# Discrete-event simulations with $\chi Py$

Bachelors Thesis

**J. LANGEDIJK**

**COORDINATOR: A.A.J. LEFEBER**

Department of Mechanical Engineering

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

# Discrete-Event Simulations with $\chi Py$

Jelle Langedijk

SE: DC2021.062
June 2021

Cover and back image taken from the *TU/e Manufacturing System Engineering* promotional clip, hosted on *The Mute Zone* (TMZ).

# PREFACE

This thesis concludes my three-year bachelor *Mechanical Engineering* at the *Eindhoven University of Technology*. I enjoyed every moment of my bachelor studies in which I could broaden my general engineering skills and find my future destination: the *Dynamics and Control* group, where I am hopefully able to continue my studies.

First, I would like to thank Assistant Professor A.A.J. Lefeber for his excellent guidance during the entire project. While not much guidance was required, the tips and additional motivation he gave helped me a lot during the development of $\chi Py$ and the writing of this thesis.

Aside of that, I would like to thank all my friends for the continuous support and interest in the project, as well as the for "off-time" I could spend with them to clear my mind.

Finally, I would like to show appreciation to my mother for her support during my bachelor studies. Without her, I would not be in the position that I am now.

Jelle Langedijk

Eindhoven, June 23, 2021

# SUMMARY

$\chi$3 is the third iteration of the $\chi$ formalism, developed by the former *Systems Engineering Group* at the *Eindhoven University of Technology* (TU/e). It is a formalism written to develop discrete-event simulations (DES) for any desired application. While $\chi$3 is proven to be a successful DES formalism, it does not come without its issues. The main issue is that its syntax is nothing like other common modeling languages like *MATLAB* and *Python*. This means that, at introduction to the formalism in the *Analysis of Production Systems* course, students have to learn a whole new language with a steep learning curve, just to be able to write basic DES models. This is rather tedious for an introduction to DES. Therefore, this thesis proposes an alternative named $\chi Py$ that is accessible for students who just get started with DES as well as for more experienced engineers to efficiently model a plethora of production lines and other discrete-event systems.

$\chi Py$ provides its users with a set of modeling and post processing tools to quickly write and run discrete-event simulations in *Python*. Most of these tools are build around an *environment* that keeps track of time and a *resource* that is used to model order queuing and processing. Both are based on the already existing *Python* discrete-event package *SimPy*. The provided tools are workstations (with support for priority handling, finite buffers, batching and machine failure with repairmen), generators, a simulation caller and post processing functions for data tables and visualisations.

All these tools are tested and compared to existing cases readily modeled using the $\chi$ formalism. Tests were concluded favourably, showing little to no deviation between the results of both languages. The comparison with the *Intel Five-Machine Six Step Mini-Fab* however brought to light that certain specific requirements of building blocks (like specific batching policies for workstations) are harder to model in $\chi Py$: it requires class and/or method inheritance and a proper understanding of the $\chi Py$ source code to implement these requirements properly.

All with all, $\chi Py$ is a good alternative to $\chi$3 and its ancestors. It allows for easier and simpler modeling of surfeit discrete event systems, like a production line, store check-out or hospital. Compared to the $\chi$ formalism, $\chi Py$ also has the advantage of build-in post processing tools to quickly validate written models.

# SYMBOL INDEX

## Table of symbols

Since in this thesis time can have any unit (like *second*, *minute*, *hour*), only a symbol overview with corresponding variable definition is given.

Table 1: Table of symbols

| **Symbol** | **Variable** |
|---|---|
| $c$ | Coefficient of variation |
| $m_f$ | Mean time to failure |
| $m_r$ | Mean time to repair |
| $r_a$ | Rate of arrival |
| $t_{\text{entry}}$ | Entry time in run environment |
| $t_{\text{exit}}$ | Entry time in run environment |
| $t_e$ | Process time |
| $t_{\text{fail}}$ | Time of failure |
| $t_{\text{fix}}$ | Time of repair |
| $t_{\text{in}}$ | Entry time in process |
| $t_{\text{now}}$ | Current time |
| $t_{\text{out}}$ | Exit time in process |
| $t_r$ | Run time |
| $w$ | Work in progress (WIP) |
| $\bar{w}$ | Mean work in progress |
| $\beta$ | Exponential scale parameter |
| $\lambda$ | Exponential rate parameter |
| $\varphi$ | Flow time |
| $\bar{\varphi}$ | Mean flow time |
| $\sigma$ | Standard deviation |

# CONTENTS

**Preface** **i**

**Summary** **iii**

**Symbol index** **v**

**1 Introduction** **3**
1.1 Current state of discrete-event modeling: $\chi 3$ . . . 3
1.2 The issues with $\chi 3$ . . . 3
1.3 $\chi Py$: A discrete-event simulation toolbox for Python . . . 3
1.4 Outline . . . 4

**2 The fundamentals of $\chi Py$** **5**
2.1 Environment . . . 5
2.2 Resource . . . 6

**3 $\chi Py$: Front-End** **9**
3.1 Simulation building blocks . . . 9
3.1.1 Station . . . 9
3.1.2 Repairman . . . 10
3.1.3 Run Environment . . . 10
3.1.4 Generator . . . 10
3.2 Simulating . . . 11
3.3 Data post processing . . . 11
3.3.1 Simulation summary console output . . . 11
3.3.2 Lots flow time diagram . . . 12
3.3.3 Station-specific performance plot . . . 12
3.4 Modeling conventions in $\chi Py$: basic model structure . . . 12

**4 $\chi Py$: Back-End** **15**
4.1 Resource . . . 15
4.2 Simulation building blocks . . . 16
4.2.1 Station . . . 16
4.2.2 Repairman . . . 17
4.2.3 Generator . . . 17
4.3 Simulating . . . 18
4.4 Data post processing . . . 18
4.4.1 Simulation summary console output . . . 18
4.4.2 Lots flow time diagram . . . 18
4.4.3 Station-specific performance plot . . . 19
4.5 Package initiation . . . 20

**5 Testing $\chi Py$** **21**
5.1 Modeling and simulation of an autonomous vehicle storage and retrieval system (AVS/RS) . . . 21
5.1.1 Case summary . . . 21
5.1.2 $\chi Py$ model structure . . . 22
5.1.3 Tested subjects . . . 22
5.2 Modeling and simulation of an Intel Five-Machine Six Step Mini-Fab . . . 23
5.2.1 Case summary . . . 23
5.2.2 Tested subjects . . . 24

**6 Conclusion and Recommendations** **25**
6.1 Conclusion . . . 25
6.2 Recommendations . . . 25

**Bibliography** **27**

**Appendices** **29**

**A Example $\chi Py$ model** **31**
A.1 The model . . . 31
A.2 Console output . . . 33
A.3 Visual output . . . 34

**B $\chi Py$ Source Code** **37**
B.1 $\chi Py$ hierarchy . . . 37
B.2 Source code listings . . . 38
B.2.1 Package Initiation . . . 38
B.2.2 Core . . . 38
B.2.3 Processes . . . 41
B.2.4 Functions . . . 46

**C Case studied with $\chi Py$** **53**
C.1 $\chi Py$ Notebook model for *" Modeling and simulation of an autonomous vehicle storage and retrieval system"* . . . 53

# CHAPTER 1

# INTRODUCTION

This first chapter of this thesis introduces the case at hand and introduces the developed solution presented in the remaining chapters of this thesis. The introduction concludes with a document outline.

## 1.1 Current state of discrete-event modeling: $\chi 3$

$\chi 3$ is the third iteration of the $\chi$ formalism, developed by the former *Systems Engineering Group* at the *Eindhoven University of Technology* (TU/e) [1]. It is a formalism written to develop discrete-event simulations (DES) for any desired application [2]. Its fundamentals are based around process interaction with synchronising channels. Each process has its own local variables, meaning that processes cannot read data from each other. The only way of communicating thus happens via the synchronising channels. The formalism is strong-typed, meaning that the type of a variable is determined at its initiation and cannot be changed during simulation. $\chi 3$ supports common statements like `for`, `while` and `if` to model iterators and conditions. Finally, a set of discrete and continuous distributions are provided to sample values from to add stochasticity inside models.

$\chi 3$ (and earlier versions) have been proven to be very capable for DES modeling in the past. It has been used for discrete-event modeling at multiple Masters Thesis cases at the TU/e, like the modeling and analysis of an Intel Five-Machine Six Step Mini-Fab and efficiency determination of a hospital's emergency department [3] [**?**]. Aside of theses, the language is also used in the third year bachelor course *Analysis of Production Systems* to give students an idea of DES [4].

## 1.2 The issues with $\chi 3$

While $\chi 3$ is proven to be a successful DES formalism, it does not come without its issues. The main issue is that its syntax is nothing like other common modeling languages like *MATLAB* and *Python*. This means that, at introduction to the formalism in the *Analysis of Production Systems* course, students have to learn a whole new language with a steep learning curve, just to be able to write basic DES models. Furthermore, most Mechanical Engineering students have little to no programming skills (aside of numerical modelling in *MATLAB* and data processing in *Python*). Finally, $\chi 3$ is only used within the Eindhoven University of Technology, making it for no use after graduating.

Needless to say at this point, there is demand for an alternative that is characterised by a more gentle learning curve and is syntax-wise very comparable to current programming languages, making it more appealing for a wider audience.

## 1.3 $\chi Py$: A discrete-event simulation toolbox for Python

This thesis proposes this solution as $\chi Py$, a *Python* DES toolbox. $\chi Py$ is based on *SimPy*, a discrete-event package for *Python* [5]. In contrast to $\chi$, *SimPy* works with the concept of resource occupation to model events and processes.

The development of *SimPy* started in 2002 by Klaus Müller, Tony Vignaux and Chang Chui as a merge between *Simula 67* and *Simscript* [6]. Throughout the years, *SimPy*'s team changed to the current team of maintainers

(Ontje Lünsdorf and Stefan Scherfke) starting at version `1.9` and went through multiple revisions that mainly focused on simplifying the toolbox [7]. The current version (at the point of writing), `4.0.1`, therefore lacks functionality *Simpy 1* had, like a graphical interface and plotting tools. Its syntax is however quite a bit easier to understand, making *SimPy 4* more beginner friendly relative to its ancestors.

While *SimPy* came a long way since 2002, it lacks out-of-the-box functionalities like machine failure, finite buffers for processes, data plotting functionality and different buffer policies (*SimPy* defaults to FIFO with no other option). Lastly, *SimPy* is made to work in conjunction with *Object Oriented Programming* (OOP), which most beginners have presumably no experience with.

$\chi Py$'s goal is to build upon the work of Ontje Lünsdorf and Stefan Scherfke by extending *SimPy*'s functionality and presenting it in a fool-proof form. This makes it accessible for students who just get started with DES as well as for more experienced engineers to efficiently model a plethora of production lines and other discrete-event systems.

## 1.4 Outline

In the remainder, first the fundamentals of $\chi Py$ are explained. Next, the toolbox is approached from two essential perspectives: the *Front-End* and *Back-End*. The front-end is the part of the code the user interacts with, like callable functions. The back-end is the part the user does not necessarily needs to interact with; the code behind the callable functions. Once all aspects of the toolbox are presented, $\chi Py$ is tested and compared to two real-life examples in chapter 5. This thesis concludes with a conclusion and set of recommendations, based on the current state of the software.

# CHAPTER 2

# THE FUNDAMENTALS OF $\chi PY$

The core of $\chi Py$, consisting of its fundamental components, pivots on the *SimPy Python* package. The fundamental components are an *environment* that keeps track of time and a *resource* that servers as a combination of a server and queue.

## 2.1 Environment

The environment is the space where all processes take place, that keeps track of its local time. The environment $\chi Py$ utilises is an exact copy of a *SimPy* environment.

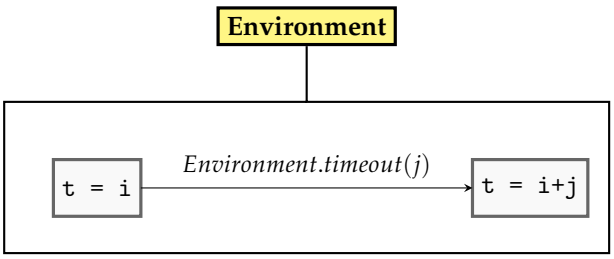


Figure 2.1: Time passing in an enviroment

Discrete time steps are taken by yielding timeouts, bounded to an environment, as depicted in Figure 2.1. The current time in the environment is `Environment.now`. Aside of timeouts, environments can also have processes, which can be seen as *Python* methods that yield timeouts. A process can be executed using `env.run(process)`, which runs that specific process until it terminates. One can also simulate until `Environment.now` reaches a prescribed value `t` using `env.run(until=t)` or to simulate until all processes are finished by calling `Environment.run` without arguments.

A simple example utilising an environment can be seen in the code block below. A machine `machine` processes four products. If the ID of a product is even, its process time is 1 time unit, and 2 time units otherwise. The entry and exit times of each product are printed as console output.

**A simple machine**

```
import chipy as cp
env = cp.environment()
def machine(env, ID) -> None:
    print(f"Product {ID} enters at time {env.now}")
    if ((ID % 2) == 0):
        yield env.timeout(1)
    else:
        yield env.timeout(2)
```

```
9       print(f"Product {ID} exits at time {env.now}")
10  def run_machine(env) -> None:
11      for ID in [1,2,3,4]:
12          yield env.process(machine(env, ID))
13  def main():
14      env.run(env.process(run_machine(env)))
15  if __name__ == "__main__":
16      main()
```

## 2.2 Resource

*χPy* handles most processes using resources, see Figure 2.2. The resource *χPy* incorporates is an advanced version of a *SimPy* Priority Resource.

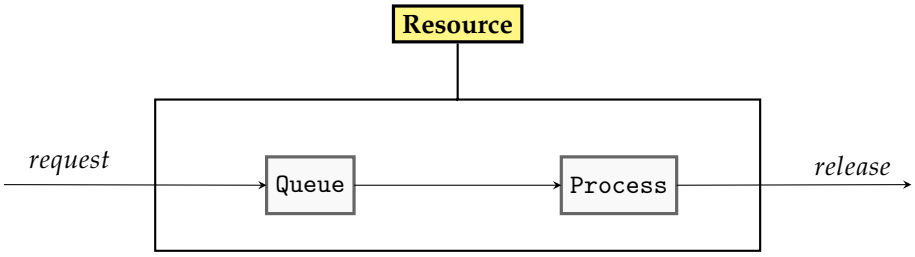


Figure 2.2: A resource

A resource can be seen as a server with a limited capacity $n$, that can serve (`Process`) $n$ requests at the same time. Once a request occupies a spot inside the resource, this spot is occupied until the resource is done serving the request. At that point, the request is released. If all $n$ spots in the resource are occupied, a new entering request cannot be served until a currently processed request is released. It is therefore placed inside a `Queue`, waiting till the request is accepted and starts processing.

A well-known real world example is a store check-out. The used resource in this case is the cashier serving customers waiting to pay, one by one ($n = 1$). The currently served customer is in the `Process` part of the resource, while in the meantime all other queuing customers are waiting inside the `Queue`.

The resource *χPy* uses to implement resources is the Priority Resource (`chipy.core.resource.PriorResource`).

**Priority Resource**

```
1 cp.PriorResource(env, cap, queuelen, queuepol)
```

The `chipy.PriorResource` class is based on the default *SimPy* `PriorityResource`, but extends its functionality with a limited queue length (`queuelen`) and two different queue policies (`queuepol`): **F**irst **I**n **F**irst **O**ut (FIFO) and **L**ast **I**n **F**irst **O**ut (LIFO). A resource needs to know in which environment it is active, making the `env` argument mandatory. The capacity `cap` defaults to 1, the queue length to ∞ and its policy to FIFO.

Initiating this class just returns an object with desired properties and does nothing until it is called. Calling a resource requires a specific syntax to ensure that the release action happens automatically. Instead of simply calling it, it is *requested* with a `with` statement as shown below.

**Requesting a Resource**

```
1 res = cp.PriorResource(env = env)
2 with res.request() as request:
3     yield request
```

A resource's release action starts once the end of the `with` statement is reached. This implies that a resource's `Process` is defined *outside* the resource. This gives the end user complete freedom in modeling a resource's process. For example, a single cashier ($n = 1 \rightarrow$ `cap=1`) helping a customer at checkout for `5` time units can be modeled using the following `with` statement:

**A cashier serving a customer**

```
import chipy as cp
env = cp.environment()
cashier = cp.PriorResource(env = env, capacity = 1)
def checkout(env, ID) -> None:
    print(f"Customer {ID} arrives at the checkout at time {env.now}")
    with cashier.request() as request:
        yield request
        yield env.timeout(5)
    print(f"Customer {ID} checked out at time {env.now}")
def run_checkout(env) -> None:
    for ID in [1,2,3,4]:
        yield env.process(checkout(env, ID))
def main():
    env.run(env.process(run_checkout(env)))
if __name__ == "__main__":
    main()
```

A request action can be expanded with additional arguments. If a request is more urgent than another, it is automatically served first, still depending on the chosen policy! A priority level is indicated with a number. The lower the number, the higher the priority of the request. More advanced is the `us` argument, which has the type `chipy.processes.station.Station`. This is used for finite queue control which is touched upon in subsection 3.1.1.

Now the core elements of $\chi Py$ are elucidated, they can be utilised in simulation building blocks. In the next two chapters, these building blocks, as well as additional functions of the $\chi Py$ package, are discussed. Two perspectives are provided: the front-end and the back-end. In the next chapter, the frond-end is explained in detail.

# CHAPTER 3

# $\chi PY$: FRONT-END

The front-end is the part of the code the end-user interacts with, in this case a simulation script and its components. The end-user does not need to understand what happens "behind the scenes" (the back-end, see chapter 4), but needs to be able to work correctly with all the building blocks $\chi Py$ provides, mostly based on the fundamentals discussed in the previous chapter. This chapter discussed these separate building blocks as well as their interactions with each other, simulation options and post processing functions. Each class and method treated here supports *Python*'s `help()` method to display all information and arguments supported.

## 3.1 Simulation building blocks

$\chi Py$ provides the end-user with a set of building blocks to model production lines.

### 3.1.1 Station

A `chipy.processes.station.Station` is a class that represents a workstation including its buffer. A workstation can be used to model any kind of process, like a car wash washing cars, a machine assembling parts or a doctor treating a patient.

**Station**

```
1 cp.Station(env = env, te = lambda: 1, cap = 1, queuesize = inf, queuepol = "FIFO", mf =
      10, mr = 1, repairman = repairman, batchsize = 1 batchpol = "hard", data_collection
      = True)
```

A stations main property is its process time $t_e$, defined as a lambda function. The process time defines how long a process should take which can be, due to the fact it is a lambda function, any desired value. It can be a constant (deterministic), but also the output of any function sample, like an exponential distribution.

Aside of the mandatory arguments, a station has multiple other properties which can be modified. A station's *capacity* (`cap`, default = 1) determines the amount of parallel processes a station can handle, like two parallel machines in a workstation, or a car wash which has place for multiple cars. Buffers have a fixed buffer capacity (`queuesize`, default = $\infty$) and handles waiting orders according to a FIFO or LIFO policy (`queuepol`, default = "FIFO"). Station breakdowns can be controlled with the time to failure $m_f$ (`m_f`, default = $\infty$) and time for repair $m_r$ (`m_r`, default = $\infty$) values. These times cycle continuously according to Equation 3.1 where $i$ is the failure index.

$$t_{f,i} = \begin{cases} m_f, & \text{for } i \in \{1\} \\ m_f + (i-1)(m_f + m_r) & \text{for } i \in \mathbb{N} \setminus \{0,1\} \end{cases} \tag{3.1}$$

This can be extended with a repairman (`repairman`, default = `None`), which is given the task to repair the machine in $m_f$ time. When a repairman is already busy at the point of request, the current repair request has to wait (according to a FIFO or LIFO policy) until the repairman has time to process the request.

Instead of processing per lot, a station can also process lots in batches of size `batch` (defaults to 1, no batching). A batch size can either be a hard condition (no smaller batches allowed) or a soft condition (smaller batches also allowed). The latter condition is often used in furnaces. The furnace is filled with all orders waiting in the queue, with a maximum value. Condition types are parsed through the `batchpol` argument.

Finally, data collection of a station can be turned on/off with the boolean `data_collection`, which defaults to `True`.

