärt und im Anschluss evaluiert.

Abstract

An important problem these days regarding the automotive industries pertains the automation issue. To enable automotive driving one has to develop Cyber-Physical systems (CPS), which allow to model and especially test such scenarios. To model e.g. trajectory planning, speed regulation or lane correction a good interface to the environment (sensors and actuators) is needed. Furthermore, one challenge of the development environment is that it is preferably clear and can be decomposed into smaller areas. This would allow, that individual developers with different expertise are able to work on separate components, which then form the whole system. Thus, Component and Connector Views (C&C) are a common method to describe the architecture of Domain-Specific Languages (DSL). While being able to give good overviews of components and relations, most systems struggle to support the implementation. This paper presents the MontiCAR Language family, which among others takes this problem into consideration and enables the simulation of autonomous driving scenarios. With the aid of an exemplary but realistic driving scenario we explain the system and evaluate it afterwards.

Table of Contents

Chapter 1

Introduction

The motivation to develop self-driving cars rises not only because user desire its comfort, but also from the fact that autonomous driving is supposed to ensure safety. In Germany alone, traffic accidents caused the loss of over 3 206 people in 2016 [Bun17]. That is why autonomous vehicles have complex software and hardware systems which have to deal with a wide spectrum of safety-relevant functions. The common way to test the ability and functions of an autonomous vehicle is to simulate the environment, because the infrastructure and tests with real vehicles is to cost-intensive [FG17]. In order to do so, Domain-Specific Languages (DSLs) are often used to describe the architecture of such a Cyber-Physical System (CPS). Considering realistic simulations, a major challenge is to model complex environments and physical laws. Therefore C&C modeling provides an appropriate abstraction layer to describe the architecture. Such an approach allows multiple developers to work on the same system, but on different components. This lets the developer focus on the functionality and also reduces the error-proneness in the design phase.

Although, enabling the development for the high-level layer is not sufficient. Many CPS tasks require to be solved by mathematical models. Thus, the low-level implementation needs to be powerful as well. Taking this problem into consideration we present MontiCAR. It has an advanced math language integrated. To increase the performance of the generated code it provides for example a matrix type system with its related operations.

Rumpe et al. [EK17] investigated Cyber-Physical-Systems in order to determine the requirements which modeling architectures have to face. The previously mentioned matrix support is one of them. Afterwards the most important C&C modeling languages are reviewed regarding twelve investigated requirements. They found out, that there is no modeling language which fulfills all requirements. Almost every language provides *static analysis and configuration parameters*. Rather rare is *unit support and unit conversion* which are provided by MontiCAR. This allows to work with physical quantities in a type-safe manner. Besides that MontiCAR types have a value range and a resolution.

To avoid redundancy in models, languages can support *port arrays* and *component arrays* which are a mechanism to interconnect and access ports and components. This is only supported by MontiCAR and six others of the 20 languages. In fact, the only thing which is not supported by MontiCAR are *differential equitation* and *acausal modeling support*. The latter is needed to model systems where the behavior of each system's component depends on the global system architecture.

In order to investigate MontiCAR this paper is structured as follows: In chapter 2 we will present MontiCAR which allows to implement C&C autonomous driving scenarios without further extensions. We provide a running example developed with MontiCAR in chapter 3, which is used to explain and illustrate the use and functionalities of MontiCAR. Afterwards we test our result by visualizing the project with a simulator. The chapter 5 concludes and evaluates the presented system.

Chapter 2

MontiCAR

MontiCore [HK10] is a framework which can be used for agile and model-driven software development. It enables an efficient development of domain-specific languages. Furthermore, MontiCore and its DSL products are used in various domains like academic and industrial research or automotive software modeling. MontiCore provides code generators, transformations and analyses in order to create DSLs with their belonging infrastructure. It enables a highly extensible DSL development using different features like its own MontiArc Architecture Description Language . Thus, we use MontiCore to develop MontiCAR DSLs which are described by this paper.

