UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

HAW HAMBURG

TFG del Grado en Ingeniería Informática

Annex A: Software Development Plan

Presented by Helen Haase
at Universidad de Burgos
January 17, 2022
Tutor: Bruno Baruque Zanón

Contents

List of Figures

List of Tables

Chapter 1

List of Acronyms

A2C Advantage Actor Critic

A3C Asynchronous Advantage Actor Critic

AGV Automated Guided Vehicle

DRL Deep Reinforcement Learning

DQN Deep Q-Network

EDD Earliest Due Date

ERM Experience Replay Buffer

FIFO First In First Out

FMS Flexible Manufacturing System

FJSP Flexible Job Shop Problem

IID independent and identically distributed

JSP Job Shop Problem

KPI Key Performance Indicator

KBGA Knowledge Based Genetic Algorithm

MDP Markov Decision Process

ML Maximal Lateness

MP Matrix Production

NN Neuronal Network

NTJ Number of Tardy Jobs

ODBC Open Database Connectivity

OPC UA Open Platform Communications Unified Architecture

PPO Proximal Policy Optimization

RL Reinforcement Learning

SGA Simple Genetic Algorithm

SRPT Shortest Remaining Process Time

SPS Siemens Plant Simulation

SPT Shortest Processing Time

TD Temporal Difference

TCT Total Completion Time

TL Total Lateness

TRPO Trust Region Policy Optimization

TT Total Tardiness

UU Uptime Utilization

WIP Work in Progress

XML Extensible Markup Language

Chapter 2

Introduction

The appendix has the purpose of presenting information specific to the software. It presents the software design process. Therefore the relevant aspects of the project management, the software requirements engineering, and the planning will be presented. The requirements engineering of all main components are explained as well as the setup for the user.

Chapter 3

Software Project Plan

3.1 Project Management

3.1.1 Scrum

Scrum is a framework used for developing and maintaining software too complex to be planned in all aspects in an early stage.

As this concept is usually used for development work in teams and not for a single person, the ideas have been adapted accordingly. In a team-based scenario different types of roles, including for example the Scrum Master and Project Owner. Additionally so called artefacts are used to organize interaction within the team. One artefact is called a sprint defining a development phase of usually two weeks. At the beginning of a sprint, a planning phase defines the goals to be achieved. A number of already defined tasks is selected and assigned a value (story point) representing the estimated time that is needed to fulfill the task at hand. This ensures, that the amount of tasks fits the defined time span.

In this project each sprint starts with a planning phase involving both author and supervisor. At the end of the sprint, the author presents the work and the finished tasks.

3.1.2 Software

3.1.2.1 Research and Planning of the Project

Notion[3] was used for the research portion of the project. It offers to share and organize files, but also to take notes or add keywords to documents a helpful feature for managing a number of documents. Documentation drafts and literature management as well as protocol management were made and shared with this tool.

3.1.2.2 GitHub and ZenHub

For version control of the software, *GitHub*[2] was used. To track the project progress as well a define milestones and deadlines, *ZenHub*[7] (an agile project mangement tool) was integrated.

3.2 Time Management

The kick-off meeting took place on the 28th of September 2021 to discuss both the idea and the scope of the work.
To give an overview of the process each of the iterations will be outlined in the following sections. Every iteration lasted around two weeks. Some of the context research was done before presenting the idea, so it is not listed as individual iteration.

A velocity tracking graph is not shown, as the meetings often took place on the first day of the sprint, so that the issues had not been closed yet. If a graph was shown, it would have shown sprints without the clossed issues, which does not reflect the reality.
Instead the following overview over the iterations is given.

3.2.1 Iteration 1 (Sep 28 - Oct 14): Setup and Further Research

The first iteration focused on setting up the infrastructure of the project. The repository as well as the project board was created. The previously done research was extended and the database of the read papers updated.

Research relating the tools and technologies of the project was done. To get familiar with Siemens Plant Simulation, small protoypes have been built.

3.2.2 Iteration 2 (Oct 14- Oct 28): Simulation Environment: Building the Model

This iteration was dedicated to building the model. For this goal, the gained expertise in the software was deepend. A first setup was done, types of parts defined, a work plan created and the communication interfaces examined. The communication between the software components was planned.

During all of the following iterations, the related chapters have been drafted for later review.

3.2.3 Iteration 3 (Oct 28 - Nov 11): Reinforcement Algorithm: The Communication

In this iteration the communication between simulation and a python script was developed.

3.2.4 Iteration 4 (Nov 11 - Nov 25): Reinforcement Algorithm: The Training Process

In this iteration the training process of the agent was finalized. The used communication was stress tested.

3.2.5 Iteration 5 (Nov 25 - Dec 9): Experiments: Setup and Automation

The focus of this iteration was to plan and automatize the experiments. To reach this goal, an experiment module was developed and linked into the existing software.

3.2.6 Iteration 6 (Dec 9 - Dec 23): Experiments: Performance and Evaluation

Problems caused by the interaction between the agent and simulation slowed down this iteration. The bug was fixed, the first experiments were run and the documentation was continued.

3.2.7 Iteration 7 (Dec 23 - Jan 9): Documentation

The last experiments were performed and the results collected.

Further parts of the documentation were added. Focus of this iteration were the evaluation of the data collected and the conclusion of the work.

3.2.8 Iteration 8 (Dec 10 - Jan 20): Proof Reading and Presentation

During the last iteration smaller adjustments to the software were implemented, the documentation and the presentation were finalized.

3.3 Viability Study

This section comprises an overview of the economical and legal viability of this work.

As this project is research oriented, the funding needs to be found either through a university or by industry sponsorship. This is the reason the following chapter gives an overview of the expenses incurred, but not a full finance plan.

3.3.1 Economical Viability

3.3.1.1 Personnel costs

The project presented has been implemented by the author as her bachelor thesis. As the author is a computer science student without a university degree, the hourly wage is estimated to be around 13€/h [1]. The finalization of the bachelor thesis is valued with 12 ECTS, which are equivalent to around 360 working hours.
Velocity tracking confirms the approximate amount of spent time on the project. It show, that 412 story points have been fulfilled during the project. Each story point is equalling around 1 working hour.

This leads to the following personnel expenses:

$412h * 13€/h = 5.356$ €

3.3.1.2 Hardware costs

For this project a Lenovo laptop has been used, that has been purchased in 2021 for around 800 €. As this device has not solely been purchased for this project, the costs can be amortized. In Germany the amortization period for a laptop is one year as of 2021 [4]. The following calculation shows the hardware expenses incurred:

Amortization period in hours = 8760h
Development period in hours = 412h
Hardware Cost per hour = $800€/8760h = 0.09€/h$
Hardware Costs during development = $0.09€/h * 412h = 37.08€$

Total hardware costs are 37.08€.

3.3.1.3 Software costs

The following table 3.1 outlines the software expenses.

Software	License Type	Cost
Windows 10 Pro	Windows 10 Pro	34.99€
Siemens Plant Simulation	Student License	0.00€
Notion	Personal Version	0.00€
Overleaf	Student Version	0.00€

Table 3.1: Software costs

No software was used, that required purchasing a license. Developed further for active use, it would be recommended to acquire a higher license level for åSPS, as this offers the option for a significantly faster communication between the software components.

Furthermore it is to say larger experiments would require a more powerful hardware to train the agent. The resulting costs should therefore be taken into account.

3.3.1.4 Total Costs

The mentioned partial costs lead to total cost as presented in table 3.2

Type	Cost
Personnel Costs	5.356€
Hardware Costs	37.08€
Software Costs	34.99€
Total Costs	5.428,07€

Table 3.2: Total costs

Even though the prototype of the software is not for commercial use, the author had the clear benefit of learning the basics of modelling with SPS and of Reinforcement Learning as well as the rules of a scientific approach to research.