Since a station is essentially an advanced resource, a lot has to place a request at the station. Placing a request is done using a station's `proc` method, that takes care of requesting a station to process the lot, as well as processing the lot itself.

**A lot requesting a station**

```
1 yield env.process(station.proc(lot = lot, us = station2, te = lambda: te))
```

Only the `lot` argument is mandatory, since it needs something to process. Optional arguments are the `us` argument to parse the `chipy.Station` object of the upcoming station to avoid queue overflow in the case of finite buffers (Default is `"NaN"`) , and the lot specific process time `te` if desired (Default is `None`).

### 3.1.2 Repairman

The `chipy.processes.repairman.Repairman` class models a repairman (or multiple repairmen with the same properties) that are linked to one or multiple stations to repair a station once it fails.

**Repairman**

```
1 cp.Repairman(env = env, cap = 1, pol = "FIFO")
```

Since a station communicates the $m_f$ value to a repairman automatically, its process time does not have to be specified. It is only necessary to parse the Environment to a repairman to initiate one properly. There are however, just like with the station, optional arguments to parse to a repairman at its initiation.

Just like a station, a repairman has a capacity (`cap`) and policy (`pol`) since it is a resource. In the specific case of the repairman, its capacity can be interpreted as the amount of repairmen available of its type. The policy is the order in which a repairman serves repair requests from stations, either FIFO or LIFO.

### 3.1.3 Run Environment

The production line itself is called the run environment. It links all stations together in a certain order and (if desired) keeps track of entry and exit times of lots. Run environments are not predefined in $\chi Py$, but have to be manually written by the end-user. This is done to give the end-user complete freedom in chaining stations based on the simulation requirements.

**Run Environment**

```
1 lots: list = []
2 def runenv(env, lot, stations):
3     yield env.process(stations["Station 1"].proc(lot = lot))
4     lots.append(lot)
```

Since a run environment is written by the end-user, its arguments are also (mostly) free to choose. For proper integration with the generator (see next subsection) and to allow to call station processes, parsing the environment and a lot going through the production line is mandatory. Stations can be directly initiated within the run environment, but it is common practise in $\chi Py$ to initiate them outside of the run environment.

For proper generator control (see next subsection), it is mandatory to keep track of the processed lots. This is generally done by storing each lot in a global list once it exits the production line.

### 3.1.4 Generator

The final core building block is the generator. A `chipy.processes.Generator` generates lots and sends them in the assigned run environment.

**Generator**

```
cp.Generator(env = env, ta = ta, runenv = runenv, lots = lots, stations = stations,
    priority = lambda: 0)
```

The mandatory arguments for a generator are self-evidently the Environment (`env`), the inverse of rate of arrival (`ta`), the run environment (`runenv`) to send lots to and a list of processed lots so the generator can stop generating once the desired amount of lots are produced. Finally, the end-user is required to parse a dictionary containing all workstations through the `station` argument. This way, the generator can initiate each station's individual failing procedure and sent all stations to the run environment.

The only optional argument is lot priority in the form of a lambda function (`priority`, default `lambda: 1`), such that any desirable method can be used to determine lot priority.

## 3.2 Simulating

*χPy* has a separate method to call and execute a simulation with desired settings: `chipy.functions.simulate.simulate`.

**Calling a Simulation**

```
cp.simulate(env = env, generator = G, method = "Time", sim_time = t)
```

This method support three types of simulations: "*Time*", "*n-processed*" and "*n-generated*". Each simulation method has its own distinct set of arguments to define its parameters. This is presented, alongside method descriptions, in Table 3.1.

Table 3.1: Three simulation options of *χPy*

| Simulation Method | Description | Arguments |
|---|---|---|
| Time | Simulate until time `t` is reached | `env, generator, method = "Time", sim_time =t` |
| *n*-processed | Simulate until *n* orders are processed, while continuing the generator process | `env, generator, method = "n-processed", lots_max = n` |
| *n*-generated | Simulate until *n* orders are generated and processed. The generator stops after n orders are generated | `env, generator, method = "n-generated", lots_max = n` |

If a custom way of simulating is desired, *SimPy*'s `env.run()` method can also be used, but requires manual setup from the end user.

## 3.3 Data post processing

Aside of simulating, *χPy* can also post process simulation data with the help of `pandas` and `matplotlib`.

### 3.3.1 Simulation summary console output

The `chipy.functions.prints.stats.stats` method generates a simulation station summary including a table of individual lot data. The summary is displayed as output in the console, and the generated table is saved to `./data/phi_data.csv`.

**Simulation Summary**

```
cp.stats(lots = lots, t_start = t_arr, t_end = t_exit, sim_time = t, savename = 'phi\
    _data.csv', generate_table = True)
```

All arguments are compulsory, except the boolean `generate_table` that controls whether the method prints and exports the lot data table (defaults to `True`) and the string `savename` (default = `'phi_data.csv'`) to define the name of the table export. The `stats` method requires the list of lots (which the run environment keeps

track of as seen earlier) to check which lots reached the end of the run environment. The list of all lot entry and exit times (`t_start` and `t_exit`) are used to determine the flow time $\varphi$, population standard deviation of the flow time $\sigma$ and coefficient of variation of the flow time $c$. The simulation time (`sim_time`) is used to print the simulation time.

### 3.3.2 Lots flow time diagram

The resulting table from the `stats` method can be visualised using `chipy.functions.prints.visualisations.flowtimediagram.flowtime_diagram`. This method also supports (automatic) `.svg` to `.pdf_tex` conversion using *Inkscape* on operating systems build around the Linux kernel.

**Flowtime Diagram**

```
1 cp.flowtime_diagram(lots = lots, t_start =  t_arr, t_end = t_exit, sim_time = t,
      savename = "flowtime.png", title = "Flowtime of lots", xlabel = "Time", ylabel = "
      Lot ID", color = "lightblue")
```

All obligatory arguments are the same as for the `stats` method. `flowtime_diagram` allows further tuning of visual aspects, like the title (`title`, default = `"Flowtime of lots"`) and the labels of both axis (`xlabel`, default = `"Time"` and `ylabel`, default = `"Lot ID"`). Finally, the color of the flow time blocks can be changed in any color supported by `matplotlib`.

Titles and axis labels can be changed to any (f-) string, meaning that model variables can be directly inserted in these plot elements. LaTeX 2ε formatting is also supported.

### 3.3.3 Station-specific performance plot

χ*Py*'s last post processing tool is the `chipy.functions.prints.visualisations.stationstats.station_stats` method. This function generates a four-panel plot containing periods of failure and the evolution of the buffer size, occupancy rate and work in progress (WIP) $w$ of a single station.

**Station Statistics**

```
1  cp.station_stats(stations = stations, station_name = key, sim_time = t, show_avg =
      False, savename = "stats.png", title = "Statistics of ", xlabel = "Time", color = "
      lightblue")
```

The method requires the dictionary with all stations (`stations`), the dictionary key of the desired station to make a plot of (`station_name`) and the simulation time (`sim_time`).

This four-panel also allows the same settings to tweak the plot with one addition: `show_avg` (defaults to `False`). When `True`, a red line indicating the average value of the buffer size, occupancy rate and WIP is shown in their respective plots.

## 3.4 Modeling conventions in χ*Py*: basic model structure

The building blocks of χ*Py* are written with a certain convention in mind. While end-users can define themselves how they want to connect the building blocks, certain specific guidelines are required to make a model work properly.

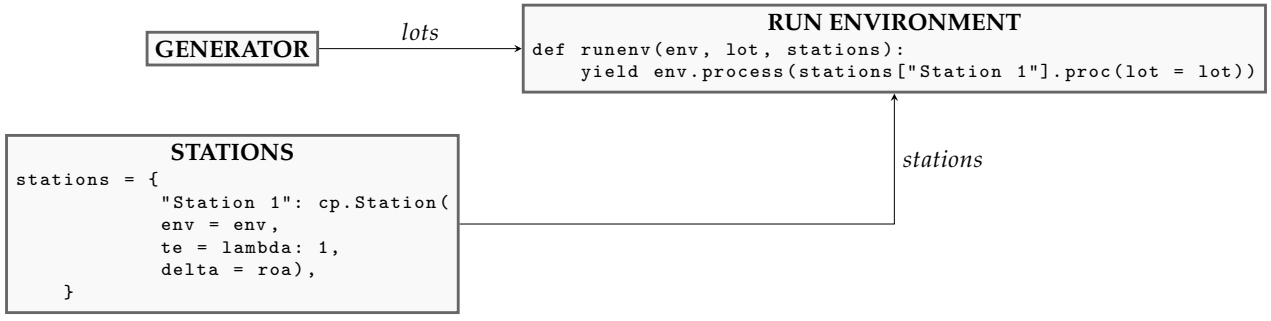


Figure 3.1: General model structure

The general structure of a χ*Py* model is depicted in Figure 3.1. It clearly displays the central role of the run

environment as a bridge between a generator and consecutive stations, as well as a medium for lots to travel through. These relations are key to understand when modeling in $\chi Py$.

At the beginning of a script, each required package is loaded into the Python kernel. Similar to $\chi$, it is common practise to define all system variables on a global level at the top of the script, after the imports. While *Python* does not require strong typing values (while $\chi$ does), it is still recommended to increase the readability of the written code. Next, all stations are initiated in a dictionary. If desired, repairmen can be linked to the stations. After all stations are added to the model, the run environment can be written, followed by the generator. The script concludes with a `main` method that calls the `simulate` method and (if desired) post processing tools. When the script is called using the `python` command, the `main` method is the code that gets executed. An example script can be seen in Appendix A that follows all described conventions. The model this script describes illustrates that, when following the convention of Figure 3.1, a functional simulation can be written swiftly. The results have also been manually verified in Appendix A.2 and A.3, concluding the validness of the individual components, as well as the modeling convention.

This chapter discussed all functions $\chi Py$ provides a user to write a DES model. The next chapter builds upon this discussion by providing the mechanics and modeling strategies of these functions. This is called the back-end, which is necessary to fully understand when applying the package to real-life cases in chapter 5.

# CHAPTER 4

# $\chi PY$: BACK-END

The back-end is the part of the code the user does not interact with, in essence what happens "behind the scenes". The code of each class and method discussed in chapter 2 and chapter 3 (excluding the environment) is explained in detail in this chapter. Appendix B contains the full source code of $\chi Py$.

## 4.1 Resource

A resource is a combination of `Queue` and `Process`, as discussed in chapter 2.

#### Request

The request in a `chipy.PriorResource` works exactly the same as in *SimPy*: it initiates a `request` object which is sent to the `queue` where it waits until it can be processed. An addition to the *SimPy* `request` is the parsing of the upcoming `chipy.Station` for the release controller through the `us` variable. Since this value is linked to a unique `request` object, this can be retrieved once the request reaches the release phase.

#### Queue

After a request is placed, it lands in the `queue` of the resource. This queue is a *Python* list that is sorted after each list append based on the policy of the queue (FIFO or LIFO) and the priority of the request. The list is first sorted in subsets per priority level. Then, each subset containing orders with the same priority level is sorted based on the predefined policy of the queue.

#### Batching

*$\chi Py$ always* batches orders before sending them in the `Process`. In the case no batching is desired, $\chi Py$ will make batches of size 1. Batch forming happens in between the `Queue` and `Process`. Each `chipy.PriorRequest` object is appended to a batch list. Once the list has the desired size, all requests are individually granted instantaneously and start with the `Process` phase. The desired size is based on the chosen batching policy. This policy is either hard (no smaller batches allowed) or soft (smaller batches also allowed).

#### Release

As discussed in chapter 2, a $\chi Py$ resource automatically releases an order after it is processed. This can however cause issues when the upcoming resource's queue is full: the queue will overflow, which should be avoided anywise.

Therefore, a resource has a release controller added on top of *SimPy*'s default release action. This controller checks if an order release results in queue overflow at the next resource. Figure 4.1 displays the general case of two resources, $n$ and $n+1$, chained in series where resource $n$ has a release controller to safeguard the queue size of resource $n+1$. Safeguarding is done in the form of a hold time delay $t_{h,n}$. The hold time is the smallest time until a queue spot is freed in the queue of resource $n+1$. Due to the fundamentals of the *SimPy* release action, the determination of $t_{h,n}$ is done only once, at the point when an order starts its release. Since there is a (likely) chance that resource $n+1$ will fail during the hold time $t_{h,n}$, the controller should also check whether this is the case or not. This is done based on the failure and repair history of resource $n+1$. Since fail behavior

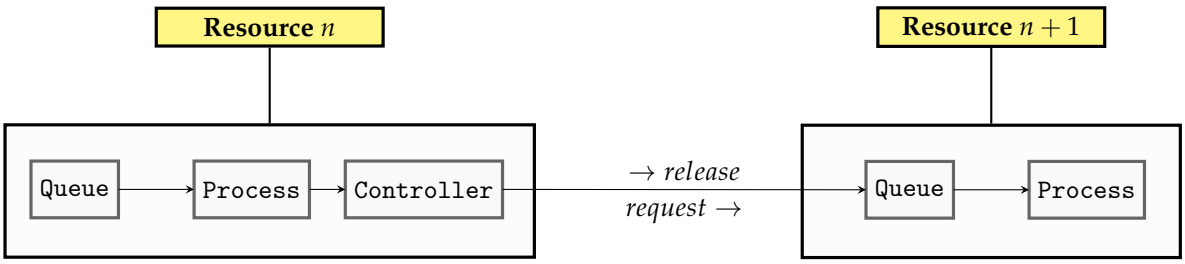


Figure 4.1: A release controller

happens on the scope of the `chipy.Station` class and not inside the resource itself (note that this is not even possible), the resource controller is only engaged when the station corresponding to resource $n+1$ is parsed as a `chipy.Station` through the `us` argument. Both cases are summarised in Equation 4.1, where $t_r$ is the run time of the current process (which is further explained in subsection 4.2.1), $t_{\text{now}}$ is the current time in the environment, $t_{\text{in}}$ is the start time of the current process $m_{r,n+1}$ is the repair time including possible waiting for an available repairman.

$$t_{h,n} = \begin{cases} \min(t_{\text{in},n+1} + t_{r,n+1} - t_{\text{now}}), & \text{when resource } n+1 \text{ will not fail within the hold time} \\ \min(t_{\text{in},n+1} + t_{r,n+1} - t_{\text{now}}) + m_{r,n+1}, & \text{when resource } n+1 \text{ will fail within the hold time} \end{cases} \tag{4.1}$$

In the case of batching, each order in a batch is treated as a separate release in FIFO, meaning that the first order that entered the batch from the queue prior to processing gets the first release. This corresponds with the way a batch of requests gets instantly granted individually.

As mentioned earlier, in the case of the presence of a repairman in resource $n+1$, the value of $m_{r,n+1}$ depends on the amount of repairmen there are available of the type linked to resource $n+1$, and how much repair requests there are in the queue. In the current version of $\chi Py$, the controller does not incorporate a general expression of relating these dependencies yet, so this will likely cause issues during simulation.

## 4.2 Simulation building blocks

Simulation building blocks are production line elements, like a station, that can be used to build a model.

### 4.2.1 Station

In essence, a station is a pre-programmed resource with a complex `Process` part. This means that its resource is just a simple `chipy.PriorResource`, as discussed in section 4.1.

#### Process time determination

There are three different cases to be considered when determining the run time $t_r$ of an order in a station. The run time is the *total time* an order spends in the `Process` step in a resource (recall Figure 2.2), which does not necessarily have to be $t_e$. The default case is when a machine is not broken: the run time equals the process time. When a machine is broken, the order will enter the machine, but the `Process` starts only when the machine is repaired. The final case involves process interruption: a machine fails while processing a order. In that case, the process is resumed once the machine has been repaired.

$$t_r = \begin{cases} t_e, & \text{when a station is not broken} \\ t_e + m_r - (t_{\text{now}} - t_{\text{fail}}), & \text{when an order enters a broken station} \\ t_e + m_r - (t_{\text{fail}} - t_{\text{in}}), & \text{when an order is interrupted by a failure} \end{cases} \tag{4.2}$$

The time step is implemented as a time out of $t_r$ time units in the corresponding environment.

#### Fail Method

The fail method handles the fail behavior of a station, initiated by a generator (see subsection 4.2.3).

First, the method yields a timeout of $m_f$ to progress to the point of failure. Once this point is reached, a station's state becomes `broken` (used for case 2 in Equation 4.2) and checks whether current `Processes` are alive to interrupt (case 3 in Equation 4.2). Next, the repair starts, which either yields a timeout of $m_r$ or requests a repairman to repair the machine for $m_r$ time units. After completing the repair, the broken state is reverted to `False`, and the cycle starts over again.

Fail cycling is based on a `while` loop with variable condition depending on the chosen simulation method (see section 4.3). If a simulation is called for a predefined duration or predefined amount of processed orders, the condition is a binary `while True` loop, otherwise the cycle ends when a predefined amount of orders is processed to avoid livelock.

#### Statistics collection

When `data_collection = True` at a station's initiation, it collects station statistics that can be visualised using the `chipy.stationstats` method. The measured statistics are:

- *Fail Periods*, by measuring the time of failure and the time of repair completion;
- *Buffer Size*, by measuring the amount of lots waiting inside a station's buffer;
- *Occupancy Rate*, defined as $\frac{\text{Active Processes}}{\text{Total Capacity}} \cdot 100\%$, by measuring the current amount of active parallel processes in a station;
- *Work In Progress w*, defined as the amount of lots, by measuring the amount of lots in the process phase of the respective station. Its average at the $i$-th measurement, $\bar{w}_i$, is calculated at the point of collection according to Equation 4.3, wherein $t_i$ is the time at the $i$-th measurement [8].

$$\bar{w}_i = \begin{cases} 0, & \text{for } i \in 0 \\ \frac{t_{i-1}}{t_i} \cdot \bar{w}_{i-1} + \frac{t_i - t_{i-1}}{t_i} \cdot w_{i-1} & \text{for } i \in \mathbb{N} \setminus \{0\} \end{cases} \tag{4.3}$$

$\chi Py$ collects statistics at the point of state mutation. For example, the occupancy rate of a stations process is measured when a lot enters the station's process as well as when it leaves the process. All measurements are two dimensional: both the time of measurement and the measured value are logged as a single measurement. The only exception are the fail behavior statistics, since the time of measurement equals the measured value. Subsection 4.4.3 discussed the processing of these data points into a visualisation.

### 4.2.2 Repairman

A `chipy.Repairman` is, just like a station, a pre-programmed resource.

#### Repair time determination

In contrast to the `chipy.Station`, the run time of a repairman is constant and determined by the $m_r$ parameter of the station it has to repair. No additional cases have to be distinguished, since the Repairman class does not implement breaks or end of shifts out of the box (which is equivalent to machine failure modeling-wise).

### 4.2.3 Generator

A generator is a special kind of iterator that generates orders on the fly and sends them inside a user-defined run environment.

#### Order generation

A `chipy.Generator` generates orders in the form of lots, which are tuples containing the lot ID (count starts at zero) and the lot priority. The lot ID is simply increased with one per lot generation and the priority level is determined by the (user) provided priority distribution. After each lot iteration, a timeout of $t_a$ is yielded in the environment the generator is bounded to.

The iterative nature of lot generation is provided by a variable `while` loop with context based condition (also seen at the `chipy.Station.fail` method). This context is in the case of the generation method the type of simulation provided by the `chipy.simulate` method, as presented in Table 4.1.

Table 4.1: The three simulation options of $\chi Py$

| Simulation Method | Condition Description | Condition |
|---|---|---|
| Time | Generator process is force stopped by `env.run(until=t)` | `lambda: 1` |
| *n*-processed | A generator process stops once *n* lots reached the end of the run environment | `lambda: (len(self.lots) < lots_max)` |
| *n*-generated | A generator process stops once *n* lots are generated | `lambda: (lot_ID < lots_max)` |

Note that in the third method, only the generator method terminates. The simulation continues until all lots are processed!

#### Station fail method initiation

The generator also takes care of initiating the fail behavior of all stations in a run environment. Since the generator requires the dictionary of stations as input argument to call the run environment each time a lot is initiated, it made the most sense to loop over all stations and call their fail method. This is done based on the simulation method provided by `chipy.simulate` (see section 4.3) to ensure that the correct variable `while` loop condition is used in each station's fail method (see subsection 4.2.1).

