Claudia Pröll, BSc

Automation of a Flow Calorimetry for Chemical Reaction Optimization

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Verfahrenstechnik

submitted to

Graz University of Technology

Supervisors

Assoc.Prof. Dipl.-Ing. Dr.techn. Heidrun Gruber-Wölfler
Dipl.-Ing. Sebastian Soritz, BSc
Institute of Process and Particle Engineering

Graz, September 2021

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date, Signature

Abstract

In this thesis, a laboratory system with a heat flow calorimeter as the core element is automated. Reaction calorimeters are used in the field of chemical and pharmaceutical process development. As a measuring device for the amount of heat absorbed or released in a chemical reaction, they provide essential information about the reaction that has taken place. The automatisation is implemented by means of the widely used programming language Python. The created implementation is divided into three basic main parts. The first part comprises the device drivers, with the help of which each device in the process chain can be controlled and operated. The second part is given by the individual strategies, which have detailed information on the sequence control. The strategies enable the individualised usage of the laboratory system. The last part of the implementation refers to the element of the implementation that is capable of executing the strategies. Meaning, this part ensures that the controlling of the system is implemented. Finally, in order to demonstrate the functionality of the created application, neutralisation experiments were conducted at different molarities. In addition to the successful execution of the application, the characteristics of the results correspond to the expectations. As the concentration and flow rate increase, the measured values approach the literature value given for the experiment. Overall, the developed application can be used successfully in the laboratory, the individual elements can be used independently of each other and also for other purposes, and due to the application architecture, the future extension with further functions does not require a great deal of effort.

Kurzfassung

Reaktionskalorimeter werden im Bereich der chemischen und pharmazeutischen Verfahrensentwicklung eingesetzt. Als Messgerät für die aufgenommene oder abgegebene Wärmemenge einer chemischen Reaktion liefern sie wesentliche Informationen über die stattgefundene Reaktion. Im Rahmen dieser Arbeit wird eine Laboranlage mit einem Wärmestromkalorimeter als Kernelement automatisiert. Diese Automatisierung wird mittels der verbreiteten Programmiersprache Python implementiert. Die erstellte Implementierung gliedert sich in drei grundlegende Hauptteile. Der erste Teil umfasst die Gerätedriver, mit deren Hilfe jedes in der Prozesskette auftretende Gerät angesteuert und bedient werden kann. Den zweiten Teil bilden die individuellen Strategien, welche über detaillierte Informationen zur Ablaufsteuerung verfügen. Durch diese Strategien wird der individualisierte Einsatz der Laboranlage möglich. Der letzte Teil bezieht sich auf jenen Teil der Implementierung, der dazu im Stande ist, die Strategien auszuführen. Das heißt, der dritte Teil sorgt dafür, dass die Steuerung der Anlage umgesetzt wird. Um die Funktionalität der erstellten Applikation zu zeigen, wurden abschließend Neutralisationsversuche bei verschiedenen Konzentrationen durchgeführt. Zusätzlich zur erfolgreichen Ausführung der Applikation, entspricht das Verhalten der Ergebnisse den Erwartungen. Mit steigender Konzentration und steigender Durchflussrate nähern sich die Messwerte dem für den Versuch gegebenen Literaturwert an. Die entwickelte Applikation kann im Allgemeinen erfolgreich im Labor eingesetzt werden, die einzelnen Elemente können unabhängig voneinander und auch für andere Aufgaben genutzt werden und aufgrund der Applikationsarchitektur ist die zukünftige Erweiterung um weitere Funktionen nicht mit großem Aufwand verbunden.

Acknowledgement

I would like to express my very great appreciation to Assoc.Prof. Dipl.-Ing. Dr.techn. Heidrun Gruber-Wölfler for her support during this thesis and for giving me the opportunity to write a thesis in her research group, as well as to Dipl.-Ing. Sebastian Soritz, BSc for his support and advice during the thesis and for the many helpful tips and suggestions. I would also like to thank my family, partner and friends who have always been supportive throughout my studies.

Contents

Abstract I

Zusammenfassung II

Acknowledgement III

List of Figures VI

List of Listings VII

List of Tables IX

List of Symbols X

1 Introduction 1
- 1.1 Task Formulation and Intended Use . . . 1
- 1.2 Structure of the Thesis . . . 1

2 Theoretical Background 3
- 2.1 Flow Chemistry . . . 3
- 2.2 Basic Principles of Calorimetry . . . 4
- 2.3 Neutralisation Reaction . . . 6
- 2.4 Process Setup and Communication Details . . . 8
- 2.5 State Machine . . . 13
  - 2.5.1 Basic Concept of the State Machine . . . 13
  - 2.5.2 Elements of the State Machine . . . 14
  - 2.5.3 Example of a State Machine . . . 15
- 2.6 Design pattern . . . 16
  - 2.6.1 Definition of design pattern . . . 16
  - 2.6.2 Factory method pattern . . . 17
  - 2.6.3 State pattern . . . 18
  - 2.6.4 Strategy pattern . . . 19

3 Application Development 21
- 3.1 Basic Principles of Application Development . . . 21
- 3.2 Use Case Specification . . . 22
- 3.3 Application Architecture . . . 23
- 3.4 Implementation . . . 25
  - 3.4.1 Implementation Approach . . . 25

3.4.2 Realisation of the State Machine Concept 25
3.4.3 Creating a State Machine 32
3.4.4 Additional State Machines of the Equipment 38
3.4.5 Realisation of the Strategy Pattern 43

4 Results and Discussion 51
4.1 Testing of the Application 51
4.2 Final Application 52
4.3 Experimental Results 57

5 Conclusion and Outlook 61

6 Experimental Procedure 63
6.1 Details on conducting the Experiments 63
6.2 Calorimeter-Thermostat Calibration 65
6.3 Calorimeter Calibration 66

7 References 68

8 Appendix 70
8.1 Application Code 70
8.1.1 Auto.py 70
8.1.2 Calibration.py 74
8.1.3 Calorimeter.py 75
8.1.4 Communication.py 77
8.1.5 Dictionary.py 78
8.1.6 Excel_Functions.py 78
8.1.7 Fisher.py 79
8.1.8 HPLC.py 85
8.1.9 Lambda.py 92
8.1.10 LayerB.py 96
8.1.11 LayerC.py 97
8.1.12 Operating_OCAE.py 98
8.1.13 Operating_OPL.py 99
8.1.14 pyState.py 99
8.1.15 pyStrategy.py 100
8.1.16 Strategy_OCAE.py 101
8.1.17 Strategy_OPL.py 104
8.2 Output Calculation from Measurement Data 106

List of Figures

1 Standard setup of a two substance system used for flow chemistry. [7] 4
2 Typical heat flow reaction calorimeter. 6
3 Devices used for the automatisation application. 8
4 Communication frame associated with serial interface RS-232. 8
5 Basic elements and their notation in state diagrams. 13
6 State diagram describing a possible measuring procedure in the laboratory. . . 15
7 Structure of the *Factory Method Pattern*. 18
8 Structure of the *State Pattern*. 19
9 Structure of the *Strategy Pattern*. 20
10 Four steps of application development. 21
11 Flowchart illustrating the application architecture and its participants. 24
12 Resulting state machine for the thermostat driver. 33
13 Resulting state machine for the composite *Configuration* state of the Fisher thermostat. 34
14 Resulting state machine for the composite *Deactivated* state of the Fisher thermostat. 35
15 Resulting state machine for the composite *Activating* state of the Fisher thermostat. 36
16 State diagram of the composite layer B state *Send_And_Check*. 37
17 State diagram of the HPLC pump driver. 39
18 State diagram of the Lambda pump driver. 41
19 State diagram of the calorimeter driver. 42
20 State diagram of the context as part of the *Strategy Pattern*. 46
21 Worksheets of the output excel file. 55
22 Relative errors of the determined molar reaction enthalpy from the first experiment at a temperature of 25°C using a concentration of 1.65 mM. 57
23 Determined molar reaction enthalpy from the second and third experiment at a temperature of 25°C using a concentration of 1 M and 2 M. 58
24 *Evaluation* worksheet containing the results of experiment 3a. 59
25 *Dia_Raw_Temp* worksheet displaying the plot of measured temperatures of experiment 3a over time. 60
26 Calibration curve between thermostat and calorimeter in the interval of 25 to 40°C. 66
27 Calibration curve to convert measured voltage into heat quantity for the temperature combination of 25°C at the calorimeter and 26°C at the thermostat. . 67

List of Listings

1 Specification of the state template. 26
2 Factory template that must be present and adhered to when implementing any state machine. 28
3 Specification of the relevant functions of the engine class. 29
4 State machine template for a composite state. 31
5 State machine template if the created state machine is not a composite state. . 31
6 Basic strategy from which any additional strategy can be built. 44
7 Template for the list entries of the operating point list. 48
8 Specifications needed for the execution of the OPL strategy. 53
9 Specifications needed for the execution of the OCAE strategy. 54
10 Initialisation and Calling of the context class. 54
11 The *Auto.py* file corresponds to the context of the strategy pattern and is therefore responsible for the execution of the individual strategies. 70
12 The *Calibration.py* file contains the calibration equation for the pumps, calorimeter and calorimeter-thermostat combination. 74
13 The *Calorimeter.py* file contains its corresponding device driver and can be used to operate this device. 75
14 In the *Communication.py* file, the library for serial communication available in Python is adapted for own purposes. 77
15 The *Dictionary.py* file contains some basic interchangeable parameters. 78
16 The *Excel_Functions.py* contains the function, which is responsible for the setup of the basic output file. 78
17 The *Fisher.py* file contains its corresponding device driver and can be used to operate this device. 79
18 The *HPLC.py* file contains its corresponding device driver and can be used to operate this device. 85
19 The *Lambda.py* file contains its corresponding device driver and can be used to operate this device. 92
20 The *LayerB.py* file contains several state classes, which are in general more complex than layer C states. 96
21 The *LayerC.py* file contains simple state classes. 97
22 The *Operating_OCAE.py* file is used to execute the *Auto.py* file using the *Output Calculation Absolute Evaluation* strategy. 98
23 The *Operating_OPL.py* file is used to execute the *Auto.py* file using the *Operation Point List* strategy. 99

24 The *pyState.py* file contains the basic state class and the engine class. Both classes are used later when creating a state machine. 99
25 The *pyStrategy.py* file contains the basic strategy class. 100
26 The *Strategy_OCAE.py* file corresponds to a concrete strategy of the strategy pattern and contains the strategy for evaluating the measurement data and for creating an Excel output file. 101
27 The *Strategy_OPL.py* file corresponds to a concrete strategy of the strategy pattern and contains the strategy which does not yet further restrict the handling of the system. 104

List of Tables

1 Serial communication protocol of the Fisher thermostat. 10
2 Serial communication protocol of the HPLC pump. 11
3 Serial communication protocol of the Lambda pump. 12
4 Concentrations of the prepared solutions for the various experiments in mol·l^{-1}. 63
5 Pumps used for the various experiments. 64
6 Parameters of the quadratic calibration curve for various temperature combinations of calorimeter and thermostat. 67

List of Symbols

ρ	density	$g{\cdot}ml^{-1}$
A	heat transfer area	m^2
c	concentration	$mol{\cdot}l^{-1}$
c_A	concentration of component A	$mol{\cdot}l^{-1}$
c_B	concentration of component B	$mol{\cdot}l^{-1}$
$c_{p,water}$	specific heat capacity of water	$J{\cdot}mol^{-1}{\cdot}K^{-1}$
c_{water}	concentration of water	$mol{\cdot}l^{-1}$
ΔH°	standard enthalpy of reaction	$kJ{\cdot}mol^{-1}$
Δh_R	molar reaction enthalpy	$kJ{\cdot}mol^{-1}$
K_i	calibration value of the pump	
M	molar mass	$g{\cdot}mol^{-1}$
m	mass	g
N	number of bits	
$\dot{n}_{A,act}$	molar flow rate of component A	$mol{\cdot}s^{-1}$
$\dot{n}_{B,act}$	molar flow rate of component B	$mol{\cdot}s^{-1}$
Q_{flow}	heat flow	W
$\dot{Q}_A$	convective heat flow of component A at the inlet	W
$\dot{Q}_B$	convective heat flow of component B at the inlet	W
$\dot{Q}_{conv}$	convective heat flow	W
$\dot{Q}_{out}$	convective heat flow at the outlet	W
$\dot{Q}_{pre}$	transmitted heat flow at the precooling element	W
$\dot{Q}_{r1}$	transmitted heat flow at the first reactor	W
$\dot{Q}_{r2}$	transmitted heat flow at the second reactor	W
$\dot{Q}_{reac}$	reaction heat flow	W
$\dot{Q}_{seg}$	transmitted heat flow at a segment	W
$\dot{Q}_{tran}$	transmitted heat flow	W
T_A	inlet temperature of component A	K
T_B	inlet temperature of component B	K
T_j	jacket temperature	K
T_n	symbol duration time	s
T_{out}	outlet temperature	K
T_r	reactor temperature	K
T_{set}	set temperature at the calorimeter	K
ΔT_A	temperature difference at the inlet of component A	K

ΔT_{B}	temperature difference at the inlet of component B	K
ΔT_{out}	temperature difference at the outlet	K
U	thermal heat transfer coefficient	$W{\cdot}m^{-2}{\cdot}K^{-1}$
U_{pre}	measured voltage at the precooling element	mV
U_{r1}	measured voltage at the first reactor	mV
U_{r2}	measured voltage at the second reactor	mV
U_{seg}	measured voltage at a segment	mV
V	volume	ml
$\dot{V}_{A}$	volumetric target flow rate of component A	$ml{\cdot}min^{-1}$
$\dot{V}_{A,act}$	volumetric flow rate of component A	$ml{\cdot}min^{-1}$
$\dot{V}_{B}$	volumetric target flow rate of component B	$ml{\cdot}min^{-1}$
$\dot{V}_{B,act}$	volumetric flow rate of component B	$ml{\cdot}min^{-1}$

1 Introduction

1.1 Task Formulation and Intended Use

Chemical reactions can be characterised by the energy released or absorbed in the form of heat. Knowledge of this heat is essential, for example, when scaling up reactions for large-scale plants. Furthermore, since this heat is directly related to the reaction rate, it provides information about the kinetics of a chemical reaction. For these reasons, reaction calorimeters are of great importance and have long been used for these purposes due to the simplicity of measuring heat quantity. If in addition the calorimeter and its equipment are to be automated, besides simplifying the work in the laboratory, this will also facilitate the reaction screening of various reactions.

In the context of this thesis, an automatisation tool in the programming language Python for a system consisting of several pumps, a thermostat and a reaction calorimeter is to be designed and implemented. For the reaction calorimeter, the heat flow calorimeter developed by MAIER et al. is to be used [1]. The objective of the development is to generate an application for the sequence control, which runs autonomously and includes error handling, starting from automatic setting of basic settings to the processing of specified operating points up to the evaluation of the measurement data received. The essential parts of the application and therefore this thesis are the subroutines which can control a single device, and also those which combine these individual subroutines to obtain the sequence control. Finally, the functionality of the application is to be demonstrated by conducting a neutralisation reaction.

The purpose of this sequence control, as mentioned earlier, is to facilitate reaction screening. Associated with this, the application is to be programmed in such a way that an optimisation algorithm can be easily integrated. By evaluating the measurement data, the optimisation algorithm could determine the specifications for the next operating points of the reactions to be conducted. Thus, the optimal reaction conditions would not only be determined by the optimisation algorithm, but also directly applied by the automatisation application.

1.2 Structure of the Thesis

This thesis is divided into six sections. Following the first section *Introduction*, the second section *Theoretical background* provides the theoretical background relevant for this work. In this respect, general aspects such as flow chemistry, reaction calorimetry and neutralisation reactions are discussed. This is followed by a description of elements which are decisive for the implementation, such as the serial interface, the individual devices and the underlying programming principles.

Subsequently, the process of designing and implementing the application is described in the third section *Program development*. At first, the use cases are defined, which are determined based on the task formulation and consideration of how the system will be operated. Based on these use cases, the architecture of the automatisation application is planned. Finally, the implementation is described.

In the fourth section *Results and discussion*, firstly the testing and the functionality of the final application is described, and secondly the results of the neutralisation experiments conducted as prove of concept are presented and discussed. The fifth section *Conclusion and outlook* contains a summary of the entire work and an outlook on what possible extensions can be made. The sixth section *Experimental procedure* provides additional information on the experimental procedure, concerning the equipment and materials used and various calibration curves. Furthermore, in the Appendix the implementation of the application can be looked up.

2 Theoretical Background

2.1 Flow Chemistry

Flow chemistry deals with conducting a chemical process using continuous flow instead of a batch process. The application of continuous flow goes hand in hand with the current trend in pharmaceutical industry to use microreactors or microreactor systems. The combination offers advantages over conventional processes such as better mixing, better heat and mass transfer as well as easier process control. [2, 3]

Discontinuous mode or batch mode refers to the process in which the required starting materials are fed to the apparatus at the beginning. The system proceeds until the desired degree of processing is reached. Subsequently, the container is completely emptied. A new process cycle is started by refilling the cleaned container. This mode of operation is common when a product is only required in small quantities, as is the case in the pharmaceutical sector. The advantage of this approach is that the reactor can be used for different products and thus offers a high degree of flexibility. However, the dead times during filling and emptying, higher energy costs, increased work effort and the varying product quality are shortcomings of this operation mode. The opposite of the batch process is the continuous mode of operation. The input of new material and discharge of products take place continuously. This mode of operation overcomes the disadvantages of the discontinuous mode of operation. It eliminates dead times, the operating costs are lower and the product quality is more consistent. [4, 5]

In the pharmaceutical industry, it used to be common to take advantage of the flexibility of the discontinuous operation mode and thus to manufacture different substances one after the other using a batch reactor. The production of a drug, starting from the synthesis of the active pharmaceutical ingredient (API) up to the manufacturing of the dosage form, consists of a process chain of many individual batch steps. Since dead times arise for each of these batch steps due to filling and emptying times, the time lost adds up. Furthermore, the individual processing steps do not always take place at the same location, which leads to costly intermediate storage and transport. For this reason, the trend in the pharmaceutical sector is towards continuous operation, i.e. the application of flow chemistry. In addition to eliminating dead times occurring in batch, the continuous approach makes it possible to combine several process steps, such as synthesis and purification. The aim is an end-to-end process which covers the entire process from the starting material to the drug as the final product. [5, 6]

The realisation of a chemical process by means of continuous flow and the application of microreactor systems is specific to each process. Figure 1 illustrates the generalised setup of a standard two-feed continuous flow system, which can be divided into six segments. The first segment consists of two pumps delivering the reagents to the reactor. Prior to the reactor, the reagents are mixed with each other. After the reactor unit, a quenching module and a unit to regulate the pressure follow. Finally, the products are collected in a container. Other optional segments to be integrated into the system are analysis tools or purification steps. [7]

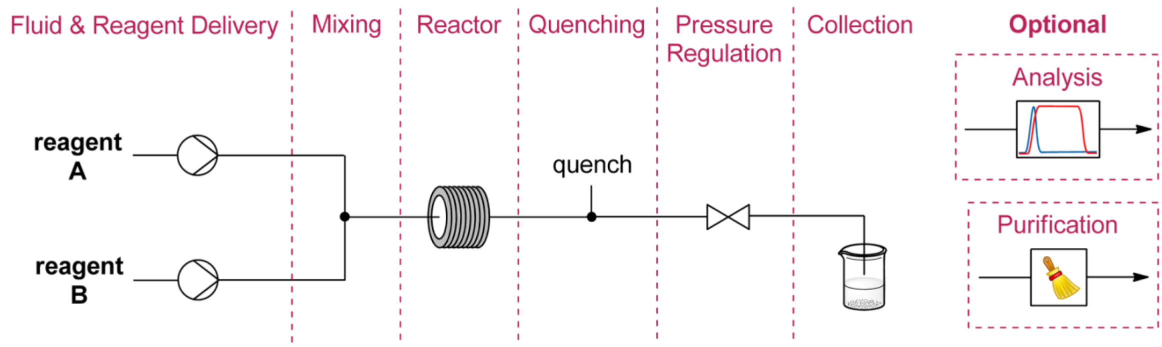


Figure 1: Standard setup of a two substance system used for flow chemistry. [7]

An additional advantage of continuous operation mode is a simple automatisation of the system. Automatisation in this context means that once the system has been set up and the preliminary experimental preparations are completed, the system runs independently. Meaning, the manual work steps during the runtime of the experiment are replaced by the automatisation. As the system is automated, the possibility of automated reaction optimisation arises as well. Reaction optimisation aims to maximise product yields and to generate kinetic reaction models while minimising the total number of experiments conducted. [8]

2.2 Basic Principles of Calorimetry

Calorimetry is defined as the quantitative measurement of heat and has been used since the 18th century. Consequently, a variety of methods have been developed, which differ in their measurement and control principles or in their operation mode. [9, 10]

Regarding the different measurement methods, a distinction is made between compensation of thermal effects, measurement of temperature differences and temperature modulation. For the compensation method, the temperature differences generated by the reaction are avoided. To do this, the heat must be either supplied or dissipated accordingly. This can be achieved, for example, by a PELTIER element. The electric current passing through the PELTIER element provides information about the amount of compensated heat. For the method of measuring temperature differences, the amount of heat is derived from the measured difference. A distinction is made between temporal and spatial method. The former measures the difference in terms of time before and after the reaction and the latter measures the difference locally at specific points in the calorimeter. The last measurement method is that of temperature modulation. Here, the aim is to determine the amount of heat required for a periodically given temperature profile. [10]

The second type of calorimeter classification, the different operation methods, concerns the way of controlling the reaction temperature. The methods are divided into isothermal, adiabatic, isoperibolic and temperature-programmed. In the isothermal operation method, the reaction temperature is kept constant, which goes hand in hand with the measurement method based on the compensations of thermal effects. In the adiabatic operation method, the cooling or heating temperature is adjusted to minimise the heat exchange between the medium and the reactor medium. In the isoperibolic operation method, the cooling or heating temperature is kept constant while the temperature change of the reactor medium is measured. The temperature-programmed operation method is related to the temperature modulation measurement method in so far as that the reaction temperature is varied with respect to a given profile. [11]

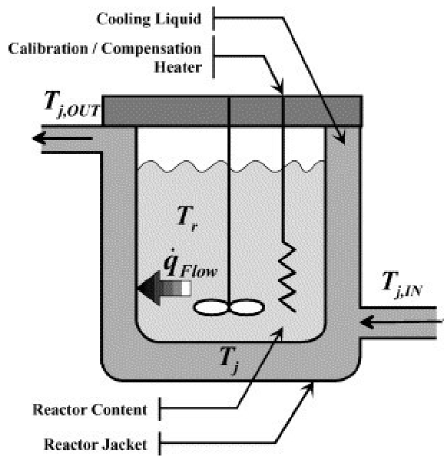
In the context of this thesis, a reaction calorimeter with an isothermal mode of operation is used. A chemical reaction is generally linked to the release or uptake of heat. Therefore, measuring heat flux is a common method to characterise those reaction processes and is referred to as reaction calorimetry. In the case of reaction calorimetry, a distinction can be made between four established methods: heat-flow reaction calorimeter, heat-balance reaction calorimeter, power-compensation reaction calorimeter and Peltier calorimeter. [11, 12] Since the heat flow calorimeter corresponds to the method of the calorimeter used, only this will be discussed in more detail. Figure 2 shows a typical heat flow reaction calorimeter and a schematic drawing of its reactor.

For the heat flow calorimeter using isothermal operation mode, the following balance is valid:

$$Q_{\mathrm{flow}} = U \cdot A \cdot (T_{\mathrm{r}} - T_{\mathrm{j}}) \tag{1}$$

Whereby Q_{flow} refers to the heat flow (W), U to the thermal heat transfer coefficient ($\mathrm{W{\cdot}m^{-2}{\cdot}K^{-1}}$), A to the heat transfer area ($\mathrm{m^2}$), T_{r} to the reactor temperature (K) and T_{j} to the jacket temperature (K). The principle of determining the heat flow is based on measuring the temperature difference and converting it by means of a calibration factor. This means that the thermal heat

(a) Mettler RC1mx heat flow reaction calorimeter [13].



(b) Schematic drawing of a heat flow reaction calorimeter [11].

Figure 2: Typical heat flow reaction calorimeter.

transfer coefficient and the heat exchange surface are combined into one factor, which must be known in order to evaluate the measurement. [12]

It is relevant how the temperature difference ($T_r - T_j$) is measured and how the reactor temperature T_r is kept constant. Thermocouple elements, which are based on the SEEBECK effect, serve as the basis for the measurement of the temperature difference. A local temperature difference generates a heat flow. The heat flow causes a voltage in the thermocouple, which is then measured. Using a calibration curve between the measured voltage and the amount of heat, the actual amount of heat can be determined. Thermocouples are also used as the basis for keeping the temperature constant, this time based on the PELTIER effect. An electric current, which is adjusted via the voltage, generates a heat flow. By means of a control unit, the voltage can be adjusted in such a way that the exact amount of heat is produced which is necessary to keep the reactor at the desired temperature. [1, 12]

2.3 Neutralisation Reaction

A neutralisation reaction refers to the reaction in which an acid and a base react with each other in an aqueous solution. The products are water and the corresponding salt from the remaining base ions and acid residue ions. The resulting pH-value of the reaction product depends on the acid and base used. [14]

The starting point is a weak or strong acid and base, each of which is put into aqueous solution. On contact with water, the acid or base begins to dissociate, whereby only strong acids or strong bases dissociate completely. In the case of the acid HA, this process results in anions A^- and oxonium ions H_3O^+ (Equation 2), and in the case of the base B, in cations B^+ and hydroxide ions OH^- (Equation 3).

$$HA + H_2O \rightleftharpoons A^- + H_3O^+ \tag{2}$$

$$B + H_2O \rightleftharpoons B^+ + OH^- \tag{3}$$

In the neutralisation reaction, hydrogen protons react with hydroxide ions. Therefore, independent of the starting materials, the same chemical reaction takes place, namely the formation of water (Equation 4). In connection with this, the occurring neutralisation heat of $\Delta H^\circ = -57.4\ \text{kJ} \cdot \text{mol}^{-1}$ is always the same [15].

$$H^+ + OH^- \rightarrow H_2O \tag{4}$$

The overall neutralisation reaction can be expressed in the following way:

$$\textit{acid} + \textit{base} \rightarrow \textit{salts} + \textit{water} \tag{5}$$

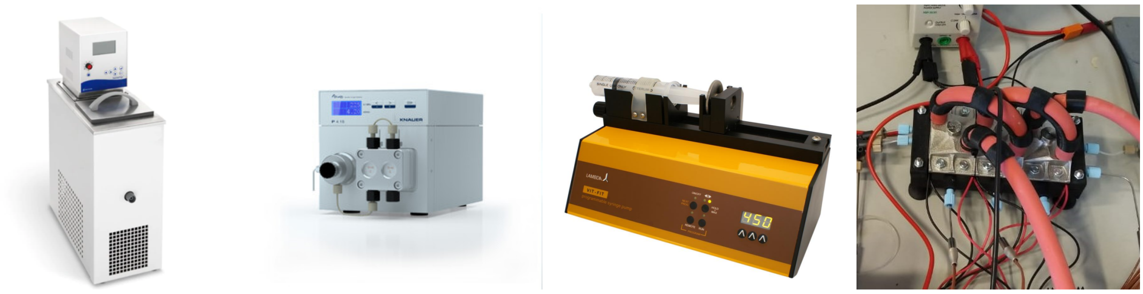
As already mentioned, the resulting pH value depends on the acid and base initially used. If a strong acid and a strong base are the starting materials of the neutralisation reaction, there is approximately the same amount of oxonium ions and hydroxide ions in solution. Consequently, there is a neutralisation of the pH value, which only assumes minimal deviations around 7. The behaviour is different when a weak acid is combined with a strong base or a strong acid with a weak base. The weaker the substance, the worse it generally dissociates and leading to an imbalance of oxonium ions and hydroxide ions in aqueous solution. For the weak acid and the strong base, hydroxide ions remain in the product, which moves the pH value into the basic range. The opposite is the case for the combination of a strong acid and a weak base. Here, oxonium ions remain in solution and the pH value is shifted to the acidic range.

In the context of this thesis, acetic acid, a weak acid, and sodium hydroxide, a strong base, are used as starting materials for the neutralisation reaction. Due to the properties of these substances, the pH value is only minimally shifted to the basic range depending on the initial concentration and thus can be used for the prove of concept of the sequence control program. The corresponding reaction equation for an acetic acid - sodium hydroxide neutralisation reaction is given in the Equation 6.

$$NaOH + CH_3COOH \rightarrow CH_3COONa + H_2O \tag{6}$$

2.4 Process Setup and Communication Details

Figure 1 illustrates a common system setup that could be found in flow chemistry. The system that is to be automated in the context of this thesis follows exactly this setup, although not all elements are electronic and can therefore not be addressed. The pumps at the beginning of the plant and the use of pumps for the optional quenching are crucial for the automatisation. Unlike in the illustration, the number of pumps is variable. Another element for automatisation is the reactor. In the context of this thesis, a reaction calorimeter is used, which supplies measurement data. Additionally, a thermostat is needed to control the temperature at the calorimeter, which is not shown separately in Figure 1. All the equipment used (thermostat, two different types of pumps and calorimeter) is shown in Figure 3.



(a) Thermostat from Fisher Scientific [16]. (b) HPLC pump from Knauer [17]. (c) Syringe pump from LAMBDA Laboratory Instruments [18]. (d) Heat flow calorimeters [1].

Figure 3: Devices used for the automatisation application.

For communicating with devices in the context of this thesis, the RS-232 serial interface is of primary importance. The transmitted message is formed by several elements, whereby the entire construct is referred to as the communication frame and is given in Figure 4. From this figure it is evident that the message consists of four elements: start bit, data frame, parity and stop bit. The start and stop bits initiate and terminate the message, the data frame contains

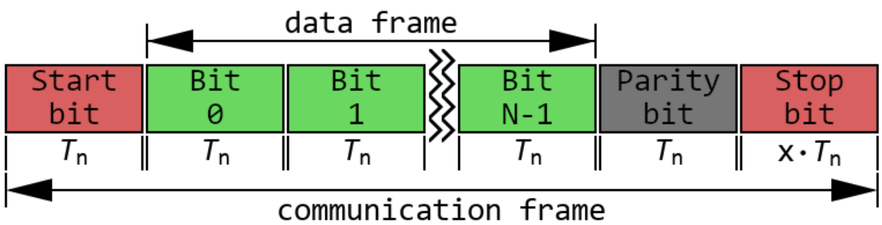


Figure 4: Communication frame associated with serial interface RS-232.

the actual message and the parity serves as an optional check in which the bits of the sent message are counted. [19]

In order to be able to send the message, a connection between the communicating devices must first be established. For this purpose, the following communication parameters are necessary, which also include details of the individual elements of the communication frame [19]:

- *Port name*: The operating system assigns a name to each serial communication port, e.g. COM1.
- *Baud rate*: The baud rate indicates the number of data bits sent per second. It refers to the number of bits of the overall communication frame and not the number of bits of the data frame.
- *Parity*: The parity check counts the number of high bits. When specifying the parity bit, the bit that completes the total number of all bits to even or odd is passed. Alternatively, it is possible that no parity check is carried out.
- *Data bits*: This specifies the number of bits N available for the data frame. The number is usually 8.
- *Stop bits*: The reciprocal of the baud rate results in the symbol duration time T_n. When specifying the stop bit, the length of this bit is given as a multiple of the symbol duration time.

For the later planning of the application architecture and its implementation, it is crucial to have detailed information about the individual devices. The necessary information includes the available functions and the communication protocol, which are specified below for each individual device.

Fisher Thermostat

The thermostat used is the Fisherbrand™ Isotemp™ R20 Refrigerated and Heated Bath Circulators from Fisher Scientific (Figure 3a). All necessary information about the device is taken from the corresponding device manual. [16]

The following settings for the communication parameters are available for the serial interface between host PC (master) and thermostat (slave):

- *Serial comm* RS-232, RS-485, Off
- *Baud* 19200, 9600, 4800, 2400, 1200, 600, 300
- *Parity* None, Odd, Even

- *Data bits* 8
- *Stop bits* 1, 2

In Table 1 all functions of the thermostat used within the implementation are specified. Additionally, the communication protocol is evident from this. The slave returns a response to every command sent by the master. The master can only send a new command when the master has received the response. Each command sent and its corresponding response are terminated with a carriage return.

Table 1: Serial communication protocol of the Fisher thermostat.

Command description	**Master command**	**Slave response**
Read displayed setpoint	RS	[*Value*]C*
Read external probe enabled	RE	[*Binary value*]
Read pump speed	RPS	[*String value*]**
Read temperature unit	RTU	[*String value*]***
Read unit on	RO	[*Binary value*]
Set displayed setpoint	SS [*Value*]	OK
Set external probe on status	SE [*Binary value*]	OK
Set pump speed	SPS [*String value*]**	OK
Set temperature unit	STU [*String value*]***	OK
Set unit on status	SO [*Binary value*]	OK

**C denotes Celsius, setting in Kelvin (K) and Fahrenheit (F) would also be possible*
***Pump speed can be specified in low (L), medium (M) and high (H)*
****Temperature unit is returned in Celsius (C), Kelvin (K) or Fahrenheit (F)*

HPLC Pump

Two different types of pumps are used for the automatisation. The first is the HPLC (High performance liquid chromatography) pump, namely the AZURA Pump P 4.1S or P 2.1S from Knauer (Figure 3b). All necessary information about the device is taken from the corresponding device manual. [17]

The settings for the serial communication interface between host PC and the HPLC pump are specified as follows:

- *Serial comm* RS-232, LAN
- *Baud* 9600
- *Parity* no parity check
- *Data bits* 8
- *Stop bits* 1

In Table 2 all functions of the HPLC pump used in the implementation are specified. Once again, the communication protocol is apparent, which is analogous to the thermostat. All commands and responses are terminated with a carriage return.

Table 2: Serial communication protocol of the HPLC pump.