2.1 MontiCAR Language Family

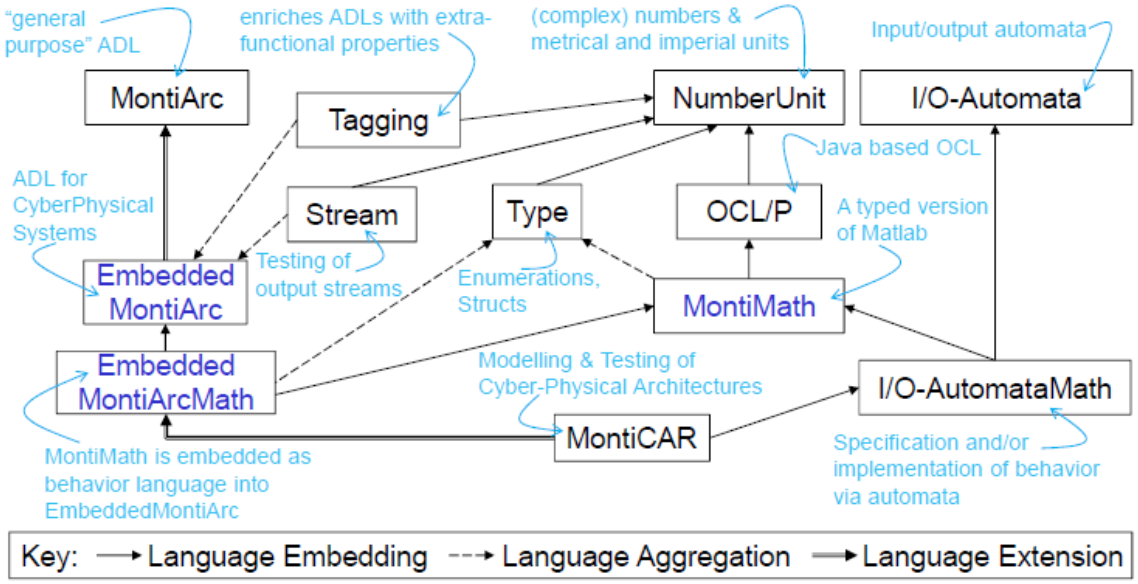

Figure 2.1: Overview project of all language definitions used in the EmbeddedMontiArc Modeling language family [www18].

MontiCAR is used as an architecture description language for Cyber-Physical Systems. In the following its two main components are described, whereby *EmbeddedMontiArc* (EMA) is considered the core language. Figure 2.1 provides an overview of all language definitions used in the EMA Modeling language family. Among others, the Figure illustrates that *EmbeddedMontiArcMath* (EMAM) extends EMA and embeds *MontiMath*. Subsequently we will discuss these languages and their connection. Besides that, we will describe *Stream* which enables output testing in order to make sure that EMA models operate as requested. This paper shows how MontiCAR allows simulating and particularly testing autonomous driving scenarios. By using Component and Connector views it enables a tightly structured system and its language is designed to be easy to learn and read.

The creation of MontiCAR models is done, as in other programming languages, within an IDE. But in this case, it does not use a local IDE but an online IDE. The written MontiCAR code is passed to a generator [FG17] which creates C++ code. This is compiled and used by a simulator [FG17] to visualize the modeled scenario. Although this paper only deals with the simulation for autonomous driving, this is just an example for the use of MontiCAR.

2.1.1 EmbededMontiArc

```
1  component Adder{
2  ports
3    in Q sum1,
4    in Q sum2,
5
6    out Q result,
7    out Q absoluteResult;
8
9    instance Absolute abs;
10   result -> abs.in1;
11   abs.out1 -> absoluteResult;
12
13 implementation Math {
14   result = sum1 + sum2;
15 }
16 }
```

Listing 2.1: Exemplary EMAM component.

EmbeddedMontiArc is a Component and Connector modeling language based on MontiArc. C&C views allow to abstract from the traditional implementation-oriented decomposition of systems and enable a vivid overview of a system hierarchy, connectivity, port names and types. As an extension of MontiArc, EMA has concepts of port and component arrays and includes a type system which provides unit support for physical quantities. Every component contains of an arbitrary number of input- and output ports. Listing 2.1 shows one sample component which contains of two input- and output ports. The behavior of these are implemented in the *implementation Math* function. Besides that, a component can have instances of other components. The implementation math function is only recommended for atomic components. Listing 2.1 shows both in order to illustrate the syntax.

In order to use these instances, one has to connect its ports. The syntax is illustrated in Listing 2.1. It connects the result of the summation to a component's input port which

calculates the absolute value. In the following, this kind of architecture is used to simulate a self-driving car.

2.1.2 MontiMath

MontiMath is a standalone language which is developed to provide support for mathematical expressions. Although it was created for the MontiCAR Language Family it can also be used in other projects which utilize MontiCore. Listing 2.2 shows an extract of the examples given in the MontiMath documentation [EMA].

```
1 Q^{2,2} mat = [1,0; 0,1];     // identity matrix
2 Q value = mat(1,1);           //get first element from matrix
3
4 Q(0:5m) smallDistance = 4;    //Units
```

Listing 2.2: Example operations using MontiMath.

The first two lines shows how a matrix can be declared and accessed. In line four a unit is created. The MontiCAR framework allows one to use metrical and imperial units to specify e.g. what physical value a component's port has (meters, kilograms, volts etc.) and with which precision it is measured (0.01 m/s, 0.25 volts etc.). This functionality is provided by the NumberUnit DSL (Figure 2.1) . More information on units can be found in [EK17].

2.1.3 EmbeddedMontiArcMath

EmbeddedMontiArcMath combines EmbededMontiArc with MontiMath. Within the *implementation Math* function in line 13 of Listing 2.1 the MontiMath environment can be used. Previously defined ports of the component are automatically available in the function and further variables can be added by using the *static* keyword as visible in Listing 2.3. The specific feature of this keyword is, that its value remains available after an execution cycle.

```
1 static Q(0:3) delayValue = 0;
```

Listing 2.3: Creating a variable which retains its value through the execution cycles.

This Listing shows the variable *delayValue* which can store values between 0 and 3. Since the *implementation Math* function is called repeatedly during runtime, the *static* keyword is needed to save a value through all iterations.

2.1.4 Stream

MontiCAR provides the ability to write unit tests for components in the form of Stream models, from which C++ unit tests are generated [Ryd18]. A Stream model can get an arbitrary number of values for any input port. It must be considered that the amount of values per input port represents the number of execution cycles where the component is tested. Thus, it is necessary that every input port gets the same amount of values. To

every combination of input values, there is an expected output value. Listing 2.4 shows an exemplary Stream test for the Adder from Listing 2.1. Within the first execution cycle, it sets the values 1 for sum1 and -2 for sum2. Our expected result should be -1 and its absolute value is 1. If the Adder is implemented.

```
1 stream tester for Adder {
2   sum1: 1 tick 2 tick 3;
3   sum2: -2 tick 0 tick 10;
4
5   result: -1.0 tick 2.0 tick 13.0 +/- 0.01;
6   absoluteResult: 1.0 tick 2.0 tick 13;
7 }
```

Listing 2.4: Testing the Adder Component with the Stream language.

correctly, this test should be passed. Furthermore Listing 2.4 shows another feature of the Stream language. In the third execution cycle the expected value is set to '13.0 +/- 0.01'. This allows to define a tolerance range which might be useful in many cases.

Chapter 3

Modeling

In order to teach the correct use of MontiCAR and provide a realistic use case, we explain several components which together shape an autonomous driving project. The projects goal is not only to have a car driving with predefined speed from arbitrary point A to B, but also to navigate automatically from its current location on to the route. The car drives to the nearest point on the route and does not get forced to drive to the starting point A. Thus, the route is trimmed afterwards. Besides that, our component detects straight road sections and curves and adapts its speed to the current situation.

In the following we describe the values given by the simulator and how they are used to calculate the three main properties which are *Engine, Brakes and Steering*. Figure 3.1 illustrates the used components and their properties. After explaining the basic structure, we will discuss the implementation of the main components and evaluate the modeling language as well as the result.

3.1 Architecture

As visible in Figure 3.1 our primary component is *Autopilot.emam*. It is inspired by an older model by [Ryd18] which had not the ability to use implementation math. Thus, the architecture of the older model is changed and also extended. One main feature of the newer architecture is that it gets the shortest path from the simulator. Our component has 11 input ports which are used by its three component instances in order to calculate Engine, Brakes and Steering. While Engine and Brakes are self-explaining we describe Steering and some oft the most important input ports of the Autopilot.emam. This list does not contain all because some are not used for our project.

- **steering:** Positive values represent a right turn and thus negative values a left.
- **timeIncrement:** Although we do not need this port for our implementation it is still very valuable. It allows to calculate the time depending on the simulation frequency, which might be useful in many scenarios.
- **compass:** Representing the orientation angle (also sometimes called yaw angle) in radians. It is measured from the positive Y axis direction to the actual vehicle's direction counter-clockwise.

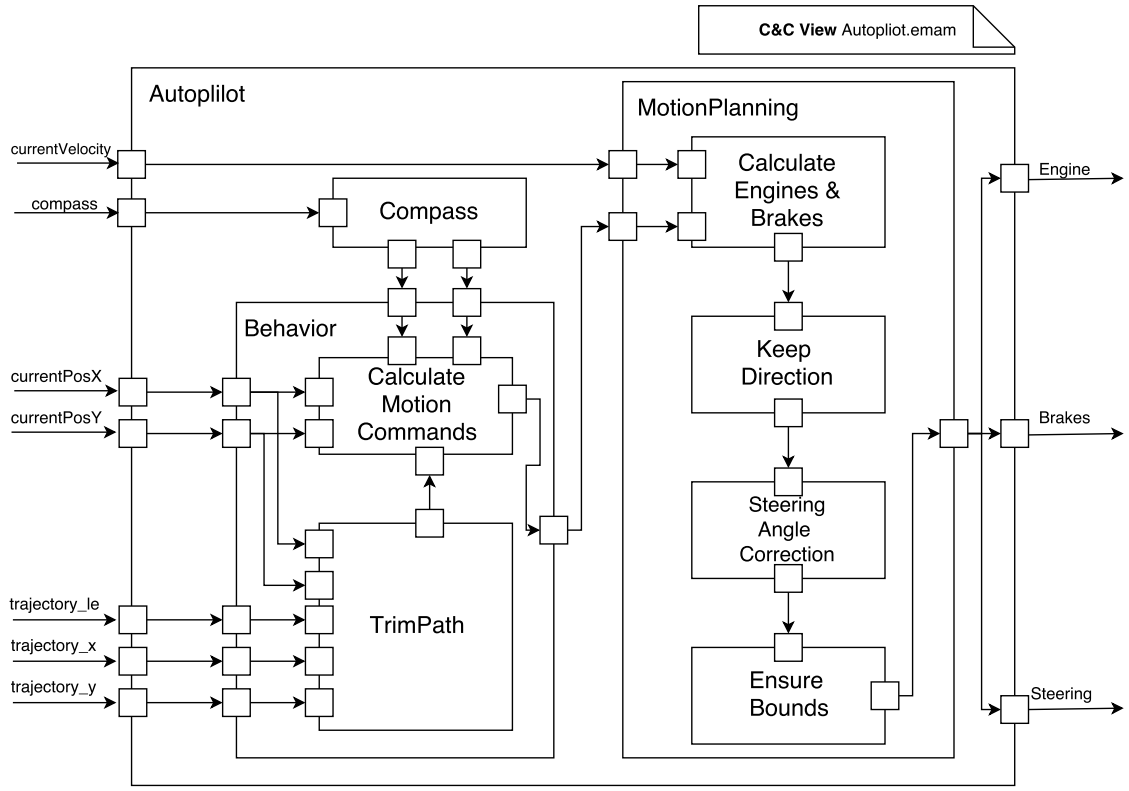

Figure 3.1: C&C view of the autopilot project.

- **currentVelocity:** After every execution cycle, the currentVelocity is compared to the desired velocity. Depending on the difference the Engine is calculated to reach the desired value. The calculation is shown in Section 3.2.2.
- **position:** Consisting of two ports (x and y) one receives the vehicles position with a precision of 1 cm.
- **trajectory_length:** This value determines the number of waypoints on the trajectory.
- **trajectory_position:** For each (x and y) the simulator provides an array which contains the coordinates of each nodes on the route.

3.2 Components

Except for *Compass* our Autopilot.emam consist of two components called *Behavior* and *MotionPlanning* which have again subcomponents as visible in Figure 3.1. In order to calculate the motion, one need to know **what** the car is supposed to do. That is the purpose of the Behavior-Subcomponent. Its output is, together with the *currentVelocity*, forwarded to *MotionPlanning*. Given the 'what', now the **how** is computed. Once the output is generated, it has to be checked whether it fits especially physical limitations. The implementation of these subcomponents is described in the following.

3.2.1 BehaviorGeneration

As the name indicates, the *TrimPath* component is supposed to trim the planned path such that the first segment is the closest to the vehicle's current position and the first point is the projection of the current position to the planned path. Hereby we distinguish between three cases:

1. **Vehicle is off the trajectory:**
   If the distance to the trajectory is bigger than a predefined threshold, the car should navigate to the first point in order to get closer to the trajectory.
2. **One last point:**
   If there is only one remaining node in the trajectory, the car should navigate directly to this point.
3. **There are at least 2 points in the trajectory:**
   Now *signedDistanceToTrajectory* is computed. This value describes on which side of the trajectory the car is located and also stores the distance to the trajectory. This is mandatory in order to determine in which direction the trajectory has to be entered.

To compute the third case, we are given two points p_1 and p_2. Furthermore the vector vec between these points and the current position of the vehicle ($posX$ and $posY$) are given. The following equation shows how the result is computed.

$$Result = -\frac{(vec.y \cdot posX) - (vec.x \cdot posY) + (p_2.x \cdot p_1.y) - (p_1.x \cdot p_2.y)}{vec.norm} \tag{3.1}$$

The orientation of a triangle (e.g. p_1, cur.pos2 and p_2)which is illustrated in Figure 3.2 is the sign of the determinant of the 3×3 matrix containing the norm vector and the Cartesian coordinates of the three vertices of the triangle.

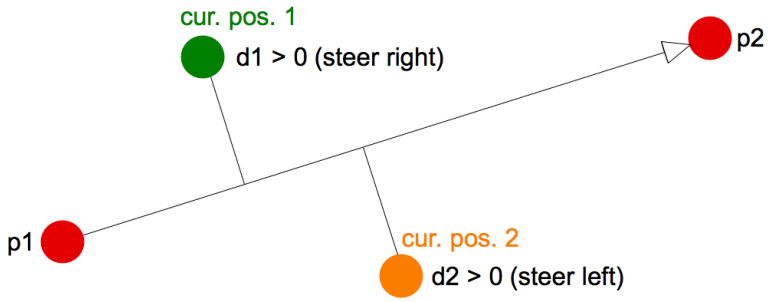

Figure 3.2: Two possible scenarios (green and orange) which are distinguished.