## 4.3 Simulating

The simulation starter `chipy.simulate` is a method which calls an environments `run` method with the proper arguments that fit the desired simulation method of the user.

#### Time

When the simulation method is *Time*, the generator process is called and its bounded environment runs till the simulation time is reached with `env.run`.

#### *n*-processed

When the simulation method is *n-processed*, the generator process is called and its bounded environment runs until the $n$ lots are processed by the run environment. Since the generator's variable `while` loop condition stops generating once this point is reached, the simulation is called using `env.run(generator.gen(...))` to ensure to stop simulating once the `generator.gen(...)` event terminates.

#### *n*-generated

When an environment is simulated using the *n-generated* method, the simulation should continue after the generator finished until the run environment is extinct, meaning that no processes are pending. This point corresponds with $n$ lots being processed, since no new lots get generated. This is achieved by initiating the generator process and running the bound environment until it is extinct using `env.run()` (argumentless).

## 4.4 Data post processing

*χPy* has three data post processing methods, as discussed in chapter 3.

### 4.4.1 Simulation summary console output

The `chipy.stats` method takes care of printing a simulation summary in the console.

#### Data sorting

The statistics printer requires a list of lot-linked entry and exit times to determine the flow time $\varphi$ of each lot, as well as its mean $\bar{\varphi}$, standard deviation $\sigma$ and coefficient of variation $c$. These lists are processed by a sorting function to extract which lots went fully through the run environment as well as their entry and exit times. From this data, $\varphi$ can be calculated for each lot. The resulting data is printed in the console using f-prints.

#### Table generation and export

The calculated data from the sorting function is inserted in a `pandas DataFrame` object that allows for easy formatting and printing. The dataframe has three columns: *lot ID*, *Flowtime* and *Entry Time*. The dataframe is by default exported as a `.csv` file (that excludes the column headers) in the `./data` directory (which is automatically made if it does not exist) to allow external post processing in for example *Excel*, *OriginLab* or *MATLAB*. Aside of exporting, the data is also shown in the console output by converting the dataframe to a printable string.

### 4.4.2 Lots flow time diagram

The `chipy.flowtime_diagram` method generates a flow time diagram using `matplotlib` displaying the time lots spend inside a run environment.

#### Data sorting

Since the flow time diagram needs essentially the same data as the simulation summary, the used sorting function is identical.

#### Plot generation

The flow time plot is a horizontal broken bar plot generated using `matplotlib.pyplot`. Each lot has its own bar with starting x-coordinate the entry time in the run environment $t_{\text{entry}}$ and width the total time spent in the environment $t_{\text{exit}} - t_{\text{entry}}$. These bars are "stacked up" each other, meaning that the latest lot is the top most bar. The y-axis ticks are assembled by extracting the lot ID's from the exit times array, which means that the order of lot stacking is always the order of exiting the run environment.

#### `.svg` to `pdf_tex` conversion

As shown in section 3.3, all post processing functions support exporting to custom filenames and formats. In the case of saving the flow time diagram as a scalable vector graphic, the image will be converted to a `pdf_tex` file due to LaTeX $2_\varepsilon$ lack of support for `.svg` importing. The conversion works in conjunction with with *Inkscape* on Linux-based operating systems. $\chi Py$ checks whether the used operating system meets these requirements. If not, the image will not be converted. A confirmation will be printed if a flow time diagram is generated and saved successfully, including a confirmation of `.pdf_tex` conversion if applicable.

### 4.4.3 Station-specific performance plot

The `chipy.station_stats` method generates a $2 \times 2$ diagram using `matplotlib` displaying the periods of failure as well as the evolution of the *queue size*, *occupancy rate* and *work in progress* over time. Similar to the flow time diagram scalable vector graphics are converted to `pdf_tex`.

#### Data sorting

Again, the sorting function is very comparable to the previous two post processing functions, but tailored to the specific way certain data is monitored, as depicted in subsection 4.2.1.

#### Failure periods

Failure periods are presented as a horizontal broken bar plot. Each bar presents one failure with starting $x$-coordinate $t_{\text{fail}}$ and width $t_{\text{fix}} - t_{\text{fail}}$. All periods are plotted at the same $y$-coordinate, hence are side by side distributed over the time span of the simulation time ($x$-axis).

#### *Queue size*, *occupancy rate* and *work in progress*

The the last three statistics are all implemented as a step plot. A measured statistic is represented by a horizontal line. At the time of a statistic value mutation, the line "steps" from its old value to the new one with a vertical line as visualised in Figure 4.2 on the next page.

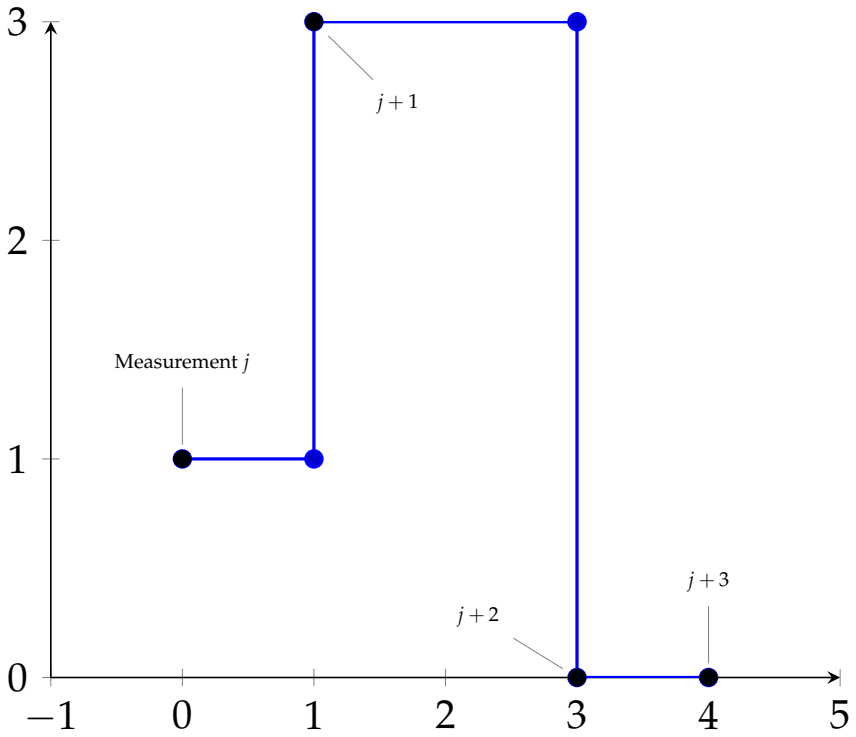


Figure 4.2: A step plot with four points of measurement

Once the last mutation is reached, a new horizontal line will not be drawn till the end of the figure, since this will continue forever, hence the fourth measurement $j + 3$ in Figure 4.2 is required to draw the last line. This point is, however, not always measured. If a simulation for example runs till $t = 63$, the station does not necessarily measure all statistics at this point in time, since it only measures at points of mutation. $\chi Py$ fixes this by checking whether the time of the bounded environment's internal clock at the point of simulation termination is included in the measurement. When this is not the case, an additional measurement point is added at the end of simulation time with the value of the previous measurement.

When `show_average = True`, the method generates a red line indicating the average value of the buffer size, occupancy rate and $w$ statistics over time. In the case of the buffer size and occupancy rate this average is the mean of all $N$ measured values $\frac{\sum_{i=1}^{N} x_i}{N}$. The average line of $w$ is calculated according to Equation 4.3. Note that in contrast to the other two statistics, this average is cumulative and evolves over time.

## 4.5 Package initiation

To avoid importing classes and methods with their full name (for example `chipy.functions.visualisation.stationstats`), $\chi Py$ has an init method located at the root of the package. The method imports all the classes and methods such that they can be used by their name, without including the exact directory path to the function. Since $\chi Py$ is not distributed through a package manager like `pip` or `conda` it has to be manually placed inside a project directory. Therefore, the init method also gives the user a confirmation telling if the package is imported correctly and which version of $\chi Py$ is imported.

Now the source code of $\chi Py$ is explored in more detail, it is time to apply all knowledge obtained on real-world cases. In the upcoming chapter, an autonomous storage system and Intel semi-conductor mini-fab are discussed. The storage system is entirely modeled in $\chi Py$ using a *Jupyter Notebook* as development environment. The Intel mini-fab is analysed to validate the completeness of the feature set $\chi Py$ provides with all functions discussed in this chapter.

# CHAPTER 5

# TESTING $\chi PY$

In this chapter, the classes and functions discussed extensively in chapter 3 and chapter 4 are tested with and compared to real-life examples. In section 5.1, $\chi Py$ is used to work out the *Modeling and simulation of an autonomous vehicle storage and retrieval system* assignment of the TU/e's third year bachelors course *Analysis of Production Systems*. Furthermore, a master thesis on modeling and analysing an Intel Five-Machine Six Step Mini-Fab in $\chi$ is compared to the $\chi Py$ package to validate if all features desirable are present in the current version.

## 5.1 Modeling and simulation of an autonomous vehicle storage and retrieval system (AVS/RS)

This section discusses the $\chi Py$ model of the AVS/RS assignment used in the third year bachelors course *Analysis of Production Systems* at the TU/e. The full model with explanation and substantiation of all model parts are presented in Appendix C.1.

### 5.1.1 Case summary

An AVS/RS is a recently developed fully automated tote storage and retrieval system, see Figure 5.1 [9]. Totes are stored in columns, laid out in several aisles. Each aisle has its own vehicle to pick up totes from the columns to bring them to the aisle's lift, that transports totes to the ground level floor, all depicted in Figure 5.2. This concludes the production line of the AVS/RS.

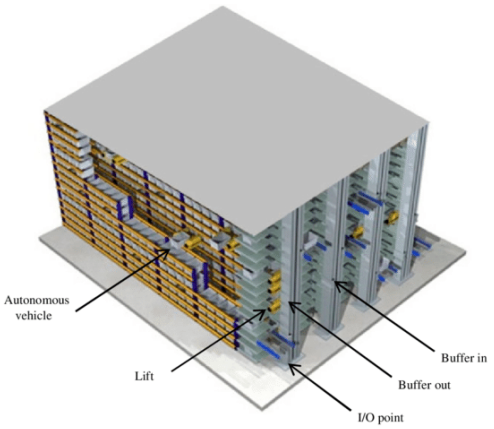


Figure 5.1: The AVS/RS line

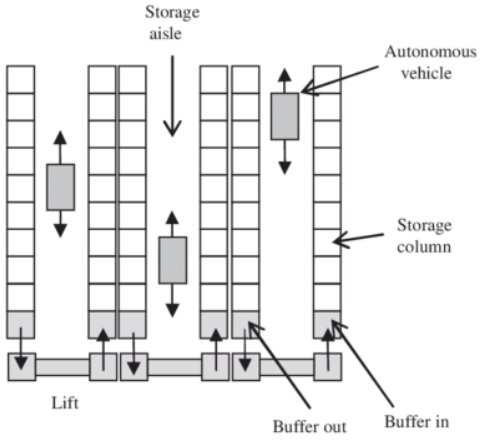


Figure 5.2: A single tier

In the assignment, only one aisle with `column` columns and `Level` tiers has to be modeled. All arriving totes from the generator are placed in a demand buffer per tier (which can be seen as a tote waiting in its column

to be picked up by the vehicle). The vehicle has a buffer size of `bc`. Both the vehicle and lift are accurately modeled according to a trapezoidal speed profile characterised by the maximum velocity and constant acceleration/deceleration of the respective machines. Aside of the travel time, pick and place times are also taken into account in the model.

### 5.1.2 $\chi Py$ model structure

The $\chi Py$ model for this case is written as a *Jupyter Notebook* integrated with the *Anaconda Python Distribution*. This is decided based on the simple installation of *Anaconda*, ease of use of interactive notebooks and cross platform availability. These aspects are all key for this specific assignment, since future students need to be able to work with $\chi Py$ in this case without any issues. $\chi Py$ is manually added by moving the package inside the same directory as the notebook. This way it can directly imported in the notebook as usual.

The model consists of nine parts, corresponding with exercise 1-9 in the assignment [9]. Using this structure, the individual parts of the AVS/RS are modeled with different degrees of complexity and tested separately to ensure that all parts work as expected. When more convenient in $\chi Py$'s context, parts may be merged in a single section.

### 5.1.3 Tested subjects

This assignment tests the modeling of basic workstations (vehicle and lift), distribution sampling for stochastic behavior, context based lot specific process times, finite buffers, custom generator processes (for specific arrival patterns and tote initiation) and context based lot paths through the run environment.

#### Basic workstation modeling

Basic workstation modeling is required in each part, but is specifically tested in parts 1-4, where the demand buffer and vehicle are modeled. These two together form essentially a single $\chi Py$ resource with queue (demand buffer) and vehicle (process). This resource (embedded in the `chipy.Station` class) works exactly as intended and produces identical results to the equivalent $\chi 3$ specifications. This also shows the strength of the $\chi Py$ package: the buffer and vehicle reduce to a single `chipy.Station` object. Note that the case of the second buffer and lift is identical.

#### Distribution sampling

The rate of arrival $t_a$ of totes, as well as the distribution of columns are stochastic. The rate of arrival is distributed exponentially with $\lambda = r_a = \frac{1}{t_a}$ according to the probability density function of Equation 5.1. The implementation is done using `random.expovariate()`, that takes the scale factor $\beta$ as argument.

$$\begin{aligned} f_{exp}(x, \lambda) &= \begin{cases} \lambda e^{-\lambda x}, & \text{for } x \geq 0 \\ 0, & \text{for } x < 0 \end{cases} \\ \lambda &= \frac{1}{\beta} \end{aligned} \tag{5.1}$$

Columns are distributed uniformly over the range of columns in an aisle per tier. Both distributions are parsed as anonymous lambda functions to avoid sample taking at station initiation. Recall from earlier chapters that $\chi Py$ requires most numeric arguments to be lambda functions, even if the argument is a constant. This means that in essence any function can be passed as input, even user-written custom distributions. Stochasticity can be found in nearly every part of the case, and is tested in all parts with favorable results.

#### Context based lot specific process times

The process time of both the vehicle and lift are a function of the maximum velocity and acceleration rate, as well as the distance they have to travel based on the `column` and `tier` value of a individual tote. Since a `chipy.Station` supports lot-specific process times, the only thing that has to be done is to implement the process time calculation in the run environment. Tote specific data (column and tier) can be directly taken from the tote tuple itself and used in the calculation. These calculations are utilised in parts 4, 8 and 9 and work as expected in combination with lot specific process times.

#### Finite buffers

The buffer in front of the lift is a finite buffer with capacity `bc`. This buffer is in $\chi Py$ fashion linked to the lift station. The release controller is activated by communicating the vehicle the lift station object though the `us` argument. The controller works as desired.

**Custom generator processes**

$\chi Py$ supports out of the box two attributes to a lot: its ID and priority level. In this case, priority is not relevant, but the column and tier are. To implement this, the `chipy.Generator` class has to be partially rewritten. This is done by inheriting the class into a new `ToteGenerator` class, which allows tweaking any part of the code as desired. In the case of tote atrributes, both column and tier are added to the `ToteGenerator` init as a lambda function and sampled when a tote is initiated.

The second necessity of inheriting $\chi Py$'s generator is to simulate the custom generation/arrival patterns used in parts 5-8. This requires additional changes to the `gen` method, including an alternative `while` loop condition. The `chipy.simulate` does therefore not work anymore, meaning that the simulation has to be called manually using `env.run()`.

**Context based lot paths through the run environment**

Not each tote walks the same path through the production line. Each tote goes through its own tier before being dropped of at the lift buffer, which is shared with all tiers (each aisle has one lift connecting all tiers with ground level).

To model this in $\chi Py$, a list of tiers is made using a `for` loop, containing a dictionary of stations per tier. This list is parsed by the `ToteGenerator` to the run environment, instead of a standard dictionary of station. This way, the tier attribute of a tote can be used as selector for the correct tier from the list. This method of tier modeling is tested in parts 7-9, and gives identical results to the $\chi 3$ equivalents.

## 5.2 Modeling and simulation of an Intel Five-Machine Six Step Mini-Fab

The final section of chapter 5 discusses the Intel Mini-Fab modeled and analysed by J.P.A. van den Berk in 2003 [3]. In contrast to the AVS/RS assignment, this case is not worked in $\chi Py$. Only the simulation components not present in the AVS/RS model are compared with the current functionality of the $\chi Py$ software to pinpoint possible shortcomings.

### 5.2.1 Case summary

The Intel Five-Machine Six Step Mini-Fab is a semiconductor fabrication plant consisting of five machines spread over three workstations: *Diffusion*, *Lithography* and *Implantation*. The workstations *Diffusion* and *Implantation* each have two identical machines, while the *Lithography* has only one (see Figure 5.3).

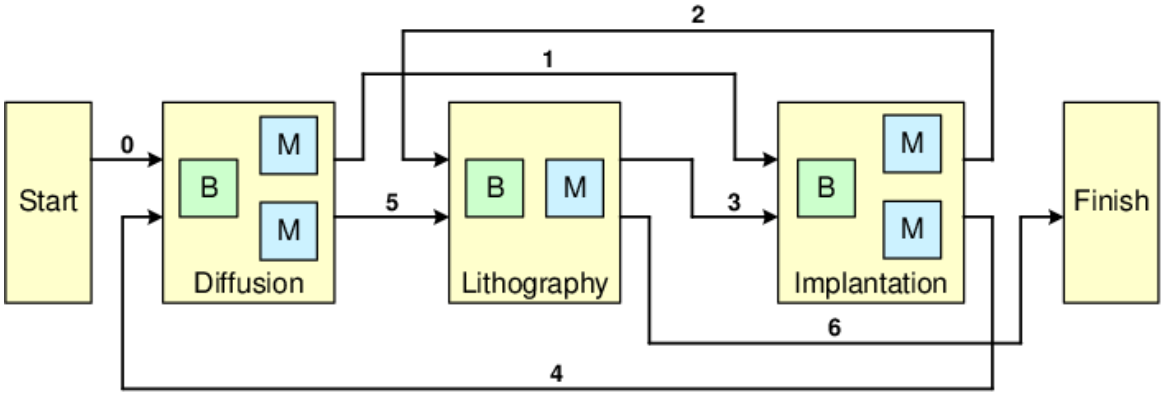


Figure 5.3: The Intel Five-Machine Six Step Mini-Fab's block diagram [3]

Three types of products ("*TEST*", and 2 commercial variants "*A*" and "*B*") go through six processing steps (three *low step*, three *high step*), twice through each station. Transport between individual workstation buffers is done by a transporter. A product waits in the buffer `B` till one of the machines `M` (in the case of the *Lithography* workstation only one) is ready to process the product. After processing, a product returns in the same buffer `B`. In the case of products of the "*TEST*" type, a product can only pass through a machine `M` once during all six steps. The diffusion machines process products in batches of size three, which should contain only products from the same step, as well. If a batch is high step, no mix of commercial products is allowed.

The generator generates three "*TEST*", 51 "*A*" and 30 "*B*" products per week, evenly distributed over the week.

Aside of workstations and a generator, there is one technician and two operators handling the transportation

of orders, as well as the setup of the *Lithography* workstation.

### 5.2.2 Tested subjects

The Intel Mini-Fab contains certain model elements not present in the AVS/RS case, that are compared against $\chi Py$'s current features to identify possible shortcomings.

#### Double purpose buffers

In the Intel Mini-Fab, each workstation has its own single buffer with characteristic capacity. This buffer `B` is used to store both incoming and served products from the same workstation. This queuing method is at odds with the principle of $\chi Py$'s resource. Once a product leaves the resource, it cannot be sent immediately in its buffer, since that would mean it has to request a spot in the workstations process again. Van den Berk modeled this double purpose buffer as two separate buffers with infinite capacity. This is however possible in $\chi Py$: the exit buffer is the buffer of the operator that transports products to the next step.

