TECHNISCHE HOCHSCHULE MITTELHESSEN | THM | CAMPUS FRIEDBERG | M Maschinenbau, Mechatronik, Materialtechnologie

Abschlussbericht „Höhere Informatik“

Thema:

„Softwaretechnische Umsetzung einer Tiefsetzsteller-Simulation in C++”

Vorgelegt von:	Philipp Blum	5482067
	Leonard Hisgen	5090044
	Adrian Lauster	5228887
	Lukas Rother	5221525
Gruppe:	Gruppe 3	
Hochschule:	Technische Hochschule Mittelhessen	
Fachbereich:	FB12 Maschinenbau, Mechatronik, Materialtechnologie	
Studiengang:	Maschinenbau Mechatronik	
Professor	Prof. Dr.-Ing. Uwe Probst	
Eingereicht am:	08.04.2024	

Inhalt

I. Einleitung & Aufbau des Programms

In der folgenden Arbeit soll die softwaretechnische Simulation eines Tiefsetzstellers in der Programmiersprache C++ und der Klassenbibliothek „MFC“ erarbeitet werden. Als Basis dieser Arbeit dienen die Kursinhalte der Module „Softwareentwicklung“ und „Höhere Informatik“. Zusätzlich orientiert sich die Programmierung an der Literatur „Objektorientiertes Programmieren. Eine Einführung für die Ingenieurwissenschaften in C++" (Probst, 2023), von Uwe Probst. Einzelne Programmabschnitte können Ähnlichkeiten zu Code - Beispielen und Übungen aus der Literatur haben. Für das Projekt wird die „STL-Bibliothek“ verwendet. Die Simulation und graphische Darstellung werden mittels eigener Klassen und Methoden umgesetzt.
Zu Beginn der Arbeit soll die Benutzeroberfläche inklusiver Graphen und Bedienelemente vorgestellt werden. Anschließend soll die Softwarearchitektur erklärt werden, welcher das Entwurfsmuster „Model View Controller“ zugrundeliegt.

A. Benutzeroberfläche

Die Benutzeroberfläche besteht aus einem Dialogfenster, das beim Start der Anwendung geöffnet wird. Dabei ist die Oberfläche grundsätzlich in drei Bereiche eingeteilt. Auf der linken Seite befinden sich die Graphen, in denen die Werteverläufe der Simulation dargestellt werden. Rechts oben ist der Schaltplan des Tiefsetzstellers abgebildet. Dieser besteht aus dem eigentlichen Schaltplan, den drei Bauelement-Controllern, die die Spule, den Kondensator und den Widerstand darstellen soll und dem Schalter, der jeweils eine geöffnete und geschlossene Position besitzt. Zusätzlich existieren drei Controller für die Eingabe der Frequenz, des Tastgrads und der Eingangspannung. Für die Darstellung der Zahlenwerte befinden sich unten rechts eine Vielzahl von Edit-Felder, die die berechneten Werte der Simulation und Controllern anzeigen.

B. Softwarearchitektur und UML-Diagramm

Das Entwurfsmuster „Model-View-Controller“ wird für die Softwarearchitektur ausgewählt (Probst, 2023). Die abstrakten Klassen *CAbstractModel* und *CAbstractController* und *CAbstractView* dienen als Basis für das Entwurfsmuster Aus diesen Mutterklassen erbt sowohl die Klasse *CModel* als auch alle Klassen der Views und Controller. Insgesamt gibt es folgende sechs Controller: *CCoilController*, *CCapacityController*, *CResistorController*, *CFrequencyController*, *CDutyCycleController* und *CVoltageController*. Diese erben aus *CAbstractController*. Aus der Klasse *CAbstractView* werden alle Views abgeleitet. Darunter *CCircuit, CGraph, CSwitch. Abbildung 4 (Anhang) zeigt das UML-Diagramm. Folgend sollen die Klassen näher erläutert werden.*

