ProdSim

Release 0.0.1

Tom Fuchs

Sep 27, 2021

CONTENTS

ProdSim

ProdSim is a process-based discrete event simulation for production environments based on the SimPy framework. The package is designed to generate large high-resolution synthetic production data sets.

The characteristics of a production system are represented by three system components, namely machines, workpieces, and a factory. These components interact with one another on the following three system layers:

- logistics
- stations
- processes

The bottom level, namely the process level, models elementary assembly or machining operations in which the properties and behavior of the system components can be influenced. The middle level, namely the station level, maps the system's buffer stores and groups machines together into stations according to a workshop or line production. At the top level, namely the layout level, workpieces are created by sources and removed by sinks. In addition, the material flow of workpieces through the production process is described.

Users must define production processes in two input files. In a JSON file, all orders, stations, and the factory are defined. In a Python script, the users specify the assembly and processing functions, the behavior of the sources and sinks, as well as global functions and user-defined distributions for attribute values.

Additionally, the package offers functionalities for the visualization of passed production processes, verification of input files, and methods for estimating the simulation runtime

The following code displays the typical usage of the package:

```
from prodsim import Environment

def main():

    # Create simulation Environment
    env = Environment()

    # Read the input files
    env.read_files('./data/process.json', './data/function.py')

    # Inspect and visualize the input data (optional)
    # env.inspect()
    # env.visualize()

    # Start the simulation
    env.simulate(sim_time=10_000, progress_bar=True, max_memory=5, bit_type=64)

    # export the output data
    env.data_to_csv("./data/output/", remove_column=['item_id'], keep_original=True)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':

    main()
```

How this documentation should be used:

The *API Reference* chapter provides an overview of all methods and their attributes as well as the corresponding data types. The *Interface Files* chapter describes the structure to be followed by the input files. These two chapters are designed as a reference for specific content. In the final *Examples* chapter, examples are chronologically matched to the later simulation study and contain all elementary features of the package. Since some modeling techniques are also explained, studying these examples is recommended before conducting one's own simulation study.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 API Reference

1.1.1 Environment

The Environment class represents the central element of the library. All offered simulation functionalities are available to the user in the methods through an object of this class. In addition, the environment controls all program-internal method calls as well as access to the process data in the background.

class environment.**Environment**

Execution Environment for the event-based production simulation.

clear_env() → None

Reinitialize the environment object between two different simulation runs.

After calling this method, a new process must be read in.

data_to_csv(*path_to_wd: str*, *remove_column: Optional[List[str]] = None*, *keep_original: bool = True*) → None

Exports the simulation data to csv files.

Parameters

- **path_to_wd** (*str*) – Path to the target directory
- **remove_column** (*List[str], optional*) – List of labels whose columns are removed before saving
- **keep_original** (*bool, optional*) – Keep an additional original file without removed columns

Raises **MissingData** – `simulate` was not called before

Note: If the passed folder does not exist, then the program creates it.

inspect() → None

Checks the passed input files for errors of logical and syntactic nature.

Raises **MissingData** – `read_files` was not called, or the data read in does not contain the arrays 'order' and 'station'

Note: This method is only a support and does not guarantee an error-free simulation run.

read_files(*path_data_file: str*, *path_function_file: str*) → None
Reads in the process input files.

Parameters

- **path_data_file** (*str*) – Path to the JSON file with the process data
- **path_function_file** (*str*) – Path to the py file with the function definitions

Raises

- **FileNotFoundError** – Files could not be found
- **MissingParameter** – The 'order' or 'station' array is not defined in the process file or an order or station object has no name
- **UndefinedFunction** – One of the referenced functions cannot be found in the function file
- **UndefinedObject** – One of the referenced orders or stations cannot be found in the data file
- **InvalidType** – The component list has an element that is not of type list, or the capacity of an order or station is not of type int
- **InvalidValue** – The capacity of an order or station is not greater than zero
- **NotSupportedParameter** – One of the values of the user-defined factory attributes has an undefined identifier

simulate(*sim_time: int*, *track_components: Optional[List[str]] = None*, *progress_bar: bool = False*, *max_memory: float = 2*, *bit_type: int = 32*) → None
Starts the simulation run.

Parameters

- **sim_time** (*int*) – Simulated time
- **track_components** (*List[str], optional*) – List of strings representing components whose process data is to be stored
- **progress_bar** (*bool, optional*) – Specifies whether a progress bar should be displayed
- **max_memory** (*float, optional*) – Maximal size of a single a numpy data array [Mb]
- **bit_type** (*int, optional*) – Bit type with which the values are stored

Raises **MissingData** – `read_files` was not called or the data read in does not contain 'order' or 'station'

visualize() → None
Launches an interactive web application to display the input data. This method initiates a local development app on a flask server on localhoast:8050.

Raises **MissingData** – `read_files` was not called or the data read in does not contain the 'order' or 'station' array

---

Note: This method initiates a local development app on a flask server on localhoast:8050.