#### Advanced repairman and operator schedule

$\chi Py$ has support for linking a certain amount of identical repairman to a station, as discussed in subsection 3.1.2. This repairman type simply keeps track of a list of requested repairs in FIFO and repairs a station with the specified repair time of that station. In the Intel Five-Machine Six Step Mini-Fab, a single repairman (*technician*) is present that also has planned breaks and meetings with colleagues. This is impossible with the provided `chipy.repairman` method, but not impossible to implement in $\chi Py$. Breaks and meetings can be implemented as scheduled process interrupts, just like machine failure is implemented in a station. It is however not chosen to implement this by default, since this would result in writing many cases for each different manufacturing plant. $\chi Py$ is meant as a toolbox with basic options to quickly execute discrete event simulations. A programmer skilled in object oriented programming can easily inherit the `chipy.repairman` class and add desired functionality. Note that class inheriting for additional functionality is already done with the `ToteGenerator` in the AVS/RS case.

#### Conditional batching

In the *Implantation* workstation, products are always processed in batches of three. This is possible in $\chi Py$ by setting the `batchsize` and `batchpol` options of the machine to `3` and `"hard"` respectively. This however does not take the batch requirements into account: a batch can only contain products from the same step and in the case of high step: only a single type of commercial products. It is again chosen to not implement this directly in $\chi Py$ for identical reasons as before. The `chipy.PriorResource` class has to be inherited to rewrite the `_trigger_put` that takes care of batching (see Appendix B.2.2).

#### Parallel processing constraints

Products of the type "*TEST*" can only enter a machine `M` once during its full processing (low and high step). This means that when a "*TEST*" product (in Figure 5.3) goes in the low step phase through the upper machine in the *diffusion* workstation, it has to go through the lower machine in the high step phase. In $\chi Py$, parallel machines/processes in a single station are modeled as a resource occupation counter, thus no distinction is made between the individual machines. As long as the capacity counter is not at it maximum value, a new request can be granted. This makes it impossible to model the desired "*TEST*" type behavior, since it requires this exact distinction. A possible way of tackling this is by inheriting the `chipy.Station` class and rewriting its `request` method (see Appendix B.2.3). In specific, the part after the request (the `Process` in Figure 2.2) can be extended by calling two identical resources (both representing one `M` in a workstation) with single capacity and infinite queue length. A `if` loop can be used to check whether the product should enter the first or second machine `M`.

Concluded can be, based on both applications, that the $\chi Py$ package succeeds in modeling a basic production line with some degree of complexity. When more specific functionality is required, $\chi Py$ on its own comes short, and requires manual expansion by the user to implement this functionality. The next chapter concludes this thesis with a conclusion and set of recommendations for future development of the package.

# CHAPTER 6

# CONCLUSION AND RECOMMENDATIONS

The final chapter of this thesis presents the conclusion and recommendations that follow from chapters 2-5.

## 6.1 Conclusion

A solution to the identified issues with the $\chi$3 formalism is proposed in the form of a Python package named $\chi Py$, based on the discrete event toolbox *SimPy*. $\chi Py$ is build around the concept of environment bounded resources that can be occupied by orders flowing through the environment. Since the main goal of $\chi Py$ is to provide a fool-proof package to quickly write discrete-event simulations, predefined classes were written that utilises resources to represent key production line elements, like workstations and repairmen. Additional methods are written to easily call different simulation methods, as well as post processing tools that generate simulation visualisations and data tables to manually post process in third party software solutions.

All this functionality is tested with an example model presented in Appendix A and by modeling an AVS/RS plant in $\chi Py$, which has also been modeled in $\chi$3. Both models have been verified with acclamatory results. The example model is verified with hand calculations and is wholly correct. The AVS/RS models are compared to the readily available $\chi$3 equivalents with zero error in deterministic cases and comparable results for stochastic cases.

$\chi Py$'s options are also compared to an *Intel Five-Machine Six Step Mini-Fab* , known for covering most key elements of a production line. While $\chi Py$ in its current form is able to model most parts in a basic sense, it lacks certain specific features. It is, however, chosen to not implement these features inside the package to keep the package simple and globally applicable. If a simulation has specific needs, $\chi Py$ classes and methods can be, as every *Python* class and method, inherited and altered to fit these needs. This is also shown in the AVS/RS case.

Concluding, $\chi Py$ is a good alternative to $\chi$3 and its ancestors. It allows for easy and simple modeling of surfeit discrete event systems, like a production line, store check-out or hospital. Compared to the $\chi$ formalism, $\chi Py$ also has the advantage of build-in post processing tools to quickly validate written models. The implementation of specific requirements on provided functions is, per contra, more difficult (but not impossible) to implement. It requires class and/or method inheritance and a proper understanding of the $\chi Py$ source code.

## 6.2 Recommendations

As became clear from the earlier chapters along with the conclusion, $\chi Py$ also has its own imperfections. While nothing can be perfect, a set of proposals are established in the form of recommendations (with a trace of a solution) for future development on the $\chi Py$ package that iron out some indicated imperfections.

First and foremost, the fundamental `chipy.PriorResource` force-combines queues and processes as one initiated class. This means that the two are inseperable, which makes it hard to model cases where this is not the case, as seen with the *Intel Mini-Fab*. A solution can be to decouple the queuing part (`chipy.PriorQueue`) as a

separate class the user can initiate and link to a station. this way, two workstations can for example share the same buffer.

Second, each station can only have one linked repairman object. This means there is no support to add repairmen of different type to the same station, while this is might desired in some models. A solution can be to provide a list of repairman objects to a station instead of a single one. This will, however, require an additional scheduler to ensure the most efficient option is always chosen: the repairman that is available first.

Another point of improvement related to machine failure can be the introduction of stochasticity to the $m_r$ and $m_f$ variables. In the current version, $\chi Py$ only allows input of a single value, and not a lambda function. Therefore, no distribution can be passed as an argument to sample from. This can be implemented by allowing a lambda function to parse as argument, but also requires caution with the release controller: it should still be able to make a correct hold time prediction!

Currently, the release controller of resource does not take into account that the hold release time $t_h$ is, in the presence of a repairman object, dependent of the amount of available repairmen and the amount of repair requests in the queue. A general expression has to be derived that relates the hold time to these dependencies that can be utilised to calculate the time it takes until the upcoming station is repaired by a repairman. The found expression can be implemented in the `trigger_get` method of the `chipy.PriorResource` class.

While not an issue in the studied cases, the used sorting function for a resource's queue is inefficient. It has to check the priority level and entry time for each individual element to sort the entire list in the correct order to select the correct element that gets processed. A better implantation of sorting can be for example bi-sectional sorting, which will likely decrease the simulation time when queue sizes become very large.

Lastly, simulation times can become exceptionally long when simulating complex systems (like the AVS/RS) for excessive amounts of orders (like $10^6$). This issue is not caused by the $\chi Py$ package directly, but a common *Python* issue since the language is quite inefficient resource-wise. Still, an effort can be done to write the source even more compact and "*pythonic*". In addition, fitting debugging tools can be used to identify the culprits.

# BIBLIOGRAPHY


[1] SE Software, "Chi 3," 2016. [Online]. Available: https://cstweb.wtb.tue.nl/chi/trunk-r9682/

[2] I. J. B. F. Adan, A. T. Hofkamp, and J. E. Rooda, "Chi 3 tutorial," Eindhoven, 11 2017.

[3] J. P. A. V. D. Van den Berk, E. Lefeber, and J. E. Rooda, "Analysis of the Intel Five-Machine Six Step Mini-Fab," *Systems Engineering*, pp. 2003–2003, 2003.

[4] I. J. B. F. Adan, E. Lefeber, S. Pogromsky, and M. Reniers, *Lecture Notes 4DC10*, december 2 ed., I. Adan, E. Lefeber, S. Pogromsky, and M. Reniers, Eds. Eindhoven: Eindhoven University of Technology, 2019.

[5] O. Lünsdorf and S. Scherfke, "SimPy." [Online]. Available: https://simpy.readthedocs.io/en/latest/

[6] ——, "History and Changes." [Online]. Available: https://simpy.readthedocs.io/en/latest/about/history.html

[7] ——, "Defense of Design." [Online]. Available: https://simpy.readthedocs.io/en/latest/about/defense_of_design.html

[8] u. Chi, I. Adan, A. Hofkamp, J. Rooda, and J. Vervoort, "Analysis of Manufacturing Systems," Eindhoven, 10 2012. [Online]. Available: http://se.wtb.tue.

[9] E. Lefeber, "Modeling and simulation of an autonomous vehicle storage and retrieval system," Eindhoven, 2020.

# Appendices

# APPENDIX A

# EXAMPLE $\chi PY$ MODEL

This appendix presents a simple example $\chi Py$ model to demonstrate all classes and functions discussed in chapter 3. The model is self-evidently written according to the $\chi Py$ modeling convention of Figure 3.1.

## A.1 The model

The model simulates a production line consisting of a generator and a single station named `Station 1`. The generator generates lots based on a deterministic rate of arrival $t_a$ $[h^{-1}]$ defined by user input. The same accounts for the simulation time $t$ $[h]$. Each time unit in the model corresponds with one minute. A single environment is initiated, to which all simulation elements are bounded.

The only workstation of the model, `Station 1`, has a global exactly determined $t_e$ of 1 $[min]$. The station consists of a single processing unit and a buffer with infinite buffer size and FIFO order sorting. Station 1 is build poorly, since it has a time of failure $m_r = 12$ $[min]$. It takes 3 $[min]$ to repair the station, which is taken care of by a single repairman. There is no case of batching; each lot is processed individually.

The run environment `runenv` keeps track of entering and exiting lots, and handles lots entering `station 1`.

A lot arrival pattern is generated by the generator `G`. The generator sends every $t_a$ a single lot in the run environment. Lots have no priority, since each lot has the same priority level 0.

The `def main()` method is the method that is be executed when the script is called from the command line. It starts the simulation using `cp.simulate` and runs the simulation until time `t` is reached. After the simulation terminated, data is post processed using the stats printer, flow time diagram generator and station statistics generator. Note that the `cp.stattion_stats` method is nested in a for loop, to ensure a plot is made for every station in the `stations` dictionary. In this case this is however not required, since there is only one station. The results of the post processing are presented in the upcoming two sections.

example.py

```
#! usr/bin/python3
#============================================================
# IMPORTS
#============================================================
import chipy as cp
#============================================================
# MODEL VARIABLES
#============================================================
t: float = float(input("Simulation Time [h] = "))*60        # Simulation Time
roa: float = float(input("Rate of Arrival [/h] = "))        # Rate of Arrival
ta: float = (roa/60)**(-1)                                  # Inter-Arrival time of lots
t_arr: list = []                                            # Arrive times of lots
t_exit: list = []                                           # Exit times of lots
lots: list = []                                             # List of processed lots
inf: float = float("inf")                                   # Infinity
env = cp.environment()                                      # Environment Initiation
```

```
#============================================================
# STATIONS
#============================================================
repairman = cp.Repairman(
        env = env,
        cap = 1,
        pol = "FIFO")
stations = {
            "Station 1": cp.Station(
                env             = env,
                te              = lambda: 1,
                cap             = 1,
                queuesize       = inf,
                queuepol        = "FIFO",
                mf              = 12,
                mr              = 3,
                repairman       = repairman,
                batchsize       = 1,
                batchpol        = "hard",
                data_collection = True)
    }
#============================================================
# LOT ARRIVATION
#============================================================
def runenv(env, lot, stations):
    arr = env.now
    t_arr.append([lot[0], arr])

    yield env.process(stations["Station 1"].proc(lot = lot))

    end = env.now
    t_exit.append([lot[0], end])
    lots.append(lot)
#============================================================
# GENERATOR
#============================================================
G = cp.Generator(
            env = env,
            ta = lambda: ta,
            stations = stations,
            runenv = runenv,
            lots = lots,
            priority = lambda: 0)
#============================================================
# MAIN
#============================================================
def main():
    cp.simulate(
            env = env,
            generator = G,
            method = "Time",
            sim_time = t)
    cp.flowtime_diagram(
            lots = lots,
            t_start =  t_arr,
            t_end = t_exit,
            sim_time = t,
            savename = "flowtime.svg",
            title = f"Flowtime Diagram ($t_a = {ta}$)",
            xlabel = "$t \, [min]$",
            ylabel = "Product ID",
            color = "maroon"
            )
    cp.stats(lots = lots,
            t_start = t_arr,
            t_end = t_exit,
            sim_time = t,
            generate_table = True)
    for key in stations:
        cp.station_stats(
            stations = stations,
            station_name = key,
            show_avg = False,
            savename = "stats.svg",
            sim_time = t)
#============================================================
```

```
93 # MISC
94 #============================================================
95 if __name__ == "__main__":
96     main()
```

To call the simulation, one executes `python3 example.py` in the script's directory.

## A.2 Console output

As can be seen from the output below, $t_a = \frac{1}{r_a} = \frac{60}{15} = 4$ $[min]$ and $t = 60$ $[min]$.

**`example.py` console output**

```
Simulation Time [h] = 1
Rate of Arrival [/h] = 15
.plots/flowtime.svg .pdf_tex conversion succesful
Flowtime Diagram saved as .plots/flowtime.svg

============================================================
SIMULATION SUMMARY
============================================================

The average flow time equals 1.4285714285714286
The (population) standard deviation of the flow time equals 0.9035079029052513
The coefficient of variation of the flow time equals 0.6324555320336759
The amount of products that passed through the run environment is 14 in 60.0 time units

============================================================
FLOW TIME PER LOT ID
============================================================

 Lot ID  Flow Time  Entry Time
      0        1.0         0.0
      1        1.0         4.0
      2        1.0         8.0
      3        4.0        12.0
      4        1.0        16.0
      5        1.0        20.0
      6        1.0        24.0
      7        3.0        28.0
      8        1.0        32.0
      9        1.0        36.0
     10        1.0        40.0
     11        2.0        44.0
     12        1.0        48.0
     13        1.0        52.0

(This table is also saved as /data/phi_data.csv)

============================================================
.plots/stats.svg .pdf_tex conversion succesful
Statistiscs Diagram of Station 1 saved as .plots/stats.svg
```

In total, 14 of the 15 generated lots fully pass through the run environment within $t$. The reason not all 15 pass is due to machine failure. Using Equation 3.1 and $m_f = 12$ , $m_r = 3$, one can determine the times of failure, as depicted in Table A.1.

Table A.1: Points of failure of Station 1

| **i** | Time of failure $[min]$ |
|---|---|
| **1** | 12 |
| **2** | 27 |
| **3** | 42 |
| **4** | 57 |

From the console output listing, it can be seen that lots 3, 7, 11 and 15 are affected by machine failure. Lot 3 enters at the time of failure, meaning that the total time it passes inside Station 1 equals $t_{r,3} = t_e + m_r = 4$ $[min]$. When lot 7 arrives at the station at $t = 28$, one minute is already spent by the repairman to repair the station meaning that $t_{r,7} = t_e + (m_r - 1) = 3$ $[min]$. The same accounts for lot 11 but in that case $t_{r,11} = 2$. The last generated lot enters the station at $t = 56$ $[min]$. Just at the point when the lot wants to exit the station, the station unfortunately fails, meaning that it has to wait another $m_r$ before leaving. At that point, $t = 60$ $[min]$, the point when the simulation stops. This means that lot 15 did not left the run environment in time.

The flow time's average $\bar{\varphi}$, population standard deviation $\sigma$ and variability $c$ can be determined manually using Equation A.1.

$$\begin{aligned} \bar{\varphi} &= \frac{\sum_{i=1}^{N} \varphi_i}{N} = \frac{11 \cdot (1) + \sum_{i=2}^{4} i}{14} = \frac{10}{7} \approx 1.43 \\ \sigma &= \sqrt{\frac{\sum_{i=1}^{N} (\varphi_i - \bar{\varphi})}{N}} = \sqrt{\frac{11 \cdot (1 - \frac{10}{7}) + \sum_{i=2}^{4} (i - \frac{10}{7})}{14}} = \sqrt{\frac{40}{49}} \approx 0.90 \\ c &= \frac{\sigma}{\bar{\varphi}} = \frac{7 \cdot \sqrt{\frac{40}{49}}}{10} = \frac{\sqrt{10}}{5} \approx 0.63 \end{aligned} \tag{A.1}$$

## A.3 Visual output

The script generates two visualisations: the statistic summary of Station 1 and the flow diagram of all lots going through the run environment.

The statistics of Station 1 are shown in Figure A.1. All fail periods indicated in Table A.1 are present in the plot. The calculated influence of machine failure on the lot run time $t_r$ can be seen in the occupancy rate plot in the top right. Lots never have to wait in the buffer, hence the queue size plot is a zero line.

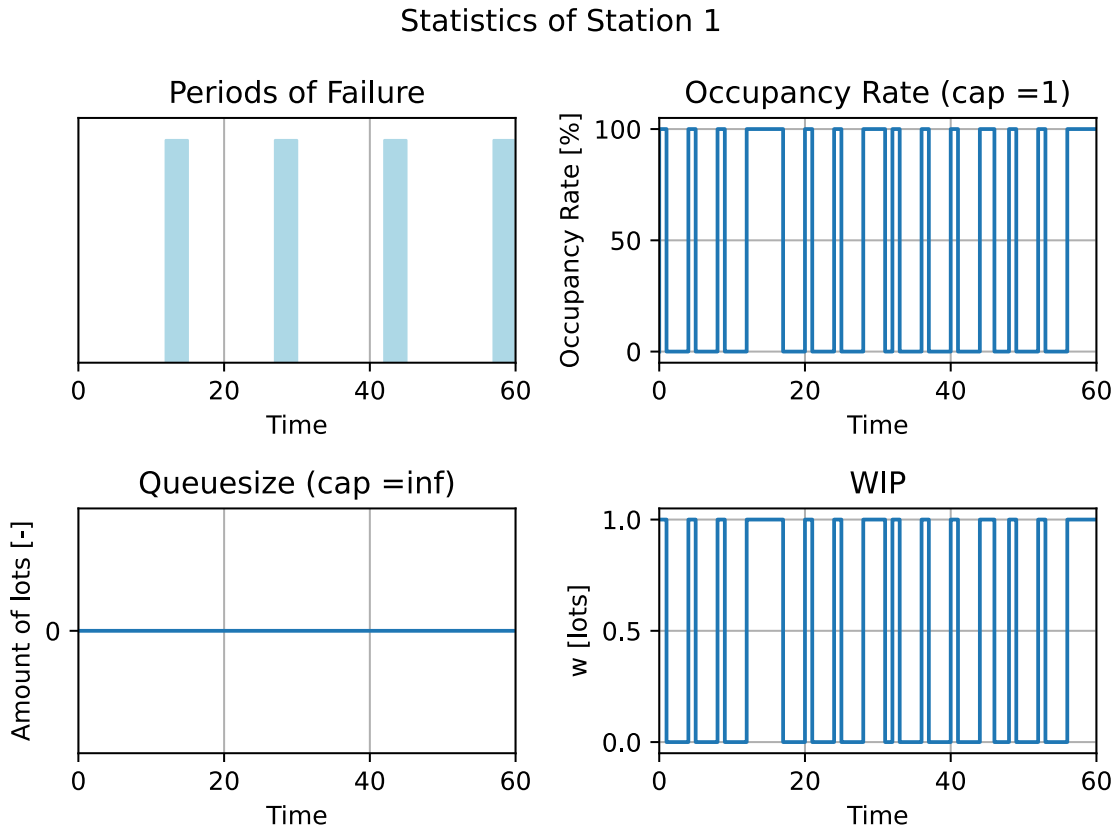


Figure A.1: Resulting statistics four-panel of `Station 1`

As already mentioned, the script also outputs a lot flow time diagram that is shown in Figure A.2. As defined in the model, the title contains the rate of arrival and the blocks have a maroon color.