3.3.2 Legal Viability

As this project had no budget, there was a need to use free of charge software only. However it is important to mention, that the academic license of Siemens Plant Simulation *may only be used for non-commercial purposes.These licenses can be used for lectures, research purposes as well as diploma theses and student research projects or other science-related projects.*[5]

Chapter 4

Software Requirement Engineering

4.1 Introduction

In this chapter the general software objectives will be presented. A use case diagram will show the general features for the end user. This then leads to formulate the functional as well as the system requirements. The requirements will get more detailed in the course of this chapter, enabling a cleaner development process.

4.2 General Objectives

Besides the objectives mentioned in chapter 2 of the Report, the software aims to support the investigation of Reinforcement Learning for modular production systems. To fulfill this requirement the following goals have been defined:

1. Modelling a production system with Siemens Plant Simulation Students License. This includes providing a setup with two different types of parts, that need to be manufactured according to a work plan. The simulation may be fully controlled by the Reinforcement Learning (RL) agent and therefore provides information about the current state and the reward properties via a XML-interface.
2. The implemented RL agent will learn how to effectively control the production system. Therefore the agent also needs to provide an XML-

interface for data exchange. The agent shall be optimized through hyperparameter tuning.

3. A series of experiments will be performed to tune the RL agent. This involves the development of an easily extendable experiment environment.The experiments will include a variation of action space, state space, reward calculation and the common hyperparameters.

4.3 Software Requirement Catalogue

This section outlines the requirements of the software. It aims to give a clear definition of the software requirements before implementation. In a real world scenario this would be the foundation of the contract between the client and the company charged with implementing the software.

In comparison with most other software, this one does not stand in continuous and immediate interaction with an end user. It rather serves the purpose of simulating a scenario defined by the user and to generate sufficient data the user can use for further analysis and experimentation. Hence the use case diagram of the software is small and can be seen in figure 4.1.

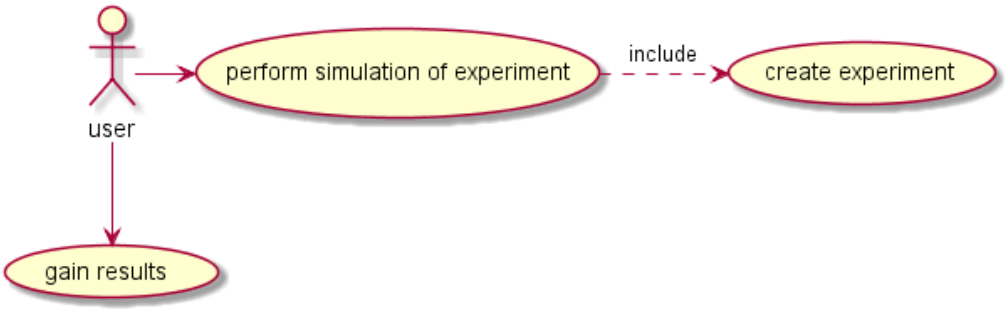

Figure 4.1: Use case diagram for the users side

In order to enable the user to perform experiment, these will need to be defined beforehand. In the scope of this work the configurations of the simulation environment will be static, while the hyperparameters and the employed RL agent are configurable within a reasonable range. The collected results shall include the trained agent as well a numerical data.

In order to structure software requirements, the software is split into three main components as shown in figure 4.2 according to their tasks. These are the simulation, interface for the data exchange and the learning agent.

Figure 4.2: Overview of the components

To give a more granular overview, the system requirements are matched to categories (compare list 4.3.1 - 4.3.3), fitting with each task. Each sub task is then explained in a use case diagram 4.1 - 4.11.

4.3.1 Simulation System Requirements: ID-01-xx

- simulation needs to include a work plan, defining which kind of parts are put in the system and in which way they are built into the final product. [ID-01-01]
- During simulation it must be ensured that each item is moved only to valid stations and follows only the correct order of production. [ID-01-02]
- While simulation is running it also has to react to a variety of triggers by the data exchange interface in order to keep communication between software components as consistent as possible [ID-01-03]
- Parts need to be spawned in an organized manner and despawn when their processing is finalized. [ID-01-04 + ID-01-05]
- Training iterations required by the RL agent need to be performed by the environment [ID-01-06]

4.3.2 Data Exchange Interface System Requirements: ID-02-xx

- The interface needs to be able to exchange information between the agent and the simulation. Therefore the information has to be encoded in a consistent format, that is easy to extend and adapt, so that changes are possible [ID-02-01 + ID-02-02]

4.3.3 Reinforcement Learning Agent System requirements: ID-03-xx

- According to setup of simulation the agent shall receive a finite defined action and state space [ID-03-01]
- Consequently to the chosen action and its effect, the agent has to receive a calculated reward [ID-03-02]
- Within the agent training, the agent has to be able to choose actions. Before transmitting a chosen action to the simulation the validity of the action has to be checked. [ID-03-03 + ID-03-04]

4.4 Requirement Specification

4.4.1 Simulation [ID-01]

The main requirement of *Simulation* is to setup a production system. The production system will serve as environment for the training of the RL agent. Its limitations and constraints are defined in chapter 4.5.

Table 4.1: Use case: Defining a work plan: ID-01-01

Use case	**Defining a work plan: ID-01-01**
Description	The work plan includes the different manufactured parts, as well as the required processing steps and processing type of each part at each machine.
Preconditions	Simulation has not started yet
Postconditions	Every machine can access its processing time via work plan
Trigger	-
Control flow	No flow control, this is a static setup before simulation start

Table 4.2: Use case: Following a work plan: ID-01-02

Use case	**Following a work plan: ID-01-02**
Description	During the process the machine accesses the work plan to read its processing time depending on the part it finds
Preconditions	Work plan is defined
Postconditions	Processing time is set for current process
Trigger	Part reaches machine
Control flow	• Part reaches machine • Machine reads out processing time for work plan

Table 4.3: Use case: Reacting to triggers: ID-01-03

Use case	**Reacting to triggers: ID-01-03**
Description	In order to control training process the simulation has to react to triggers
Preconditions	Receiving one of the triggers listed in "Trigger"
Postconditions	-
Trigger	• **1:** Command.xml written includes info to: – **a:** reset the simulation – **b:** move item – **c:** notice setup is done – **d:** notice training is done • **2:** Part reaches machine exit
Control flow	• on **1a:** reset simulation, reset variables • on **1b:** wait until process is done, move item from source to destination (defined in command.xml) • on **1c:** start simulation • on **1d:** stop simulation • on **2:** start iteration (4.6)

Table 4.4: Use case: Spawning of parts: ID-01-04

Use case	**Spawning of parts: ID-01-04**
Description	Items shall be spawned in the source, whenever the source is empty
Preconditions	Spawning: simulation is started, source is empty
Postconditions	source is occupied
Trigger	-
Control flow	-

Table 4.5: Use case: Despawning of parts: ID-01-05

Use case	**Despawning of parts: ID-01-05**
Description	A part shall be despawned when it reaches the drain
Preconditions	Part is completely produced
Postconditions	Part is removed from system, production statistics include production of this part
Trigger	Part enters drain
Control flow	• Part enters drain • Production statistics are updated • if the demand is exceeded, the termination of the operation is signaled • despawn item

Table 4.6: Use case: Perform iteration: ID-01-06

Use case	**Perform iteration: ID-01-06**
Description	Performing an iteration of state-move-reward cycle
Preconditions	only one thread may perform an iteration, the action must be valid, simulation is started, work plan is defined
Postconditions	-
Trigger	Part is in exit of machine
Control flow	• Part is in exit (which turns the current state into a state, that requires an action) • Write current state • Wait until an action is written • Perform action – Wait until current processing step of machine is finished – Move Part from source to destination (defined in command) • Write reward information (includes terminal) • if state is terminal, write state information again

Note, that besides the training process consisting of multiple episodes (compare: [Perform iteration: ID-01-06]) an evaluation process takes place. The difference is that the agent will not update its internal parameters when receiving a reward, but the learned behaviour is evaluated.