Command description	**Master command**	**Slave response**
Read flow in μl/min	FLOW?	FLOW:[*Value*]
Read pressure in 0.1 MPa	PRESSURE?	PRESSURE:[*Value*]
Read minimum pressure in 0.1 MPa	PMIN[*Pump head*]?*	PMIN[*Pump head*]:[*Value*]
Read maximum pressure in 0.1 MPa	PMAX[*Pump head*]?	PMAX[*Pump head*]:[*Value*]
Set flow in μl/min	FLOW:[*Value*]	OK
Set pressure in 0.1 MPa	PRESSURE:[*Value*]	OK
Set minimum pressure in 0.1 MPa	PMIN[*Pump head*]:[*Value*]	OK
Set maximum pressure in 0.1 MPa	PMAX[*Pump head*]:[*Value*]	OK

**A distinction is made between maximum flow rate of 10 ml·min^{-1} or 50 ml·min^{-1}, inserting the numbers in the command for each case.*

Lambda Pump

The second type of pump used is a syringe pump, namely the Polyvalent programmable syringe pump - LAMBDA VIT-FIT from LAMBDA Laboratory Instruments (Figure 3c). All necessary information about the device is taken from the corresponding device manual. [18]

The settings for the serial communication interface between host PC and the Lambda pump are specified as follows:

- *Serial comm* RS-232, RS-485
- *Baud* 2400
- *Parity* Odd

- *Data bits* 8
- *Stop bits* 1

The communication with the Lambda pump is simpler than that of the previously mentioned devices. A command can be sent to switch on and set the flow rate at the same time. For switching off, this command can be used with a flow rate equal to zero. If this command is sent by the master, the slave will not return an answer. Another command can be used to query which flow rate is currently set. An overview of how the two commands are used in the implementation and how they are composed is given in Table 3.

Table 3: Serial communication protocol of the Lambda pump.

Command Description	**Master command**	**Slave response**
Read pump settings	*# ss mm G qs cr*	*< mm ss r ddd qs cr*
Set flow and turn pump on	*# ss mm r ddd qs cr*	-
Turn pump off	*# ss mm r* 000 *qs cr*	-
Explanation of indices		
ss	pump address	
mm	host-PC address	
r	pusher movement to the left (infusion)	
ddd	speed of rotation	
qs	control sum in HEX format	
cr	carriage return	
G	indicates the request for data	

Calorimeter

The heat flow calorimeter developed by MAIER et al. is used as the reaction calorimeter for the automatisation. The reactor of the calorimeter is divided into three segments, the precooling element, in which the components are not yet mixed with each other, and the two reactor elements. The measuring and the operating principles of this calorimeter are described in section 2.2 and in literature [1], additional information regarding the calibration is given in section 6.3 and the calculation method for evaluating the measurement results is given in the Appendix 8.2. The following settings are defined for the serial communication interface of the heat flow calorimeter [1]:

- *Serial comm* RS-232
- *Baud* 9600
- *Parity* no parity check
- *Data bits* 8
- *Stop bits* 1

The calorimeter has the characteristic of constantly sending values without having them explicitly requested. Consequently, the values can be retrieved at any time. Also relevant is the setting of a target temperature, for which the command has the following format:

$$< 1, Value >$$

Whereby the temperature in degrees Celsius is set for *Value*.

2.5 State Machine

2.5.1 Basic Concept of the State Machine

In the context of this thesis, the concept of state machines will later be used for the implementation of device drivers and the sequence control. State machines offer a way to describe the behaviour of a system. This is achieved by assigning states, which each describe a specific situation, and state changes, which provide the transition between the states. Furthermore, internal activities, which describe actions that are completed within the state, and events, which trigger state changes, are important aspects. The graphical representation of a state machine is called state diagram (Figure 5). The most important elements are explained in more detail below. All subsequent descriptions of the state machine are taken from the book *UML 2.5: Das umfassende Handbuch* by KECHER et al. [20].

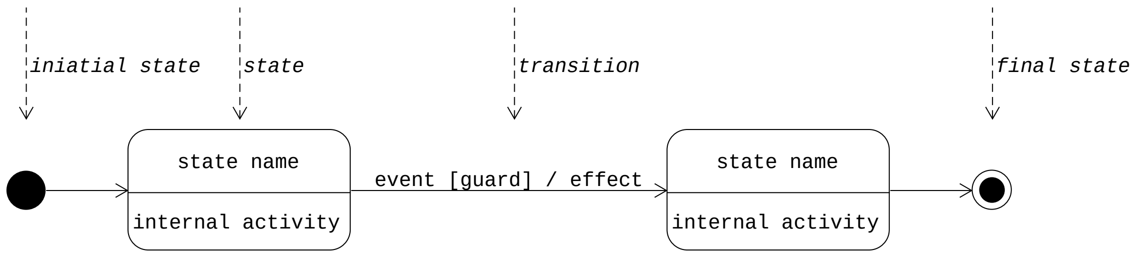


Figure 5: Basic elements and their notation in state diagrams.

2.5.2 Elements of the State Machine

State

Every state describes a specific situation of the system. As already mentioned, the sum of all states should cover the entire observable system behaviour. A distinction is made between static and dynamic states. Their difference depends on what happens during the active state. If the active state itself does not execute any action, for example it waits for an external input, it is referred to as a static state. If the active state itself performs an action, it is called a dynamic state.

Transition and Event

Transition refers to the one-way state change between two states and consists of up to three parts: the event, the guard and the effect. In the simplest case, the event is sufficient, but must be present for every transition. This is because the event triggers the transition and determines exactly which state is entered. A variety of different events can be distinguished. The events relevant for this thesis are:

- *Call event*:
  This is the simplest type of event. The event received is like a request to perform a certain operation. The state machine then changes to the state that performs this operation. For example, one receives an instruction from the supervisor to immediately carry out certain measurements in the laboratory.

- *Signal event*:
  These events are signals that enter the system from outside. In other words, the system receives information. The state machine now has the possibility to react to this information. For example, the fire alarm goes off during a measurement in the laboratory.

- *Change event*:
  This event is triggered under a certain condition. For example, after the laboratory work is finished, the laboratory can be cleaned up.

- *Time event*:
  This event is a special version of the change event. Here the condition refers to a point in time or a time period. For example, the measurement is finished when the measured value is constant for two minutes.

As already mentioned besides the event the guard and the effect are also parts of the transition.

The guard can be added in order to introduce a safety barrier. This means that the transition is only executed when the guard allows it. For example, the laboratory cannot be left until it has been cleaned up. The effect is an action that does not occur in the state but during the transition. The action is characterised by the fact that it can be done instantaneously. For example, the light is switched on when entering the laboratory.

Composite State

A composite state is when a state consists of several states and thus forms its own state machine. This is also referred to as a hierarchical state diagram. For example, several activities can be carried out in the laboratory. Each individual activity could in turn be divided into several steps.

2.5.3 Example of a State Machine

In Figure 6 an example of a state machine is given. The example illustrates how the process of a measurement in the laboratory could be described by means of states.

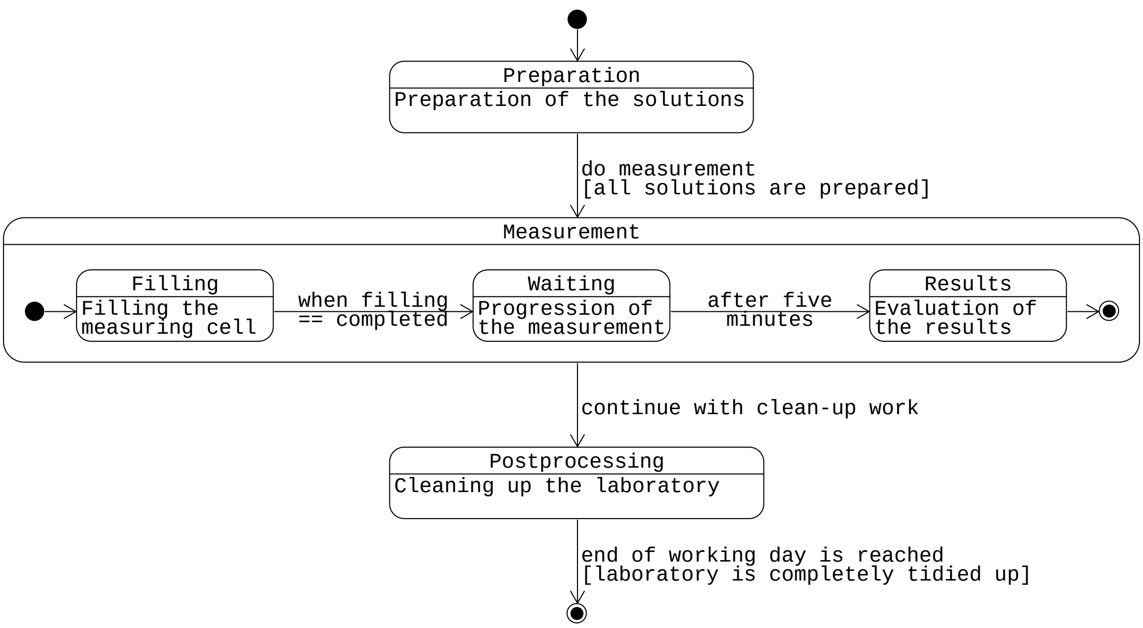


Figure 6: State diagram describing a possible measuring procedure in the laboratory.

Starting is in the *Preparation* state. Here, the internal activity is the preparation of the solutions to be measured. With the request that a measurement is to be made, a transition can be triggered (call event). However, before proceeding to the measurement process and thus carrying out the transition, a guard is first used to check whether all solutions have actually been prepared. Only if the result of this query is positive, the transition to the *Measurement* state is triggered.

The *Measurement* state is a composite state. It forms its own state machine consisting of three states. It starts in the *Filling* state. The internal activity is the filling of the measuring cell with the prepared solution. Here, a change event, whose condition is the completed filling of the system, triggers the transition to the *Waiting* state. The now active state, which in contrast to all other states is static, waits for the measurement to progress. A time event that refers to a period of time is used for the transition. The measurement is considered finished after five minutes. Subsequently, the *Results* state is entered. The internal activity is the recording and evaluation of the results. The next transition causes the composite *Measurement* state to enter the final state, which in turn causes the composite state to be exited and the next state, *Postprocessing*, to be entered. Again, a change event is used to trigger the transition, with the condition that the overall process of the measurement is completed.

The *Postprocessing* state is the last state which can be entered. The internal activity is cleaning up the laboratory. The last event is again a time event, which is related to a point in time, the end of the working day. Again, a guard is integrated which checks whether the laboratory is really completely tidied up; it cannot be left before this.

2.6 Design pattern

2.6.1 Definition of design pattern

The theory presented in the following sections is based on the book *Design patterns: Abstraction and reuse of object-oriented design* by GAMMA et al., who are also known as the *Gang of Four* and are responsible for the establishment of this subject area [21]. Since the beginnings of object-oriented programming, certain problems have been identified that can be found repeatedly in slightly modified forms. In order to organise these problems as clearly, efficiently and extensibly as possible, pattern implementations have been created. These pattern implementations are also called design patterns. A typical design pattern consists of the following four elements:

- *Pattern name:*
  A meaningful pattern name makes it possible on the one hand to guess what the pattern

does without going into further detail, and on the other hand to facilitate easy communication between developers.

- *Problem*:
  This indicates the problem for which the pattern was planned or which problems can be solved by the pattern. If a pattern is to be chosen for one's own purpose, the pattern whose problem is most similar to one's own problem is always selected. In the ideal case, the problem is even congruent. If there is no suitable pattern yet, the question can be asked whether the problem can be generalised and a new pattern can be developed from it.

- *Solution*:
  The solution specifies how the problem is broken down and solved. It specifies which objects are required, how they are interrelated and how they interact with each other. The solution thus provides a description for the structuring, which is not to be confused with an implementation.

- *Consequences*:
  By using the pattern, the main problem is solved, but other side effects can occur. These can be positive or negative. If these are known for the respective patterns, they are indicated so that the user is informed about them in advance. Negative side effects can be for example the use of a lot of storage space, increased runtimes, a considerable effort for expansion when requirements change over time.

2.6.2 Factory method pattern

Problem

The motivation to use a *Factory Method Pattern* is given when objects with the same interface are to be created, but the exact class and the number of possible classes are not yet specified. The design pattern thus provides an interface for object creation and delegates the instantiation of the various individual classes to subclasses.

Solution

The solution of the *Factory Method Pattern* is best explained by its structure. In Figure 7, this structure is given, along with the participants of the design pattern.

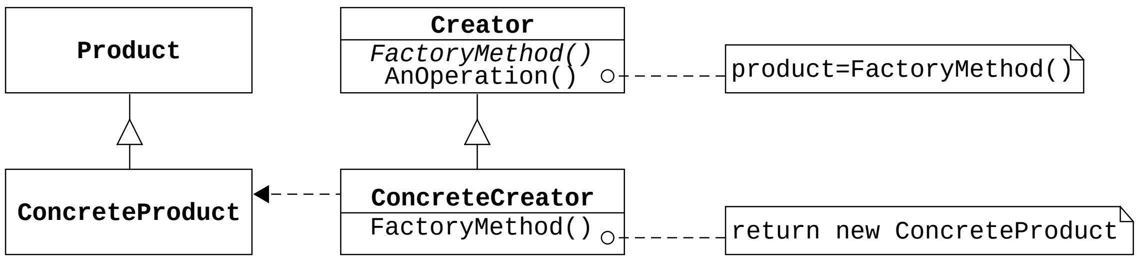


Figure 7: Structure of the *Factory Method Pattern*.

The design pattern has the participants *Creator*, *Product*, *Concrete Creator* and *Concrete Product*. The *Creator* is characterised by two properties. First, it contains the basic *Factory Method*, which returns the object *Product*. Secondly, the *Creator* has the ability to call the *Factory Method* function itself to create the *Product* object. The subclasses mentioned in the problem statement are the *Concrete Creators*. These inherit from the *Creator* and therefore create objects that have the same interface as the *Product* object. In order to instantiate a specific class, the inherited functions are adapted accordingly in the *Concrete Creator*. The object that is returned by a *Concrete Creator* is called *Concrete Product*.

Consequences

The greatest advantage of the *Factory Method Pattern* is the decoupling between the *Creator* and the *Concrete Products*. This means that new *Product* types can easily be added to an existing application at a later stage. However, the use of this design pattern leads to the creation of subclasses, which, with a high number of subclasses, could lead to a considerable increase in the complexity of the code.

2.6.3 State pattern

Problem

A problem is to be solved by using a state machine (cf. section 2.5). This means that the behaviour of an object is to be described by means of internal states. The *State Pattern* offers a possibility to model these internal states and their possible state changes.

Solution

In Figure 8 the structure and participants of the *State Pattern* are given.

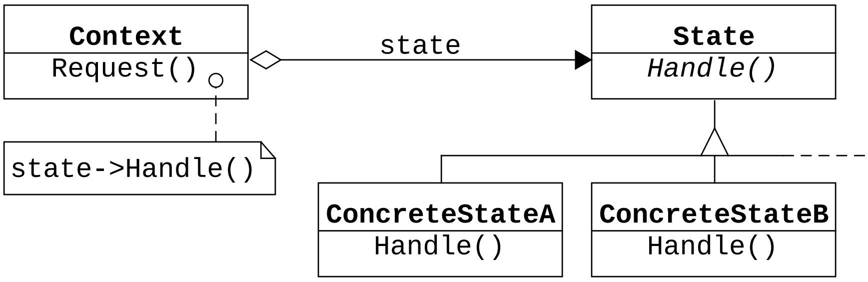


Figure 8: Structure of the *State Pattern*.

The design pattern features the participants *State*, *Concrete State* and *Context*. The *State* object forms the basic class for describing a state. It therefore contains all the functions that are necessary to operate a state. The individual states are implemented by the *Concrete States*. For this purpose, the *Concrete States* inherit from the State object and adapt the inherited functions accordingly. The last participant, the *Context*, has the functionality of the state machine. This means that it has the information about which combinations of states and state changes exist and ensures that the instance of the current state is stored according to the situation. The *Context* thus defines the interface for using the state machine.

Consequences

The structuring of the state pattern has several positive side effects. New states and associated new behaviours can be easily added. Furthermore, this pattern is highly maintainable. Although many problems can be solved by means of a state machine, the number of states required can increase greatly as the complexity of the problem increases. Consequently, the implementation effort also increases considerably. Therefore, it should always be considered in advance whether a solution using a state machine is worthwhile for a given problem or whether another solution strategy would be more effective.

2.6.4 Strategy pattern

Problem

The *Strategy Pattern* is used if different variants of an algorithm are necessary and these are not to be implemented by means of an overall algorithm. Meaning, a given problem has a main scheme that is valid for all applications. However, each individual application requires one or more additional individual features.

Solution

The *Strategy Pattern* suggests that an object is defined for each variant of the algorithm. Another participant then receives the corresponding object depending on the desired variant and executes it. The exact structure and participants of the *Strategy Pattern* are given in Figure 9.

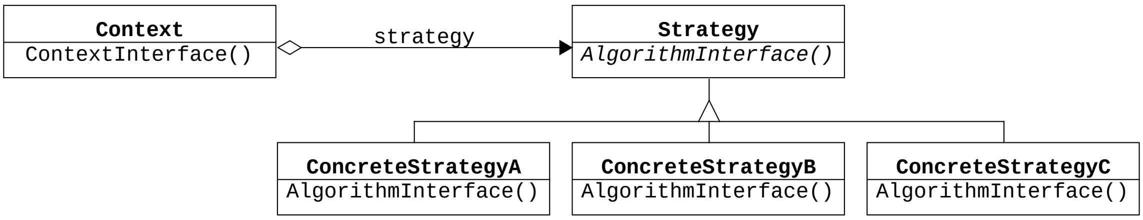


Figure 9: Structure of the *Strategy Pattern*.

This pattern has the participants *Strategy*, *Concrete Strategy* and *Context*. The *Strategy* represents the base class, which contains all the necessary functions for all further strategies. The *Concrete Strategy* objects inherit from the base class and adapt the functions of the *Concrete Strategy* accordingly. The *Context* has a general algorithm that is designed in such a way that it first calls the functions of the base class and second can execute all strategies with it.

Consequences

If a subfunction of the algorithm is passed to the *Strategy*, the remaining code in the *Context* object is simplified. As a result, the *Context* becomes clearer and more readable for a third party. Furthermore, the use of *Strategy Patterns* makes it possible to easily add new strategies later on, as long as they correspond to the basic schema.

When applying the *Strategy Pattern*, an assignment must be made for the algorithm as to which tasks are taken over by the *Strategy* and which by the *Context*. This boundary must be clearly defined, which is not easy to implement for every problem.

3 Application Development

3.1 Basic Principles of Application Development

In general, application development can be divided into four successive steps. These are the formulation of the problem, the development of an architecture, the implementation according to the architecture and the testing of the application. The last three parts form a cycle, as the architecture is adapted according to the errors found during testing and the changes are subsequently implemented. This principle is illustrated in Figure 10.

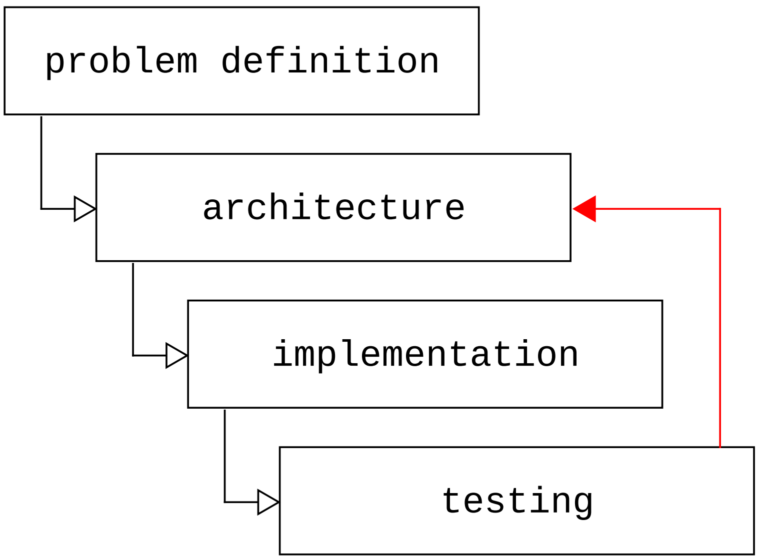


Figure 10: Four steps of application development.

In the following subsections, the first three points - problem definition, architecture and implementation - are dealt with in more detail. As errors are constantly being corrected due to the ongoing testing process, only the final version of the application architecture and the implementation are presented here.

The testing process in this thesis involves three steps. First, each individual class is tested for functionality immediately after its implementation. Secondly, those elements of the application concerning one of the devices are tested first without and then with the corresponding device. Thirdly, the entire application is also tested first without devices and then with the devices. As part of the last aspect, neutralisation experiments are carried out in order to demonstrate not only that the processing of operating points works, but also that a correct evaluation of measurement data is provided. More details on testing are given in section 4.1.

3.2 Use Case Specification

The use cases are specified at the beginning of the thesis and do not yet contain any implementation details. It is important that they are well formulated and that the sum of all use cases covers the desired scope of functionality. This means that no relevant use case should be overlooked, because once the architecture has been defined, implementing a new use case can prove to be difficult. In the context of this thesis, three use cases are defined.

The first use case, *Specific Use Case*, is the use of the installation for the measurement of a reaction between two different components. At least one operating point is to be handled during the execution. For this purpose, a list of operating points with at least one list entry is defined in advance. An operating point consists of a specification that determines the duration of the operating point, a set temperature and the flow rates of the components. Furthermore, in this use case the measurement results of the calorimeter are to be recorded, evaluated and saved in a file. Specific data for the evaluation of the measurement is therefore also required for this use case.

The second use case, *Optimisation Use Case*, is based on the first use case and the intended use of the application. In addition to conducting a measurement, it should be possible to apply an optimisation algorithm. Input of this application are the results of the calculation from the measurement data and output of the application is a new operating point. The initially defined operating point list can therefore always be extended by any number of new operating points after each processed point, insofar as the capacities (e.g. volume of the solution provided) permit. Since the implementation of a reaction optimisation software is not part of this work, the aim of this use case is to provide an interface between the automatisation and optimisation program.

The third use case, *Standard Use Case*, is the use of the installation without evaluating the generated measurement data and under no exact specification of the number of pumps. This has several advantages. The information required for an evaluation is not needed for this use case and therefore does not have to be specified. Furthermore, in this case the system can be operated without any pumps at all. The residual behaviour is similar to the *Specific Use Case*. Again, a list of predefined operating points with at least one entry is to be processed and the measurement data obtained from the calorimeter is to be saved in a file.

3.3 Application Architecture

In order to define the application architecture the use cases are compared with each other. It is important to define the main similarities and significant differences between the use cases. When considering the commonalities, the following points arise:

- When using the setup, at least the calorimeter and the thermostat are always present. This means that at least the specification for the duration of the operating point and the set temperature must be specified for the operating point.
- In all cases, the number of pumps is specified at the beginning and does not change during the run. Consequently, an equipment list can be set up for each use case at the outset, which is then valid for the entire runtime.
- An operation run consists of one or more operating points, in all cases existing operating points must be processed in sequence.
- The formatting of the data generated by the calorimeter during the run is always the same. Furthermore, for each use case there is the requirement to save the received data without processing it to a file.

When considering the differences, the following points can be identified:

- Depending on the use case, the data generated by the calorimeter is to be evaluated. Related to this, the required set of input information for the program varies.
- The length of the operating point list is not known in advance for each use case. Depending on the use case, it can be manipulated during the run.

The aim is to find a sequence that can ideally satisfy both the commonalities and the differences. This means that the sequence searched for can execute each of the use cases. Figure 11 illustrates one possible solution by means of a flowchart.

The workflow is started at the decision whether an operating point to be executed exists. If there is none, the workflow is finished. If the decision is positive, the first step of the workflow follows. Thereby, information on the operating point is fetched. In the next step, the operating point is set. Subsequently, a loop is initiated, which contains the two processes of reading the data and processing the data. The loop is terminated with the positive decision whether the operating point is finished. Since the entire process is also a loop, the workflow starts again after the termination of the operating point.

Based on this workflow, the architecture can be planned as a final step. For this purpose, participants are assigned to each element in the flowchart, which is shown by means of various colours in Figure 11. For the first decision, whether there is an operating point to

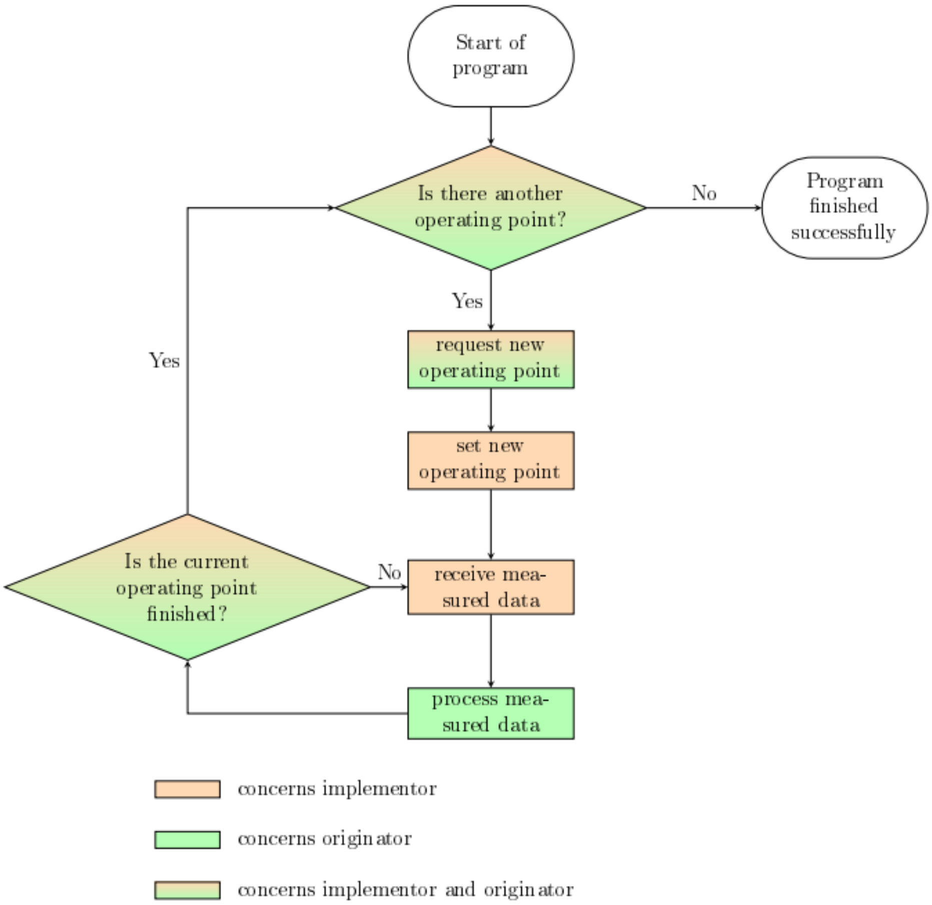


Figure 11: Flowchart illustrating the application architecture and its participants.

be executed, two participants can be assigned. The first participant, the implementor, asks the question, while the second participant, the originator, answers the question. In the next process, these two participants are involved again. The implementor gets the data of the operating point from the originator. The next process in the flowchart has only one participant which is the implementor who makes sure that the operating point is set. Subsequently, the two processes of reading in and processing the data follow. The reading is done by the implementor and the processing by the originator. The final decision is handled in the same way as the previous decision. The implementor asks whether the operating point is finished, while the originator provides the answer.

When looking at the participants, it is noticeable that the implementor has no independent information on the operating points. Consequently, the implementor's task is to work through exactly one use case, whereby any of the three can be assumed here. In contrast, the originator has the information about the operating points and also the information about what happens with the received data. Hence the implementor deals with the originally mentioned commonalities of the use cases and the originator deals with the mentioned differences. Therefore, this problem can be solved using a strategy pattern for the subsequent implementation. Within the Strategy Pattern, the implementor is the context, which will be equipped with a specific strategy. The context handles the communication with the devices. The originator is the specific strategy. This means that, if necessary, a specific strategy can be written for each individual use case.

3.4 Implementation

3.4.1 Implementation Approach

The architecture given in the previous section specifies the functionality of the context and the individual strategies. How the application achieves this functionality is an implementation detail. Since the implementation procedure is bottom up, the relevant elements of the context are implemented first.

The essential task of the context is the communication with the individual devices and the operation of these. Both the communication and the operation as well as the context itself are implemented by means of state machines. Therefore, the general realisation of the state machine concept is discussed first in the following subsection. Based on this, a description is given of how an example problem (operation of the thermostat) can be abstracted and the corresponding state machine can be constructed. Subsequently, the state machines of the remaining devices are explained.

For the implementation of the context, the relevant functions of the strategy and thus the basic strategy must be defined first. Afterwards, the state machine of the context is implemented. The final section addresses the entire strategy pattern and thus the three participants, strategy, context and specific strategy, as the combination of all participants gives the sequence control.

3.4.2 Realisation of the State Machine Concept

The state machine concept (cf. section 2.5) is to be implemented by means of a design pattern, the state pattern (cf. section 2.6.3). Based on this design pattern, an implementation template is created for each of the two participants, the state and the context, which are given the names state template and state machine template. For the state machine template, however, two more elements are needed. The first element is a factory, which handles the building of the states, and the second is the engine, which operates the states. Since the factory is also an implementation template, it will be referred to as the factory template in the following.

State Template

In general, defining a template for the state within the context of the state pattern ensures that each individual state created later has the same functions. To create the template, it is necessary to consider which functions must be available for the operation of states.

First, a state is initialised, i.e. it is entered. For this purpose, an *enter* function is defined, which has an input variable, namely the name of the state. In the next step, it should be possible to execute the state, for which a *call* function is defined. If the state is no longer needed, it is to be exited. Therefore, an *exit* function is created. Two more functions are necessary to operate the states. For special states, it may be the case that they themselves must be able to react to an event from outside. For this purpose, a *handle event* function is defined. Furthermore, the name of the state should be able to be queried externally. The *get state* function thus returns the state designation that the *enter* function has received.

The state template is implemented according to the state pattern as a separate class from which the individual states can later inherit. In the context of this thesis, this class is called *State_Base* and is given in Listing 1.

```
class State_Base:

    def enter(self, name):
        self.name = name

    def __call__(self):
        return None

    def exit(self):
        return

    def handle_event(self, event):
        return False

    def get_state(self):
        return self.name
```

Listing 1: Specification of the state template.

If a new state is built from this template later, the individual functions are overridden corresponding to the state. The *enter* function receives all parameters relevant to the state and eventually needs to provide other parameters for the *call* function. For example, the *enter* function receives a time as input variable from which a deadline is to be defined for the *call* function. The *call* function is equipped with the internal activity of the state. For example, the *call* function receives the ability to trigger the event that leaves the active state after the expiration of a given deadline. Finally, the *handle event* function can be adjusted too.

Factory Template

In order for a state to exist, it must be built and entered. This task is done by the factory. The factory is an essential part of the engine, the object that operates the states, and thus should have the same structure for each state machine. Therefore, the application of a design pattern is suitable here. The factory template is designed based on the factory method pattern introduced in the section 2.6.2. The basic principle of decoupling the creator and the concrete products is fulfilled in this template, but the introduced factory template has a much simpler structure than the original design pattern.

The factory consists of two functions. The first is the *initialisation* function, whereby the factory receives the necessary parameters for each possible state. The second is the *create state* function. From the sum of all possible states, the desired state is built and entered according to the call of this function. In contrast to the implementation of the state template, no general class is created for the factory template from which inheritance is possible. Instead, each factory used must be specifically defined each time according to the template presented in Listing 2.

Listing 2 demonstrates that the factory is always defined as a class. The *initialisation* function in this template receives three parameters that are saved within the class so that they are accessible to all other functions. When implementing this function for an individual factory, the incoming parameters are adapted correspondingly.

The second function, *create state*, has the state name as an incoming variable. In this example, there are two possible states that can be built: *State 1* and *State 2*. By means of a conditional operation, these two are queried. If one of the two branches is entered, the correct state is instantiated and entered according to the input state name. The function then returns the object of the entered state. If the factory is called with a state name for which no state exists, the last line triggers an error message. When implementing this function for an individual factory, it is important to ensure that the input parameter remains the same and that the object of the entered state is always returned. The state names, the instantiation of the state class and the entering of this instance must be adapted for each state machine.

```
class factory:
  def __init__(self, parameter_1, parameter_2, parameter_3):
    self.parameter_1 = parameter_1
    self.parameter_2 = parameter_2
    self.parameter_3 = parameter_3

  def create_state(self, state_name):
    if state_name == "State_1":
      st = State_1_class()
      st.enter(state_name, self.parameter_1, self.parameter_3)
      return st
    elif state_name == "State_2":
      st = State_2_class()
      st.enter(state_name, self.parameter_2, self.parameter_3)
      return st
    raise Exception("Unhandled State in Factory")
```

Listing 2: Factory template that must be present and adhered to when implementing any state machine.

Engine Object

The engine is responsible for operating the states. For example, it deals with retrieving the call function of the active state, handling occurring events, etc. In contrast to the previous elements of the state machine, no template is necessary in this case. The engine class is defined only once and instantiated in each state machine. The complete implementation of the engine is given in the Appendix 8.1.14. For a better understanding of the engine class, its relevant functions are given in Listing 3 and are explained below.

- *__init__*:
  This is the initialisation function of the engine. The function receives a table with the information which state transfers to which state with which event. In other words, it is a list with all possible states and transitions. Furthermore, the factory of the state machine and the initial state are handed over. Finally, the variable self.cur is defined, which will later be continuously overridden with the active state.

- *enter*:
  This function creates the specified initial state using the factory.

- *search_in_table*:
  This function has an event as an imput parameter. The state and transition table is

searched for this event by means of a loop. If there is an entry with this event for the active state, the active state is exited. The new state, which can also be taken from the table, is built and entered using the factory. If this procedure is successful, i.e. the specified event actually triggers a transition, the function additionally returns *True*. Otherwise, the function returns *False*. This feature makes it possible to query externally whether the event has been handled.

```
class Engine:

  def __init__(self, table, factory, init_state):
    self.tab = table
    self.fac = factory
    self.init_state = init_state
    self.cur = None

  def enter(self):
    self.cur = self.fac.create_state(self.init_state)

    if self.cur is None:
      raise Exception("Factory has created None")

  def search_in_table(self, event):
    for tran in self.tab:
      if not tran[0] == self.cur.get_state():
        continue
      if not tran[1] == event:
        continue

      self.cur.exit()
      self.cur = self.fac.create_state(tran[2])
      return True
    return False

  def tick(self):
    ent = self.cur()
    if ent is None:
      return None
    if self.search_in_table(ent):
      return None
    return ent

  def handle_event(self, event):
    if self.search_in_table(event):
      return True
    if self.cur.handle_event(event):
      return True
    return False
```

Listing 3: Specification of the relevant functions of the engine class.