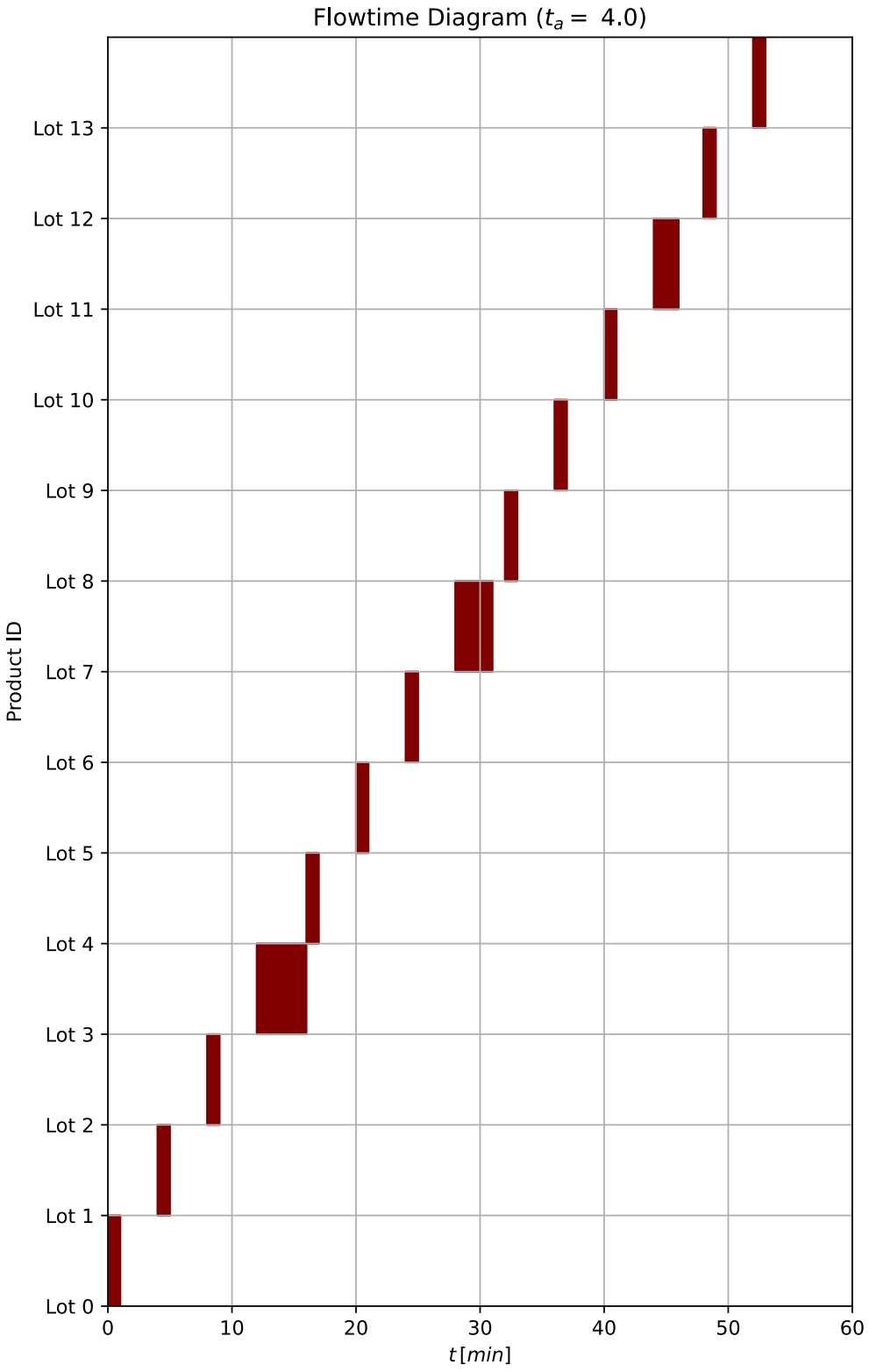


Figure A.2: Resulting of flow time diagram

It can be seen that most lots have the same process time, except for the lots that are in Station 1 while it is broken. Flow times never overlap each other, indicating a constant queue size of zero. Note that the time intervals when a lot is occupying the station correspond with the 100% occupancy rate intervals of Figure A.1.

# APPENDIX B

# $\chi PY$ SOURCE CODE

This appendix presents the source code of the $\chi Py$ package. In the first section the package structure is presented visually. In Appendix B.2, the full source code of the each class and method is shown.

## B.1 $\chi Py$ hierarchy

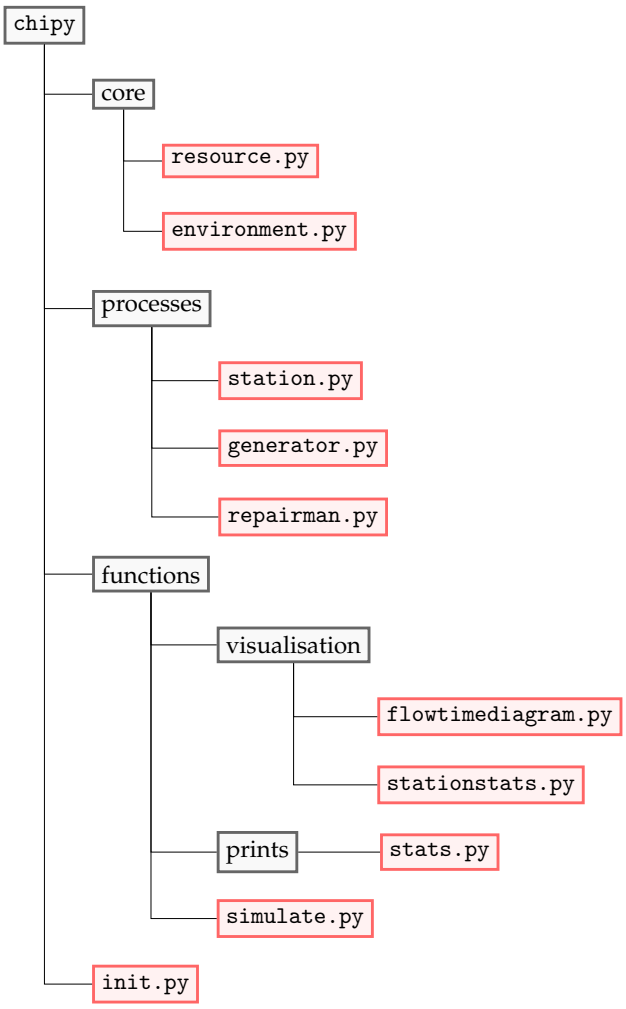


Figure B.1: $\chi Py$ package structure.

## B.2 Source code listings

### B.2.1 Package Initiation

chipy.__init__.py

```
#! usr/bin/python3
ver = "20210615"
print(f"ChiPy version {ver} imported succesfully.\nUse Python's help() (e.g. help(chipy.
    Station.proc)) for additional information of ChiPy\ncomponents.\n ")
from .core.environment import environment
from .functions.visualisation.flowtimediagram import flowtime_diagram
from .functions.visualisation.stationstats import station_stats
from .processes.station import Station
from .processes.generator import Generator
from .processes.repairman import Repairman
from .core.resource import BasicResource, PriorQueue, PriorResource
from .functions.simulate import simulate
from .functions.prints.stats import stats
```

### B.2.2 Core

chipy.core.environment.py

```
#! usr/bin/python3

#=============================================================================
# ~ ENVIRONMENT ~
#  An exact copy of the simpy.Environment()
#=============================================================================
#============================================================
# IMPORTS
#============================================================
import simpy
import chipy
#============================================================
# METHOD(S)
#============================================================
def environment():
    #------------------------------------------------------------
# HELP
#------------------------------------------------------------
    '''
    #------------------------------------------------------------
    # INFO
    #------------------------------------------------------------

    A environments which keeps track of time and holds processes.

    #------------------------------------------------------------
    # ARGUMENTS
    #------------------------------------------------------------

    This method returns a simpy.Environment class object and has no arguments.
    '''
    return simpy.Environment()
```

chipy.core.resource.py

```
#! usr/bin/python3
#============================================================================
# ~ RESOURCE ~
#  Resources for the Station Class
#============================================================================
#============================================================
# IMPORTS
#============================================================
import simpy
import chipy
import numpy as np
from typing import TYPE_CHECKING, Optional
import sys
#============================================================
# CLASS(ES)
#============================================================
#------------------------------------------------------------
# REQUEST ACTION
#------------------------------------------------------------
class PriorRequest(simpy.resources.resource.PriorityRequest):
    def __init__(self, resource: 'Resource', priority: int = 0, preempt: bool = True, us
     = "NaN"):
        super().__init__(resource, priority, preempt)
        self.us = us
#------------------------------------------------------------
# RELEASE ACTION
#------------------------------------------------------------
class PriorRelease(simpy.resources.resource.Release):
    def __init__(self, resource: 'Resource', request: 'Request'):
        super().__init__(resource, request)
        pass
#------------------------------------------------------------
# BASIC RESOURCE (DEPRICATED)
#------------------------------------------------------------
class BasicResource(simpy.Resource):
    pass
#------------------------------------------------------------
# BUFFER/QUEUE
#------------------------------------------------------------
class PriorQueue(list):
    def __init__(self, maxlen: float = float("inf"), pol: str = "FIFO"):
        super().__init__()
        self.maxlen     = maxlen
        self.queuepol   = pol
    def append(self, item) -> None:
        if self.maxlen is not None and len(self) >= self.maxlen:
            raise RuntimeError("ChiPy Error: Queue is full")
        if self.queuepol == "FIFO":
            super().append(item)
            super().sort(reverse = False, key=lambda e: e.key)
        elif self.queuepol == "LIFO":
            super().insert(0,item)
            super().sort(reverse = True, key=lambda e: e.key)
        else:
            raise RuntimeError(f"ChiPy Error: Queue policy \"{self.queuepol}\"not
    supported")
#------------------------------------------------------------
# PRIORITY RESOURCE
#------------------------------------------------------------
class BufferError(Exception):
    pass
class PriorResource(simpy.PriorityResource):
    PutQueue = PriorQueue
    def __init__(self, env, capacity: int = 1, queuesize: float = float("inf"), queuepol
    : str = "FIFO", batchsize: int = 1, batchpol: str = "hard"):
        super().__init__(env, capacity)
        self.queuesize  = queuesize
        self.queuepol   = queuepol
        self.batchsize  = batchsize
        self.batchpol   = batchpol
        self.put_event_batch = []
        self.put_queue = self.PutQueue(self.queuesize, self.queuepol)
    request = simpy.core.BoundClass(PriorRequest)
```

```
71      release = simpy.core.BoundClass(PriorRelease)
72      def _do_put(self, event) -> None:
73          if len(self.users) < self.capacity:
74              self.users.append(event)
75              event.usage_since = self._env.now
76              event.succeed()
77      def _trigger_put(self, get_event) -> None:
78          idx = 0
79          put_event_batch = []
80          while idx < len(self.put_queue):
81              put_event = self.put_queue[idx]
82              if not put_event.triggered:
83                  put_event_batch.append(put_event)
84              if (self.batchpol == "hard"):
85                  self.cond = lambda: (len(put_event_batch) == self.batchsize)
86              elif (self.batchpol == "soft"):
87                  self.cond = lambda: ((len(put_event_batch) <= self.batchsize) and ((len(
    put_event_batch) > 0)))
88              else:
89                  raise RuntimeError(f"ChiPy Error: Batch policy \"{self.batchpol}\" not
    supported")
90              if (self.cond()):
91                  for j in np.arange(0,(len(put_event_batch)),1):
92                      event = put_event_batch[j]
93                      if (self.batchsize > 1):
94                          self._capacity_org = self._capacity
95                          self._capacity = self.batchsize*self._capacity
96                      proceed = self._do_put(event)
97                      if (self.batchsize > 1):
98                          self._capacity = self._capacity_org
99                  put_event_batch.clear()
100             if not put_event.triggered:
101                 idx += 1
102             elif self.put_queue.pop(idx) != put_event:
103                 raise RuntimeError('Put queue invariant violated')
104
105     def hold(self, t_hold) -> None:
106         yield self._env.timeout(t_hold)
107     def _trigger_get(self, put_event) -> None:
108         idx = 0
109         while idx < len(self.get_queue):
110             get_event       = self.get_queue[idx]
111             us              = get_event.request.us
112             t_hold_start    = self._env.now
113             t_hold_end      = self._env.now
114             t_hold          = 0
115             t_hold_list     = []
116             if (us == "NaN"):
117                pass
118             elif (len(us.res.put_queue) == us.queuesize):
119                 t_hold_release = self._env.now
120                 for j in range(0,len(us.run_list),1):
121                     t_hold_list.append((us.run_list[j][1]+us.run_list[j][2])-self._env.
    now)
122                 t_hold = min(t_hold_list)
123                 t_hold_list.clear()
124                 if (len(us.fix_list) == 0):
125                     t_last_fix = 0
126                 else:
127                     t_last_fix = us.fix_list[-1]
128                 if (us.mf-(self._env.now - t_last_fix) <= t_hold):
129                     if (us.repairman == None):
130                         t_hold = t_hold + us.mr
131                     else:
132                         t_hold = t_hold + (us.mr)
133                 t_hold_start = self._env.now
134                 self._env.run(self._env.process(self.hold(t_hold)))
135                 t_hold_end = self._env.now
136             try:
137                 if ((t_hold_end - t_hold) != (t_hold_start)):
138                     raise BufferError
139             except BufferError as e:
140                 pass
141             else:
142                 proceed = self._do_get(get_event)
143                 if not get_event.triggered:
```

```
144                     idx += 1
145                 elif self.get_queue.pop(idx) != get_event:
146                     raise RuntimeError('Get queue invariant violated')
147                 if not proceed:
148                     break
```

### B.2.3 Processes

chipy.processes.station.py

```
1  #! usr/bin/python3
2  #=============================================================================
3  # ~ STATION ~
4  #   A Station
5  #=============================================================================
6  #============================================================
7  # REQUIRED IMPORTS
8  #============================================================
9
10 import numpy as np
11 import chipy as cp
12 import simpy
13
14 #============================================================
15 # CLASS(ES)
16 #============================================================
17
18 class Station(object):
19 #------------------------------------------------------------
20 # HELP
21 #------------------------------------------------------------
22     '''
23     #------------------------------------------------------------
24     # INFO
25     #------------------------------------------------------------
26
27     A Station that processes lots
28
29     #------------------------------------------------------------
30     # ARGUMENTS
31     #------------------------------------------------------------
32
33     env: chipy.environment
34         The Environment.
35     te:
36         The process time, expressed as a lambda function. Always parse the process time
       as an argument to the station object at initiation, except when a lot-specific
       process time is desired. Execute help(chipy.Station.proc) for more details on lot-
       specific process times Default: None.
37     cap: int
38         The capacity of a station. In other words: the amount of parallel machines in a
       station. Default: 1.
39     queuesize: int
40         The size of a stations buffer. Default: float("inf").
41     queuepol: str
42         The buffer policy. FIFO or LIFO. Default: FIFO.
43     mf: float
44         Mean time till failure. Default: float("inf").
45     mr: float
46         Mean time of repair. Default: float("inf").
47     repairman: chipy.Repairman Object
48         A repairman (or identical set of repairmen) that are requested to repair a
       machine at failure. Default: None.
49     batchsize: int
50         The size of a batch. Default: 1.
51     Batchpol: str
52         The buffer policy. "hard" or "soft". Default: "hard".
53     data_collection: str
54         Boolean to determine if a station keeps track of non-obligatory statistics for
       post processing.
55         Slows down simulation. Default: False.
56     '''
57 #------------------------------------------------------------
```

```
58 # INIT
59 #-------------------------------------------------------------
60     def __init__(self, env, te = None, cap: int = 1, queuesize: int = float("inf"),
       queuepol: str = "FIFO", mf: float = float("inf"), mr: float = float("inf"),
       repairman = None, batchsize: int = 1, batchpol: str =  "hard", data_collection: bool
        = False):
61         self.env                = env
62         self.te                 = te
63         self.cap                = cap
64         self.queuesize          = queuesize
65         self.queuepol           = queuepol
66         self.mf                 = mf
67         self.mr                 = mr
68         self.repairman          = repairman
69         self.batchsize          = batchsize
70         self.batchpol           = batchpol
71         self.data_collection    = data_collection
72         self.res                = cp.PriorResource(self.env, self.cap, self.queuesize,
       self.queuepol, self.batchsize, self.batchpol)
73         self.broken             = False
74         self.num_broken         = 0
75         self.num_processed      = 0
76         self.p                  = None
77         self.fail_list          = []
78         self.fix_list           = []
79         self.buffersize_list    = []
80         self.occupancyrate_list = []
81         self.phi_list           = []
82         self.wip_list           = []
83         self.avgwip_list        = []
84         self.run_list           = []
85 #-------------------------------------------------------------
86 # DATA COLLECTION
87 #-------------------------------------------------------------
88     def bs(self) -> None:
89         if (self.data_collection == False):
90             pass
91         else:
92             self.buffersize_list.append([self.env.now, len(self.res.put_queue)])
93     def ocr(self) -> None:
94         if (self.data_collection == False):
95             pass
96         else:
97             if (self.batchsize > 1):
98                 cap = self.batchsize*self.cap
99             else:
100                cap = self.cap
101            self.occupancyrate_list.append([self.env.now, (self.res.count)/(cap)*100])
102    def wip(self) -> None:
103        if (self.data_collection == False):
104            pass
105        else:
106            if (len(self.wip_list) == 0):
107                self.avgwip_list.append([self.env.now , self.res.count])
108            else:
109                i = (len(self.wip_list) -1)
110                avgwip = ((self.avgwip_list[i-1][0])/(self.env.now))*self.avgwip_list[i
       -1][1]+((self.env.now-self.avgwip_list[i-1][0])/(self.env.now))*self.wip_list[i
       -1][1]
111                self.avgwip_list.append([self.env.now , avgwip])
112            self.wip_list.append([self.env.now , self.res.count])
113
114 #-------------------------------------------------------------
115 # RUN
116 #-------------------------------------------------------------
117     def run(self, lot, t_run) -> None:
118         self.run_list.append([lot[0],self.env.now,t_run])
119         self.bs()
120         yield self.env.timeout(t_run)
121 #-------------------------------------------------------------
122 # REQUEST
123 #-------------------------------------------------------------
124     def request(self,lot,us,te = None) -> float:
125         with self.res.request(priority=lot[1], us = us) as req:
126                 yield req
127                 self.wip()
```

```
128                 self.ocr()
129                 t_in = self.env.now
130                 if (te == None):
131                     if (self.te == None):
132                         raise RuntimeError("ChiPy Error: No process time (te) defined")
133                     else:
134                         t_e = self.te()
135                 else:
136                     t_e = te()
137                 try:
138                     if (self.broken == False):
139                         self.p = self.env.process(self.run(lot, t_e))
140                         yield self.p
141                     else:
142                         self.p =  self.env.process(self.run(lot, t_e+(self.mr-(self.env.
    now-self.t_fail))))
143                         yield self.p
144                 except simpy.Interrupt:
145                     itm = [item for item in self.run_list if item[0] == lot[0]]
146                     self.run_list.remove(itm[0])
147                     self.p = self.env.process(self.run(lot, (t_e + self.mr - (self.
    t_fail-t_in))))
148                     yield self.p
149                 return t_in
150 #-------------------------------------------------------------
151 # PROCESS
152 #-------------------------------------------------------------
153     def proc(self, lot, us = "NaN", te = None) -> None:
154 #-------------------------------------------------------------
155 # HELP
156 #-------------------------------------------------------------
157         '''
158         #-------------------------------------------------------------
159         # INFO
160         #-------------------------------------------------------------
161 
162         Request a Station to process a lot.
163 
164         #-------------------------------------------------------------
165         # ARGUMENTS
166         #-------------------------------------------------------------
167 
168         lot:
169             The lot that passes through the system
170         te:
171             The lot-specific process time (if desired), expressed as a lambda function.
    Default: None.
172         us: chipy.Station Object
173             The upcoming station (hence "us")  to control hold-time for timed releases
    to avoid buffer overflow. Default: "NaN"
174         '''
175         self.bs()
176         t_in    = yield self.env.process(self.request(lot, us, te))
177         t_out   = self.env.now
178         itm     = [item for item in self.run_list if item[0] == lot[0]]
179         self.run_list.remove(itm[0])
180         self.phi_list.append(t_out-t_in)
181         self.num_processed += 1
182         self.bs()
183         self.wip()
184         self.ocr()
185 #-------------------------------------------------------------
186 # FAIL BEHAVIOUR
187 #-------------------------------------------------------------
188     def fail(self, lots_max = None) -> None:
189         if (lots_max == None):
190             cond = lambda: 1
191         else:
192             cond = lambda: (self.num_processed < (lots_max))
193         while (cond()):
194             if (self.mf == float("inf")):
195                 return
196             yield self.env.timeout(self.mf)
197             self.broken = True
198             self.t_fail = self.env.now
199             self.fail_list.append(self.t_fail)
```

```
200             self.num_broken += 1
201             if (self.p != None):
202                 if (self.p.is_alive == True):
203                     self.p.interrupt()
204             if (self.repairman == None):
205                 yield self.env.timeout(self.mr)
206             else:
207                 yield self.env.process(self.repairman.repair(self.mr))
208             self.t_fix  = self.env.now
209             self.fix_list.append(self.t_fix)
210             self.broken = False
```