4.4.2 Interface Data Exchange: the Communication module

A communication module is required as interface for the data exchange. While the simulation is run in SPS using *SimTalk*, the agent will be modelled in python. Both softwares need to communicate to transfer information about the system configuration, current state, command and reward. The following use cases show the requirements of this software component.

Table 4.7: Use case: Writing information: ID-02-01

Use case	**Writing information: ID-02-01**
Description	Depending on the information exchanged, the required information need to be obtained and written in a uniform format by the simulation and the agent
Preconditions	• **1:** work plan is defined, production system is set up • **2:** at least one part could be moved to another machine • **3:** movement action is performed • **4:** current state has been received
Postconditions	-
Data	• **1:** configuration • **2:** state • **3:** reward • **4:** command – **a:** reset the simulation – **b:** move item – **c:** notice setup is done – **d:** notice training is done

Table 4.8: Use case: Reading information: ID-02-02

Use case	**Reading information: ID-02-02**
Description	The agent and the simulation have to be able to read information by the other component
Preconditions	A trigger for the creation of the information was obtained
Postconditions	the read information is accessible within the component
Trigger	for creation of the information

While a client would usually not be interested in implementation details, the developer needs to know on which trigger information is written and read. It is important to find a uniform format and have a clear mapping between *Simulation* ⇔ xml ⇔ *Dataexchanger* ⇔ *Agent*.

4.4.3 RL Agent

The RL agent is at the core of the software and is the component that will operate as controller of the production system. In order to realize this, it needs to be able to perform the steps of the training loop. The following use cases outline these. Note, that the internal algorithm will not be implemented by hand and is therefore is not listed here.

Table 4.9: Use case: Defining the action and state space: ID-03-01

Use case	**Defining the action and state space: ID-03-01**
Description	The agent shall learn from a finite action and state space, that needs to be defined depending on the configuration of the production system
Preconditions	configuration details of the simulation of the production system are given
Postconditions	the action and state space have a defined length
Trigger	-

Table 4.10: Use case: Calculating the reward: ID-03-02

Use case	**Calculating the reward: ID-03-02**
Description	The agent optimizes aiming for the actions that return the highest reward. It shall be determined according to the consecutive state of the simulation
Preconditions	performance of the action is finished
Postconditions	reward value is defined
Trigger	-

Table 4.11: Use case: Validity check of an action: ID-03-03

Use case	**Validity check of an action: ID-03-03**
Description	Only valid actions shall be executed by the simulation requiring a validity check
Preconditions	An action is choosen
Postconditions	The action is either defined as valid or invalid
Definition of valid actions	• the source from which a part shall be moved is occupied • the source contains a part matching the action's part type • the destination is not occupied • the action is a defined action as per work plan

Table 4.12: Use case: Choosing an action: ID-03-04

Use case	**Choosing an action: ID-03-04**
Description	The agent chooses an action to be performed by simulation depending on the internal learning algorithm of the agent
Preconditions	The current state is known
Postconditions	An action is chosen

4.5 Simulation Constraints

The simulation is limited by the scope of this work, even though it is most certainly interesting to consider material flow, optimize machine layout or include coordination of tool material and product transportation as well as other factors. Consequently it is important to mention the limitations of the prototype, in order to able to compare the agent with other algorithms. The following list gives an overview of the setup and its limitations. Note, that most of the limitations are similar to ones encountered in similar research work.

- There are two types of parts that can be manufactured, one requiring three processing steps, the other four. There are eight machines, that are used in different processing steps. One source and two drains are used
- There is no delay in the availability of material or tool
- Only one job order will be executed at one station at a time
- All processing times are deterministic and pre-selected but vary from machine to machine
- Machines do not breakdown and therefore do not require maintenance
- There is no transportation time between machines,hence the type of transportation and its coordination is not considered, so that subproblems like the allocation of transportation, route planning and collision control are excluded from the scope of this work
- The final version of the simulation does not include any intermediate buffer
- The production is not correlated to any external demand, production will start whenever the source is free
- Whenever a product is fully produced it is removed immediately without any delay

The resulting setup of the simulation is visualized in figure 4.3.

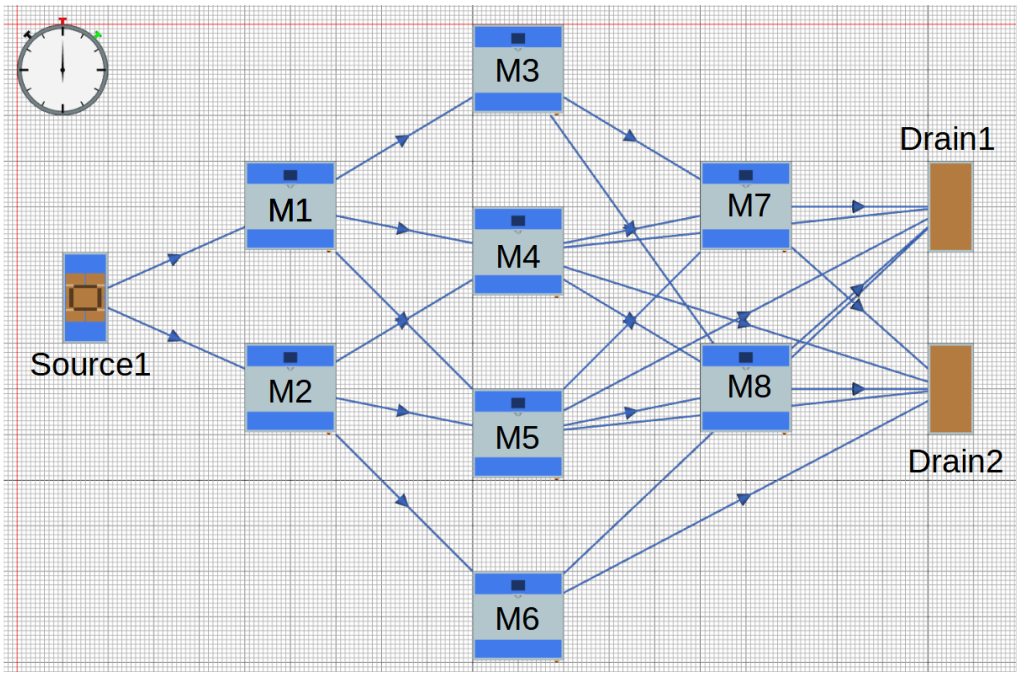

Figure 4.3: Setup of the simulation

Chapter 5

Design Specification

5.1 Introduction

Out of chapter 4.3 emerged the necessity for data exchange.The following section gives an overview over the procedural design, the data exchange and the architecture of the software.

The procedural design shows exemplary a training cycle. The evaluation cycle is very similar and for that reason not shown. Afterwards the goals for the architecture are explained and classes of the project shown.

5.2 Transferred information

The following subsection gives an overview of the encoding of the transferred information. There are five kinds of files, that serve different purposes as shown in table 5.1. The goal was to create a file structure, that is easily adaptable, so that the software has not to be changed significantly for other system configurations.

File	Information	Occurence
config.xml	setup of system, workplan, production times	Once before the start of the simulation
command.xml	either setup done, experiment done or move with parameters	repeatedly during simulation, answers to state.xml
state.xml	occupation, part type, remaining processing time	repeatedly during simulation, begin of each trajectory
reward.xml	reward properties, terminal	repeatedly during simulation, answers to action.xml

Table 5.1: Overview of used files

Configuration file includes the work plan for every part type. The following extract shows as an example that for PartA the second processing step (P2) includes the options to move the part from M1 to M3, from M1 to M4 or M1 to M5. Underneath M3, M4, M5 the different processing times are listed.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<config>
        <PartA type="string">
                [...]
                <P2 type="string">
                        <M1 type="string">
                                <M3 type="string">
                                        <procTime type="float">600</procTime>
                                </M3>
                                <M4 type="string">
                                        <procTime type="float">900</procTime>
                                </M4>
                                <M5 type="string">
                                        <procTime type="float">1200</procTime>
                                </M5>
                        </M1>
                </P2>
                [...]
        </PartA>
        [...]
</config>
```

Listing 1: extract of a configuration file

For the command file it is important to differentiate a command from an action. Actions are chosen by the agents and are behaviours that the agent displays in the environment. This can for example be the movement of a

part. A command in this case is a collective term and includes actions as well as other instructions, used for coordination of the two software components. It could for example be a COMMAND_EXPERIMENT_DONE command. An instance for a command file of the type *move* is shown in example 2. The required information is included in the type of command.

```
<?xml version="1.0" ?>
<command>
  <commandtype type="string">move</commandtype>
    <parttype type="string"PartA</parttype>
    <source type="string"M1</source>
    <destination type="string"M3</destination>
</command>
```

Listing 2: a command file, type *move*

The state file is written whenever a state requiring an action is reached, meaning the agent has to choose, where to go next, after the part has been processed. The chosen properties are listed underneath each station of the production system the chosen properties are listed. These are easy to extend in case the user wants to experiment with the available information of the state for the agent. Example 3 shows the the machine M1. Its properties are shown below.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<state>
        <machines>
                <M1>
                        <occupied type="boolean">false</occupied>
                        <parttype type="string">None</parttype>
                        <remainingproctime type="float">-1</remainingproctime>
                </M1>
                [...]
        </machines>
</state>
```

Listing 3: extract of a state file

The reward file may vary. The following instance includes information about the throughput per hours, as this was used to calculate a specific reward function. Every reward file needs to include the terminal parameter that indicates whether the current episode will be terminated because of the last chosen action (compare the sequence diagram 5.2). Every reward file also includes the passed time in seconds in simulation that may be used for later statistics.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<reward>
        <throughputperhour type="float">2</throughputperhour>
        <terminal type="boolean">false</terminal>
        <totalTime type="time">13920</totalTime>
</reward>
```

Listing 4: the reward file

5.3 Data Exchange

When the design of the agent was finalized and the production system was developed, the communication for the information exchange had to be created. The transferred information and its individual purposes are explained in more detail in chapter 5.2. The procedure of the training is explained in chapter 5.4.

The basic idea is that before the agent is learning, the production system communicates his set up, that includes for example the number of machines and which order each product is manufactured. When a processing step of an item is finalized, the part is ready to be moved the next position. Therefore the simulation writes a state file, so that the agent can choose an action that in turn can be read by the simulation. After performing the action the given reward is then communicated to the agent and the process starts again as soon as a part can be moved to the next position.

The communication between the two software components however proved to be difficult when using the student's license of SPS. The following subsection will show the resulting constraints. Afterwards the current solution is presented.

5.3.1 Options and Limitations

While Python does not have strong limitations when it comes to synchronization and inter-process communication, SPS does.
Per definition one of the goals of the project is, that it only use free of charge software. Having this constraint it is not possible to use one the offered interfaces for the Inter-Process Communication such as Open Database Connectivity (ODBC), Open Platform Communications Unified Architecture (OPC UA), SIMIT or the Socket interface, that are only accessible within a paid license. The student version only offers an Extensible Markup

Language (XML) interface for data transfer, which was therefore used. However this interface is usually used to load data into simulation at the beginning or storing it in a file at the end.
Since in this case it also used to read and write into files at the same time, synchronization is required. As far as the author is aware, Simtalk does not offer synchronization tools. It only offers waituntil-loops that interrupt methods until a certain condition within the production system is met.

5.3.2 Procedure of Data Exchange

As the author was not able to perform atomic actions and therefore synchronize certain parts of the software, it was not possible to guarantee expected behaviour in every single case.

However multiple precautions that have been taken to synchronize the software as best as possible. The procedure using the XML-interface resulting from the limitation is shown in figure 5.1.

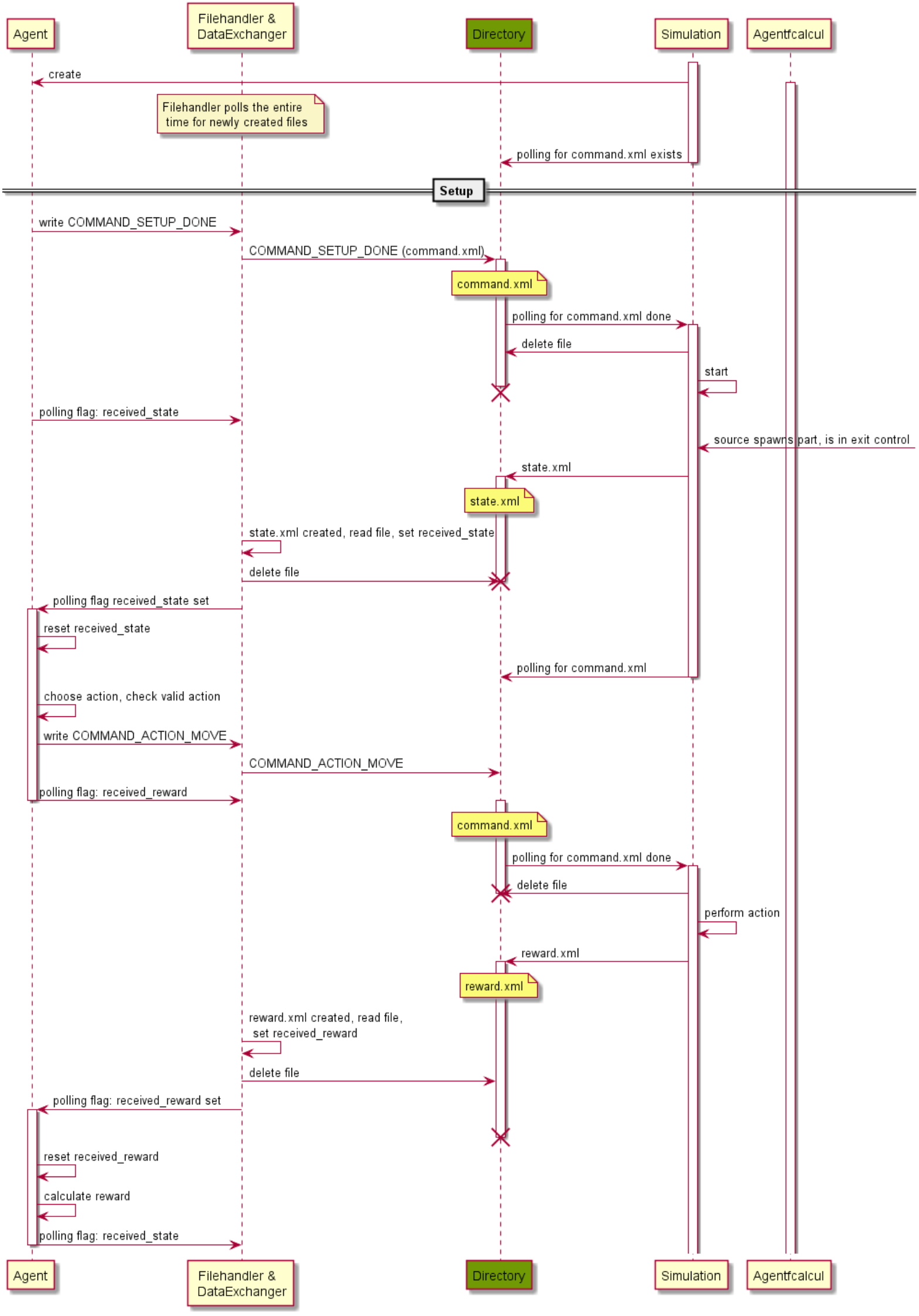
Agent
Filehandler & DataExchanger
Directory
Simulation
Agentfcalcul
create
Filehandler polls the entire time for newly created files
polling for command.xml exists
Setup
write COMMAND_SETUP_DONE
COMMAND_SETUP_DONE (command.xml)
command.xml
polling for command.xml done
delete file
start
polling flag: received_state
source spawns part, is in exit control
state.xml
state.xml
state.xml created, read file, set received_state
delete file
polling flag received_state set
reset received_state
polling for command.xml
choose action, check valid action
write COMMAND_ACTION_MOVE
COMMAND_ACTION_MOVE
polling flag: received_reward
command.xml
polling for command.xml done
delete file
perform action
reward.xml
reward.xml
reward.xml created, read file, set received_reward
delete file
polling flag: received_reward set
reset received_reward
calculate reward
polling flag: received_state
Agent
Filehandler & DataExchanger
Directory
Simulation
Agentfcalcul

Figure 5.1: Data exchange at the beginning of the software and one trajectory of a valid action; an inactive lifeline means, that the component is currently polling

The sequence diagram contains four participants, three of them being software components. Note that the participants have been partly merged to guarantee a better overview of the synchronization.

- **Agent**: the agent is a representative component for the rapper of the simulation on the python side, the main function and the agent.
- **Filehandler and Dataexchanger**: responsible for the communication
- **Simulation**: The production system modelled in SPS, works as training environment for the agent

Furthermore the communication directory depicted shows the existence of the files with their lifeline. The other lifeline of the objects show the times when they are active and not polling.
First of all it is to mention that the *Filehandler* observes the creation of files. Whenever a file is created, it triggers and reads the file and stores the included information.

The software is started by pressing the start button of the simulation. It then creates the agent. The agent starts polling for the existence of the command file. After finishing the setup of the agent and its internal components, it lets the *Dataexchanger* write a *command.xml* file containing the command *COMMAND_SETUP_DONE*. The *Agent* now stops polling, reads the file, stores the data and deletes the file afterwards. It then starts the simulation of the production system. Meanwhile the agent starts to poll for the flag *received_state*.

As soon as a part has finished the current processing step in a machine and reaches the exit of a machine, a *state.xml* file is written by the *Simulation*. The *Filehandler* notices the creation, reads the file, sets the *received_state* and finally deletes the file. The *Agent* now stops polling for the flag. It resets the flag and then chooses an action and checks its validity. In invalid scenarios no communication necessary. The sequence diagram shows the case, when an action is valid and therefore needs to be performed by the simulation. The agents prompts the *Dataexchanger* to write a *command.xml*

file with a *COMMAND_ACTION_MOVE* command. After the *Simulation* has written the state file it starts to poll for the existence of a *command.xml* file. Now that this file exists the polling is stopped,the file is deleted and the action performed. Afterwards the *Simulation* writes the *reward.xml* file for which the *Agent* started to poll for. The *Filehandler* notices the creation of the file, reads it, sets the *received_reward* flag and then deletes it. The *Agent* stops the poll, resets the flag and calculates the final reward for the action. Then the polling for the next state starts.

A new trajectory begins. After receiving the state the agent can choose which command to choose depending on the situation.

5.4 Procedural Design

After focusing on the synchronization of the data exchange, this section will outline the flow control of the software as well as the interaction of the components.

5.4.1 Flow Control

The flow control is visualized in figure 5.2.

The sequence diagram contains the following four participants:

- **Agent**: agent is a component that represents the internal agent logic, as the algorithm that chooses an action
- **Main**: this software component runs the training and evaluation loop
- **Filehandler and Dataexchanger**: are the components, that are reponsible for the data exchange