- *tick*:
  This function first calls the *call* function of the active state. The response of the *call* function is stored in the variable *ent*. The next step is to check what is saved in the variable *ent*. If *ent* is None, nothing else needs to be done and the *tick* function is terminated. If *ent* is unequal to None, the *search_in_table* function checks whether the variable *ent* contains an executable event. If this is the case, the *tick* function is terminated with the execution of the transition. If both branches are ineffective, the *tick* function returns the variable *ent*.

- *handle_event*:
  This function asks the engine to handle a certain event. First, the table is searched to see if the event is contained in it. If there is no entry for the combination of active state and given event, the *handle_event* function of the active state is called and checked whether it can handle this event itself. Again, the function returns *True* if successful and *False* if unsuccessful.

State Machine Template

When specifying the state machine template, a distinction is made between two cases. The first case is given when the built state machine is again a state. In other words, the state machine of a composite state is to be defined. The second case occurs when the state machine is not a composite state. However, both cases have the following elements in common:

- Each state machine class contains the factory class described in this section.
- The initialisation function or its equivalent always defines the table of possible combinations of active states, state changes and target states that apply to the state machine. In addition, the factory and the engine are initialised in this function.
- In all cases, there must be a function that calls the *tick* function of the engine.

In the first case, the implementation of a state machine of a composite state, the structure of the object is already given by the state template. Meaning, the composite state inherits all the necessary functions from the *State_Base* as usual. The only distinctive difference between the composite state and an ordinary state is that the former is additionally equipped with the factory necessary for a state machine. The behaviour as a state machine is achieved by overriding the inherited functions. In Listing 4, the state machine template is given for the case of a composite state. The template shows which changes of the inherited functions are mandatory.

```
class Composite_State(State_Base):
  class factory: ...
    # Definition corresponding to Listing 2

  def enter(self, name, par_1, par_2, par_3):
    super().enter(name)
    self.tab = [
      ["State_1",    "next",    "State_2"],
      ["State_2",    "back",    "State_1"],
    ]
    self.fac = Composite_State.factory(par_1, par_2, par_3)
    self.en = Engine(self.tab, self.fac, "State_1")
    self.en.enter()

  def __call__(self):
    self.en.tick()

  def exit(self):
    self.en.exit()
    super().exit()
```

Listing 4: State machine template for a composite state.

In the second case, the state is not a composite state, which means that the structure is not given by the *State_Base*. The state machine template for this case is given in Listing 5. It can be seen that the structure here is very similar to the previous template.

According to this template, a class is defined for the state machine. Within this class, the factory class and five other functions are defined. Analogous to the *enter* function and the *call* function of the composite state, the initialisation function and the *tick* function are defined here. The next two functions defined in the given template are the *__del__* function and the *get_state* function. These two functions also appear in the composite state with the same functionality. The only significant difference from the previous template is the last function, which appears here for the first time. This function returns the state name of the active state specified in the initialisation function. For an individual implementation of a state machine, further functions or classes can be added to the object or the existing functions can be overridden correspondingly.

```
class State_Machine():
  class factory: ...
    # Definition corresponding to Listing 2

  def __init__(self, name, par_1, par_2, par_3):
    self.name = name
    self.tab = [
      ["State_1",    "next",    "State_2"],
      ["State_2",    "back",    "State_1"],
    ]
```

```
        self.fac = State_Machine.factory(par_1, par_2, par_3)
        self.en = Engine(self.tab, self.fac, "State_1")
        self.en.enter()

    def tick(self):
        self.en.tick()

    def __del__(self):
        self.en.exit()

    def get_state(self):
        return self.en.get_state()

    def get_name(self):
        return self.name
```

Listing 5: State machine template if the created state machine is not a composite state.

3.4.3 Creating a State Machine

This section describes how to proceed when planning and implementing a state machine. To illustrate this planning process, the Fisher thermostat is used as example.

The first step is to consider which functions are necessary to operate the thermostat. In order to set an operating point, it must be possible to set a specific temperature at the thermostat. For the subsequent operation of the operating point, switch-on and switch-off functions of the thermostat pump are required. In addition to these functions that concern the processing of the operating points, there are also functions that are necessary for a one-time setting. Such a configuration is important as the system could be used by other people. General settings can be adjusted by them and, if unnoticed, lead to errors when the unit is operated again. These one-time settings include the temperature unit, which can be specified in Celsius, Fahrenheit and Kelvin according to the manual. Furthermore, the distinction between external or internal temperature sensor and the setting of the pump speed are relevant.

The next step is to gather the necessary information about the thermostat used. A comparison with its device manual shows that all the desired functions mentioned earlier are available. If this is not the case, the device is not suited for automatisation. Furthermore, the thermostat has the ability that the individual specifications can not only be set by the master, but can also be queried. It is worth noting that each setting of the master triggers the response *OK* of the slave if successfully executed. The set of functions and their denotations, which are finally used for the implementation, are described in more detail in section 2.4.

Based on the functions mentioned, the states entered by the thermostat and their transitions are determined. Starting from the *Configuration* state, one-time settings are configured. Regarding the switch-on and switch-off functions of the thermostat pump, the following four different states result: *Deactivated*, *Activating*, *Activated* and *Deactivating*. Since the temperature is set while the pump is activated or deactivated, there is no separate state for this issue. By assuming the pump is deactivated in the *Configuration* state, the event *next* causes a transition from the active state to the *Deactivated* state. In order to enable error handling later, an *Error* state is defined, which can be entered from any other state triggered by the *error* event. The remaining transitions can be obtained from the Figure 12, which illustrates the state diagram resulting from these definitions.

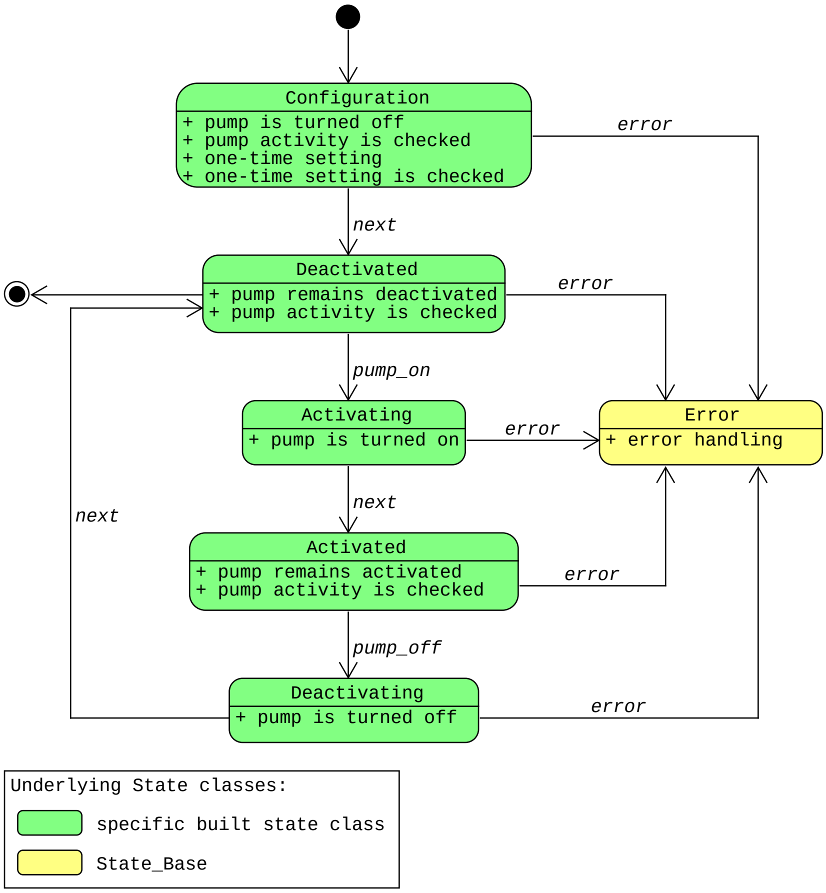


Figure 12: Resulting state machine for the thermostat driver.

Designing Layer A States

For each state in Figure 12, which are referred to as layer A states, one or more internal activities can be assigned. However, the sum of the internal activities of each of these states is still very extensive. Therefore, constructing a composite state for each individual state is recommended. Consequently, a state machine is defined for each layer A state.

While the thermostat is in the *Configuration* state, the following tasks must be carried out or at least considered:

- Ensure the pump is turned off.
- One-time settings are to be made and verified.
- Error handling should be supplementable later.

For this reason, this layer A state starts with switching off the pump and then checking the pump activity. Subsequently, the three settings and checks of the temperature unit, the pump speed and the temperature sensor used are carried out. At the end, the *Finished* state is defined, which generates the event *next*, which triggers the transition from the *Configuration* state to the *Deactivated* state. For error handling within the *Configuration* state, an *Error* state is defined. This state can be entered within the *Configuration* state by all states that perform either a setting or a query. Analogous to the *Finished* state, the *Error* state triggers the event *error*, which results in the transition from the *Configuration* state to the *Error* state (one level higher). In Figure 13 the state machine for the composite *Configuration* state is shown.

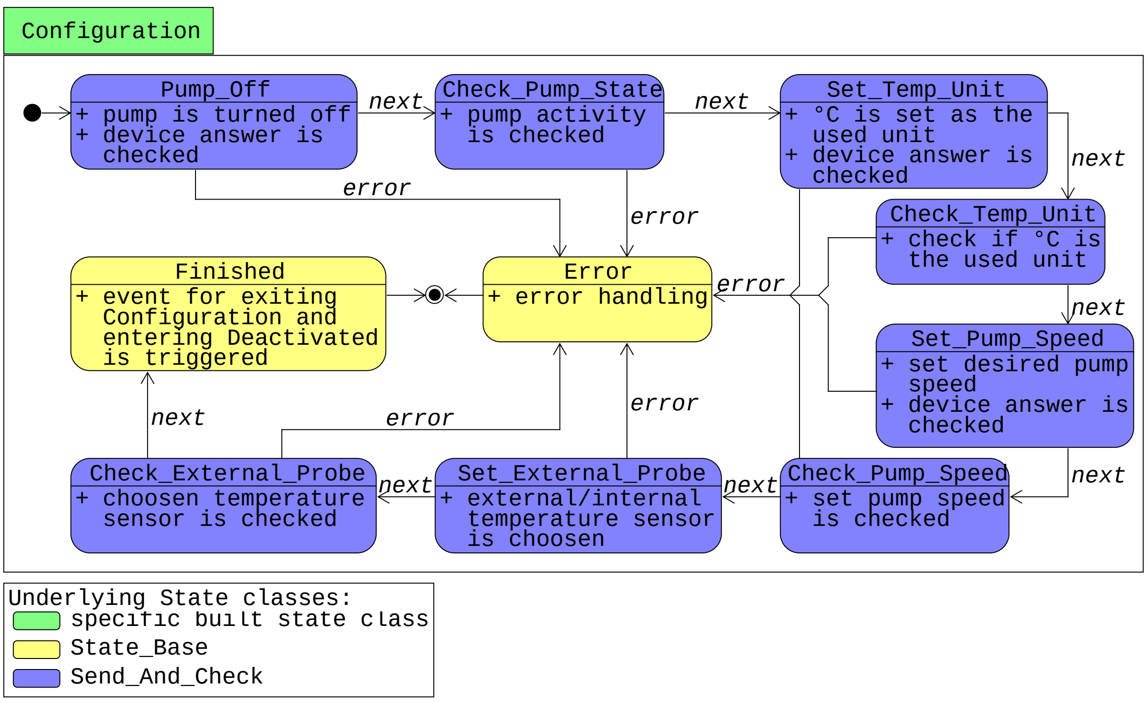


Figure 13: Resulting state machine for the composite *Configuration* state of the Fisher thermostat.

The planned state machines for the composite *Deactivated* and *Activated* state are very similar and can therefore be explained together. The tasks of these states can be defined as followed:

- Pump activity must be checked regularly.
- If there is a new set temperature, it is to be set and checked.
- Again, error handling should be possible.

Figure 14 shows the state machine for the composite *Deactivated* state. The composite *Deactivated* or *Activated* state starts with the pump activity check. Depending on the state, it is asked whether the pump is switched off or on. Since the thermostat is likely to remain in one of these two states for a longer period of time and the constant query of the pump activity is not necessary, a delay state called *Waiting* is introduced. After the expiration of a given deadline the *Waiting* state is exited and the *Check_Pump_State* state is entered again. Furthermore, the state machine is set up in such a way that if a new temperature is to be set, this is only handled during the *Waiting* state. In this way, it is ensured that the *Check_Pump_State* state is not aborted under any circumstances, while the *Waiting* state is aborted for this purpose. A state is then defined for the temperature setting and its check. The *Error* state is also specified and has the same functionality as in the composite *Configuration*

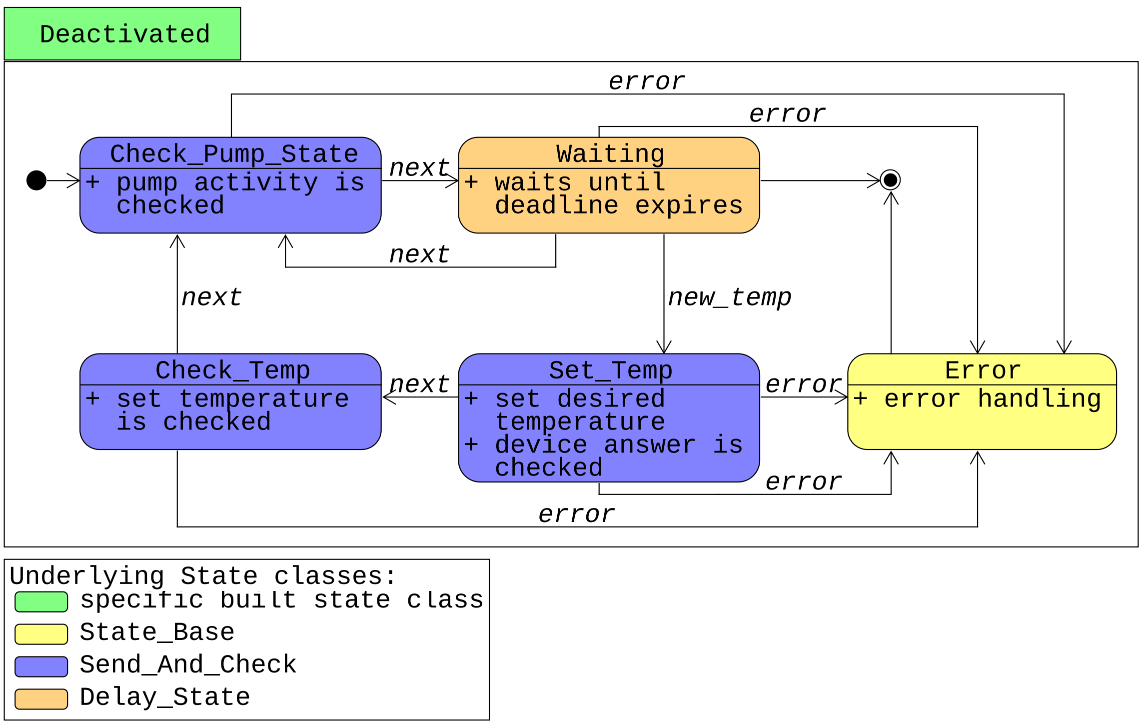


Figure 14: Resulting state machine for the composite *Deactivated* state of the Fisher thermostat.

state. In order to leave the *Deactivated* or *Activated* state, the request to switch the pump on or off can be made externally at any time. As soon as the *Waiting* state is entered and there is no new temperature to be set, the layer A state can be terminated.

Finally, the *Activating* and *Deactivating* states can be explained together. The tasks of these states can be defined as followed:

- Depending on the state the pump must be switched on or off.
- Subsequently, the pump activity must be checked.
- Error handling should be possible.

After the pump has been switched on or off and the pump activity has been checked, the *Finished* state is entered. Analogous to the *Finished* state of the composite *Configuration* state, an event is also triggered, which results in a transition from *Activating* to *Activated* or from *Deactivating* to *Deactivated*. Figure 15 shows the state machine for the composite *Activating* state.

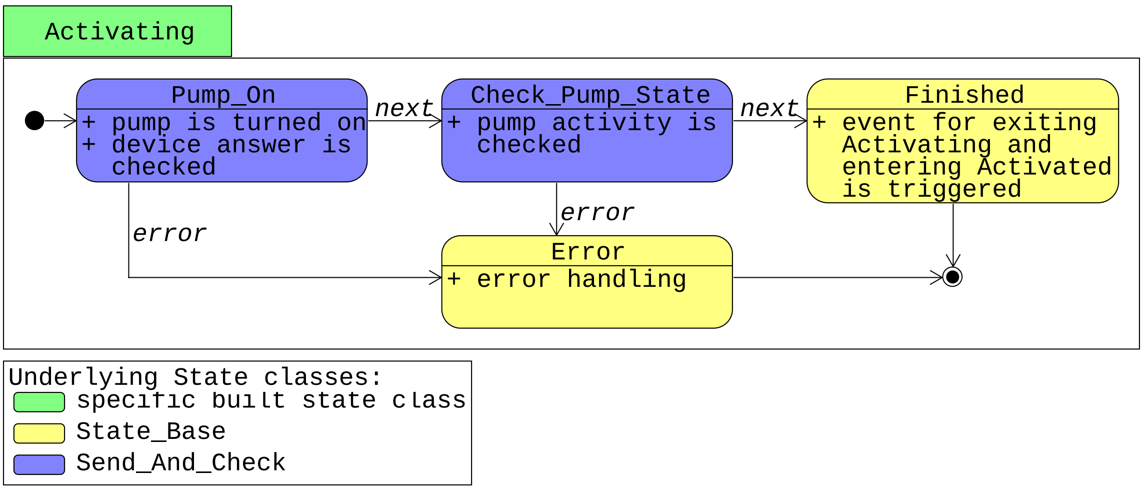


Figure 15: Resulting state machine for the composite *Activating* state of the Fisher thermostat.

Designing Layer B States

When describing the composite layer A states, a large number of new states have emerged. Due to the functionality of the states, they can be grouped into three different state classes, which are indicated by different highlighting in Figure 13, 14 and 15. These few states are referred to as layer B states. Due to the assigned internal activity, some of these are again composite states.

The first layer B state addresses the task of doing a configuration or a query. The state passes through the three internal states of sending a command, checking the response received and beeing finished, which is why the state is also referred to as *Send_And_Check*. Since the check can be negative, this state again contains an *Error* state. Furthermore, the *Send_And_Check* state has the following special feature: Depending on the specification, the sequence can be repeated. This has the advantage that in case of a one-time communication problem, the *Error* state is not entered immediately. Figure 16 shows the state machine for the composite *Send_And_Check* state.

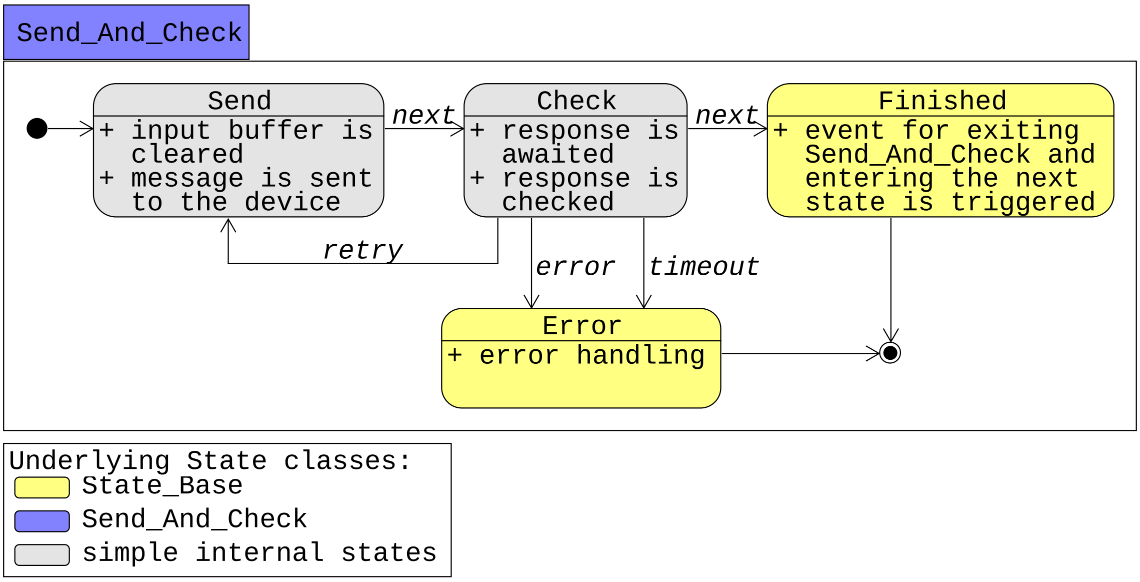


Figure 16: State diagram of the composite layer B state *Send_And_Check*.

The remaining two layer B states are simpler and therefore no more composite states are necessary. For the layer A states *Deactivated* and *Activated*, a delay state occurs. This state receives a deadline when entering, does not actively do anything during this period of time and triggers an event after its expiration. Furthermore, the internal states *Finished* and *Error* are supposed to trigger corresponding events. Therefore, it is sufficient to define these as simple base states without specific content.

Designing Layer C States

For the composite Layer B state *Send_And_Check*, an internal activity must now be assigned to each of the internal states, which are referred to as layer C states. In Figure 16, the two Layer C states are highlighted using gray colour.

The *Send* state ensures that the input buffer is cleared first and then sends a given message as a byte array to the device. Afterwards, the state returns an event. The *Check* state is waiting for a response. If the response does not arrive within a certain period of time, a timeout event is triggered. After the response is received, it is checked whether it corresponds to the expectation. If the check is positive, the state returns the regular event. However, if the check is negative, an error event is triggered.

Final Implementation

Having specified the structure of the state machine top down, it can be implemented. In the context of this thesis, the state machine is implemented bottom up, starting with the layer C and layer B states. When implementing the state machines, the state template and state machine template presented in chapter 3.4.2 are applied. Furthermore, all layer B and layer C states are formulated in such a general way that they can be used later for the implementation of the state machines of the remaining equipment.

When implementing the layer A states, the composite layer B state *Send_And_Check* is used for the first time. Depending on the application, the messages and checkers used for the responses differ. The checkers in particular are not one-liners and are therefore additionally implemented as classes, though this is not discussed in more detail here.

Finally, the layer A states are combined to form the thermostat driver, resulting in the state machine given in Figure 12. For later purposes, the interface is of importance. Including which information is needed to create the driver instance and which functions can be called externally. According to this implementation, the name of the driver, a setting class and a communication handle are required for the instantiation of the thermostat driver. The functions that can be called externally are the ability to switch the pump on or off and to specify the temperature. With the setting class and all the callable functions, the requirements for the thermostat stated at the beginning of this section are fulfilled.

The exact implementation of all states mentioned in this section and the additional functions required for the thermostat driver can be looked up in the Appendix 8.1.7, 8.1.10 and 8.1.11.

3.4.4 Additional State Machines of the Equipment

This section briefly describes the state machines of the remaining devices: HPLC pump, Lambda pump and the calorimeter. The architecture of each state machine is displayed and compared with that of the thermostat driver. The exact implementation of the individual state machines can be looked up in the Appendix 8.1.3, 8.1.8 and 8.1.9. The functions of the devices relevant for the implementation and the applicable settings for the serial communication interface are given in section 2.4.

HPLC Pump

The state machine of the HPLC driver can be seen in Figure 17. It is evident that there is a high degree of similarity to the state machine of the thermostat driver.

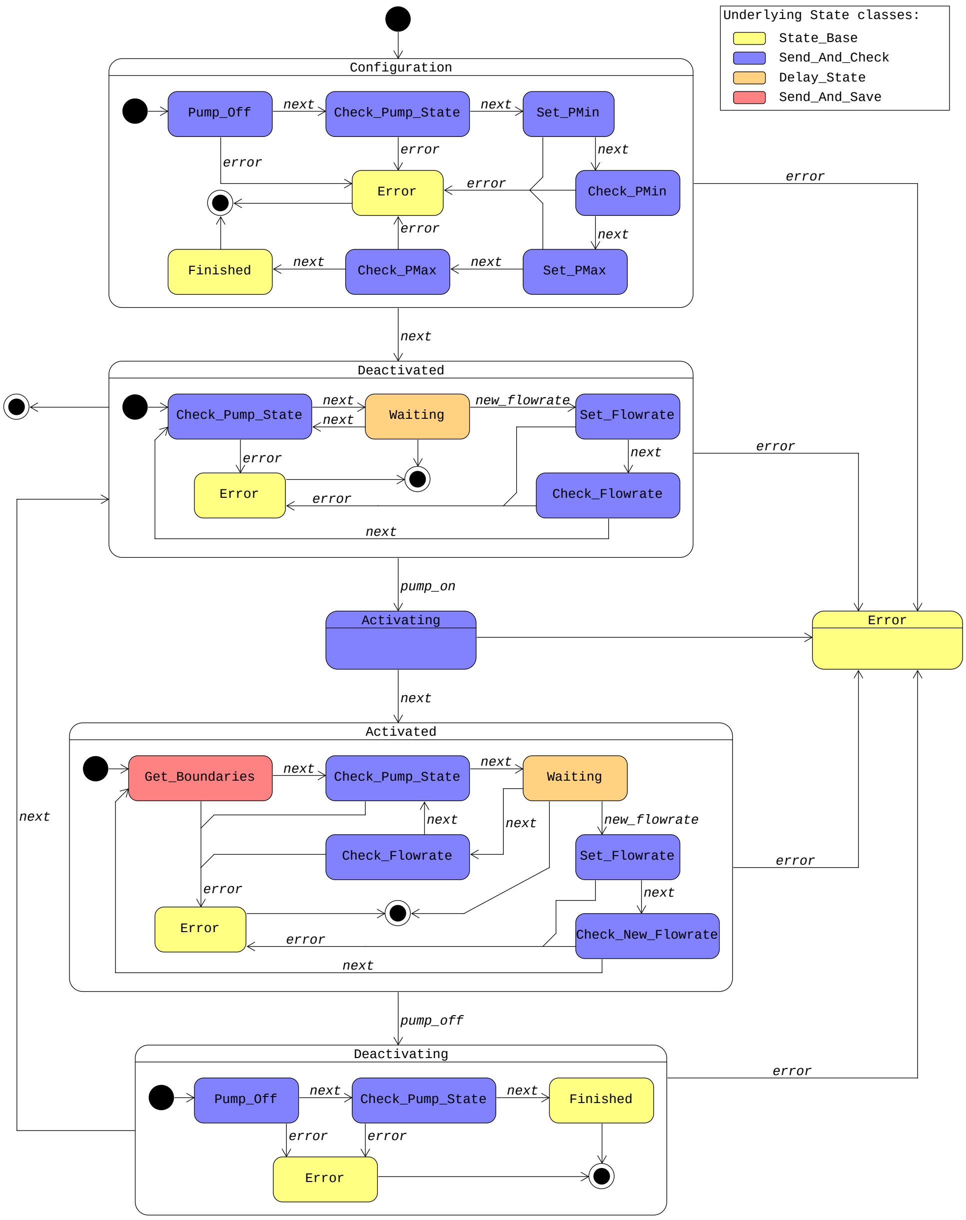


Figure 17: State diagram of the HPLC pump driver.

As preliminary settings can also be specified for the HPLC pump, the state machine again starts in a *Configuration* state. This state has the same structure as the *Configuration* state of the thermostat driver. The only difference is that the individual *Send_and_Check* classes are equipped with the functions relevant for the HPLC pump. This concerns the setting and subsequent checking of a pressure range that must not be left during the runtime of the pump. Having terminated the configuration, the HPLC pump switches to the *Deactivated* state. Again, this state has the same structure as the corresponding state for the thermostat driver. The only difference is that not the temperature but the flow rate is set and subsequently checked.

If the HPLC pump is to be switched on, the *Deactivated* state is exited and the *Activating* state is entered. The latter is implemented using the *Send_and_Check* state instead of using the corresponding state of the thermostat driver. The reason for this is a different check of the pump activity of the HPLC pump than that of the thermostat. In case of the thermostat, the activity could be checked directly via a separate query. No such function is available for the HPLC pump. Therefore, the pressure is checked instead. In the *Deactivated* state, the pressure becomes minimal but not zero, so a query is possible. However, at the moment of switching on, the pressure to be checked is unknown. Thus, the query is omitted.

Following the *Activating* state, the *Activated* state is entered. There are noticeable differences between this and the *Activated* state of the thermostat driver, which also result from the different pump activity query. The *Get_Boundaries* state is entered first, which periodically asks the HPLC pump for the pressure and subsequently saves the response. The state is not exited until a predefined number of pressure values is reached and the mean value and standard deviation are calculated from these. Implementing the *Get_Boundaries* state a new state class, the *Send_And_Save* class, is developed. Another difference between the *Activated* states is the distinction made for the flow rate check. Creating two different states makes the subsequent transition easier to handle. If a new flow rate is set, the *Get_Boundaries* state must be re-entered, as different pressures may occur for a different flow rate.

The remaining two states are the *Dectivated* state and the *Error* state. The former corresponds exactly to the *Deactivated* state of the thermostat driver, i.e. all objects contained therein are even given the same labels. The latter is defined as a basic state class, as is every *Error* state that occurs at any level in a driver state machine.

The interface of the HPLC driver class becomes important during implementation and its later use. For the instantiation, a specific driver name, the specification of the settings, the calibration curve valid for the specific pump and the communication handle are required. The relevant functions to be called externally to operate the pump are switching it on and off and setting a flow rate.

Lambda Pump

Figure 18 illustrates the state machine of the Lambda driver. It contains the familiar states *Deactivating*, *Deactivated*, *Activating*, *Activated* and *Error*. No default settings are possible for the Lambda pump, which is why the *Configuration* state is omitted for this state machine.

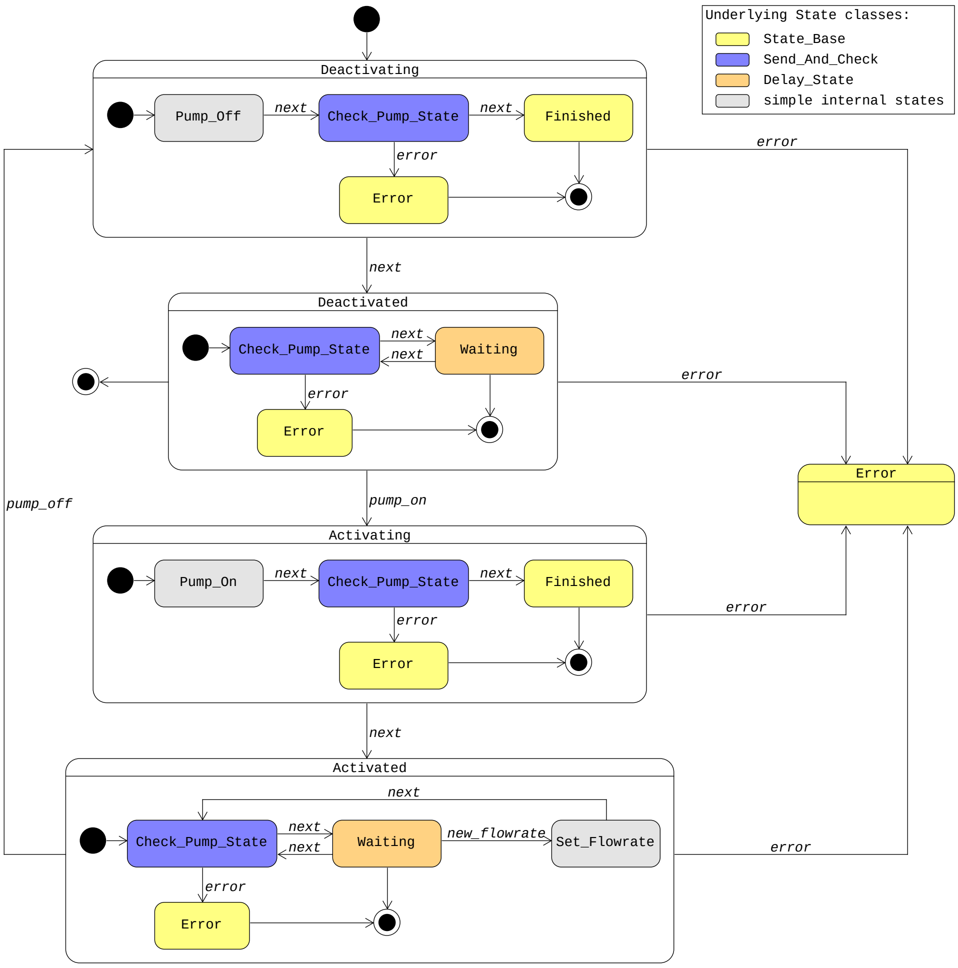


Figure 18: State diagram of the Lambda pump driver.

Since the communication between master and slave is simpler for the Lambda pump than for the previous units, its state machine also simplifies. The two states *Deactivating* and *Activating* generally have the same internal states and the same sequence as the corresponding states of the thermostat driver. The only difference is that the first state *Pump_Off* or *Pump_On* does not correspond to a *Send_And_Check* state, but instead to the simpler layer C *Send* state.

The other two composite states, *Deactivated* and *Activated*, have a simpler structure compared to those of the thermostat driver. Since the setting of the flow rate is coupled with the switching on of the pump, no flow rate can be set in the *Deactivated* state. Thus, all internal states associated with this task are omitted in the *Deactivated* state. In the *Activated* state, the flow rate can be changed, but again only by means of the simpler layer C *Send* state.

The interface of the Lambda driver class is similar to that of the HPLC driver. For the instantiation, a specific driver name, the calibration curve valid for the specific pump and the communication handle are required. The relevant functions that need to be called externally to operate the pump are the same as for the HPLC pump, switching on and off and setting a flow rate.