chipy.processes.generator.py

```
1  #! usr/bin/python3
2  #============================================================================
3  # ~ GENERATOR ~
4  #   Generator for lots. Also sends lots in their `runenv`
5  #============================================================================
6  #============================================================
7  # CLASS(ES)
8  #============================================================
9  class Generator(object):
10     '''
11     #------------------------------------------------------------
12     # INFO
13     #------------------------------------------------------------
14
15     Lot generator that generates lots and sends them in a 'runenv'
16
17     #------------------------------------------------------------
18     # ARGUMENTS
19     #------------------------------------------------------------
20
21     env: simpy.Environment
22         The Environment.
23     ta:
24         Inter Arrival Time of lots. The inverse of the Rate of Arrival. Should be
       provided as a lambda function.
25     stations: dict
26         A dictionary containing all chipy.station objects used in the simulation
27     runenv:
28         The method that describes station interaction.
29     lots: list
30         A list containing all processed lots with priority levels as a tuple.
31     priority: Lambda
32         Lambda function describing the distribution of priority levels. Default: lambda:
        0.
33     '''
34     def __init__(self, env, ta, runenv, lots, stations, priority = lambda: 0):
35         self.env = env
36         self.ta = ta
37         self.stations = stations
38         self.runenv = runenv
39         self.lots = lots
40         self.priority = priority
41     def gen(self, env, method = "Time", lots_max = None) -> None:
42         if (method == "Time"):
43             cond = lambda: 1
44             for key in self.stations:
45                 self.env.process(self.stations[key].fail(None))
46         elif (method == "n-processed"):
47             cond = lambda: (len(self.lots) < lots_max)
48             for key in self.stations:
49                 self.env.process(self.stations[key].fail(None))
50         elif (method == "n-generated"):
51             cond = lambda: (lot_ID < lots_max)
52             for key in self.stations:
53                 self.env.process(self.stations[key].fail(lots_max))
54         lot_ID: int = 0
55         while (cond()):
56             lot = (lot_ID, self.priority())
57             self.env.process(self.runenv(env, lot, self.stations))
58             yield env.timeout(self.ta())
```

```
59              lot_ID += 1
```

chipy.processes.repairman.py

```
#! usr/bin/python3
#==============================================================================
# ~ REPAIRMAN ~
#   A Repairman that repairs a station
#==============================================================================
#============================================================
# REQUIRED IMPORTS
#============================================================
import numpy as np
import chipy as cp
import simpy
#============================================================
# CLASS(ES)
#============================================================
class Repairman(object):
#------------------------------------------------------------
# HELP
#------------------------------------------------------------
    '''
    #--------------------------------------------------------
    # INFO
    #--------------------------------------------------------

    A Station that processes lots

    #--------------------------------------------------------
    # ARGUMENTS
    #--------------------------------------------------------

    env: chipy.environment
        The Environment.
    cap: int
        The amount of repairmen present of thsi type. Default: 1.
    pol: str
        The request handling order policy. FIFO or LIFO. Default: FIFO.
    '''
#------------------------------------------------------------
# INIT
#------------------------------------------------------------
    def __init__(self, env, cap: int = 1, pol: str = "FIFO"):
        self.env = env
        self.cap = cap
        self.pol = pol
        self.res = cp.PriorResource(self.env, self.cap, float("inf"), self.pol)
        self.wait_list = []
        self.run_list = []
#------------------------------------------------------------
# REPAIR
#------------------------------------------------------------
    def rep(self, t_run) -> None:
        self.wait_list.pop(0)
        self.run_list.append([self.env.now,t_run])
        yield self.env.timeout(t_run)
    def request(self, mr, prior) -> None:
        with self.res.request(priority=prior) as req:
            self.wait_list.append([self.env.now,mr])
            yield req
            yield self.env.process(self.rep(mr))
            self.run_list.pop(0)
    def repair(self, mr, prior: int = 0) -> None:
        yield self.env.process(self.request(mr, prior))
```

### B.2.4 Functions

chipy.functions.simulate.py

```
#! usr/bin/python3
#============================================================================
# ~ SIMULATE ~
#  Starts a Simulation
#============================================================================
#============================================================
# IMPORTS
#============================================================
import simpy
import chipy
#============================================================
# METHOD(S)
#============================================================
def simulate(env, generator, method: str = "Time", sim_time: float =60, lots_max: int = 1):
#------------------------------------------------------------
# HELP
#------------------------------------------------------------
    '''
    #------------------------------------------------------------
    # INFO
    #------------------------------------------------------------

    Generates a flowtime diagram of all lots that completed the simulation.
    Note that for svg to pdf_tex conversion, 'Inkscape' has to be installed.

    #------------------------------------------------------------
    # ARGUMENTS
    #------------------------------------------------------------

    env: chipy.environment
        The Environment.
    generator: chipy.Generator object
        The generator used in the simulation, bound to env.
    method: string
        The method of simulation:
            - "Time": Simulate until time 'sim_time' is reached. Additional required argument(s): sim_time.
            - "n-processed": Simulate until 'lots_max' orders are processed, while continuing the generator process. Additional required argument(s): lots_max.
            - "n-generated": Simulate until n orders are generatedand processed. The generator stops aftern orders are generated. Additional required argument(s): lots_max.
    sim_time: float
        Simulation time. Default: 60.
    lots_max: int
        Maximum amount of lots to be processed/generated. Default: 1.
    '''
#------------------------------------------------------------
# SIMULATION METHOD SELECTOR
#------------------------------------------------------------
    if (method == "Time"):
        env.process(generator.gen(env, method, None))
        env.run(until=sim_time)
    elif (method == "n-processed"):
        event = env.process(generator.gen(env, method, lots_max))
        env.run(event)
    elif (method == "n-generated"):
        env.process(generator.gen(env, method, lots_max))
        env.run()
    else:
        raise RuntimeError(f"ChiPy Error: {method} is an unsupported simulation method.")
```

chipy.functions.prints.stats.py

```
#! usr/bin/python3

#=============================================================================
# ~ STATSPRINTER ~
#   Prints Statistics Of Simulation
#=============================================================================
#============================================================
# IMPORTS
#============================================================

import simpy
import numpy as np
import pandas as pd
import random as rd
import os
import matplotlib.pyplot as plt
import chipy as cp

#============================================================
# METHOD(S)
#============================================================

def stats(lots: list, t_start: list, t_end: list, sim_time: float, savename: str = '
    phi_data.csv', generate_table: bool = True):
#------------------------------------------------------------
# HELP
#------------------------------------------------------------
    '''
    #------------------------------------------------------------
    # INFO
    #------------------------------------------------------------

    Prints a summary of the simulation in the console. Exports table to a .csv file.

    #------------------------------------------------------------
    # ARGUMENTS
    #------------------------------------------------------------

    lots: list
        A list containing all processed lots with priority levels as a tuple.
    t_start: list
        A list with a lots entry time in the simulation.
    t_exit: list
        A list with a lots exit time in the simulation as a tuple.
    sim_time: float
        The duration of the simulation. When running for n lots, this is the latest time
     value of t_exit.
    savename: str
        The savename and file extension of the generated table. Default: 'phi_data.csv'
    generate_table: bool
        True if a DataFrame Table should be generated. Slightly increases computing time
    . Default: True.
    '''
#------------------------------------------------------------
# LIST INITIATION
#------------------------------------------------------------
    lots_num: list  = []
    start: list     = []
    end: list       = []
    phi: list       = []
    df_obj: list    = []
#------------------------------------------------------------
# DATA PREPARATION
#------------------------------------------------------------
    for j in range(0,len(lots),1):
        lots_num.append(lots[j][0])
    for j in lots_num:
        itm = [item for item in t_start if item[0] == j]
        start.append(itm[0][1])
    for j in range(0,len(start),1):
        end.append(t_end[j][1])
    for j in range(0,len(lots),1):
        phi.append(end[j] - start[j])
```

```
71      if (generate_table == True):
72          for j in range(0,len(lots),1):
73              df_obj.append([lots_num[j],phi[j],start[j]])
74          df_phi = pd.DataFrame(df_obj, columns=['Lot ID', 'Flow Time','Entry Time' ])
75  #------------------------------------------------------------
76  # PRINTS
77  #------------------------------------------------------------
78      print(f'')
79      print(f'============================================================')
80      print(f'SIMULATION SUMMARY')
81      print(f'============================================================')
82      print(f'')
83      print(f'The average flow time equals {np.mean(phi)}')
84      print(f'The (population) standard deviation of the flow time equals {np.std(phi)}')
85      print(f'The coefficient of variation of the flow time equals {np.std(phi)/np.nanmean
        (phi)}')
86      print(f'The amount of products that passed through the run environment is {len(phi)}
         in {sim_time} time units')
87      print(f'')
88      print(f'============================================================')
89      if (generate_table == False):
90          return None
91      print(f'FLOW TIME PER LOT ID')
92      print(f'============================================================')
93      print(f'')
94      print(df_phi.to_string(index=False))
95      if not os.path.exists('data'):
96          os.makedirs('data')
97      df_phi.to_csv("data/"+savename , index=False, header = False)
98      print(f'')
99      savedir = "data/"+savename
100     print(f"(This table is also saved as /{savedir})")
101     print(f'')
102     print(f'============================================================')
```

chipy.functions.visualisation.flowtimediagram.py

```
1  #! usr/bin/python3
2
3  #================================================================================
4  # ~ FLOWTIME DIAGRAM GENERATOR ~
5  #   Generates a flowtime diagran for lots flowing trough a production system
6  #================================================================================
7  #============================================================
8  # REQUIRED IMPORTS
9  #============================================================
10 import numpy as np
11 import os
12 import sys
13 import matplotlib.pyplot as plt
14 #============================================================
15 # MODULE(S)
16 #============================================================
17 def flowtime_diagram(lots: list, t_start: list, t_end: list, sim_time: float, savename:
       str = "flowtime.png", title: str = "Flowtime of lots", xlabel: str = "Time", ylabel:
        str = "Lot ID" , color: str = "lightblue") -> None:
18 #------------------------------------------------------------
19 # HELP
20 #------------------------------------------------------------
21     '''
22     #------------------------------------------------------------
23     # INFO
24     #------------------------------------------------------------
25
26     Generates a flowtime diagram of all lots that completed the simulation.
27
28     #------------------------------------------------------------
29     # ARGUMENTS
30     #------------------------------------------------------------
31
32     lots: list
33         A list containing all processed lots with priority levels as a tuple.
34     t_start: list
35         A list with a lots entry time in the simulation.
```

```
36      t_exit: list
37          A list with a lots exit time in the simulation as a tuple.
38      sim_time: float
39          The duration of the simulation. When running for n lots, this is the latest time
         value of t_exit.
40      savename: str
41          The savename and file extension of the generated plot. Default: "flowtime.png".
42      title: str
43          The title of the generated plot. Default: "Flowtime of lots".
44      xlabel: str
45          The label of the horizontal axis of the generated plot. Default: "Time".
46      ylabel: str
47          The label of the vertical axis of the generated plot. Default: "Lot ID".
48      color: str
49          The color of the generated plot. Default: "lightblue".
50      '''
51  #------------------------------------------------------------
52  # RUNTIME ERRORS
53  #------------------------------------------------------------
54      if lots == [] :
55              raise RuntimeError("ChiPy Error: Lots list is empty")
56
57      if t_start == [] :
58              raise RuntimeError("ChiPy Error: t_start list is empty")
59
60      if t_end == [] :
61              raise RuntimeError("ChiPy Error: t_end list is empty")
62  #------------------------------------------------------------
63  # DATA PREPARATION
64  #------------------------------------------------------------
65      y_label: list = []
66      lots_num: list = []
67      start: list = []
68      end: list = []
69      for j in range(0,len(lots),1):
70          lots_num.append(lots[j][0])
71      for j in lots_num:
72          itm = [item for item in t_start if item[0] == j]
73          start.append(itm[0][1])
74      for j in range(0,len(start),1):
75          end.append(t_end[j][1])
76      savedir = "plots/" + savename
77  #------------------------------------------------------------
78  # FLOWTIME DIAGRAM
79  #------------------------------------------------------------
80      fig, ax = plt.subplots()
81      for j in range(0,len(lots),1):
82          ax.broken_barh([(start[j] , end[j]-start[j])], (1*j,1), color = color)
83          y_label.append("Lot %s" % (lots[j][0]))
84      fig.set_figheight(10)
85      ax.set_ylim(0,len(lots))
86      ax.set_xlim(0, sim_time)
87      ax.set_xlabel(xlabel)
88      ax.set_ylabel(ylabel)
89      ax.set_yticks(np.arange(0,len(lots),1))
90      ax.set_yticklabels(y_label)
91      ax.set_title(title)
92      ax.grid(True)
93      fig.tight_layout()
94  #------------------------------------------------------------
95  # TEX CONVERSION
96  #------------------------------------------------------------
97      if not os.path.exists('plots'):
98          os.makedirs('plots')
99      plt.savefig("plots/" + savename)
100     plotdir = None
101     if (sys.platform == "linux" or sys.platform == "linux2"):
102         for file in os.listdir("./plots"):
103             if file.endswith(savename[:savename.index(".")] + ".svg"):
104                 plotdir = (os.path.join("./plots", file))
105         if (plotdir != None):
106             try:
107                 os.system("inkscape -D %s  -o %s --export-latex" % (plotdir, "./plots/"
    + savename[:savename.index(".")] + ".pdf"))
108                 print(f".{savedir} .pdf_tex conversion succesful")
109             except:
```

```
                pass
        else:
            pass
#-------------------------------------------------------------
# CONFIRMATIONS
#-------------------------------------------------------------
    print(f"Flowtime Diagram saved as .{savedir}")
```

chipy.functions.visualisation.stationstats.py

```
#! usr/bin/python3
#=============================================================================
# ~ STATION STATISTICS GENERATOR ~
#   Generates station specific data visualisations
#=============================================================================
#=============================================================
# IMPORTS
#=============================================================
import numpy as np
import os
import sys
import matplotlib.pyplot as plt
#=============================================================
# MODULE(S)
#=============================================================
def station_stats(stations, station_name: str, sim_time: float, show_avg: bool = False,
    savename: str = "stats.png", title: str = "Statistics of ", xlabel: str = "Time",
    color: str = "lightblue") -> None:
#-------------------------------------------------------------
# HELP
#-------------------------------------------------------------
    '''
    #-----------------------------------------------------------
    # INFO
    #-----------------------------------------------------------

    Generates a flowtime diagram of all lots that completed the simulation.
    Note that for svg to pdf_tex conversion, 'Inkscape' has to be installed.

    #-----------------------------------------------------------
    # ARGUMENTS
    #-----------------------------------------------------------

    stations: dict
        A dictionary containing all chipy.station objects used in the simulation
    station_name: str
        The exact name of the station (used in the stations dict) the plot should be
    generated for.
    sim_time: float
        The duration of the simulation. When running for n lots, this is the latest time
     value of t_exit.
    show_avg: bool
        Display a red average line in the plot when True. Default: False.
    savename: str
        The savename and file extension of the generated plot. Default: "stats.png".
    title: str
        The title of the generated plot. When the argument is unchanged, the station
    name is dynamically added to the title. Default: "Statistics of ".
    xlabel: str
        The label of the horizontal axis of the generated plot. Default: "Time".
    color: str
        The color of the generated plot. Default: "lightblue".
    '''
#-------------------------------------------------------------
# FIGURE INITIATION
#-------------------------------------------------------------
    station = stations[station_name]
    fig, ax = plt.subplots(2,2)
    if (title == "Statistics of "):
        title = title + station_name
    if (show_avg == True):
        fig.suptitle(title+"\n (Red Line is Average)")
    else:
        fig.suptitle(title)
```

```
60      savedir = "plots/" + savename
61  #-------------------------------------------------------------
62  # FAIL PERIODS
63  #-------------------------------------------------------------
64      if ("simtime" not in station.fix_list) and (len(station.fail_list) == len(station.
        fix_list) + 1):
65          # FIX TO ADD FAILURE COLUMN OF FIX THAT COMPLETES AFTER 'simtime'
66          station.fix_list.append(sim_time)
67      for j in range(0,len(station.fix_list),1):
68          ax[0,0].broken_barh([(station.fail_list[j] , station.fix_list[j]-station.
        fail_list[j])], (0.1,0.2), color = color)
69      ax[0,0].set_xlim(0, sim_time)
70      ax[0,0].set_ylim(0.2)
71      ax[0,0].set_xlabel(xlabel)
72      ax[0,0].set_yticks([])
73      ax[0,0].set_title("Periods of Failure")
74      ax[0,0].grid(True)
75  #-------------------------------------------------------------
76  # BUFFERSIZE PLOT
77  #-------------------------------------------------------------
78      time = []
79      size = []
80      for j in range(0,len(station.buffersize_list),1):
81          time.append(station.buffersize_list[j][0])
82          size.append(station.buffersize_list[j][1])
83      if "simtime" not in time:
84          # FIX TO COMPLETE POST STEP LINE IN MATPLOTLIB
85          time.append(sim_time)
86          size.append(size[-1])
87      size_avg = [np.mean(size)]*(len(time))
88      ax[1,0].step(time,size, where="post")
89      if (show_avg == True):
90          ax[1,0].step(time,size_avg, color="red", where="post")
91      ax[1,0].set_xlim(0, sim_time)
92      ax[1,0].set_xlabel(xlabel)
93      ax[1,0].set_ylabel("Amount of lots [-]")
94      ax[1,0].set_title("Queuesize (cap =" + str(station.queuesize) + ")")
95      ax[1,0].grid(True)
96      if (sum(size) == 0):
97          ax[1,0].set_yticks([0])
98      fig.tight_layout()
99  #-------------------------------------------------------------
100 # OCCUPANCY RATE PLOT
101 #-------------------------------------------------------------
102     time = []
103     occupancyrate = []
104     for j in range(0,len(station.occupancyrate_list),1):
105         time.append(station.occupancyrate_list[j][0])
106         occupancyrate.append(station.occupancyrate_list[j][1])
107     if "simtime" not in time:
108         # FIX TO COMPLETE POST STEP LINE IN MATPLOTLIB
109         time.append(sim_time)
110         occupancyrate.append(occupancyrate[-1])
111     occupancyrate_avg = [np.mean(occupancyrate)]*(len(time))
112     ax[0,1].step(time,occupancyrate, where = "post")
113     if (show_avg == True):
114         ax[0,1].step(time,occupancyrate_avg, color="red")
115     ax[0,1].set_xlim(0, sim_time)
116     ax[0,1].set_xlabel(xlabel)
117     ax[0,1].set_ylabel("Occupancy Rate [%]")
118     ax[0,1].set_title("Occupancy Rate (cap =" + str(station.cap) + ")")
119     ax[0,1].grid(True)
120     fig.tight_layout()
121 #-------------------------------------------------------------
122 # WIP PLOT
123 #-------------------------------------------------------------
124     time = []
125     w = []
126     w_avg = []
127     for j in range(0,len(station.wip_list),1):
128         time.append(station.wip_list[j][0])
129         w.append(station.wip_list[j][1])
130         w_avg.append(station.avgwip_list[j][1])
131     if "simtime" not in time:
132         # FIX TO COMPLETE POST STEP LINE IN MATPLOTLIB
133         time.append(sim_time)
```

```
134         w.append(w[-1])
135         w_avg.append(w_avg[-1])
136     ax[1,1].step(time,w,  where = "post")
137     if (show_avg == True):
138         ax[1,1].step(time,w_avg, color="red",  where = "post")
139     ax[1,1].set_xlim(0, sim_time)
140     ax[1,1].set_xlabel(xlabel)
141     ax[1,1].set_ylabel("w [lots]")
142     ax[1,1].set_title("WIP")
143     ax[1,1].grid(True)
144     fig.tight_layout()
145 #------------------------------------------------------------
146 # TEX CONVERSION
147 #------------------------------------------------------------
148     if not os.path.exists('plots'):
149         os.makedirs('plots')
150     plotdir = None
151     if (savename[:savename.index(".")] == "stats"):
152         savename = station_name.replace(" ", "") + "_" + savename
153     plt.savefig("plots/" + savename)
154     plotdir = None
155     if (sys.platform == "linux" or sys.platform == "linux2"):
156         for file in os.listdir("./plots"):
157             if file.endswith(savename[:savename.index(".")] + ".svg"):
158                 plotdir = (os.path.join("./plots", file))
159         if (plotdir != None):
160             try:
161                 os.system("inkscape -D %s  -o %s --export-latex" % (plotdir, "./plots/"
    + savename[:savename.index(".")] + ".pdf"))
162                 print(f".{savedir} .pdf_tex conversion succesful")
163             except:
164                 pass
165         else:
166             pass
167 #------------------------------------------------------------
168 # CONFIRMATIONS
169 #------------------------------------------------------------
170     print(f"Statistiscs Diagram of {station_name} saved as .{savedir}")
```

# APPENDIX C

# CASE STUDIED WITH $\chi PY$

This appendix contains the *Jupyter Notebook* of the *" Modeling and simulation of an autonomous vehicle storage and retrieval system"* case, discussed in chapter 5.