1) CAbstractModel und CModel

CAbstractModel dient als Grundlage für das Entwurfsmuster. Es besitzt jeweils eine Liste für die Views und Controller. Über die Methode *registerView* und *registerCtr* können alle initialisierten Elemente zu den Listen hinzugefügt werden. Die beiden Methoden *notifyViews* und *notifyCtr* informieren alle Elemente über bestehende Änderungen, die dann jeweils mit der *update* Methode ausgerufen werden. Die virtuelle Methode *CalcModel* soll der Unterklasse *CModel* für weitere Berechnungen zur Verfügung gestellt werden.
Die Klasse *CModel* erbt aus *CAbstractModel*. In dieser Klasse sind alle Klassen, Methoden und Variablen enthalten, die für die Simulation erforderlich sind. Dazu gehören alle Instanzen von *CData*. In *CData* werden die Daten der Simulation gespeichert. Zusätzlich befinden sich in *CModel* die Instanzen von *CControllableObject*. Diese Klasse dient kurz zusammengefasst zur Überwachung der Controller. Für die Simulation des Tiefsetzstellers besitzt *CModel* die Klasse *CSolver* und einen Zeiger auf das abstrakte „BuckConverterModel“. Diese Klassen werden in den folgenden Kapiteln näher erläutert. Mit der Methode *CalcModel* können die Simulationsdaten berechnet und folgend in *CData* kopiert werden. Somit können die Views auf diese Daten für die Darstellung darauf zugreifen. Schließlich besitzt *CModel* einige Set-Methoden um dem Model alle Informationen aus den Editfeldern der Dialogklassen zu erhalten.

2) CAbstractView und konkrete Views

a) Übersicht aller Views

CAbstractViews dient als Mutterklasse aller Views. Es beinhaltet einen Zeiger auf das abstrakte Model. Zusätzlich besitzen die Views Variablen für sowohl die Position als auch für die Fenstergröße des Dialogfensters. Mit den beiden Variablen *xFactor* und *yFactor* können Skalierungen für die *draw()*-Methode durchgeführt werden. Somit können die Views beim beispielsweise Maximieren entsprechend der Fenstergrößenänderung vergrößert oder verkleinert werden.
Die konkreten Views wie beispielsweise *CCircuit* oder *CGraph* sind Unterklassen von *CAbstractView*. Diese Klassen besitzen jeweils eine eigene *draw()*-Methode, die jeweils auf dem Dialogfenster zeichnet. So wird beispielsweise der Schaltplan oder der Graphenverlauf gezeichnet.

b) Visualisierung der Datenwerte

Um die Ergebnisse, die mittels der Differentialgleichung berechnet werden, und die gegebenen Anfangsbedingungen für den Nutzer anschaulich zu gestalten, wird die Klasse *CGraph*

eingesetzt. Innerhalb der Datei *Graph.cpp* sind sämtliche grundlegenden Methoden enthalten, um die Koordinatensysteme zu zeichnen und die Verläufe der Ergebnisse übersichtlich darzustellen. Neben den erforderlichen Set-Methoden zur Anpassung der Größe von Werteverläufen und Koordinatensystemen entsprechend der aktuellen Fenstergröße, verfügt die Klasse über Funktionen zur Beschriftung der Achsen, zur Übertragung von Datenwerten in den Koordinatenraum sowie zur Zeichnung von Koordinatensystemen und Graphen.
Die Aufrufe dieser Funktionen erfolgen innerhalb der Klasse *CBuckConverterDLG*, wie im Abschnitt 4) beschrieben. Hier werden die aktuellen Fenstergrößenwerte an die *CGraph*-Klasse übergeben, um die Größenberechnung des Koordinatensystems durchzuführen und anzupassen.