- **Simulation**: offers the agent the environment to train in

When the simulation is started, it writes the config file, based on the current setup of the simulated production system. Afterwards it starts the python script, running the agent. Simulation then waits until it receives the XML-file containing the information that the setup is done. During this setup the python script creates the required objects. This includes the agent, the python environment and the *Filehandler*, tracking the creation of XML-files. Whenever the *Filehandler* receives a file matching its filter, the

Dataexchanger updates its values.

The configuration file is read and on the bases of its content the action space for the agent gets calculated.The training loop can start. Within the training loop the act-execute-observe pattern is used. In each episode the iteration is repeated as long as the simulation has not sent a termination notice with the reward-file.

In the beginning a XML-file containing the COMMAND_SETUP_DONE information is written to the simulation, which then resets and finally starts the simulation. Items spawn at the source of the production system and triggers the exit control when reaching the exit of the machine. The simulation then notes the current state in a XML-file and starts waiting for an action to perform. Meanwhile the training loop waits until the current state is defined. When this is the case, the internal function *act()* of the agent is called, choosing an action based on the type of agent. The action gets executed. Two different scenarios are possible at this point.

If the action is invalid, there will be no correspondence with the simulation. The reward will be calculate directly, state and terminal remain the same. The terminal variable indicates whether a state is a final state of the process. The reason for not sending the invalid action is to prevent unnecessary data exchange. Also the state does not change to the exchange of this file would be redundant. As the steps within an iteration will be performed very often, cutting down useless computation seemed sensible.
If the action is valid this interaction becomes necessary. The chosen action will be transmitted to the simulation and executed. The reward and terminal variable as well as the new state will be returned. Now the simulation continues and polls for a new incoming command.

Finally the agent method *observe()* checks the reward and the terminal variable. If terminal variable is set to *true* the episode ends. a COMMAND_RESET will be transferred to the simulation leading to a reset of the simulation for the next episode containing another trajectory $S_n, A_n, R_{n+1}, S_{n+1}$....

At the end of the training process the simulation receives a COMMAND_TRAINING_DONE and stops the simulation. The environment and agent then will be closed.

The evaluation process is considered an individual project, so that the user does not necessarily have to perform it. The evaluation is very similar to the training process the only difference being, that the agent does not update from the last episodes.

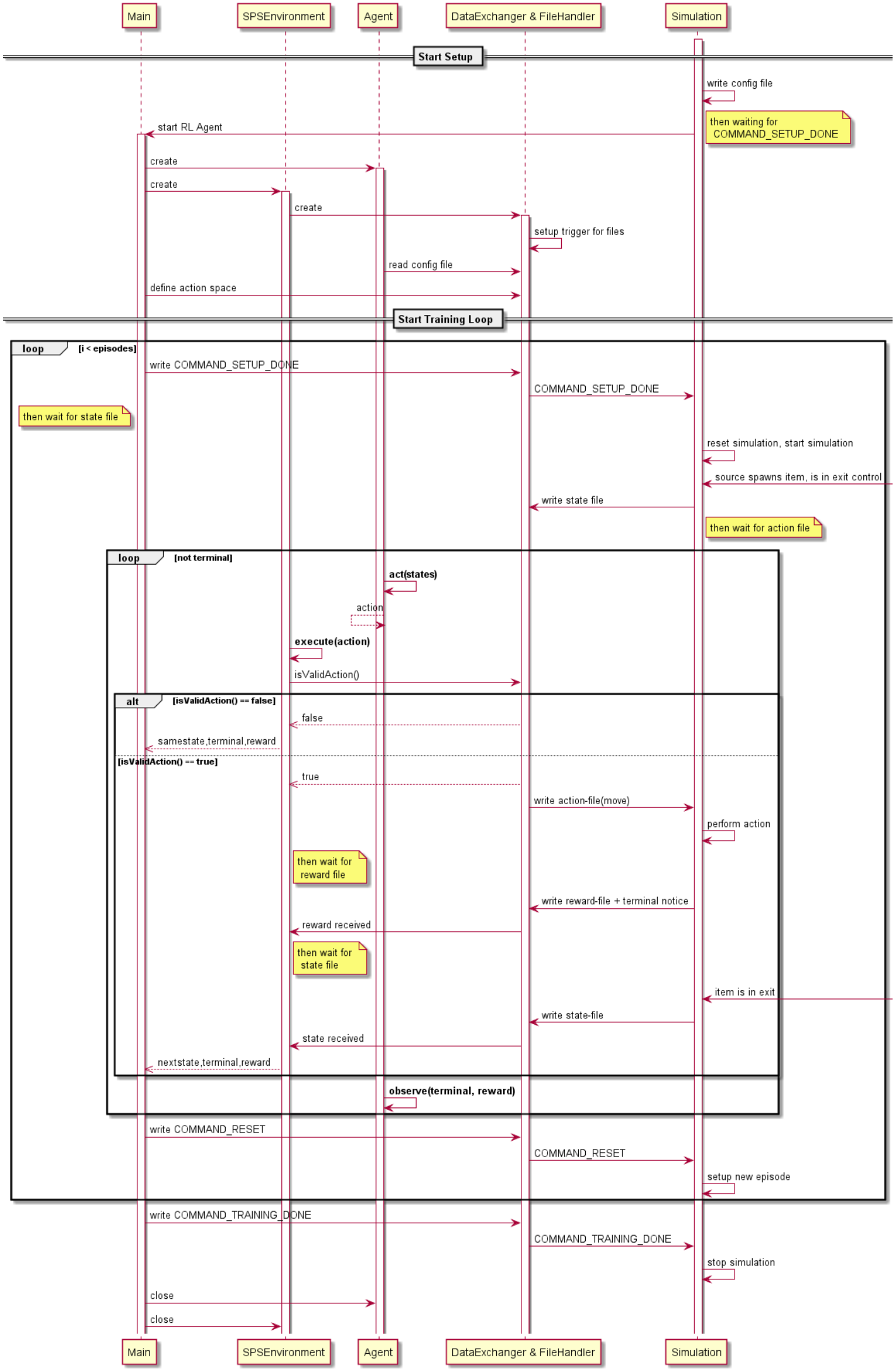
Main
SPSEnvironment
Agent
DataExchanger & FileHandler
Simulation
Start Setup
write config file
then waiting for COMMAND_SETUP_DONE
start RL Agent
create
create
create
setup trigger for files
read config file
define action space
Start Training Loop
loop [i < episodes]
write COMMAND_SETUP_DONE
COMMAND_SETUP_DONE
then wait for state file
reset simulation, start simulation
source spawns item, is in exit control
write state file
then wait for action file
loop [not terminal]
act(states)
action
execute(action)
isValidAction()
alt [isValidAction() == false]
false
samestate,terminal,reward
[isValidAction() == true]
true
write action-file(move)
perform action
then wait for reward file
write reward-file + terminal notice
reward received
then wait for state file
item is in exit
write state-file
state received
nextstate,terminal,reward
observe(terminal, reward)
write COMMAND_RESET
COMMAND_RESET
setup new episode
write COMMAND_TRAINING_DONE
COMMAND_TRAINING_DONE
stop simulation
close
close
Main
SPSEnvironment
Agent
DataExchanger & FileHandler
Simulation

Figure 5.2: Sequence diagram of the flow control

5.5 Architecture

This section gives an overview of the aspects relevant for the software architecture design. An overview of the software is given and extended by a class diagram. An entity-relation diagram was not included, as each object of the software only existed once.

The main focus of the architecture is to keep it as modular as possible. The communication between the software should be easily exchangeable. Therefore the reading and writing functions were encapsulated.

It should be easy to exchange the reward function, the included state information, the action type or the agent. Therefore the methods calculating the rewards deciding which actions are considered to be valid and which state information is used, were kept modular.

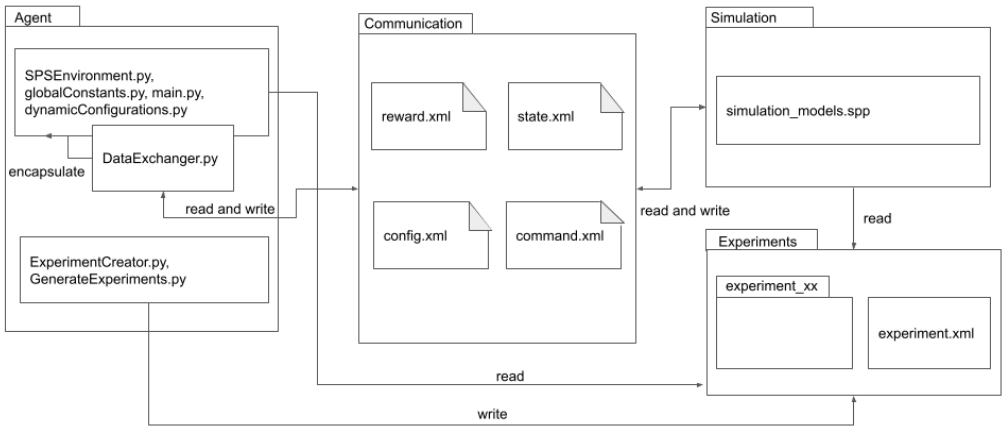

Figure 5.3: Interaction of the different software components

Figure 5.3 shows the interaction of the different software components. The four main folders including software parts are the agent and the simulation,

as well as a folder for communication and experiments. The experiments can be configured in the *ExperimentCreator*. By running *GenerateExperiments*, the experiments will be created in the experiments folder. From this folder the environment can read the different experiments, adjust the settings accordingly and start the agent with the current experiment identifier, creating the according configuration on the agent's side.

During the training and evaluation process, environment and agent do not interact directly. The agent has an environment wrapper, internally using the functionalities of the *DataExchanger*, managing communication on the agent's side. The simulation also has functions encapsulating the way to communicate. Both sides in the end exchange information over the communication folder by reading and writting the XML-files explained in chapter 5.2.

After performing the experiments, results can be taken and the set of experiments replaced with new ones.

The following class diagrams (figures 5.5 and 5.4) give a more detailed overview of the internal methods and variables

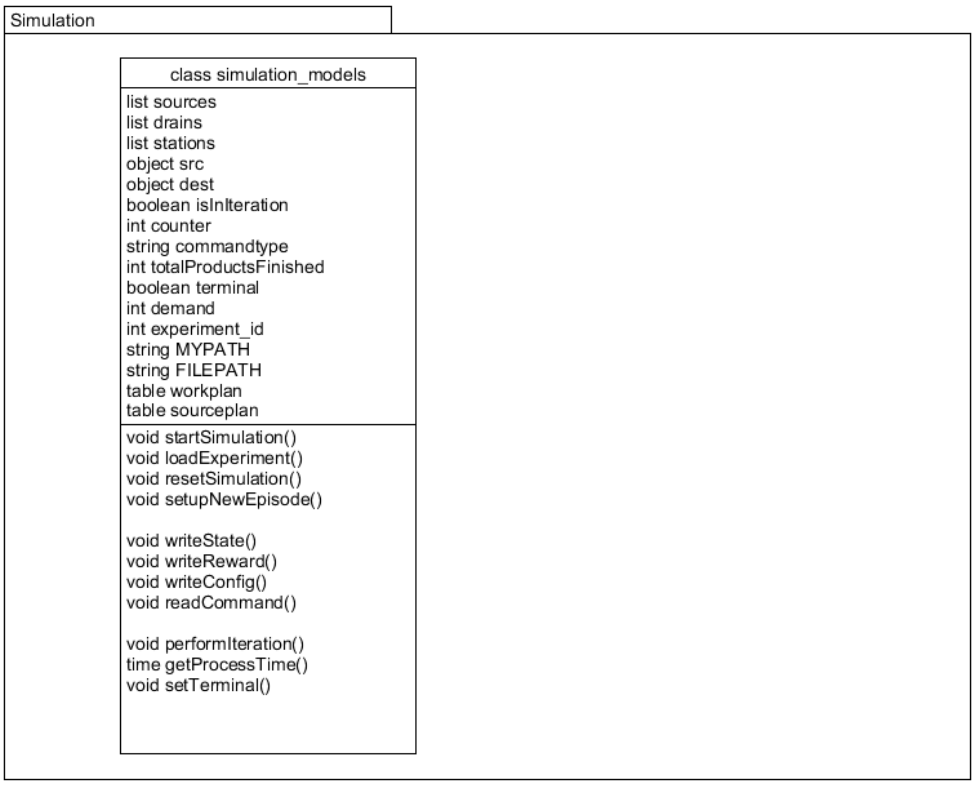

Figure 5.4: Overview of the simulation functions and variables

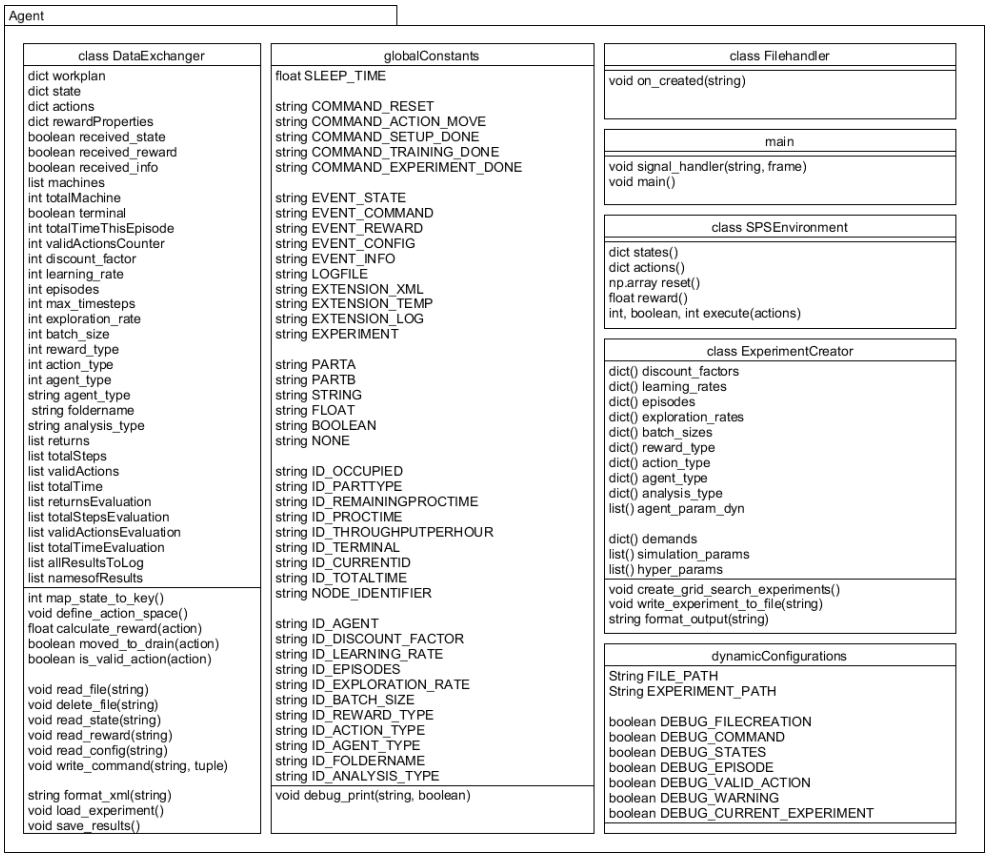

Figure 5.5: Overview of agents classes

Chapter 6

Technical Programming Documentation

6.1 Introduction

This section is structured as a guide to install and setup the developed software. For usage of the software the reader may refer to the user manual (compare chapter 7). For a better guidance an overview of the repository is given.

6.2 Directory Structure

Figure 5 shows the directory structure of the project, consisting of three main folder: the *src* folder (including all source files), the *data* folder (providing all raw data of the experiments for traceability and transparency) and the *documentation* folder.

The *src* folder contains four subfolders. The *simulation* folder includes the SPS simulation and a required batch file used for file handling. The *agent* folder includes all agent related source files as well as the configuration files for the experiment setup and general dynamic configurations. The *analysis* folder contains the script for analysing and plotting the results. *Communication* file is an empty folder, to be used for software communication. The *experiments* folder holds the currently configured experiments. It includes the *experiment.xml*, containing the list of the chosen experiments. Each experiment has its own folder that will be filled during the training process

with the trained agent, the configurations and the obtained results.

All performed experiments have been collected and stored in the *data* folder.

```
.
|
+---data
|   +---experiments_training_duration_1
|   |        dem-100-df-0.99-lr-0.001-eps-100-expl-0.001-bat-10--mayer-act-1-ppo
|   |   |       agent-0.data-00000-of-00001
|   |   |       agent-0.index
|   |   |       agent.json
|   |   |       checkpoint
|   |   |       results.txt
|   |   |         dem-100-df-0.99-lr-0.001-eps-100-expl-0.001-bat-10--mayer-act-1-ppo.xml
|   +---[... next experiment run]
|   |   |
+---documentation
|   +---pdfs
|   |   |   report.pdf
|   |   |   annex_A.pdf
|   |   |   annex_B.pdf
|   +---texfiles
|   |   |   [...all texfiles]
+---src
|   +---simulation
|   |      simulation_models.spp
|   |      simulation_models.spp.bak
|   |      filehandler.bat
|   |
|   +---agent
|   |      main.py
|   |      SPSEnvironemnt.py
|   |      globalConstants.py
|   |      DataExchanger.py
|   |      ExperimentCreator.py
|   |      GenerateExperiments.py
|   |      dynamicConfigurations.py
|   +---analysis
|   |       plotter.ipynb
|   +---experiments
|   |   |   experiment.xml
|   |   |   dem-100-df-0.99-lr-0.001-eps-100-expl-0-bat-1-mayer_+_10000_-5-act-1-ppo
|   |   |   dem-100-df-0.99-lr-0.001-eps-100-expl-0.01-bat-1-mayer_+_10000_-5-act-1-ppo
|   |   |
|   |   +---dem-100-df-0.99-lr-0.001-eps-100-expl-0.001-bat-10--mayer-act-1-ppo
|   |   |       agent-0.data-00000-of-00001
|   |   |       agent-0.index
|   |   |       agent.json
|   |   |       checkpoint
|   |   |       results.txt
|   |   |        dem-100-df-0.99-lr-0.001-eps-100-expl-0.001-bat-10--mayer-act-1-ppo.xml
|   |   |
|   |   +---[... next experiment file]
|
.
```

Listing 5: Directory structure

6.3 User Requirements

This software is only executable for the Windows operating system. In order to use the software, the user must install Siemens Plant Simulation 16. The minimal system requirements are the following [6]:

Operating System	Windows 8.1/10
Memory (RAM)	4GB
Hard Disk Space	2GB
Processor	x86-64 Processor

Table 6.1: Minimal System Requirements for SPS

As the software performs a lot of writing and reading commands as well as a computational heavily algorithm, it is recommended to use a computer with at least a 8GB RAM.

6.4 Program Installation

The installation is split into two parts. Firstly it is explained how to install and setup SPS. Afterwards the required python libraries and setups are described.

It is necessary to download the repository from Github. The easiest way to do so, is clicking on the green "code" button and selecting "download ZIP". Unzip the downloaded repository.

6.4.1 Siemens Plant Simulation

For this project the Student license of Siemens Plant Simulation was used. The user can obtain the software on this website: `https://www.plm.automation.siemens.com/plmapp/education/plant-simulation/en_us/free-software/student`. Registration is required. After downloading the software, follow the installation wizard.

After installing the software, load the simulation model *simulation_models.spp* from the *.\src\simulation* folder by double click. Please refer to chapter 6.2, if you need clarification about the location of mentioned files. When the model is loaded, go to the tab *Windows*, and make sure the Console is

activated.

Next configure the variables *MYPATH* and *FILEPATH*.
They are located on the right side close to the start button of the model. Double click the variable you want to edit. Change *MYPATH* to the path, that leads to *.\src*. For example the path could look as follows, depending on the chosen location of the project.

```
MYPATH = C:\matrix-optimization-master\src
```

The *FILEPATH* has to be changed to lead towards the predefined folder *.\src\communication*

```
FILEPATH = C:\matrix-optimization-master\src\communication
```

6.4.2 Python

Executing the entire software requires python version 3.9.. Check which python library is installed try by running the following command.

python –version

If the user does not have this installed yet, they may refer to this installation guide `https://docs.python-guide.org/starting/installation/#installation`.
It is important to disable the path length limit of 260 characters. Also make sure, to add the PATH variable when using the installation wizard.

After installing python, *pip*, a tool, can be installed to simplify installation of the python libraries.

```
py -m pip -version
```

If it is missing, install it using this reference: `https://packaging.python.org/en/latest/tutorials/installing-packages/`.

Finally you can install missing libraries using the following commands.

```
 pip install watchdog
pip install tensorforce
```

After the installation is done, open the file *dynamicConfigurations.py*. It is located in *.\src\agent*. Make sure to change *FILE_PATH* and *EXPERIMENT_PATH*. The experiment path may point to the experiment folder *.\src\experiments* and the *FILE_PATH* may refer to the folder for communication as set in the simulation. Note, that it is essential to set the *FILE_PATH* towards the same folder as for the *FILEPATH* in SPS. Make sure to change the path according to the following syntax. The final backslashes are required.

```
FILE_PATH = r"C:\\matrix-optimization-master\\src
\\communication\\"
```

```
EXPERIMENT_PATH = r"C:\\matrix-optimization-master\\src
\\experiments \\"
```

Optionally you may choose which debug prints shall be logged in the logfile by setting them to *True*. Changing these options is not mandatory and are set off by default. If running large experiments the resulting logging file can be very large depending on the chosen logging settings. The debug settings are displayed in table 6.2.

DEBUG_FILECREATION	logs which files are created
DEBUG_COMMAND	logs which command type is written
DEBUG_STATES	logs each state that is exchanged
DEBUG_EPISODE	logs each new episode
DEBUG_VALID_ACTION	logs which action is chosen, and whether it is valid
DEBUG_WARNING	logs when files are tried to be accessed at the same time
DEBUG_CURRENT_EXPERIMENT	logs the current experiment configurations

Table 6.2: Logging options

The repository comes with an exemplary experiment. The setup up of individual experiments is explained in chapter 7.

The setup is now finished. You may start the software by clicking the start button on the right hand side of the model in the simulation software. Note, that SPS very quickly start its own debugger. That is not a problem, it only happens when the software is waiting for an answer longer than expected. You do not have to open or close the debugger, just let the computation continue. You press enter once in the command window of the current experiment, to have this window back in the foreground.

If a run of experiments is interrupted, the system can be reset by setting the *experiment_id* to zero or to experiment identifier that shall be run next. Note, that the training has always to be performed before running and evaluation cycle.

When the author tried installing the software on different devices, sometimes a *tensorflow.python.framework.error_impl.*
AlreadyExistsError: Another metric with the same name already exists error appeared at the start. This happens, when the internally used *keras* library does not match the *tensorflow* version used by the *tensorforce* library. To solve this problem adapt the *keras* library by using the following command.

```
pip install keras == 2.6.*
```

6.5 Testing

In comparison to other software projects, this project did not use a test first approach or unit and integration tests performed during and after the implementation phase of the software project.

In this case only two aspects could be tested.
One is the representation of data, that has been done manually by analyzing the files.
The other and most important one is communication, as this is time critical, direct testing was not possible, during the implementation of the time critical aspects. A number of stress tests have been performed, during running the software to check its stability.

Chapter 7

User Manual

7.1 Introduction

The following chapter gives an overview on how to run the software and individualize experiments.

7.2 Setup of a New Experiment Set

First of all make sure you followed all steps of this installation guide carefully, before changing the experiment setup. Try out the example experiment first to see, whether everything is set up correctly.

Then remove the current content of the *.\src\experiments* folder. Afterwards open the *ExperimentCreator.py* script located in the *.\src\agent* folder. In the constructor, you find a variety of values that can be customized. The following table 7.1 shows the parameters, their datatypes and their ranges.

Parameter	**Datatype**	**Range**
Discount factor	float	[0,1]
Learning rate	float	(0,1]
Episode	float	[1,∞^{*}]
Exploration rate	float	[0,1]
Batch size	int	[1,∞^{*1}]
Reward type	string	{"mayer","mayer_-_100_-5" ,"mayer_-_10000_-5", } "mayer_-_10000_-25"
Agent type	string	{"ppo","ddqn"}
Demand	int	[1,∞^{*2}]

Table 7.1: List of adjustable parameters.
*increases the computation time immensely, 200 episodes take around 4 hours to compute for the training alone, maximum test is 400,
*1 the batch size has to be smaller than the amount of episodes,
*2 the demand also drastically increases the computation time, the recommended value is 100, maximum tested is 300

Save the changes and execute the script *GenerateExperiments.py*, which is located in the same folder. This will generate a variety of experiments in form of folders as well as an *experiment.xml* file, containing a list of all experiments in the experiment folder. To check, if it was done correctly, check the *.\src\experiments* folder for the content.
The setup is finished. You may start the software by clicking the Startbutton on the right side of the prototype in simulation software.

If a invalid value is found during the execution due to for example a spelling mistake, you do not have to repeat the entire list of experiments. Locate the experiment, where you want to start again in *experiment.xml* and note the *ID*. Go back to SPS and change the *experiment_id* variable, that is located to the right side of the start button to the chosen *ID*. Click on the start button to continue.

Bibliography

[1] Berufsstart. https://www.berufsstart.de/gehalt/werkstudent/informatik.php. Last accessed: 12-01-2022.

[2] GitHub. https://github.com. Last accessed: 12-01-2022.

[3] Notion Labs. https://www.notion.so/. Last accessed: 12-01-2022.

[4] Stefanie Schilling. Abschreibung computer afa 2021 verkürzt. https://www.wir-beraten-zukunft.de/abschreibung-computer-afa-2021-verkuerzt, 2021. Last accessed: 12-01-2022.

[5] Siemens. Akademische lizenzen. https://plant-simulation.de/produktueberblick/lizenzarten/akademische-lizenzen/. Last accessed: 12-01-2022.

[6] Siemens. Minimale pc konfiguration. https://plant-simulation.de/produktueberblick/systemanforderungen/. Last accessed: 12-01-2022.

[7] ZenHub. https://www.zenhub.com/. Last accessed: 12-01-2022.