## C.1 $\chi Py$ Notebook model for *" Modeling and simulation of an autonomous vehicle storage and retrieval system"*

This is an LaTeX export of Jupyter Notebook that simulates a "Modeling and simulation of an autonomous vehicle storage and retrieval system" in $\chi Py$. It is the intermediate assignment of the third year course Analysis of Production Systems of the BSc Mechanical Engineering at the Eindhoven University of Technology. It is used to test the capabilities of the written $\chi Py$ *Python* package as alternative to $\chi 3$.

Each new part starts with a `%reset -f` line, removing all variables and imports to avoid conflicts with previous modeling parts. This also means that part can be executed separately.

Each part is verified by comparing its output with the equivalent $\chi 3$ specification. The result of this comparison is favorable: $\chi Py$ is perfectly able to model the AVS/RS plant with zero error in deterministic cases. If stochasticity is introduced, the resulting outputs of multiple simulations in both languages are near-identical.

### Part 1 and 2 | Generator and Demand Buffer

[1]:
```
%reset -f
```

**Imports**

Imports Required Packages

[2]:
```
import sys
!{sys.executable} -m pip install simpy
import simpy
import numpy as np
import pandas as pd
import random as rd
import os
import sys
import matplotlib.pyplot as plt
import chipy as cp
```

### Model Variables

[3]:
```
arrive: float = 70                                          # Rate of arrival
depth: int = 55                                             # The number of columns
number_of_orders: int = 10000                               # Number of orders to process␣
 ↪
totes: list = []                                            # List of processed orders
inf = float("inf")                                          # Infinity
env = cp.environment()                                      # Initiate Environment
```

### Workstations

[4]:
```
workstations = {
            "Vehicle": cp.Station(env = env, te = lambda: 0)
    }
```

### Run Environment (runenv)

[5]:
```
t_arr: list = []      # Arrive times of totes
t_exit: list = []     # Exit times of totes
def runenv(env, tote, workstations):
    arr = env.now
    t_arr.append([tote[0], arr])

    yield env.process(workstations["Vehicle"].proc(tote))

    end = env.now
    t_exit.append([tote[0], end])
    totes.append(tote)
```

### Generator

(Note: *χPy* works by default with the term `lots` for an item going through a system.)

To add column data to each individual tote, the `chipy.Generator` class has to be inherited in a new class, that simply adds another piece of data to the lot tuple.

Look inside the source code for more details. Content shown here is copied form there and slightly altered to match the case. Since the entire assignment never tests machine failure, the fail initiation code is left out the newly defined gen method.

[6]:
```
class ToteGenerator(cp.Generator):
    def __init__(self, env, ta, runenv, lots, stations, priority = lambda: 0, column =␣
 ↪lambda: 0):
        super().__init__(env, ta, runenv, lots, stations, priority)
        self.column = column
    def gen(self, env, method = "Time", lots_max = None) -> None:
        if (method == "Time"):
            cond = lambda: 1
        elif (method == "n-processed"):
            cond = lambda: (len(self.lots) < lots_max)
        elif (method == "n-generated"):
            cond = lambda: (lot_ID < lots_max)
        lot_ID: int = 0
        while (cond()):
            lot = (lot_ID, self.priority(), self.column())
            self.env.process(self.runenv(env, lot, self.stations))
            yield env.timeout(self.ta())
            lot_ID += 1
```

[7]:
```
G = ToteGenerator(
            env = env,
            ta = lambda: rd.expovariate((arrive)**(-1)),
            stations = workstations,
            runenv = runenv,
            lots = totes,
            column = lambda: rd.randint(1, depth)          #See https://numpy.org/doc/
 ↪stable/reference/random/generated/numpy.random.randint.html#numpy.random.randint
)
```

**Simulation**

[8]:
```
cp.simulate(env = env ,
            generator = G,
            method = "n-generated",
            lots_max = number_of_orders)
```

**Output**

(Note: χ*Py* works by default with the term `lots` for an item going through a system.)

[9]:
```
cp.stats(lots = totes,
        t_start = t_arr,
        t_end = t_exit,
        sim_time = t_exit[-1][1],
        generate_table = False)
```

```
============================================================
SIMULATION SUMMARY
============================================================

The average flow time equals 0.0
The standard deviation of the flow time equals 0.0
The variability of the flow time equals nan
The amount of products that passed through the run environment is 10000 in
702968.5803835165 time units

============================================================
```

## Part 3 | Vehicle as machine

[10]:
```
%reset -f
```

**Imports**

[11]:
```
import sys
#!{sys.executable} -m pip install simpy
#import simpy
import numpy as np
import pandas as pd
import random as rd
import os
import sys
import matplotlib.pyplot as plt
import chipy as cp
```

### Model Variables

[12]:
```
arrive: float = 70                                          # Rate of arrival
depth: int = 55                                             # The number of columns
number_of_orders: int = 10000                               # Number of orders to process␣
 ↪
totes: list = []                                            # List of processed orders
inf = float("inf")                                          # Infinity
env = cp.environment()                                      # Initiate Environment
```

### Workstations

Now the Vehicle will be modeled as a machine. This means a `te` value will now be assigned according to the assignment. This is a determinsitic time of 40.

[13]:
```
workstations = {
            "Vehicle": cp.Station(env = env, te = lambda: 40)
    }
```

### Run Environment (`runenv`)

[14]:
```
t_arr: list = []      # Arrive times of totes
t_exit: list = []     # Exit times of totes
def runenv(env, tote, workstations):
    arr = env.now
    t_arr.append([tote[0], arr])
    yield env.process(workstations["Vehicle"].proc(tote))
    end = env.now
    t_exit.append([tote[0], end])
    totes.append(tote)
```

### Generator

[15]:
```
class ToteGenerator(cp.Generator):
    def __init__(self, env, ta, runenv, lots, stations, priority = lambda: 0, column =␣
 ↪lambda: 0):
        super().__init__(env, ta, runenv, lots, stations, priority)
        self.column = column
    def gen(self, env, method = "Time", lots_max = None) -> None:
        if (method == "Time"):
            cond = lambda: 1
        elif (method == "n-processed"):
            cond = lambda: (len(self.lots) < lots_max)
        elif (method == "n-generated"):
            cond = lambda: (lot_ID < lots_max)
        lot_ID: int = 0
        while (cond()):
            lot = (lot_ID, self.priority(), self.column())
            self.env.process(self.runenv(env, lot, self.stations))
            yield env.timeout(self.ta())
            lot_ID += 1
```

[16]:
```
G = ToteGenerator(
            env = env,
            ta = lambda: rd.expovariate((arrive)**(-1)),
            stations = workstations,
            runenv = runenv,
            lots = totes,
            column = lambda: rd.randint(1, depth)          #See https://numpy.org/doc/
 ↪stable/reference/random/generated/numpy.random.randint.html#numpy.random.randint
```

```
)
```

**Simulation**

[17]:
```
cp.simulate(env = env ,
            generator = G,
            method = "n-generated",
            lots_max = number_of_orders)
```

**Output**

[18]:
```
cp.stats(lots = totes,
        t_start = t_arr,
        t_end = t_exit,
        sim_time = t_exit[-1][1],
        generate_table = False)
```

```
============================================================
SIMULATION SUMMARY
============================================================

The average flow time equals 67.24724098339088
The standard deviation of the flow time equals 38.041717611862
The variability of the flow time equals 0.5656993068497452
The amount of products that passed through the run environment is 10000 in
703825.1705427187 time units

============================================================
```

## Part 4 | Vehicle accurately modeled

[19]:
```
%reset -f
```

**Imports**

[20]:
```
import sys
#!{sys.executable} -m pip install simpy
#import simpy
import numpy as np
import pandas as pd
import random as rd
import os
import sys
import matplotlib.pyplot as plt
import chipy as cp
from math import sqrt
```

**Model Variables**

[21]:
```
arrive: float = 70                          # Rate of arrival
depth: int = 55                             # The number of columns
number_of_orders: int = 10000               # Number of orders to process
lv: float = 3.0                             # Time to load/unload the vehicle
dv: float = 0.5                             # Unit width clearance
vmaxv: float = 1.5                          # Maximum velocity of the vehicle
av: float = 1.0                             # Acceleration/deceleration of the vehicle
totes: list = []                            # List of processed orders
inf = float("inf")                          # Infinity
```

```
env = cp.environment()                          # Initiate Environment
```

### Workstations

Since the process time `te` of a tote is now dependent on its column number, parsing the value through at its initiation is not suitable for this case. Luckily, *χPy* supports lot-specifc process times out of the box. The process time can be left undetermined at initiation, but needs to be passed to thr station once a lot enters, which is done in the `runenv`.

[22]:
```
workstations = {
            "Vehicle": cp.Station(env = env)
    }
```

### Run Environment (`runenv`)

[23]:
```
t_arr: list = []      # Arrive times of totes
t_exit: list = []     # Exit times of totes
def runenv(env, tote, workstations):
    arr = env.now
    t_arr.append([tote[0], arr])
#-------------------------------------------------------------
# VEHICLE
#-------------------------------------------------------------
    dmin = (vmaxv**(2))/(av)
    d = tote[2]*dv
    if (d <= dmin):
        t_travel = sqrt((4*d)/(av))
    elif (d > dmin):
        t_travel = (d-dmin)/(vmaxv) + (2*vmaxv)/(av)
    te = 2*(t_travel + lv)

    yield env.process(workstations["Vehicle"].proc(lot=tote,te=lambda: te))
    end = env.now
    t_exit.append([tote[0], end])
    totes.append(tote)
```

### Generator

[24]:
```
class ToteGenerator(cp.Generator):
    def __init__(self, env, ta, runenv, lots, stations, priority = lambda: 0, column =␣
 ↪lambda: 0):
        super().__init__(env, ta, runenv, lots, stations, priority)
        self.column = column
    def gen(self, env, method = "Time", lots_max = None) -> None:
        if (method == "Time"):
            cond = lambda: 1
        elif (method == "n-processed"):
            cond = lambda: (len(self.lots) < lots_max)
        elif (method == "n-generated"):
            cond = lambda: (lot_ID < lots_max)
        lot_ID: int = 0
        while (cond()):
            lot = (lot_ID, self.priority(), self.column())
            self.env.process(self.runenv(env, lot, self.stations))
            yield env.timeout(self.ta())
            lot_ID += 1
```

[25]:
```
G = ToteGenerator(
            env = env,
            ta = lambda: rd.expovariate((arrive)**(-1)),
```

```
            stations = workstations,
            runenv = runenv,
            lots = totes,
            column = lambda: rd.randint(1, depth)          #See https://numpy.org/doc/
 ↪stable/reference/random/generated/numpy.random.randint.html#numpy.random.randint
)
```

**Simulation**

[26]:
```
cp.simulate(env = env ,
            generator = G,
            method = "n-generated",
            lots_max = number_of_orders)
```

**Output**

[27]:
```
cp.stats(lots = totes,
        t_start = t_arr,
        t_end = t_exit,
        sim_time = t_exit[-1][1],
        generate_table = False)
```

```
============================================================
SIMULATION SUMMARY
============================================================

The average flow time equals 38.17907824305325
The standard deviation of the flow time equals 20.861513117667798
The variability of the flow time equals 0.5464121733076044
The amount of products that passed through the run environment is 10000 in
703804.3681310746 time units

============================================================
```

## Part 5 + 6 | Lift as two stage machine with single tier buffer

- For **PART 5**: `bc = inf`
- For **PART 6**: `bc = 1`

[28]:
```
%reset -f
```

**Imports**

[29]:
```
import sys
#!{sys.executable} -m pip install simpy
#import simpy
import numpy as np
import pandas as pd
import random as rd
import os
import sys
import matplotlib.pyplot as plt
import chipy as cp
from math import sqrt
```

### Model Variables

[30]:
```
inf = float("inf")                                  # Infinity
arrive: float = 70                                  # Rate of arrival
depth: int = 55                                     # The number of columns
lv: float = 3.0                                     # Time to load/unload the vehicle
dv: float = 0.5                                     # Unit width clearance
vmaxv: float = 1.5                                  # Maximum velocity of the vehicle
av: float = 1.0                                     # Acceleration/deceleration of the vehicle
ll: float = 2.0                                     # Time to load/unload the lift
tl: float = 23.0                                    # Time to move the lift to its destination
bc: int = 1                                         # Buffer capacity
totes: list = []                                    # List of processed orders
env = cp.environment()                              # Initiate Environment
```

### Workstations

The way of modelling the lift is slightly different compared to the χ3 way. Since χ*Py* works with the concept of resource occupation (and χ3 mainly with channels) we will model the lift with an `te` of `2(ll+tl)`. This way, the part of "informing the lift to go to tier x" already counts as resource occupation, since a single tote specifically requests the list pick it up.

[31]:
```
workstations = {
            "Vehicle": cp.Station(env = env, te = lambda: 0),
            "Lift": cp.Station(env = env, te = lambda: 2*(ll + tl), queuesize = bc)
    }
```

### Run Environment (`runenv`)

To make finite buffers work properly (a generator cannot hold products), the vehicle with process time zero is put inbetween, that has a proper release hold control action when the upcoming buffer is full.

[32]:
```
t_arr: list = []      # Arrive times of totes
t_exit: list = []     # Exit times of totes

def runenv(env, tote, workstations):
    arr = env.now
    t_arr.append([tote[0], arr])

#-------------------------------------------------------------
# VEHICLE (NOTE: NOT MODELLED IN THIS PART!)
#-------------------------------------------------------------

    yield env.process(workstations["Vehicle"].proc(lot=tote,us=workstations["Lift"]))

#-------------------------------------------------------------
# LIFT
#-------------------------------------------------------------

    print("Generator: Tote %s Informed Lift to go to tier %s at time %s" %␣
 ↪(tote[0],tote[2], env.now))
    yield env.process(workstations["Lift"].proc(tote))
    print("Exit: Tote %s has completely left the Lift and has been received by the Exit␣
 ↪at time %s" % (tote[0], env.now));

    end = env.now
    t_exit.append([tote[0], end])
    totes.append(tote)
```

### Generator

In this part, the generator works quite a bit different compared to the other cases. See the assignment template `chi` code. Note that the Lift has to travel to pickup the tote first, then travels back to drop the tote at the desired location. Then it waits till the new tote arrives and moves back to pick it up. Since we will eventually need columns, they are still left in the generator, in contrast to the `chi` template shown in the assignment.

Note that the case here is modelled slightly different. Where the $\chi 3$ model starts the delay once the lift has completed its process, the delay in the case of the `ToteGenerator` is done per tote generation, like a rate of arrival. If one wants to wait till the lift has completed its process, the `self.env.process(self.runenv(env, lot, self.stations))` statement has to be yielded. The output still seems to be a bit off in the favour of the $\chi Py$'s implementation: It properly does the second delay of the list (`10`) instead of a delay of zero (in the case of the $\chi 3$ template). Note however, that using the provided delay list (that $\chi Py$ handles properly in comparison to $\chi 3$) the buffer capacity of the Lift buffer is never reached (since the process time of the lift is determisnistically 23). This can indeed be seen when running the simulation for both values of `bc` (`inf` and `1`)

Also note that the second print the $\chi 3$ tempate has is not present here. It can, however, be added by inheriting the `chipy.Station` `class` and rewriting the `run` method, splitting its timeout in 2 separate timeouts of `(ll + tl)` and to print the missing statment between the two timeouts.

[33]:
```
class ToteGenerator(cp.Generator):
    def __init__(self, env, ta, runenv, lots, stations, priority = lambda: 0, column =␣
 ↪lambda: 0, tier = lambda: 1):
        super().__init__(env, ta, runenv, lots, stations, priority)
        self.column = column
        self.tier = tier
        self.delays = [1.0, 10.0, 100.0, 100.0]

    def gen(self, env, lots_max = "NaN") -> None:
        lot_ID: int = 0
        while (len(self.delays) > 0):
            yield env.timeout(self.delays[0])
            self.delays = self.delays[1:]
            lot = (lot_ID, self.column(), self.tier())
            self.env.process(self.runenv(env, lot, self.stations))
            lot_ID += 1
```

[34]:
```
G = ToteGenerator(
            env = env,
            ta = lambda: rd.expovariate((arrive)**(-1)),
            stations = workstations,
            runenv = runenv,
            lots = totes,
            column = lambda: rd.randint(1, depth+1),
            tier = lambda: 1
)
```

### Simulation

In this case, the simulation method does not fit any of the three methods the `cp.simulate` method allows, since the `ToteGenerator` is altered such to only handle the specific required case. Therefore, the environments `run` method has to be called manually.

[35]:
```
event = env.process(G.gen(env))
env.run()
```

```
Generator: Tote 0 Informed Lift to go to tier 1 at time 1.0
Exit: Tote 0 has completely left the Lift and has been received by the Exit at
time 51.0
Generator: Tote 1 Informed Lift to go to tier 1 at time 51.0
Exit: Tote 1 has completely left the Lift and has been received by the Exit at
time 101.0
Generator: Tote 2 Informed Lift to go to tier 1 at time 111.0
```

```
Exit: Tote 2 has completely left the Lift and has been received by the Exit at
time 161.0
Generator: Tote 3 Informed Lift to go to tier 1 at time 211.0
Exit: Tote 3 has completely left the Lift and has been received by the Exit at
time 261.0
```

**Output**

Outputting is handled via `runenv` in this case. See the print of the Simulation section. As extra, to check finite buffer behaviour, the `stats` method will also be called below. Note that the second tote (ID = 1) has a larger flow time, since the lift still needs 40 time units to process the first tote (ID = 0). This is thus expected behaviour.

[36]:
```
cp.stats(lots = totes,
        t_start = t_arr,
        t_end = t_exit,
        sim_time = t_exit[-1][1],
        generate_table = True)
```

```
============================================================
SIMULATION SUMMARY
============================================================

The average flow time equals 60.0
The standard deviation of the flow time equals 17.320508075688775
The variability of the flow time equals 0.2886751345948129
The amount of products that passed through the run environment is 4 in 261.0
time units

============================================================
FLOW TIME PER LOT ID
============================================================

 Lot ID  Flow Time  Entry Time
      0       50.0         1.0
      1       90.0        11.0
      2       50.0       111.0
      3       50.0       211.0

(This table is also saved as /data/phi_data.csv)

============================================================
```

## Part 7 | Buffer with multiple tiers

The figure below shows the model with tiers.

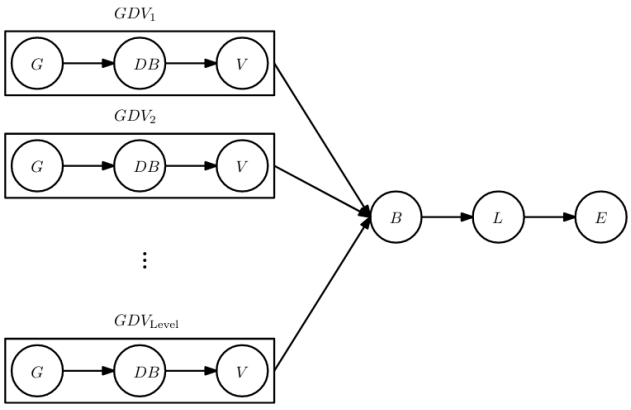


Figure C.1: Block diagram of the AVS/RS

(Note: in Part 7, the vehicle is still left out!)