3) CAbstractController und konkrete Controller

Die Controller besitzen grundsätzlich den gleichen Aufbau wie die Views. Sie funktionieren analog zu *CAbstractView* und den konkreten Views. So besitzt die Klasse *CAbstractController* beispielsweise ebenfalls einen Zeiger auf *CAbstractModel* und sämtliche Parameter für die Skalierung. Wichtig für die Funktion der Controller ist aber noch die Information über die ursprüngliche und der aktuellen Position. Diese Informationen werden in *CPoint* unter *org* und *act* gespeichert. Die maximale Breite oder Höhe der Controller werden über die Variablen *minPx* und *maxPx* gespeichert. Zusätzlich besitzt *CAbstractController* einen Zeiger auf *CControllableObject* Diese Klasse überprüft ob, sich die Controller darauf, dass sich die Werte von *act* innerhalb der Grenzen befindet. Dies passiert mit der Methode *void CControllableObject::setActualValue(double actValue)*. Um den Controller schließlich zu steuern, besitzt die abstrakte Controllerklasse noch die beiden Variablen *hotHeight* und *hotWidht*. Diese beiden Variablen sind der steuerbare Bereich der Controller. Die konkreten Controller übernehmen analog zu den Views die Methoden der abstrakten Klasse. Jeder Controller besitzt seine eigene *draw()*-Funktion und führt eigene Berechnungen (z.B. für die Induktivität) durch. Mit der Methode *calcActualValue* wird die aktuelle Position in Abhängigkeit der übergebenen Position berechnet.

4) BuckConverterDlg

Die Dialogklasse *BuckConverterDlg* ist mit dem Dialogfenster verknüpft. Die Klasse dient dazu eine Verbindung zwischen des Dialogfensters und der Steuerelemente und dem *CModel* zu erstellen. Dafür wird in *BuckConverterDlg* die Klasse *CModel* instanziiert. Zusätzlich werden in *BuckConverterDlg* alle Views und Controller instanziiert, die direkt mit dem Dialogfenster interagieren sollen. In dem Konstruktor von *BuckConverterDlg* bekommen alle Objekte ihre Startbedingungen. Es fällt auf, dass alle Views und Controller ihre Position nicht im Konstruktor übergeben bekommen. Standardmäßig werden alle Position auf (0,0) gesetzt. Die Positionen werden in der Methode *Buck_ConverterDlg::OnInitDialog()* gesetzt. Dies hat den Grund, dass in dieser Methode jeweils die individuelle Fenstergröße berechnet wird und immer die relative Position übergeben wird. Der Vorteil dieser ist, dass sich alle Views und Controller immer auf ihrer relativen Position befinden, unabhängig von der Bildschirmgröße- und Auflösung. Da die Anwendung auch im Vollbildmodus genutzt werden soll wird in der Methode *Buck_ConverterDlg::OnSize()* die geänderte Fenstergröße berechnet. Die berechnete Größe wird an alle Views und Controller übergeben, sodass eine entsprechende Größenskalierung durchgeführt werden kann.
Für die Anwendung der Controller werden die Methoden *Buck_ConverterDlg::OnMouseMove, ::OnLButtonDown, :: OnLButtonUp* verwendet. Diese interagieren mit dem Maus Input und rufen die entsprechenden Methoden auf. So wird beim Linksklick die Methode *OnLButtonDown* aufgerufen. Hier wird überprüft, ob ein Controller ausgewählt wurde. Wenn dies der Fall ist, wird ein Zeiger auf diesen Controller zurückgegeben und in *CAbstractController* AbstractController* gespeichert. Wenn die Maus nun bewegt wird, wird *OnMouseMove* ausgeführt. Es werden alle entsprechenden Methoden der Controller ausgeführt, die für die Berechnung notwendig sind. Beim Anheben der linken Maustaste wird *AbstractController* einem Nullzeiger übergeben.

II. Simulation

Ein wesentlicher Bestandteil des Programms ist die Simulation des Tiefsetzstellers. Zu diesem Zweck muss ein mathematisches Modell des Schaltkreises vorliegen, das dann numerisch gelöst werden soll. In diesem Kapitel werden die dafür benötigten elektrotechnischen Grundlagen dargelegt. Zudem wird die gewählte Simulationslogik vorgestellt, sowie alle damit verbundenen Modulklassen und Schnittstellen.

A. Elektrotechnische Grundlagen

Die allgemeine Funktion eines Tiefsetzstellers ist das Herabsetzen einer Eingangsspannung auf eine gewünschte, niedrigere Ausgangsspannung. Dies erfolgt mithilfe von Pulsweitenmodulation, also einer gezielten, hochfrequenten Betätigung eines Schalters. Der elektrische Schaltkreis des Systems ist in Abbildung 1 dargestellt.

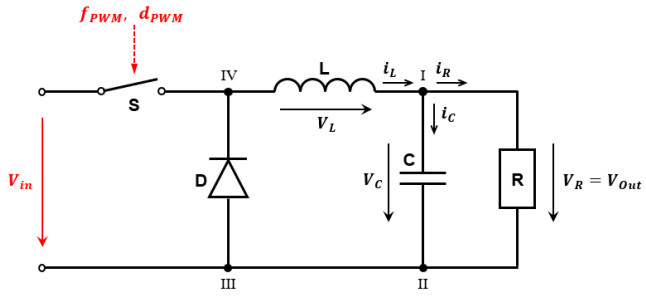


Abbildung 1: Tiefsetzsteller (Schaltkreis)

Der Schaltkreis besteht aus den beiden Energiespeichern Spule (L) und Kondensator (C), sowie einem Schalter S, einer Diode D und einem Verbraucher. In dem aufgeführten Schaltkreis ist der Verbraucher lediglich ein Widerstand R. Bei den rot markierten Größen handelt es sich um die Systemeingangsgrößen. Dazu zählen die anliegende Eingangsspannung V_{in}, sowie die Schaltfrequenz f_{PWM} und den Tastgrad d_{PWM}. Letztere Größe drückt das Verhältnis zwischen Ein- und Ausschaltdauer aus. Die wesentliche Systemausgangsgröße ist die Ausgangsspannung V_{Out}, welche in diesem Fall der Widerstandsspannung V_R entspricht.
Aus der Schalterdynamik resultiert, dass es zwei unterschiedliche Schaltkreiszustände gibt: In der Einschaltphase liegt die Eingangsspannung an und versorgt die Energiespeicher mit Strom. Die Induktivität der Spule hält die höhere Eingangsspannung vom Ausgang fern. In der Ausschaltphase liegt die Eingangsspannung nicht mehr an. Der Schaltkreis wird stattdessen über den Diodenzweig geschlossen. Der Verbraucher wird dann von den sich entladenden Energiespeichern mit Strom gespeist. Die Aufstellung der Gleichungen, die das Schaltkreisverhalten ausdrücken, erfolgt mithilfe der Kirchhoffschen Regeln. Im Folgenden werden der instationäre und der stationäre Zustand getrennt voneinander betrachtet.

1) Instationärer Zustand

Die äußere Schaltkreismasche besteht aus der Eingangsspannung V_{in}, der Ausgangsspannung V_{out} und der Spule L, durch die der Spulenstrom i_L fließt. Es kann die folgende Spannungsbilanz aufgestellt werden:

$$V_L = L\frac{di_L}{dt} = V_{in} - V_{out} \tag{1}$$

Aus der RC-Masche ergibt sich, dass die Ausgangsspannung $V_{out} = V_R$ mit der Kondensatorspannung V_C übereinstimmt. Am Knoten I (Abbildung 1) teilt sich der Spulenstrom i_L auf den Kondensatorstrom i_C und den Ausgangs- bzw. Widerstandsstrom i_R auf:

$$i_L = i_R + i_C = \frac{V_{out}}{R} + C\frac{dV_{out}}{dt} \tag{2}$$

Gleichung (1) kann nun nach V_{out} umgestellt und in Gleichung (2) eingesetzt werden, was zu einer Differentialgleichung zweiter Ordnung für den Spulenstrom i_L führt:

$$\frac{d^2i_L}{dt^2} = \frac{1}{LCR}V_{in} - \frac{1}{RC}\frac{di_L}{dt} - \frac{1}{LC}i_L \tag{3}$$

Es handelt sich also um ein schwingfähiges System. Alternativ kann eine Differentialgleichung für die Ausgangsspannung formuliert werden. Diese lautet dann:

$$\frac{d^2V_{out}}{dt^2} = \frac{1}{LC}V_{in} - \frac{1}{RC}\frac{dV_{out}}{dt} - \frac{1}{LC}V_{out} \tag{4}$$

Um die Ausschaltphase zu modellieren, müssen in den angeführten Gleichungen die Terme mit V_{in} zu null gesetzt werden. Alle Gleichungen stützen sich dabei auf die Annahme, dass die Diode keine Eigendynamik besitzt.

2) Stationärer Zustand

Im eingeschwungenen Zustand stellen sich keine statischen Endwerte für die Ausgangsspannung und den Strom ein, stattdessen führen diese Größen aufgrund des Ein- und Ausschaltens eine periodische Dauerschwingung aus. Der Schaltertastgrad d_{PWM} bestimmt dabei das Verhältnis zwischen Einschaltdauer t_{on} und Ausschaltdauer t_{off} und auch das zwischen der Eingangsspannung V_{in} und der mittleren Ausgangsspannung V_{out}:

$$d_{PWM} = \frac{V_{out}}{V_{in}} = \frac{t_{on}}{T_{PWM}} \tag{5}$$

Dabei ist T_{PWM} die Periodendauer des PWM-Signals:

$$T_{PWM} = t_{on} + t_{off} = \frac{1}{f_{PWM}} \tag{6}$$

Unter der Annahme einer konstanten Ausgangsspannung im stationären Zustand ist die Ableitung des Spulenstroms i_L gemäß Gleichung (1) ebenfalls konstant. Der Spulenstrom nimmt im stationären Zustand dementsprechend periodisch linear zu und ab, wie es in Abbildung 2 dargestellt ist.

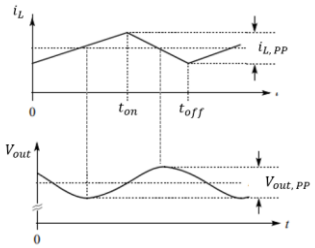


Abbildung 2: Ausgangsspannung und Spulenstrom im stationären Zustand (ETH Zürich, 2004)

Die stationäre Spulenstromschwingweite $i_{L,PP}$ beläuft sich infolgedessen auf:

$$i_{L,PP} = \frac{V_{in} - V_{out}}{L}t_{on} = \frac{V_{out}}{L}t_{off} \tag{7}$$

Für $V_{out,PP}$ gilt:

$$V_{out,PP} = i_{L,PP} \frac{T_{PWM}}{8C} \quad (8)$$

Eine ausführliche Herleitung von Gleichung (8) wird in (ETH Zürich, 2004) gegeben. Wie in Abbildung 2 erkannt werden kann, hat der stationäre Spannungsverlauf von V_{out} zum Zeitpunkt des Umschaltens einen Nulldurchgang. Die Spannungsamplituden werden dann jeweils nach der halben Einschalt- bzw. Ausschaltdauer erreicht. An diesen Stellen ist die Ableitung der Ausgangsspannung gleich null.

B. Klassen und Schnittstellen

Die in Abschnitt A dargelegten Gleichungen werden für Simulation des Tiefsetzstellers benötigt. Diese finden sich in der Modellklasse *CBuckConverterModel* wieder. Diese Klasse enthält darüber hinaus alle Modellparameter (Widerstand, Induktivität, …). Der Programmanwender hat die Möglichkeit, das Tiefsetzstellermodell sowohl im stationären als auch im instationären Zustand zu simulieren. Zu diesem Zweck wird in dem Simulationsmodul das Strategiemuster angewendet. Eine Übersicht über die Bestandteile liefert das folgende, vereinfachte Klassendiagramm (Abbildung 3). Die wichtigsten Klassenmethoden sind darauf rot markiert:

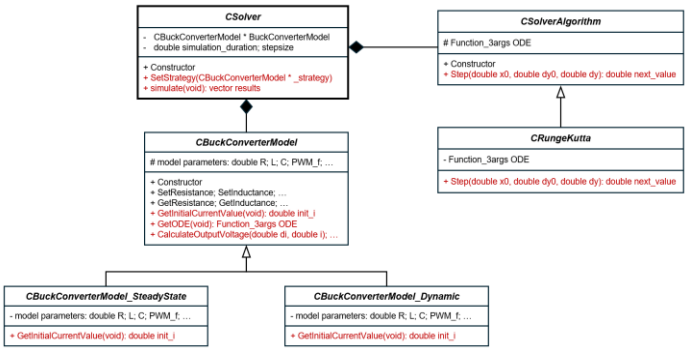


Abbildung 3: Klassendiagramm des Simulationsmoduls

Die Klassen *CBuckConverterModel_SteadyState* und *CBuckConverterModel_Dynamic* erben von der abstrakten Modellklasse, die bereits für beide Fälle die wichtigsten Informationen enthält. Für die Simulation des stationären und des dynamischen Verhaltens wird jeweils die gleiche Systemdifferentialgleichung (Abschnitt A.1)) verwendet. Der Unterschied der beiden Anwendungsfälle liegt im Startwert: Die dynamische Simulation geht von einer sprungförmigen Eingangsspannung aus. Dabei ist der Startwert *init_i* gleich null. Beim stationären Fall hingegen wird der Startwert so gewählt, dass sich Modell zum Simulationsstart bereits im eingeschwungenen Zustand befindet. Die Berechnung von *init_i* im stationären Fall erfolgt so, dass sich die Ausgangsspannung zu Beginn genau im „Tal“ befindet. Der Schalter hat dann gerade die Hälfte einer Einschaltdauer durchlaufen. Die Spannungsableitung ist zu dem Zeitpunkt gleich null (siehe Abbildung 2). Für die Berechnung dieses Startwerts wird Gleichung (8) herangezogen. Die Methode *GetInitialCurrentValue* gibt je nach Fall den entsprechenden Wert zurück, von dem dann die Klasse *CSolver* Gebrauch macht.

Die Tiefsetzstellermodellklassen verfügen über die Methode *GetODE*, welche als Rückgabewert die vollständige Differentialgleichung liefert. Dafür wird zunächst innerhalb der Klasse mithilfe der Vorlage *‘std::function‘* der neue Typ *Function_3args* definiert. Es handelt sich dabei um eine Lambda-Funktion, die drei Argumente vom Typ *double* entgegennimmt und selbst auch einen *double*-Rückgabewert ausgeben kann. Die Differentialgleichung, die durch Gleichung (3) beschrieben wird, ist nämlich eine Funktion von den drei Argumenten: Zeit, Stromableitung und Strom und berechnet mit diesen Größen die zweite Stromableitung. Die Lambda-Funktion nimmt in der Methode *GetODE* die Modellparameter an und wird dann ausgegeben.
Für die Lösung der Differentialgleichung wird das Runge-Kutta-Verfahren angewendet. Der entsprechende Algorithmus befindet sich in der gleichnamigen Klasse *CRungeKutta*. Diese erbt von der Klasse *CSolverAlgorithm*, die den Member vom Typ *Function_3args* definiert. Sie macht also vom gleichen Typ Gebrauch, der auch in der Modellklasse definiert ist. Die virtuelle Methode *step* greift den Member (die Differentialgleichung) auf und führt damit einen numerischen Integrationsschritt aus. Dafür nimmt die Methode die drei Integrationsstartwerte *x0*, *dy0* und *dy* entgegen. Der Integrationsalgorithmus ändert sich dann in den abgeleiteten Klassen, je nachdem, welcher Solver verwendet werden soll.
Der Berechnungsalgorithmus und das Modell werden in der Klasse *CSolver* zusammengeführt. *CSolver* stellt den Klienten des hier angewendeten Strategiemusters dar: Die Klasse beinhaltet einen Zeiger auf die abstrakte Modellklasse und kann mit der Methode *SetStrategy* zwischen stationärer und instationärer Simulation wechseln. Bei dem Wechsel werden die Modellparameter der ursprünglichen Modellinstanz auf die neue übertragen. Die Methode *simulate* führt die Simulation aus, die durch das mathematische Modell, den Berechnungsalgorithmus, die Simulationsdauer und die numerische Schrittweite definiert wird.

III. LITERATURVERZEICHNIS

ETH Zürich. (April 2004). Von https://ethz.ch/content/dam/ethz/special-interest/itet/power-electronic-systems-lab/images/Education/lab-courses/power-electronics/IE1-Tiefsetzsteller-Script.pdf abgerufen

Probst, U. (2023). *Objektorientiertes Programmieren "Eine Einführung für die Ingenieurwissenschaften in C++".* München: HANSER.

IV. Anhang

A. UML-Diagramm

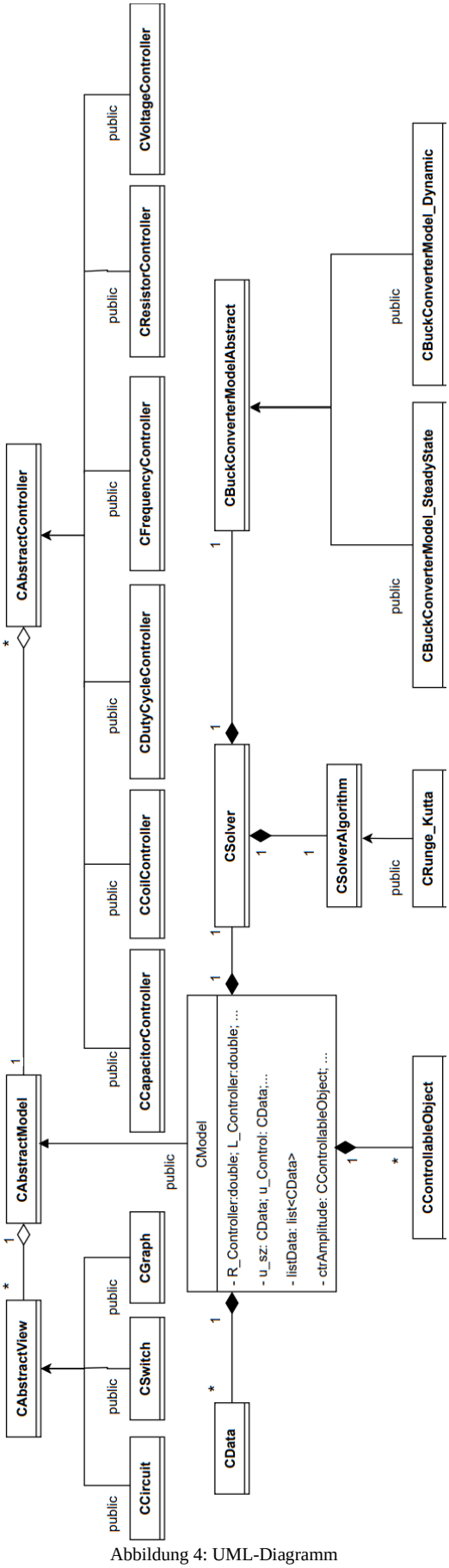
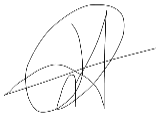


Abbildung 4: UML-Diagramm

B. Eidesstattliche Erklärung

Wir erklären hiermit, dass wir die vorliegende Arbeit selbstständig durchgeführt und verfasst haben. Dazu haben wir keine anderen als die angeführten Behelfe verwendet und wissenschaftliche Texte oder Daten nicht unbefugt verwertet. Die Arbeit ist in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung.

Gießen, 08.04.2024	
Ort, Datum	Philipp Blum
Gießen, 08.04.2024	
Ort, Datum	Leonard Hisgen
Gießen, 08.04.2024	
Ort, Datum	Adrian Lauster
Gießen, 08.04.2024	
Ort, Datum	Lukas Rother