Now that we know where the vehicle is located, we are able to compute the next point on our route. Figure Figure 3.3 illustrates two cases which have to be distinguished.

1. There are only two points on the trajectory. In that case we take the current position, p_1 and p_2 and calculate with Bezier interpolation the next point.
2. If there are more than two points on the trajectory, we skip the first one and calculate the next point by using the current position, p_2 and p_3.

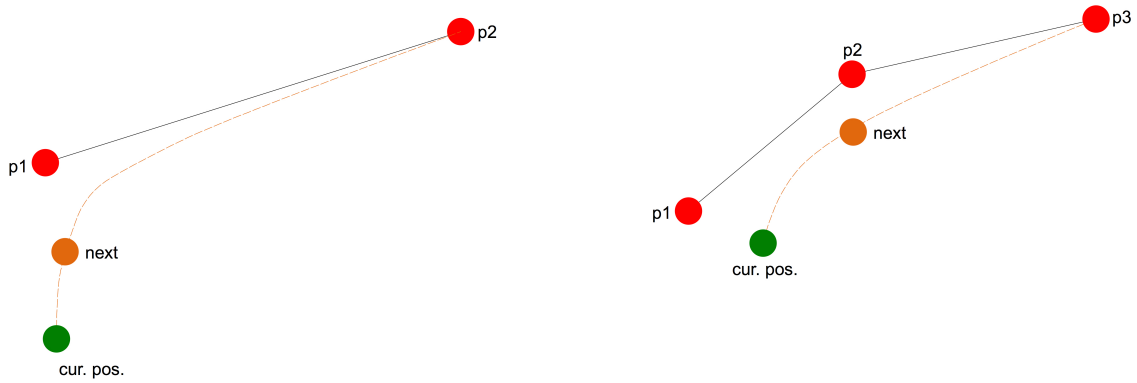

Figure 3.3: Bezier interpolation over *current pos*, p_1 *and* p_2.

After the path is trimmed, we have to determined the previously mentioned 'what' by calculating the motion commands. One of them is the desired speed. For this, we traverse along the trajectory until an angle is found which is bigger than a predefined threshold. In that case the speed is scaled down with respect to the angle while the measured distance to finish or to the curve determines the scaling factor.
Furthermore we have to calculate the *DesiredDirection*. This is done by subtracting the current positions from the next position as shown in Listing 3.1.

```
1 desiredDirectionX = next.x - currentPosition.x
2 desiredDirectionY = next.y - currentPosition.y
```

Listing 3.1: Determine desiredDirection using the interpolated point *next*.

3.2.2 MotionPlanning

Once the 'what' is determined, we now want to compute how the desired values as in *desiredSpeed* and *desiredSteeringAngle* can be achieved. That is why one of MotionPlanning's main components is a Proportional–Integral–Derivative controller (a control loop feedback mechanism, short PID).

PID

The project goals require the car to drive at a specific speed. In order to control the car, one is only able to set the Engine-, Brakes- and Steering- values. Engine is set greater 0 to

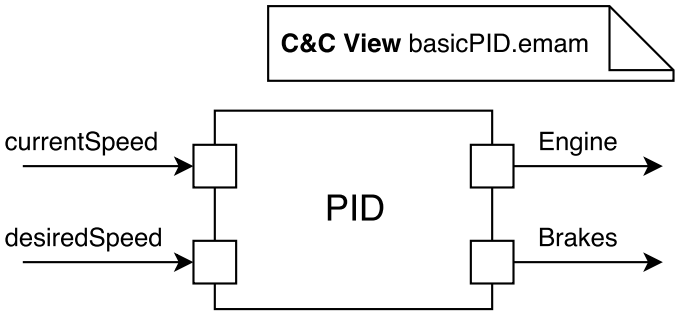

Figure 3.4: Simple Controller which decide to accelerate or brake.

accelerate, so that at some point the car reaches the desired speed. This is where a PID controller is needed to stop accelerating or even brake and accelerate again when fallen below the desired speed.

The PID Controller has two input- and two output ports as illustrated in Figure 3.4. If the *desiredSpeed* is greater than *actualSpeed* the PID component sets the Engine value and deactivates the brakes and vice versa if desiredSpeed crosses the actualSpeed. At this point, the car either accelerates or brakes with full strength, which is not a smooth way to drive a car.

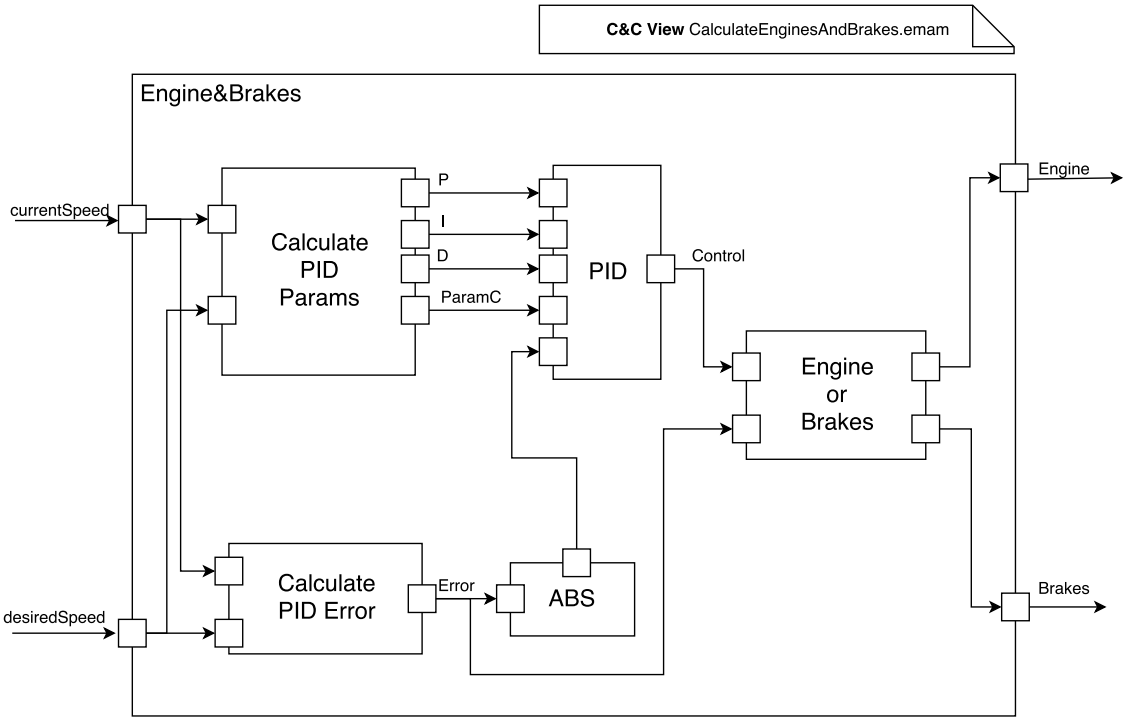

Figure 3.5: Calculating engines and brakes with a linear PID controller.

Figure 3.5 shows a PID controller which takes the amount of the speed difference into consideration. It consists of 5 subcomponents which are described in the following.

1. **PID Error:** The component's main purpose is to increase the error if its bigger

than a threshold. This enables a stronger acceleration or deceleration.

2. **Abs:** Since the PID component computes its output by multiplying the calculated PID error with other input values (visible in Figure 3.5), the error has to be always positive. That is why the component Abs calculates the absolute value of the error.

3. **PID Params:** The coefficients for the proportional, integral, and derivative terms were determined by empirical values [Ryd18].

4. **Brake or Engine:** Figure 3.5 shows, that the unchanged error gets forwarded to this component. The algebraic sign of the error decides whether the car is supposed to accelerate or decelerate. The output of the PID component (*control*) determines the strength.

5. **PID:** Listing 3.2 shows the implementation of the PID component. In line 8 the current error is compared to the previous error. Since there is no previous error during the first iteration, *isPrevErrorSpecified* is introduced in order to skip the comparison. The *control* is made up us follows (line 10). Each of the PID parameters are multiplied to their belonging counterpart. While the error is computed in another component, the integral and the derivative are calculated here in line 5 and 8. The sum of these multiplications forms then the degree of acceleration or deceleration.

```
1  implementation Math {
2    static B isPrevErrorSpecified = false;
3    static Q prevError = 0.0;
4    static Q acc = 0.0;
5    acc = paramDecayCoefficient * acc + error;
6    Q drv = 0;
7    if isPrevErrorSpecified
8      drv = error - prevError;
9    end
10   control = paramP * error + paramI * acc + paramD * drv;
11   prevError = error;
12   isPrevErrorSpecified = true;
13  }
```

Listing 3.2: Implementation Math of calculating the PID control.

Steering

Now that we have calculated Brakes and Engine only Steering is reaming. Figure 3.6 shows the two major factors which are needed to calculate the desired steering angle.

1. **Signed Angle:** Describes the angle difference between the desired direction and the current direction coming from the current position. In Figure 3.6 the angle is negative because the vehicle needs to steer left.

2. **SignedDistanceToTrajectory:** Describes the direct distance to the next point on the trajectory and also determines on which side of the route the current position is located. The value is negative in Figure 3.6 because the vehicle needs to steer left in order to decrease the trajectory error.

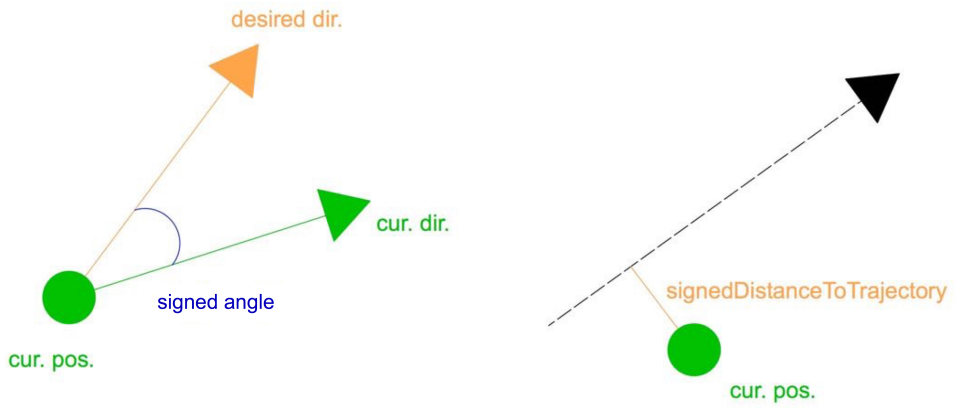

Figure 3.6: Two values which are needed to determine the desired steering angle.

The following equation shows how these two values form together the desired steering value.

$$\text{Steering} = \text{SignedAngle}(\text{curDir}_{vector}, \text{desiredDir}_{vector}) + \text{TrajectoryError} \tag{3.2}$$

Since real tires are limited in their maximum steering angle, we have to ensure that the calculated value is within these bounds. Listing 3.3 shows how a component with this purpose can look like. By connecting different constants this component is used to make sure Engine, Brakes and Steering does not exceed their bounds. Thus, this is a good example for the reusability of components in order to save work, time and complexity.

```
component EnsureBounds {
  ports
    in Q lowerBound,
    in Q upperBound,
    in Q input,
    out Q output;

  implementation Math {
    if input < lowerBound
      output = lowerBound;
    elseif input > upperBound
      output = upperBound;
    else
      output = input;
    end
  }
}
```

Listing 3.3: EnsureBounds is an exemplary reusable component.

Chapter 4

Simulation

Having the implementation done, one wants to see the result as if it would appear in the real world. To enable a preferable realistic environment the simulation map represents a map from the real world.

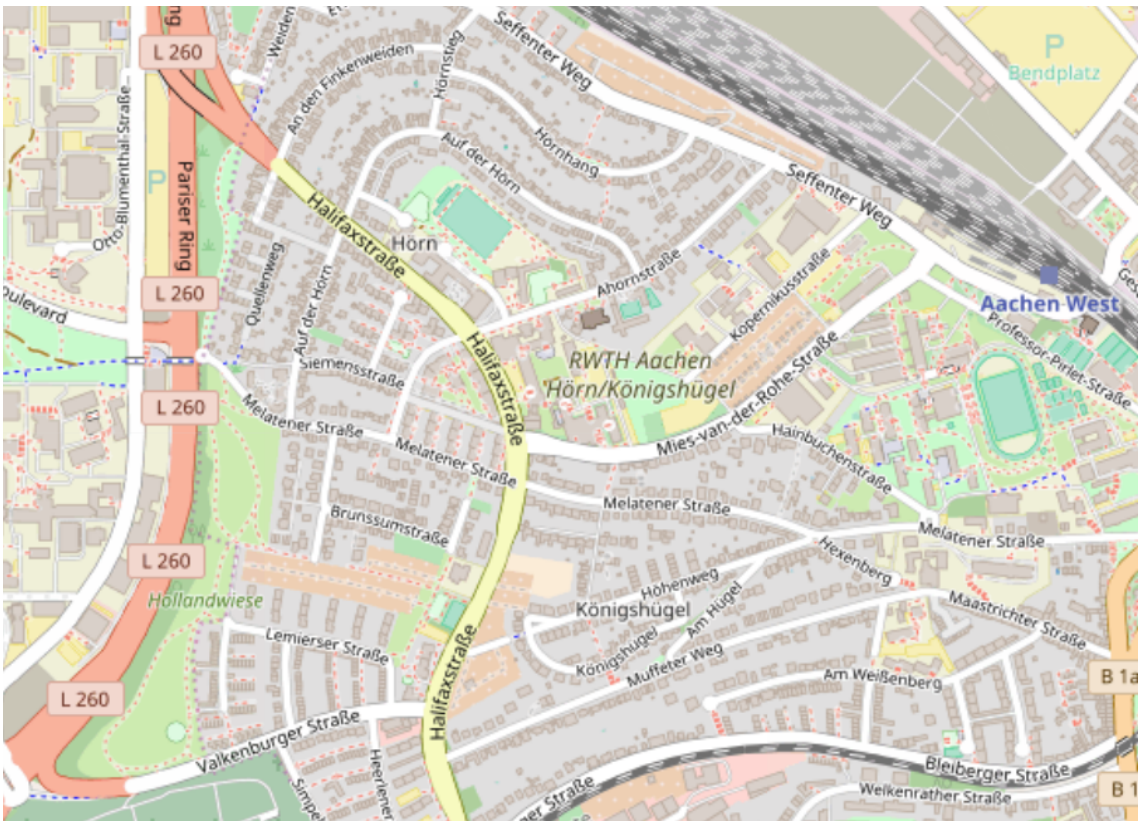

Figure 4.1: Map used for simulating self-driving car scenarios [Ryd18].

The map shows the streets of the area called Königshügel in Aachen, Germany. OpenStreetMap (OSM) provides the map data in form of either an isolated node or a list of nodes [HW08]. While smaller objects like trees or a traffic light are represented by a single node, streets consists of several nodes. Since linear vector operations are used for all calculations it is necessary that maps are represented in the Euclidean Space. Choosing

nodes for the simulations is done by using scenarios which contain the information about the start and finish node.

4.1 Result

Figure 4.2: Simulating a self-driving car with MonitCAR

Figure 4.2 displays a screenshot of our simulation. One can choose between several views during runtime. When the simulation is started, the physical input values are getting transformed by our autopilot.emam in order to move the car. We created three test case scenarios. First we let the car drive straight forward for a set time and speed. The second scenario wants the car to drive a bit, make a turn. Finally we let the car drive from point A to B in order to make a u-turn. The results are filmed and uploaded[1]. The implementation is uploaded as well[2].

[1] https://www.dropbox.com/sh/sovyk2nhbhbt8k0/AADk3ANZezrZ-k1PfDc52dzDa?dl=0

[2] https://github.com/EmbeddedMontiArc/EMAM2Cpp/tree/master/src/test/resources/de/rwth/armin/modeling/autopilot

Chapter 5

Conclusion

We investigated a modeling language which is announced to have a rich feature set. In order to examine its actual performance, we tried to implement a self-driving car. The car is supposed to find its way onto the route, detect straight road sections in order to accelerate and tight curves to decelerate.
The framework provides a reasonable amount of physical input values. Thus, the design phase is reduced to the conceptual design. At this, the benefits of C&C modeling are clearly noticeable. First of all one creates a component for every desired feature. Then it gets clear early on that subcomponents are needed. The developer intuitively decomposes the components and narrows them down to the low-layer implementation. One great advantage of MontiCAR is that developer can implement their components behavior within the same file. The MontiMath environment turns out to be very powerful. During the implementation we faced no operation which is not supported by MontiMath. Thus MontiCAR enables one to model both architecture and behavior. Architecture is modeled via components and their interconnections and behavior is modeled by using MontiMath (implementation Math function).

Nevertheless one point of criticism in MontiCAR is that it does not provide data structure reuse. We noticed in our implementations that some components used e.g the same data structure. But since this components were independent from each other, one had to copy and paste the implementations.
Another fact that might bother developers is that the simulation can only be tested when the implementation of the whole system is done. Of course one can test individual components using stream tests. But since the outcome dependents on the interaction of several components, the visualization has to wait until the end.

As mentioned in the introduction, MontiCAR provides ten of twelve features which are desired for DSLs [EK17]. In chapter 3 we used some of them to implement the self-driving car. Each of them contribute to an intuitive, understandable but also powerful system.

MontiCAR is still being developed. Currently a web page is developed which allows to edit and execute EMAM code. When executed, the web page should run the simulation. Besides that, the language family gets permanently enhanced.

Acknowledgments

I would like to thank Prof. Dr. Bernhard Rumpe for supervising this paper. I am very thankful for Michael von Wenckstern who helped with constructive feedback during my work which helped me to improve this project. Finally I would like to thank Alexander Rydin who was a huge support during design and implementation phase.

Bibliography

[Bun17] Statistisches Bundesamt.
Verkehrsunfälle. Destatis, 2017.

[EK17] Bernhard Rumpe Michael von Wenckstern Evgeny Kusmenko, Alexander Roth.
Modeling Architectures of Cyber-Physical Systems. Springer, 2017.

[EMA] EmbeddedMontiArc documentation website
`www.github.com/EmbeddedMontiArc/Documentation/`.

[FG17] Alexander Roth Bernhard Rumpe Michael von Wenckstern Filippo Grazioli, Evgeny Kusmenko.
Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles. Springer, 2017.

[HK10] Bernhard Rumpe Holger Krahn, Steven Völkel.
MontiCore: a framework for compositional development of domain specific languages. Springer, 2010.

[HW08] Mordechai Haklay and Patrick Weber. Openstreetmap: User-Generated Street Maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.

[Ryd18] Alexander Rydin.
Modeling of Component-and-Connector Architectures for Autonomous Vehicles. 2018.

[www18] EmbeddedMontiArc Git Repository
`https://github.com/EmbeddedMontiArc/`, January 2018.