Only 1 generator will (and can) be used, due to the nature of $\chi Py$. Each Tier will be modelled as a separate dictionary, all stored in a list. Since a `tote` is a tuple with a field that describes its tier, `runenv` can select the proper dict from the Tiers list to use. This also immediatly shows the power of the `runenv` method: as long as it is possible in Python, a lot (in this case tote) can take any path desirable!

The buffer will be modeled as a shared buffer with **total** capacity `bc`. Therefore this part is a bit different compared to its $\chi 3$ counterpart. Due to the resource concept $\chi Py$ is build upon, a buffer (queue) is always linked to a station's process, and cannot be decoupled!

[37]:
```
%reset -f
```

### Imports

[38]:
```
from math import sqrt
import sys
import simpy
import numpy as np
import pandas as pd
import random as rd
import os
import sys
import matplotlib.pyplot as plt
import chipy as cp
```

### Model Variables

[39]:
```
inf = float("inf")                          # Infinity
arrive: float = 70                          # Rate of arrival
depth: int = 55                             # The number of columns
lv: float = 3.0                             # Time to load/unload the vehicle
dv: float = 0.5                             # Unit width clearance
vmaxv: float = 1.5                          # Maximum velocity of the vehicle
av: float = 1.0                             # Acceleration/deceleration of the vehicle
ll: float = 2.0                             # Time to load/unload the lift
tl: float = 23.0                            # Time to move the lift to its destination
bc: int = 1                                 # Buffer capacity
Level: int = 2                              # The number of tiers
totes: list = []                            # List of processed orders
env = cp.environment()                      # Initiate Environment
```

### Workstations

First, a "shared" lift will be initiated. Next, a single tier is formed using a `dict` (like the general workstation dictionaries). This dictionary will be re-initiated `Level` times. Since the Lift is defined before this loop, its Generator Object ID will be the same in each Tier. In essence, the lift is therefore shared over all tiers.

In contrast to the assignment, the Vehicle will already be implemented in the tiers. Its buffer length is `inf` and `te = 0`. The reason why it is still already implemented is to test of the concept of sending totes in the desired tiers actually works.

The last three prints are to check if each tier is initiated properly. Note that the lift object is identical in both tiers!

[40]:
```
Lift = cp.Station(env = env, te = lambda: 2*(ll + tl), queuesize = bc)
Tiers = []

for j in range(0,Level,1):
    Tiers.append({
        "Vehicle": cp.Station(env = env, te = lambda: 0),
        "Lift": Lift
    })
```

```
print(len(Tiers))
print(Tiers[0])
print(Tiers[1])
```

```
2
{'Vehicle': <chipy.processes.station.Station object at 0x000001D25E1E4130>,
'Lift': <chipy.processes.station.Station object at 0x000001D25E1E4100>}
{'Vehicle': <chipy.processes.station.Station object at 0x000001D25E1E4250>,
'Lift': <chipy.processes.station.Station object at 0x000001D25E1E4100>}
```

**Run Environment (runenv)**

[41]:
```
t_arr: list = []      # Arrive times of totes
t_exit: list = []     # Exit times of totes

def runenv(env, tote, Tiers):
    arr = env.now
    t_arr.append([tote[0], arr])

    tier = tote[2] - 1

#-------------------------------------------------------------
# VEHICLE
#-------------------------------------------------------------

    yield env.process(Tiers[tier]["Vehicle"].proc(tote, Tiers[tier]["Lift"]))

#-------------------------------------------------------------
# LIFT
#-------------------------------------------------------------

    print("Generator: Tote %s Informed Lift to go to tier %s at time %s" %␣
 ↪(tote[0],tote[2], env.now))
    yield env.process(Tiers[tier]["Lift"].proc(tote))
    print("Exit: Tote %s has completely left the Lift and has been received by the Exit␣
 ↪at time %s" % (tote[0], env.now));

    end = env.now
    t_exit.append([tote[0], end])
    totes.append(tote)
```

**Generator**

Note that the behaviour of this generator is slightly different than the χ3 equivalent! Where the χ3 version generates 2 products per tier 1 time unit apart, this version takes the delays as usual and randomly generates a corresponding lot tier. For a similar approach to the χ3 variant, take a look at the generator of exersice 9. The way it handles per tier generation can be used here as well. The 1 time unit delay per tier generation can be added as a `env.timeout(1)` within the nested tier generation loop.

In Part 8, it is chosen to model the generator exactly as in the χ3 equivalent! In this exersice it has however been left as is, to model/test different generation methods and check if χ*Py* gives expected results in both cases!

[42]:
```
class ToteGenerator(cp.Generator):
    def __init__(self, env, ta, runenv, lots, stations, priority = lambda: 0, column =␣
 ↪lambda: 0, tier = lambda: 1):
        super().__init__(env, ta, runenv, lots, stations, priority)
        self.column = column
        self.tier = tier
        self.delays = [1.0, 10.0, 100.0, 100.0]
```

```
    def gen(self, env, lots_max = "NaN") -> None:
        lot_ID: int = 0
        while (len(self.delays) > 0):
            yield env.timeout(self.delays[0])
            self.delays = self.delays[1:]
            lot = (lot_ID, self.column(), self.tier())
            self.env.process(self.runenv(env, lot, self.stations))
            lot_ID += 1
```

[43]:
```
G = ToteGenerator(
            env = env,
            ta = lambda: rd.expovariate((arrive)**(-1)),
            stations = Tiers,
            runenv = runenv,
            lots = totes,
            column = lambda: rd.randint(1, depth+1),
            tier = lambda: rd.randint(1, Level)
)
```

**Simulation**

[44]:
```
event = env.process(G.gen(env))
env.run()
```

```
Generator: Tote 0 Informed Lift to go to tier 1 at time 1.0
Exit: Tote 0 has completely left the Lift and has been received by the Exit at
time 51.0
Generator: Tote 1 Informed Lift to go to tier 2 at time 51.0
Exit: Tote 1 has completely left the Lift and has been received by the Exit at
time 101.0
Generator: Tote 2 Informed Lift to go to tier 1 at time 111.0
Exit: Tote 2 has completely left the Lift and has been received by the Exit at
time 161.0
Generator: Tote 3 Informed Lift to go to tier 2 at time 211.0
Exit: Tote 3 has completely left the Lift and has been received by the Exit at
time 261.0
```

**Output**

Outputting is handled via `runenv` in this case. See the print of the Simulation section. As extra, The `stats` method will also be called below. Note that due to the chosen generation method, the output is the same as in Exersice 6.

[45]:
```
cp.stats(lots = totes,
        t_start = t_arr,
        t_end = t_exit,
        sim_time = t_exit[-1][1],
        generate_table = True)
```

```
============================================================
SIMULATION SUMMARY
============================================================

The average flow time equals 60.0
The standard deviation of the flow time equals 17.320508075688775
The variability of the flow time equals 0.2886751345948129
The amount of products that passed through the run environment is 4 in 261.0
time units

============================================================
```

```
FLOW TIME PER LOT ID
============================================================

 Lot ID  Flow Time  Entry Time
      0       50.0         1.0
      1       90.0        11.0
      2       50.0       111.0
      3       50.0       211.0

(This table is also saved as /data/phi_data.csv)

============================================================
```

## Part 8 | Lift accurately modelled

This is essentially the same as Part 7, but just like with the `Vehicle`, now `te` will be calculated (based on its velocity and acceleration profile) beforehand to model the lift accuratly.

[46]:
```
%reset -f
```

### Imports

[47]:
```
from math import sqrt
import sys
import simpy
import numpy as np
import pandas as pd
import random as rd
import os
import sys
import matplotlib.pyplot as plt
import chipy as cp
```

### Model Variables

[48]:
```
inf = float("inf")                          # Infinity
arrive: float = 70                          # Rate of arrival
depth: int = 55                             # The number of columns
lv: float = 3.0                             # Time to load/unload the vehicle
dv: float = 0.5                             # Unit width clearance
vmaxv: float = 1.5                          # Maximum velocity of the vehicle
av: float = 1.0                             # Acceleration/deceleration of the vehicle
ll: float = 2.0                             # Time to load/unload the lift
al: float = 7.0                             # Acceleration/deceleration of lift
dl: float = 0.8                             # Unit height clearance
vmaxl: float = 5.0                          # Maximum velocity of lift
bc: int = 1                                 # Buffer capacity
Level: int = 2                              # The number of tiers
totes: list = []                            # List of processed orders
env = cp.environment()                      # Initiate Environment
```

### Workstations

Note that in this part, the Vehicle is still not modelled. This will be done in Part 9, where all simulation bits are put together.

[49]:
```
Lift = cp.Station(env = env)
Tiers = []

for j in range(0,Level,1):
    Tiers.append({
```

```
        "Vehicle": cp.Station(env = env, te = lambda: 0),
        "Lift": Lift
    })
```

**Run Environment (runenv)**

[50]:
```
t_arr: list = []      # Arrive times of totes
t_exit: list = []     # Exit times of totes

def runenv(env, tote, Tiers):
    arr = env.now
    t_arr.append([tote[0], arr])

    tier = tote[2] - 1

#-------------------------------------------------------------
# VEHICLE
#-------------------------------------------------------------

    yield env.process(Tiers[tier]["Vehicle"].proc(tote, Tiers[tier]["Lift"]))

#-------------------------------------------------------------
# LIFT
#-------------------------------------------------------------

    dminl = (vmaxl**2)/(al)
    d_lift = (tote[2])*dl
    if (d_lift <= dminl):
        t_travel_lift = sqrt(4*d_lift/al)
    elif (d_lift > dminl):
        t_travel_lift = (d_lift-dminl)/vmaxl + (2*vmaxl)/al
    te_lift = 2*(ll + t_travel_lift)

    print("Generator: Tote %s Informed Lift to go to tier %s at time %s" %␣
↪(tote[0],tote[2], env.now))
    yield env.process(Tiers[tier]["Lift"].proc(lot = tote,te = lambda: te_lift))
    print("Exit: Tote %s has completely left the Lift and has been received by the Exit␣
↪at time %s" % (tote[0], env.now));

    end = env.now
    t_exit.append([tote[0], end])
    totes.append(tote)
```

**Generator**

Note that the behaviour of this generator is different compared to exersice 7, and now corresponds better with the generator of the *χPy* equivalent! Also the results are (obviously) the same, meaning that the tiers to their work properly!

[51]:
```
class ToteGenerator(cp.Generator):
    def __init__(self, env, ta, runenv, lots, stations, priority = lambda: 0, column =␣
↪lambda: 0, tier = lambda: 1):
        super().__init__(env, ta, runenv, lots, stations, priority)
        self.column = column
        self.tier = tier
        self.delays = [10.0, 100.0, 100.0, 100.0]

    def gen(self, env, lots_max = "NaN") -> None:
        lot_ID: int = 0
        while (len(self.delays) > 0):
```

```
            lot = (lot_ID, self.column(), self.tier[0])
            self.env.process(self.runenv(env, lot, self.stations))
            yield env.timeout(1)
            lot_ID += 1
            lot = (lot_ID, self.column(), self.tier[1])
            self.env.process(self.runenv(env, lot, self.stations))
            lot_ID += 1
            yield env.timeout(self.delays[0]-1)
            self.delays = self.delays[1:]
```

[52]:
```
G = ToteGenerator(
            env = env,
            ta = lambda: rd.expovariate((arrive)**(-1)),
            stations = Tiers,
            runenv = runenv,
            lots = totes,
            column = lambda: rd.randint(1, depth+1),
            tier = range(1, Level+1, 1)
)
```

**Simulation**

[53]:
```
event = env.process(G.gen(env))
env.run()
```

```
Generator: Tote 0 Informed Lift to go to tier 1 at time 0
Generator: Tote 1 Informed Lift to go to tier 2 at time 1
Exit: Tote 0 has completely left the Lift and has been received by the Exit at
time 5.352246807565627
Generator: Tote 2 Informed Lift to go to tier 1 at time 10.0
Generator: Tote 3 Informed Lift to go to tier 2 at time 11.0
Exit: Tote 1 has completely left the Lift and has been received by the Exit at
time 11.264612582500657
Exit: Tote 3 has completely left the Lift and has been received by the Exit at
time 17.176978357435686
Exit: Tote 2 has completely left the Lift and has been received by the Exit at
time 22.529225165001314
Generator: Tote 4 Informed Lift to go to tier 1 at time 110.0
Generator: Tote 5 Informed Lift to go to tier 2 at time 111.0
Exit: Tote 4 has completely left the Lift and has been received by the Exit at
time 115.35224680756562
Exit: Tote 5 has completely left the Lift and has been received by the Exit at
time 121.26461258250065
Generator: Tote 6 Informed Lift to go to tier 1 at time 210.0
Generator: Tote 7 Informed Lift to go to tier 2 at time 211.0
Exit: Tote 6 has completely left the Lift and has been received by the Exit at
time 215.35224680756562
Exit: Tote 7 has completely left the Lift and has been received by the Exit at
time 221.26461258250066
```

**Output**

Outputting is handled via `runenv` in this case. See the print of the Simulation section. As extra, The `stats` method will also be called below. Note that due to the chosen generation method, the output is the same as in Exersice 6. Again, look at exersice 9's generator to model it like the $\chi$3 equivalent.

[54]:
```
cp.stats(lots = totes,
        t_start = t_arr,
        t_end = t_exit,
        sim_time = t_exit[-1][1],
        generate_table = True)
```

```
============================================================
SIMULATION SUMMARY
============================================================

The average flow time equals 8.19459771157948
The standard deviation of the flow time equals 2.7375067054786246
The variability of the flow time equals 0.334062366674859
The amount of products that passed through the run environment is 8 in
221.26461258250066 time units

============================================================
FLOW TIME PER LOT ID
============================================================

 Lot ID  Flow Time  Entry Time
      0   5.352247         0.0
      1  10.264613         1.0
      3   6.176978        11.0
      2  12.529225        10.0
      4   5.352247       110.0
      5  10.264613       111.0
      6   5.352247       210.0
      7  10.264613       211.0

(This table is also saved as /data/phi_data.csv)

============================================================
```

## Part 9 | Entire system

In this final part, the entire system will be modelled with a "normal" `ta` (instead of the delay list). Since we work with a single Generator (since *χPy* only works with one), it has to be altered to generate totes at the correct rate for the correct Tier. This is done by extending the `chipy.Generator.gen` method.

[55]:
```
%reset -f
```

### Imports

[56]:
```
from math import sqrt
import sys
import simpy
import numpy as np
import pandas as pd
import random as rd
import os
import sys
import matplotlib.pyplot as plt
import chipy as cp
import time
```

### Model Variables

[57]:
```
inf = float("inf")                          # Infinity
lv: float = 3.0                             # Time to load/unload the vehicle
dv: float = 0.5                             # Unit width clearance
vmaxv: float = 1.5                          # Maximum velocity of the vehicle
av: float = 1.0                             # Acceleration/deceleration of the vehicle
ll: float = 2.0                             # Time to load/unload the lift
dl: float = 0.8                             # Unit height clearance
vmaxl: float = 5.0                          # Maximum velocity of lift
```

```
al: float = 7.0                                 # Acceleration/deceleration of lift
bc: int = inf                                   # Buffer capacity
arrive: float = 70                              # Rate of arrival
Level: int = 9                                  # The number of tiers
depth: int = 55                                 # The number of columns
number_of_orders: int = 1*10**(4)               # Number of orders to process
Tiers: list = []                                # List of individual tiers
totes: list = []                                # List of processed orders
env = cp.environment()                          # Initiate Environment
```

**Workstations**

Note that the througput $\delta$ of the `Vehicle` is the total rate of arrival divided by the amount of (parallel) tiers.

[58]:
```
Lift = cp.Station(env = env, queuesize = bc)

for j in np.arange(0,Level,1):
    Tiers.append({
        "Vehicle": cp.Station(env = env, delta = (arrive/Level)),
        "Lift": Lift
    })
print(len(Tiers))
print(Tiers[0])
print(Tiers[1])
```

```
9
{'Vehicle': <chipy.processes.station.Station object at 0x000001D25E1F85E0>,
'Lift': <chipy.processes.station.Station object at 0x000001D25E1F8250>}
{'Vehicle': <chipy.processes.station.Station object at 0x000001D25E1F8A60>,
'Lift': <chipy.processes.station.Station object at 0x000001D25E1F8250>}
```

**Run Environment (`runenv`)**

[59]:
```
t_arr: list = []      # Arrive times of totes
t_exit: list = []     # Exit times of totes
phi: list = []        # Phi list to calculate average flowtime per lot, like in chi

def runenv(env, tote, Tiers):
    arr = env.now
    t_arr.append([tote[0], arr])

    tier = tote[2] - 1
    # NOTE Dictionary Indexing starts at zero!

#-------------------------------------------------------------
# VEHICLE
#-------------------------------------------------------------

    dminv = (vmaxv**(2))/(av)
    d_vehicle = tote[1]*dv

    if (d_vehicle <= dminv):
        t_travel_vehicle = sqrt((4*d_vehicle)/(av))
    elif (d_vehicle > dminv):
        t_travel_vehicle = (d_vehicle-dminv)/(vmaxv) + (2*vmaxv)/(av)

    te_vehicle = 2*(t_travel_vehicle + lv)
    yield env.process(Tiers[tier]["Vehicle"].proc(lot = tote,te = lambda: te_vehicle,us =␣
 ↪Tiers[tier]["Lift"]))

#-------------------------------------------------------------
```

```
# LIFT
#-------------------------------------------------------------

    dminl = (vmaxl**2)/(al)
    d_lift = (tote[2])*dl
    # NOTE d = dl(i+1), i \in {0,Level-1}. tote[2] = Level
    if (d_lift <= dminl):
        t_travel_lift = sqrt(4*d_lift/al)
    elif (d_lift > dminl):
        t_travel_lift = (d_lift-dminl)/vmaxl + (2*vmaxl)/al
    te_lift = 2*(ll + t_travel_lift)
    yield env.process(Tiers[tier]["Lift"].proc(lot = tote,te = lambda: te_lift))

    end = env.now
    t_exit.append([tote[0], end])
    phi.append(end-arr)
    totes.append(tote)
```

**Generator**

Note that the used generator works slightly different than the generator in the χ3 equivalent. Tests have been done with a predictable deterministic arrival pattern that resulted in exactly the same results for both languages.

[60]:
```
class ToteGenerator(cp.Generator):
    def __init__(self, env, ta, runenv, lots, stations, priority = lambda: 0, column =␣
 ↪lambda: 0, tier = lambda: 1):
        super().__init__(env, ta, runenv, lots, stations, priority)
        self.column = column
        self.level = tier
        self.levels = range(1,self.level+1,1)
    def gen(self, env, method = "Time", lots_max = None) -> None:
        if (method == "Time"):
            cond = lambda: 1
        elif (method == "n-processed"):
            cond = lambda: (len(self.lots) < lots_max)
        elif (method == "n-generated"):
            cond = lambda: (lot_ID < lots_max)
        j: int = 0
        lot_ID: int = 0
        while (cond()):
            yield env.timeout(self.ta())
            lot = (lot_ID, self.column(), self.levels[j])
            self.env.process(self.runenv(env, lot, self.stations))
            lot_ID += 1
            j += 1
            if (j > (self.level-1)):
                j = 0
```

[61]:
```
G = ToteGenerator(
            env = env,
            ta = lambda: rd.expovariate((arrive)**(-1)),
            stations = Tiers,
            runenv = runenv,
            lots = totes,
            column = lambda: rd.randint(1, depth),
            tier = Level
)
```

**Simulation**

[62]:
```
start_sim = time.time()
cp.simulate(env = env ,
            generator = G,
            method = "n-generated",
            lots_max = number_of_orders)
print(f"It took {time.time()-start_sim} [s] to perform the simulation")
```

```
It took 3.565502405166626 [s] to perform the simulation
```

**Summary Output**

[63]:
```
cp.stats(lots = totes,
        t_start = t_arr,
        t_end = t_exit,
        sim_time = t_exit[-1][1],
        generate_table = False)
```

```
============================================================
SIMULATION SUMMARY
============================================================

The average flow time equals 34.969698714621906
The standard deviation of the flow time equals 10.78373141874099
The variability of the flow time equals 0.30837358670842596
The amount of products that passed through the run environment is 10000 in
691883.5892018371 time units

============================================================
```

Eindhoven University of Technology

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY