RWTH Aachen University
Software Engineering Group

Model Predictive Trajectory Control and Simulation for Self-Driving Vehicles

Master Thesis

presented by
Richter, Christoph

1st Examiner: Prof. Dr. B. Rumpe

2nd Examiner: Prof. Dr.-Ing. Stefan Kowalewski

Advisor: Dipl.-Ing. Evgeny Kusmenko

The present work was submitted to the Chair of Software Engineering

Aachen, August 9, 2018

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Name, Vorname/Last Name, First Name

Matrikelnummer (freiwillige Angabe)
Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties. I have used 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.

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever 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.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
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 through 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.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Ort, Datum/City, Date

Unterschrift/Signature

Kurzfassung

Softwareentwicklung für Cyber-Physikalische Systeme hat in den letzten Jahren stark an Bedeutung gewonnen. Steigende Sicherheitsanforderungen zusammen mit immer komplexeren Steuerungssystemen sind speziell für die Automobilindustrie eine große Herausforderung. In der Automobilindustrie kommen häufig C&C Modellierungssprachen zum Einsatz, um domänenspezifischen Anforderungen gerecht zu werden. Die MontiCAR-Sprachfamilie hilft durch ein striktes Typsystem und die Unterstützung von physikalischen Einheiten den Anforderungen gerecht zu werden.

Zurzeit unterstützt MontiCAR noch nicht die Modellierung und Lösung von Optimierungsproblemen, wie sie zum Beispiel in modelprädiktiven Regelungen zum Einsatz kommen. Diese Arbeit erweitert MontiCAR, sodass dies möglich ist. Die Sprache MontiMathOpt erweitert die bestehende Sprache MontiMath um die Fähigkeit Optimierungsprobleme, möglichst nah an der mathematischen Formulierung, zu modellieren. Diese Sprache wird in die C&C Sprache EmbeddedMontiArcMathOpt (EMAMOpt) eingebettet. Ein flexibel erweiterbarer Codegenerator erzeugt C++ Code aus EMAMOpt Modellen, um Optimierungsprobleme zu lösen. Dabei werden mehrere Optimierungssolver unterstützt.

Um die Fähigkeiten von EMAMOpt zu testen wurde eine modellprädiktive Fahrbahnregelung entworfen und in dem Simulator MontiSim, einem Simulator für autonom fahrende Fahrzeuge getestet.

Durch die sehr einfache Modellierung und Lösung von Optimierungsproblemen wird MontiCAR zu einem leistungsfähigen Framework, welches für diverse Anwendungsgebiete, wie zum Beispiel modelprädiktive Regelungen, maschinelles Lernen oder Energieoptimierung, eingesetzt werden kann.

Abstract

Software development for Cyber-Physical Systems has grown in the last years. Safety requirements and fast growing complexity of control systems are a huge challenge for the automotive industry. C&C modeling architectures support the development process of these software products and are accepted in industry. The MontiCAR modeling family provides powerful tools to cope with safety requirements and develop software for CPS by providing efficient modeling techniques, a strict type system and unit support.

MontiCARs problem is that optimization problems, needed for example for model predictive control, cannot be modeled and solved intuitively. This work extends the MontiCAR framework by language MontiMathOpt such that optimization problems can be modeled. EmbeddedMontiMathOpt (EMAMOpt) enables the embedding of MontiMathOpt into the C&C architecture. A flexible code generator supporting multiple optimization solvers has been developed to generate code from EMAMOpt models to solve the modeled problems.

To prove its capabilities, an MPC trajectory controller has been designed using EMAMOpt. This trajectory controller was integrated and tested in the autonomous driving simulator MontiSim.

MontiCAR is extended by a powerful and simple optimization interface which can be used in a wide area of applications.

Contents

Chapter 1

Introduction

Software development for Cyber-Physical Systems (CPSs) has become very important during the last decade. The connection between mechanical and electrical systems and software is one key challenge in the industry [LE14]. In automotive industry safety relevant functions like trajectory planning, lane correction, engine control and battery management need robust development methods. To cope with the rising complexity of software systems and interdisciplinary challenges, component and connector (C&C) models are used in software development. Especially in automotive development, C&C languages are used. C&C models have the advantage that complex features can be decomposed into smaller components which can be developed by domain experts [KRRvW17]. Additionally, the software components have to be tested and verified because of their safety relevant functions [GKR+17].

To improve this model based approach, the textual modeling language MontiCAR [KRRvW17] was developed at the chair for software engineering at RWTH Aachen University. Compared to other C&C modeling languages like Simulink [TM18], Modelica [A+05] or SysML [FMS14] used in industry, MontiCAR offers a strict type system, accuracy and unit support. MontiCAR models can be statically verified. These features improve the safety requirements.

MontiCAR was evaluated on different use cases. Trajectory control aims to make the vehicle follow a given trajectory with minimal deviation of the actual path to the reference path. Therefore, the trajectory regulator controls engine, brakes and steering of the vehicle. A trajectory controller was already developed by [Ryn18] using MontiCAR. A PID controller [Abe14] was used to minimize the deviation to the desired trajectory. This approach does not use any predictions about the expected vehicle behavior. Trajectory control can be improved using model predictive control (MPC) [Abe14] [KPSB15].

Model predictive control was developed in the end of the 1970s. MPC describe a class of control methods which use a model of the underlying control process to predict the future state of the system over a finite time interval [Abe14]. MPC has been proven to be successful in autonomous driving scenarios [KPSB15].

In MPC, every time step a mathematical optimization problem is solved. An optimization problem is a problem where a function has to be either minimized or maximized under given constraints [BV04]. Different solving methods exist for different optimization problem classes. The software component solving the optimization problem is called *solver*.

The goal of this work is to develop a trajectory controller using MPC to improve the existing controller developed using MontiCAR. However, the MontiCAR framework can be optimized. An evaluation of various use cases has shown that MontiCAR is not able to solve optimization problems easily. In order to use MPC, it is required to minimize the error between planned and actual trajectory, by solving an optimization problem.

The MontiCAR language family is extended such that it is able to model optimization problems close to their mathematical formulation. Based on these models, executable code is generated. This code shall support multiple solvers for optimization, whereas the model shall be independent of the used solver. Focus of this work is language development which enables MontiCAR to solve optimization problems of a general way. When the framework is extended, this version should be used to design a trajectory controller using MPC.

The next paragraph gives an outline of this work. It describes the procedure how to accomplish the previously stated goal of this thesis and gives background knowledge. This introduction presents context, motivation, problem and goal of this thesis. After the introduction, related work of topics domain specific languages, modeling optimization problems and model predictive trajectory control will be discussed to give an overview of previous research about this topic. In the following chapter, preliminaries will give theoretical background knowledge about optimization, model predictive control and car models which act as model in MPC. After that, the tools are described to extend the language MontiCAR and its foundations, as well as MontiCAR itself. Not only the language development is covered, but also a few parts about simulation and development of these controllers.

The first challenge of this work is to choose an appropriate solver, which is capable to deal with optimization problems occurring in MPC. For this purpose, multiple solvers are compared according to different properties. Based on that comparison, a decision for two solvers is made. The first solver is used to optimize the MPC formulation and other more general nonlinear problems. The second solver serves as proof of concept, to demonstrate that solvers can be exchangeable and fulfill different requirements.

In the next step MontiCAR is extended to deal with optimization problems. The first task is the extension of MontiCAR's math language. The focus is on simple modeling of optimization problems, close to their mathematical representation. The developed extension of the math language then is integrated into the C&C language EmbeddedMontiArcMath, as part of MontiCAR. Subsequently, the existing C++ code generator is extended to generate solving code using the selected solvers for the given optimization problem. Mechanisms will be used to make this generator easily expendable and allow adding more solvers in a simple way.

After the language infrastructure is developed, it is used to model and generate a trajectory controller using MPC. This controller will be used in the simulator for autonomous driving cars MontiSim.

Chapter evaluation discusses the results of the development. The resulting language extension is evaluated with different kinds of optimization problems. Its syntax and modeling capabilities are compared to both, modeling and programming languages. The development process of the trajectory controller is reviewed and alternatives are evaluated.

At the end of this thesis the results will be summarized and future work will be discussed.

Notation The notation used in this master thesis is the standard notation used in math. Number spaces are the real numbers $\mathbb{R}$, rational numbers $\mathbb{Q}$, whole numbers $\mathbb{Z}$, natural

numbers starting with 1 $\mathbb{N}$ and complex numbers $\mathbb{C}$. Matrices and vectors over those spaces are expressed as $\mathbb{R}^m$ for column vectors and $\mathbb{R}^{m \times n}$. In the text column vectors are written as transposed row vectors:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = (x_1, x_2, x_3)^T \tag{1.1}$$

The elements of a vector $x \in \mathbb{R}^m$ are denoted as $x_1, \ldots, x_i, \ldots, x_m$, where index i indicates the row / column of the element. Matrix elements of matrix $A \in \mathbb{R}^{m \times n}$ are written as $a_{11}, \ldots, a_{ij}, \ldots, a_{mn}$. Index i denotes the row of the element and j the column. Note that in the presented modeling languages matrix element access is expressed as `x(i,j)`.
Sets of size k are enclosed with curly bracket and the set elements are separated by comma: $S = \{elem_1, \ldots, elem_k\}$. The closed interval between two numbers a and b is written as $[a, b]$. For non-closed interval an ordinary bracket is used, for instance $(-\infty, \infty)$ or $[0, \infty)$.

A function f which maps an input of k elements of a number space to one element in another number space is denoted as: $f : \{\mathbb{R}, \ldots, \mathbb{R}\} \to \mathbb{R}$. Conditions on certain values are denoted by $(v \mid condition)$. Derivatives of the function $f(x)$ are denoted as $f'(x) = f\frac{df}{dx}$ (first derivative) and $f''(x) = f\frac{df}{dx^2}$. In physics derivatives over time t are often expressed as $\dot{f}(t) = f'(t)$ and $\ddot{f}(t) = f''(t)$. This notation is also used in this work.

Chapter 2

Related Work

A model based development approach for autonomous driving cars was presented in *Model-Based Development of Self-Adaptive Autonomous Vehicles using the SMARDT Methodology* [HKK+18]. The SMARDT (Specification Methodology Applicable to Requirements, Design, and Testing) enables automated test generation, based on the requirement specification and design models formalized in SysML. The paper *Modeling Architectures of Cyber-Physical Systems* [KRRvW17] presents the modeling language family MontiCAR which is designed to model CPS effectively. MontiCAR's advantages, compared to other modeling languages like Simulink, Modelica or SysML, are for example unit and accuracy support. MontiCAR already had been used to model and implement a trajectory controller for autonomous driving cars in the master thesis *Modelling of Component-and-Connector Architectures for Autonomous Vehicles* [Ryn18]. This trajectory controller uses a PID controller to calculate the needed signals for engine, brakes and steering.

There are many papers published, presenting MPC controller for trajectory control [Fal07] [Abb11] [CGG+13] [KPSB15]. Some approaches solved the nonlinear MPC optimization problem directly using NLP solvers, others linearized the problem to archive better performance. The difference to this work is that for designing the MPC trajectory controller, the model based approach from MontiCAR was used.

Approaches like *CVXGEN: a code generator for embedded convex optimization* [MB12a] describe code generation of convex optimization problems. *Code generation for receding horizon control* [MWB10] already generates code to solve MPC problems automatically, in an optimized way.

This work chooses a similar approach compared to [MB12a]. A modeling language with integrated code generator is developed to solve optimization problems. The difference to [MB12a] is that not only convex optimization is used. This work combines the domain specific modeling language MontiCAR with the modeling and code generation of optimization problems. Thus, the benefits of MontiCAR are combined with simple and efficient code generation for optimization problems.

Chapter 3

Preliminaries

This section discusses theoretical and technological prerequisites, needed to implement an optimization language and develop a trajectory controller for autonomous driving cars and its simulation.

3.1 Theoretical Background

First the theoretical background for optimization problems are presented. Subsequently, the idea of model predictive control (MPC) is covered. Finally, car models are presented, needed as model for the MPC controller.

3.1.1 Optimization

This section is based on the definition of optimization problems by [BV04]. A mathematical optimization problem has the following general form:

$$\begin{array}{lll} \text{minimize} & f_0(x) & \\ \text{subject to} & f_i(x) \leq b_i, & i = 1, \dots m. \end{array} \tag{3.1}$$

In this case $x \in \mathbb{R}^n$ is called *optimization variable*. Function $f_0 : \mathbb{R}^n \to \mathbb{R}$ is the *objective function*. The set of functions $\{f_i : \mathbb{R}^n \to \mathbb{R} | i = 1, \dots, b_m; m \in \mathbb{N}\}$ are called *constraints*. The constraint functions are bound by constants $b_1, \dots, b_m$ and are called *bounds* of the constraint. A vector x^* is called *optimal* if it has the smallest *objective value*(result of the objective function) over all possible vectors satisfying the constraints.

Equation 3.1 shows a minimization problem. It is also possible to formulate maximization problems (sometimes also called allocation problems). A maximization problem can be easily transformed into a minimization problem by negating the objective function: $\max_{x \in \mathbb{R}^n} f(x) = \min_{x \in \mathbb{R}^n} -f(x)$.

For convenience, short hand notations are introduced:

$$\begin{array}{lll} \min\limits_{x \in \mathbb{R}^n} & f(x) & \\ \text{s.t.} & b_L \leq g_i(x) \leq b_U & i = 1, \dots, m \end{array} \tag{3.2}$$

Problem Class	Formula	Description
Linear Program (LP)	$f_i(cx+dy) = cf_i(x)+df_i(y) \mid \forall x, y \in \mathbb{R}^n \forall c, d \in \mathbb{R}, i = 0, \dots m$	objective function and constraint functions are linear
Convex Program (CP)	$f_i(cx+dy) \leq cf_i(x)+df_i(y) \mid \forall x, y \in \mathbb{R}^n \forall c, d \in \mathbb{R}, i = 0, \dots m$	objective function and constraint functions are convex
Quadratic Program (QP)	$f_0(x) = cx + x^T Qx \; f_i(x) = dx + x^T R_i \leq b_i \mid i = 1, \dots m$	objective function and constraint functions are quadratic
Nonlinear Program (NLP)	$f_i(x) \mid f_i$ nonlinear	objective function or constraint functions are not linear
Mixed Integer Linear Programming (MIP)	$f_i(cx + dy) = cf_i(x) + df_i(y) \mid \exists x_j, y_j \in \mathbb{Z}^n \forall c, d \in \mathbb{R}, i = 0, \dots m$	Additionally to LP some or all variables can only be whole numbers.

Table 3.1: Optimization Problem Classes [BV04] [GAM18]

In this case, the constraints $g(x)$ have a lower b_L and an upper bound b_U. If $b_L = b_U$ the operator $\leq$ can be replaced by equality $=$.

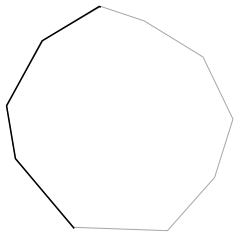
Optimization problems can be grouped into classes, characterized by properties of the objective function and the constraints. Table 3.1 lists the most important ones for this work.

In mixed integer programming, classes exist for QP and NLP respectively MIQP and MINLP.

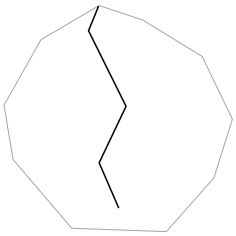
An algorithm that computes the solution of an optimization problem for a certain class is called *solution method*. Since the late 1940s, there have been many algorithms developed to solve an optimization problem of a certain class. There are solvers taking advantage of sparsity of a problem or other properties. A problem is called sparse, if the constraints only depend on a few variables [BV04].

There are effective solution methods for linear and convex and quadratic problems. For nonlinear problems, there is no effective method available in general. Optimization can be divided into local and global optimization. Instead of finding all optimal points, in local optimization only a local optimum is calculated dependent on an initial point. Typically, local optimization methods are faster and more scalable than global methods. The most important disadvantage is that finding a global optimum is not guaranteed. Global optimization methods find the true optimum but are more computational expensive [BV04].

Prominent solving algorithms are the simplex algorithms [BV04] and interior point methods [BV04]. Solutions of an optimization problem can be considered as polyeder in n dimensional space. Every solution x corresponds to one node of the polyeder (also called simplex). While simplex algorithms iterate over the surface, interior point methods choose an iterative path through the inner of the polyeder (see figure 3.1). Figure 3.1 shows a 3 dimensional polyeder. On the left side, a solution of the simplex algorithm is visualized, on the right a solution of an interior point algorithm. Interior points methods have the advantage that they typically converge faster for sparse problems than simplex methods. Simplex methods have the advantage that they take benefit from already computed solutions and thus need less iteration steps, if the problem was changed slightly (warm start) [BV04].



(a) Simplex Algorithm



(b) Interior Points Algorithm

Figure 3.1: Visualization of simplex and interior points algorithm

Concrete solving methods are presented together with the corresponding solver in chapter 4.

3.1.2 Model Predictive Control

Model predictive control (MPC) describes a class of controller that have the following properties [Abe14]:

1. Explicit usage of a *process model* to predict future states z_t of the system.
2. Calculation of the output signal is based on the minimization of a *cost function*.
3. In every time step an optimal signal is computed for the next states within the prediction horizon h_p. Only the first output signal is used as control output. The future states are calculated again in the next time step.

In the case that a system state $z \in \mathbb{R}^k$ should be controlled such that it reaches a reference state $z_{ref} \in \mathbb{R}^k$, an MPC minimization problem can be formulated as follows [Abe14]:

$$\begin{aligned} \min_{u \in \mathbb{R}^{n \times h_p}} \quad & \sum_{i=1}^{h_p} ||z_i - z_{ref,i}||^2 \\ \text{s.t.} \quad & u_{min} \quad \leq \quad u_i \quad \leq u_{max} \qquad \forall i \in [1..n] \end{aligned} \tag{3.3}$$

The prediction horizon h_p indicates how many future state predictions are made. u denotes the output signal of the controller for every step within the prediction horizon. Only the first column of u is used as output signal in the current time step. The other columns are only used for calculating the future state and are discarded after each time step. z_i is the predicted state in time step i. This term actually is a function dependent on the process model and the previous time step. Let $f : \mathbb{R}^n \times \mathbb{R}^k \to \mathbb{R}^k$ be the state update function of the underlying model. This function calculates the future state z_{i+1} as follows:

$$z_{i+1} = f(u_i, z_i) \tag{3.4}$$

$z_{ref,i}$ denotes the desired reference state at time step i. Bounds u_{min} and u_{max} are constant bounds on the output signal u. Term $||z_i - z_{ref,i}||^2$ denotes the error made at time step i of the prediction horizon. In this simple case the error is just the mean squared error (MSE) of actual state to the reference state.

To have more control over the system behavior positive semi definite diagonal weight matrices $Q \in \mathbb{R}^k$ and $R \in \mathbb{R}^n$ are introduced [Abe14] [KPSB15]:

$$\begin{aligned} \min_{u \in \mathbb{R}^{n \times h_p}} \quad & \sum_{i=1}^{h_p} (z_i - z_{ref,i})^T Q (z_i - z_{ref,i}) + \sum_{i=0}^{h_p - 1} (u_{i+1} - u_i)^T R (u_{i+1} - u_i) \\ \text{s.t.} \quad & u_{min} \quad \leq \quad u_i \quad \leq u_{max} \qquad \forall i \in [1..n] \end{aligned} \tag{3.5}$$

The cost function now calculates a weighted error. The weights in Q control the influence of a single element of z on the total error. Weights R weight the difference on the control outputs $u_{i+1} - u_i$. Higher weights on the control output lead to a smoother behavior of the control signal u.

There can be different formulations but this one is most relevant for this thesis.

MPC is the most used advanced control mechanism in industry. It has multiple advantages compared to other control mechanisms [Abe14]:

- Simple handling of multiple coupled system states.
- Explicit defined bounds onto input and output signals.
- Implicit compensation of dead time.
- Consideration of future reference trajectories.
- More intuitive for engineers without education in control theory.

MPC is used in the trajectory controller for this thesis.

3.1.3 Car Models

The goal was to design a trajectory controller for autonomous driving cars using MPC. After introducing model predictive control there has to be a*model* which describes the vehicle's behavior. A car model describe the vehicles dynamic behavior, dependent on some input signals. These models are needed to predict future states of the vehicle in the MPC controller.In the subsequent model the state vector $z = (x, y, \psi, v)^T \in \mathbb{R}^4$ consists of x and y position in the global coordinate system, yaw angle ψ (orientation in the global coordinate system) and the current velocity v. The input vector u depends on the car model. In the following subsection two different car models are presented.

Kinematic Bicycle Model

The bicycle model is a simplification of a real vehicle, where only two tires are represented (similar to a bicycle) [Cor13]. Kinematic means that only the kinematic of the vehicle is modeled. Dynamic elements like e.g. force are not considered in the kinematic version, in contrast to the dynamic bicycle model [KPSB15]. Figure 3.2 visualizes the model and contains all relevant parameters.

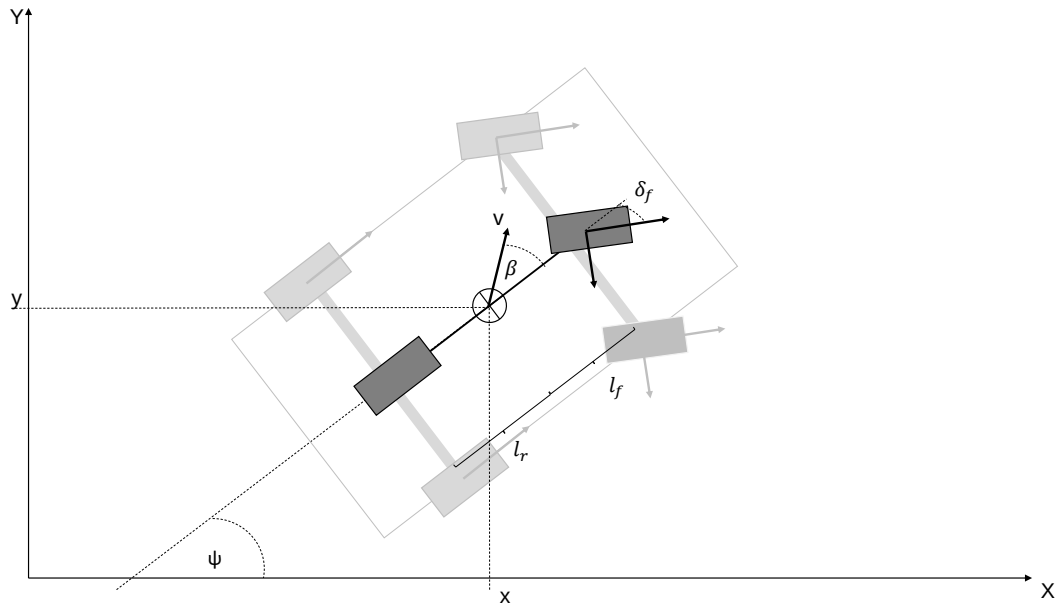


Figure 3.2: Kinematic Bicycle Model [KPSB15]

Input parameters $u = (a, \delta_f)^T \in \mathbb{R}^2$ are the acceleration a and the steering angle of the front wheel δ_f.

The state update function $f : \mathbb{R}^2 \times \mathbb{R}^4 \to \mathbb{R}^4$ is given by:

$$f(u, z) = z + \dot{z} = \begin{pmatrix} x \\ y \\ \psi \\ v \end{pmatrix} + \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \\ \dot{v} \end{pmatrix} \tag{3.6}$$

with derivative values [KPSB15]:

$$\begin{array}{rcl} \dot{x} & = & v\cos(\psi + \beta) \\ \dot{y} & = & v\sin(\psi + \beta) \\ \dot{\psi} & = & \frac{v}{l_r}\sin(\beta) \\ \dot{v} & = & a \\ \beta & = & \tan^{-1}(\frac{l_r}{l_f + l_r} tan(\delta_f)) \end{array} \tag{3.7}$$

Figure 3.2 visualizes the model parameters. Angle β indicates the direction of the velocity at the mass center in relation to the vehicle axis. l_f and l_r are the distances from the vehicle's center of mass to the front respectively rear wheel. Note that the state update function even for this simple model is nonlinear.

Matlab Vehicle Model

MathWorks® has developed a vehicle model [EN18] which in the following is called the Matlab Vehicle Model or just Matlab Model. In contrast to the Kinematic Bicycle Model,

the Matlab Model is a dynamics model. It respects the tire dynamic of all 4 wheels, as well as longitudinal C_x and lateral tire stiffness C_y. C_A is the air resistance coefficient, m the mass of the vehicle and l_f, l_r the distances of front and rear axle to the center of gravity (see figure 3.3).

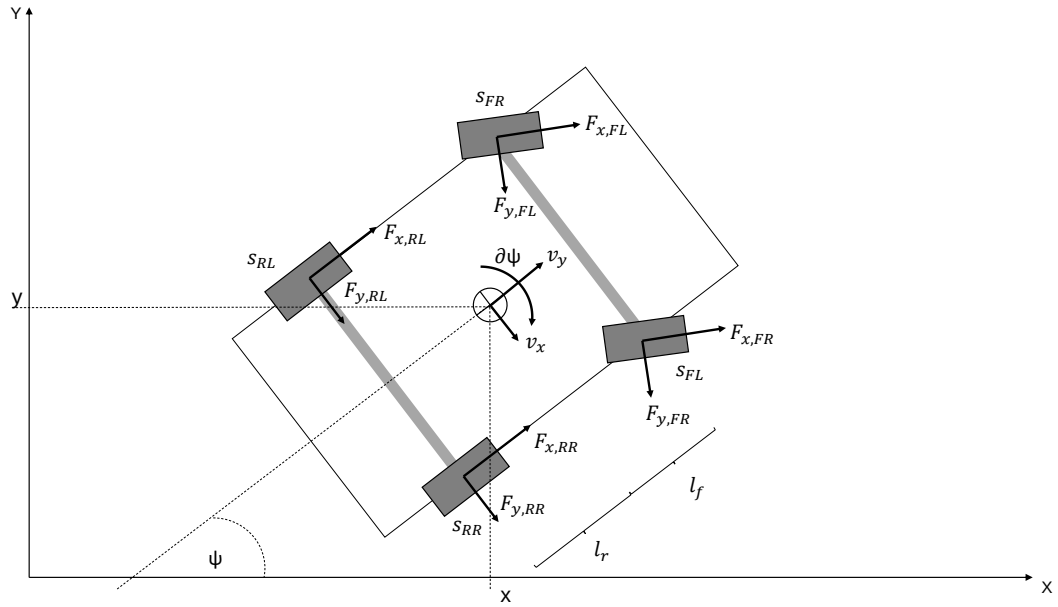


Figure 3.3: Matlab Vehicle Model [EN18]

Input vector $u \in \mathbb{R}^5$ is defined as $u = (s_{FL}, s_{FR}, s_{RL}, s_{RR}, \delta_f)^T$ where s is the slip ratio of the tire and δ_f is the steering angle. In the Matlab Model the state vector is defined as $z = (\dot{x}, \dot{y}, \dot{\psi})^T \in \mathbb{R}^3$ The state update depends on differential equations $\ddot{x}, \ddot{y}$ and $\ddot{\psi}$ respectively.

$$\begin{aligned}
\ddot{x} &= \dot{y} * \dot{\psi} + \tfrac{1}{m} && ((F_{x,FL} + F_{x,FR}) * \cos(\delta_f) \\
&&& -(F_{y,FL} + F_{y,FR}) * \sin(\delta_f) \\
&&& +F_{x,RL} + F_{x,RR} \\
&&& -C_A * \dot{x}^2) \\
\ddot{y} &= -\dot{x} * \dot{\psi} + \tfrac{1}{m} && ((F_{x,FL} + F_{x,FR}) * \sin(\delta_f) \\
&&& +(F_{y,FL} + F_{y,FR}) * \cos(\delta_f) \\
&&& +F_{y,RL} + F_{y,RR}) \\
\ddot{\psi} &= \tfrac{1}{J} * (l_f && ((F_{x,FL} + F_{x,FR}) * \sin(\delta_f) \\
&&& +(F_{y,FL} + F_{y,FR}) * \cos(\delta_f)) \\
&&& -l_r * (F_{y,RL} + F_{y,RR}))
\end{aligned} \tag{3.8}$$

F is the force at the tire and J a moment of inertia. The tire forces are approximated by

$$\begin{aligned}
&F_{x,i} = C_x s_i \\
&F_{y,i} = C_y \alpha_i && \forall i \in \{FL, FR, RL, RR\} \\
&J = \tfrac{1}{(\frac{1}{2}(l_f + l_r))^2 m}
\end{aligned} \tag{3.9}$$

where α_i is the tire slip angle of tire i.

$$\begin{aligned} \alpha_F j &= \delta_f - \tan^{-1}(\dot{y} - l_f \frac{\dot{\psi}}{\dot{x}}) \quad \forall j \in \{L, R\} \\ \alpha_R j &= \tan^{-1}(\dot{y} - l_r \frac{\dot{\psi}}{\dot{x}}) \quad \forall j \in \{L, R\} \end{aligned} \tag{3.10}$$

3.2 Technological Prerequisites

In order to describe the MPC formulation, *domain specific languages* (*DSL*) are used. In computer science DSLs are languages, which allow to model a technical problem with its properties in an easier and faster way. They help the developer to describe requirements and to implement solutions [KKP+14]. The following subsections present tools, which are used to create the DSL EmbeddedMontiArcMathOpt (EMAMOpt). EMAMOpt is used to describe the controller used for trajectory control with its MPC formulation.
In particular MontiCore, a language workbench tool to create own languages [GKR+08] and MontiCAR, a textual modeling language for cyber-physical systems [KRRvW17] is explained. After the language prerequisites, the simulator framework MontiSim is introduced.

3.2.1 MontiCore

This section is based on the MontiCore Reference Manual 5.0 [HR17]. MontiCore is a framework for developing textual DSLs [GKR+08]. It enables modular grammar based on language definition. Multiple languages can be composed by inheritance, aggregation or embedding. Additionally, MontiCore provides an infrastructure for code generation.

Grammar

Grammars are the central notation in MontiCore. They describe context free grammars in a notation based on the Extended Backus Naur Form [HR17] [ASU86]. MontiCore uses ANTLR [Par13], an object oriented parser generator, as backend. Concrete and abstract syntax are defined by the grammar directly. From the grammar a parser, the *abstract syntax tree* (*AST*) and an infrastructure for managing symbols and scopes (*SMI*) are generated automatically.

To make grammars reusable MontiCore allows component grammars. A component grammar is an incomplete language in which a nonterminal can be declared, but no production rule is given. Component grammars can be defined by the keyword *component* (see listing 3.1). Nonterminals with no production rules can be defined by the keyword *external*. They are similar to abstract functions in Java. The nonterminal production rule must be implemented in a complete language using this component grammar. For component grammars there is no parser generated.

```
1 component grammar MyComponentGrammar {
2   external SomeNonTerminal;
3 }
```

Listing 3.1: MontiCore component grammar [HR17].

MontiCore already provides basic grammars which should be used to create own DSLs. They support the concept of reusability. The most important grammars for this work are presented in figure 3.4. Syntax and semantics of this diagram are explained later in figure 3.10. At this point, it is enough to know that the arrows model the interdependencies between the languages.

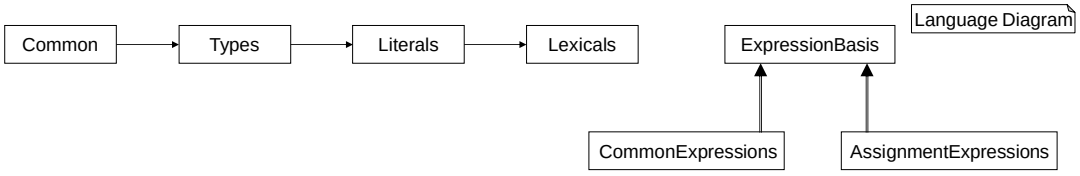


Figure 3.4: MontiCore Grammars Overview

The presented grammars are component grammars.

Abstract Syntax Tree

The abstract syntax tree (*AST*) is generated from the grammar automatically. For each nonterminal in the grammar, an AST class is generated. In the concrete syntax of every nonterminal an AST node is generated. An AST node is realized by an instance of the AST class in Java.

Different nonterminals have to be mapped to different AST classes. In the following a few mappings are explained:

Nonterminals on the right-hand side are mapped to the resulting AST class by composition (see figure 3.5).

Figure 3.5: Mapping of one nonterminal to AST [HR17].

If one nonterminal contains multiple nonterminals (+ or $\star$), they are composed as lists (see figure 3.6):

Figure 3.6: Mapping of multiple nonterminals to AST [HR17].

Optional nonterminals composed with `?` or `|` are mapped using Java Optionals [Abt18] (see figure 3.7):

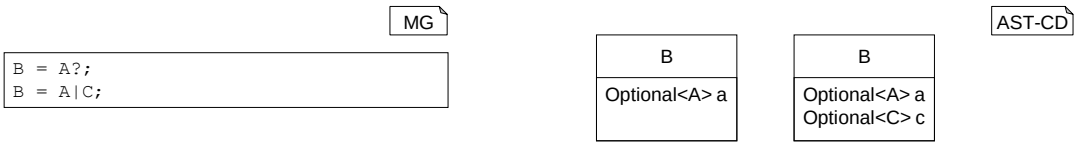


Figure 3.7: Mapping of optional nonterminals to AST [HR17].

Additionally, inheritance, interfaces and enumerations are mapped straightforward to the AST class diagram representation.

MontiCore provides a mechanism to add hand written code to the AST classes. For integrating handwritten code an AST class with the same name and package as the automatically generated class has to be created. This class must inherent form the automatically generated top class (see figure 3.8):

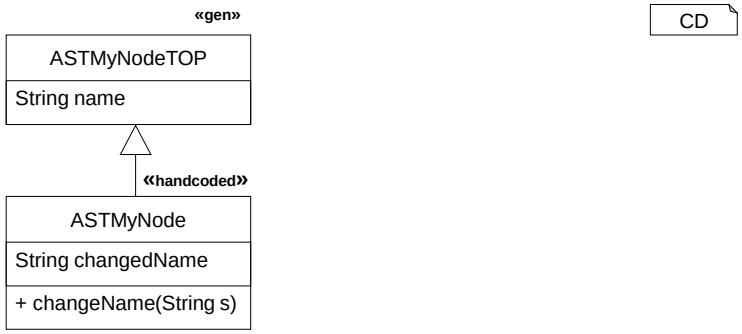


Figure 3.8: Handwritten AST extension [HR17].

This has the advantage that generated code can be reused and is automatically integrated in the framework.

Symbol Table

A *symbol* is an abstraction of AST nodes and nonterminals. It contains all relevant information, such as a name and context specific information. To access symbols easily, a *symbol table* is introduced. A symbol table is a data structure which stores all symbols in a hash map. In the symbol table each symbol can be accessed by its name. To reduce the visibility of all symbols scopes are introduced. Scopes are usually defined by a nonterminal that contains brackets. They divide the symbol table into smaller units. Symbols can be generated with the keyword *symbol* (see listing 3.2).

```
grammar MyGrammarWithSymbol {
  symbol scope SomeScope = "{" body:SomeScopeBody "}";
}
```

Listing 3.2: MontiCore Symbol creation example [HR17].

The symbol table is an additional abstraction layer from AST nodes and the grammar.

Context Conditions

Since MontiCore is based on context-free grammars, a mechanism is required to also handle context sensitive languages. For this purpose, MontiCore introduces *context conditions* (CoCo). CoCos are predicates that allow context sensitive restrictions of the language. They define the set of correct and well-formed models of a language [HR17].

CoCos can be checked at different positions during language creation:

- After building the AST.
- After or while creating the symbol table from the AST.
- After a modification of the AST.

Checking CoCos during or after creation of the symbol table has the advantage that symbols provide an abstraction layer and information may be easier to check. If the symbols are not needed, CoCos also can be checked in the AST only.

The needed infrastructure to create and check CoCos is generated by MontiCore automatically. Each CoCo gets a unique identifier of the form: (0xABC01). This is the error code displayed if the condition fails.

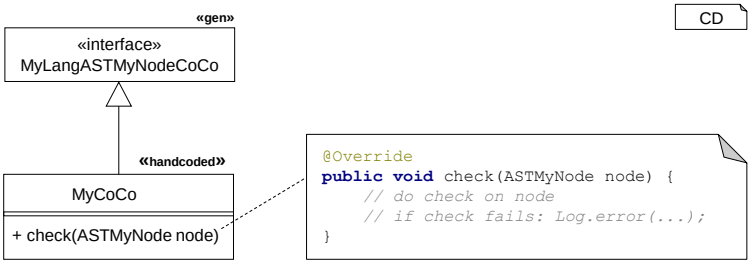


Figure 3.9: Automatically generated infrastructure to create a CoCo.

Figure 3.9 shows the entry point to create a new CoCo. The CoCo interface is generated automatically and declares the function `check(ASTNode)` for the corresponding AST node. A new CoCo has to implement this interface.

Best practice for a context condition is that every CoCo has a unique error code to easily identify errors. Error messages should be human readable, compact and contain the error's source code position. For the implementation MontiCore's logging infrastructure is used.

Language Composition

Modularity is a good practice in software development. MontiCore enables this concept also for languages. Languages can be composed by several other language modules or sub languages [Völ11]. This is one key feature MontiCore provides. It allows reusing grammars similar to classes in object oriented programming.

Basically MontiCore supports three different types of language composition:

1. *Language aggregation* is the usage of several grammar artifacts that are combined in the target language. The language artifacts still remain their own compilation unit. The target language is able to use all defined nonterminal symbols from the aggregated languages. Language aggregation is similar to aggregation in UML class diagrams.

2. *Language inheritance* or *language extension* is a mechanism to reuse an existing language. In this form of composition modifications of languages elements are allowed. Similar to object oriented programming nonterminals can be overwritten. For instance the difference between language inheritance and language extension is that extension avoids overwriting language elements such that they are not compatible to the parent version. Language extension is always backwards compatible, while language inheritance just reuses the parent language but may change the behavior of existing language elements. Note that MontiCore also allows multiple inheritance.

3. *Language embedding* allows embedding one entire language artifact into another language. This means that whole model parts of the embedded language are integrated into the target language. This composition is similar to aggregation in the backend but is more related to the integration of the embedded language into the frontend. An example for language embedding is shown in section 3.2.2.

Figure 3.10 shows the visualization which is used in the following chapters to describe language composition. The arrows are inspired by UML class diagrams and the notation [KRRvW17] have used. These diagrams are indicated as *Language Diagram* or in short form *LD* (similar to class diagram, object diagram etc.) and visualize the language relations and interdependencies. In the diagrams this indicator is written in its full form because it is not standardized.

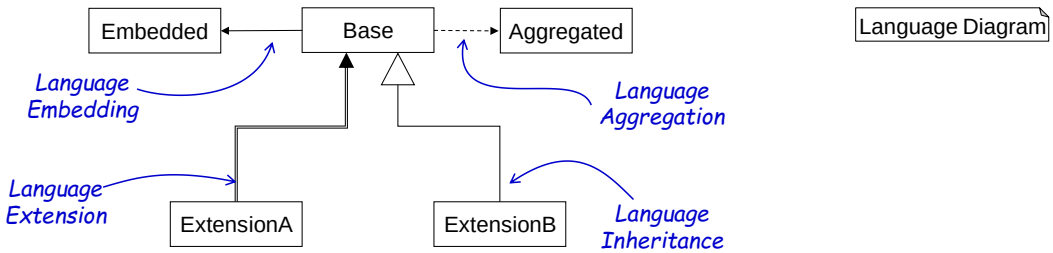


Figure 3.10: Language extension visualization.

3.2.2 MontiCAR

In the last section MontiCore, a powerful tool to create DSLs, is presented. The MontiCAR modeling language family [KRRvW17], a collection of DSLs which are used to model cyper-physical systems, has been created using MontiCore. MontiCAR aims to describe specific features of CPS in the domain of automotive or robotic e.g. trajectory planning, battery management or engine control. In comparison to other DSLs (e.g. Simulink [TM18], Modelica or SysML) in this domain MontiCAR offers, unit and accuracy support, a strict type system, component and connector arrays and name and index based auto-connections as well as an advanced math language [KRRvW17].

Figure 3.11 gives an overview about the different languages used in MontiCAR. The basis is built by the commonly used languages *NumberUnit*, *Resolution*, *Ranges*, *Types*, *Stream*, *Tagging* and *Common*. Languages *Math* and *EmbeddedMontiArc* are built on top of the common languages and serve as intermediate languages which are then combined to *EmbeddedMontiArcMath*, providing all previously mentioned properties.

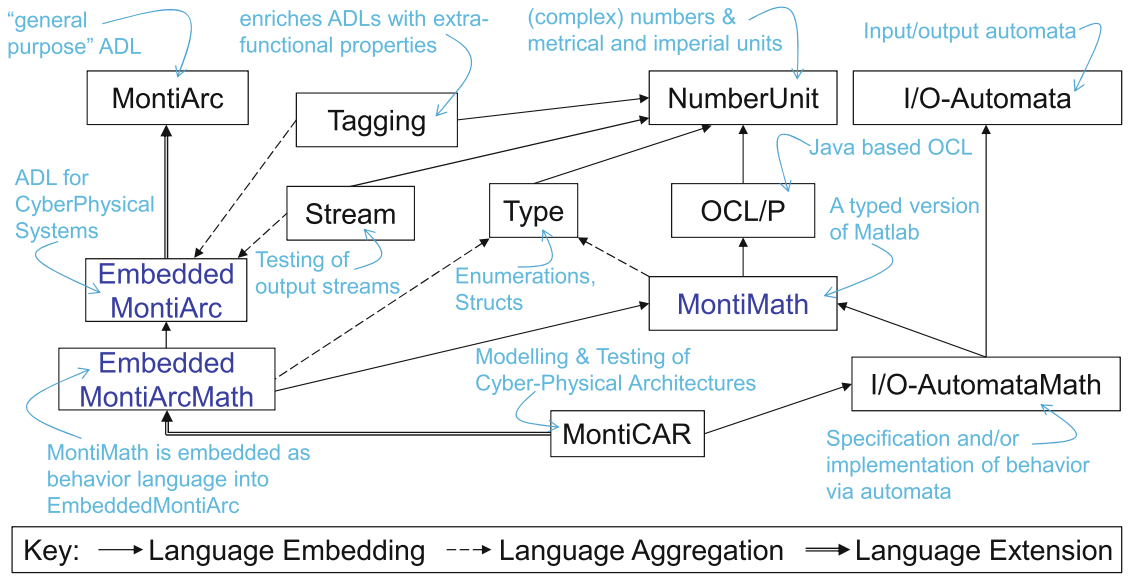


Figure 3.11: MontiCAR modeling language family [KRRvW17]

Tagging allows adding extra-functional properties to the C&C models [MRRvW16]. Stream is a DSL to test EmbeddedMontiArc (EMA) models. Stream models can be used as unit tests for EMA models [Ryn18]. Figure 3.11 visualizes the concept the of the language family. This concept is not fully implemented yet and may change because an agile software development process. Languages *I/O-Automata*, *I/O-AutomataMath*, *OCL-P* and *MontiCAR* are either not integrated into the framework or not implemented at the moment and thus are less relevant for this work.

NumberUnit The language NumberUnit extends MontiCore's basic grammar Literals (see figure 3.12).

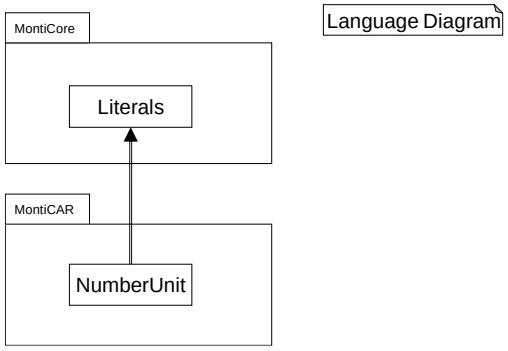


Figure 3.12: NumberUnit language hierarchy

Additional to the provided number tokens NumberUnit adds the following features:

1. Unit support for numbers. Every number optionally is followed by a unit. Units are divided into SI-units and imperial units. All base units for example length, temperature, weight, time, etc. and common derived units for instance energy, acceleration, force etc. are supported. Unit prefixes like kilo, centi, milli, nano, etc. are also supported.

2. Infinity support. Sometimes modeling infinity is needed. Ordinary numbers cannot represent infinity thus an infinity token ∞ is introduced. NumberUnit supports plus infinity and minus infinity.

3. Additionally, complex numbers are introduced. Complex numbers have a real part and an imaginary part.

All those features are collected in the nonterminal *NumberWithUnit*, which can be either a number without unit, a number with unit, a complex number or infinity.

Types Language *Types* extend the type system which is provided by Monti Core (without any language composition). Figure 3.13 gives an overview of the dependent languages. *Ranges* and *Resolution* introduce an interval and an accuracy in which a number can be defined. These serve as base for types like the natural numbers $\mathbb{N}$ which are defined in the interval $[1, \ldots, \infty]$ where the resolution is restricted to whole numbers.

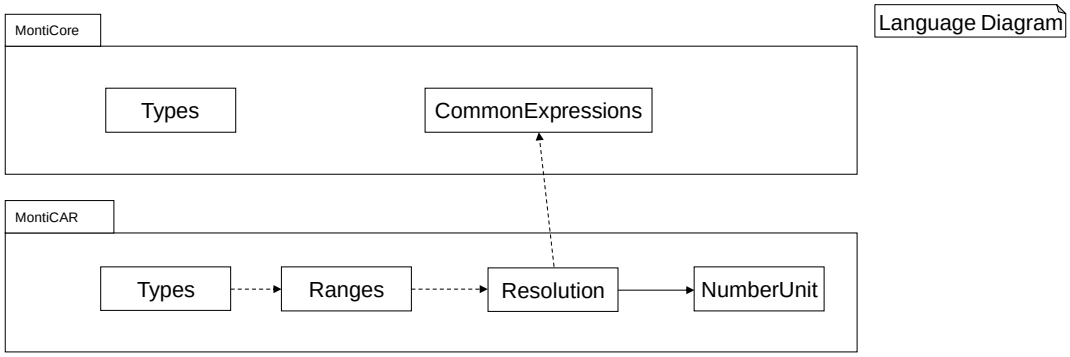


Figure 3.13: Types language hierarchy

The standard data types known in programming languages like double, integer etc. are not suitable to describe physical systems. The data types introduced in this language aim to describe numbers more mathematically. Types natural (N), whole (Z), rational (Q) and boolean (B) are used to describe the range of numbers (with unit) as well as their resolution. Additionally, the range of numbers can be expressed by intervals. Listing 3.3 shows the grammar definition in MontiCore. A definition of the natural numbers would look like this: `(1:1:∞)`.

```
1 Range =
2   "("
3   lowerBound:UnitNumberResolution ":"
4   (step:RangeStepResolution ":")?
5   upperBound:UnitNumberResolution
6   ")";
```

Listing 3.3: Interval definition in grammar Ranges.mc4.

Common The language *Common* serves as a single extension point that other languages can inherit from. It aggregates Types, Ranges and NumberUnit. Additionally, Common provides array support and certain keywords which can be used in sub-languages.

EmbeddedMontiArc EmbeddedMontiArc [KRRvW17] is based on the architecture description language (*ADL*) MontiArc [Hab16] and serves as the main language in MontiCAR. MontiArc is an ADL for logical architectures of distributed, interactive systems. Additional to MontiArc EMA adds unit support (see 3.2.2) the type system described in section 3.2.2, and port and component arrays. EMA follows the C&C architecture, allows connecting components via ports and create sub-components. The grammar is based on Common (see figure 3.14).

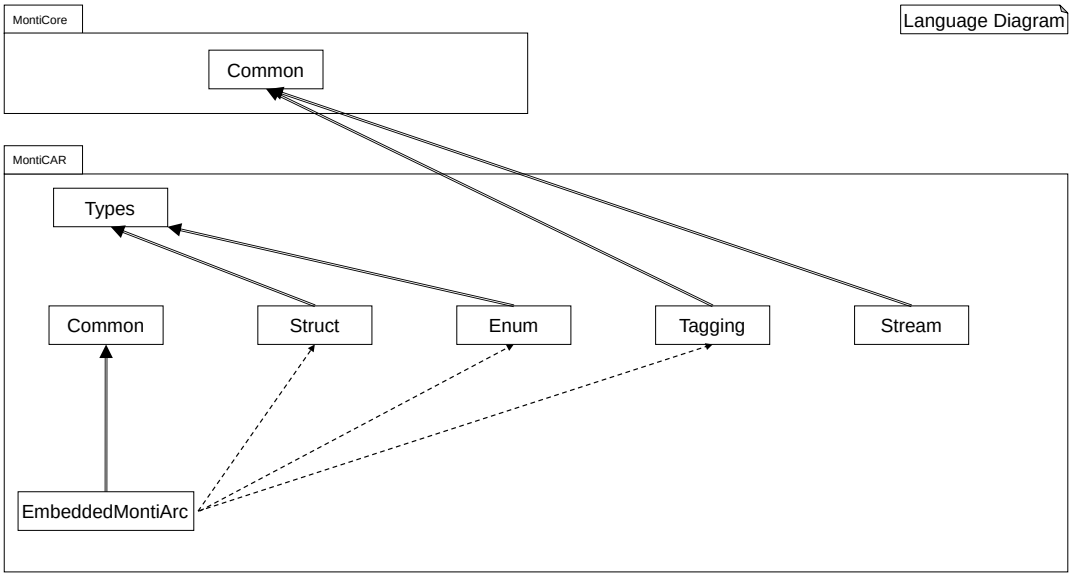


Figure 3.14: EmbeddedMontiArc language hierarchy

EmbeddedMontiArc models can be enriched with extra functionality using the *Tagging* language. Additionally, the presented type system the language *Struct* and *Enum* provide support of C like structs [Ryn18]. Steam can be used to test EMA components but there exist no composition between them.

MontiMath The language MontiMath (also called math) is a mathematical modeling language [KRRvW17]. MontiMath is matrix based and its syntax is similar to MATLAB [MAT18b]. It supports units, rational numbers, dimension information, range information and it keeps track of algebraic matrix properties (see [HJ12]). MontiCore has a strict type system to prevent runtime errors, which would be critical in CPS. At version 0.0.10 MontiMath does not reuse other languages from MontiCAR but defines types, ranges, statements and expressions by its own. An instance of MontiMath is called script and has filename suffix ".m". The language's skeleton builds a script which contains a list of *Statements*. A statement is an independent expression which can stand on its own (e.g. variable assignments or control flow statements). An *Expression* returns any value, but cannot stand on its own in MontiMath. This is a contrast to that other programming languages like Java where expressions have no effect or MATLAB where the result is printed to the command line output.

EmbeddedMontiArcMath EmbeddedMontiArcMath (EMAM) embeds MontiMath into EmbeddedMontiArc as shown in figure 3.15. This allows EMA to perform calculations on incoming and outgoing ports.

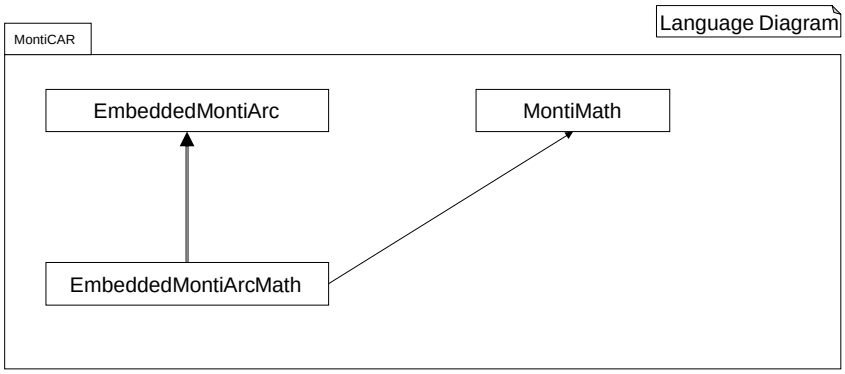


Figure 3.15: EmbeddedMontiArcMath language hierarchy

A math implementation can be added to the EMAM model using the keyword *implementation math*. This embedding of the math language allows to define the behavior of atomic EMA components.

EMAM to C++ Generator To generate code from the models MontiCAR provides a code generator (EMAM2CPP) which generates C++ code out of EMAM models [Sch18]. The generator is extended to also handle structs and enums [Ryn18] as well as tagging [Hel18]. EMAM2CPP is not a template based code generator, but uses a blue print approach which allows adding functions and lines of code dynamically. Additionally, it should generate code for the connection to the simulator MontiSim (see section 3.2.3) and optimize generated code by threading and algebraic optimizations [Sch18].

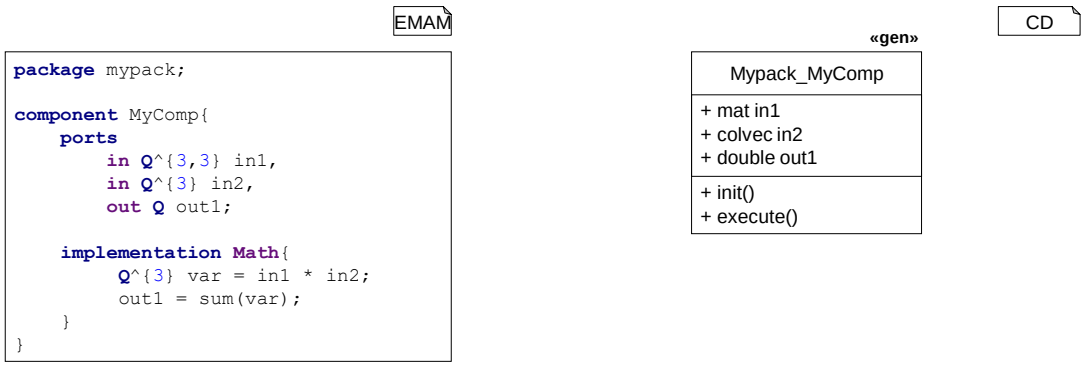


Figure 3.16: Code generation EMAM to C++

The key component is the class *ComponentConverter* (see figure 3.17) which takes a symbol as input and converts them into C++ code. Symbols can be either EMA symbols *ComponentSymbol* and *PortSymbol* or *MathExpressionSymbol*s. For every component a C++ class is generated which has the ports as public fields (see figure 3.16). Every MontiMath variable is initialized in an *init()* function. The *MathStatements* are executed in the *execute()* function. To generate the correct math instruction accordingly *ComponentConverter* delegates the math symbols to the static components `ExecuteMethodGenerator`

which chooses the C++ implementation. EMAM2CPP potentially supports multiple C++ backends to execute the BLAS operations on matrices. However, Armadillo [SC16] is used at version 0.0.18.

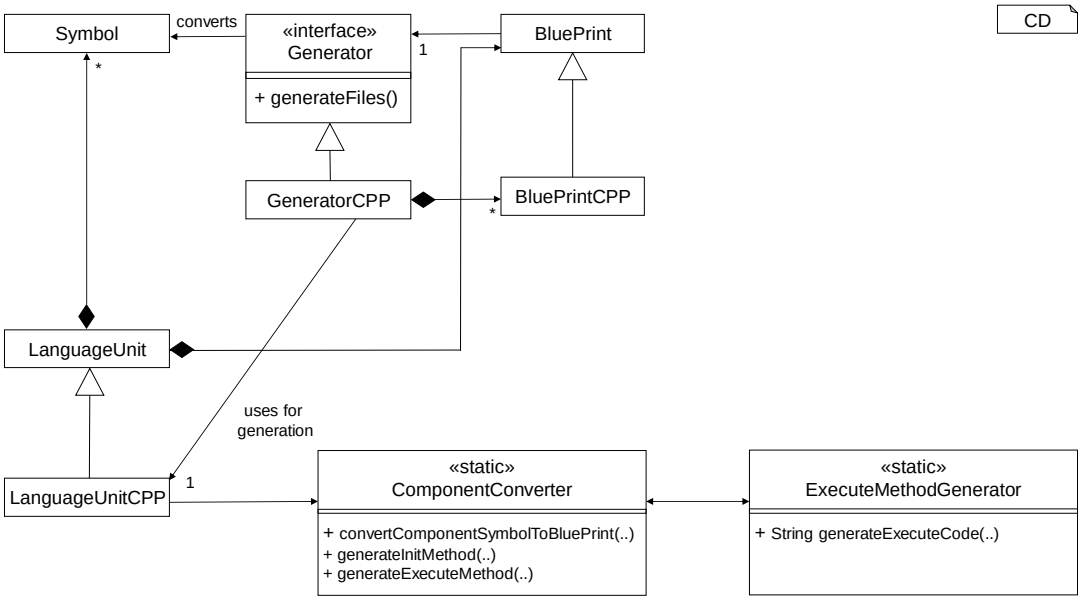


Figure 3.17: C++ Generator Architecture

Concluding this section, the modeling language family MontiCAR is designed for cyber-physical systems with focus on type and unit safety. This is implemented using base languages *Types*, *Ranges* and *NumberUnit*. Languages *MontiMath* and *EmbeddedMontiArc* should reuse those languages to ensure type safety. *EmbeddedMontiArcMath* combines MontiMath and EmbeddedMontiArc to have a full functional C&C modeling language.

3.2.3 Simulator

The MontiCAR [GKR+17] framework contains a simulator called MontiSim. MontiSim's purpose is to allow simulation of autonomous driving scenarios in a virtual environment [Ilo18]. It uses real world map data. The simulator provides a distributed simulation. It is hosted on a server and provides a graphical user interface via the web browser. The visualization of the simulation is done via WebGL and ThreeJS. Figure 3.18 shows the architecture of the simulator. It is zone based and can handle multiple clients. The simulation controller executes one simulation step per time frame. There can be multiple controllers, sensors and cars connected to the simulator.

3.2.4 EmbeddedMontiArcStudio

EmbeddedMontiArcStudio [EMA18] is a development suite for EmbeddedMontiArc and all its components. It combines the language framework MontiCAR with the simulator MontiSim. Further it provides an online IDE including syntax highlighting and sanity checks for EMAM models. It provides effective testing methods for EMAM models and offers the presented code generator.

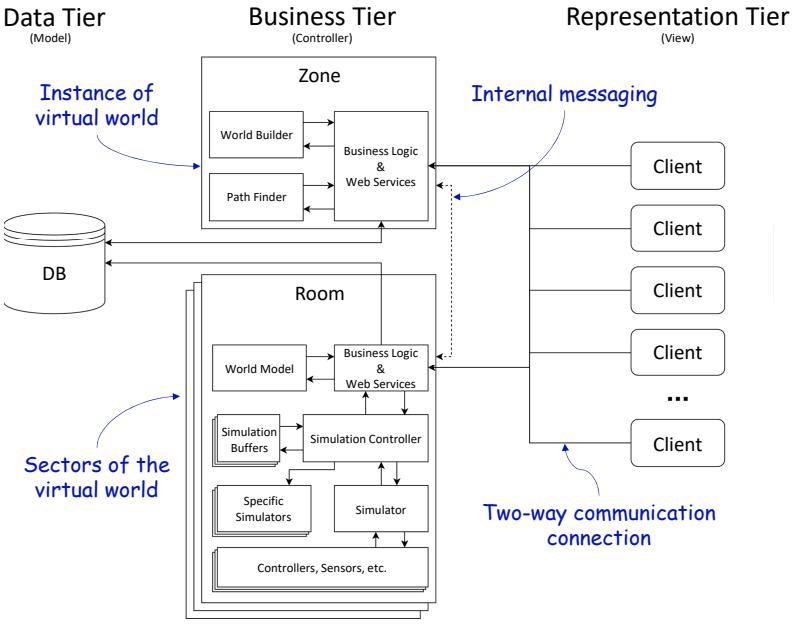


Figure 4.1: Architecture overview

Figure 3.18: MontiSim architecture

Chapter 4

Optimization Solver

Independent from the extension of MontiCAR optimization problems have to be solved. This is why a solver for optimization problems had to be chosen. In order to be integrated in MontiCAR and to solve MPC formulations, the solver has to fulfill certain requirements:

R1 Solve NLP. MPC formulations are nonlinear. To not being forced to linearize the objective function the solver should support NLP problems. Another advantage is that with NLP most optimization problems can be solved which enables to use the modeling language for a large set of applications.

R2 Solve problems fast. The trajectory controller will have to solve an optimization problem every few 100 ms to work in a vehicle for autonomous driving.

R3 Solver should be available under a public license, optionally open source. This assures that MontiCAR can be used by anyone, independent of a license for a solver.

R4 Provide a C++ interface. MontiCAR's code generator produces C++ code out of EMAM models. In order to reuse this generator it is required that the solver can be interfaced via C++.

A second solver has to be chosen to demonstrate the extendability of the developed solution. This solver does not necessarily have to fulfill requirements mentioned above, but might have complementary properties to cover a preferably large set of features.

In the subsequent section, multiple solvers are evaluated according to these requirements.

4.1 Comparison

In this section a set of solvers is introduced shortly. Table 4.1 lists a selection of solvers which might be relevant. A full list of reviewed solver can be found in the appendix in table A.1. The comparison of the solvers is based on [GAM18] and [CVX18]. The results are collected in this thesis.

Table 4.2 shows an overview of supported problems by the solver. In this table only LP, QP, NLP, DNLP, MIP, MIQCP and MINLP are listed. There exist many more problem classes and more solvers.

Solver	Vendor	Description
CPLEX 12.7	IBM ILOG	High-performance LP/MIP/QP solver
GUROBI 7.5	Gurobi Optimization	State of the Art Mathematical Programming Solver for LP/QP/MIP
IPOPT 3.12	COIN-OR Foundation	Interior Point Optimizer for large scale nonlinear programming
MOSEK 8	MOSEK ApS	Large scale LP/MIP plus conic and convex non-linear programming system
SeDuMi	J. Sturm, I. Polik	SeDuMi is a Matlab package for solving convex optimization problems involving linear equations and inequalities, second-order cone constraints, and semidefinite constraints (linear matrix inequalities).
SCIP 4.0	Zuse Institute Berlin et.al.	High-performance Constraint Integer Programming solver

Table 4.1: Short description of a selection of solvers [GAM18] [CVX18] [MS18].

Nevertheless, the most important problem class is NLP since it is listed in the requirements (R1). Not all the listed solvers provide a C++ interface (R4). For instance some are only accessible via Matlab.

Applying R3 only IPOPT remains as practically useful solver. Table 4.3 and table 4.4 show benchmark results form NLP and LP solvers on a large set of problems. The score on the right column is the scaled shifted geometric mean of runtime [Mit18a] [Mit18b]. This score is used to compare the runtime of the tested solvers. One is the best benchmark score, higher values are worse.

Compared to the other NLP solvers tested in table 4.3 IPOPT has a respectable score (5.08. Compared to Matlab's solver for NLP this is a real improvement. This benchmark result proves that IPOPT is also a good choice for performance (R2).

The result of the comparison is that IPOPT is used as solver for NLP. It fulfills all requirements from section 4. The second solver will be CPLEX. CPLEX is a solver for LP, QP, MIP and MIQP. It has the disadvantage that it is a proprietary solver. It advantages are a simple C++ interface and good documentation.

The next sections describe the chosen solvers IPOPT and CPLEX in detail.

	LP	QP	NLP	DNLP	MIP	MIQCP	MINLP
ALPHAECP						✓	✓
ALGLIB	✓	✓	✓	✓			
ANTIGONE 1.1		✓	✓	✓		✓	✓
BARON	✓	✓	✓	✓	✓	✓	✓
BDMLP	✓				✓		
BONMIN 1.8						✓	✓
CBC 2.9	✓				✓		
CONOPT 3	✓	✓	✓	✓			
CONOPT 4	✓	✓	✓	✓			
COUENNE 0.5		✓	✓	✓		✓	✓
CPLEX 12.7	✓	✓			✓	✓	
DECIS	✓						
DICOPT						✓	✓
GLOMIQO 2.3		✓				✓	
GUROBI 7.5	✓	✓			✓	✓	
GUSS	✓	✓	✓	✓	✓	✓	✓
IPOPT 3.12	✓	✓	✓	✓			
KESTREL	✓	✓	✓	✓	✓	✓	✓
KNITRO 10.2	✓	✓	✓	✓		✓	✓
LGO	✓	✓	✓	✓			
LINDO 11.0	✓	✓	✓	✓	✓	✓	✓
LINDOGLOBAL 11.0	✓	✓	✓	✓	✓	✓	✓
LOCALSOLVER 7.0		✓	✓	✓	✓	✓	✓
MIDACO			✓		✓		✓
MINOS	✓	✓	✓	✓			
MOSEK 8	✓	✓		✓	✓		
MSNLP		✓	✓	✓			
NLPEC							
OQNLP		✓	✓	✓		✓	✓
PATH							
SBB						✓	✓
SCIP 4.0		✓	✓	✓	✓	✓	✓
SNOPT	✓	✓	✓	✓			
SOLVEENGINE	✓				✓		
SOPLEX 3.0	✓						
XA	✓				✓		
XPRESS 31.01	✓				✓	✓	

Table 4.2: Overview solver with problem class [GAM18] [CVX18] [MS18].

Solver	scaled shifted geometric mean
IPOPT-3.12.10	5.08
KNITRO-11.0	1
LOQO-7.03	25.4
PENNON-0.9	21.1
SNOPT-7.7	35.0
CONOPT-3.17A	40.7
WORHP-1.11	5.67
XPRESS-8.5.1	1.16
FMINCON-2016a	17.8

Table 4.3: Benchmark NLP [Mit18a]

Solver	scaled shifted geometric mean
CPLEX-12.8.0	1.88
GUROBI-8.0.0	1
MOSEK-8.1.0.49	5.24
XPRESS-8.5.1	1
CLP-1.16.10	1.87
Google-GLOP	11.2
SOPLEX-3.1.0	18.1
LP_SOLVE-5.5.2	194
GLPK-4.64	45
MATLAB-R2018a	11.4
SAS-OR-14.3	5.61

Table 4.4: Benchmark LP [Mit18b]

4.2 IPOPT

IPOPT (Interior Point OPTimizer) is an open source solver for large-scale nonlinear optimization. It solves NLP problems of the form listed in table 3.1. IPOPT is part of the COIN-OR Foundation Inc. which aims to manage an open source project in operations research. COIN-OR is a non-profit educational foundation. Objective function and constraints can be linear, nonlinear, convex or non-convex. It helps if the functions are twice continuously differentiable [WLMK11].

IPOPT uses an interior-point line search filter method. This makes IPOPT suitable for large problems. Sparse and dense problems can be solved efficiently [Wäc09]. This method is a local optimization method, thus IPOPT may only find local optima and is sensitive to an initial point. The first and second order derivative is needed compute the solution of a problem. These can be computed using automatic differentiation methods such as ADOL-C [Wal09] or CppAD [Bel12].

This paragraph shortly presents the most important parts of the implemented algorithm based on [Wäc09]. IPOPT replaces inequality constraints by equality constraints and a new bounded slack variable.

$$\begin{array}{rlcl} \min\limits_{x\in\mathbb{R}^n} & f(x) & & \\ \text{s.t.} & c(x) & = & 0 \\ & x & \geq & 0 \end{array} \tag{4.1}$$

In the interior point (or barrier) method an auxiliary barrier problem formulation is considered:

$$\begin{array}{rlcl} \min\limits_{x\in\mathbb{R}^n} & \phi_\mu(x) = f(x) - \mu \sum_{i=1}^{n} ln(x_i) & & \\ \text{s.t.} & c(x) & = & 0 \end{array} \tag{4.2}$$

$\mu > 0$ is called barrier parameter, $\phi(x)$ is called barrier objective function. The overall strategy is to solve a sequence of barrier problems, with $\mu \to 0$ until a certain accuracy is reached. The first order optimally conditions for the barrier problem can be denoted as:

$$\begin{aligned} \nabla f(x) + \nabla c(x) y - z &= 0 \\ c(x) &= 0 \\ XZe - \mu e &= 0 \\ x, z &\geq 0 \end{aligned} \tag{4.3}$$

with $\mu = 0$, $y \in \mathbb{R}^m$ and $z \in \mathbb{R}^n$. X and Z are diagonal matrices with x and z as diagonal elements. e is the unity vector. Conditions 4.3 are fulfilled by minimal and maximal points and saddle points. To ensure a minimal point is found a Hessian regularization has to be applied. A more detailed description of the algorithm can be found in [Wäc09].

IPOPT provides a method to locally solve a NLP optimization problem. IPOPT already is used successfully in applications like the dynamic modeling and optimization platform JModelica.org, as solver in GAMS, for computation of nonlinear scalings for reaction diffusion systems, unsupervised learning for exemplar-based models, EMSO Process Simulator, solving inverse kinematics of a humanoid robot and many more [COI18]. This proves its suitability for the MontiCAR framework.

4.3 CPLEX

CPLEX, also called IBM ILOG CPLEX Optimizer is a solver for LP, QP, MIP and MIQP optimization problems developed by IBM. IBM commercially sells CPLEX. Academic trial licenses are available. CPLEX is optimized for network flow problems and takes advantage of this special structure to solve problems faster [Cor15]. It uses a C++ interface which is called *concert* providing programming libraries. CPLEX is also available for many other programming languages.

CPLEX implement different optimizers based on the simplex algorithm. Additionally, it implements primal-dual logarithmic barrier algorithms and a sifting algorithm [Cor16].

CPLEX is chosen as a second solver for the MontiCAR framework because it has complementary properties to IPOPT, which supports the concept of adding different solvers to the framework.

Chapter 5

EmbeddedMontiArcOpt

This chapter deals with the extension of EmbeddedMontiArcMath (see section 3.2.2). The goal is to extend EMAM such that it can model optimization problems. The code generator has to be extended in a way that it can generate code which is able to solve the optimization problem. This ability is needed for the trajectory controller. Since the trajectory controller should use MPC, it is required to model and solve MPC problems. For this the MontiCAR framework has to be used.

The first paragraph explains preparations for the new languages. Then the math language MontiMathOpt, capable of modeling optimization problems, is presented. MontiMathOpt then is used as behavior description for EmbeddedMontiArc models. This new language is called EmbeddedMontiArcMathOpt (EMAMOpt). Finally, a code generator which generates C++ code from EMAMOpt models is presented.

5.1 Preparation

MontiCore 5.0.0 Before EmbeddedMontiArcMath is extended, it is upgraded to the newest MontiCore version 5.0.0. This step is not mandatory but it ensures MontiCore's newest features are available and bugs are fixed.
For this upgrade it is also necessary to upgrade all parent languages of EmbeddedMontiArcMath (see section 3.2.2).
The MontiCore upgrade changes the following:

- Name of Java Optionals end with postfix *Opt*.
- Lists of nonterminals have the name postfix *List*.
- *WS* tokens are not passed to the parser.

Especially the last point has important consequences for the math language. Matrix definitions like `[1 -1; 1 - 1]` and `[1-1;1-1]` now become ambiguous. It either can be a 2×2 matrix containing numbers 1 and -1 or a column vector with two elements which are $1 - 1 = 0$. Before the upgrade a whitespace had separated two columns, now this is not possible anymore. For this reason in case of this ambiguity columns have to be separated by commas which was optional before.

Resolve name conflicts One problem in MontiCAR was that code snippets were copied and pasted in other languages. This lead to various naming conflicts in the composition of languages. To solve those conflicts, the copied nonterminals of languages had to be collected in the common language. All languages strictly had to use the common language components to be compatible to each other.

MontiCAR's common languages were copied from older MontiCore versions. Reasons were grammar modifications compared to the grammars provided by MontiCore. This complicates reusability. With version 5.0.0 these modifications can be done by extending and overwriting existing nonterminals.

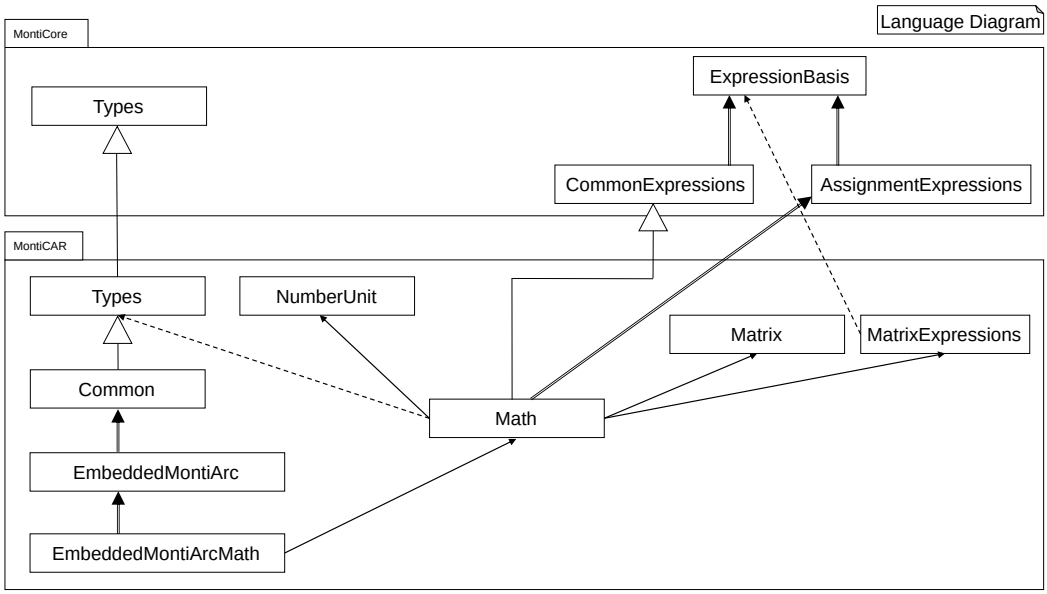


Figure 5.1: MontiCar language hierarchy refactored

Comparing figure 5.1 to figure 3.11 the changes between the two versions are visualized. MontiCAR's grammar *Types* (also called Types2 to avoid naming conflicts) extends language *Types* provided by MontiCore. MontiCAR's grammar *Commons* (also called Common2) extends language *Types2*. It cannot extend MontiCore's Common language because this language blocks the keyword *static* which is used in EMAM to declare static variables. In MontiMath static is used as matrix property and is not recognized if Common2 would extend Common.

Child languages *EmbeddedMontiArc, EmbeddedMontiArcBehaviour, EmbeddedMontiArcMath* as well as the generator for *EmbeddedMontiArcMath* are affected by these changes.

In addition to the usage of MontiCore's core grammars NumberUnit, ElementType and Range were existing in *Math* as well as in *Types2* and *Ranges*. NumberUnit also exists as independent component and was used by all languages in MontiCAR. As a consequence, NumberUnit is moved to languagescommon. ElementType was defined in Types2. Range was defined in Ranges.

Math In contrast to figure 3.11 the implementation of MontiMath did not reuse any common languages (for example Types or Ranges). Figure 5.2 shows the refactored lan-

guage hierachy of MontiMath. For the sake of simplicity elements which stay the same are not displayed. They stay similar to figure 3.13 and figure 3.14.

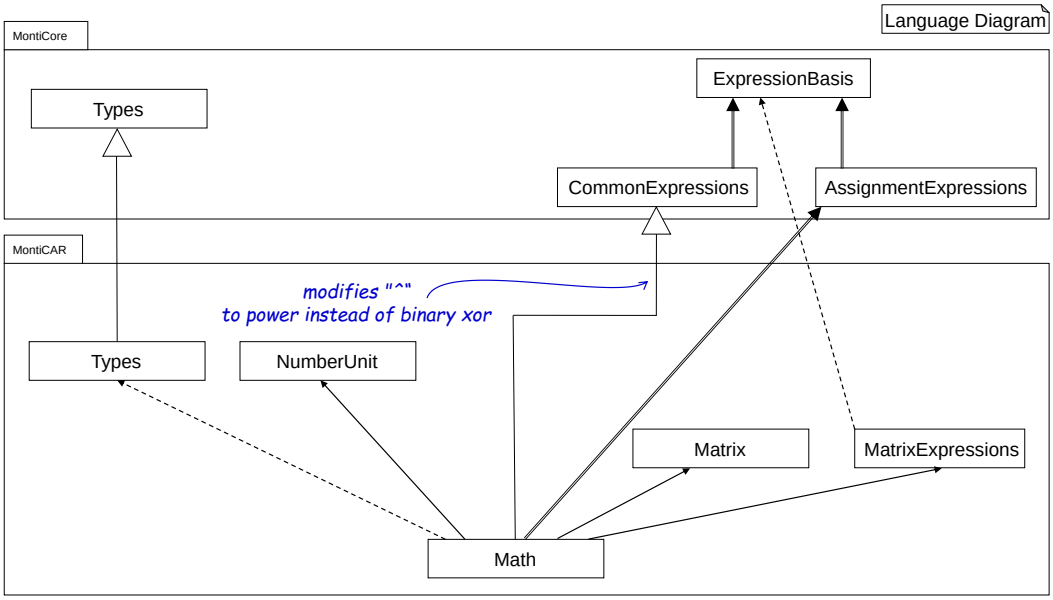


Figure 5.2: MontiMath: Language Hierarchy

Instead of defining NumberUnit by itself the common MontiCAR version is used. ElementType, Ranges and Dimensions are used from Types2. CommonExpressions contain basic arithmetic expressions such as addition, subtraction, multiplication and boolean expressions. For assignment expressions such as `+=`, `-=`, `*=`, `++`, etc. AssignmentExpressions is used. MatrixExpressions additionally contain matrix specific arithmetic operations like element-wise arithmetic operations. Matrix is used for matrix element access (for example `A`(i, j) to access the element in the i*th* row and j*th* column `A`$(:, 1)$ to access the whole first column).

MathSymbolTableCreator has to implement `setRealThis` for all parent languages because these methods are not implemented there.

EMAM2CPP The generator EMAM2CPP is in a very early state of development and contains several bugs which need to be fixed before extending it for EMAMOpt models to generate MPC controller. The following enhancements are made:

- Column vector generation: EMAM2CPP for every matrix data type generates *mat* which is Armadillo's data type for a matrix in C++. Vectors now are generated as type *rowvec* and *colvec* respectively.
- Cube support: MontiMath potentially supports matrices and tensors of unlimited dimensions. The generator only supported matrices with one or two dimensions. The backend Armadillo also is able to represent cubes (3 dimensional matrices). Thus, the class *TypeConverter* was modified such that it also generates type cube.
- Armadillo 0-based indexing

- Element-wise multiplication: The backend armadillo uses operator `%` for element-wise matrix multiplication. MontiMath uses `".*"`. This has to be adapted in class *ExecuteMethodGeneratorMatrixExpressionHandler*, which then asks *MathBackend* for the correct string.
- Sum support: The current state was that the command `sum(X)`, where `X` is a matrix or a vector, sum up all elements of its input using Armadillo's `accu` function. Armadillo and also MATLAB do support a sum command `sum(X, dim)` $dim \in \{1, 2\}$ which calculates the sum of all row vectors (if $dim = 1$ or not present) or the sum of all column vectors (if $dim = 2$) of a matrix. These commands were used directly in the implementation.

This list is not complete. Several other bugs are fixed beside this.

5.2 MontiMath Opt

To describe optimization problems MontiMath has to be extended. This section deals with the conceptional extension and their implementation. First the syntax has to be defined, the grammar from MontiMath has to be extended, a symbol table has to be created and then context conditions have to be defined. The goal is to extend MontiMath in a way that it can describe optimization problems close to their mathematical representation(see section 3.1.1).

5.2.1 Syntax

This subsection covers the choice of a syntax for optimization problems, a concrete example as introduction and the implementation in MontiCore.

Choice of Syntax As mentioned above the aim is to develop a syntax which is close to the mathematical one. To not start from scratch the following four optimization languages are reviewed:

1. CVXGEN [MB12b]
2. AMPL [FGK90]
3. GAMS [ER08]
4. MATLAB [MAT18b]

CVXGEN is a tool which allows modeling of convex problems with an integrated C code generator. The optimization problems are modeled in a high level language. CVXGEN generates optimized code which is not dependent on libraries. The efficient implementation allows embedding to solver into real time applications [MB12b].

AMPL(A Mathematical Programming Language) is a modeling language for math problems which aims to express problems close to their mathematical notation. AMPL is

maintained by AMPL Optimization Inc, originally developed by Robert Fourer, David M. Gay, and Brian W. Kernighan at Bell Laboratories. It is optimized to express large scale optimization problems and support the whole life cycle of formulation, testing, deployment and maintenance of optimization problems. Optimization data, variables, objective function and constraints can be modeled. AMPL supports many solvers including CPLEX, Gurobi, Xpress and IPOPT [FGK90].

GAMS (General Algebraic Modeling System) is a high-level modeling language for math and optimization. GAMS is optimized for complex, large scale modeling and optimization applications. The model and used data can be used independent from each other to create reusable models. More than 25 different solvers are supported [ER08]. Compared to AMPL, GAMS allows a modeling closer to the use case problem, but not as close to the mathematical notation as AMPL or CVXGEN.

Matlab is a programming language for mathematical problems. In contrast to AMPL, GAMS and CVXGEN it is not specialized on optimization problems, but more on the numeric solution of problems using matrices and vectors. Matlab allows code generation of C and C++ code optimized for the execution on embedded platforms. The modeling language Simulink is widely used for engineering tasks [TM18]. Nevertheless, Matlab has own solvers for optimization problems of different classes. Also third party solvers can be used by the integrated usage of C, C++ or Fortran binaries using MEX-files.

Table 5.1 shows an example for the syntax of the different languages.

In CVXGEN parameters and variables have to be declared in an extra block. Then the keyword *minimize* or *maximize* defines the optimization type followed by the objective function. The constraints are declared after the *subject to* keyword. The optimization statement is concluded with the keyword *end*.

AMPL also first defines the optimization variable with the keyword *var*. The objective function declaration also determines whether minimization or maximization is performed. Each constraint gets an identifier and is declared using the keyword *subject to*. Optimization is performed using the command *solve* at the end.

In GAMS variables are declared similar to CVXGEN and AMPL using the keyword *Variable*. Similar to AMPL the constraints on the optimization variable are declared here. The difference to AMPL is that at this position also an initial value for the iteration is defined. AMPL does this separately. CVXGEN defines the constraints on x in the subject to block. An initial value does not have to be specified. GAMS uses keyword *Equation* for the objective function definition as well as for equality and inequality constraints. To solve a problem the keyword *solve* followed by the optimization type (for example *minimizing*), the objective function and a solver which should be used.

The syntax of *MATLAB* is completely different to the other optimization languages. In MATLAB the optimization problem is solved using a function call. For linear objective functions parameters A, b and the optimization variable x is defined. The nonlinear constraints are defined in the argument *nonlcon*. If the optimization problem is nonlinear an object *problem* is passed as argument to the function *fmincon*. In MATLAB *problem.objective* defines the objective function. Further details can be found in table 5.1.

MontiMathOpt tries to include all advantages of these presented languages. It will be based on the CVXGEN syntax because it is close to the mathematical syntax. In contrast to the other languages MontiMathOpt will define the optimization variable in brackets after the

keyword *minimize* or (*maximize*). MontiMath already provides a type system, and range constraints that will be included. Thus, the variable definition becomes shorter compared to other languages. In contrast to all other languages the optimization expression will deliver a return value which directly returns the objective value. Solver parameters (for example starting value, display parameters, etc.) will not be included in the syntax.

Example To clarify how the syntax can look like listing 5.1 shows a short example how an optimization statement could look like in MontMathOpt:

```
1 Q objVal = minimize(Q x)
2   x^2;
3 subject to
4   -1 <= x <= 1;
5   x^2 <= 1;
6 end;
```

Listing 5.1: MontiMathOpt grammar.

It contains a return value (objective value), the keyword *minimize* to indicate a minimization, an optimization variable declaration, most important the objective function, and two constraints in the subject to section. How the concrete and abstract syntax is defined will be shown in the next section.

Grammar in MontiCore MontiMathOpt extends the grammar of MontiMath. Using MontiCore it is easily possible to extend MontiMath's *math.mc4* using the language composition mechanism extension [Völ11].

mathopt.mc4 is the grammar file for MontiMathOpt (see listing 5.2). The key nonterminal in this grammar is *OptimizationStatement*. Since this is the only nonterminal of this grammar, only this statement can be used in a math script. The other defined nonterminals are used within this expression.

The OptimizationStatement starts with an optional return value, which can be either an existing scalar variable or a new defined one. Then the keyword *minimize* or *minimize* indicates what kind of optimization will be calculated. After that, the optimization variable is declared in brackets. Here one can either use an existing variable or a newly defined one. Note that if the variable already exists, its value will be overwritten by the optimal value. If the variable already exists, values dependent on the optimization variable are substituted in the optimization statement by the previous expressions. That simplifies the notation of the objective function if it contains function. Unfortunately MontiMath does not allow user defined functions, thus this substitution has to be introduced. Next nonterminal is the objective function of the optimization problem. This defines a function dependent on the optimization variable which should be minimized or maximized. It is an *Expression* (see MontiMath section 3.2.2) which returns a scalar value. Next is the keyword *subject to* which is followed by a list of arbitrary many OptimizationConditions. The keyword *end* indicates the end of the optimization statement. Lines $14 - 26$ further specify what exactly an OptimizationCondition is. *OptimizationSimpleCondition* is the simplest one: A boolean compare expression restricted to operators <=, == and >=. Operators $<$ and $>$ are not allowed by construction of the language because they do not make sense for infinite small numbers (which number $x \in \mathbb{R}$ is bigger than 1?). *OptimizationBoundsCondition* (also called box constraint) is inspired by the mathematical formulation

CVXGEN

```
parameters
  A (3,10)
  b (3)
  Q (10,10) psd  # quadratic penalty.
  c (10)  # linear cost term.
end

variables
  x (10)
end

minimize
  quad(x, Q) + c'*x
subject to
  A*x == b
  0 <= x <= 1  # box constraint on x.
end
```

AMPL

```
# Definition of the variables with bounds
var x {i in 1..4}, >= 1, <= 5;

# objective function
minimize obj:  x[1]*x[4]*(x[1] + x[2] + x[3]) + x[3];

# and the constraints
subject to c1:  x[1]*x[2]*x[3]*x[4] >= 25;

subject to c2:  x[1]^2+x[2]^2+x[3]^2+x[4]^2 = 40;

# Now we set the starting point:
let x[1] := 1;
let x[2] := 5;
let x[3] := 5;
let x[4] := 1;

# Solve the optimization problem
solve;
```

GAMS

```
Variable x1, x2, obj;
x1.lo = -10; x2.lo = -15; // lower bounds
x1.l = 8; x2.l = -14; // initial value
x1.up = 20; x2.up = 20; // upper bounds

Equation objdef, eqs, ineqs;

objdef.. obj =e= 10*sqr(sqr(x1) - x2) + sqr(x1 - 1);
eqs.. x1 =e= x1*x2;
ineqs.. 3*x1 + 4*x2 =l= 25;

Models m / all /;
* x1.l = 1; x2.l = 1; // optimal values

solve m minimizing obj using nlp;
```

MATLAB

```
% linear objective function
x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)

% non linear objective function
options =
optimoptions('fmincon','Display','iter','Algorithm','sqp');
problem.options = options;
problem.solver = 'fmincon';
problem.objective = @(x)100*(x(2)-x(1)^2)^2 + (1-x(1))^2;
problem.x0 = [0,0];

function [c,ceq] = unitdisk(x)
c = x(1)^2 + x(2)^2 - 1;
ceq = [ ];
problem.nonlcon = @unitdisk;

x = fmincon(problem)
```

Table 5.1: Comparison optimization languages syntax

```
1  package de.monticore.lang;
2
3  grammar MathOpt extends de.monticore.lang.Math
4  {
5      OptimizationStatement implements Statement =
6      // return value
7      (ObjectiveValue =)?
8      ("minimize" | "maximize") "(" Type? OptimizationVar ")"
9        objectiveFunction:Expression ";"
10     "subject to"
11       constraint:OptimizationCondition*
12     "end";
13
14   OptimizationCondition = (
15     OptimizationSimpleCondition |
16     OptimizationBoundsCondition |
17     OptimizationForLoopCondition);
18
19   OptimizationSimpleCondition =
20     left:Expression ("<=" | ">=" | "==") right:Expression ";";
21
22   OptimizationBoundsCondition =
23     lower:Expression "<=" expr:Expression "<=" upper:Expression ";";
24
25   OptimizationForLoop =
26         "for" head:MathForLoopHead body:OptimizationCondition+ "end"
              ;
27 }
```

Listing 5.2: MontiMathOpt grammar.

of an optimization problem as well as the technical requirement of the solver. It defines a lower bound, the bounded expression and an upper bound for the value. These three values are separated by <=. Often a third kind of constraint is need where a variable is iterated: For example $x(i) \leq 0 \forall i \in [1..n-1]$. Other optimization languages solve this problem by introducing a new keyword (for example *forall*). In MontiMathOpt reuses an already existing concept: For loops already provide a possibility to iterate over a loop variable. Thus, this concept is extended to OptimizationConditions.

Note: It would also be possible to omit the *Objective Value* in the grammar and make the rest an expression. Then the statement in listing 5.1 would be an *AssignmentStatement*. Reasons against this implementation were that the MontiMathOpt models become less natural to read if for example an optimization expression is used inside an if condition or for loop header. To prevent this kind of models the language does not allow such constructs. However, this is not a limitation to the language because the objective value can be stored in a variable which then can be used. This additionally has the advantage that it enforces the objective value to be scalar by restricting the return type.

A challenge for MontiMathOpt is to include all possible constraints. Obviously the explicit constraints given in the subject to block have to be respected, but also MontiMath's range restrictions. The example in listing 5.3. It shows that that equality, inequality and box constraints as well as loop constraints are supported explicitly. Additionally, constraints on the objective value and the objective variable have to be respected. Especially in code generation all of those constraints have to be respected and passed to the solver.

```
Z n = 3;
Q(-2:5) y = minimize(Q(-1:1)^{3,3}  x)
    sum(x + x + x + x + x);
subject to
    (-3) <= (5 * x) <= (1);
    x(1,1) == 0;
    0.5 <= x(2,1);
    x(:,n) >= [1/4;1/4;1/4];
    for i = 1:n
        x(i, 2) == 0.01;
    end
end
```

Listing 5.3: Possible Contraints in MontiMathOpt

The next section focuses on the technical implementation of symbol table and language composition.

5.2.2 Language Composition and Symbol Table

As mentioned in section 3.2.1 MontiCore provides mechanisms to easily compose languages. In this case MontiMathOpt extends MontiMath. Thus, all of MontiMath's properties are adopted to MontiMathOpt (see figure 5.3).

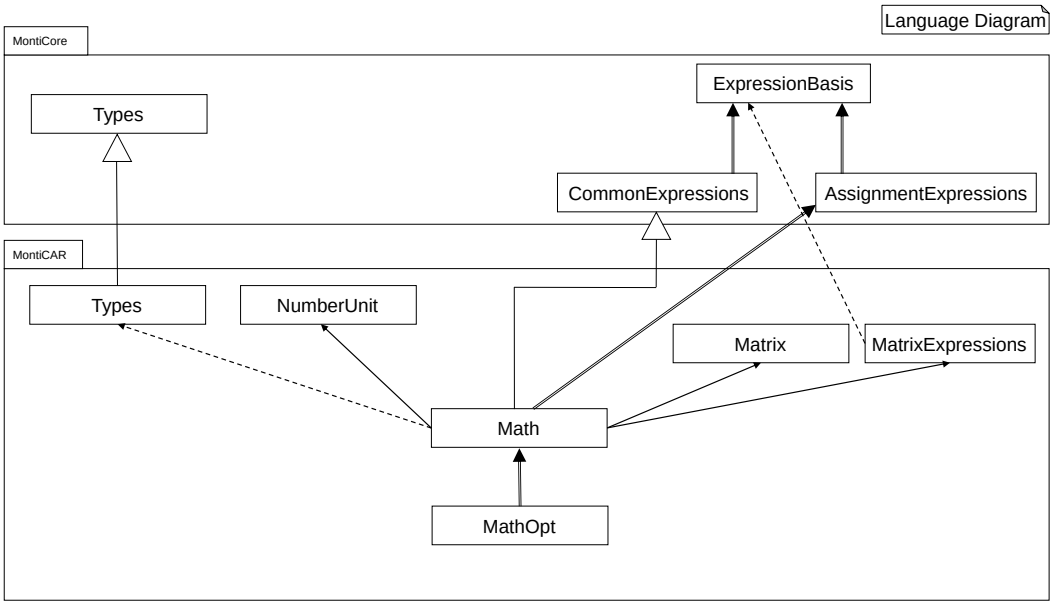


Figure 5.3: MontiMathOpt: Language Hierarchy

Using MontiCore parser and abstract syntax tree are generated automatically for the new language. The new symbol table is handwritten and not generated by MontiCore. Thus, a class *MathOptSymbolTableCreator* has to be created for MontiMathOpt. Following the principle of language extension [Naz17] *MathOptSymbolTableCreator* directly inherits from *MathSymbolTableCreator*.

MontiMathOpt exports the symbols *MathOptimizationStatementSymbol* and *MathOptimizationConditionSymbol* (see figure 5.4). The key symbol is *MathOptimizationStatementSymbol* which provides access to all relevant information of the optimization statement.

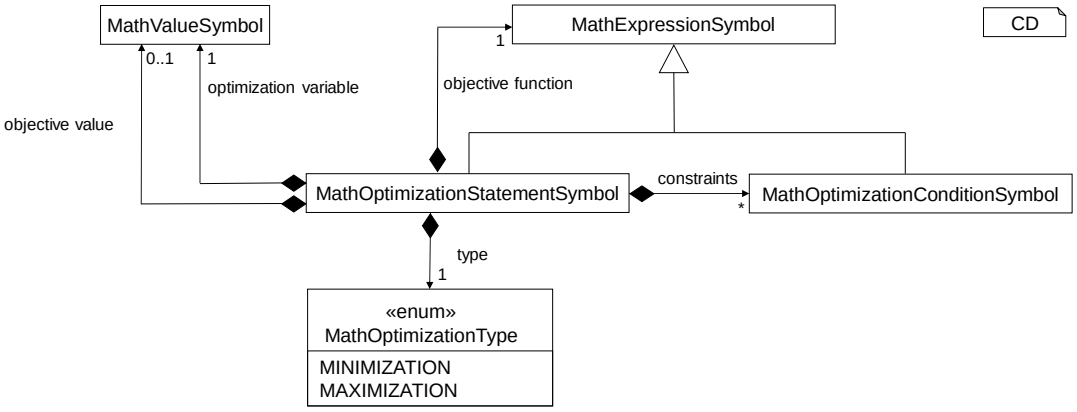


Figure 5.4: MontiMathOpt: Symbols

5.2.3 Context Conditions

By construction of MontiMathOpt's grammar many conditions are already implemented. Additionally, to the CoCo's defined in MontiMath the following CoCo's have to be defined for consistent usage of MontiMathOpt:

- **0xC0001** (error) Objective function has to return a scalar value.
- **0xC0002** (error) Return variable has to be scalar.
- **0xC0003** (error) Optimization variable has to be defined correctly.
- **0xC0004** (warning) The lower bound of a box constraint has to be smaller or equal the upper bound.

The CoCos are implemented using MontiCore's infrastructure for context conditions (see 3.2.1) displayed in figure 5.5. Two classes *OptimizationStatementCoCo* and *OptimizationConditionCoCo* are implemented using the generated interface provided by MontiCore. The corresponding *check* functions call sub-functions which each implements one context condition.

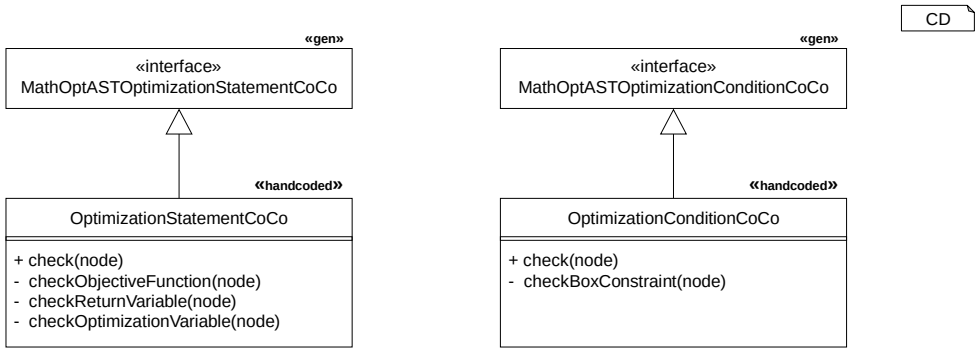


Figure 5.5: Class Diagram: CoCo Implementation using MontiCore

5.2.4 Test

To assure correct functionality unit tests are implemented. The following components are tested:

- Parser
- AST
- SymbolTableCreator
- Symbols
- Context Conditions

Unit tests for the parser and the symbol table creator are reused form MontiMath. The test classes are extended to also check the newly added functionality. Therefore, some of MontiMath's test classes has to be adapted such that the parser and the language is available in sub classes using *protected* visibility. Java class *ParserMathOptTest* checks if a set of test models can be parsed. The set includes different configurations of possible MathOpt models including minimization and maximization, with and without return value, equality constraints, inequality constraints, box constraints, for loop constraints, optimization of scalar values, vectors and matrices and models of different problem classes LP and NLP.

Tests on the AST are realized in class *MathOptASTTest*. In contrast to the parser test in the AST only single ASTNodes are checked. The first test is always if the ASTNode can be parsed. The correct assignment of AST properties is verified. Tested nodes are the AST Nodes generated from the grammar (see listing 5.2) including the OptimizationStatement, return values, variable definition and the different types of constraints.

Tests on symbol table creator and symbols are tested against the set of MathOpt models. The correct creation and linking from symbols to the AST nodes are verified, as well as the assigned properties and functions of the symbols. For symbols *MathOptimizationStatement* and *MathOptimizationCondition* all functions were tested to archive a high statement coverage.
Finally, the context conditions are tested using positive and negative examples from AST nodes.

5.3 EmbeddedMontiArcMathOpt

MontiMathOpt provides a way to describe optimization problems within math scripts. To model MPC controller in a C&C architecture MontiMathOpt has to be embedded into EmbeddedMontiArcMath. This newly composed language is called *EmbeddedMontiMathOpt* (EMAMOpt). Again MontiCore provides mechanisms to implement this easily. EmbeddedMontiArcMathOpt extends EmbeddedMontiArcMath and embeds MontiMathOpt (see figure 5.6).

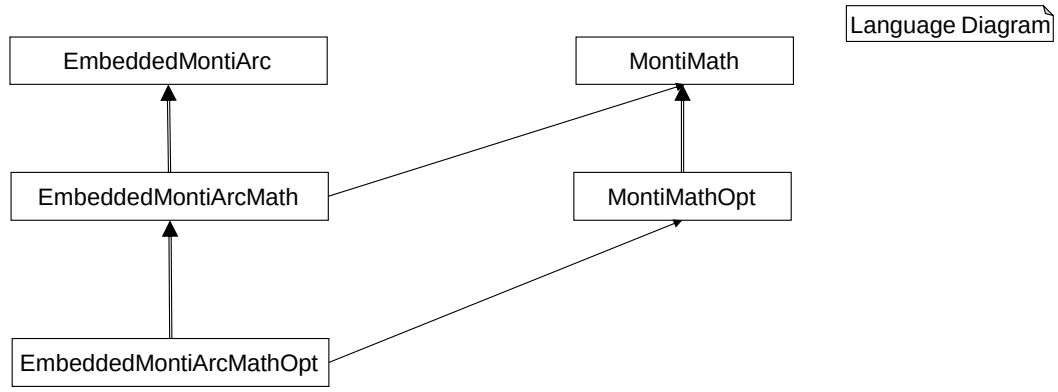


Figure 5.6: EmbeddedMontiArcMathOpt: Language Hierarchy

The key part in EMAMOpt is to create the symbol table correctly. Because this language uses multiple languages the symboltable creator cannot simply be overwritten, but a delegate visitor had to be used [Naz17]. The class diagram in figure 5.7 shows the implementation of *EmbeddedMontiArcMatOptSymbolTableCreator*.

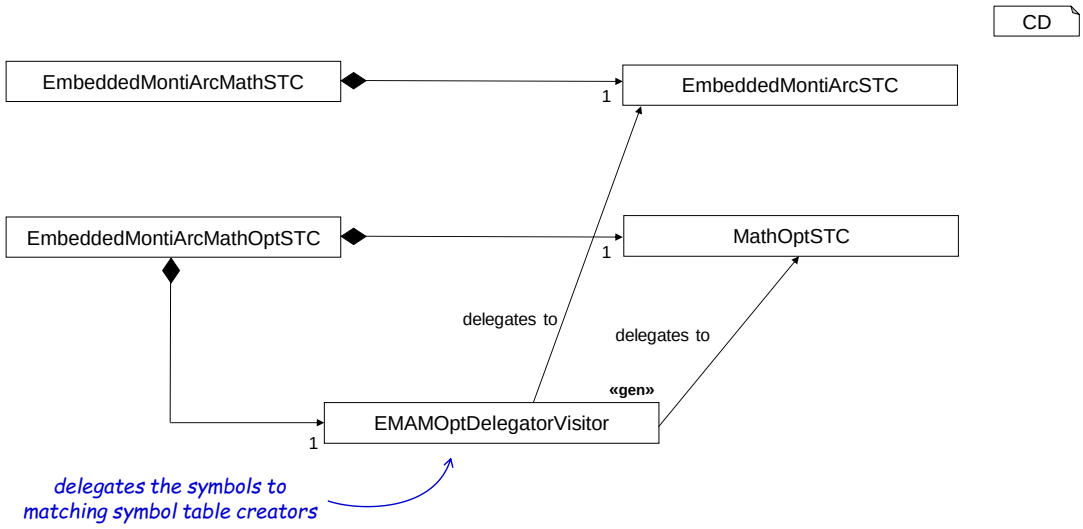


Figure 5.7: EmbeddedMontiArcMathOpt: SymbolTableCreator

EMAMOptDelegatorVisitor is generated by MontiCore. It is used in EmbeddedMontiArcMatOptSymbolTableCreator to delegate symbols from the different languages to a symbol table creator which can handle those symbols.

No more adoptions has the be made, the rest basically stays the same as in EMAM. EMAMOpt models have the same structure as EMAM models. The file name stays the same. EMAMOpt now is able to model an MPC controller. The next section will present a solution to generate C++ code from the EMAMOpt model.

5.4 EmbeddedMontiArcMathOpt to C++ Generator

To generate executable code EmbeddedMontiArcMathOpt models have to be converted to C++ code. EMAM already provides a working C++ code generator (see section 3.2.2). This generator is extended to assure compatibility to EMAM models. The following paragraphs will discuss how the C++ code generator for EmbeddedMontiArcMathOpt (*EMAMOpt2CPP*) is built in general, how the solver code for optimization problems is generated and how CMake files are generated.

5.4.1 Requirements

Before presenting the implemented solution, a few requirements of the code generator will be discussed. R1 and R2 obviously are the most important requirements because otherwise EMAMOpt models can never be generated. Of course also EMAM models should be generated using the EMAMOpt2CPP generator (R3).

R1 Generator must be able to produce code which solves optimization problems.

R2 Generator must be able to handle MontiMathOpt symbols.

R3 Generator must be able to also generate C++ code for EMAM models.

R4 Generator must be able to overwrite code generation for MontiMath and EmbeddedMontiArc symbols.

R5 Generator should support multiple solvers for optimization problems.

R6 Generator should be able to decide for a solver according to the optimization problem class.

R7 Generator should also be able to generate CMake files.

Similar to for example GAMS [ER08] multiple solvers for optimization problems should be supported (R5). The choice of the solver should depend on the problem class (R6). Because compilation and linking of C++ code is not an easy task CMake files should be generated to collect all dependencies on the math backend and the different solvers (R7).

5.4.2 Generator Architecture

EMAM2CPP Extension

The first step to create a new C++ generator for EMAMOpt models is to extend the existing generator EMAM2CPP. The class diagram in figure 5.8 shows the inheritance of the new main Java class *GeneratorEMAMOpt2CPP*. One problem is that EMAM2CPP is not designed to be expendable. It used many static classes which are globally available in the project (see figure 3.17). Especially the static components *ExecuteMethodGenerator* and *MathFunctionFixer* cannot be extended which is a problem because they handle the generation of MontiMath symbols. The extended generator additionally has to handle MontiMathOpt symbols.

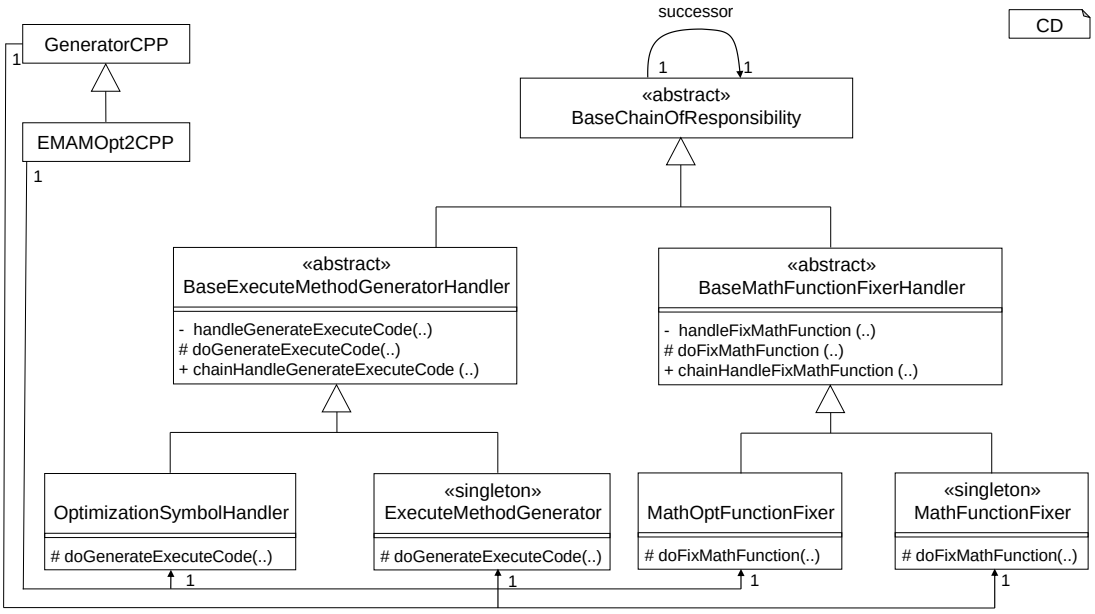


Figure 5.8: EmbeddedMontiArcMathOpt to C++ Generator Class Diagram

This problem is solved by applying the software design patterns *Singleton* and *Chain-of-Responsibilities* [GHJV95]. The singleton pattern ensures that former static methods can be overwritten. The Chain-of-Responsibilities-Pattern helps to replace and extend the huge if then else block which selects the matching function for the math symbols in ExecuteMethodGenerator dynamically. Now in case one math symbol cannot be handled a successor can be asked. The UML object diagram in figure 5.9 shows the implementation of the chain. The object of class OptimizationSymbolHandler handles all new MathOpt symbols. If it cannot handle a symbol they are forwarded to ExecuteMethodGenerator. This order has the advantage that OptimizationSymbolHandler also can modify the generation of EMAM symbols. The construction of the Chain-of-Responsibilities-Pattern also would allow the reverse order if modifying existing behavior is not desired. An alternative to the Chain-of-Responsibilities-Pattern would have been the Decorator pattern [GHJV95] or simple delegation. A decorator would have been harder to implement in the static structure. Delegation would be less dynamic than the implemented solution.

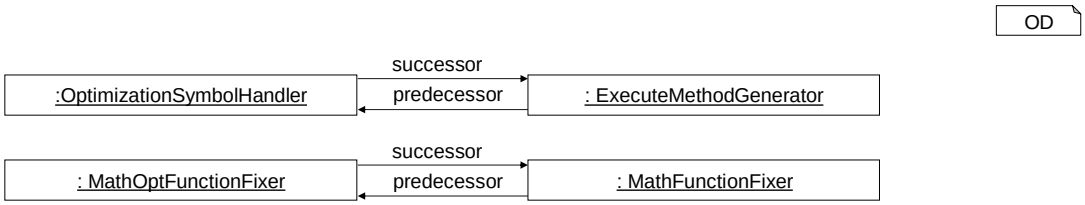


Figure 5.9: EmbeddedMontiArcMathOpt to C++ Generator Chain of Responsibilities

This implementation provides a basic framework to extend the EMAM2CPP generator. Requirements R3 and R4 are covered by this implementation.

Problem Structure

Requirement R6 states that the generator has to choose the right solver for the right problem class. Since MontiMathOpt does not provide information about the problem class, the generator has to determine the problem class. Using object oriented programming makes it easy to model the problem classes as Java classes. Due to inheritance a solver for a more complex problem class can automatically solve problems of the less complex child classes.

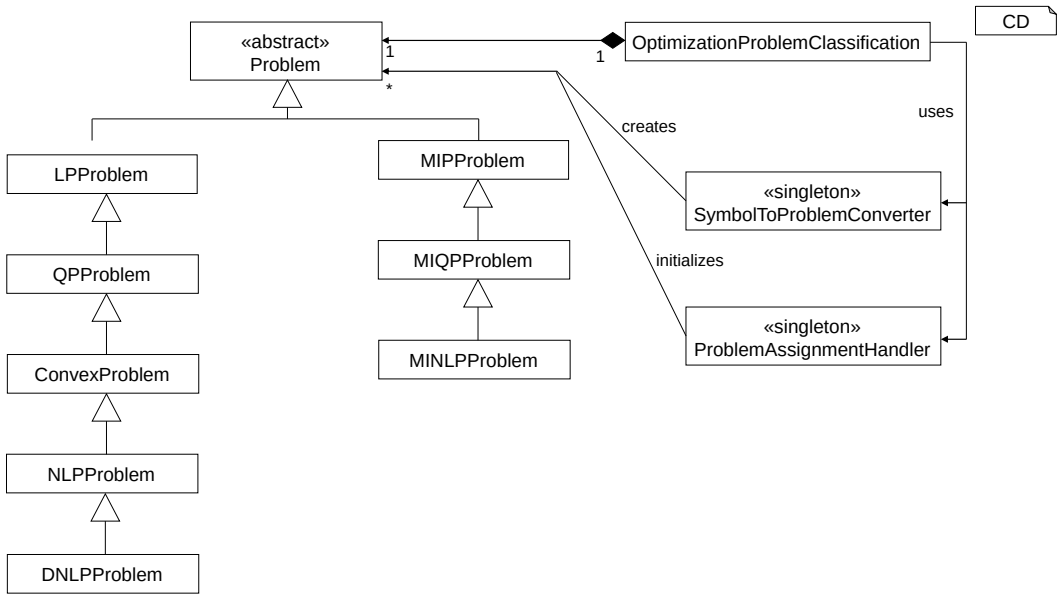


Figure 5.10: Problem Class Diagram

Figure 5.10 shows the inheritance structure of optimization problem classes. Class *OptimizationProblemClassification* classifies an object of class MathOptimizationStatement into one of the problems. Class symbol to problem type then creates a problem instance from the symbol.

Solver Structure

To implement different solvers (R5) the bridge pattern [GHJV95] is used. The bridge pattern allows dynamic switching of the implementation during run time. Figure 5.11 shows a class diagram of the bridge pattern implementation.

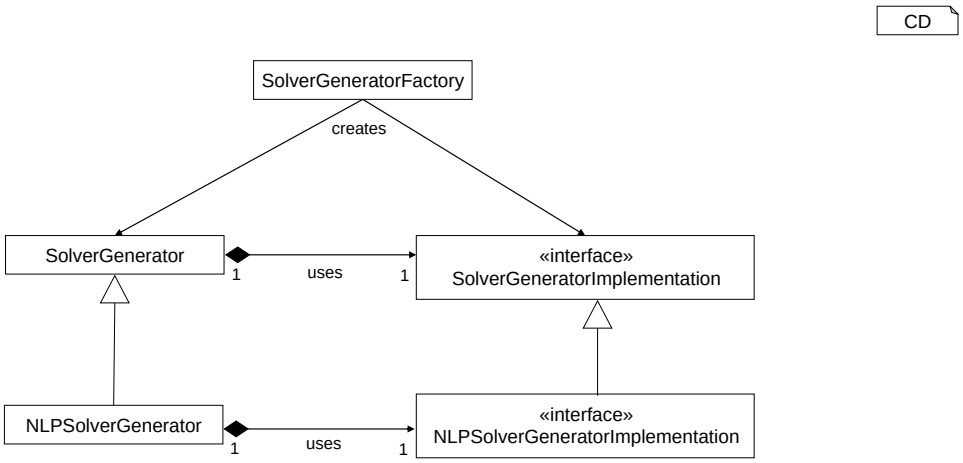


Figure 5.11: Solver Class Diagram

The abstract base class for solver code generation is *SolverGenerator*. This class aggregates an interface *SolverGeneratorImplementation*. SolverGeneratorImplementation can be implemented by any concrete solver generator like for example IPOPT or CPLEX. To assure that a solver is only used for its corresponding problem, SolverGenerator and SolverGeneratorImplementation are sub classed by for example NLPSolverGenerator and its implementation respectively.

Solver Options Different solver can have several options which can be passed to the solver. On the other hand MontiMathOpt does not specify any options because only the language models the problem, but does not provide any configuration properties. These properties can be passed to the generator. This enables a decoupled modeling of the problem and choosing a solver for the problem. Solver Options are stored in a class SolverOptions. To be as generic as possible this class is realized as a hash map which maps the option name to a value, both as string.

5.4.3 Solver

Section 5.4.2 shows the creation of a framework to add custom generators which generate code for those solvers. As a proof of concept two different solver generators were implemented. The first supported solver code is IPOPT. IPOPT is a solver for nonlinear programming problems and needed to solve the MPC problem for the trajectory controller

(see also section 4.2). As an example MIP and QP CPLEX (see section 4.3) code is generated. The next two paragraphs show how the code generators for the two solvers are designed.

IPOPT

The solver generator for IPOPT is realized using the previously implemented SolverGenerator following the bridge pattern. As key implementation entry the class *IpoptSolverGeneratorImplementation* is created. This class is the implementation part of the bridge pattern. IpoptSolverGeneratorImplementation uses a template based generation approach. It uses the existing freemarker template infrastructure provided by EMAM2CPP (see class diagram in figure 5.12).

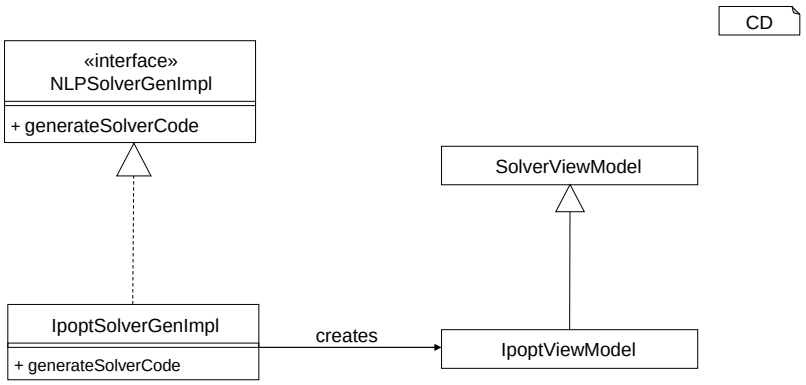


Figure 5.12: IpoptSolverGeneratorImplementation Class Diagram

The EMAMOpt component is generated the same way as already explained in section 3.2.2. The component uses a generated class *CallIpopt* to solve the optimization problem (see figure 5.13). CallIpopt is generated for every optimization problem individually. It uses the solver IPOPT.

IPOPT requires the first and second order derivatives of the objective function. To avoid adding these manually IPOPT is interfaced via CppAD [Bel12]. CppAD is a software framework for algorithmic differentiation. It provides an interface to IPOPT such that the declaration of the derivatives is not necessary, but is computed by CppAD. CppAD requires that the optimization variable is declared as active variable. This type will be abbreviated by *typedef CppAD::AD<double> adouble*. The type *adouble* overloads all common operators like $+, *, sin$ etc. The question is how the IPOPT interface [Wäc09] can be combined with the Armadillo interface.

Therefore, the generated C++ code contains a class *ADMat* which glues Armadillo and CppAD together. The class diagram shown in figure 5.13 describes how the combination of CppAD and Armadillo is implemented. Unfortunately Armadillo does only support base types for matrices *float*, *double*, *std::complex<float>*, *std::complex<double>*, *short*, *int*, *long*, and unsigned versions of *short*, *int* and *long* [ARM18]. This type list cannot be extended without modifying the Armadillo library. Type *field* is a matrix like structure

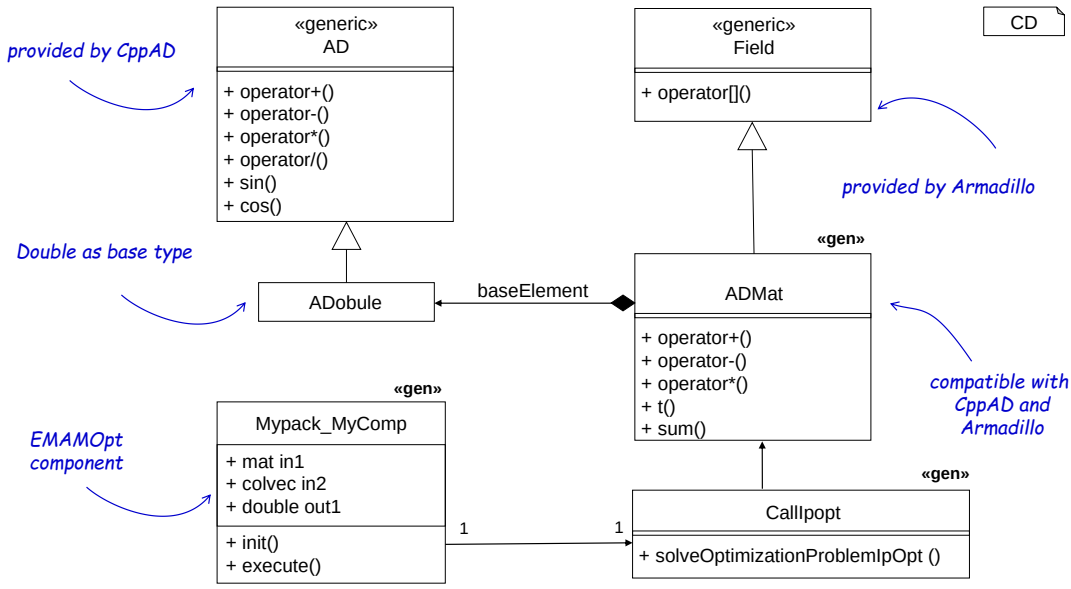


Figure 5.13: Generated C++ for IPOPT Solver

which can contain arbitrary data types. Field has the disadvantage that arithmetic operations are not supported. Class *ADMat* extends field containing data elements of type adouble. Operators had to be overwritten by hand. The class diagram in figure 5.13 lists all currently supported operations by ADMat.

IPOPT can be configured using several options [WB06]. These are passed to the IpoptView-Model via the class SolverOptions. IPOPT only supports minimization of a problem. To perform a maximization the objective function is multiplied by minus one.

All classes presented in figure 5.13 are realized in C++ as header only classes. This has the advantage that building and linking the C++ code is simplified and is often used for C++ libraries.

CPLEX

As a second solver CPLEX is chosen. It can solve quadratic and mixed integer problems (see 4.3). The main purpose for the integration of CPLEX is to demonstrate how another solver can be integrated by the generator. Cplex C++ interface is divided into two parts: One for modeling the problem and one for solving the problem [Cor15].

The CPLEX solver code generator is integrated in the created solver generator framework. In contrast to IPOPT the interfaces for QP and MIQP are implemented. The view model additionally contains reserved variable names and commands needed in the CPLEX interface (see figure 5.14).

Type *IloNumExprArg* is the result of an expression containing the optimization variable of type *IloNumVar*. Similar to the IPOPT interface from the last section 5.4.3 these types have to be glued to Armadillo expressions. Existing templates for IPOPT were reused for the CPLEX solver generator (see figure 5.15).

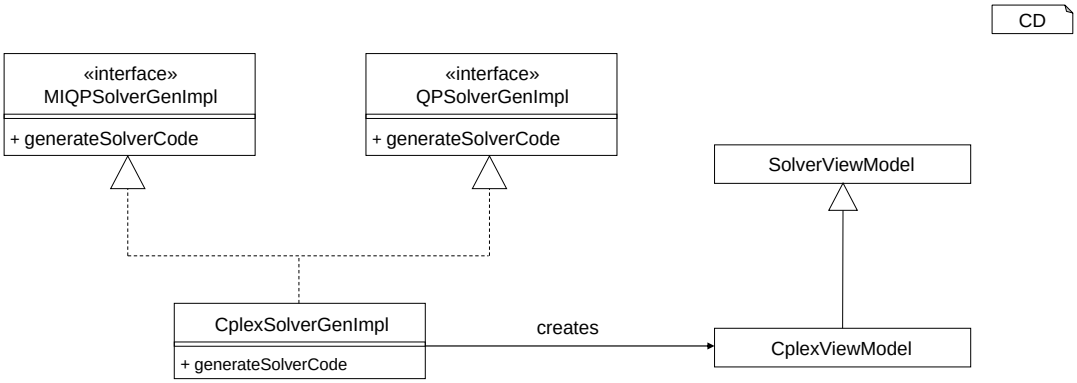


Figure 5.14: CplexSolverGeneratorImplementation Class Diagram

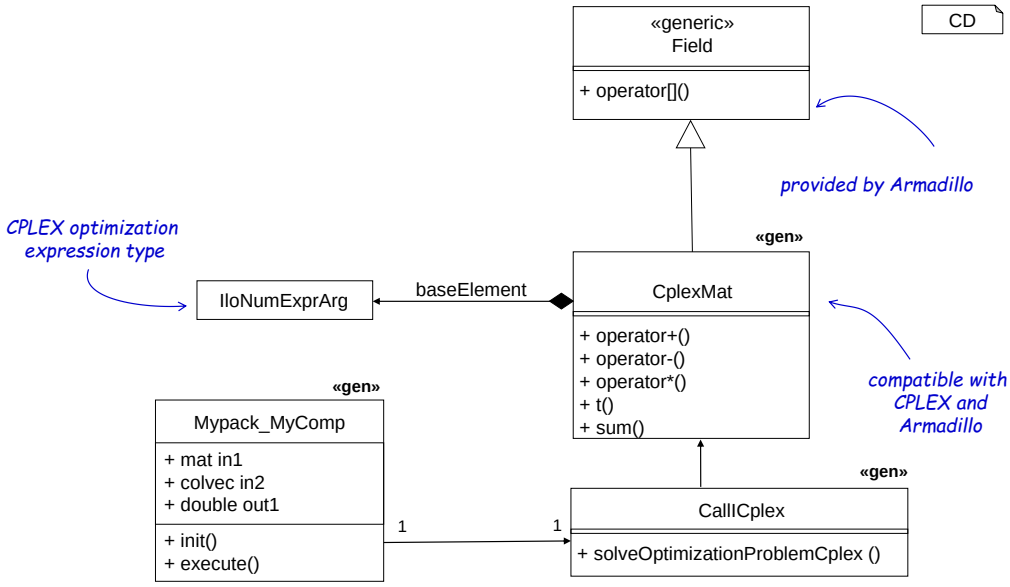


Figure 5.15: Generated C++ for CPLEX Solver

5.4.4 CMake Generation

A problem with the generated C++ code is that its not clear how to build the generated code. For compilation and building it might be required to pass additional include directories or link libraries that are not generated but the resulting code relies on. Especially the in this work presented dependencies on optimization frameworks like IPOPT or CPLEX are a difficult challenge. A common used solution for this kind of problem is the build tool CMake [CMa18]. CMake is an open source cross platform build tool that generates native makefiles for every platform [MH15]. The generation of CMake files (R7) would add three benefits to the EMAM to C++ code generator:

1. One single execution point to build the code.

2. Cross platform build support.

3. Dependency management including library linking and library header path supply.

The basic starting point of CMake is the file *CMakeLists.txt*. In this file the project is defined. It can be configured as an executable or a library should be build.

[Hel18] has already developed a CMake generator which is capable to generate C++ code from EMAM models. This generator is based on the bridge pattern [GHJV95] and thus allows flexible changing of the generator implementation. This CMakeGenerator builds a static library from the generated EMAM component. Listing 5.4 shows project definition, creation of a static library of the EMAM component and the export of the CMake module. The presented listing shows is a free marker template used to generate the CMakeLists.txt file.

```
1 project(${compName} LANGUAGES CXX)
2 add_library(${compName} ${compName}.h)
3 export(TARGETS ${compName} FILE ${compName}.cmake)
```

Listing 5.4: CMakeLists.txt showing library definition

The first step is to integrate the generator EMAMOpt2CPP in this framework. As shown in figure 5.16 EMAMOpt2CPP implements the GeneratorImpl interface and thus can be part of the bridge pattern of the CMakeGenerator.

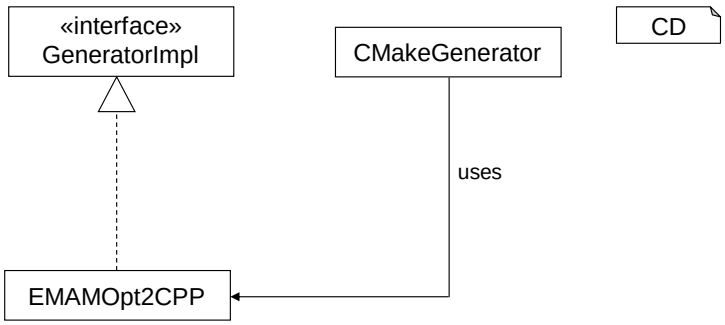


Figure 5.16: CMake generator integration.

The problem with this solution is that the CMakeLists.txt file is generated statically. No dynamic configuration in the Java code was possible. This covers R1 and R2 from the list above but not dependency management. Used libraries and header files have to be located on the standard search path of the operating system if any exist. CMake is able to find modules on its own using the *find_package* command [CMa18]. *FindPackage.cmake* files define paths to include directories and libraries which have to be linked. This solution also cannot be used for the EMAM2CPP generator although it would be sense to have CMake generation for this generator as well.

EMAMOpt2CPP also supports CMake's find package command. It improves the template based generation of the CMakeLists.txt by reusing the mechanisms EMAM2CPP provides. To generate the FindPackage files class *CMakeConfig* are added to the generator (see figure

5.17). This allows a dynamic configuration of the generated CMake files. This solution is implemented in the base generator class GeneratorCPP to also have this feature available for EMAM2CPP.

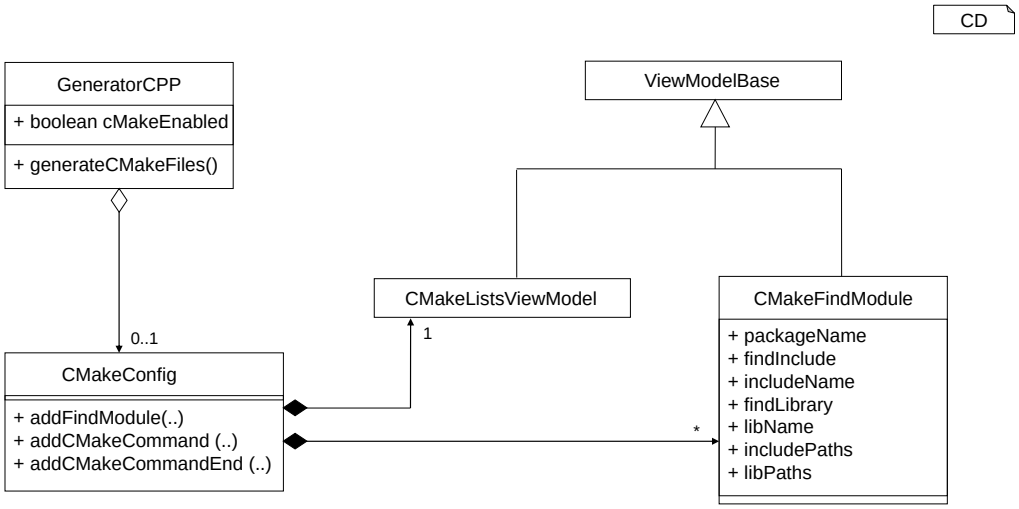


Figure 5.17: CMake Generator in EMAMOpt2CPP class diagram.

The CMakeConfig is publicly available in the generator class. Thus, any class of the generator can add their own CMake dependency to the generator by adding a CMakeFindModule object to the list stored in CMakeConfig. CMakeFindModule can find header directories and or librarys. Additional to the operating system default search paths provided by CMake, the generated find_package file will search in the environment variables for a variable *PackageName_HOME*. Custom search locations can be added in the Java code.

The first dependency which is always added is Aramdillo because EMAM2CPP uses it as backend framework for all math operations. Additionally, the *IPOPTGeneratorImplementation* class (see figure 5.12) adds its dependencies to IPOPT, CPPAD and their required libraries.

Another feature added to the static CMakeLists template are dynamic hooks which allow to insert any CMake command as a string either after the find module definitions or at the end of the file. These hooks were used to additionally generate a shared library for the simulation and create executables for Stream tests (see section 5.4.6).

Through these steps EMAM2CPP and EMAMOpt2CPP generators are capable to generate CMake files for every generated EMAM and EMAMOpt component which builds the component as static library and adds all its dependencies to the compiler and linker.

5.4.5 EMAMOpt2CPP command line interface

The generator EMAMOpt2CPP is implemented as described in the last subsections. To use it from outside a command line interface had to be implemented. The command line interface is based on the EMAM2CPP commandline interface. Table 5.2 contains the already existing command line parameters in EMAM2CPP. Newly added parameters are contained

in table 5.3. Finally, the additional command line parameters of EMAMOpt2CPP are listed in table 5.4.

Option	Description	Values	Optional
-models-dir	Directory where EMAM and EMAMOpt models are stored.	e.g. /home/ Embedded-MontiArcStudio/models/	
-root-model	Fully qualified name of root model.	e.g. de.rwth.monticar.mpc .trajectoryControllerMPC	
-output-dir	Directory where the generated C++ files are located.	e.g. /home/ Embedded-MontiArcStudio/cpp/	
-flag-generate-tests	Defines if stream tests shall be generated.	$b \in \{false, true\}$	✓
-flag-use-algebraic	Defines if algebraic optimizations shall be performed on the generated math code.	$b \in \{false, true\}$	✓
-flag-use-threading	Defines if threading optimizations shall be performed.	$b \in \{false, true\}$	✓
-flag-generate-autopilot-adapter	Indicates if a JNI connection to the simulator should be generated.	$b \in \{false, true\}$	✓
-check-model-dir	Model directory will be checked for creation of component and stream list.	$b \in \{false, true\}$	✓
-flag-generate-server-wrapper	Indicates if code is generated to wrap the model in a server.	$b \in \{false, true\}$	✓

Table 5.2: Already existing command line parameters for EMAM2CPP

Option	Description	Example	Optional
-flag-generate-cmake	Indicates if CMake files shall be generated.	$b \in \{false, true\}$	✓

Table 5.3: New added command line parameters for EMAM2CPP

Option	Description	Example	Optional
-print-level	Verbosity of IPOPT solver output [WLMK11].	$v \in [0, 12]$	✓
-max-iter	Maximal number of iterations [WLMK11].	$n \in \mathbb{N}$	✓
-numeric-tol	Approximate accuracy in first order necessary conditions [WLMK11].	$v \in (0, \infty)$	✓
-point-perturbation-radius	Maximal perturbation of an evaluation point [WLMK11].	$n \in \mathbb{N}$	✓
-cplex-alg	Algorithm to solve the optimization problem [Cor15].	$v \in \{AutoAlg, Dual, Primal, Barrier, Network, Sifting, Concurrent\}$	✓

Table 5.4: New command line parameters for EMAM2CPPOpt

5.4.6 Testing

The language and generator were tested using unit tests. To verify the created languages JUnit [GB99] tests for the parser, AST, CoCos and symbol table were created according to [HR17]. The parser is tested using multiple test models which had to be parsed successfully. The AST is tested against single expressions. Symbol table and CoCos were additionally tested using whole test models of the language and the corresponding properties were checked.

For the generator tests on the Java code and on the generated C++ code had to be established. For verifying if the generator is able to produce C++ code the generated files were compared to reference solutions line by line. For verifying the C++ code is compilable and works correct stream tests [Ryn18] were used on the EMAMOpt components.

Chapter 6

Trajectory Controller

EmbeddedMontiArcMathOpt now provides a powerful basis to model and implement a model predictive trajectory controller. This controller is integrated in the MontiSim framework. The following sections describe the controller design and the integration into the given MontiSim framework (see section 3.2.3). The trajectory controller serves as proof of concept to solve MPC problems.

6.1 Controller Design

MontiSim already provides an interface for trajectory control. For simplicity this interface is reused by the designed trajectory controller. Figure 6.1 presents the trajectory controller design. $\mathcal{I}$ denotes the set of inputs with $\mathcal{I} = \{z, trajectory\}$. Inputs are the vehicle state z at the current time step t with $z = (x, y, \psi, v)^T \in \mathbb{Q}^4$ and the planned high level trajectory with $trajectory = (trajectory_x, trajectory_y)^T \in \mathbb{Q}^2$. The controller is designed to be human readable, thus the input vectors are split into their components in figure 6.1. Absolute vehicle position x and y in the coordinates of the simulator from the state vector z correspond to in-ports x and y from the diagram. ψ denotes the yaw-angle (absolute orientation of the vehicle) and is referenced in the diagram with *compass* and *yaw*. v denotes the current velocity of the vehicle. *trajectory* contains the absolute simulator coordinates of the planned trajectory. $\mathcal{O} = \{engine, steering, brakes\}$ is the set of outputs of the trajectory controller which are used by the simulated car.

The trajectory controller consists of three main components: A path planner (PathPlanner), a low level MPC trajectory controller (MPC) and an actuator controller (ActuatorController). Figure 6.1 shows a visualization this the trajectory controller. PathPlanner gets z and $trajectory$ as input and calculates z_{ref} as output. $z_{ref} = (x_{ref,i}, y_{ref,i}, \psi_{ref,i}, v_{ref,i})^T \in \mathbb{Q}^{4 \times h_p} \forall i : 1 \leq i \leq h_p$ is a low level reference trajectory over the prediction horizon h_p. MPC has input z, z_{ref} and u_{prev} and output $u = (a, \delta_f)^T$. a is the acceleration and δ_f is the steering angle of the front wheels. ActuatorController receives u as input and converts it into actuator values for $engine, brakes$ and $steering$. Path planner, MPC and actuator controller are detailed presented in the next paragraphs.

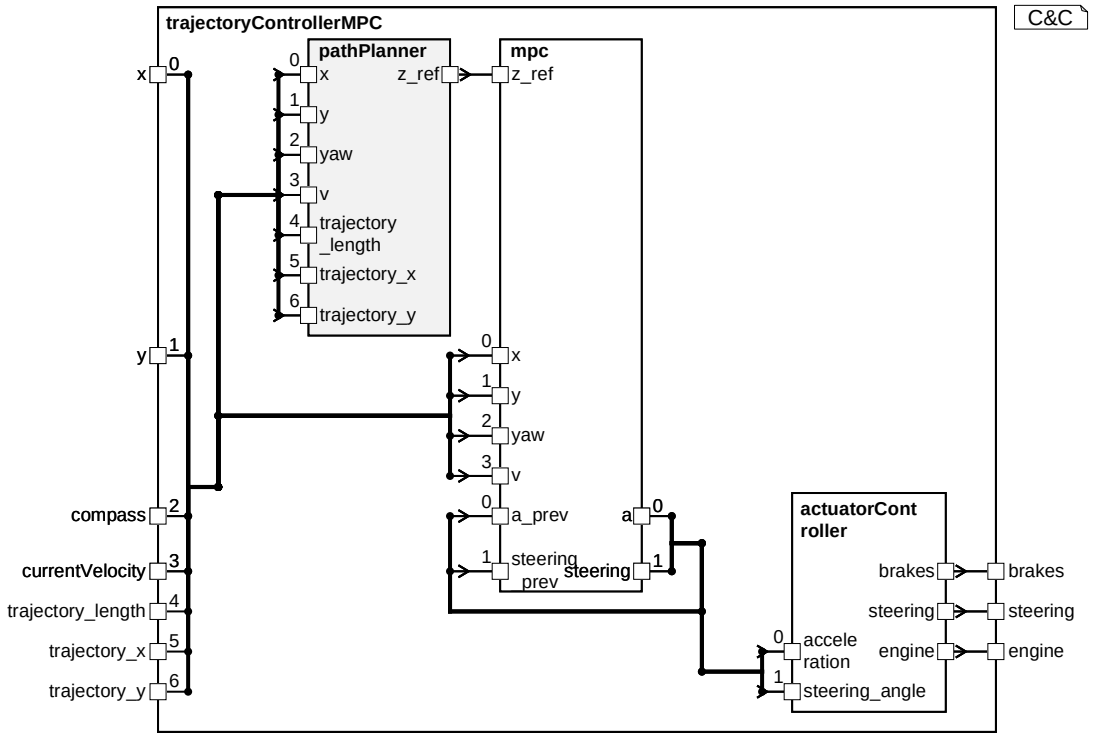


Figure 6.1: Trajectory Controller Design

Path Planner The design of the path planner is presented in figure 6.2. It has sub components PathTrimmer, ReferencePathCreator and VelocityController.

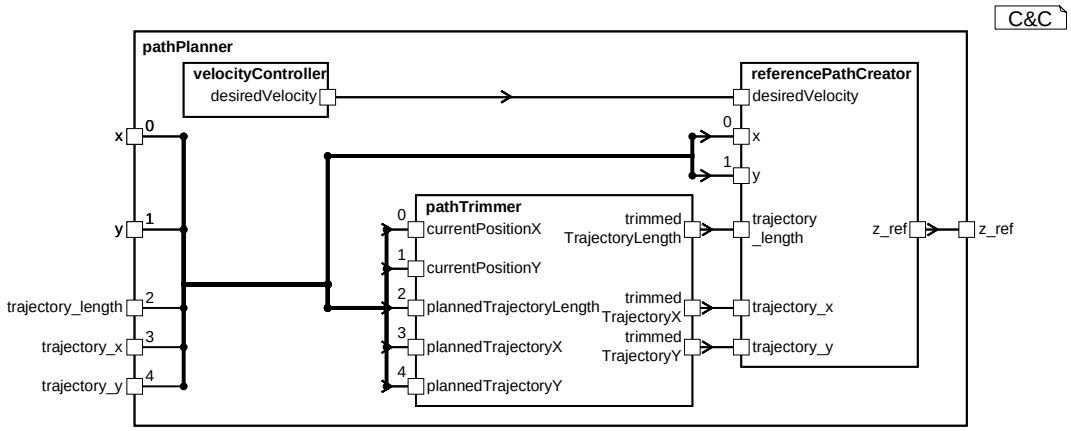


Figure 6.2: Trajectory Controller Design

The PathTrimmer component is reused from [Ryn18]. It trims the path to the current position of the vehicle. The waypoints are projected on the current line segment the vehicle is on. VelocityController returns the desired velocity of the vehicle. In this design it returns a constant predefined velocity. ReferencePathCreator takes the trimmed high level trajectory as input and creates a low level reference path for the MPC-controller. As

proposed in [KPSB15] the path planner creates points on the high level trajectory with its distance depending on the desired velocity (see figure 6.3). The reference yaw-angle is always the direction to the next way point, the reference velocity is the desired velocity.

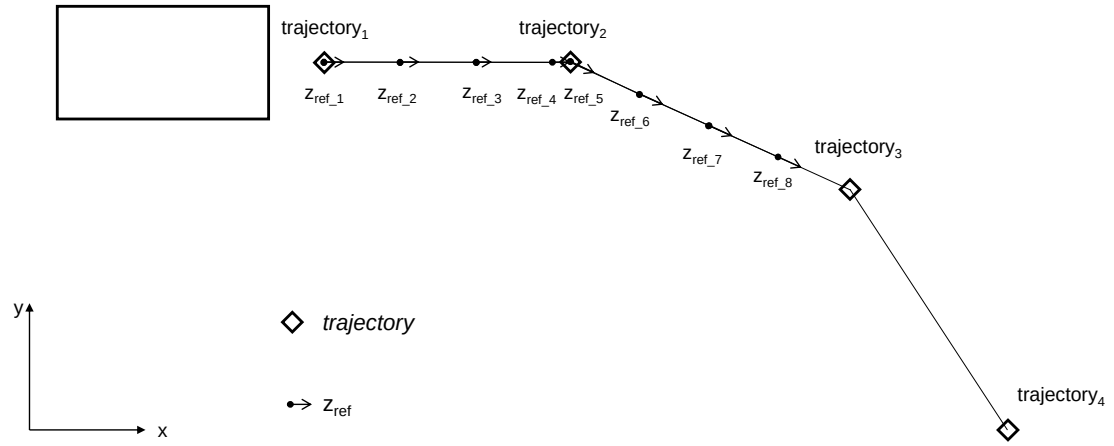


Figure 6.3: Example reference path

The distance d between the reference way points is calculated by equation $d = v_{ref} * dt$ where v_{ref} is the desired velocity and dt is the time increment at each simulation step. The example in figure 6.2 shows 4 trajectory points and a reference path z_{ref} with prediction horizon $h_p = 8$.

MPC Controller The MPC controller is the heart of the trajectory controller. It minimizes the distance $z_t - z_{ref}$ for the whole prediction horizon h_p under given constraints. In this case $z_t \in \mathbb{Q}^{4 \times h_p}$ (Equation 6.1) shows the constraints on the optimization variable u.

$$\begin{aligned} u_{min} &= \begin{pmatrix} a_{min} \\ \delta_{f,min} \end{pmatrix} = \begin{pmatrix} -3m/s^2 \\ -0.785rad \end{pmatrix}, \quad u_{max} = \begin{pmatrix} a_{max} \\ \delta_{f,max} \end{pmatrix} = \begin{pmatrix} 2.5m/s^2 \\ 0.785rad \end{pmatrix} \\ du_{min} &= \begin{pmatrix} da_{min} \\ d\delta_{f,min} \end{pmatrix} = \begin{pmatrix} -3m/s^3 \\ -0.5rad/s \end{pmatrix}, \quad du_{max} = \begin{pmatrix} da_{max} \\ d\delta_{f,max} \end{pmatrix} = \begin{pmatrix} 2.5m/s^3 \\ 0.5rad/s \end{pmatrix} \end{aligned} \tag{6.1}$$

A diagonal matrix $costQ \in \mathbb{Q}^{4 \times 4}$ is defined to weight the error. x, y get a higher weight, to model that keeping the position is more important than ψ and v which have lower weights. The following minimization problem is solved in every time step:

$$\begin{aligned} \min_{u \in \mathbb{Q}^{2 \times h_p}} \quad & \textstyle\sum_{i=1}^{h_p} (z_{t,i} - z_{ref,i})^T costQ (z_{t,i} - z_{ref,i}) \\ \text{s.t.} \quad & z_{t,1} = z \\ & u_0 = u_{prev} \\ & z_{t,i+1} = f(z_{t,i}, u_i) \qquad \forall i \in [1..h_p - 1] \\ & u_{min,i} \leq u_i \leq u_{max,i} \qquad \forall i \in [1..h_p] \\ & du_{min,i} \leq u_i - u_{i-1} \leq du_{max,i} \qquad \forall i \in [1..h_p] \end{aligned} \tag{6.2}$$

The update function $f(f(z_{t,i}, u_i))$ is defined by the used car model. For this controller the Kinematic Bicycle Model (see 3.1.3) is used. The EMAMOpt model of the Kinematic Bicycle MPC controller is attached in listing B.4. The full EMAMOpt model of the trajectory controller is attached in B.3

Actuator Controller As already explained the actuator controller maps u to *engine*, *brakes* and *steering*. The following transfer function is used:

$$\begin{aligned} actuatorController : u = \begin{pmatrix} a \\ \delta_f \end{pmatrix} \rightarrow \begin{pmatrix} engine \\ brakes \\ steering \end{pmatrix} with \\ actuatorController(u) = \begin{cases} a & |a > 0 \\ -a & |a \leq 0 \\ \delta_f \end{cases} \end{aligned} \tag{6.3}$$

6.2 Connection to the Simulator

The trajectory controller design is presented in the last section. The next step is to connect the controller to the simulator. Since the simulator's interface had not been changed, the already developed methods [Ryn18], [Ilo18] can be used. To connect the C++ controller code with the simulator written in Java a JNI interface is used. The flag *flag-generate-autopilot-adapter* of the generator indicates that an interface for that purpose has to be created. The difference to the previous methods is that building the shared library in C++ is done using CMake.

This chapter had shown the design and integration of a trajectory controller into a simulator for autonomous driving. The next chapter will discuss the results of the language development as well as the trajectory controller.

Chapter 7

Evaluation

In this chapter the developed results of chapter 5 and chapter 6 are evaluated. The first sections the extension of the MontiCAR modeling family by MontiMathOpt, Embedded-MontiMathOpt and the code generator EMAMOpt2CPP are evaluated. The other sections evaluate the developed trajectory controller using MontiCAR.

7.1 EmbeddedMontiArcMathOpt

For the evaluation of EMAMOpt and MontiMathOpt these models are compared to the mathematical notation of the problem. An instance of the transportation problem and an example of a nonlinear problem are used as case study. The syntax of MontiMathOpt will be compared to optimization modeling languages and other programming language interfaces.

7.1.1 Modeling of Optimization Problems

To demonstrate EMAMOpt's and especially MontiMathOpt's modeling capabilities, two different prominent optimization problems are modeled using EMAMOpt. The first is a transportation problem, the second an example for a nonlinear problem.

Transportation Problem A transportation problem is a linear optimization problem. There exist m factories, which have a production capacity of $A \in \mathbb{R}^m$. Additionally, there exist n markets with demand $b \in \mathbb{R}^n$. Cost matrix $c \in \mathbb{R}^{m \times n}$ defines the shipping costs from factories to markets. The transportation problem aims to minimize the shipping costs, under the constraints that (1) all markets are supplied and (2) all amounts of the factories are shipped. This instance of the transportation problem is provided by [ER08].

The mathematical formulation is given by equation 7.1, the formulation in EMAMOpt is presented in listing 7.1. The full EMAMOpt model is attached to the appendix listing

B.1.

$$\begin{array}{rll} \min\limits_{x\in\mathbb{R}^{m\times n}} & \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} & \\ \text{s.t.} & \sum_{j=1}^{n} x_{ij} = a_i & \forall i \\ & \sum_{i=1}^{m} x_{ij} = b_j & \forall j \\ & x_{ij} \geq 0 & \forall i \forall j \end{array} \tag{7.1}$$

```
Q y = minimize(Q^{2, 3} x)
  sum(c .* x);
subject to
    sum(x, 2) == A;
    sum(x, 1) == b;
    x >= 0;
end
```

Listing 7.1: Transportation Problem Modelled in EMAMOpt

The comparison of both formulations shows that MontiMathOpt's syntax is very close to the mathematical one. Taking benefit of MontiMath's matrix notation, the expressions are further simplified. Another advantage is that MontiMathOpt is able to integrate vector expressions into the constraints. This avoids a potential vector access on every element in this case. Nevertheless, this is also possible using for loops which is the equivalent to the $\forall$ operator (see section 5.2).

NLP Example Next, a NLP optimization problem is evaluated using EMAMOpt. This example was formulated in Problem 71 of the standard Hock-Schittkowski collection [HS81]. This example should demonstrate that there is no major difference between modeling LP or NLP problems. Again the mathematical formulation is shown in equation 7.2, the EMAMOpt formulation in listing 7.2 and the full EMAMOpt model in appendix B.2.

$$\begin{array}{rlll} \min\limits_{x\in\mathbb{R}^4} & x_1x_4(x_1+x_2+x_3) & & \\ \text{s.t.} & x_1x_2x_3x_4 \geq 25 & & \\ & x_1^2+x_2^2+x_3^2+x_4^2 = 40 & & \\ & 1 \leq x & \leq & 5 \end{array} \tag{7.2}$$

```
Q y = minimize(Q^{4} x)
    x(1) * x(4) * (x(1) + x(2) + x(3)) + x(3);
subject to
    x(1) * x(2) * x(3) * x(4) >= 25;
    x(1)^2) + x(2)^2 + x(3)^2 + x(4)^2 == 40;
    1 <= x <= 5;
end
```

Listing 7.2: Nonlinear Problem Modelled in EMAMOpt

The comparison of both formulations again demonstrate how close EMAMOpt and MontiMathOpt are at the mathematical notation. This is an example to demonstrate that equality, inequality and box constraints are supported. This clearly simplifies modeling.

The model in listing 5.3 is also relevant for the evaluation of constraints. MontiMathOpt is able to combine explicit constraints defined by the optimization problem and implicit range

constraints modeled in MontiMath. The extensive simultaneously use of both constraint types is possible but the is difficult to understand (see listing 5.3). Thus, an approach of good modeling practice is required. It is recommended to model all constrains explicitly in the subject to block. This simplifies reading.

7.1.2 Comparison to other Interfaces

In this subsection MontiMathOpt's syntax and interface is compared to optimization problems expressed in programming languages and also other modeling languages.

EMAMOpt already wraps the C++ interface of its solvers IPOPT and CPLEX. It is obvious that expressing an optimization problem in one of this C++ interfaces is more difficult than in EMAMOpt. The used interfaces are attached in appendix C.1 and appendix C.2. Another relevant case is the comparison to GUROBI's (see A.1) Python interface.

```
# Create a new model
    m = Model("mip1")

# Create variables
x = m.addVar(vtype=GRB.BINARY, name="x")
y = m.addVar(vtype=GRB.BINARY, name="y")
z = m.addVar(vtype=GRB.BINARY, name="z")

# Set objective
m.setObjective(x + y + 2 * z, GRB.MAXIMIZE)

# Add constraint: x + 2 y + 3 z <= 4
m.addConstr(x + 2 * y + 3 * z <= 4, "c0")

# Add constraint: x + y >= 1
m.addConstr(x + y >= 1, "c1")

m.optimize()
```

Listing 7.3: GUROBI python interface example [GO18]

Listing 7.3 shows an example problem which is solved using GUROBI's python interface. First of all, a new model is created. Subsequently, optimization variable, objective function and constraints are added to the model. The problem is solved with the procedure call *optimize*. Compared to the C++ interface of IPOPT and CPLEX, Gurobi's phython interface is less difficult. It is simpler than Matlab's solver *fmincon* (see table 5.1). Nevertheless, GUROBI's and Matlab's interfaces are sequences of function calls. A one-to-one mapping to the mathematical notation is not always possible.

In order to address this problem, modeling languages are used. Modeling languages for optimization are already presented in section 5.2.1. The main differences between *GAMS*, *AMPL* and *CVXGEN* are already discussed in section 5.2.1.

	CVXGEN	AMPL	GAMS	Matlab	GUROBI	**EMAMOpt**
Modeling Language	✓	✓	✓	P	-	✓
Type System	dynamic	dynamic	dynamic	dynamic	dynamic	strict
Unit Support	-	-	P	-	-	✓
C&C Architecture	-	-	-	✓	-	✓
3*rd* Party Solver Support	✓	✓	✓	✓	-	✓
Matrix Support	✓	✓	✓	✓	✓	✓

Table 7.1: Feature Comparison

Table 7.1 compares the features offers by languages CVXGEN, AMPL, GAMS, Matlab (and Simulink), GUROBI (Python interface) and EMAMOpt. The comparison to the python interface of GUROBI (only one solver) is not representative because it is no modeling language. But it is integrated into this comparison to show the benefits of modeling languages compared to solvers directly. All presented languages except EMAMOpt use a dynamic type system. This has the advantage of simpler modeling without considering the type. EMAM's strict type system has the advantage of range and accuracy support. EMAMOpt is the only modeling language that fully supports units. In GAMS units can be added as description to the values but does not include compatibility checks. Simulink (as part of Matlab) and EMAMOpt are the only modeling languages which are able to model a C&C architecture. Third party solver are supported by all modeling languages. GUROBI is a solver itself, so this is irrelevant in this case.

7.1.3 Code Generator

This subsection evaluates the Code Generator EMAM2CPP and EMAMOpt2CPP. The main problem with EMAM2CPP is that it is not clear how to compile, build and execute an EMAM model. This problem is solved by introducing the automatic generation of CMake files. *FindModule* allows a flexible adding of dependencies on demand. That does not solve the problem that users have to install the required software, but at least the user get a hint what kind of modules are required. That includes needed libraries and include headers. EMAMOpt2CPP takes benefit from this in the case of adding dependencies to the needed solvers. Listing 7.4 shows how IPOPT's dependencies can be added using just a few lines of code.

The implemented cooks in the *CMakeLists.txt* (see appendix C.3) have been proofed to allow flexible extension of the generated CMake file for different purposes. Listing 7.5 shows how effective for instance additional executable for Stream tests can be established.

Finally, the Chain-Of-Responsibilities pattern allows an extension of the code generation out of symbols. This feature is used by EMAMOpt2CPP to extend the existing generator. A dynamic extension or behavior change was made possible. Also other extensions may profit from this development.

Next EMAMOpt2CPP is evaluated. The aim is to design a flexible code generator which is able to generate code that solves the optimization problems. Multiple solvers should be

```
public List<CMakeFindModule> getCMakeDependencies() {
        CMakeFindModule findCoinBlas = new CMakeFindModule("CoinBlas
            ", "", "coinblas", new ArrayList<String>(), new ArrayList
            <String>(), false, true, true);
        CMakeFindModule findIPOpt = new CMakeFindModule("Ipopt", "
            coin/IpNLP.hpp", "ipopt", new ArrayList<String>(), new
            ArrayList<String>(), true, true, true);
        CMakeFindModule findCPPAD = new CMakeFindModule("CPPAD", "
            cppad/ipopt/solve.hpp", "", new ArrayList<String>(), new
            ArrayList<String>(), true, false, true);
        return Arrays.asList(findCoinBlas, findIPOpt, findCPPAD);
}
```

Listing 7.4: Add CMake dependecies for IPOPT

```
cmake.addCMakeCommandEnd("include_directories(test)");
cmake.addCMakeCommandEnd("add_executable(StreamTests test/tests_main
    .cpp)");
cmake.addCMakeCommandEnd("target_compile_definitions(StreamTests
    PRIVATE CATCH_CONFIG_MAIN=1 ARMA_DONT_USE_WRAPPER)");
```

Listing 7.5: Usage of hooks in the CMakeLists.txt template

supported. An simple extension of the supported solvers should be possible. The following steps show how any new solver can be added to this framework:

1. Create a solver implementation for the bridge pattern. The class has to implement the *SolverGeneratorImplementation* interface according to its solving capabilities. This class implements the main functionality of the solver code generation. There are no restrictions how the solver code generator is implemented. A template based approach like for IPOPT and CPLEX can be used, but also a BluePrint based approach which is used by EMAM2CPP.

2. The solver then has to be added to the class *SolverFactory*. Here it is necessary to define which class of optimization problems can be handled by the solver.

3. Add the solver to the enumeration *solver*.

EMAMOpt2CPP is able to generate code to solve LP, QP, MIP (if CPLEX is installed), and NLP problems. The options for the different solver can be passed via command line interface or in Java code via the SolverOptions class. That decouples modeling and solving an optimization problem.

7.2 Trajectory Controller

In this section the development of a trajectory controller using the newly added features to the MontiCAR framework EmbeddedMontiArcMathOpt is discussed. The first part is about the development of the MPC controller using EmbeddedMontiArcMathOpt. The last subsection presents an alternative modeling approach using Matlab/Simulinks Model Predictive Control Toolbox [Mat18a].

7.2.1 Controller Design using EmbeddedMontiMathOpt

This subsection discusses the development of an MPC trajectory controller using the Kinematic Bicycle Model. Listing 7.6 shows the optimization statement used in the MPC controller.

```
1  // optimization variable
2  Z hp = 2;
3          Q^{2, hp} u = zeros(2,hp);
4
5          Q^{4, hp} zt = [z(1) + z(4) * cos(z(3) + atan(l_r / (l_f +
               l_r) * tan(u(2, 1)))) * dt, z(1) + z(4) * cos(z(3) + atan
               (l_r / (l_f + l_r) * tan(u(2, 1)))) * dt + z(4) + u(1,1)
                * dt * cos(z(3) + atan(l_r / (l_f + l_r) * tan(u(2, 2)))
               ) * dt;
6
7                          z(2) + z(4) * sin(z(3) + atan(l_r / (l_f +
                               l_r) * tan(u(2, 1))))  * dt, z(1) + z(4) *
                                sin(z(3) + atan(l_r / (l_f + l_r) * tan(u
                               (2, 1)))) * dt + z(4) + u(1,1)  * dt * sin
                               (z(3) + atan(l_r / (l_f + l_r) * tan(u(2,
                               2)))) * dt;
8
9                          z(3) + z(4) / l_r * sin(atan(l_r / (l_f + l_r
                               ) * tan(u(2, 1))))  * dt, z(3) + z(4) /
                               l_r * sin(atan(l_r / (l_f + l_r) * tan(u
                               (2, 1))))  * dt + z(4) + u(1,1)  * dt /
                               l_r * sin(atan(l_r / (l_f + l_r) * tan(u
                               (2, 2))))  * dt;
10
11                         z(4) + u(1,1)  * dt, z(4) + u(1,1)  * dt + u
                               (1,2)  * dt];
12
13 // minimization statement
14 Q cost = minimize(u)
15     abs(sum((zt - z_reft)' * costQ * (zt - z_reft)));
16 subject to
17   for i = 1:hp
18     u_min <= u(:,i) <= u_max;
19   end
20   for i = 1:hp-1
21     du_min <= u(:,i+1) - u(:,i) <= du_max;
22   end
23   du_min <= u(:,1) - u_prev <= du_max;
24 end
```

Listing 7.6: Optimization Statement in the MPC Controller using the Bicycle Model

The forward declaration of z_t helps to keep the objective function short. Nevertheless, defining the state vector for the entire prediction horizon (in this case 2) is complex. The forward declaration and substitution of the optimization variable reaches its limits here. The problem is that no user defined functions can be defined in MontiMath and MontiMathOpt. This has to be improved. A solution to this problem is to allow

The Matlab Model (see section 3.1.3) was not implemented. The first reason for this was that the simulator interface currently does not support the needed input parameters.

The second reason was that MontiCAR does not explicitly support differential equations [KRRvW17]. This makes the modeling and the code generation difficult. Future works should take a look on using the Matlab model as model for the MPC trajectory controller.

The explicit embedding of the state update function of the required car model is contradictory to the C&C architecture. A component based integration of the car model is desired. This would require that not only math expressions can be optimized, but whole sub components. Therefore, the syntax of EmbeddedMontiArc needs to be extended to allow another type of atomic components, respectively optimization components. That is another approach to deal with the missing ability to model functions.

7.2.2 Alternative Modeling Frameworks

An alternative to modeling the MPC trajectory controller is Matlab/Simulinks Model Predictive Control Toolbox [Mat18a].

The Model Predictive Control Toolbox has many advantages that the developed MontiCAR framework also as:

- Third party solver support.
- A specialized IDE to design and simulate the controller (EmbeddedMontiArcStudio enables this partially on MontiCAR).
- Code generation.

Some advantages from the Model Predictive Control Toolbox compared to MontiCAR are (beside that the Model Predictive Control Toolbox is a commercial product with much more development afford):

- Runtime adjustments of weights and constraints
- Explicit support for MPC.

MontiCAR's advantages compared to the Model Predictive Control Toolbox are as already mentioned in section 3.2.2 the stricter type system, unit and accuracy support.

Chapter 8

Conclusion and Future Work

In this study modeling language family MontiCAR has been extended to support optimization problems. The language MontiMathOpt extends MontiMath and allows effective modeling of optimization problems. MontiMathOpt is embedded in language EmbeddedMontiArcMathOpt to allow its usage in C&C models. These models can be used in the same way as EmbeddedMontiArcMath. MontiCore supports the development of the language extensions by providing mechanisms for modular language development.

EmbeddedMontiArcMathOpt is able to model standard optimization problems like transportation problems and nonlinear problems. Also, model predictive control problems can be solved using this language.

The code generator EMAMOpt2CPP allows the generation of C++ code from EMAMOpt models. Its flexible architecture allows simple adding of different solvers for optimization problems. IPOPT is supported as default solver. Thus, the framework is able to solve NLP optimization problems.The solver CPLEX is added as proof of concept for LP, QP and MIP problems. Additional CMake file generation allows a platform independent build of the generated code.

To demonstrate EMAMOpt capabilities in Cyber-Physical Systems a trajectory controller for autonomous driving cars is developed. This controller uses model predictive control to minimize the difference of the actual trajectory to a reference trajectory. The fully automatically generated controller was integrated into the simulator MontiSim and in the integrated development environment for EmbeddedMontiArc EmbeddedMontiArcStudio.

If differential equations have to be modeled and solved, EmbeddedMontiArcMathOpt reaches its limits in case of simplicity. Differential equations are not supported by MontiCAR yet. Optimizing differential equations directly is not supported. Even if there would be a solving method for differential equations integrated in MontiCAR, this probably have to be supported by the optimization framework explicitly. The only way to express and solve differential equations in this setup is to linearize the differential equation. This linearized form can be expressed and solved using the current math language and optimization framework. Another way is to implement numerical solving approaches as EMAM components.

However, future work should solve or improve the following features:

- Support of additional optimization solver. For example a solver supporting convex optimization would supplement the existing ones.

- Support for modeling and solving differential equations. The first step is to extend MontiMath such that it can model differential equations. Then the code generator has to be extended such that it is able to solve this differential equation. A backend capable of solving differential equations has to be chosen.
- Implementation of advanced car models like the Matlab model.
- A controller design which uses the car model as sub component. This requires whole optimization components which can be an extension of EmbeddedMontiArc.
- Support of functions in MontiMath and MontiMathOpt. The lack of defining functions makes writing math functions complicated because the whole function has to be expressed directly as expression.
- Solving optimization problems of other interesting domains. Of course EMAMOpt is designed to cope with CPS. However, the simple optimization interface makes it to an interesting choice for solving optimization problems in domains like Operations Research, Energy Minimization, Machine Learning and of course other CPS like UAV's or robotics.

The simple interface for solving optimization problems, combined with all the features EMAM offers, makes EMAMOpt to a powerful tool with an intuitive development environment. There are many use cases where EMAMOpt can be applied to.

Bibliography

[A+05] Modelica Association et al. Modelica language specification. *Linköping, Sweden*, 2005.

[Abb11] Muhammad Awais Abbas. *Non-linear model predictive control for autonomous vehicles*. PhD thesis, UOIT, 2011.

[Abe14] Dirk Abel. *Regelungstechnik*. Aachener Forschungsgesellschaft Regelungstechnik e.V., 2014.

[Abt18] Dietmar Abts. *Lambdas, Streams und Optional*, pages 219–224. Springer Fachmedien Wiesbaden, Wiesbaden, 2018.

[ARM18] Armadillo C++ library for linear algebra & scientific computing `http://arma.sourceforge.net/docs.html`, july 2018.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Bel12] Bradley M Bell. Cppad: a package for c++ algorithmic differentiation. *Computational Infrastructure for Operations Research*, 57, 2012.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[CGG+13] A. Carvalho, Y. Gao, A. Gray, H. E. Tseng, and F. Borrelli. Predictive control of an autonomous ground vehicle using an iterative linearization approach. In *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, pages 2335–2340, Oct 2013.

[CMa18] CMake 3.12.0-rc2 Documentation `https://cmake.org/cmake/help/v3.12/`, july 2018.

[COI18] Success Stories and Active Work of Ipopt Users `https://projects.coin-or.org/Ipopt/wiki/SuccessStories`, july 2018.

[Cor13] Peter Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB*, pages 65–78. Springer Publishing Company, Incorporated, 1st edition, 2013.

[Cor15] IBM Corp. *IBM ILOG CPLEX Optimization Studio Getting Started with CPLEX*, 2015.

[Cor16] IBM Corp. *IBM ILOG CPLEX Optimization Studio CLEX User's Manual*, 2016.

[CVX18] CVX Users' Guide `http://web.cvxr.com/cvx/beta/doc/solver.html`, july 2018.

[EMA18] EmbeddedMontiArcStudio - Development Suite for EmbeddedMontiArc `https://www.embeddedmontiarc.com/`, july 2018.

[EN18] MathWorks Erik Narby. Modeling a Vehicle Dynamics System `https://de.mathworks.com/help/ident/examples/modeling-a-vehicle-dynam` 2018.

[ER08] Richard E. Rosenthal. Gams – a user's guide. 01 2008.

[Fal07] Paolo Falcone. *Nonlinear model predictive control for autonomous vehicles*. PhD thesis, Università del Sannio, 2007.

[FGK90] Robert Fourer, David M. Gay, and Brian W. Kernighan. A modeling language for mathematical programming. *Manage. Sci.*, 36(5):519–554, May 1990.

[FMS14] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[GAM18] GAMS Documentation `https://gams.com/24.8/docs/welcome.html`, july 2018.

[GB99] Erich Gamma and Kent Beck. Junit: A cook's tour. *Java Report*, 4(5), 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Monticore: A framework for the development of textual domain specific languages. In *Companion of the 30th International Conference on Software Engineering*, ICSE Companion '08, pages 925–926, New York, NY, USA, 2008. ACM.

[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.

[GO18] LLC Gurobi Optimization. Gurobi Documentation `https://www.gurobi.com/documentation/8.0/`, 2018.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Aachener Informatik-Berichte, Software Engineering, Band 24. Shaker Verlag, September 2016.

[Hel18] Alexander Hellwig. Modeling and simulation of cooperative vehicles with monticar and the ros simulation framework. Master's thesis, RWTH Aachen, 2018.

[HJ12] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 2 edition, 2012.

[HKK+18] Steffen Hillemacher, Stefan Kriebel, Evgeny Kusmenko, Mike Lorang, Bernhard Rumpe, Albi Sema, and Michael von Wenckstern. Model-based development of self-adaptive autonomous vehicles using the smardt methodology. *6th International Conference on Model-Driven Engineering and Software Development*, 2018.

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

[HS81] Willi Hock and Klaus Schittkowski. Test examples for nonlinear programming codes, volume 187 of lecture notes in economics and mathematical systems. *Springer-Verlag, Berlin-New York*, 59(87):103, 1981.

[Ilo18] Petyo Ilov. Software architectures of distributed multi-user simulation of autonomous driving vehicles. Master's thesis, RWTH Aachen, 2018.

[KKP+14] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design guidelines for domain specific languages. *CoRR*, abs/1409.2378, 2014.

[KPSB15] J. Kong, M. Pfeiffer, G. Schildbach, and F. Borrelli. Kinematic and dynamic vehicle models for autonomous driving control design. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 1094–1099, June 2015.

[KRRvW17] Evgeny Kusmenko, Alexander Roth, Bernhard Rumpe, and Michael von Wenckstern. *Modeling Architectures of Cyber-Physical Systems*, pages 34–50. Springer International Publishing, Cham, 2017.

[LE14] Torsten Dellmann Lutz Eckstein. *Mechatronische Systeme in der Fahrzeugtechnik*. fka - Forschungsgesellschaft Kraftfahrtwesen mbH Aachen, 2014.

[Mat18a] MathWorks. Model Predictive Control Toolbox `https://de.mathworks.com/products/mpc/features.html`, 2018.

[MAT18b] MATLAB - MathWorks `https://de.mathworks.com/products/matlab.html`, july 2018.

[MB12a] Jacob Mattingley and Stephen Boyd. Cvxgen: a code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, Mar 2012.

[MB12b] Jacob Mattingley and Stephen Boyd. Cvxgen: a code generator for embedded convex optimization. *Optimization and Engineering*, 13(1):1–27, Mar 2012.

[MH15] Ken Martin and Bill Hoffman. *Mastering CMake: a cross-platform build system: version 3.1*. Kitware, 2015.

[Mit18a] H.D. Mittelmann. AMPL-NLP Benchmark `http://plato.asu.edu/ftp/ampl-nlp.html`, june 2018.

[Mit18b] H.D. Mittelmann. Benchmark of Simplex LP solvers `http://plato.asu.edu/ftp/lpsimp.html`, june 2018.

[MRRvW16] Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Consistent extra-functional properties tagging for component and connector models. In *ModComp@ MoDELS*, pages 19–24, 2016.

[MS18] H.D. Mittelmann and P Spellucci. Decision tree for optimization software, 2018.

[MWB10] J. Mattingley, Y. Wang, and S. Boyd. Code generation for receding horizon control. In *2010 IEEE International Symposium on Computer-Aided Control System Design*, pages 985–992, Sept 2010.

[Naz17] Pedram Mir Seyed Nazari. *MontiCore: Efficient Development of Composed Modeling Language Essentials*. Shaker, 2017.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[Ryn18] Alexander Ryndin. Modelling of component-and-connector architectures for autonomous vehicles. Master's thesis, RWTH Aachen, 2018.

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

[Sch18] Sascha Schneiders. Documentation for repository emam2cpp, may 2018.

[TM18] Inc The MathWorks. Simulink® user's guide. *MATLAB & SIMULINK*, Technical Report R2018a, 2018.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Wäc09] Andreas Wächter. Short tutorial: getting started with ipopt in 90 minutes. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[Wal09] Andrea Walther. Getting started with adol-c. 01 2009.

[WB06] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, Mar 2006.

[WLMK11] Andreas Waechter, Carl Laird, F Margot, and Y Kawajir. *Introduction to IPOPT: A tutorial for downloading, installing, and using IPOPT*, 2011.

Appendix A

Evaluated Solvers

Table A.1: Full Solvers Overview [GAM18] [CVX18] [**?**]

Solver	Vendor	Description
ALPHAECP	Abo University	MINLP solver based on the extended cutting plane (ECP) method
ALGLIB	ALGLIB Project	Numerical Library including optimization for NLP, DNLP, QP
AMPL	GAMS Development Corp	A link to solve GAMS models using solvers within the AMPL modeling system
ANTIGONE 1.1	Princeton University	Deterministic global optimization for MINLP
BARON	The Optimization Firm, LLC	Branch-And-Reduce Optimization Navigator for proven global solutions
BDMLP	GAMS Development Corp	LP and MIP solver that comes with any GAMS system
BENCH	GAMS Development Corp	A utility to facilitate benchmarking of GAMS solvers and solution verification
BONMIN 1.8	COIN-OR Foundation	COIN-OR MINLP solver implementing various branch-and-bound and outer approximation algorithms
CBC 2.9	COIN-OR Foundation	High-performance LP/MIP solver
CONOPT 3	ARKI Consulting and Development	Large scale NLP solver
CONOPT 4	ARKI Consulting and Development	Large scale NLP solver
CONVERT	GAMS Development Corp	Framework for translating models into scalar models of other languages
COUENNE 0.5	COIN-OR Foundation	Deterministic global optimization for (MI)NLP
CPLEX 12.7	IBM ILOG	High-performance LP/MIP solver
DE	GAMS Development Corp	Generates and solves the deterministic equivalent of a stochastic program, included in EMP/SP
DECIS	G. Infanger, Inc.	Large scale stochastic programming solver

DICOPT	EDRC, Carnegie Mellon University	Framework for solving MINLP models
EXAMINER	GAMS Development Corp	A tool for examining solution points and assessing their merit
GAMSCHK	Bruce McCarl	A System for Examining the Structure and Solution Properties of Linear Programming Problems Solved using GAMS
GLOMIQO 2.3	Princeton University	Branch-and-bound global optimization for mixed-integer quadratic models
GUROBI 7.5	Gurobi Optimization	State of the Art Mathematical Programming Solver for LP/QP/MIP
GUSS	GAMS Development Corp	A framework for solving many instances of related models efficiently (Gather-Update-Solver-Scatter)
IPOPT 3.12	COIN-OR Foundation	Interior Point Optimizer for large scale nonlinear programming
JAMS	GAMS Development Corp	Solver to reformulate extended mathematical programs (incl. LogMIP)
KESTREL	NEOS	Framework for using remote NEOS solvers with a local GAMS system
KNITRO 10.2	Artelys	Large scale NLP solver
LGO	Pinter Consulting Services	A global-local nonlinear optimization solver suite
LINDO 11.0	Lindo Systems Inc.	A stochastic solver from Lindo Systems, Inc. Includes an unrestricted version of LINDOGLOBAL
LINDOGLOBAL 11.0	Lindo Systems Inc.	MINLP solver for proven global solutions
LINGO	GAMS Development Corp	A link to solve GAMS models using solvers within the LINGO modeling system
LOCALSOLVER 7.0	Innovation 24	Hybrid neighborhood local search solver
LS	Least Square Solver	A Linear Regression Solver for GAMS
MIDACO	MIDACO-SOLVER	MIP, MINLP, NLP solver supported by ESA and Astrium
MILES	University of Colorado at Boulder	MCP solver
MINOS	Stanford University	NLP solver
MOSEK 8	MOSEK ApS	Large scale LP/MIP plus conic and convex non-linear programming system
MSNLP	OptTek Systems and Optimal Methods	Multi-start method for global optimization
NLPEC	GAMS Development Corp	MPEC to NLP translator that uses other GAMS NLP solvers

OQNLP	OptTek Systems and Optimal Methods	Multi-start method for global optimization
PATHNLP	University of Wisconsin - Madison	Large scale NLP solver for convex problems
PATH	University of Wisconsin - Madison	Large scale MCP solver
PYOMO	GAMS Development Corp	A link to solve GAMS models using solvers within the PYOMO modeling system
SeDuMi	J. Sturm, I. Polik	SeDuMi is a Matlab package for solving convex optimization problems involving linear equations and inequalities, second-order cone constraints, and semidefinite constraints (linear matrix inequalities).
SDPT3	K.C. Toh, M. Todd, R. Tutuncu	Designed to solve conic optimization problems
SBB	ARKI Consulting and Development	Branch-and-Bound algorithm for solving MINLP models
SCIP 4.0	Zuse Institute Berlin et.al.	High-performance Constraint Integer Programming solver
SNOPT	Stanford University	Large scale SQP based NLP solver
SOLVEENGINE	Satalia	Link to use solvers of the Satalia SolveEngine with a local GAMS system
SOPLEX 3.0	Zuse Institute Berlin	High-performance LP solver
XA	Sunset Software	Large scale LP/MIP solver
XPRESS 31.01	FICO	High performance LP/MIP solver

Appendix B

Models

B.1 Optimization Models

```
// transportation problem example (linear)
// see https://www.gams.com/products/simple-example/

package de.rwth.monticar.optimization;

component TransportationProblem{

    ports out Q^{3, 2} xOut,
          out Q yOut;

    implementation Math
    {
        // define problem
        Q m = 2;
        Q n = 3;

        // define A, b
        Q^{2, 1} A = [350; 600];
        Q^{3, 1} b = [325; 300; 275];

        // cost matrix
        Q ^{m, n} c = [2.5, 1.7, 1.8; 2.5, 1.8, 1.4];

        // minimization problem
        Q y = minimize(Q^{2, 3} x)
            sum(c .* x);
        subject to
            sum(x, 2) == A;
            sum(x, 1) == b;
            x >= 0;
        end
        xOut = x;
        yOut = y;
    }
}
```

Listing B.1: Transportation Problem

```
// example problem, number 71 from the Hock-Schittkowsky test suite
// W. Hock and K. Schittkowski.
// Test examples for nonlinear programming codes.
// Lecture Notes in Economics and Mathematical Systems, 187, 1981.
// doi: 10.1007/978-3-642-48320-2.

package de.rwth.monticar.optimization;

component HS71{

    ports out Q^{4} xOut,
          out Q yOut;

    implementation Math{
        Q y = minimize(Q^{4} x)
            x(1) * x(4) * (x(1) + x(2) + x(3)) + x(3);
        subject to
            x(1) * x(2) * x(3) * x(4) >= 25;
            x(1)^2) + x(2)^2 + x(3)^2 + x(4)^2 == 40;
      1 <= x <= 5;
        end
        xOut = x;
        yOut = y;
    }
}
```

Listing B.2: HS71 Nonlinear Problem

B.2 Trajectory Controller Models

```
package de.rwth.monticar.mpc;

import de.rwth.monticar.mpc.bicycle.*;
import de.rwth.monticar.mpc.trajectory.*;

component TrajectoryControllerMPC {
  port
    // length of simulation time frame
    in  Q (0.0 s : 0.001 s : 1.0 s)             timeIncrement,

    // current velocity
    in  Q (0.0 m/s : 0.01 m/s : oo m/s)         currentVelocity,

    // current vehicle's position
    in  Q (-oo m : 0.01 m : oo m)               x,
    in  Q (-oo m : 0.01 m : oo m)               y,

    // current vehicle's yaw angle
    in  Q (-oo rad : 0.001 rad : oo rad)        compass,

    // current engine actuation
    in  Q (0.0 m/s : 0.001m/s : 2.5 m/s)        currentEngine,

    // current steering actuation
    // negative value: left turn
```

```
26      // positive value: right turn
27      in  Q (-0.785 rad : 0.001 rad : 0.785 rad)   currentSteering,
28
29      // current brakes actuation
30      in  Q (0.0 m/s : 0.001 m/s : 3.0 m/s)        currentBrakes,
31
32      // planned trajectory (with look ahead 100-200m)
33      // represented by two arrays trajectory_x[] and trajectory_y[]
34      // which both have length trajectory_length
35      in  Z (0 : 100)                              trajectory_length,
36      in  Q (-oo m : 0.01 m : oo m) ^ {1,100}      trajectory_x,
37      in  Q (-oo m : 0.01 m : oo m) ^ {1,100}      trajectory_y,
38
39      // output actuation commands
40      out Q (0.0 m/s : 0.001 m/s : 2.5 m/s)        engine,
41      out Q (-0.785 rad : 0.001 rad : 0.785 rad)   steering,
42      out Q (0.0 m/s : 0.001 m/s : 3.0 m/s)        brakes;
43
44  // trajectory planer
45  instance PathPlaner                              pathPlanner;
46
47  // mpc trajectory controller
48  instance BicycleMPC                              mpc;
49
50  // actuator controller
51  instance ActuatorController                      actuatorController;
52
53  // connect trajectory planer
54  connect x                                        ->      pathPlanner
        .x;
55  connect y                                        ->      pathPlanner
        .y;
56  connect compass                                  ->      pathPlanner
        .yaw;
57  connect currentVelocity                          ->      pathPlanner
        .v;
58
59  connect trajectory_length                        ->      pathPlanner
        .trajectory_length;
60  connect trajectory_x                             ->      pathPlanner
        .trajectory_x;
61  connect trajectory_y                             ->      pathPlanner
        .trajectory_y;
62
63  // connect mpc trajectory controller
64  connect x                                        ->      mpc.x;
65  connect y                                        ->      mpc.y;
66  connect compass                                  ->      mpc.yaw;
67  connect currentVelocity                          ->      mpc.v;
68  connect pathPlanner.z_ref                        ->      mpc.z_ref;
69
70  connect mpc.a                                    ->      mpc.a_prev;
71  connect mpc.steering                             ->      mpc.
        steering_prev;
72
```

```
73    // mpc to actuator controller
74    connect mpc.a                                    ->
          actuatorController.acceleration;
75    connect mpc.steering                             ->
          actuatorController.steering_angle;
76
77    // output
78    connect actuatorController.engine                ->      engine;
79    connect actuatorController.steering              ->      steering;
80    connect actuatorController.brakes                ->      brakes;
81
82  }
```