Calorimeter

In Figure 19 the state machine for the calorimeter is given. The calorimeter has a fundamentally different functionality than the thermostat and the pumps, consequently the state machine is completely different. Though the state machine is simpler than the previous ones, new state classes have to be designed for most of the occurring states.

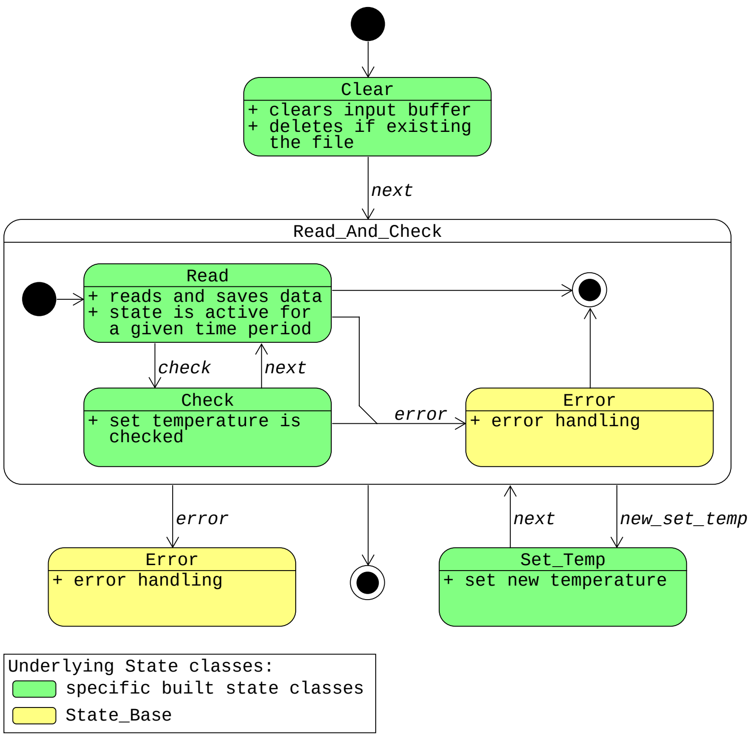


Figure 19: State diagram of the calorimeter driver.

The calorimeter starts off in the *Clear* state. It ensures an emptying of the input buffer of the calorimeter. In addition, it is checked whether the file in which the data is to be saved already exists. If this is the case, the file is deleted. This is because the data in the file will be added later and not overridden. If an old file with entered data were to be used, the data would be mixed in this case.

The event next causes a transition from the *Clear* state to the *Read_And_Check* state, which is a composite state. The internal state *Read* ensures that the data is read, stored internally and externally in the file. The internal state *Check* is entered at regular intervals. It checks the set temperature of the calorimeter. The *Read_And_Check* state also contains an *Error* state, which in turn enables error handling.

If a new temperature is to be set at the calorimeter, there is a transition from *Read_And_Check* to the *Set_Temp* state. After the temperature is successfully set, this state is exited and the *Read_And_Check* state is re-entered.

For the interface of the calorimeter driver class, the instantiation and the externally called functions are once again significant. When the driver class is instantiated, the driver name and the communication handle are passed. Additionally, an empty list is passed in which the measurement data is written later. Only one function that can be called externally is required, namely the one for setting the temperature.

3.4.5 Realisation of the Strategy Pattern

According to the architecture defined in section 3.3, the sequence control is implemented by means of a strategy pattern. The given architecture determines which tasks are handled by the strategy and which by the context. Therefore, the interface between strategy and context is first defined in more detail. This is achieved by formulating the basic functions of the strategy, i.e. the creation of a basic strategy class. Subsequently, the sequences relevant to the context (cf. Figure 11), which were specified in the architecture, are implemented by means of a state machine. Finally, the individual strategies for the use cases specified at the beginning (cf. section 3.2) are formulated in more detail based on the basic strategy class.

Basic Strategy Class

The functions of the strategy result from the workflow of the architecture. First, it is queried whether a new operating point exists and if so, its information is passed from the strategy to the context. For this purpose, the function *get_operation_point* is defined. The next task in the workflow, which brings the strategy back into action, is processing the data. In order to be able to process data, the strategy must first receive the data from the context, which is covered by the *push_value* function. The last element in the workflow is the query whether the operation point is finished, which is carried out by the *point_complete* function.

In addition to these functions that directly affect workflow, there are other functions required for the strategy. The *has_error* function is introduced for error handling. For the evaluation of the measurement data and thus for the *Specific Use Case*, the *push_actual_flowrate* function is required, which returns the actual flow rate set at the pumps. Furthermore, it should be possible for the strategy to give instructions to the context shortly before the program is terminated. This is achieved by the *get_finish_instruction* function.

Apart from the functions, a class is defined in the basic strategy which contains the relevant information of the operating points for the context. Meaning the operating point passed to the basic strategy class contains the three sets of information: specification, which determines the duration of the operating point, set temperature and flow rates of the components. The operating point that is passed on via the class created in the basic strategy class contains all previously mentioned information except the duration information. This is because this term can be specified differently and when handled by the strategy, the methodology is easily interchangeable.

The basic strategy is implemented according to the strategy pattern as a separate class from which the individual strategies can inherit. In the context of this thesis, this class is called *Strategy_Base* and is given in Listing 6.

```
class Strategy_Base:
  class operation_point_information:
    def __init__(self, temperature, flowrate_list):
      self.temperature = temperature
      self.flowrate_list = flowrate_list

    def get_temperature(self):
      return self.temperature

    def get_flowrate(self, idx):
      return self.flowrate_list[idx]

    def get_number_of_pumps(self):
      return len(self.flowrate_list)

  def get_operation_point(self):
    return None
```

```python

    def push_value(self, value):
        return

    def point_complete(self):
        return False

    def has_error(self):
        return False

    def push_actual_flowrate(self, val):
        return

    def get_finish_instruction(self):
        return None
```

Listing 6: Basic strategy from which any additional strategy can be built.

Context Class

The strategy has the relevant information on the program workflow, such as which devices are present in the system, which working points are to be set and how they are to be processed. For the sake of completeness, it should be mentioned that the strategy can also take over actions, such as the evaluation of measurement data. However, all the points mentioned have in common that there is no direct contact with the individual units and the information about the workflow is only available but not used. Consequently, these tasks, the communication with the devices and the execution of the sequence control, are carried out by the context. The context is planned based on the given architecture and the previously defined functions of the basic strategy. In the scope of this work, the context is implemented analogously to the drivers as a state machine. The corresponding state diagram is given in Figure 20.

If no errors occur, the state machine runs through the states *Apply_Configuration*, *List_Processing* and *Finished* in sequence. In the *Apply_Configuration* state, the system is prepared for the upcoming processing of the operating points. This includes the task of bringing all existing devices featuring a pump into the *Deactivated* state. In connection with this, those units featuring a *Configuration* state must pass through this state, as this is the only way they can reach the *Deactivated* state.

The *List_Processing* state is more complex than the other states and is therefore a composite state. Furthermore, communication between context and strategy becomes relevant. When entering *List_Processing*, the decision is first made whether another operation point is to be processed. For this purpose, the *get_operation_point* function of the strategy is called.

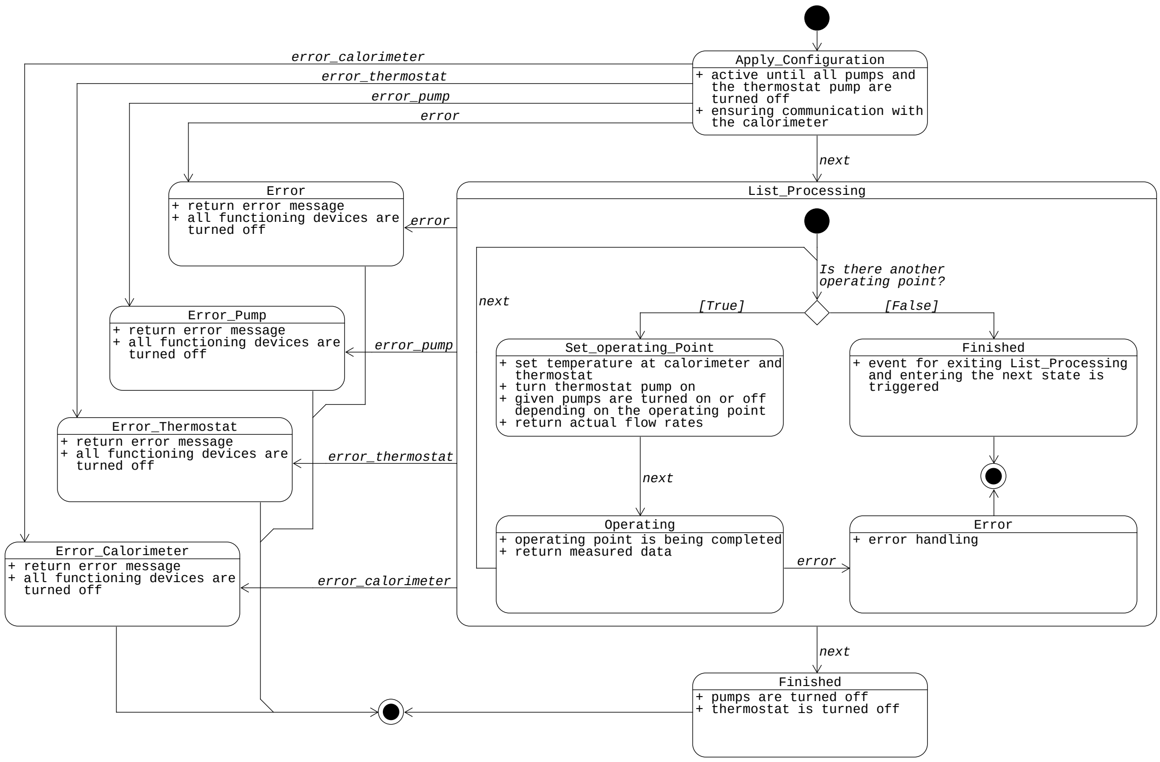


Figure 20: State diagram of the context as part of the *Strategy Pattern*.

If the strategy passes an operating point to the context, the *Set_Operating_Point* state is subsequently entered. All relevant settings for the operating point are made in this state. First, the operating temperature is set at the calorimeter and the thermostat. The temperature at the thermostat is always set slightly higher according to the calibration (cf. section 6.2). Secondly, the thermostat pump and the pumps according to the operating point are switched on. The flow rate that is actually set at the pumps is subsequently returned to the strategy. With this, the *Set_Operating_Point* state is terminated and the *Operating* state is entered. It consists of the three functions of the strategy *push_value*, *point_complete* and *has_error*. Thus, the measurement data are forwarded to the strategy and a query is made whether the operating point is completed. The last function informs whether an error has occurred on the part of the strategy. If an error occurs during the *Operating* state, the *Error* state is entered. Otherwise, after completion of the *Operating* state, the system returns to the decision at the beginning. If there is no further operating point, the *Finished* state is entered, which triggers the event to exit the *List_Processing* state.

The *Finished* state of this state machine is not a basic state class, unlike all previous *Finished* states. The state class built for this purpose ensures that all devices featuring a pump are transferred to their *Deactivated* state in a specific order. First the pumps are deactivated, next the thermostat pump is deactivated. Furthermore, the last function of the strategy the *get_finish_instruction* function is called. This state class can also be used for the individual error states. In this case, those devices that can still be addressed are switched off in the same order mentioned before. Regarding the error states, it can be seen that there are four different states. Three of them concern the individual units pumps, thermostat and calorimeter. In this way, the error handling can be treated in a device-specific manner. The state machine is aware of which device the error originates from and, if necessary, the error state for the individual devices can be adjusted individually. The last error state is the standard error state, which is used if the context or the strategy have an error and not the devices.

Once again, the resulting interface after implementation of the context is important. When instantiating the context, the strategy to be used, a list of the pumps used (name and communication port), a list of the thermostat information (communication port and settings if desired), the communication port of the calorimeter and an empty list for the measurement data must be provided. The functions that are called externally are reduced to the *tick* and the *get_state* function. More details regarding the implementation of the state machine class for the context is given in the Appendix 8.1.1. The additional functions and classes that are necessary to give the context its described functionality are also given therein.

Concret Strategies

As sequence control is the combination of context and strategy, the last step in program development is to plan the concrete strategies. The aim is to cover all three of the initially defined use cases, starting with the *Standard Use Case*.

OPERATION_POINT_LIST STRATEGY

The basic idea of this concrete strategy, which fulfils the *Standard Use Case*, is to provide a list of all operating points at the beginning and to work through these points one after the other. Thus, this concrete strategy is also referred to as *Operation_Point_List strategy*. Consequently, the input is only the list of operating points. For this purpose, a class is first defined which specifies the format of an entry in this list, which is given in Listing 7.

As can be seen from Listing 7, the specification of the duration of the operating point is given for this concrete strategy via an absolute time value. Furthermore, the entry also contains a desired set temperature and a list that assigns a flow rate to each existing pump. The length of the flow rate list results in the number of pumps, which can be requested by means of the *get_number_of_pumps* function also contained in this class. When executing the final program later, it is important that the length of the flow rate list matches the length of the pump list used to create the context. Listing 7 also gives an example of how the creation of an operating point list, and thus the instantiation of this class, can be done.

```
class operation_point_list_entry:
  def __init__(self, time_ms, temperature, flowrate_list):
    self.time_ms = time_ms
    self.temperature = temperature
    self.flowrate_list = flowrate_list

  def get_time_ms(self):
    return self.time_ms

  def get_temperature(self):
    return self.temperature

  def get_flowrate(self, idx):
    return self.flowrate_list[idx]

  def get_number_of_pumps(self):
    return len(self.flowrate_list)

example_list = [
  operation_point_list_entry(3, 25, []),
  operation_point_list_entry(3, 27, [1, 2]),]
```

Listing 7: Template for the list entries of the operating point list.

The next step is to plan the class for the concrete strategy. For this purpose, the functions are inherited from the basic strategy and adapted one after the other. Basically, the processing of the operating points consists of three recurring steps. First, the desired temperature must be set at the calorimeter, followed by regular checks to see whether the temperature has already remained stable. During this time period, the pump of the thermostat must be switched on, but the other pumps must be switched off. Secondly, a deadline is defined and the pumps are switched on according to the operating point. Thirdly, the system waits until the deadline expires and records the measured data. If the list of operating points consists of several entries, where each entry has the same set temperature, the first step does not have to be repeated each time.

For a simple handling of these three steps, states are defined again. Since the implementation of a sophisticated state machine is too time-consuming in relation to the complexity of the three steps, a simple state query by means of branches is sufficient. In other words, regarding the relevant functions inherited from the basic strategy, the individual states are entered via branches and the internal activity of the state is defined therein. The implementation results in the following overriding of the individual functions, whereby only fundamental changes are discussed here. Details of the implementation are provided in the Appendix 8.1.17.

- *__init__* :
  In this function, the incoming operating point list is first saved and additional variables are predefined for the strategy. It is worth noting that the internal active state is set to that of the temperature adjustment (*temperature equilibration*).

- *get_operation_point* :
  First it is checked whether there is still an entry to be edited in the initially received list. If there is not, the function is completed. If there is a point, it is saved as the current operating point. The next step is to check whether there is a new set temperature in the current operating point. If there is a new temperature, the internal state is set to *temperature equilibration*. Furthermore, a deadline is set for how long the temperature adjustment may take before the strategy reports an error. The function returns an object with the temperature to be set and the flow rates of the pumps set to zero, which is then executed by the context. If there is no new temperature, the internal state is set to that of setting a deadline (*setting deadline*). In this case, an object is returned with the temperature to be set and the flow rates of the pumps corresponding to the operating point.

- *push_value* :
  The function is only adapted insofar as the measurement data received from the context is saved within the strategy.

- *point_complete* :
  This function includes a query for each of the three possible states. If the *temperature equilibration* state is entered, the system waits for at least the last 10 measured temperatures to be within an absolute deviation of ± 0.1 for each of the three reactors of the calorimeter. Whereby a failure rate of 10 % is tolerated. If the *setting deadline* state is entered, the function sets a deadline based on the time specified in the operating point. The state is subsequently set to that of waiting until the deadline has expired (*waiting for deadline*). If the *waiting for deadline* state is entered, it is only checked whether the deadline has already expired.

- *has_error* :
  This function checks whether the deadline defined for the temperature adjustment has not yet expired. If it takes too long, the context is notified of an error.

OUTPUT_CALCULATION_ABSOLUTE_EVALUATION STRATEGY

This concrete strategy addresses the implementation of the *Specific Use Case*. In this use case, the measurement data received is to be evaluated in addition. Therefore, the implementation of the previous strategy (predefined operating point list and its processing) is completely adopted and extended with the additional requirements. The concrete strategy is named *Output_Calculation_Absolute_Evaluation strategy* due to the fact that the time period in which the data is evaluated is specified from the outset as an absolute value. Care must be taken that the duration of the evaluation does not exceed the duration of the operating point. The implementation is given in the Appendix 8.1.16.

The notable amendments made in the implementation of this strategy are narrowed down to the following points:

- Within the *__init__* function, an Excel file is prepared for the output. Tables and headings are predefined. The substance data are subsequently entered into the Excel file.
- Within the *push_value* function, the evaluation of the measurement data described in section 2.2 is implemented and results are written in the file. In addition, the measurement data are also saved in the Excel file.
- Within the scope of this strategy, the *get_finish_instruction* function is given a functionality. Two diagrams are created from the sum of all available measurement data. The first shows the individual inlet, outlet, reactor and set temperature over time. The second shows the measured voltage at the three reactors over time. Finally, the entire Excel file is formatted.

The *Output_Calculation_Absolute_Evaluation strategy* is designed to fulfil not only the *Specific Use Case* but also the *Optimisation Use Case*. Thus, the strategy offers a possible interface for reaction optimisation. For this purpose, the opimisation algorithm would have to be integrated into the *push_value* function, where the results of the evaluation are already available. Furthermore, due to the way of implementing the processing of the operating points, adding additional operating points in between is not a problem.

4 Results and Discussion

4.1 Testing of the Application

As already mentioned in section 3.1, application development consists of the four tasks of formulating the problem, developing an architecture, implementing the architecture and testing the application. The testing process, which is directly linked to the implementation, provides important insights into the interim results of the application development. The procedure chosen for this purpose is to test simple elements of the code immediately after implementation. In this way, occurring malfunctions and possible future sources of error are easier to identify and assign. If an error or a possible source of error is detected, the respective part of the application is immediately corrected.

An example of debugging is the addition of a retry variable to the layer B state *Send_And_Check*. After testing the thermostat device driver, it became apparent that it changes to the error state after an indeterminable amount of time during operation. In order to get to the bottom of this behaviour, the communication between the host PC and the thermostat that takes place during the runtime of the application was given as an output on the console. It was apparent that, contrary to expectations and the communication protocol of the device, the response to a request from the host PC was not always received. According to the former *Send_And_Check* state, in case of not receiving an answer or getting an unexpected answer, the error state is entered immediately. By introducing the retry variable, the request is repeated after a negative response. How often the request should be repeated before entering the error state is specified via this variable. Since the state class is to remain applicable for all other devices, the specification of 0 repetitions is possible.

In addition to immediately checking simple elements, testing those parts of the code that directly affect the devices is of importance too, as the previous debugging example demonstrates. In the simplest case, this testing procedure includes sending commands to the device and receiving its responses. In the next step, the layer A states concerning the corresponding devices are to be checked. It is important that each of the internal states of the composite layer A states are run through. Finally, the entire driver is checked. Each state change that is specified in the table of the state machine driver must be entered at least once.

As a final test, the entire application is run, whereby neutralisation experiments are conducted for this purpose. Since there are two strategies, the experiments are carried out with both of these strategies. In this context, the choice of strategy has no influence on the experiment itself.

The only difference is that one strategy provides a complete evaluation of the measurement data, whereas with the other strategy the evaluation must be done by the operator. Therefore, further differentiation is not necessary when illustrating the experiment results in section 4.3.

4.2 Final Application

The final application features the two strategies *Operation_Point_List* (OPL) and *Output_Calculation_Absolute_Evaluation* (OCAE). In order to be able to execute the application, a separate file is written for each strategy. In the following, the necessary input for each strategy, the initialisation of the context class, the output of the application and the execution of the application are described.

Input for the Operation_Point_List Strategy

For the OPL strategy, the list of operation points is required first. For this purpose, the array *operation_point_list* is created, in which the desired operating points can be entered. Each entry corresponds to the *operation_point_list_entry* class, which is defined for the OPL strategy. A list entry consists of three elements. First, the desired operating point duration is entered in ms. Here, care must be taken when entering that no duration is specified that is too short, since each operating point must first stabilise in order to obtain accurate measurement results. The second specification is the desired temperature at the calorimeter in °C. A temperature with two decimal digits could be specified here, but in this case care must be taken that a corresponding calibration curve is available. Otherwise, the application would enter the error state. The third information is the list of volumetric flow rates in ml·min^{-1}.

In the next step, the used devices are defined for the strategy. First, the pumps are assigned. For this purpose, an array is created, which can also have zero entries if no pumps are desired for the execution of the application. However, each pump entry requires two elements. First element is the designation for the pump. In case of the pumps available in the laboratory, there are the following six designations: *Lambda 1*, *Lambda 2*, *Lambda 3*, *HPLC A*, *HPLC B* and *HPLC C*. Secondly, the correct port name for the corresponding pump must be specified. The remaining devices are the thermostat and the calorimeter. Here, in case of the thermostat, the port name is written in an array, and in the case of the calorimeter, it is saved in a variable.

After the operating points and the devices have been specified, the strategy class can be initialised. Listing 8 shows the implementation of the specifications necessary for the OPL strategy. The complete implementation of the file for executing this strategy is given in Appendix 8.1.13.

```
# List of operating points
  operation_point_list = [
    Strategy_OPL.operation_point_list_entry(3*6E4, 25, [6, 6]),
    Strategy_OPL.operation_point_list_entry(5*6E4, 35, [5, 5]),
  ]

# Devices used
  User_Pumps = [["Lambda 1", "COM1"], ["Lambda 2", "COM2"]]
  User_Fisher = ["COM3"]
  Portname_Calorimeter = "COM4"

# Setting up the strategy
  strategy = Strategy_OPL.Operation_Point_List(
 operation_point_list)
```

Listing 8: Specifications needed for the execution of the OPL strategy.

Input for the Output_Calculation_Absolute_Evaluation Strategy

The OCAE strategy also requires a list of operating points first, which is structured analogously to the OPL strategy. However, in the case of the *operation_point_list_entry* class, it should be noted that this is defined here via the OCAE strategy. The subsequent definition of the devices is again identical.

In the next step, the additional data follow. Since this strategy immediately evaluates the data received from the calorimeter, the strategy needs a stabilisation time (here: dead time), a name for the created Excel file and the substance data. The dead time refers to the period of time, starting from the beginning of an operating point, within which the measured data is to be excluded from evaluation. For the Excel file, it must be considered that it is always saved in the same folder. If the name is not changed, the existing file will be overwritten. Finally, the substance data are defined in the *substance_data* class, which is defined in the strategy. All entries are made in a two dimensional array. This is because two participating substances are assumed in this strategy. It is therefore important to always keep the same order for substance A and substance B. Furthermore, the first substance entered should correspond to the limiting substance. The first entry for the substance data list is the weighed-in mass in g, the second is the volume used for the preparation of the solution in ml, the third is the molar mass corresponding to the substances in $g{\cdot}mol^{-1}$ and the fourth is the substance assignment to the pumps. Listing 9 shows the implementation of the specifications necessary for the OCAE strategy. In this example, it can be seen that substance B is filled in the pump

Lambda 1. Furthermore, it is evident that substance B weighs 5 g and a volume of 50 ml is used for the solution preparation.

```
# List of operating points
  operation_point_list = [
    Strategy_OCAE.operation_point_list_entry(3E5, 25, [6, 6]),
    Strategy_OCAE.operation_point_list_entry(3E5, 35, [5, 5]),
  ]

# Devices used
  User_Pumps = [["Lambda 1", "COM1"], ["Lambda 2", "COM2"]]
  User_Fisher = ["COM3"]
  Portname_Calorimeter = "COM4"

# Additional data
  dead_time = 1*6E4
  excel_file_name = "strategy_ocae"
  substance_data = Strategy_OCAE.substance_data([7, 5], [100,
 50], [40.01, 60.05], ["B", "A"])

# Setting up the strategy
  strategy = Strategy_OCAE.
 Output_Calculation_Absolute_Evaluation
  (operation_point_list, substance_data, dead_time,
 excel_file_name)
```

Listing 9: Specifications needed for the execution of the OCAE strategy.

Analogous to the OPL strategy, the OCAE strategy can be initialised after defining the required information. The complete implementation of the file for executing this strategy is given in Appendix 8.1.12.

Initialisation of the Context

According to the strategy pattern, the context class is the same for every strategy and is to be initialised with the specific strategy. The code given in Listing 10 is therefore the same for all strategies. Once the class has been initialised, it only needs to have its *tick* function called regularly, which ensures that the context state machine is run. The regular call is achieved by means of a loop that is exited as soon as the context has reached the finished state or an error state.

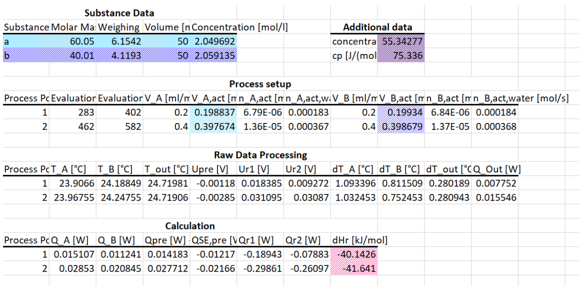
```
# Setting up the automatization
  automat = Auto.matization(strategy, User_Pumps, User_Fisher,
 Portname_Calorimeter)

# Automatization is called until the end state is reached
```

```
    while(True):
      automat.tick()
      if automat.get_state() == "Finished":
        break
      if automat.get_state() == "Error_Thermostat" or automat.get_state() == "Error_Pump" or automat.get_state() == "Error_Calorimeter" or automat.get_state() == "Error":
        break
    print("Done")
```

Listing 10: Initialisation and Calling of the context class.

Output of the Application

For both strategies, the output is a log file containing the complete data from the calorimeter. In case of the OCAE strategy, a second output is obtained with the Excel file. In Figure 21, the four resulting worksheets of the output Excel file are given. It can be seen that the



Substance Data

Substance	Molar Ma	Weighing	Volume [n	Concentration [mol/l]
a	60.05	6.1542	50	2.049692
b	40.01	4.1193	50	2.059135

Additional data

concentra	55.34277
cp [J/(mol	75.336

Process setup

Process Pc	Evaluatior	Evaluatior	V_A [ml/m	V_A,act [m	n_A,act [m	n_A,act,wa	V_B [ml/m	V_B,act [m	n_B,act [m	n_B,act,water [mol/s]
1	283	402	0.2	0.198837	6.79E-06	0.000183	0.2	0.19934	6.84E-06	0.000184
2	462	582	0.4	0.397674	1.36E-05	0.000367	0.4	0.398679	1.37E-05	0.000368

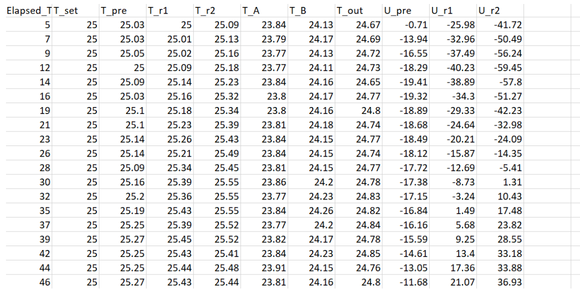
Raw Data Processing

Process Pc	T_A [°C]	T_B [°C]	T_out [°C]	Upre [V]	Ur1 [V]	Ur2 [V]	dT_A [°C]	dT_B [°C]	dT_out [°C	Q_Out [W]
1	23.9066	24.18849	24.71981	-0.00118	0.018385	0.009272	1.093396	0.811509	0.280189	0.007752
2	23.96755	24.24755	24.71906	-0.00285	0.031095	0.03087	1.032453	0.752453	0.280943	0.015546

Calculation

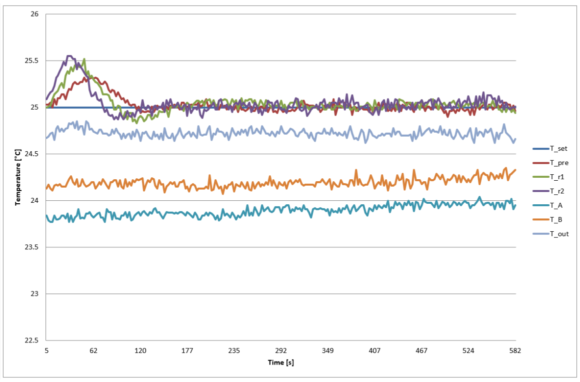
Process Pc	Q_A [W]	Q_B [W]	Qpre [W]	QSE,pre [V	Qr1 [W]	Qr2 [W]	dHr [kJ/mol]
1	0.015107	0.011241	0.014183	-0.01217	-0.18943	-0.07883	-40.1426
2	0.02853	0.020845	0.027712	-0.02166	-0.29861	-0.26097	-41.641

(a) *Evaluation* worksheet containing the substance data and the evaluation of the measurement data.



Elapsed_T	T_set	T_pre	T_r1	T_r2	T_A	T_B	T_out	U_pre	U_r1	U_r2
5	25	25.03	25	25.09	23.84	24.13	24.67	-0.71	-25.98	-41.72
7	25	25.03	25.01	25.13	23.79	24.17	24.69	-13.94	-32.96	-50.49
9	25	25.05	25.02	25.16	23.77	24.13	24.72	-16.55	-37.49	-56.24
12	25	25	25.09	25.18	23.77	24.11	24.73	-18.29	-40.23	-59.45
14	25	25.09	25.14	25.23	23.84	24.16	24.65	-19.41	-38.89	-57.8
16	25	25.03	25.16	25.32	23.8	24.17	24.77	-19.32	-34.3	-51.27
19	25	25.1	25.18	25.34	23.8	24.16	24.8	-18.89	-29.33	-42.23
21	25	25.1	25.23	25.39	23.81	24.18	24.74	-18.68	-24.64	-32.98
23	25	25.14	25.26	25.43	23.84	24.15	24.77	-18.49	-20.21	-24.09
26	25	25.14	25.21	25.49	23.84	24.15	24.74	-18.12	-15.87	-14.35
28	25	25.09	25.34	25.45	23.81	24.15	24.77	-17.72	-12.69	-5.41
30	25	25.16	25.39	25.55	23.86	24.2	24.78	-17.38	-8.73	1.31
32	25	25.2	25.36	25.55	23.77	24.23	24.83	-17.15	-3.24	10.43
35	25	25.19	25.43	25.55	23.84	24.26	24.82	-16.84	1.49	17.48
37	25	25.25	25.39	25.52	23.77	24.2	24.84	-16.16	5.68	23.82
39	25	25.27	25.45	25.52	23.82	24.17	24.78	-15.59	9.25	28.55
42	25	25.25	25.43	25.41	23.84	24.23	24.85	-14.61	13.4	33.18
44	25	25.25	25.44	25.48	23.91	24.15	24.76	-13.05	17.36	33.88
46	25	25.27	25.43	25.44	23.81	24.16	24.8	-11.68	21.07	36.93

(b) *Raw_Data_Com* worksheet containing the measurement data.



(c) *Dia_Raw_Temp* worksheet displaying the plot of measured temperatures over time.

(d) *Dia_Raw_Voltage* worksheet displaying the plot of measured voltages over time.

Figure 21: Worksheets of the output excel file.

output is divided into the four topics substance data, process setup, raw data processing and calculation. First, the substance data, which is the input for the concrete strategy, is given. Secondly, the period during which the evaluation is carried out, the desired and actual flow rates of both components and the flow rate converted to the water content of the components are stated. Thirdly, the results averaged over the evaluation period, the calculated temperature differences and the outgoing heat quantity are given. Finally, the remaining heat quantities (incoming heat quantity, reactor heat quantity) and the resulting molar reaction enthalpy are listed. The relevant data are highlighted in colour.

Execution of the Application

The final application can be used in the laboratory. To do this, the system must first be set up accordingly. A calorimeter and a thermostat are required in any case. The number of pumps can vary. Subsequently, a decision should be made as to what the system will be used for. For an experiment involving two components, the OCAE or the OPL strategy can be used. For an experiment with only one component (for example to determine the specific heat capacity of a substance) or more than two components, the OCAE strategy cannot be used and the OPL strategy remains. In this context, the decision of which strategy to use is therefore based on how many components are used in the experiment. The number of components does not necessarily have to be equal to the number of pumps. The number of pumps is greater than or equal to the number of components. This means, that in order to have more solvent available for an experiment using syringe pumps, two pumps can be used with the same substance. When entering the specifications in the application, care should be taken that no mistakes are made. After a strategy has been selected, the input data should be entered into the corresponding operation file. Regarding the number of operating points, it is important to ensure that enough initial substance is provided for the specified flow rates and operating times. Once the system has been completely set up, the tightness has been checked and the initial substances have been fed to the pumps accordingly, the application can be started.

Now the automated process follows, replacing the typical laboratory work. In terms of the application sequence, all units are first switched off. This ensures that the application is aware of the initial state of all the units and prevents the system from behaving incorrectly. Afterwards, the calorimeter and thermostat are set to the corresponding operating temperature and the pump of the thermostat is switched on. The temperature is then adjusted in the calorimeter. The application automatically detects when the temperature has reached a constant value and subsequently switches to the first operating point. Now all the operating points entered in the operating point list are processed. If there are temperature changes between the points, the pumps are switched off for this period of time so that the initial substances are not consumed unnecessarily. When all operating points have been processed, the application is finished. If the application does not run correctly, it ends in an error state with an output indicating which device caused the error.

After finishing the application, the user has the log file with the data of the performed measurement. If the application was carried out using the OCAE strategy, the data of this measurement is already evaluated. The advantage of the OCAE strategy is that the data is not calculated afterwards but continuously during the measurement. This strategy can therefore be combined very well with an optimisation software.

4.3 Experimental Results

In this subsection, the results of the neutralisation experiments conducted as part of the verification of the functionality of the developed sequence control are presented and discussed. The experiments are carried out at the three different concentrations of 1.65 mM, 1 M and 2 M and are subsequently referenced as first, second and third experiment respectively. The third experiment is carried out twice, thus a distinction is made between experiment 3a and 3b. For every experiment the same set temperature at the calorimeter of 25°C is used.

The first experiment (Figure 22) differs from the other experiments insofar as the measurement of the amount of heat is almost undetectable due to the concentration being lower by a factor of

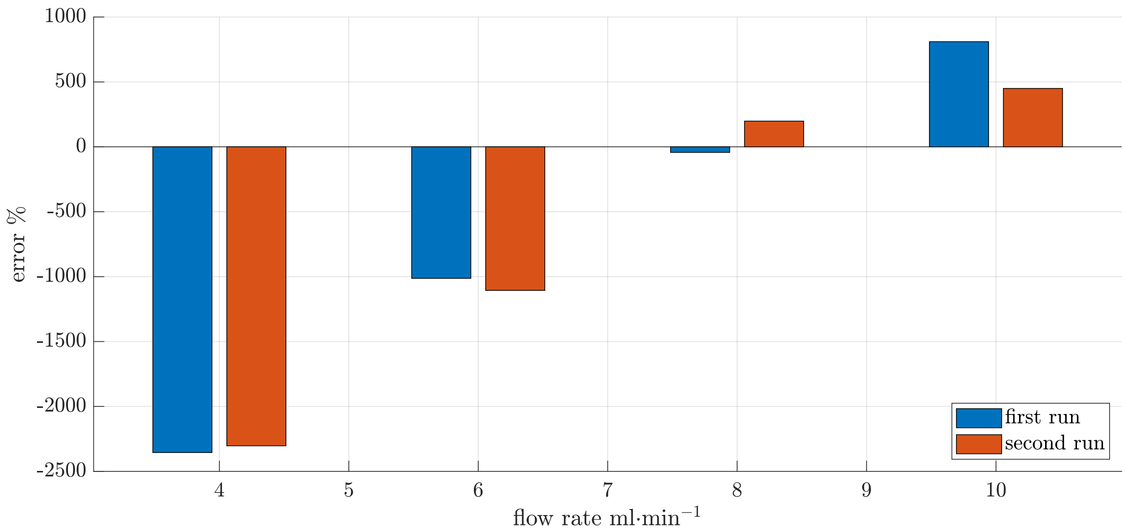


Figure 22: Relative errors of the determined molar reaction enthalpy from the first experiment at a temperature of 25°C using a concentration of 1.65 mM.

10^3 compared to the others. As a result, there are huge deviations of up to more than 1000 %. Figure 22 illustrates the deviations of the determined molar reaction enthalpy in relation to the literature value of $\Delta h_R = -57.4$ kJ · mol^{-1} given in section 2.3. In addition to the huge deviations, it remains worth mentioning that the individual errors seem to repeat at a given flow rate. Since each flow rate was only measured twice, further measurements are needed to check whether this behaviour is actually reproducible. If so, the offset error could be corrected and measurements of low concentrations would become feasible.

The remaining experiments 2, 3a and 3b are summed up in Figure 23. The Excel worksheet with the evaluation results of the neutralisation experiment (Figure 24) and the diagram of the temperatures during the experiment (Figure 25) are given exemplarily for experiment 3a.

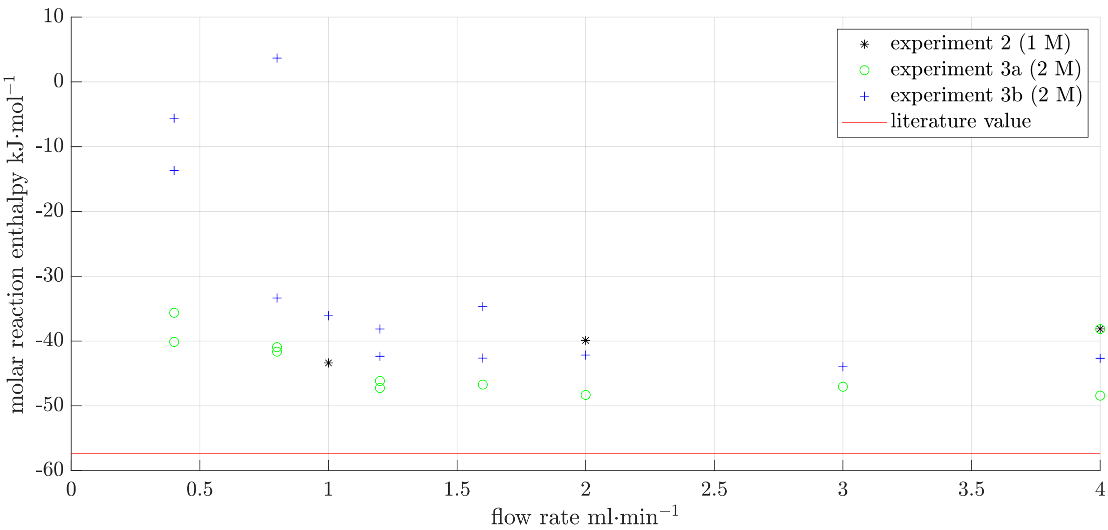


Figure 23: Determined molar reaction enthalpy from the second and third experiment at a temperature of 25°C using a concentration of 1 M and 2 M.

In contrast to the first experiment, the deviation from the literature value is much smaller, which is why the absolute values are shown instead of the relative error. The literature value of the molar reaction enthalpy is depicted via the red line. As can also be seen in Figure 22, independent of the concentration, the error decreases with increasing flow rate and approaches the literature value. Above a flow rate of 2 ml·min^{-1}, the result obtained for the molar reaction enthalpy is approximately constant for all experiments presented in Figure 23. An offset error of about 26% remains in all experiments. This may be due to possible flow disturbances in the tubes, leading to the flow rate of the pumps not being maintained, or due to the fact that the ambient conditions during the experiments do not correspond to those at which the calorimeter calibration is carried out.

Substance Data

Substance	Molar Mas	Weighing |	Volume [m	Concentration [mol/l]
a	60.05	6.1542	50	2.049692
b	40.01	4.1193	50	2.059135

Additional data

concentrat	55.34277
cp [J/(moll	75.336

Process setup

Process Pc	Evaluation	Evaluation	V_A [ml/m	V_A,act [m	n_A,act [m	n_A,act,wa	V_B [ml/m	V_B,act [m	n_B,act [m	n_B,act,water [mol/s]
1	258	377	0.2	0.198837	6.79E-06	0.000183	0.2	0.19934	6.84E-06	0.000184
2	439	559	0.4	0.397674	1.36E-05	0.000367	0.4	0.398679	1.37E-05	0.000368
3	621	738	0.6	0.59651	2.04E-05	0.00055	0.6	0.598019	2.05E-05	0.000552
4	860	979	0.2	0.198837	6.79E-06	0.000183	0.2	0.19934	6.84E-06	0.000184
5	1039	1159	0.4	0.397674	1.36E-05	0.000367	0.4	0.398679	1.37E-05	0.000368
6	1219	1338	0.6	0.59651	2.04E-05	0.00055	0.6	0.598019	2.05E-05	0.000552
7	1398	1518	0.8	0.795347	2.72E-05	0.000734	0.8	0.797359	2.74E-05	0.000735
8	1580	1699	1	0.994184	3.4E-05	0.000917	1	0.996698	3.42E-05	0.000919
9	1759	1879	1.5	1.491276	5.09E-05	0.001376	1.5	1.495048	5.13E-05	0.001379
10	1939	2040	2	1.988368	6.79E-05	0.001834	2	2.005856	6.88E-05	0.00185
11	2163	2283	2	1.988368	6.79E-05	0.001834	2	2.005856	6.88E-05	0.00185

Raw Data Processing

Process Pc	T_A [°C]	T_B [°C]	T_out [°C]	Upre [V]	Ur1 [V]	Ur2 [V]	dT_A [°C]	dT_B [°C]	dT_out [°C	Q_Out [W]
1	23.69434	24.05264	24.80774	-0.00124	0.018076	0.005151	1.30566	0.947358	0.192264	0.00532
2	23.78736	24.14528	24.80396	-0.00538	0.031603	0.030175	1.212642	0.854717	0.196038	0.010848
3	23.83314	24.3251	24.69235	-0.00866	0.067096	0.044828	1.166863	0.674902	0.307647	0.025536
4	23.9066	24.18849	24.71981	-0.00118	0.018385	0.009272	1.093396	0.811509	0.280189	0.007752
5	23.96755	24.24755	24.71906	-0.00285	0.031095	0.03087	1.032453	0.752453	0.280943	0.015546
6	24.00769	24.37923	24.62269	-0.00582	0.061545	0.046915	0.992308	0.620769	0.377308	0.031318
7	24.07038	24.51692	24.57135	-0.00857	0.109368	0.040437	0.929615	0.483077	0.428654	0.04744
8	24.12115	24.59442	24.51692	-0.00984	0.150808	0.044533	0.878846	0.405577	0.483077	0.066829
9	24.19736	24.71679	24.47528	-0.01251	0.242337	0.04564	0.802642	0.283208	0.524717	0.108885
10	24.22622	24.83133	24.50422	-0.01389	0.334343	0.060326	0.773778	0.168667	0.495778	0.137602
11	24.25308	24.65308	24.66808	-0.00763	0.254588	0.044943	0.746923	0.346923	0.331923	0.092125

Calculation

Process Pc	Q_A [W]	Q_B [W]	Qpre [W]	- QSE,pre [V	Qr1 [W]	Qr2 [W]	dHr [kJ/mol]
1	0.01804	0.013122	0.014668	-0.01649	-0.18678	-0.04414	-35.6399
2	0.033509	0.023678	0.048233	-0.00895	-0.30298	-0.25511	-40.9411
3	0.048366	0.028045	0.074855	-0.00156	-0.60787	-0.37897	-47.2508
4	0.015107	0.011241	0.014183	-0.01217	-0.18943	-0.07883	-40.1426
5	0.02853	0.020845	0.027712	-0.02166	-0.29861	-0.26097	-41.641
6	0.041131	0.025796	0.051807	-0.01512	-0.56018	-0.39663	-46.1591
7	0.051377	0.026766	0.074132	-0.00401	-0.97101	-0.34183	-46.7206
8	0.060713	0.028089	0.08443	-0.00437	-1.32701	-0.37647	-48.3182
9	0.083174	0.029422	0.106081	-0.00651	-2.11337	-0.38584	-47.0482
10	0.10691	0.023509	0.11723	-0.01319	-2.90387	-0.51024	-48.4309
11	0.1032	0.048355	0.066508	-0.08505	-2.21863	-0.37995	-38.1519

Figure 24: *Evaluation* worksheet containing the results of experiment 3a.

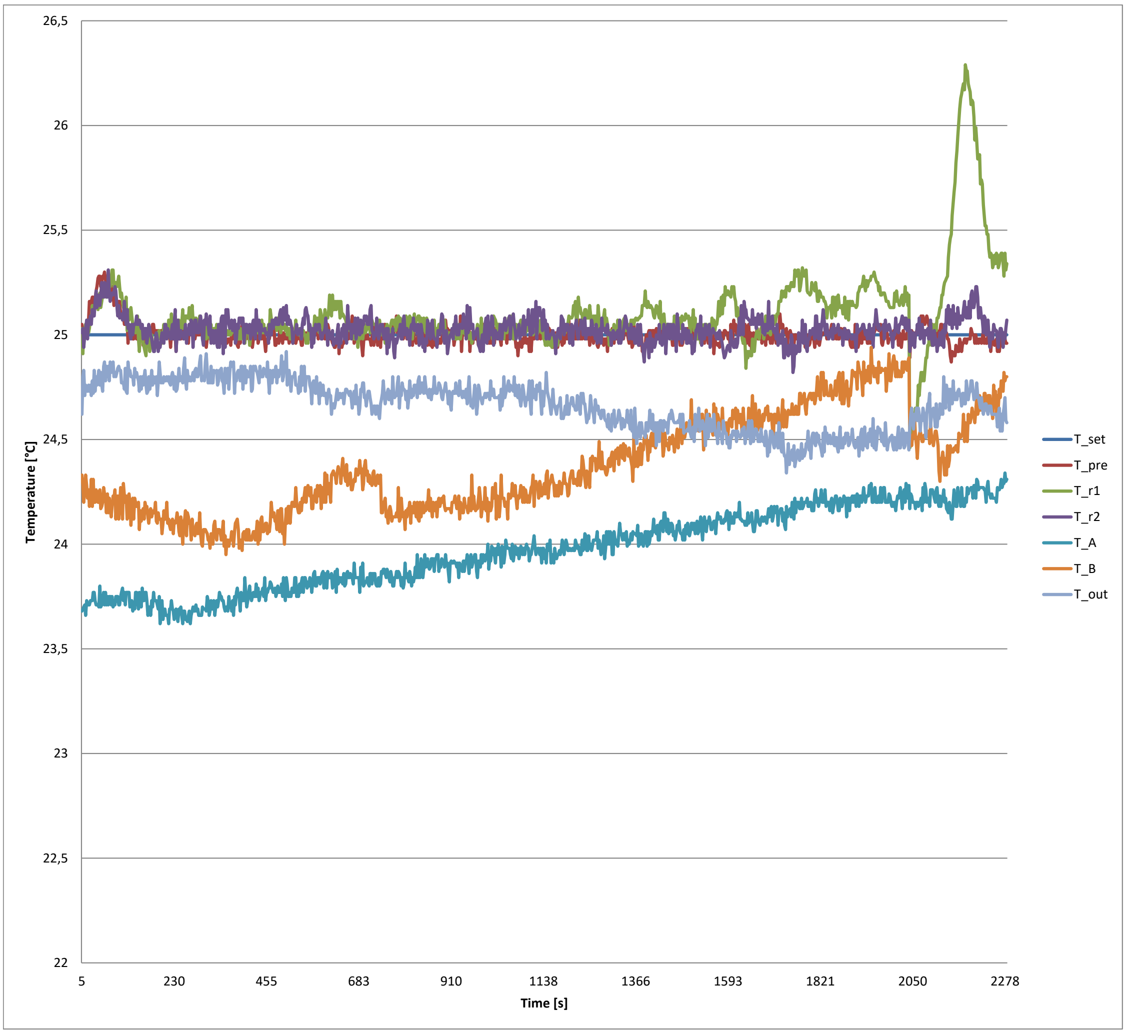


Figure 25: *Dia_Raw_Temp* worksheet displaying the plot of measured temperatures of experiment 3a over time.

5 Conclusion and Outlook

In this thesis, an application for the sequence control of a system setup revolving around the calorimeter developed by MAIER et al. is designed and implemented [1]. The process system consists of several elements to be controlled by the application. First, there is a variable number of pumps which convey one or more reagents to the calorimeter. Regarding the controllable pumps, two different types are available, a syringe pump and a HPLC pump. Following the pumps is the heat flow calorimeter as the reactor of the system setup. Finally, the thermostat is the last element to be controlled in the process system.

The application can be divided into several subroutines. For each individual device, there is a separate object that can be used to operate only the single device. These objects are each referenced as drivers and implemented using the concept of the state machines and the state design pattern. In general terms, the devices each have a state in which they are activated and a state in which they are deactivated. If the state change of activation or deactivation is rather complex, a separate state is implemented instead of a simple state change. If basic settings are to be set on the unit, there is a separate state for this purpose, which is only entered once at the beginning. For possible error handling, an error state is inserted for each state machine, which currently leads to the device or the system being shut down.

Additional subroutines deal with the implementation of the application using the strategy pattern. This means that the application has two files for the written strategies and one file for the executability of the individual strategies. The first strategy is very general and ensures that the application can be used in a very wide range. This means that there is no restriction on the number of pumps, which makes it possible to handle the system without pumps, with only one pump or to use pumps for the quench step. The strategy can also be used to create calibration curves for the calorimeter or the thermostat and calorimeter combination. The second strategy is more specific and limits the number of pumps used to at least two, one per reagent. Furthermore, it enables an automated evaluation of the measurement data, which is why the substance data must be specified at the beginning. Both, the strategies and the subroutine which executes the strategies and controls the individual device drivers are again realised using state machines.

The functionality of the written application is demonstrated by conducting neutralisation reactions at a temperature of 25°C. Acetic acid and sodium hydroxide at concentrations of 1.65 mM, 1 M and 2 M are used for this purpose. The results for the molar reaction enthalpy at 1.65 mM reveal that, regardless of the flow rate used, the concentration is too low to be adequately detected. Consequently, the results differ significantly from the literature value of $\Delta h_R = -57.4\ \mathrm{kJ \cdot mol^{-1}}$. The results at the two higher concentrations, gradually reach the literature value at a flow rate higher than 2 $\mathrm{ml{\cdot}min^{-1}}$ with a remaining deviation of 26 %.

By implementing the automatisation application by means of strategies, a high degree of flexibility is provided for the use and expansion of the application. With regard to the use of the application, the simple strategy provides a wide range in which the system can be used automatically, and the more specific strategy is reduced to a reaction between two reactants with known substance data, but with the advantage of also having the evaluation of the measurement data automated. Regarding the extension of the application, it is possible to add an infinite number of further strategies for various problems by adopting the structure of a strategy given in this thesis. The advantage of the new strategies is that the remaining part of the application does not have to be changed, so that previous strategies can still be used.

A possible extension, which is also directly related to the intended use of the written application, is the addition of an optimisation algorithm. For this purpose, the specific strategy could be adopted and the optimisation added. The optimisation leads to the calculation of the optimal operating point from the measured data, which is subsequently set. Another possible development of the application results from the current implementation of error handling. All errors are currently handled in the same way and result in the system being shut down in a safe, predefined sequence. If necessary, frequently occurring errors can be identified and, if possible, handled differently so that it is not always necessary to restart the whole system. In the end, the development of another calorimeter driver should be mentioned as a possible development, which can not only handle the calorimeter with the three segments, but also those with more segments. In this context, it would be useful to design the driver in such a way that the number of segments present in the calorimeter can be variable for the driver.

6 Experimental Procedure

6.1 Details on conducting the Experiments

For the neutralisation experiment, acetic acid from the company *Sigma-Aldrich* with a purity of 99.8 % and sodium hydroxide pellets from the company *Carl Roth* with a purity of $\geq$ 99 % are used. Since this is a neutralisation, the same concentration is used for the acid and the base. The three concentrations of 1.65 mM, 1 M and 2 M are selected, and the experiments are referred to as the first, second and third experiment, respectively. The third experiment is performed twice, which is why a distinction is made between experiment 3a and 3b.

Preparation of the Solutions

At the beginning of each experiment, the solutions corresponding to the selected concentration are prepared. For this purpose, the quantity (volume) in which the solution is to be prepared is defined beforehand. Then the mass of the substance to be weighed in can be calculated as follows:

$$m\,[\mathrm{g}] = V\,[\mathrm{ml}] \cdot c\,[\mathrm{mol{\cdot}l^{-1}}] \cdot M\,[\mathrm{g{\cdot}mol^{-1}}] \cdot 10^{-3} \tag{7}$$

Whereby m refers to the mass to be weighed, V to the total volume of the solution, c to the concentration of the solution and M to the molar mass of the substance used. The weighed mass is placed in a volumetric flask, which is subsequently filled with deionised water. After reformulating the Equation 7, the actual concentration can be calculated from the weighed-in mass.

In the laboratory, first the sodium hydroxide solution is prepared. The actual concentration of the basic solution is determined from the weighed mass and from this in turn the necessary mass of acetic acid is calculated. Since acetic acid is a liquid, the mass is converted into the necessary volume by means of its density of $\rho = 1.049\ \mathrm{g \cdot ml^{-1}}$. The acetic acid is then

Table 4: Concentrations of the prepared solutions for the various experiments in $\mathrm{mol{\cdot}l^{-1}}$.

	1st experiment	**2nd experiment**	**3rd experiment (a)**	**3rd experiment (b)**
acetic acid	$1.6719 \cdot 10^{-3}$	1.0087	2.0497	2.0323
sodium hydroxide	$1.6496 \cdot 10^{-3}$	1.0067	2.0591	2.0326

pipetted, and the pipetted quantity is weighed and used for the subsequent calculation of the actual concentration. The actual concentrations prepared for the individual experiments are given in Table 4 in the unit of mol·l^{-1}.

Experimental Setup used

A detailed overview of the general setup of the system is given in section 2.4. That section also provides details on the thermostats, the two different types of pumps and the calorimeter.

When conducting the experiment, the calorimeter and the thermostat are always present. The only difference are the pumps used in each experiment. Since the application is designed to fit the equipment available in the laboratory, it is worth noting that there are three pumps for each of the two types (HPLC or Lambda). Therefore they are referenced as A, B and C in the case of the HPLC pumps and 1, 2 and 3 in the case of the Lambda pumps. This distinction is particularly important for the Lambda pump, as each pump has a different calibration curve between the set rotation speed and the flow rate. For the HPLC pump, attention must be paid to the different pump heads, which are automatically selected correctly when the explicit HPLC pump is selected in the application. Details of the pumps used for the individual experiments 1, 2, 3a and 3b are given in Table 5.

Table 5: Pumps used for the various experiments.

	1st experiment	**2nd experiment**	**3rd experiment**	
			(a)	**(b)**
acetic acid	Lambda 1	Lambda 2, Lambda 3	Lambda 1	HPLC A
sodium hydroxide	Lambda 2	HPLC B	Lambda 2	HPLC B

Performing the Experiment by means of the Application

For the two experiments 3a and 3b, the completed sequence control application given in the Appendix 8.1 is used. For the other experiments, a previous version of the application, or only parts of the application, are used. After the solutions have been prepared and the equipment has been set up, the specifications for conducting the measurement are entered into the application. The procedure corresponds to the one described in section 4.2.

First, the list of operating points is defined. An operating point consists of the operating point duration, the set temperature and the list of flow rates. In the case of the time period, a duration of 3 minutes is chosen for all experiments and the set temperature is always set to 25°C. For the flow rates, values in a range from 0.2 to 4 ml·min^{-1} for each pump are selected. In the next step, the devices used and their port names are specified. For this purpose, the pump names are first assigned according to the experiment (Table 5). For the thermostat and calorimeter, only the port name is required.

6.2 Calorimeter-Thermostat Calibration

A target temperature is to be set on the calorimeter. The thermostat is used to supply the calorimeter with a temperature higher than the target temperature. The temperature is then adjusted at the calorimeter by further cooling using a PELTIER element until the target temperature is reached. The thermostat is used because cooling down only by means of the PELTIER element during the reaction is not practical. The temperature must always be above the set temperature, as the PELTIER element used can only work in one direction, i.e. cooling. The built-in control unit of the calorimeter thus becomes active once the temperature measured at the reactors is above the target temperature for the first time.

When conducting various experiments, it is therefore important to know for which set temperature on the calorimeter which temperature must be set on the thermostat. Furthermore, it is important to note that heat is lost through the thermostat's hoses. This heat loss increases as the temperature rises, which is why a uniform temperature difference between target temperature and thermostat temperature is not practical and instead a calibration curve is created for this purpose.

As a design choice during implementation, only integers are to be set for the thermostat temperature in terms of the calibration curve. Thus, instead of a continuous function, a discrete function is created for the calibration. For this purpose, the discrete values 25, 30, 35 and 40 are selected to cover the interval from 25 to 40°C, which are set as target temperature on the calorimeter. In each case, the same temperature is set at the thermostat and increased until the target temperature is actually reached at the calorimeter. The resulting diagram is given in Figure 26.

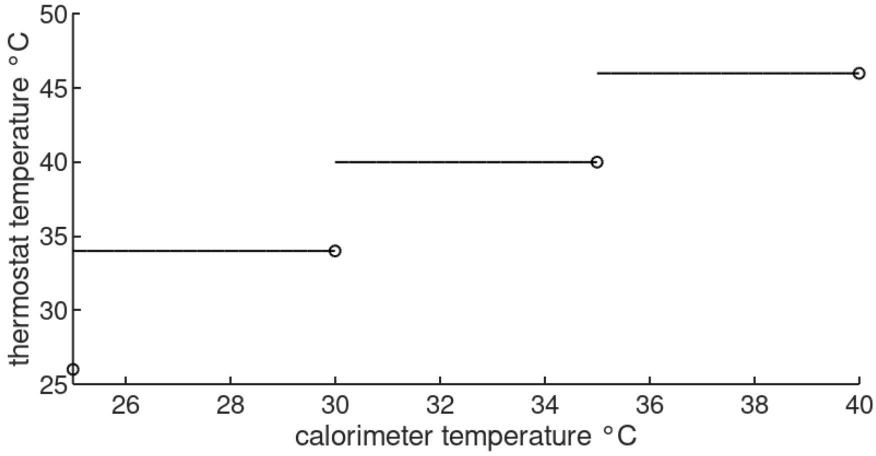


Figure 26: Calibration curve between thermostat and calorimeter in the interval of 25 to 40°C.

6.3 Calorimeter Calibration

In order to determine the heat quantity using the SEEBECK effect, a quadratic calibration curve between the measured voltage and heat quantity is required. A calibration is always valid for the combination of the set temperature at the calorimeter and the corresponding temperature at the thermostat.

For calibration, a power supply is connected to the heat foils integrated in the calorimeter and various voltage and current tuples are set. Through this introduced power, the foils heat up and simulate the formation of a heat flux resulting from a chemical reaction. The PELTIER element of the calorimeter acts in the same way as in normal operation mode and dissipates the heat generated to keep the calorimeter at a constant temperature. The voltage resulting due to the heat flux is measured via the SEEBECK element. The calibration curve can then be generated from the known power input and the measured voltage.

The calibration curves are determined for the set temperatures at the calorimeter of 25, 30, 35 and 40°C (and for the temperatures at the thermostat derived from the calorimeter-thermostat calibration curve, cf. Figure 26). The resulting parameter tuples *a*, *b* and *c* for the quadratic function at the different temperatures are given in Table 6. The graphical illustration of the calibration curve based on the measured data is given as an example in Figure 27.

Table 6: Parameters of the quadratic calibration curve for various temperature combinations of calorimeter and thermostat.

calorimeter: 25°C; thermostat: 26°C			
	precooling	1st reactor	2nd reactor
a	0.6518	0.4091	0.4039
b	8.3911	8.5318	8.7333
c	-0.0598	0.0261	-0.0475

calorimeter: 35°C; thermostat: 40°C			
	precooling	1st reactor	2nd reactor
a	0.4921	0.6368	0.3782
b	8.3027	8.1373	8.5837
c	0.1518	0.1511	0.1580

calorimeter: 30°C; thermostat: 34°C			
	precooling	1st reactor	2nd reactor
a	0.9217	0.6194	0.7441
b	8.0548	8.1511	8.3637
c	0.0291	0.0944	0.0420

calorimeter: 40°C; thermostat: 46°C			
	precooling	1st reactor	2nd reactor
a	0.7618	0.7157	0.4186
b	7.8524	7.8297	8.2974
c	0.2782	0.2332	0.2775

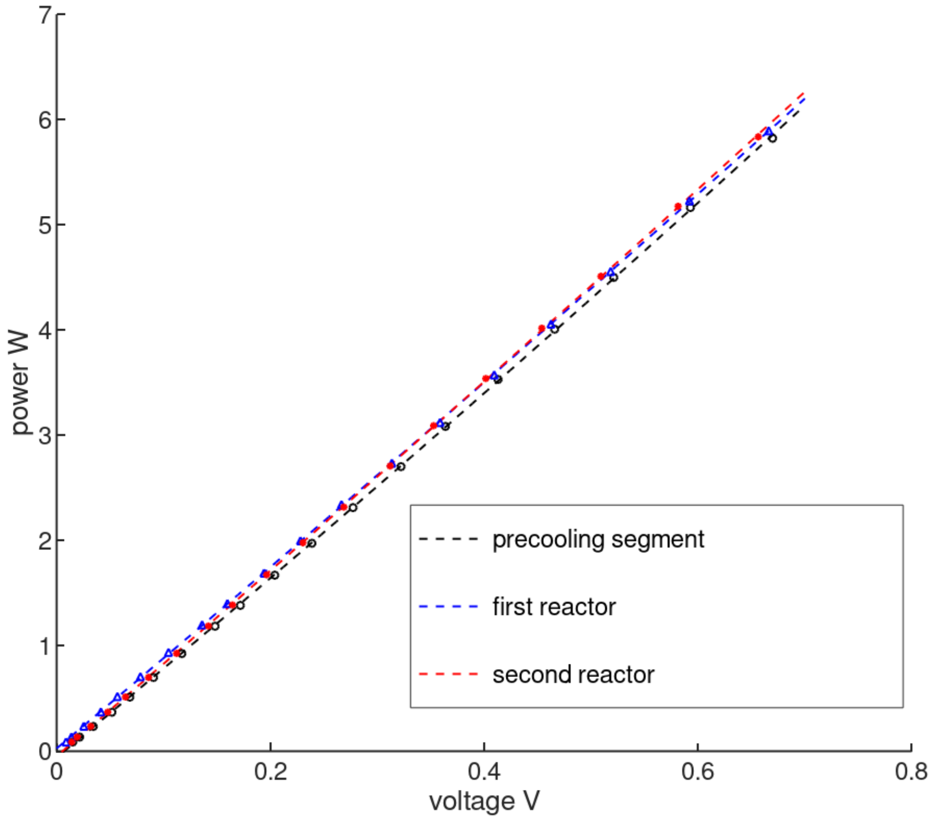


Figure 27: Calibration curve to convert measured voltage into heat quantity for the temperature combination of 25°C at the calorimeter and 26°C at the thermostat.

7 References

[1] M. C. Maier, M. Leitner, C. O. Kappe, and H. Gruber-Woelfler. A modular 3d printed isothermal heat flow calorimeter for reaction calorimetry in continuous flow. *Reaction Chemistry & Engineering*, 5(8):1410–1420, 2020.

[2] K. F. Jensen. Flow chemistry—microreaction technology comes of age. *AIChE Journal*, 63(3):858–869, 2017.

[3] J. Deng, J. Zhang, K. Wang, and G. Luo. Microreaction technology for synthetic chemistry. *Chinese Journal of Chemistry*, 37(2):161–170, 2019.

[4] O. Levenspiel. *Chemical reaction engineering*. John Wiley & Sons, 1999.

[5] P. Kleinebudde, J. Khinast, and J. Rantanen. *Continuous manufacturing of pharmaceuticals*. John Wiley & Sons, 2017.

[6] D. J. am Ende and M. T. am Ende. *Chemical Engineering in the Pharmaceutical Industry, Active Pharmaceutical Ingredients*. Wiley, 2019.

[7] M. B. Plutschack, B. Pieber, K. Gilmore, and P. H. Seeberger. The hitchhiker's guide to flow chemistry ii. *Chemical reviews*, 117(18):11796–11893, 2017.

[8] B. J. Reizman and K. F. Jensen. Feedback in flow for accelerated reaction development. *Accounts of chemical research*, 49(9):1786–1796, 2016.

[9] S. M. Sarge, G. W. H. Höhne, and W. Hemminger. *Calorimetry: fundamentals, instrumentation and applications*. John Wiley & Sons, 2014.

[10] S. B. Warrington and G. W. H. Höhne. Thermal analysis and calorimetry. *Ullmann's Encyclopedia of Industrial Chemistry*, 2000.

[11] A. Zogg, F. Stoessel, U. Fischer, and K. Hungerbühler. Isothermal reaction calorimetry as a tool for kinetic analysis. *Thermochimica acta*, 419(1-2):1–17, 2004.

[12] C. Eveleigh. Literature review of isothermal reaction calorimetry. *University of Ottawa*, 2016.

[13] Mettler Toledo. *RC1mx Reaction Calorimeter. Documentation*.

[14] H. Bannwarth, B. P. Kremer, and A. Schulz. *Basiswissen Physik, Chemie und Biochemie*. Springer, 2007.

[15] E. Riedel and H.-J. Meyer. *Allgemeine und anorganische Chemie*. de Gruyter, 2018.

[16] Fisher Scientific. *Isotemp® Circulators/Baths. User's Manual*. December 15, 2016.

[17] Knauer. *AZURA Pump P 2.1S/P 4.1S Instructions*. Document No. V6870.

[18] Lambda Laboratory Instruments. *Operation Manual Syringe Pump - Infusion Pump*.

[19] T. Flik. *Mikroprozessortechnik und Rechnerstrukturen*. Springer-Verlag, 2005.

[20] C. Kecher, A. Salvanos, and R. Hoffmann-Elbern. *UML 2.5: Das umfassende Handbuch*. Rheinwerk Verlag, 2017.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.

8 Appendix

8.1 Application Code

8.1.1 Auto.py

```python
# This file contains the state machine class for the automatization
# and the following additional relevant classes and functions for
# the construction of this class: the required state classes, which
# can be assigned to the layers (A) and (B), and functions for
# generating the drivers of each device.

# library/modules from python:
import time

# own scripts:
import Calorimeter
import Communication
import Dictionary
import Fisher
import HPLC
import Lambda
import pyState

# layer (B) states:
class Set_Operating_Point(pyState.State_Base):
    def enter(self, operating_point_strategy, pump_list, thermostat, calorimeter, operation_point):
        super().enter("Set_Operating_Point")
        print("entering state Set_Operating_Point")
        self.operating_point_strategy = operating_point_strategy
        self.pump_list = pump_list
        self.thermostat = thermostat
        self.calorimeter = calorimeter
        self.operation_point = operation_point

    def __call__(self):
        current_temp = self.operation_point.get_temperature()
        self.calorimeter.set_target_Temp(current_temp)
        self.thermostat.set_target_temp(Dictionary.calorimeter_thermostat["calibration"].forward(current_temp))
        self.thermostat.activate_pump()

        pump_actual_flowrates = []
        for idx in range(len(self.pump_list)):
            tmp_flowrate = self.operation_point.get_flowrate(idx)
            pump_actual_flowrates.append(self.pump_list[idx].set_target_flowrate(tmp_flowrate))
            if tmp_flowrate > 0.0:
                self.pump_list[idx].activate_pump()
            else:
                self.pump_list[idx].deactivate_pump()

        self.operating_point_strategy.push_actual_flowrate(pump_actual_flowrates)
        return "next"

class Operating(pyState.State_Base):
    def enter(self, operating_point_strategy, calodata, calodata_idx):
        super().enter("Operating")
        print("entering state Operating")
        self.operating_point_strategy = operating_point_strategy
        self.calodata = calodata
        self.calodata_idx = calodata_idx

    def __call__(self):
        push_empty = True
        while self.calodata_idx[0] < len(self.calodata):
            push_empty = False
            self.operating_point_strategy.push_value(self.calodata[self.calodata_idx[0]])
            self.calodata_idx[0] += 1
        if push_empty:
            self.operating_point_strategy.push_value(None)

        if self.operating_point_strategy.has_error():
            return "error"

        if self.operating_point_strategy.point_complete():
            return "next"

# layer (A) states:
class Apply_Configuration(pyState.State_Base):
    """This state runs the configuration state of all devices and puts them all in a deactivated mode."""
    def enter(self, pump_list, thermostat, calorimeter, calodata):
        super().enter("Apply_Configuration")
        print("entering state Apply_Config")
        self.pump_list = pump_list
        self.thermostat = thermostat
        self.calorimeter = calorimeter
```

```python
        self.calodata = calodata

        self.deadline = time.monotonic_ns() + 60 * 1E9

    def __call__(self):

        for itm in self.pump_list:
            itm.tick()

        self.thermostat.tick()
        self.calorimeter.tick()

        if self.deadline < time.monotonic_ns():
            return "error"

        if self.thermostat.get_state() == "Error":
            return "error_thermostat"

        if self.calorimeter.get_state() == "Error":
            return "error_calorimeter"

        for itm in self.pump_list:
            if itm.get_state() == "Error":
                return "error_pump"

        for itm in self.pump_list:
            if not itm.get_state() == "Deactivated":
                return None

        if not self.thermostat.get_state() == "Deactivated":
            return None

        if len(self.calodata) == 0:
            return None

        return "next"

class List_Processing(pyState.State_Base):
    """This state ensures the processing of the operating points."""
    class factory:
        def __init__(self, pump_list, thermostat, calorimeter, calodata, operating_point_strategy):
            self.pump_list = pump_list
            self.thermostat = thermostat
            self.calorimeter = calorimeter
            self.operating_point_strategy = operating_point_strategy
            self.calodata = calodata
            self.calodata_idx = [len(calodata)]

        def create_state(self, state_name):
            if state_name == "Set_Operating_Point":
                tmp_operation_point = self.operating_point_strategy.get_operation_point()
                if tmp_operation_point is not None:
                    st = Set_Operating_Point()
                    st.enter(self.operating_point_strategy, self.pump_list, self.thermostat, self.calorimeter, tmp_operation_point)
                    return st
                else:
                    st = pyState.State_Base()
                    st.enter("Finished")
                    return st
            elif state_name == "Operating":
                st = Operating()
                st.enter(self.operating_point_strategy, self.calodata, self.calodata_idx)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, pump_list, thermostat, calorimeter, calodata, operating_point_strategy):
        super().enter("List_Processing")
        print("entering state List_Processing")
        self.tab = [
            ["Set_Operating_Point", "next",    "Operating"],
            ["Operating",           "next",    "Set_Operating_Point"],
            ["Operating",           "error",   "Error"],
            ]
        self.pump_list = pump_list
        self.thermostat = thermostat
        self.calorimeter = calorimeter
        self.fac = List_Processing.factory(pump_list, thermostat, calorimeter, calodata, operating_point_strategy)
        self.en = pyState.Engine(self.tab, self.fac, "Set_Operating_Point")
        self.en.enter()

    def __call__(self):
        for itm in self.pump_list:
            itm.tick()
        self.thermostat.tick()
        self.calorimeter.tick()

        self.en.tick()

        if self.thermostat.get_state() == "Error":
            return "error_thermostat"
```

```python

        if self.calorimeter.get_state() == "Error":
            return "error_calorimeter"

        for itm in self.pump_list:
            if itm.get_state() == "Error":
                return "error_pump"

        if self.en.get_state() == "Error":
            return "error"

        if self.en.get_state() == "Finished":
            return "next"

    def exit(self):
        self.en.exit()
        super().exit()

class Deactivating(pyState.State_Base):
    """This state first switches off the pumps and then the thermostat. It is used for the regular shutdown as well as for
     the error shutdown."""
    def enter(self, state_name, leave_thermostat_on, pump_list, thermostat, calorimeter, operating_point_strategy,
     next_state):
        super().enter(state_name)
        print("entering state", state_name)

        if state_name.find("Error") != -1:
            if state_name == "Shutdown_Error_Pump":
                for itm in pump_list:
                    if itm.get_state() == "Error":
                        print(next_state, "from pump", itm.get_name())
            else:
                print(next_state)

        self.leave_thermostat_on = leave_thermostat_on
        self.pump_list = pump_list
        self.thermostat = thermostat
        self.calorimeter = calorimeter
        self.next_state = next_state

        self.counter = 0
        for itm in self.pump_list:
            itm.deactivate_pump()

        operating_point_strategy.get_finish_instruction()

    def __call__(self):

        for itm in self.pump_list:
            itm.tick()

        self.thermostat.tick()
        self.calorimeter.tick()

        if self.name == self.next_state:
            return None

        if self.counter == 0:
            for itm in self.pump_list:
                if not (itm.get_state() == "Deactivated" or itm.get_state() == "Error"):
                    return None

            if self.leave_thermostat_on:
                self.name = self.next_state
                return None
            else:
                self.counter = 1
                return None

        if self.counter == 1:
            self.thermostat.deactivate_pump()
            self.counter = 2
            return None

        if self.counter == 2:
            if not (self.thermostat.get_state() == "Deactivated" or self.thermostat.get_state() == "Error"):
                return None
            else:
                self.name = self.next_state
                return None

# generate the drivers:
def generate_hplc(name, port, calibration_func, head, _PMin_ = None, _PMax_ = None):
    if head is None:
        raise Exception("No pump head is given")
    settings = HPLC.Driver.Settings(head)
    if not _PMin_ is None and not _PMax_ is None:
        settings.set_PMinMax(_PMin_, _PMax_)

    ch = Communication.Handle(port, 9600, Communication.Handle.PARITY_NONE, 1)
    return HPLC.Driver(name, settings, calibration_func, ch) # HPLC.dummy_cmd_handle()

def generate_lambda(name, port, calibration_func, address):
    ch = Communication.Handle(port, 2400, Communication.Handle.PARITY_ODD, 1)
    return Lambda.Driver(name, address, calibration_func, ch) # Lambda.dummy_cmd_handle()
```

```
def generate_fisher(port, _pump_speed = None, _ext_probe = None):
    settings = Fisher.Driver.Settings()
    if not _pump_speed is None:
        settings.set_pump_speed(_pump_speed)
    if not _ext_probe is None:
        settings.set_external_probe(_ext_probe)

    ch = Communication.Handle(port, 9600, Communication.Handle.PARITY_NONE, 1)
    return Fisher.Driver("Fisher", settings, ch) # Fisher.dummy_cmd_handle())

def generate_calorimeter(port, datalist):
    ch = Communication.Handle(port, 9600, Communication.Handle.PARITY_NONE, 1)
    return Calorimeter.Driver("Calo", datalist, ch)

def initialize_thermostat(thermostat_specification):
    if len(thermostat_specification) == 0:
        raise Exception("There is no given specification for the thermostat")
    if len(thermostat_specification) == 1:
        return generate_fisher(thermostat_specification[0])

    _pump_speed = None
    _ext_probe = None

    num = len(thermostat_specification)-1
    for idx in range(num):
        if type(thermostat_specification[idx+1]) == str:
            _pump_speed = thermostat_specification[idx+1]
        elif type(thermostat_specification[idx+1]) == int or type(thermostat_specification[idx+1]) == float:
            _ext_probe = thermostat_specification[idx+1]
        else:
            raise Exception("Given thermostat setting cannot be handled")

    return generate_fisher(thermostat_specification[0], _pump_speed, _ext_probe)

def initialize_single_pumpdriver(pump_cfg_entry):
    length = len(pump_cfg_entry)
    if length < 2:
        raise Exception("Invalid list entry")

    pump_name = pump_cfg_entry[0]

    if pump_name == "HPLC A" or pump_name == "HPLC B" or pump_name == "HPLC C":
        if length == 2:
            return generate_hplc(pump_name, pump_cfg_entry[1], Dictionary.pump_calibration[pump_name], Dictionary.pump_head[pump_name])
        if length == 4:
            return generate_hplc(pump_name, pump_cfg_entry[1], Dictionary.pump_calibration[pump_name], Dictionary.pump_head[pump_name], pump_cfg_entry[2], pump_cfg_entry[3])
        raise Exception("Invalid list entry")

    if pump_name == "Lambda 1" or pump_name == "Lambda 2" or pump_name == "Lambda 3":
        return generate_lambda(pump_name, pump_cfg_entry[1], Dictionary.pump_calibration[pump_name], Dictionary.lambda_address[pump_name])
    raise Exception("Invalid list entry")

def initialize_all_pumpdrivers(pump_list):

    ret = []
    for itm in pump_list:
        ret.append(initialize_single_pumpdriver(itm))
    return ret

# state machine class for the automatization:
class matization:

    class factory:
        def __init__(self, operating_point_strategy, pump_list, thermostat, calorimeter, calodata):
            self.operating_point_strategy = operating_point_strategy
            self.pump_list = pump_list
            self.thermostat = thermostat
            self.calorimeter = calorimeter
            self.calodata = calodata

        def create_state(self, state_name):
            if state_name == "Apply_Configuration":
                st = Apply_Configuration()
                st.enter(self.pump_list, self.thermostat, self.calorimeter, self.calodata)
                return st
            elif state_name == "List_Processing":
                st = List_Processing()
                st.enter(self.pump_list, self.thermostat, self.calorimeter, self.calodata, self.operating_point_strategy)
                return st
            elif state_name == "Finished":
                st = Deactivating()
                st.enter("Shutdown_{}".format(state_name), False, self.pump_list, self.thermostat, self.calorimeter, self.operating_point_strategy, state_name)
                return st
            elif state_name == "Error" or state_name == "Error_Calorimeter" or state_name == "Error_Thermostat" or state_name == "Error_Pump":
                st = Deactivating()
                st.enter("Shutdown_{}".format(state_name), False, self.pump_list, self.thermostat, self.calorimeter, self.operating_point_strategy, state_name)
                return st
```

```
            raise Exception("Unhandled State in Factory")

    def __init__(self, operating_point_strategy, User_Pumps, User_Fisher, Portname_Calorimeter):
        self.tab = [
            ["Apply_Configuration",        "next",      "List_Processing"],
            ["List_Processing",            "next",      "Finished"],

            ["Apply_Configuration",        "error_calorimeter",     "Error_Calorimeter"],
            ["List_Processing",            "error_calorimeter",     "Error_Calorimeter"],

            ["Apply_Configuration",        "error_thermostat",      "Error_Thermostat"],
            ["List_Processing",            "error_thermostat",      "Error_Thermostat"],

            ["Apply_Configuration",        "error_pump",            "Error_Pump"],
            ["List_Processing",            "error_pump",            "Error_Pump"],

            ["Apply_Configuration",        "error",                 "Error"],
            ["List_Processing",            "error",                 "Error"],
            ]

        self.calodata = []

        self.thermostat  = initialize_thermostat(User_Fisher)
        self.calorimeter = generate_calorimeter(Portname_Calorimeter, self.calodata)
        self.pump_list = initialize_all_pumpdrivers(User_Pumps)

        self.fac = matization.factory(operating_point_strategy, self.pump_list, self.thermostat, self.calorimeter, self.
      calodata)
        self.en = pyState.Engine(self.tab, self.fac, "Apply_Configuration")
        self.en.enter()

    def tick(self):
        self.en.tick()

    def __del__(self):
        self.en.exit()

    def get_state(self):
        return self.en.get_state()
```

Listing 11: The *Auto.py* file corresponds to the context of the strategy pattern and is therefore responsible for the execution of the individual strategies.

8.1.2 Calibration.py

```
# This file contains classes with the corresponding calibration
# for each device. The calibration values are specified in the
# dictionary.
class Pumps:
    def __init__(self, cali_val):
        self.cali_val = cali_val

    def __call__(self, value):
        return value * self.cali_val

    def forward(self, value):
        return value * self.cali_val

    def backward(self, value):
        return value/self.cali_val

class Thermostat:
    """This class ensures that the correct temperature is selected at the thermostat for a given set temperature of the
      calorimeter."""
    def __init__(self, lower_limit, upper_limit, list):
        self.list = sorted(list, key=lambda entry: entry[0])
        self.upper_limit = upper_limit
        self.lower_limit = lower_limit

    def forward(self, value):
        if value < self.lower_limit:
            raise Exception("Calorimeter set temperature is too low")
        if not value <= self.upper_limit:
            raise Exception("Calorimeter set temperature is too high")

        for itm in self.list:
            if value <= itm[0]:
                return itm[1]

        raise Exception("Given set temperature is above the given calibration table")

class Calorimeter:
    def __init__(self, list):
       self.list = list
       if not len(self.list) == 3:
           raise Exception("Given parameter list is incomplete")

    def forward(self, value_list):
        tmp = []
```

```
        if not len(value_list) == 3:
            raise Exception("Given evaluation data list is incomplete")

        for idx in range(3):
            tmp.append((self.list[idx][0]*value_list[idx]*value_list[idx]+self.list[idx][1]*value_list[idx]+self.list[idx][2])*(-1))

        return tmp
```

Listing 12: The *Calibration.py* file contains the calibration equation for the pumps, calorimeter and calorimeter-thermostat combination.

8.1.3 Calorimeter.py

```
# This file contains the driver class for the calorimeter and the
# required state classes, which can be assigned to the layers (A)
# and (B).

# library/modules from python:
import math
import os
import re
import time

# own scripts:
import pyState

# layer (B) states:
class Read_Data(pyState.State_Base):
    """This state queries and stores the data from the calorimeter."""
    def enter(self, name, path, datalist, timeout_error_s, timeout_check_s, com_handle):
        super().enter(name)
        self.com_handle = com_handle
        self.pattern_values = re.compile(r"(\S+)\t(\S+)\t(\S+)\t(\S+)\t(\S+)\t(\S+)\t(\S+)\t(\S+)\t(\S+)\t(\S+)\t(\S+)\t(\S+)\t(\S+).*[\r\n]")
        self.pattern_line_complete = re.compile(r"(.+)\n")

        self.deadline_error_delta = timeout_error_s * 1E9
        self.deadline_error = time.monotonic_ns() + self.deadline_error_delta
        self.deadline_check = time.monotonic_ns() + timeout_check_s * 1E9

        self.datalist = datalist
        self.out_path = path
        self.current_line = ""

    def __call__(self):
        self.current_line = self.current_line + self.com_handle.receive().decode('utf-8')
        tmp = self.pattern_line_complete.match(self.current_line)

        while not tmp is None:
            self.deadline_error = time.monotonic_ns() + self.deadline_error_delta
            tmp_val = self.pattern_values.match(tmp.group(0))
            if not tmp_val is None:
                with open(self.out_path, 'a') as fout:
                    fout.write(tmp.group(1))

                    line = []
                    for idx in range(11):
                        line.append(float(tmp_val.group(idx+1)))
                    self.datalist.append(line)

            self.current_line = self.current_line[tmp.end():]
            tmp = self.pattern_line_complete.match(self.current_line)

        if self.deadline_check < time.monotonic_ns():
            return "check"
        if self.deadline_error < time.monotonic_ns():
            return "error"
        return None

class Check_Set_Temp(pyState.State_Base):
    """This state checks the set temperature."""
    def enter(self, name, datalist, set_Temp):
        super().enter(name)
        self.list_Temp = float(datalist[len(datalist)-1][1])
        self.set_Temp = set_Temp

    def __call__(self):
        if math.isnan(self.set_Temp[0]):
            return "next"
        if float(self.set_Temp[0]) == self.list_Temp:
            return "next"
        return "error"

# layer (A) states:
class Clear(pyState.State_Base):
    """This state ensures a proper starting point for the calorimeter."""
    def enter(self, name, path, com_handle):
```

```
        super().enter(name)
        self.com_handle = com_handle
        self.out_path = path

    def __call__(self):
        if os.path.isfile(self.out_path):
            os.remove(self.out_path)

        self.com_handle.clear_input_buffer()

        return "next"

class Read_And_Check(pyState.State_Base):
    """This state combines the substates "Read_Data" and "Check_Set_Temp"."""
    class factory:
        def __init__(self, path, datalist, set_Temp, com_handle):
            self.datalist = datalist
            self.set_Temp = set_Temp
            self.com_handle = com_handle
            self.out_path = path

        def create_state(self, state_name):
            if state_name == "Read":
                st = Read_Data()
                st.enter(state_name, self.out_path, self.datalist, 7, 10, self.com_handle)
                return st
            elif state_name == "Check":
                st = Check_Set_Temp()
                st.enter(state_name, self.datalist, self.set_Temp)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, path, datalist, target_Temp, set_Temp, com_handle):
        super().enter(name)
        self.target_Temp = target_Temp
        self.set_Temp = set_Temp
        self.tab = [
            ["Read",    "check",      "Check"],
            ["Read",    "error",      "Error"],
            ["Check",   "next",       "Read"],
            ["Check",   "error",      "Error"],
            ]
        self.fac = Read_And_Check.factory(path, datalist, self.set_Temp, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Read")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Error":
            return "error"

        if not self.en.get_state() == "Read":
            return None

        if math.isfinite(self.target_Temp[0]) and not self.target_Temp[0] == self.set_Temp[0]:
            self.set_Temp[0] = self.target_Temp[0]
            return "new_set_Temp"

    def exit(self):
        self.en.exit()
        super().exit()

class Set_Temp(pyState.State_Base):
    """This state sets a new set temperature at the calorimeter."""
    def enter(self, name, set_Temp, com_handle):
        super().enter(name)
        self.com_handle = com_handle
        self.set_Temp = set_Temp

    def __call__(self):
        if math.isnan(self.set_Temp[0]):
            return "next"

        text = "<1,{:02.2f}>".format(float(self.set_Temp[0]))
        com = bytearray(text.encode('utf-8'))
        self.com_handle.send(com)
        return "next"

# driver class for the calorimeter:
class Driver:

    class factory:
        def __init__(self, datalist, target_Temp, set_Temp, com_handle):
            self.target_Temp = target_Temp
            self.set_Temp = set_Temp
            self.com_handle = com_handle

            self.datalist = datalist
            self.out_path = "test.log"
```

```python

        def create_state(self, state_name):
            if state_name == "Clear":
                st = Clear()
                st.enter(state_name, self.out_path, self.com_handle)
                return st
            elif state_name == "Read_And_Check":
                st = Read_And_Check()
                st.enter(state_name, self.out_path, self.datalist, self.target_Temp, self.set_Temp, self.com_handle)
                return st
            elif state_name == "Set_Temp":
                st = Set_Temp()
                st.enter(state_name, self.set_Temp, self.com_handle)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def __init__(self, name, datalist, com_handle):
        self.name = name
        self.tab = [
            ["Clear",           "next",          "Read_And_Check"],
            ["Read_And_Check",  "new_set_Temp",  "Set_Temp"],
            ["Read_And_Check",  "error",         "Error"],
            ["Set_Temp",        "next",          "Read_And_Check"],
            ]
        self.target_Temp = [float("nan")]
        self.set_Temp = [float("nan")]

        self.fac = Driver.factory(datalist, self.target_Temp, self.set_Temp, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Clear")
        self.en.enter()

    def tick(self):
        self.en.tick()

    def __del__(self):
        self.en.exit()

    def get_state(self):
        return self.en.get_state()

    def get_name(self):
        return self.name

    # In the following, the functions are defined to obtain the settings for the calorimeter (from outside).
    def set_target_Temp(self, val):
        self.target_Temp[0] = val
```

Listing 13: The *Calorimeter.py* file contains its corresponding device driver and can be used to operate this device.

8.1.4 Communication.py

```python
# This file contains the serial interface. For this purpose, the
# library "serial" is used and adapted in the class "Handle" for
# the own application.

# library/modules from python:
import serial

class Handle:
    PARITY_NONE = serial.PARITY_NONE
    PARITY_EVEN = serial.PARITY_EVEN
    PARITY_ODD = serial.PARITY_ODD

    def __init__(self, port, baudrate, parity, stopbits):
        self.com = serial.Serial(port, baudrate, 8, parity, stopbits, timeout=0)

    def clear_input_buffer(self):
        self.com.reset_input_buffer()

    def send(self, msg):
        self.com.write(msg)

    def receive(self):
        ans = self.com.read(100)
        return ans
```

Listing 14: In the *Communication.py* file, the library for serial communication available in Python is adapted for own purposes.

8.1.5 Dictionary.py

```
# This file contains general settings and values that do not have to be
# changed every time the application is run, but it is still convenient to
# be able to change these values.

# own scripts:
import Calibration

pump_calibration = {
    "HPLC A": Calibration.Pumps(1000),
    "HPLC B": Calibration.Pumps(1000),
    "HPLC C": Calibration.Pumps(1000),
    "Lambda 1": Calibration.Pumps(20.117),
    "Lambda 2": Calibration.Pumps(80.265),
    "Lambda 3": Calibration.Pumps(20.294),
    }

calorimeter_thermostat = {
    "calibration": Calibration.Thermostat(25, 40, [[25,26], [30, 34], [35, 40], [40, 46]]),
    "40": Calibration.Calorimeter([[0.7618, 7.8524, 0.2782], [0.7157, 7.8297, 0.2332], [0.4186, 8.2974, 0.2775]]),
    "35": Calibration.Calorimeter([[0.4921, 8.3027, 0.1518], [0.6368, 8.1373, 0.1511], [0.3782, 8.5837, 0.1580]]),
    "30": Calibration.Calorimeter([[0.9217, 8.0548, 0.0291], [0.6194, 8.1511, 0.0944], [0.7441, 8.3637, 0.0420]]),
    "25": Calibration.Calorimeter([[0.6518, 8.3911, -0.0598], [0.4091, 8.5318, 0.0261], [0.4039, 8.7333, -0.0475]]),
    }

pump_head = {
    "HPLC A": 50,
    "HPLC B": 50,
    "HPLC C": 10,
    }

lambda_address = {
    "Lambda 1": 2,
    "Lambda 2": 2,
    "Lambda 3": 2,
    }

calculation_data = {
    "concentration": 0.997/18.015*1000,        # mol/l
    "cp": 75.336,                               # J/(molK)
    }
```

Listing 15: The *Dictionary.py* file contains some basic interchangeable parameters.

8.1.6 Excel_Functions.py

```
# This file contains a function that creates an Excel file for
# the OCAE (Output Calculation Absolute Evaluation) strategy.
# The created Excel file already contains all headings and
# substance data.

# library/modules from python:
from openpyxl import Workbook

# own scripts:
import Dictionary

def create_excel(substance_data, file_name):
    wb = Workbook()

    # create all sheets
    sheet = []
    sheet_names = ["Evaluation", "Raw_Data_COM"]

    sheet.append(wb.active)
    sheet[0].title = sheet_names[0]
    sheet.append(wb.create_sheet(sheet_names[1]))

    # setup the evaluation sheet
    counter = 1
    ret_counter = []

    title = [["Substance Data"], ["Process setup"], ["Raw Data Processing"], ["Calculation"]]
    data = [["Substance", "Molar Mass [g/mol]", "Weighing [g]", "Volume [ml]", "Concentration [mol/l]"],
            ["Process Points", "Evaluation Start Time", "Evaluation End Time", "V_A [ml/min]", "V_A,act [ml/min]", "n_A,act
        [mol/s]", "n_A,act,water [mol/s]", "V_B [ml/min]", "V_B,act [ml/min]", "n_B,act [mol/s]", "n_B,act,water [mol/s]"],
            ["Process Points", "T_A [°C]", "T_B [°C]", "T_out [°C]", "Upre [V]", "Ur1 [V]", "Ur2 [V]", "dT_A [°C]", "dT_B [
      °C]", "dT_out [°C]", "Q_Out [W]"],
            ["Process Points", "Q_A [W]", "Q_B [W]", "Qpre [W] - cp flux", "QSE,pre [W]", "Qr1 [W]", "Qr2 [W]", "dHr [kJ/
      mol]"]]

    for idx in range(4):
        sheet[0].append(title[idx])
        sheet[0].append(data[idx])
        counter += 2
        ret_counter.append([counter-2, counter, len(data[idx])])

    # insert substance data to the evaluation sheet
    sheet[0].insert_rows(idx=3, amount = 3)
    for idx in range(2):
```

```
        sheet[0].cell(row=idx+3, column=1).value = chr(idx+97)
        sheet[0].cell(row=idx+3, column=2).value = substance_data.get_molar_mass()[idx]
        sheet[0].cell(row=idx+3, column=3).value = substance_data.get_weighing()[idx]
        sheet[0].cell(row=idx+3, column=4).value = substance_data.get_volume()[idx]
        sheet[0].cell(row=idx+3, column=5).value = substance_data.get_concentration()[idx]

    sheet[0].cell(row=2, column=8).value = "Additional data"
    sheet[0].cell(row=3, column=8).value = "concentration [mol/l]"
    sheet[0].cell(row=3, column=9).value = Dictionary.calculation_data["concentration"]
    sheet[0].cell(row=4, column=8).value = "cp [J/(molK)]"
    sheet[0].cell(row=4, column=9).value = Dictionary.calculation_data["cp"]

    # remember at which position the titles are and where rows have to be inserted later
    sheet[0].insert_rows(idx=8, amount=1)
    sheet[0].insert_rows(idx=11, amount=1)

    val = 3
    for idx in range(1,4):
        ret_counter[idx][0] += val
        ret_counter[idx][1] += val
        val += 1
    ret_counter[0][1] = 5

    # setup the evaluation sheet
    sheet[1].append(["Elapsed_Time", "T_set", "T_pre", "T_r1", "T_r2", "T_A", "T_B", "T_out", "U_pre", "U_r1", "U_r2"])

    wb.save("{}.xlsx".format(file_name))

    return wb, sheet, ret_counter
```

Listing 16: The *Excel_Functions.py* contains the function, which is responsible for the setup of the basic output file.

8.1.7 Fisher.py

```
# This file contains the driver class for the Fisher thermostat and
# the following additional relevant classes for the construction and
# simple testing of the driver class: the dummy communication handle,
# the response checkers and all layer (A) states.

# library/modules from python:
import math
import re

# own scripts:
import LayerB
import pyState

# dummy communication handle:
class dummy_cmd_handle():
    """This class can be used for testing the driver. Thus, no actual Fisher thermostat is needed."""
    def __init__(self):
        self.resp = "OK\r\n"
        self.val = 0
        self.resp_RO = "0"
        self.resp_RPS = "M"
        self.resp_RE = "0"

    def send(self, msg):
        # print(msg)
        if msg.decode("ASCII") == "RO\r":
            self.resp = "{}\r".format(self.resp_RO)
        elif msg.decode("ASCII") == "SO 1\r":
            self.resp = "OK\r\n"
            self.resp_RO = "1"
        elif msg.decode("ASCII") == "SO 0\r":
            self.resp = "OK\r\n"
            self.resp_RO = "0"
        elif msg.decode("ASCII") == "STU C\r":
            self.resp = "OK\r\n"
        elif msg.decode("ASCII") == "RTU\r":
            self.resp = "C\r\n"
        elif msg.decode("ASCII") == "SPS M\r":
            self.resp = "OK\r\n"
            self.resp_RPS = "M"
        elif msg.decode("ASCII") == "SPS L\r":
            self.resp = "OK\r\n"
            self.resp_RPS = "L"
        elif msg.decode("ASCII") == "SPS H\r":
            self.resp = "OK\r\n"
            self.resp_RPS = "H"
        elif msg.decode("ASCII") == "RPS\r":
            self.resp = "{}\r".format(self.resp_RPS)
        elif msg.decode("ASCII") == "SE 0\r":
            self.resp = "OK\r\n"
            self.resp_RE = "0"
        elif msg.decode("ASCII") == "SE 1\r":
            self.resp = "OK\r\n"
```

```python
            self.resp_RE = "1"
        elif msg.decode("ASCII") == "RE\r":
            self.resp = "{}\r".format(self.resp_RE)
        elif msg.decode("ASCII") == "RS\r":
            self.resp = "{:.1f}C\r".format(self.val)
        elif msg.decode("ASCII") == "SS 26.0\r":
            self.resp = "OK\r\n"
            self.val = 26.0
        elif msg.decode("ASCII") == "SS 25.0\r":
            self.resp = "OK\r\n"
            self.val = 25.0

    def clear_input_buffer(self):
        # print("...clear...")
        return
    def receive(self):
        # print("...receive... {}".format(self.resp))
        return bytearray(self.resp.encode("ASCII"))

# response checkers:
class check_response_base:
    """This class forms the basis for all other response checkers."""
    def __init__(self, resp):
        self.resp = resp

    def __call__(self, ans):
        if ans.decode("ASCII") == self.resp:
            return True
        return False

class check_OK(check_response_base):
    def __init__(self):
        super().__init__("OK")

class check_0(check_response_base):
    def __init__(self):
        super().__init__("0")

class check_1(check_response_base):
    def __init__(self):
        super().__init__("1")

class check_set_Temp(check_response_base):
    def __init__(self, resp):
        self.resp = float(resp)

    def __call__(self, ans):
        answer = ans.decode("ASCII")
        find_pattern = re.compile(r"([\d\.]*)C")
        if not find_pattern.match(answer) is None:
            value = float(find_pattern.match(answer).group(1))
        else:
            return False

        if value == self.resp:
            return True
        return False

# layer (A) states:
class Deactivated(pyState.State_Base):
    """This state checks whether the pump of the thermostat is still switched off and whether the set temperature is correct. A different set temperature can be set and the pump can be activated from outside."""
    class factory:
        def __init__(self, set_temp, com_handle):
            self.set_temp = set_temp
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "RO", check_0(), self.com_handle, 3)
                return st
            elif state_name == "Set_Temp":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "SS {:.1f}".format(self.set_temp[0]), check_OK(), self.com_handle, 3)
                return st
            elif state_name == "Check_Temp":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "RS", check_set_Temp(self.set_temp[0]), self.com_handle, 3)
                return st
            elif state_name == "Waiting":
                st = LayerB.Delay_State()
                st.enter(state_name, 500, "next")
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, target_temp, set_temp, com_handle):
        super().enter(name)
        self.target_temp = target_temp
        self.set_temp = set_temp
```

```
        self.pump_on_flag = False

        self.tab = [
            ["Check_Pump_State",    "next",        "Waiting"],
            ["Check_Pump_State",    "error",       "Error"],
            ["Waiting",             "next",        "Check_Pump_State"],
            ["Waiting",             "new_temp",    "Set_Temp"],
            ["Set_Temp",            "next",        "Check_Temp"],
            ["Set_Temp",            "error",       "Error"],
            ["Check_Temp",          "next",        "Check_Pump_State"],
            ["Check_Temp",          "error",       "Error"],
            ]
        self.fac = Deactivated.factory(set_temp, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Check_Pump_State")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Error":
            return "error"

        if not self.en.get_state() == "Waiting":
            return None

        if math.isfinite(self.target_temp[0]) and not self.target_temp[0] == self.set_temp[0]:
            self.set_temp[0] = self.target_temp[0]
            self.en.handle_event("new_temp")
            return None
        elif self.pump_on_flag:
            return "pump_on"

    def exit(self):
        self.en.exit()
        super().exit()

    def handle_event(self, event):
        if event == "request_pump_on":
            self.pump_on_flag = True
            return True
        return False

class Activated(pyState.State_Base):
    """This state checks whether the pump of the thermostat is still switched on and whether the set temperature is correct. A different set temperature can be set and the pump can be deactivated from outside."""
    class factory:
        def __init__(self, set_temp, com_handle):
            self.set_temp = set_temp
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "RO", check_1(), self.com_handle, 3)
                return st
            elif state_name == "Set_Temp":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "SS {:.1f}".format(self.set_temp[0]), check_OK(), self.com_handle, 3)
                return st
            elif state_name == "Check_Temp":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "RS", check_set_Temp(self.set_temp[0]), self.com_handle, 3)
                return st
            elif state_name == "Waiting":
                st = LayerB.Delay_State()
                st.enter(state_name, 500, "next")
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, target_temp, set_temp, com_handle):
        super().enter(name)
        self.target_temp = target_temp
        self.set_temp = set_temp
        self.pump_off_flag = False

        self.tab = [
            ["Check_Pump_State",    "next",     "Waiting"],
            ["Check_Pump_State",    "error",    "Error"],
            ["Waiting",             "next",     "Check_Temp"],
            ["Waiting",             "new_temp", "Set_Temp"],
            ["Set_Temp",            "next",     "Check_Temp"],
            ["Set_Temp",            "error",    "Error"],
            ["Check_Temp",          "next",     "Check_Pump_State"],
            ["Check_Temp",          "error",    "Error"],
            ]
        self.fac = Activated.factory(set_temp, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Check_Pump_State")
        self.en.enter()

```

```
    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Error":
            return "error"

        if not self.en.get_state() == "Waiting":
            return None

        if math.isfinite(self.target_temp[0]) and not self.target_temp[0] == self.set_temp[0]:
            self.set_temp[0] = self.target_temp[0]
            self.en.handle_event("new_temp")
            return None
        elif self.pump_off_flag:
            return "pump_off"

    def exit(self):
        self.en.exit()
        super().exit()

    def handle_event(self, event):
        if event == "request_pump_off":
            self.pump_off_flag = True
            return True
        return False

class Activating(pyState.State_Base):
    """This state activates the pump of the thermostat and checks whether the switch-on has worked."""
    class factory:
        def __init__(self, com_handle):
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Pump_On":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "SO 1", check_OK(), self.com_handle, 3)
                return st
            elif state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "RO", check_1(), self.com_handle, 3)
                return st
            elif state_name == "Finished":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, com_handle):
        super().enter(name)
        self.tab = [
            ["Pump_On",            "next",     "Check_Pump_State"],
            ["Pump_On",            "error",    "Error"],
            ["Check_Pump_State",   "next",     "Finished"],
            ["Check_Pump_State",   "error",    "Error"],
            ]
        self.fac = Activating.factory(com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Pump_On")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Finished":
            return "next"
        if self.en.get_state() == "Error":
            return "error"

    def exit(self):
        self.en.exit()
        super().exit()

class Deactivating(pyState.State_Base):
    """This state deactivates the pump of the thermostat and checks whether the shutdown has worked."""
    class factory:
        def __init__(self, com_handle):
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Pump_Off":
                st = LayerB.Send_And_Check()
                st.enter( state_name, "SO 0", check_OK(), self.com_handle, 3)
                return st
            elif state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter( state_name, "RO", check_0(), self.com_handle, 3)
                return st
            elif state_name == "Finished":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
```

```python
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, com_handle):
        super().enter(name)
        self.tab = [
            ["Pump_Off",          "next",     "Check_Pump_State"],
            ["Pump_Off",          "error",    "Error"],
            ["Check_Pump_State",  "next",     "Finished"],
            ["Check_Pump_State",  "error",    "Error"],
            ]
        self.fac = Deactivating.factory(com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Pump_Off")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Finished":
            return "next"
        if self.en.get_state() == "Error":
            return "error"

    def exit(self):
        self.en.exit()
        super().exit()

class Configuration(pyState.State_Base):
    """This state deactivates the pump of the thermostat and adjusts all initial settings."""
    class factory:
        def __init__(self, settings, com_handle):
            self.settings = settings
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Pump_Off":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "SO 0", check_OK(), self.com_handle, 3)
                return st
            elif state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "RO", check_0(), self.com_handle, 3)
                return st
            elif state_name == "Set_Temp_Unit":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "STU C", check_OK(), self.com_handle, 3)
                return st
            elif state_name == "Check_Temp_Unit":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "RTU", check_response_base("C"), self.com_handle, 3)
                return st
            elif state_name == "Set_Pump_Speed":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "SPS {}".format(self.settings.get_pump_speed()), check_OK(), self.com_handle, 3)
                return st
            elif state_name == "Check_Pump_Speed":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "RPS", check_response_base(self.settings.get_pump_speed()), self.com_handle, 3)
                return st
            elif state_name == "Set_External_Probe":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "SE {}".format(self.settings.get_external_probe()), check_OK(), self.com_handle, 3)
                return st
            elif state_name == "Check_External_Probe":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "RE", check_response_base("{}".format(self.settings.get_external_probe())), self.com_handle, 3)
                return st
            elif state_name == "Finished":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, settings, com_handle):
        super().enter(name)
        self.tab = [
            ["Pump_Off",           "next",     "Check_Pump_State"],
            ["Pump_Off",           "error",    "Error"],
            ["Check_Pump_State",   "next",     "Set_Temp_Unit"],
            ["Check_Pump_State",   "error",    "Error"],
            ["Set_Temp_Unit",      "next",     "Check_Temp_Unit"],
            ["Set_Temp_Unit",      "error",    "Error"],
            ["Check_Temp_Unit",    "next",     "Set_Pump_Speed"],
            ["Check_Temp_Unit",    "error",    "Error"],
            ["Set_Pump_Speed",     "next",     "Check_Pump_Speed"],
```

```
            ["Set_Pump_Speed",       "error",     "Error"],
            ["Check_Pump_Speed",     "next",      "Set_External_Probe"],
            ["Check_Pump_Speed",     "error",     "Error"],
            ["Set_External_Probe",   "next",      "Check_External_Probe"],
            ["Set_External_Probe",   "error",     "Error"],
            ["Check_External_Probe","next",       "Finished"],
            ["Check_External_Probe","error",      "Error"],
            ]
        self.fac = Configuration.factory(settings, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Pump_Off")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Finished":
            return "next"
        if self.en.get_state() == "Error":
            return "error"

    def exit(self):
        self.en.exit()
        super().exit()

# driver class for the Fisher thermostat:
class Driver:

    class Settings:
        def __init__(self):
            self._pump_speed = "L"
            self._ext_probe = 0

        def get_pump_speed(self):
            return self._pump_speed
        def get_external_probe(self):
            return self._ext_probe
        def set_pump_speed(self, val):
            if val == 'L' or val == 'M' or val == 'H':
                self._pump_speed = val
                return
            raise Exception("Invalid value (L, M, H)")
        def set_external_probe(self, val):
            if val == 0 or val == 1:
                self._ext_probe = val
                return
            raise Exception("Invalid value (0, 1)")

    class factory:
        def __init__(self, settings, target_temp, set_temp, com_handle):
            self.settings = settings
            self.target_temp = target_temp
            self.set_temp = set_temp
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Configuration":
                st = Configuration()
                st.enter(state_name, self.settings, self.com_handle)
                return st
            elif state_name == "Deactivated":
                st = Deactivated()
                st.enter(state_name, self.target_temp, self.set_temp, self.com_handle)
                return st
            elif state_name == "Activated":
                st = Activated()
                st.enter(state_name, self.target_temp, self.set_temp, self.com_handle)
                return st
            elif state_name == "Deactivating":
                st = Deactivating()
                st.enter(state_name, self.com_handle)
                return st
            elif state_name == "Activating":
                st = Activating()
                st.enter(state_name, self.com_handle)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def __init__(self, name, settings, com_handle):
        self.name = name
        self.tab = [
            ["Configuration",          "next",        "Deactivated"],
            ["Configuration",          "error",       "Error"],
            ["Deactivated",            "pump_on",     "Activating"],
            ["Deactivated",            "error",       "Error"],
            ["Activating",             "next",        "Activated"],
            ["Activating",             "error",       "Error"],
            ["Activated",              "pump_off",    "Deactivating"],
            ["Activated",              "error",       "Error"],
            ["Deactivating",           "next",        "Deactivated"],
            ["Deactivating",           "error",       "Error"],
```

```
            ]
        self.target_temp = [float("nan")]
        self.set_temp = [float("nan")]

        self.target_pump_state = False

        self.fac = Driver.factory(settings, self.target_temp, self.set_temp, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Configuration")
        self.en.enter()

    def tick(self):
        self.en.tick()

        if self.en.get_state() == "Configuration":
            return

        if self.en.get_state() == "Deactivated" and self.target_pump_state:
            self.en.handle_event("request_pump_on")
            return

        if self.en.get_state() == "Activated" and not self.target_pump_state:
            self.en.handle_event("request_pump_off")
            return

    def __del__(self):
        self.en.exit()

    def get_state(self):
        return self.en.get_state()

    def get_name(self):
        return self.name

    # In the following, the functions are defined to obtain the settings for the Fisher thermostat (from outside).
    def set_target_temp(self, val):
        self.target_temp[0] = val

    def activate_pump(self):
        self.target_pump_state = True

    def deactivate_pump(self):
        self.target_pump_state = False
```

Listing 17: The *Fisher.py* file contains its corresponding device driver and can be used to operate this device.

8.1.8 HPLC.py

```
# This file contains the driver class for the HPLC pump and the
# following additional relevant classes and functions for the
# construction and simple testing of the driver class: the dummy
# communication handle, two special substates, the response
# checkers and all layer (A) states.

# library/modules from python:
import re
import statistics
import time

# own scripts:
import LayerB
import LayerC
import pyState

# dummy communication handle:
class dummy_cmd_handle():
    """This class can be used for testing the driver. Thus, no actual HPLC pump is needed."""
    def __init__(self):
        self.press = 0
        self.flow = 0
        self.pump = False
        self.resp = "0"

    def send(self, msg):
        # print(msg)
        if msg.decode("ASCII") == "PRESSURE?\r":
            if self.pump == False:
                self.resp = "PRESSURE:0\r"
            elif self.flow <= 0:
                self.resp = "PRESSURE:0\r"
            else:
                self.resp = "PRESSURE:30\r"
        elif msg.decode("ASCII") == "PMIN50: 0\r":
            self.resp = "PMIN50:OK\r"
        elif msg.decode("ASCII") == "PMIN50?\r":
            self.resp = "PMIN50:0\r"
        elif msg.decode("ASCII") == "PMAX50: 100\r":
            self.resp = "PMAX50:OK\r"
        elif msg.decode("ASCII") == "PMAX50?\r":
```

```
            self.resp = "PMAX50:100\r"
        elif msg.decode("ASCII") == "FLOW: 06000\r":
            self.resp = "FLOW:OK\r"
            self.flow = 6000
        elif msg.decode("ASCII") == "FLOW: 00000\r":
            self.resp = "FLOW:OK\r"
            self.flow = 0
        elif msg.decode("ASCII") == "FLOW?\r":
            self.resp = "FLOW:{:05}\r".format(self.flow)
        elif msg.decode("ASCII") == "ON\r":
            self.pump = True
            self.resp = "ON:OK\r"
        elif msg.decode("ASCII") == "OFF\r":
            self.pump = False
            self.resp = "OFF:OK\r"

    def clear_input_buffer(self):
        # print("...clear...")
        return

    def receive(self):
        # print("...receive... {}".format(self.resp))
        return bytearray(self.resp.encode("ASCII"))

# two special substates:
# These states are required by the HPLC pump to initially query the
# system pressure, generate a reference value from it and later check
# for this reference value.
class Save_Answer(pyState.State_Base):
    """This layer (C) state receives, checks and saves the response received."""
    def enter(self, name, timeout_ms, com_handle, datalist, boundaries, next_event, timeout_event, done_event):
        super().enter(name)
        self.com_handle = com_handle
        self.deadline = time.monotonic_ns() + timeout_ms * 1000000

        self.next_event = next_event
        self.timeout_event = timeout_event
        self.done_event = done_event

        self.datalist = datalist
        self.boundaries = boundaries
        self.response = bytearray()

    def __call__(self):
        end_of_frame = '\r'.encode("ASCII")
        tmp = self.com_handle.receive()
        for chr in tmp:
            if chr == end_of_frame[0]:
                str_ans = find_pattern(self.response)
                self.datalist.append(float(str_ans))
                if len(self.datalist) < 10:
                    return self.next_event
                # calculation of the mean and standard deviation
                self.boundaries[0] = statistics.mean(self.datalist)
                self.boundaries[1] = statistics.stdev(self.datalist)
                return self.done_event
            self.response.append(chr)

        if time.monotonic_ns() > self.deadline:
            return self.timeout_event
        return None

class Send_And_Save_Data(pyState.State_Base):
    """This layer (B) state combines the substates sending a command and waiting and saving the response."""
    class factory:
        def __init__(self, datalist, boundaries, com_handle):
            self.datalist = datalist
            self.boundaries = boundaries
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Send":
                st = LayerC.Send_Command()
                st.enter(state_name, "PRESSURE?", self.com_handle, "next")
                return st
            elif state_name == "Save":
                st = Save_Answer()
                st.enter(state_name, 1000, self.com_handle, self.datalist, self.boundaries, "next", "timeout", "done")
                return st
            elif state_name == "Waiting":
                st = LayerB.Delay_State()
                st.enter(state_name, 500, "next")
                return st
            elif state_name == "Finished":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")
```

```python

    def enter(self, name, boundaries, com_handle):
        super().enter(name)
        self.tab = [
            ["Send",            "next",         "Save"],
            ["Save",            "next",         "Waiting"],
            ["Save",            "timeout",      "Error"],
            ["Save",            "done",         "Finished"],
            ["Waiting",         "next",         "Send"],
            ]
        self.datalist = []

        self.fac = Send_And_Save_Data.factory(self.datalist, boundaries, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Send")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Finished":
            return "next"
        if self.en.get_state() == "Error":
            return "error"

    def exit(self):
        self.en.exit()
        super().exit()

# response checkers:
def find_pattern(ans):
    answer = ans.decode("ASCII")
    find_pattern = re.compile(r"\w*:(.*)")
    if not find_pattern.match(answer) is None:
        str_ans = find_pattern.match(answer).group(1)
        return str_ans
    else:
        return False

class check_response_base:
    """This class forms the basis for all other response checkers."""
    def __init__(self, resp):
        self.resp = resp

    def __call__(self, ans):
        # step 1: find pattern
        str_ans = find_pattern(ans)

        # step 2: compare pattern
        if str_ans == self.resp:
            return True
        return False

class check_ok(check_response_base):
    def __init__(self):
        super().__init__("OK")

class check_flow(check_response_base):
    def __init__(self, val):
        self.val = val

    def __call__(self, ans):
        str_ans = find_pattern(ans)
        if int(str_ans) == self.val:
            return True
        return False

class check_boundaries(check_response_base):
    def __init__(self, boundaries):
        self.lower = 0          # boundaries[0] - 3 * boundaries[1]
        self.upper = 450        # boundaries[0] + 3 * boundaries[1]

    def __call__(self, ans):
        str_ans = find_pattern(ans)
        val = float(str_ans)
        if val <= self.upper and val >= self.lower:
            return True
        return False

class check_0(check_response_base):
    def __init__(self):
        super().__init__(0)

    def __call__(self, ans):
        str_ans = find_pattern(ans)
        val = float(str_ans)
        if val <= 40:
            return True
        return False

# layer (A) states:
class Configuration(pyState.State_Base):
    """This state deactivates the HPLC pump and adjusts all initial settings."""
    class factory:
        def __init__(self, settings, com_handle):
            self.head = settings.get_head()
```

```python
            self.settings = settings
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Pump_Off":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "OFF", check_ok(), self.com_handle, 0)
                return st
            elif state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "PRESSURE?", check_0(), self.com_handle, 0)
                return st
            elif state_name == "Set_PMin":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "PMIN{:.2}: {:.0f}".format(str(self.head), self.settings.get_PMin()), check_ok(), self.com_handle, 0)
                return st
            elif state_name == "Check_PMin":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "PMIN{:.2}?".format(str(self.head)), check_response_base("{:.0f}".format(self.settings.get_PMin())) , self.com_handle, 0)
                return st
            elif state_name == "Set_PMax":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "PMAX{:.2}: {:.0f}".format(str(self.head), self.settings.get_PMax()), check_ok(), self.com_handle, 0)
                return st
            elif state_name == "Check_PMax":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "PMAX{:.2}?".format(str(self.head)), check_response_base("{:.0f}".format(self.settings.get_PMax())), self.com_handle, 0)
                return st
            elif state_name == "Finished":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, settings, com_handle):
        super().enter(name)
        self.tab = [
            ["Pump_Off",            "next",     "Check_Pump_State"],
            ["Pump_Off",            "error",    "Error"],
            ["Check_Pump_State",    "next",     "Set_PMin"],
            ["Check_Pump_State",    "error",    "Error"],
            ["Set_PMin",            "next",     "Check_PMin"],
            ["Set_PMin",            "error",    "Error"],
            ["Check_PMin",          "next",     "Set_PMax"],
            ["Check_PMin",          "error",    "Error"],
            ["Set_PMax",            "next",     "Check_PMax"],
            ["Set_PMax",            "error",    "Error"],
            ["Check_PMax",          "next",     "Finished"],
            ["Check_PMax",          "error",    "Error"],
            ]
        self.fac = Configuration.factory(settings, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Pump_Off")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Finished":
            return "next"
        if self.en.get_state() == "Error":
            return "error"

    def exit(self):
        self.en.exit()
        super().exit()

class Deactivated(pyState.State_Base):
    """This state checks whether the HPLC pump is still switched off and whether anything has changed in the settings and adjusts them if necessary."""
    class factory:
        def __init__(self, set_flowrate, com_handle):
            self.set_flow = set_flowrate
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "PRESSURE?", check_0(), self.com_handle, 0)
                return st
            elif state_name == "Set_Flowrate":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "FLOW: {:05.0f}".format(self.set_flow[0]), check_ok(), self.com_handle, 0)
                return st
            elif state_name == "Check_Flowrate":
                st = LayerB.Send_And_Check()
```

```
                st.enter(state_name, "FLOW?", check_flow(self.set_flow[0]), self.com_handle, 0)
                return st
            elif state_name == "Waiting":
                st = LayerB.Delay_State()
                st.enter(state_name, 500, "next")
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, target_flowrate, set_flowrate, com_handle):
        super().enter(name)
        self.target_flowrate = target_flowrate
        self.set_flowrate = set_flowrate
        self.pump_on_flag = False
        self.tab = [
            ["Check_Pump_State", "next",        "Waiting"],
            ["Check_Pump_State", "error",       "Error"],
            ["Waiting",          "next",        "Check_Pump_State"],
            ["Waiting",          "new_flowrate", "Set_Flowrate"],
            ["Set_Flowrate",     "next",        "Check_Flowrate"],
            ["Set_Flowrate",     "error",       "Error"],
            ["Check_Flowrate",   "next",        "Check_Pump_State"],
            ["Check_Flowrate",   "error",       "Error"],
            ]
        self.fac = Deactivated.factory(set_flowrate, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Check_Pump_State")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Error":
            return "error"

        if not self.en.get_state() == "Waiting":
            return None

        if not self.target_flowrate[0] == self.set_flowrate[0]:
            self.set_flowrate[0] = self.target_flowrate[0]
            self.en.handle_event("new_flowrate")
            return None
        elif self.pump_on_flag:
            return "pump_on"

    def exit(self):
        self.en.exit()
        super().exit()

    def handle_event(self, event):
        if event == "request_pump_on":
            self.pump_on_flag = True
            return True
        return False

class Activated(pyState.State_Base):
    """This state checks whether the HPLC pump is still running at the correct flow rate and whether anything has changed in the settings and adjusts them if necessary."""
    class factory:
        def __init__(self, set_flowrate, com_handle):
            self.set_flow = set_flowrate
            self.com_handle = com_handle
            self.boundaries = [0,0]

        def create_state(self, state_name):
            if state_name == "Get_Boundaries":
                st = Send_And_Save_Data()
                st.enter(state_name, self.boundaries, self.com_handle)
                return st
            elif state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "PRESSURE?", check_boundaries(self.boundaries), self.com_handle, 0)
                return st
            elif state_name == "Set_Flowrate":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "FLOW: {:05.0f}".format(self.set_flow[0]), check_ok(), self.com_handle, 0)
                return st
            elif state_name == "Check_Flowrate":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "FLOW?", check_flow(self.set_flow[0]), self.com_handle, 0)
                return st
            elif state_name == "Check_New_Flowrate":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "FLOW?", check_flow(self.set_flow[0]), self.com_handle, 0)
                return st
            elif state_name == "Waiting":
                st = LayerB.Delay_State()
                st.enter(state_name, 500, "next")
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
```

```
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, target_flowrate, set_flowrate, com_handle):
        super().enter(name)
        self.target_flowrate = target_flowrate
        self.set_flowrate = set_flowrate
        self.pump_off_flag = False

        self.tab = [
            ["Get_Boundaries",      "next",         "Check_Pump_State"],
            ["Get_Boundaries",      "error",        "Error"],
            ["Check_Pump_State",    "next",         "Waiting"],
            ["Check_Pump_State",    "error",        "Error"],
            ["Waiting",             "next",         "Check_Flowrate"],
            ["Waiting",             "new_flowrate", "Set_Flowrate"],
            ["Set_Flowrate",        "next",         "Check_New_Flowrate"],
            ["Set_Flowrate",        "error",        "Error"],
            ["Check_New_Flowrate",  "next",         "Get_Boundaries"],
            ["Check_New_Flowrate",  "error",        "Error"],
            ["Check_Flowrate",      "next",         "Check_Pump_State"],
            ["Check_Flowrate",      "error",        "Error"],
            ]
        self.fac = Activated.factory(set_flowrate, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Get_Boundaries")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Error":
            return "error"

        if not self.en.get_state() == "Waiting":
            return None

        if not self.target_flowrate[0] == self.set_flowrate[0]:
            self.set_flowrate[0] = self.target_flowrate[0]
            self.en.handle_event("new_flowrate")
            return None
        elif self.pump_off_flag:
            return "pump_off"

    def exit(self):
        self.en.exit()
        super().exit()

    def handle_event(self, event):
        if event == "request_pump_off":
            self.pump_off_flag = True
            return True
        return False

class Deactivating(pyState.State_Base):
    """This state deactivates the HPLC pump and checks whether the shutdown has worked."""
    class factory:
        def __init__(self, com_handle):
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Pump_Off":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "OFF", check_ok(), self.com_handle, 0)
                return st
            elif state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "PRESSURE?", check_0(), self.com_handle, 0)
                return st
            elif state_name == "Finished":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, com_handle):
        super().enter(name)
        self.tab = [
            ["Pump_Off",            "next",     "Check_Pump_State"],
            ["Pump_Off",            "error",    "error"],
            ["Check_Pump_State",    "next",     "Finished"],
            ["Check_Pump_State",    "error",    "Error"],
            ]
        self.fac = Deactivating.factory(com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Pump_Off")
        self.en.enter()

    def __call__(self):
        self.en.tick()
```

```python

        if self.en.get_state() == "Finished":
            return "next"
        if self.en.get_state() == "Error":
            return "error"

    def exit(self):
        self.en.exit()
        super().exit()

# driver class for the HPLC pump:
class Driver:

    class Settings:
        def __init__(self, head):
            self._PMin_ = 0
            self._PMax_ = 100
            self.head = head
            if self.head != 10 and self.head != 50:
                raise Exception("Invalid pump head")

        def get_PMin(self):
            return self._PMin_
        def get_PMax(self):
            return self._PMax_
        def get_head(self):
            return self.head

        def set_PMinMax(self, pmin, pmax):
            if pmin >= pmax:
                raise Exception("Minimum can't be above maximum")
            if pmin < 0:
                raise Exception("Minimum is out of boundaries")
            if self.head == 10 and pmax > 400:
                raise Exception("Maximum is out of boundaries")
            if self.head == 50 and pmax > 150:
                raise Exception("Maximum is out of boundaries")

            self._PMin_ = pmin
            self._PMax_ = pmax

    class factory:
        def __init__(self, settings, target_flowrate, set_flowrate, com_handle):
            self.settings = settings
            self.target_flowrate = target_flowrate
            self.set_flowrate = set_flowrate
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Configuration":
                st = Configuration()
                st.enter(state_name, self.settings, self.com_handle)
                return st
            elif state_name == "Deactivated":
                st = Deactivated()
                st.enter(state_name, self.target_flowrate, self.set_flowrate, self.com_handle)
                return st
            elif state_name == "Activated":
                st = Activated()
                st.enter(state_name, self.target_flowrate, self.set_flowrate, self.com_handle)
                return st
            elif state_name == "Deactivating":
                st = Deactivating()
                st.enter(state_name, self.com_handle)
                return st
            if state_name == "Activating":
                st = LayerB.Send_And_Check()
                st.enter(state_name, "ON", check_ok(), self.com_handle, 0)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def __init__(self, name, settings, calibration_func, com_handle):
        self.name = name
        self.tab = [
            ["Configuration",        "next",        "Deactivated"],
            ["Configuration",        "error",       "Error"],
            ["Deactivated",          "pump_on",     "Activating"],
            ["Deactivated",          "error",       "Error"],
            ["Activating",           "next",        "Activated"],
            ["Activating",           "error",       "Error"],
            ["Activated",            "pump_off",    "Deactivating"],
            ["Activated",            "error",       "Error"],
            ["Deactivating",         "next",        "Deactivated"],
            ["Deactivating",         "error",       "Error"],
            ]
        self.calibration_func = calibration_func
        self.target_flowrate = [0]
        self.set_flowrate = [float("nan")]

        self.target_pump_state = False
```

```python

        self.fac = Driver.factory(settings, self.target_flowrate, self.set_flowrate, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Configuration")
        self.en.enter()

    def tick(self):
        self.en.tick()

        if self.en.get_state() == "Configuration":
            return

        if self.en.get_state() == "Deactivated" and self.target_pump_state:
            self.en.handle_event("request_pump_on")
            return

        if self.en.get_state() == "Activated" and not self.target_pump_state:
            self.en.handle_event("request_pump_off")
            return

    def __del__(self):
        self.en.exit()

    def get_state(self):
        return self.en.get_state()

    def get_name(self):
        return self.name

    # In the following, the functions are defined to obtain the settings for the HPLC pump (from outside).
    def set_target_flowrate(self, val):
        self.target_flowrate[0] = round(self.calibration_func.forward(val))
        return self.calibration_func.backward(self.target_flowrate[0])

    def activate_pump(self):
        self.target_pump_state = True

    def deactivate_pump(self):
        self.target_pump_state = False
```

Listing 18: The *HPLC.py* file contains its corresponding device driver and can be used to operate this device.

8.1.9 Lambda.py

```python
# This file contains the driver class for the Lambda pump and the
# following additional relevant classes for the construction and
# simple testing of the driver class: the dummy communication handle,
# generation of the commands that will later be sent to the pump,
# the response checker and all layer (A) states.

# library/modules from python:
import re

# own scripts:
import LayerB
import pyState

# dummy communication handle:
class dummy_cmd_handle:
    """This class can be used for testing the driver. Thus, no actual Lambda pump is needed."""
    def __init__(self):
        self.resp = "<0201r1232D\r\n"

    def send(self, msg):
        # print(msg)
        if msg.decode("ASCII") == "#0201r000E8\r":
            self.resp = "<0102r0002D\r"
        elif msg.decode("ASCII") == "#0201r123EE\r":
            self.resp = "<0102r1232D\r"
        elif msg.decode("ASCII") == "#0201r321EE\r":
            self.resp = "<0102r3212D\r"

    def clear_input_buffer(self):
        # print("...clear...")
        return None

    def receive(self):
        # print("...receive... {}".format(self.resp))
        return bytearray(self.resp.encode("ASCII"))

# generation of the commands that will later be sent to the pump:
class build_set_msg:
    """This class builds the command with the checksum for setting a flow rate."""
    def __init__(self, address, ddd):
        self.address = address
        self.ddd = ddd

    def __call__(self):
        mm = 1
```

```
        step1 = "#{:02d}{:02d}r{:03.0f}".format(self.address, mm, self.ddd)
        qs = sum(bytearray(step1.encode("ASCII"))) & 0xFF
        msg = "{}{:02X}".format(step1, qs)

        return msg

class build_read_msg:
    """This class builds the command with the checksum for the query which flow rate is set."""
    def __init__(self, address):
        self.address = address

    def __call__(self):
        mm = 1
        step1 = "#{:02d}{:02d}G".format(self.address, mm)
        qs = sum(bytearray(step1.encode("ASCII"))) & 0xFF
        msg = "{}{:02X}".format(step1, qs)

        return msg

# response checker:
class check_response:
    """This class compares the received answer "ans" with the expected answer "resp"."""
    def __init__(self, resp):
        self.resp = resp

    def __call__(self, ans):
        # step 1: find pattern
        answer = ans.decode("ASCII")
        find_pattern = re.compile(r"<\d{4}(\w{1})(\d{3})\S*")
        if not find_pattern.match(answer) is None:
            lrinfo = find_pattern.match(answer).group(1)
            ddd = int(find_pattern.match(answer).group(2))
        else:
            return False

        # step 2: compare pattern
        if ddd == self.resp and lrinfo == "r":
            return True
        else:
            return False

# layer (A) states:
class Deactivating(pyState.State_Base):
    """This state deactivates the Lambda pump and checks whether the shutdown has worked."""
    class factory:
        def __init__(self, address, com_handle):
            self.address = address
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Pump_Off":
                st = LayerB.Send()
                st.enter(state_name, build_set_msg(self.address, 0)(), self.com_handle, "next")
                return st
            elif state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter(state_name, build_read_msg(self.address)(), check_response(0), self.com_handle, 0)
                return st
            elif state_name == "Finished":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, address, com_handle):
        super().enter(name)
        self.tab = [
            ["Pump_Off",          "next",    "Check_Pump_State"],
            ["Check_Pump_State",  "next",    "Finished"],
            ["Check_Pump_State",  "error",   "Error"],
            ]
        self.fac = Deactivating.factory(address, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Pump_Off")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Finished":
            return "next"
        if self.en.get_state() == "Error":
            return "error"

    def exit(self):
        self.en.exit()
        super().exit()

class Deactivated(pyState.State_Base):
    """This state checks whether the Lambda pump is still switched off and waits whether the pump should be switched on
      again."""
    class factory:
```

```
        def __init__(self, address, com_handle):
            self.address = address
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter(state_name, build_read_msg(self.address)(), check_response(0), self.com_handle, 0)
                return st
            elif state_name == "Waiting":
                st = LayerB.Delay_State()
                st.enter(state_name, 500, "next")
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, address, com_handle):
        super().enter(name)
        self.pump_on_flag = False
        self.tab = [
            ["Check_Pump_State",  "next",   "Waiting"],
            ["Check_Pump_State",  "error",  "Error"],
            ["Waiting",           "next",   "Check_Pump_State"],
            ]
        self.fac = Deactivated.factory(address, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Check_Pump_State")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Error":
            return "error"

        if not self.en.get_state() == "Waiting":
            return None

        if self.pump_on_flag:
            return "pump_on"

    def exit(self):
        self.en.exit()
        super().exit()

    def handle_event(self, event):
        if event == "request_pump_on":
            self.pump_on_flag = True
            return True
        return False

class Activating(pyState.State_Base):
    """This state activates the Lambda pump and checks whether the switch-on has worked."""
    class factory:
        def __init__(self, set_flowrate, address, com_handle):
            self.flow = set_flowrate
            self.address = address
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Pump_On":
                st = LayerB.Send()
                st.enter(state_name, build_set_msg(self.address, self.flow[0])(), self.com_handle, "next")
                return st
            elif state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter(state_name, build_read_msg(self.address)(), check_response(self.flow[0]), self.com_handle, 0)
                return st
            elif state_name == "Finished":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, target_flowrate, set_flowrate, address, com_handle):
        super().enter(name)
        self.tab = [
            ["Pump_On",           "next",    "Check_Pump_State"],
            ["Check_Pump_State",  "next",    "Finished"],
            ["Check_Pump_State",  "error",   "Error"],
            ]
        set_flowrate[0] = target_flowrate[0]
        self.fac = Activating.factory(set_flowrate, address, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Pump_On")
        self.en.enter()

    def __call__(self):
        self.en.tick()
```

```python

        if self.en.get_state() == "Finished":
            return "next"
        if self.en.get_state() == "Error":
            return "error"

    def exit(self):
        self.en.exit()
        super().exit()

class Activated(pyState.State_Base):
    """This state checks whether the Lambda pump is still running at the correct flow rate and whether anything has changed in the settings and adjusts them if necessary."""
    class factory:
        def __init__(self, set_flowrate, address, com_handle):
            self.flow = set_flowrate
            self.address = address
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Check_Pump_State":
                st = LayerB.Send_And_Check()
                st.enter(state_name, build_read_msg(self.address)(), check_response(self.flow[0]), self.com_handle, 0)
                return st
            elif state_name == "Waiting":
                st = LayerB.Delay_State()
                st.enter(state_name, 500, "next")
                return st
            elif state_name == "Set_Flowrate":
                st = LayerB.Send()
                st.enter(state_name, build_set_msg(self.address, self.flow[0])(), self.com_handle, "next")
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, target_flowrate, set_flowrate, address, com_handle):
        super().enter(name)
        self.target_flowrate = target_flowrate
        self.set_flowrate = set_flowrate
        self.pump_off_flag = False

        self.tab = [
            ["Check_Pump_State",  "next",         "Waiting"],
            ["Check_Pump_State",  "error",        "Error"],
            ["Waiting",           "next",         "Check_Pump_State"],
            ["Waiting",           "new_flowrate", "Set_Flowrate"],
            ["Set_Flowrate",      "next",         "Check_Pump_State"],
            ]
        self.fac = Activated.factory(set_flowrate, address, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Check_Pump_State")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Error":
            return "error"

        if not self.en.get_state() == "Waiting":
            return None

        if not self.target_flowrate[0] == self.set_flowrate[0]:
            self.set_flowrate[0] = self.target_flowrate[0]
            self.en.handle_event("new_flowrate")
            return None
        elif self.pump_off_flag:
            return "pump_off"

    def exit(self):
        self.en.exit()
        super().exit()

    def handle_event(self, event):
        if event == "request_pump_off":
            self.pump_off_flag = True
            return True
        return False

# driver class for the Lambda pump:
class Driver:

    class factory:
        def __init__(self, address, target_flowrate, set_flowrate, com_handle):
            self.address = address
            self.target_flowrate = target_flowrate
            self.set_flowrate = set_flowrate
            self.com_handle = com_handle

        def create_state(self, state_name):
            if state_name == "Deactivating":
                st = Deactivating()
```

```
                st.enter(state_name, self.address, self.com_handle)
                return st
            elif state_name == "Deactivated":
                st = Deactivated()
                st.enter(state_name, self.address, self.com_handle)
                return st
            elif state_name == "Activated":
                st = Activated()
                st.enter(state_name, self.target_flowrate, self.set_flowrate, self.address, self.com_handle)
                return st
            elif state_name == "Activating":
                st = Activating()
                st.enter(state_name, self.target_flowrate, self.set_flowrate, self.address, self.com_handle)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def __init__(self, name, address, calibration_func, com_handle):
        self.name = name
        self.tab = [
            ["Deactivating",            "next",     "Deactivated"],
            ["Deactivating",            "error",    "Error"],
            ["Deactivated",             "pump_on",  "Activating"],
            ["Deactivated",             "error",    "Error"],
            ["Activating",              "next",     "Activated"],
            ["Activating",              "error",    "Error"],
            ["Activated",               "pump_off", "Deactivating"],
            ["Activated",               "error",    "Error"],
            ]
        self.calibration_func = calibration_func
        self.target_flowrate = [0]
        self.set_flowrate = [-1]

        self.target_pump_state = False

        self.fac = Driver.factory(address, self.target_flowrate, self.set_flowrate, com_handle)
        self.en = pyState.Engine(self.tab, self.fac, "Deactivating")
        self.en.enter()

    def tick(self):
        self.en.tick()

        if self.en.get_state() == "Deactivated" and self.target_pump_state and self.target_flowrate[0] != 0:
            self.en.handle_event("request_pump_on")
            return

        if self.en.get_state() == "Activated" and (not self.target_pump_state or self.target_flowrate[0] == 0):
            self.en.handle_event("request_pump_off")
            return

    def __del__(self):
        self.en.exit()

    def get_state(self):
        return self.en.get_state()

    def get_name(self):
        return self.name

    # In the following, the functions are defined to obtain the settings for the Lambda pump (from outside).
    def set_target_flowrate(self, val):
        self.target_flowrate[0] = round(self.calibration_func.forward(val))
        return self.calibration_func.backward(self.target_flowrate[0])

    def activate_pump(self):
        self.target_pump_state = True

    def deactivate_pump(self):
        self.target_pump_state = False
```

Listing 19: The *Lambda.py* file contains its corresponding device driver and can be used to operate this device.

8.1.10 LayerB.py

```
# This file contains relevant classes for the generation of the HPLC,
# Lambda and Thermostat drivers. The classes listed in this file are
# possible states on the middle (second) layer (B). Higher layers (A)
# can be built from these classes.

# library/modules from python:
import time

# own scripts:
import LayerC
import pyState
```

```

class Send_And_Check(pyState.State_Base):
    """This state combines the substates sending a command, waiting for the response and checking the response."""
    class factory:
        def __init__(self, msg, checker, com_handle, retry_count):
            self.msg = msg
            self.checker = checker
            self.com_handle = com_handle
            self.retry_count = [retry_count]

        def create_state(self, state_name):
            if state_name == "Send":
                st = LayerC.Send_Command()
                st.enter(state_name, self.msg, self.com_handle, "next")
                return st
            elif state_name == "Check":
                st = LayerC.Wait_For_Answer()
                st.enter(state_name, 1000, self.com_handle, self.checker, "next", "timeout", self.retry_count, "retry", "
    error")
                return st
            elif state_name == "Finished":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            elif state_name == "Error":
                st = pyState.State_Base()
                st.enter(state_name)
                return st
            raise Exception("Unhandled State in Factory")

    def enter(self, name, msg, checker, com_handle, retry_count):
        super().enter(name)
        self.tab = [
            ["Send",  "next",    "Check"],
            ["Check", "next",    "Finished"],
            ["Check", "retry",   "Send"],
            ["Check", "timeout", "Error"],
            ["Check", "error",   "Error"],
            ]
        self.fac = Send_And_Check.factory(msg, checker, com_handle, retry_count)
        self.en = pyState.Engine(self.tab, self.fac, "Send")
        self.en.enter()

    def __call__(self):
        self.en.tick()

        if self.en.get_state() == "Finished":
            return "next"
        if self.en.get_state() == "Error":
            return "error"

    def exit(self):
        self.en.exit()
        super().exit()

class Delay_State(pyState.State_Base):
    """This state waits for the given time."""
    def enter(self, name, delay_time_ms, next_event):
        super().enter(name)
        self.deadline = time.monotonic_ns() + delay_time_ms * 1000000
        self.next_event = next_event

    def __call__(self):
        if time.monotonic_ns() > self.deadline:
            return self.next_event
        return None

# Special case for the Lambda pump: In case no response is expected to
# a sent command, Send_And_Check cannot be used on layer (B), instead
# Send_Command from layer (C) is used.
from LayerC import Send_Command as Send
```

Listing 20: The *LayerB.py* file contains several state classes, which are in general more complex than layer C states.

8.1.11 LayerC.py

```
# This file contains relevant classes for the generation of the HPLC,
# Lambda and Thermostat drivers. The classes listed in this file are
# possible states on the lowest (third) layer (C). Higher layers (A)
# and (B) can be built from these classes.

# library/modules from python:
import time

# own scripts:
import pyState

```

```python
class Send_Command(pyState.State_Base):
    """This state sends a command to a device."""
    def enter(self, name, msg, com_handle, next_event):
        super().enter(name)
        self.msg = bytearray((msg + "\r").encode("ASCII"))
        self.com_handle = com_handle
        self.next_event = next_event

    def __call__(self):
        self.com_handle.clear_input_buffer()
        self.com_handle.send(self.msg)
        return self.next_event

class Wait_For_Answer(pyState.State_Base):
    """This state waits for the response of a device and checks whether it corresponds to the expected response."""
    def enter(self, name, timeout_ms, com_handle, response_checker, next_event, timeout_event, retry_count, retry_event,
      error_event):
        super().enter(name)
        self.com_handle = com_handle
        self.deadline = time.monotonic_ns() + timeout_ms * 1000000

        self.response_checker = response_checker

        self.next_event = next_event
        self.timeout_event = timeout_event
        self.retry_count = retry_count
        self.retry_event = retry_event
        self.error_event = error_event

        self.response = bytearray()

    def __call__(self):
        end_of_frame = '\r'.encode("ASCII")
        tmp = self.com_handle.receive()
        for chr in tmp:
            if chr == end_of_frame[0]:
                if self.response_checker(self.response):
                    return self.next_event
                if self.retry_count[0] > 0:
                    self.retry_count[0] -= 1
                    return self.retry_event
                print(self.response)
                return self.error_event
            self.response.append(chr)

        if time.monotonic_ns() > self.deadline:
            return self.timeout_event
        return None
```

Listing 21: The *LayerC.py* file contains simple state classes.

8.1.12 Operating_OCAE.py

```python
# This file can be used later to run the OCAE (Output Calculation
# Absolute Evaluation) strategy. For individual experiments, the
# operating point list, the operating time, the dead time, the file
# name and the devices information can be adapted.

# own scripts:
import Auto
import Strategy_OCAE

operating_time = 0.3*60*1E3
dead_time = 0.1*60*1E3
excel_file_name = "strategy_ocae"

# List of operating points
operation_point_list = [
    Strategy_OCAE.operation_point_list_entry(operating_time, 25, [6.1, 6.05]),
    Strategy_OCAE.operation_point_list_entry(operating_time, 25, [6.1, 6.05]),
    Strategy_OCAE.operation_point_list_entry(operating_time, 25, [6.1, 6.05]),
    Strategy_OCAE.operation_point_list_entry(operating_time, 25, [6.1, 6.05]),
]

# Substance data
substance_data = Strategy_OCAE.substance_data([4, 6], [50, 50], [40.01, 60.05], ["B", "A"])

# Devices used
User_Pumps = [["Lambda 1", "COM12"], ["Lambda 3", "COM11"]] # [["HPLC A", "COM12"], ["HPLC B", "COM11"]]
User_Fisher = ["COM8"]
Portname_Calorimeter = "COM6"

# Setting up the strategy
strategy = Strategy_OCAE.Output_Calculation_Absolute_Evaluation(operation_point_list, substance_data, dead_time,
      excel_file_name)

# Setting up the automatization
automat = Auto.matization(strategy, User_Pumps, User_Fisher, Portname_Calorimeter)
```

```
# Automatization is called until the end state is reached
while(True):
    automat.tick()
    if automat.get_state() == "Finished":
        break
    if automat.get_state() == "Error_Thermostat" or automat.get_state() == "Error_Pump" or automat.get_state() == "
        Error_Calorimeter" or automat.get_state() == "Error":
        break
print("Done")
```

Listing 22: The *Operating_OCAE.py* file is used to execute the *Auto.py* file using the *Output Calculation Absolute Evaluation* strategy.

8.1.13 Operating_OPL.py

```
# This file can be used later to run the OPL (Operation Point List)
# strategy. For individual experiments, the operating point list,
# the operating time and the devices information can be adapted.

# own scripts:
import Auto
import Strategy_OPL

operating_time = 0.3*60*1E3

# List of operating points
operation_point_list = [
    Strategy_OPL.operation_point_list_entry(operating_time, 25, [6.1, 6.05]),
    Strategy_OPL.operation_point_list_entry(operating_time, 25, [6.1, 6.05]),
]

# Devices used
User_Pumps = [["Lambda 1", "COM12"], ["Lambda 3", "COM11"]] # [["HPLC A", "COM12"], ["HPLC B", "COM11"]]
User_Fisher = ["COM8"]
Portname_Calorimeter = "COM6"

# Setting up the strategy
strategy = Strategy_OPL.Operation_Point_List(operation_point_list)

# Setting up the automatization
automat = Auto.matization(strategy, User_Pumps, User_Fisher, Portname_Calorimeter)

# Automatization is called until the end state is reached
while(True):
    automat.tick()
    if automat.get_state() == "Finished":
        break
    if automat.get_state() == "Error_Thermostat" or automat.get_state() == "Error_Pump" or automat.get_state() == "
        Error_Calorimeter" or automat.get_state() == "Error":
        break
print("Done")
```

Listing 23: The *Operating_OPL.py* file is used to execute the *Auto.py* file using the *Operation Point List* strategy.

8.1.14 pyState.py

```
# This file contains the basic class for creating a state, from which
# all further states inherit later, and the engine class that is
# responsible for building (this is done via the factory) and running
# the states, which is always called in the state machine.

class State_Base:

    def enter(self, name):
        self.name = name
        # print("entering state --- {}".format(self.name))

    def __call__(self):
        return None

    def exit(self):
        # print("exiting state --- {}".format(self.name))
        return

    def handle_event(self, event):
        return False

    def get_state(self):
        return self.name

class Engine:

    def __init__(self, table, factory, init_state):
```

```
        self.tab = table
        self.fac = factory
        self.init_state = init_state
        self.cur = None

    def enter(self):
        self.cur = self.fac.create_state(self.init_state)

        if self.cur is None:
            raise Exception("Factory has created None")

    def __del__(self):
        self.exit()

    def exit(self):
        if self.cur is not None:
            self.cur.exit()
            self.cur = None

    def search_in_table(self, event):
        for tran in self.tab:
            if not tran[0] == self.cur.get_state():
                continue
            if not tran[1] == event:
                continue

            self.cur.exit()
            self.cur = self.fac.create_state(tran[2])
            return True
        return False

    def tick(self):
        ent = self.cur()
        if ent is None:
            return None
        if self.search_in_table(ent):
            return None
        return ent

    def handle_event(self, event):
        if self.search_in_table(event):
            return True
        if self.cur.handle_event(event):
            return True
        return False

    def get_state(self):
        return self.cur.get_state()
```

Listing 24: The *pyState.py* file contains the basic state class and the engine class. Both classes are used later when creating a state machine.

8.1.15 pyStrategy.py

```
# This file contains the basic class for creating a strategy, from
# which all further strategies inherit later. Inheritance ensures
# that all essential functions are always included in a strategy.

class Strategy_Base:
    class operation_point_information:
        def __init__(self, temperature, flowrate_list):
            self.temperature = temperature
            self.flowrate_list = flowrate_list

        def get_temperature(self):
            return self.temperature

        def get_flowrate(self, idx):
            return self.flowrate_list[idx]

        def get_number_of_pumps(self):
            return len(self.flowrate_list)

    def get_operation_point(self):
        return None

    def push_value(self, value):
        return

    def point_complete(self):
        return False

    def has_error(self):
        return False

    def push_actual_flowrate(self, val):
        return

    def get_finish_instruction(self):
```

```
        return None
```

Listing 25: The *pyStrategy.py* file contains the basic strategy class.

8.1.16 Strategy_OCAE.py

```
# This file contains the Output Calculation Absolute Evaluation
# strategy and its necessary classed.

# library/modules from python:
from enum import Enum
from openpyxl.chart import LineChart, Reference
from openpyxl.styles import Alignment, Border, Font, PatternFill, Side
import math
import time

# own scripts:
import Dictionary
import Excel_Functions
import pyStrategy

class operation_point_list_entry:
    """This class turns the user's input into an object, making it easier to handle the operating points."""
    def __init__(self, time_ms, temperature, flowrate_list):
        self.time_ms = time_ms
        self.temperature = round(temperature)
        self.flowrate_list = flowrate_list

    def get_time_ms(self):
        return self.time_ms

    def get_temperature(self):
        return self.temperature

    def get_flowrate(self, idx):
        return self.flowrate_list[idx]

    def get_number_of_pumps(self):
        return len(self.flowrate_list)

class substance_data:
    def __init__(self, weighing_g, volume_ml, molar_mass_gpermol, pump_substance_assignment_list):
        self.weighing = weighing_g
        self.volume = volume_ml
        self.molar_mass = molar_mass_gpermol
        self.list = pump_substance_assignment_list

        if len(self.weighing) != 2 or len(self.volume) != 2 or len(self.molar_mass) != 2:
            raise Exception("Substance data is not complete")

    def get_weighing(self):
        return self.weighing

    def get_volume(self):
        return self.volume

    def get_molar_mass(self):
        return self.molar_mass

    def get_concentration(self):
        concentration = [] # mol/l
        for idx in range(2):
            concentration.append(self.weighing[idx]/(self.volume[idx] * 1E-3)/self.molar_mass[idx])
        return concentration

    def get_pump_substance_assignment_list(self):
        return self.list

class Output_Calculation_Absolute_Evaluation(pyStrategy.Strategy_Base):
    class States(Enum):
        TEMPERATURE_EQUILIBRATION = 0,
        SETTING_DEADLINE = 1,
        WAITING_FOR_DEADLINE = 2,

    def __init__(self, operation_point_list, substance_data, dead_time_ms, excel_name):
        self.list = operation_point_list
        self.substance_data = substance_data
        self.dead_time = dead_time_ms

        self.idx = 0
        self.cur_temp = float("NaN")
        self.cur_deadline = 0
        self.min_time = 0
        self.cur_operation_point = None
        self.state = Output_Calculation_Absolute_Evaluation.States.TEMPERATURE_EQUILIBRATION
        self.datalist = []

        # variables for calculation
        self.process_point = 0

```

```python
        # create excel file
        self.excel_name = excel_name
        [self.workbook, self.sheet, self.counter] = Excel_Functions.create_excel(self.substance_data, self.excel_name)

        # sanity check
        for idx in range(len(self.list)):
            if not len(self.substance_data.list) == len(self.list[idx].flowrate_list):
                raise Exception("Length of substance pump assignment list and flow rate list do not match")

            if not self.dead_time < self.list[idx].time_ms:
                raise Exception("The dead time is longer than the operating time, so there is no evaluation time")

            try:
                tmp = "{:d}".format(int(self.list[idx].temperature))
                Dictionary.calorimeter_thermostat[tmp]
            except KeyError:
                raise Exception("No calorimeter calibration is given for the given set temperature")

    def get_operation_point(self):
        if not self.idx < len(self.list):
            self.cur_operation_point = None
            return None

        self.cur_operation_point = self.list[self.idx]
        if not self.cur_operation_point.get_temperature() == self.cur_temp:
            self.cur_deadline = time.monotonic_ns() + 10 * 60 * 1E9
            self.state = Output_Calculation_Absolute_Evaluation.States.TEMPERATURE_EQUILIBRATION
            self.cur_temp =  self.cur_operation_point.get_temperature()
            return pyStrategy.Strategy_Base.operation_point_information(self.cur_temp, [0] * self.cur_operation_point.get_number_of_pumps())

        self.state = Output_Calculation_Absolute_Evaluation.States.SETTING_DEADLINE
        self.idx += 1
        return pyStrategy.Strategy_Base.operation_point_information(self.cur_temp, self.cur_operation_point.flowrate_list)

    def push_value(self, line):
        if line is None:
            return

        self.datalist.append(line)
        self.sheet[1].append(line)

        if not self.state == Output_Calculation_Absolute_Evaluation.States.WAITING_FOR_DEADLINE:
            return

        # one-time calculation
        if self.min_time < time.monotonic_ns() and self.waiting_counter == 0:
            self.starting_idx = len(self.datalist)-1
            self.waiting_counter = 1

            self.process_point += 1
            self.evalutaion_time = [self.datalist[self.starting_idx][0], None]
            self.set_volume_flowrate = [0, 0]
            self.actual_volume_flowrate = [0, 0]
            self.actual_molar_flowrate = [0, 0]
            self.actual_water_molar_flowrate = [0, 0]

            for idx in range(1,4):
                self.sheet[0].insert_rows(idx=self.counter[idx][1], amount=1)
                if idx == 1:
                    for jdx in range(1,4):
                        self.counter[jdx][1] += 1
                        if jdx != 1:
                            self.counter[jdx][0] += 1
                if idx == 2:
                    for jdx in range(2,4):
                        self.counter[jdx][1] += 1
                    self.counter[3][0] += 1
                if idx == 3:
                    self.counter[idx][1] += 1

            for idx in range(len(self.substance_data.list)):
                if self.substance_data.list[idx] == "A":
                    jdx = 0
                if self.substance_data.list[idx] == "B":
                    jdx = 1
                self.set_volume_flowrate[jdx] += self.cur_operation_point.flowrate_list[idx]
                self.actual_volume_flowrate[jdx] += self.actual_flowrate_list[idx]
                self.actual_molar_flowrate[jdx] += self.actual_flowrate_list[idx] * self.substance_data.get_concentration()[idx] / 6E4
                self.actual_water_molar_flowrate[jdx] += self.actual_flowrate_list[idx] * Dictionary.calculation_data["concentration"] / 6E4

            # process setup entry
            self.sheet[0].cell(row=self.counter[1][1]-1, column=1).value = self.process_point
            self.sheet[0].cell(row=self.counter[1][1]-1, column=2).value = self.evalutaion_time[0]
            self.sheet[0].cell(row=self.counter[1][1]-1, column=4).value = self.set_volume_flowrate[0]
            self.sheet[0].cell(row=self.counter[1][1]-1, column=5).value = self.actual_volume_flowrate[0]
            self.sheet[0].cell(row=self.counter[1][1]-1, column=6).value = self.actual_molar_flowrate[0]
            self.sheet[0].cell(row=self.counter[1][1]-1, column=7).value = self.actual_water_molar_flowrate[0]
            self.sheet[0].cell(row=self.counter[1][1]-1, column=8).value = self.set_volume_flowrate[1]
            self.sheet[0].cell(row=self.counter[1][1]-1, column=9).value = self.actual_volume_flowrate[1]
            self.sheet[0].cell(row=self.counter[1][1]-1, column=10).value = self.actual_molar_flowrate[1]
```

```python
            self.sheet[0].cell(row=self.counter[1][1]-1, column=11).value = self.actual_water_molar_flowrate[1]

        # ongoing calculation
        if self.waiting_counter == 1:
            mean_values = []
            temp_difference = []
            heat_flux_outside = []
            heat_flux_reactor = None
            enthalpy_difference = None

            # process setup entry
            self.evalutaion_time[1] = self.datalist[len(self.datalist)-1][0]
            self.sheet[0].cell(row=self.counter[1][1]-1, column=3).value = self.evalutaion_time[1]

            # raw data processing entry (mean values)
            self.sheet[0].cell(row=self.counter[2][1]-1, column=1).value = self.process_point
            for idx in range(5,11):
                mean = 0
                counter = 0
                for jdx in range(self.starting_idx, len(self.datalist)):
                    counter += 1
                    mean += self.datalist[jdx][idx]
                mean_values.append(mean/counter)
                self.sheet[0].cell(row=self.counter[2][1]-1, column=idx-3).value = mean_values[idx-5]

            # raw data processing and calculation entry (temperature difference and outside heat flux)
            for idx in range(3):
                temp_difference.append(self.cur_operation_point.temperature - mean_values[idx])
                self.sheet[0].cell(row=self.counter[2][1]-1, column=idx+8).value = temp_difference[idx]

                if not idx == 2:
                    tmp = self.actual_volume_flowrate[idx] * Dictionary.calculation_data["concentration"] * Dictionary.calculation_data["cp"] * temp_difference[idx] / 6E4
                    heat_flux_outside.append(tmp)
                    self.sheet[0].cell(row=self.counter[3][1]-1, column=idx+2).value = heat_flux_outside[idx]

                else:
                    tmp = sum(self.actual_water_molar_flowrate) * Dictionary.calculation_data["cp"] * temp_difference[idx]
                    heat_flux_outside.append(tmp)
                    self.sheet[0].cell(row=self.counter[2][1]-1, column=self.counter[2][2]).value = heat_flux_outside[idx]

            # calculation entry (reactor heat flux and enthalpy difference)
            self.sheet[0].cell(row=self.counter[3][1]-1, column=1).value = self.process_point
            heat_flux_reactor = Dictionary.calorimeter_thermostat["{:d}".format(int(self.cur_operation_point.temperature))].forward(mean_values[3:])
            heat_flux_reactor.insert(1, heat_flux_reactor[0]-sum(heat_flux_outside[:2]))

            for idx in range(len(heat_flux_reactor)):
                self.sheet[0].cell(row=self.counter[3][1]-1, column=idx+4).value = heat_flux_reactor[idx]

            enthalpy_difference = (sum(heat_flux_reactor[1:])+heat_flux_outside[2]) / (self.actual_molar_flowrate[0]*1000)
            self.sheet[0].cell(row=self.counter[3][1]-1, column=self.counter[3][2]).value = enthalpy_difference

            # save changes
            self.workbook.save("{}.xlsx".format(self.excel_name))

    def point_complete(self):
        if self.state == Output_Calculation_Absolute_Evaluation.States.TEMPERATURE_EQUILIBRATION:
            val = 10
            if len(self.datalist) < val:
                return False

            for col_idx in [2, 3, 4]:
                valid_count = 0
                for idx in range(val):
                    if abs(self.datalist[len(self.datalist)-1-idx][col_idx] - self.cur_operation_point.get_temperature()) < 0.1:
                        valid_count += 1
                if valid_count < math.ceil(val*0.9):
                    return False
                    # return True if dummy is used
            return True
        elif self.state == Output_Calculation_Absolute_Evaluation.States.SETTING_DEADLINE:
            self.cur_deadline = time.monotonic_ns() + self.cur_operation_point.get_time_ms() * 1E6
            self.min_time = time.monotonic_ns() +  self.dead_time
            self.state = Output_Calculation_Absolute_Evaluation.States.WAITING_FOR_DEADLINE
            self.waiting_counter = 0
            return False
        elif self.state == Output_Calculation_Absolute_Evaluation.States.WAITING_FOR_DEADLINE:
            if self.cur_deadline < time.monotonic_ns():
                self.workbook.save("{}.xlsx".format(self.excel_name))
                return True
            else:
                return False
        raise Exception("You should not land here")

    def has_error(self):
        if not self.state == Output_Calculation_Absolute_Evaluation.States.TEMPERATURE_EQUILIBRATION:
            return False

        if self.cur_deadline < time.monotonic_ns():
            print("set_temp is not reached at the reactor")
            return True
        return False
```

```python
    def push_actual_flowrate(self, val):
        self.actual_flowrate_list = val

    def get_finish_instruction(self):
        # generate charts
        Dia_Raw_Temp = LineChart()

        Dia_Raw_Temp.y_axis.title = "Temperature [°C]"
        y_data = Reference(self.sheet[1], min_col = 2, min_row = 1, max_col = 8, max_row = len(self.datalist)+1)
        Dia_Raw_Temp.add_data(y_data, titles_from_data = True)

        Dia_Raw_Temp.x_axis.title = "Time [s]"
        Dia_Raw_Temp.x_axis.tickLblSkip = math.ceil(len(self.datalist)/10)
        x_data = Reference(self.sheet[1], min_col = 1, min_row = 2, max_row = len(self.datalist)+1)
        Dia_Raw_Temp.set_categories(x_data)

        chart1 = self.workbook.create_chartsheet("Dia_Raw_Temp")
        chart1.add_chart(Dia_Raw_Temp)

        Dia_Raw_Voltage = LineChart()

        Dia_Raw_Voltage.y_axis.title = "Voltage [mV]"
        y_data = Reference(self.sheet[1], min_col = 9, min_row = 1, max_col = 11, max_row = len(self.datalist)+1)
        Dia_Raw_Voltage.add_data(y_data, titles_from_data = True)

        Dia_Raw_Voltage.x_axis.title = "Time [s]"
        Dia_Raw_Voltage.x_axis.tickLblSkip = math.ceil(len(self.datalist)/10)
        x_data = Reference(self.sheet[1], min_col = 1, min_row = 2, max_row = len(self.datalist)+1)
        Dia_Raw_Voltage.set_categories(x_data)

        chart2 = self.workbook.create_chartsheet("Dia_Raw_Voltage")
        chart2.add_chart(Dia_Raw_Voltage)

        # formatting
        substance_a_color = "3BCCFF"
        substance_b_color = "3D33FF"
        result_color = "FF087F"
        add_data_color = "4B0082"

        for idx in range(4):
            self.sheet[0].cell(row=self.counter[idx][0], column=1).font = Font(bold=True)
            self.sheet[0].cell(row=self.counter[idx][0], column=1).alignment = Alignment(horizontal="center")
            self.sheet[0].merge_cells(start_row=self.counter[idx][0], start_column=1, end_row=self.counter[idx][0], end_column=self.counter[idx][2])

            for jdx in range(1, self.counter[idx][2]+1):
                self.sheet[0].cell(row=self.counter[idx][0]+1, column=jdx).border = Border(bottom=Side(border_style="thick"))

        for idx in range(self.process_point):
            self.sheet[0].cell(row=self.counter[1][0]+idx+2, column=5).fill = PatternFill("lightUp", fgColor=substance_a_color)
            self.sheet[0].cell(row=self.counter[1][0]+idx+2, column=9).fill = PatternFill("lightUp", fgColor=substance_b_color)
            self.sheet[0].cell(row=self.counter[3][0]+idx+2, column=8).fill = PatternFill("lightUp", fgColor=result_color)

        for idx in range(1, self.counter[0][2]+1):
            self.sheet[0].cell(row=3, column=idx).fill = PatternFill("lightTrellis", fgColor=substance_a_color)
            self.sheet[0].cell(row=4, column=idx).fill = PatternFill("lightTrellis", fgColor=substance_b_color)

        self.sheet[0].cell(row=2, column=8).font = Font(bold=True)
        self.sheet[0].cell(row=2, column=8).alignment = Alignment(horizontal="center")
        self.sheet[0].merge_cells(start_row=2, start_column=8, end_row=2, end_column=9)
        for idx in range(2):
            self.sheet[0].cell(row=idx+3, column=9).fill = PatternFill("lightTrellis", fgColor=add_data_color)
            self.sheet[0].cell(row=3, column=idx+8).border = Border(top=Side(border_style="thick"))

        self.workbook.save("{}.xlsx".format(self.excel_name))
```

Listing 26: The *Strategy_OCAE.py* file corresponds to a concrete strategy of the strategy pattern and contains the strategy for evaluating the measurement data and for creating an Excel output file.

8.1.17 Strategy_OPL.py

```python
# This file contains the Operation Point List strategy and its
# necessary classes.

# library/modules from python:
from enum import Enum
import math
import time

# own scripts:
import pyStrategy

class operation_point_list_entry:
    """This class turns the user's input into an object, making it easier to handle the operating points."""
    def __init__(self, time_ms, temperature, flowrate_list):
        self.time_ms = time_ms
```

```python
        self.temperature = temperature
        self.flowrate_list = flowrate_list

    def get_time_ms(self):
        return self.time_ms

    def get_temperature(self):
        return self.temperature

    def get_flowrate(self, idx):
        return self.flowrate_list[idx]

    def get_number_of_pumps(self):
        return len(self.flowrate_list)

class Operation_Point_List (pyStrategy.Strategy_Base):
    class States(Enum):
        TEMPERATURE_EQUILIBRATION = 0,
        SETTING_DEADLINE = 1,
        WAITING_FOR_DEADLINE = 2,

    def __init__(self, operation_point_list):
        self.list = operation_point_list
        self.idx = 0
        self.cur_temp = float("NaN")
        self.cur_deadline = 0
        self.cur_operation_point = None
        self.state = Operation_Point_List.States.TEMPERATURE_EQUILIBRATION
        self.datalist = []

    def get_operation_point(self):
        if not self.idx < len(self.list):
            self.cur_operation_point = None
            return None

        self.cur_operation_point = self.list[self.idx]
        if not self.cur_operation_point.get_temperature() == self.cur_temp:
            self.cur_deadline = time.monotonic_ns() + 10 * 60 * 1E9
            self.state = Operation_Point_List.States.TEMPERATURE_EQUILIBRATION
            self.cur_temp =  self.cur_operation_point.get_temperature()
            return pyStrategy.Strategy_Base.operation_point_information(self.cur_temp, [0] * self.cur_operation_point.get_number_of_pumps())

        self.state = Operation_Point_List.States.SETTING_DEADLINE
        self.idx += 1
        return pyStrategy.Strategy_Base.operation_point_information(self.cur_temp, self.cur_operation_point.flowrate_list)

    def push_value(self, line):
        if line is not None:
            self.datalist.append(line)

    def point_complete(self):
        if self.state == Operation_Point_List.States.TEMPERATURE_EQUILIBRATION:
            val = 10
            if len(self.datalist) < val:
                return False

            for col_idx in [2, 3, 4]:
                valid_count = 0
                for idx in range(val):
                    if abs(self.datalist[len(self.datalist)-1-idx][col_idx] - self.cur_operation_point.get_temperature()) < 0.1:
                        valid_count += 1
                if valid_count < math.ceil(val*0.9):
                    # return False
                    return True
            return True
        elif self.state == Operation_Point_List.States.SETTING_DEADLINE:
            self.cur_deadline = time.monotonic_ns() + self.cur_operation_point.get_time_ms() * 1E6
            self.state = Operation_Point_List.States.WAITING_FOR_DEADLINE
            return False
        elif self.state == Operation_Point_List.States.WAITING_FOR_DEADLINE:
            if self.cur_deadline < time.monotonic_ns():
                return True
            else:
                return False
        raise Exception("You should not land here")

    def has_error(self):
        if not self.state == Operation_Point_List.States.TEMPERATURE_EQUILIBRATION:
            return False

        if self.cur_deadline < time.monotonic_ns():
            print("set_temp is not reached at the reactor")
            return True
        return False
```

Listing 27: The *Strategy_OPL.py* file corresponds to a concrete strategy of the strategy pattern and contains the strategy which does not yet further restrict the handling of the system.

8.2 Output Calculation from Measurement Data

The molar reaction enthalpy is to be calculated from the measurement data obtained from the calorimeter. For this purpose, the actual volumetric flow rates $\dot{V}_{A,act}$ and $\dot{V}_{B,act}$ are determined in the first step using the target flow rates $\dot{V}_A$ and $\dot{V}_B$ and the known calibration values K_i of the used pumps (Equation 8 and 9).

$$\dot{V}_{A,act} = \frac{\dot{V}_A}{K_i} \tag{8}$$

$$\dot{V}_{B,act} = \frac{\dot{V}_B}{K_i} \tag{9}$$

Based on the known concentrations c_A and c_B of the two components and the previously calculated actual volume fluxes, the actual mole fluxes $\dot{n}_{A,act}$ and $\dot{n}_{B,act}$ are derived (Equation 10 and 11).

$$\dot{n}_{A,act} = \frac{\dot{V}_{A,act}}{60} \cdot \frac{c_A}{10^3} \tag{10}$$

$$\dot{n}_{B,act} = \frac{\dot{V}_{B,act}}{60} \cdot \frac{c_B}{10^3} \tag{11}$$

In the next step, certain measurement data are averaged over the time interval relevant for the evaluation of the operating point. These specific measurement data include the temperatures T_A and T_B of the two components at the inlet, the temperature T_{out} at the outlet and the measured voltages U_{pre}, U_{r1} and U_{r2} in each of the three segments of the calorimeter. Subsequently, the temperature differences ΔT_A, ΔT_B and ΔT_{out} to the set temperature T_{set} are calculated from the three averaged temperatures (Equation 12-14).

$$\Delta T_A = T_{set} - T_A \tag{12}$$

$$\Delta T_B = T_{set} - T_B \tag{13}$$

$$\Delta T_{out} = T_{set} - T_{out} \tag{14}$$

The general heat balance given in Equation 15 applies to the reactor plate of the calorimeter. The temporal change of the heat quantity is given by the convective $\dot{Q}_{conv}$, the transmitted $\dot{Q}_{tran}$ and the reaction heat flux $\dot{Q}_{reac}$.

$$\frac{dQ}{dt} = -\dot{Q}_{conv} - \dot{Q}_{tran} + \dot{Q}_{reac} \tag{15}$$

Using the calculated temperature differences, the convective heat flows $\dot{Q}_{\mathrm{A}}$ and $\dot{Q}_{\mathrm{B}}$ at the inlet and the convective heat flow $\dot{Q}_{\mathrm{out}}$ at the outlet of the calorimeter can be computed. To calculate these heat fluxes, the actual volumetric flow rates, the concentration of water c_{water} and the specific heat capacity of water $c_{\mathrm{p,water}}$ are needed in addition to the temperature differences (Equation 16-18).

$$\dot{Q}_{\mathrm{A}} = \frac{\dot{V}_{\mathrm{A}}}{60} \cdot \frac{c_{\mathrm{water}}}{10^3} \cdot \Delta T_{\mathrm{A}} \cdot c_{\mathrm{p,water}} \tag{16}$$

$$\dot{Q}_{\mathrm{B}} = \frac{\dot{V}_{\mathrm{B}}}{60} \cdot \frac{c_{\mathrm{water}}}{10^3} \cdot \Delta T_{\mathrm{B}} \cdot c_{\mathrm{p,water}} \tag{17}$$

$$\dot{Q}_{\mathrm{out}} = \left(\frac{\dot{V}_{\mathrm{A,act}}}{60} \cdot \frac{c_{\mathrm{water}}}{10^3} + \frac{\dot{V}_{\mathrm{B,act}}}{60} \cdot \frac{c_{\mathrm{water}}}{10^3} \right) \cdot \Delta T_{\mathrm{out}} \cdot c_{\mathrm{p,water}} \tag{18}$$

The transmitted heat flux is derived from the measured data. For this purpose, the calibration curve of the calorimeter is evaluated for each segment as given in Equation 19. The corresponding heat quantity $\dot{Q}_{seg}$ is determined from the measured voltage U_{seg}. The transmitted heat flux then results from the sum of these three heat quantities $\dot{Q}_{\mathrm{pre}}$, $\dot{Q}_{\mathrm{r1}}$ and $\dot{Q}_{\mathrm{r2}}$ (Equation 20).

$$\dot{Q}_{seg} = (-1) \cdot \left(a \cdot U_{seg}^2 + b \cdot U_{seg} + c \right) \tag{19}$$

$$\dot{Q}_{\mathrm{tran}} = \dot{Q}_{\mathrm{pre}} + \dot{Q}_{\mathrm{r1}} + \dot{Q}_{\mathrm{r2}} \tag{20}$$

Since in a steady state operation the change of the heat quantity equals zero, regarding the heat balance in Equation 15 the reaction heat flux results in the sum of the convective and the transmitted heat fluxes (Equation 21).

$$\dot{Q}_{\mathrm{reac}} = \dot{Q}_{\mathrm{conv}} + \dot{Q}_{\mathrm{tran}} \tag{21}$$

Finally, the molar reaction enthalpy Δh_{R} can be calculated from the reaction heat flux and the actual mole flux of the limiting component assuming complete conversion (Equation 22).

$$\Delta h_{\mathrm{R}} = \frac{\dot{Q}_{\mathrm{reac}}}{\dot{n}_{\mathrm{A,act}} \cdot 10^3} = \frac{-\dot{Q}_{\mathrm{A}} - \dot{Q}_{\mathrm{B}} + \dot{Q}_{\mathrm{out}} + \dot{Q}_{\mathrm{pre}} + \dot{Q}_{\mathrm{r1}} + \dot{Q}_{\mathrm{r2}}}{\dot{n}_{\mathrm{A,act}} \cdot 10^3} \tag{22}$$