Listing B.3: Trajectory Controller

```
1  // Kinematic Bicycle model
2  // @author Christoph Richter
3
4  package de.rwth.monticar.mpc.bicycle;
5
6  component BicycleMPC<Q dt = 0.1, Q l_f = 1, Q l_r = 1, Z hp = 1>
7  {
8      ports
9          // state vector z
10         in Q                                          x,
11         in Q                                          y,
12         in Q                                          yaw,
13         in Q (0.0 m/s : 0.01 m/s : oo m/s)            v,
14
15         // reference trajectory
16         in Q^{4, 5}                                  z_ref,
17
18         // previous control in/output
19         in Q (-3.0 m/s^2 : 2.5 m/s^2)                 a_prev,
20         in Q (-0.785 rad : 0.001 rad : 0.785 rad)     steering_prev,
21
22         out Q (-3.0 m/s^2 : 2.5 m/s^2)                a,
23         out Q (-0.785 rad : 0.001 rad : 0.785 rad)    steering;
24
25     implementation Math
26     {
27         // workaround for generator bug -> declare generic type
               arguments
28         Q dt = 0.1 s;
29         Q l_f = 1 m;
30         Q l_r = 1 m;
31
32         // create state vector
33         Q^{4} z;
34         z(1) = x;
35         z(2) = y;
36         z(3) = yaw;
37         z(4) = v;
38         // create previous input vector
39         Q^{2} u_prev;
40         u_prev(1) = a_prev;
41         u_prev(2) = steering_prev;
```

```
42
43          // static bounds on u
44          Q^{2} u_min = [0.0 m/s^2; -0.785];
45          Q^{2} u_max = [2.5 m/s^2; 0.785];
46          Q^{2} du_min = [-3 m/s^3; -0.5];
47          Q^{2} du_max = [2.5 m/s^3; 0.5];
48
49          // define cost matrices
50          Q^{4,4} costQ = eye(4,4);
51          Q^{2,2} costR = eye(2,2);
52          Q^{2,2} costRBar = eye(2,2);
53
54          Q^{4,1} z_reft;
55          z_reft(:, 1) = z_ref(:,1);
56
57          // optimization variable
58      Z hp = 2;
59          Q^{2, hp} u = zeros(2,hp);
60
61          Q^{4, hp} zt = [z(1) + z(4) * cos(z(3) + atan(l_r / (l_f +
                l_r) * tan(u(2, 1)))) * dt, z(1) + z(4) * cos(z(3) + atan
                (l_r / (l_f + l_r) * tan(u(2, 1)))) * dt + z(4) + u(1,1)
                 * dt * cos(z(3) + atan(l_r / (l_f + l_r) * tan(u(2, 2)))
                ) * dt;
62
63                          z(2) + z(4) * sin(z(3) + atan(l_r / (l_f +
                               l_r) * tan(u(2, 1))))  * dt, z(1) + z(4) *
                                sin(z(3) + atan(l_r / (l_f + l_r) * tan(u
                               (2, 1)))) * dt + z(4) + u(1,1)  * dt * sin
                               (z(3) + atan(l_r / (l_f + l_r) * tan(u(2,
                               2)))) * dt;
64
65                          z(3) + z(4) / l_r * sin(atan(l_r / (l_f + l_r
                               ) * tan(u(2, 1))))  * dt, z(3) + z(4) /
                               l_r * sin(atan(l_r / (l_f + l_r) * tan(u
                               (2, 1))))  * dt + z(4) + u(1,1)  * dt /
                               l_r * sin(atan(l_r / (l_f + l_r) * tan(u
                               (2, 2))))  * dt;
66
67                          z(4) + u(1,1)  * dt, z(4) + u(1,1)  * dt + u
                               (1,2)  * dt];
68
69          // minimization statement
70          Q error = minimize(u)
71            abs(sum((zt - z_reft)' * costQ * (zt - z_reft)));
72          subject to
73        for i = 1:hp
74          u_min <= u(:,i) <= u_max;
75        end
76        for i = 1:hp-1
77          du_min <= u(:,i+1) - u(:,i) <= du_max;
78        end
79        du_min <= u(:,1) - u_prev <= du_max;
80          end
81
```

```
82          // assign output
83          a = u(1,1);
84          steering = u(2,1);
85      }
86  }
```

Listing B.4: Kindematic Bicycle MPC

Appendix C

Templates

```
#ifndef __${viewModel.callSolverName?upper_case}_H__
#define __${viewModel.callSolverName?upper_case}_H__
#include<armadillo>
#include<cppad/ipopt/solve.hpp>
#include "ADMat.h"

using CppAD::AD;
using namespace arma;

typedef CPPAD_TESTVECTOR(double) Dvector;
typedef CPPAD_TESTVECTOR(CppAD::AD<double>) ADvector;

namespace AnonymNS${viewModel.id}
{
  using CppAD::AD;
  using namespace arma;

  static Dvector gl = Dvector();
  static Dvector gu = Dvector();
  static ADvector g = ADvector();

  <#list viewModel.knownVariablesWithType as var>
  static ${var};
  </#list>

  class FG_eval_${viewModel.callSolverName} {
    public:
      typedef CPPAD_TESTVECTOR(AD<double>) ADvector;

      void operator()(ADvector &fg,const ADvector &x) {

        // create active optimization var
        <#if viewModel.optimizationVariableDimensions?size == 0>
        ${viewModel.optimizationVariableTypeActive} ${viewModel.
            optimizationVariableName} = x[0];
        <#else>
        ${viewModel.optimizationVariableTypeActive} ${viewModel.
            optimizationVariableName} = ${viewModel.
            optimizationVariableTypeActive}(<#list viewModel.
            optimizationVariableDimensions as dim>${dim}<#sep>, </#
```

```
            list>);
37          for (int i${viewModel.id} = 0; i${viewModel.id} < x.size();
               i${viewModel.id}++)
38          {
39            ${viewModel.optimizationVariableName}(i${viewModel.id}) =
                 x[i${viewModel.id}];
40          }
41          </#if>
42
43          // f(x)
44          int i${viewModel.id} = 0;
45          <#if viewModel.optimizationProblemType.name() == "
               MINIMIZATION">
46          fg[i${viewModel.id}] = toADouble(${viewModel.
               objectiveFunction});
47          <#else>
48          fg[i${viewModel.id}] = -1 * toADouble( ${viewModel.
               objectiveFunction} );
49          </#if>
50
51          // g_i(x)
52          i${viewModel.id}++;
53          <#list viewModel.constraintFunctions as g>
54          addConstraintFunction(${g}, fg, i${viewModel.id});
55          </#list>
56          //
57          return;
58        }
59
60      private:
61        adouble toADouble(const ADMat &value) { return value.at(0);}
62
63        adouble toADouble(const adouble &value) { return value;}
64
65        void addConstraintFunction(const adouble &value, ADvector &fg,
               int &i) {
66          fg[i] = value;
67          i++;
68        }
69
70        void addConstraintFunction(const ADMat &value, ADvector &fg,
              int &i) {
71          for(int j = 0; j < value.size(); j++) {
72            fg[i] = value[j];
73            i++;
74          }
75        }
76    };
77  }
78
79  using namespace arma;
80
81  class ${viewModel.callSolverName}
82  {
83    private:
```

```
84
85      static void addConstraint(const double &lower, const adouble &
            expr, const double &upper) {
86        AnonymNS${viewModel.id}::gl.push_back(lower);
87        AnonymNS${viewModel.id}::gu.push_back(upper);
88        AnonymNS${viewModel.id}::g.push_back(expr);
89      };
90
91      static void addConstraint(const double &lower, const ADMat &expr
            , const double &upper) {
92        for (int i = 0; i < expr.size(); i++) {
93          AnonymNS${viewModel.id}::gl.push_back(lower);
94          AnonymNS${viewModel.id}::gu.push_back(upper);
95          AnonymNS${viewModel.id}::g.push_back(expr[i]);
96        }
97      };
98
99      static void addConstraint(const double &lower, const ADMat &expr
            , const mat &upper) {
100       assert(expr.size() == upper.size());
101       for (int i = 0; i < expr.size(); i++) {
102         AnonymNS${viewModel.id}::gl.push_back(lower);
103         AnonymNS${viewModel.id}::gu.push_back(upper[i]);
104         AnonymNS${viewModel.id}::g.push_back(expr[i]);
105       }
106     };
107
108     static void addConstraint(const mat &lower, const ADMat &expr,
            const double &upper) {
109       assert(lower.size() == expr.size());
110       for (int i = 0; i < expr.size(); i++) {
111         AnonymNS${viewModel.id}::gl.push_back(lower[i]);
112         AnonymNS${viewModel.id}::gu.push_back(upper);
113         AnonymNS${viewModel.id}::g.push_back(expr[i]);
114       }
115     };
116
117     static void addConstraint(const mat &lower, const ADMat &expr,
            const mat &upper) {
118       assert(lower.size() == expr.size());
119       assert(expr.size() == upper.size());
120       for (int i = 0; i < expr.size(); i++) {
121         AnonymNS${viewModel.id}::gl.push_back(lower[i]);
122         AnonymNS${viewModel.id}::gu.push_back(upper[i]);
123         AnonymNS${viewModel.id}::g.push_back(expr[i]);
124       }
125     };
126
127     static void addConstraint(const double &lower, const arma::
            subview_field<adouble> &expr, const double &upper) {
128       for (int i = 0; i < expr.n_rows; i++) {
129         for (int j = 0; j < expr.n_cols; j++) {
130           AnonymNS${viewModel.id}::gl.push_back(lower);
131           AnonymNS${viewModel.id}::gu.push_back(upper);
132           AnonymNS${viewModel.id}::g.push_back(expr[i, j]);
```

```
133         }
134       }
135     };
136 
137     static void addConstraint(const mat &lower, const arma::
          subview_field<adouble> &expr, const double &upper) {
138       assert(lower.n_cols == expr.n_cols);
139       assert(lower.n_rows == expr.n_rows);
140       for (int i = 0; i < expr.n_rows; i++) {
141         for (int j = 0; j < expr.n_cols; j++) {
142           AnonymNS${viewModel.id}::gl.push_back(lower[i, j]);
143           AnonymNS${viewModel.id}::gu.push_back(upper);
144           AnonymNS${viewModel.id}::g.push_back(expr[i, j]);
145         }
146       }
147     };
148 
149     static void addConstraint(const double &lower, const arma::
          subview_field<adouble> &expr, const mat &upper) {
150       for (int i = 0; i < expr.n_rows; i++) {
151         for (int j = 0; j < expr.n_cols; j++) {
152           AnonymNS${viewModel.id}::gl.push_back(lower);
153           AnonymNS${viewModel.id}::gu.push_back(upper[i, j]);
154           AnonymNS${viewModel.id}::g.push_back(expr[i, j]);
155         }
156       }
157     };
158 
159     static void addConstraint(const mat &lower, const arma::
          subview_field<adouble> &expr, const mat &upper) {
160       for (int i = 0; i < expr.n_rows; i++) {
161         for (int j = 0; j < expr.n_cols; j++) {
162           AnonymNS${viewModel.id}::gl.push_back(lower[i, j]);
163           AnonymNS${viewModel.id}::gu.push_back(upper[i, j]);
164           AnonymNS${viewModel.id}::g.push_back(expr[i, j]);
165         }
166       }
167     };
168 
169     static void addConstraintOnX(Dvector &xl, Dvector &xu, const
          double &lower, int index, const double &upper) {
170       xl[index] = std::fmax(xl[index], lower);
171       xu[index] = std::fmin(xu[index], upper);
172     };
173 
174   public:
175     static bool solveOptimizationProblemIpOpt(
176     ${viewModel.optimizationVariableType} &x${viewModel.id},
177     double &y${viewModel.id}<#if 0 < viewModel.
          knownVariablesWithType?size>,</#if>
178     <#list viewModel.knownVariablesWithType as arg>
179     const ${arg}<#sep>,</#sep>
180     </#list>
181     )
182     {
```

```
183         bool ok = true;
184 
185         // declare opt var
186         <#if viewModel.optimizationVariableDimensions?size == 0>
187         ${viewModel.optimizationVariableTypeActive} ${viewModel.
                optimizationVariableName} = 0;
188         <#else>
189         ${viewModel.optimizationVariableTypeActive} ${viewModel.
                optimizationVariableName} = ${viewModel.
                optimizationVariableTypeActive}(<#list viewModel.
                optimizationVariableDimensions as dim>${dim}<#sep>, </#list
                >);
190         </#if>
191 
192         // assign parameter variables
193         <#list viewModel.knownVariables as var>
194         AnonymNS${viewModel.id}::${var} = ${var};
195         </#list>
196 
197         typedef CPPAD_TESTVECTOR(double)Dvector;
198 
199         // number of independent variables (domain dimension for f and
                g)
200         size_t nx = ${viewModel.numberVariables?c};
201         // number of constraints (range dimension for g)
202         size_t ng = ${viewModel.numberConstraints?c};
203         // initial value of the independent variables
204         Dvector xi(nx);
205         int i${viewModel.id} = 0;
206         <#list viewModel.initX as x>
207         xi[i${viewModel.id}] = ${x};
208         i${viewModel.id}++;
209         </#list>
210         // lower and upper limits for x
211         Dvector xl(nx),xu(nx);
212         i${viewModel.id} = 0;
213         <#list viewModel.xL as x>
214         xl[i${viewModel.id}] = ${x};
215         i${viewModel.id}++;
216         </#list>
217         i${viewModel.id} = 0;
218         <#list viewModel.xU as x>
219         xu[i${viewModel.id}] = ${x};
220         i${viewModel.id}++;
221         </#list>
222 
223         // limits for special matrix elements of x
224         <#list viewModel.xMatrixElementConstraints as element>
225         addConstraintOnX(xl, xu, ${element});
226         </#list>
227 
228         // lower and upper limits for g
229         Dvector gl(ng),gu(ng);
230         <#list viewModel.constraintFunctions as g>
```

```
231       addConstraint(${viewModel.gL[g?index]}, ${g}, ${viewModel.gU[g
              ?index]});
232       </#list>
233
234       // object that computes objective and constraints
235       AnonymNS${viewModel.id}::FG_eval_${viewModel.callSolverName}
              fg_eval;
236
237       // options
238       std::string options;
239       <#list viewModel.options as option>
240       options+="${option}\n";
241       </#list>
242       // place to return solution
243       CppAD::ipopt::solve_result<Dvector> solution;
244
245       // solve the problem
246       CppAD::ipopt::solve<Dvector, AnonymNS${viewModel.id}::
              FG_eval_${viewModel.callSolverName}>(
247         options, xi, xl, xu, AnonymNS${viewModel.id}::gl, AnonymNS${
                viewModel.id}::gu, fg_eval, solution);
248
249       // Check some of the solution values
250       ok&=solution.status==CppAD::ipopt::solve_result<Dvector>::
              success;
251
252       // assign solution values
253       <#if viewModel.optimizationVariableDimensions?size == 0>
254       x${viewModel.id} = solution.x[0];
255       <#else>
256       for (int i${viewModel.id} = 0; i${viewModel.id} < solution.x.
              size(); i${viewModel.id}++)
257       {
258         x${viewModel.id}(i${viewModel.id}) = solution.x[i${viewModel
                .id}];
259       }
260       </#if>
261       // objective value
262       <#if viewModel.optimizationProblemType.name() == "MINIMIZATION
              ">
263       y${viewModel.id} = solution.obj_value;
264       <#else>
265       y${viewModel.id} = -1 * solution.obj_value;
266       </#if>
267
268         std::cout<<std::endl<<"z ="<<std::endl<< z <<std::endl;
269         std::cout<<std::endl<<"z_ref ="<<std::endl<< z_ref <<std::
                endl;
270
271
272       // print short message
273       std::cout<<std::endl<<std::endl<<"Solving status: "<<solution.
              status<<"!"<<std::endl;
274       std::cout<<"${viewModel.optimizationProblemType.name()?
              capitalize} variable value: "<<std::endl<<"x = "<<std::endl
```

```
          <<x${viewModel.id}<<std::endl;
275       std::cout<<"${viewModel.optimizationProblemType.name()?
             capitalize} objective value: "<<std::endl<<"y = "<<y${
             viewModel.id}<<std::endl;
276       return ok;
277     }
278 };
279
280 #endif
```

Listing C.1: Freemarker Template for IPOPT interface

```
1  #ifndef __${viewModel.callSolverName?upper_case}_H__
2  #define __${viewModel.callSolverName?upper_case}_H__
3
4  #include <ilcplex/ilocplex.h>
5  #include "armadillo.h"
6  #include "CplexMat.h"
7
8
9  class ${viewModel.callSolverName}
10 {
11   private:
12
13     static IloNumExprArg toScalar(const CplexMat &value) { return
          value.at(0);}
14
15     static IloNumExprArg toScalar(const IloNumExprArg &value) {
          return value;}
16
17     static void addConstraint(IloRangeArray &con${viewModel.id},
          const double &lower, const IloNumExprArg &expr, const double
          &upper) {
18       con${viewModel.id}.add(lower <= expr <= upper);
19     };
20
21     static void addConstraint(IloRangeArray &con${viewModel.id},
          const double &lower, const CplexMat &expr, const double &
          upper) {
22       for (int i = 0; i < expr.size(); i++) {
23         con${viewModel.id}.add(lower <= expr[i] <= upper);
24       }
25     };
26
27     static void addConstraint(IloRangeArray &con${viewModel.id},
          const double &lower, const CplexMat &expr, const mat &upper)
          {
28       assert(expr.size() == upper.size());
29       for (int i = 0; i < expr.size(); i++) {
30         con${viewModel.id}.add(lower <= expr[i] <= upper[i]);
31       }
32     };
33
34     static void addConstraint(IloRangeArray &con${viewModel.id},
          const mat &lower, const CplexMat &expr, const double &upper)
          {
35       assert(lower.size() == expr.size());
```

```
36        for (int i = 0; i < expr.size(); i++) {
37          con${viewModel.id}.add(lower[i] <= expr[i] <= upper);
38        }
39      };
40
41      static void addConstraint(IloRangeArray &con${viewModel.id},
            const mat &lower, const CplexMat &expr, const mat &upper) {
42        assert(lower.size() == expr.size());
43        assert(expr.size() == upper.size());
44        for (int i = 0; i < expr.size(); i++) {
45          con${viewModel.id}.add(lower[i] <= expr[i] <= upper[i]);
46        }
47      };
48
49      static void addConstraint(IloRangeArray &con${viewModel.id},
            const double &lower, const arma::subview_field<IloNumExprArg>
             &expr, const double &upper) {
50        for (int i = 0; i < expr.n_rows; i++) {
51          for (int j = 0; j < expr.n_cols; j++) {
52            con${viewModel.id}.add(lower <= expr[i, j] <= upper);
53          }
54        }
55      };
56
57      static void addConstraint(IloRangeArray &con${viewModel.id},
            const mat &lower, const arma::subview_field<IloNumExprArg> &
            expr, const double &upper) {
58        assert(lower.n_cols == expr.n_cols);
59        assert(lower.n_rows == expr.n_rows);
60        for (int i = 0; i < expr.n_rows; i++) {
61          for (int j = 0; j < expr.n_cols; j++) {
62            con${viewModel.id}.add(lower[i, j] <= expr[i, j] <= upper)
                ;
63          }
64        }
65      };
66
67      static void addConstraint(IloRangeArray &con${viewModel.id},
            const double &lower, const arma::subview_field<IloNumExprArg>
             &expr, const mat &upper) {
68        for (int i = 0; i < expr.n_rows; i++) {
69          for (int j = 0; j < expr.n_cols; j++) {
70            con${viewModel.id}.add(lower <= expr[i, j] <= upper[i, j])
                ;
71          }
72        }
73      };
74
75      static void addConstraint(IloRangeArray &con${viewModel.id},
            const mat &lower, const arma::subview_field<IloNumExprArg> &
            expr, const mat &upper) {
76        for (int i = 0; i < expr.n_rows; i++) {
77          for (int j = 0; j < expr.n_cols; j++) {
78            con${viewModel.id}.add(lower[i, j] <= expr[i, j] <= upper[
                i, j]);
```

```
79          }
80        }
81      };
82
83    public:
84      static bool solveOptimizationProblemCplex(
85        ${viewModel.optimizationVariableType} &x${viewModel.id},
86        double &y${viewModel.id}<#if 0 < viewModel.
              knownVariablesWithType?size>,</#if>
87        <#list viewModel.knownVariablesWithType as arg>
88        const ${arg}<#sep>,</#sep>
89        </#list>
90      )
91      {
92        ILOSTLBEGIN
93        IloEnv env;
94          try {
95            IloModel model(env);
96            IloNumVarArray var${viewModel.id}(env);
97            IloRangeArray con${viewModel.id}(env);
98            IloEnv env = model.getEnv();
99            <#list viewModel.xL as xl>
100           var${viewModel.id}.add(IloNumVar(env, ${xl}, ${viewModel.
                  xU[xl?index]}));
101           </#list>
102           // create active optimization var
103           <#if viewModel.optimizationVariableDimensions?size == 0>
104           ${viewModel.optimizationVariableTypeActive} ${viewModel.
                  optimizationVariableName} = var${viewModel.id}[0];
105           <#else>
106           ${viewModel.optimizationVariableTypeActive} ${viewModel.
                  optimizationVariableName} = ${viewModel.
                  optimizationVariableTypeActive}(<#list viewModel.
                  optimizationVariableDimensions as dim>${dim}<#sep>, </#
                  list>);
107           for (int i${viewModel.id} = 0; i${viewModel.id} < var${
                  viewModel.id}.getSize(); i${viewModel.id}++)
108           {
109             ${viewModel.optimizationVariableName}(i${viewModel.id})
                    = var${viewModel.id}[i${viewModel.id}];
110           }
111           </#if>
112           <#if viewModel.optimizationProblemType.name() == "
                  MINIMIZATION">
113           model.add(IloMinimize(env, toScalar(${viewModel.
                  objectiveFunction})));
114           <#else>
115           model.add(IloMaximize(env, toScalar(${viewModel.
                  objectiveFunction})));
116           </#if>
117           <#list viewModel.constraintFunctions as g>
118           addConstraint(con${viewModel.id}, ${viewModel.gL[g?index
                  ]}, ${g}, ${viewModel.gU[g?index]});
119           </#list>
120           model.add(con${viewModel.id});
```

```
121
122           IloCplex cplex(model);
123           cplex.solve();
124
125           env.out() << "Solution status = " << cplex.getStatus() <<
                 endl;
126           env.out() << "Solution value  = " << cplex.getObjValue()
                 << endl;
127
128           IloNumArray vals(env);
129           cplex.getValues(vals, var${viewModel.id});
130           env.out() << "Values        = " << vals << endl;
131           cplex.getSlacks(vals, con${viewModel.id});
132           env.out() << "Slacks        = " << vals << endl;
133
134           cplex.exportModel("mipex1.lp");
135         }
136         catch (IloException& e) {
137           cerr << "Concert exception caught: " << e << endl;
138         }
139         catch (...) {
140           cerr << "Unknown exception caught" << endl;
141         }
142
143         env.end();
144
145         return 0;
146     }
147 };
148
149 #endif
```

Listing C.2: Freemarker Template for CPLEX interface

```
cmake_minimum_required(VERSION 3.5)
set(CMAKE_CXX_STANDARD 11)

project(${viewModel.compName} LANGUAGES CXX)

#set cmake module path
set(CMAKE_MODULE_PATH ${r"${CMAKE_MODULE_PATH}"} ${r"${
    CMAKE_CURRENT_SOURCE_DIR}"}/cmake)

# add dependencies
<#list viewModel.moduleDependencies as var>
find_package(${var.packageName} <#if var.required>REQUIRED<#else>
    OPTIONAL</#if>)
<#if var.findPath>set(INCLUDE_DIRS ${r"${INCLUDE_DIRS}"} ${r"${"}${
    var.packageName}${r"_INCLUDE_DIRS}"})</#if>
<#if var.findLibrary>set(LIBS ${r"${LIBS}"} ${r"${"}${var.
    packageName}${r"_LIBRARIES}"})</#if>
</#list>

# additional commands
<#list viewModel.cmakeCommandList as cmd>
${cmd}
</#list>

# create static library
include_directories(${r"${INCLUDE_DIRS}"})
add_library(${viewModel.compName} ${viewModel.compName}.h)
target_include_directories(${viewModel.compName} PUBLIC ${r"${
    CMAKE_CURRENT_SOURCE_DIR}"})
target_link_libraries(${viewModel.compName} PUBLIC ${r"${LIBS}"})
set_target_properties(${viewModel.compName} PROPERTIES
    LINKER_LANGUAGE CXX)

# export cmake project
export(TARGETS ${viewModel.compName} FILE ${viewModel.compName}.
    cmake)

# additional commands end
<#list viewModel.cmakeCommandListEnd as cmd>
${cmd}
</#list>
```

Listing C.3: Freemarker Template for CMAKELists.txt

```
cmake_minimum_required(VERSION 3.5)
set(CMAKE_CXX_STANDARD 11)

project(${viewModel.compName} LANGUAGES CXX)

#set cmake module path
set(CMAKE_MODULE_PATH ${r"${CMAKE_MODULE_PATH}"} ${r"${
    CMAKE_CURRENT_SOURCE_DIR}"}/cmake)

# add dependencies
<#list viewModel.moduleDependencies as var>
find_package(${var.packageName} <#if var.required>REQUIRED<#else>
    OPTIONAL</#if>)
<#if var.findPath>set(INCLUDE_DIRS ${r"${INCLUDE_DIRS}"} ${r"${"}${
    var.packageName}${r"_INCLUDE_DIRS}"})</#if>
<#if var.findLibrary>set(LIBS ${r"${LIBS}"} ${r"${"}${var.
    packageName}${r"_LIBRARIES}"})</#if>
</#list>

# additional commands
<#list viewModel.cmakeCommandList as cmd>
${cmd}
</#list>

# create static library
include_directories(${r"${INCLUDE_DIRS}"})
add_library(${viewModel.compName} ${viewModel.compName}.h)
target_include_directories(${viewModel.compName} PUBLIC ${r"${
    CMAKE_CURRENT_SOURCE_DIR}"})
target_link_libraries(${viewModel.compName} PUBLIC ${r"${LIBS}"})
set_target_properties(${viewModel.compName} PROPERTIES
    LINKER_LANGUAGE CXX)

# export cmake project
export(TARGETS ${viewModel.compName} FILE ${viewModel.compName}.
    cmake)

# additional commands end
<#list viewModel.cmakeCommandListEnd as cmd>
${cmd}
</#list>
```

Listing C.4: Freemarker Template for CMAKE find_package