1.1.2 Estimator

The Estimator class offers some functionalities through which the runtime behavior of the simulation can be estimated. Alternatively, a reference simulation with a short simulation time can be performed, and the measured simulation time can be scaled proportionally. However, the function est_function is especially useful for developing suitable process functions.

class estimator.**Estimator**

Estimator for estimating the expected simulation time.

est_attribute(*distribution: List[tuple]*, *num_station: int*, *track: bool*) → float

Estimates the time caused by additional attributes.

Parameters

- **distribution** (*List[tuple]*) – List of attributes to be estimated
- **num_station** (*int*) – Number of stations that workpieces of the order under consideration pass through
- **track** (*bool*) – Indicates whether the order is being tracked

Returns Estimated additional simulation time for additional attributes

Return type float

est_function(*function: Callable*, *num_station: int*, *track: bool*, *imports: Optional[List[str]] = None*, *objects: Optional[Dict[str, object]] = None*, *item_attributes: Optional[Dict[str, list]] = None*, *machine_attributes: Optional[Dict[str, list]] = None*, *factory_attributes: Optional[Dict[str, list]] = None*) → float

Estimates the time caused by a specific function.

Parameters

- **function** (*Callable*) – List of attributes to be estimated
- **num_station** (*int*) – Number of stations at which the process function is called
- **track** (*bool*) – Indicates whether the order is being tracked
- **imports** (*List[str], optional*) – List of all used import statements
- **objects** (*List[object], optional*) – List of all used objects
- **item_attributes** (*Dict[str, list], optional*) – List of all item attributes used in the function
- **machine_attributes** (*Dict[str, list], optional*) – List of all machine attributes used in the function
- **factory_attributes** (*Dict[str, list], optional*) – List of all factory attributes used in the function

Raises **InvalidFunction** – Function name is 'function1'

Returns Estimated time for a single function call

Return type float

est_item(*track: bool*) → float

Estimates the time for creating a workpiece.

Parameters **track** (*bool*) – Indicates whether the order is being tracked

Returns Estimated simulation time for creating a workpiece without attributes

Return type float

est_station(*track: bool*) → float
Estimates the time caused by the recursive process logic.

Parameters **track** (*bool*) – Indicates whether the order is being tracked

Returns Estimated simulation time for simply passing through stations (without functions and item attributes)

Return type float

1.2 Interface Files

This chapter defines the structure of the two input interface files and the options available to the user for mapping production processes. First, the elements of the JSON file describing the simulation objects are presented, followed by the different function types of the py file.

- *Data file*
- *Function file*

A further section describes the possible distributions used to initialize the attributes of simulation objects when they are created.

- *Attribute values*

Note: A subset of the exceptions listed in the following sections will only be thrown when the inspect method is called.

1.2.1 Production structure

Each production process is defined in its own JSON file. This file contains a top-level object with two required attributes and one optional one. The structure of these attributes is described as follows:

1. *Order*
2. *Station*
3. *Factory*

Order

The Order attribute is an attribute of the top-level production process object and is of the type JSON Array. This array contains JSON objects and defines an order that combines all of the information about a particular order. The attributes of an order are differentiated into predefined and user-defined attributes. Any attribute whose name is not predefined is considered a user-defined attribute. In this section, all predefined attributes are described in detail. The possible characteristics of the user-defined attributes are described in a separate section. *section*.

Note: Only the individual parameters are described below. In *example 01* , a concrete example of this file is given.

`name`

The name is a required parameter of the data type String. It will later serve as an identifier for the different jobs and should therefore be unique.

	Value	Explanation
Optional	no	
Default value	/	
Exceptions	MissingParameter	If name was not set
Warnings	BadType	If name isn ot a string

Warning: Since the suffix '_x' references identical assembly workpieces that are assembled in different process steps (see *process function*), the name cannot have such a suffix.

`priority`

The priority is an optional integer parameter. It determines the processing order when multiple jobs request the same scarce resource. If no priorities are set, then the program determines its order. A small value corresponds to a high priority. If several orders do not use the same station, then the priorities have no meaning.

	Value	Explanation
Optional	yes	
Default value	10	
Exceptions	InvalidType	If priority is not an integer
	InvalidValue	If priority is less than one

`storage`

The storage is an optional integer parameter that specifies the storage capacity of the final store of an order. The storage is a piece value.

Note: Even though this parameter is optional, it should always be set if there is no perfect understanding of the process; otherwise, situations may occur where an increasing number of item objects are stored in stores over the simulation time. This would lead to memory overload and slow the simulation speed.

	Value	Explanation
Optional	yes	
Default value	infinite	
Exceptions	InvalidType	If capacity is not an integer
	InvalidValue	If capacity is less than one

`source`

The source is a required parameter of type string. The function from the *production functions* file with the corresponding name is assigned to this order.

	Value	Explanation
Optional	yes	
Default value	/	
Exceptions	UndefinedFunction	Function is not defined in the passed file
	InvalidFunction	Function is not a generator function
	InvalidSignature	Function does not have exactly two arguments
	InvalidYield	Yielded object is not of type int or Timeout
	InfiniteLoop	Source contains an infinite loop
	MissingParameter	No source was defined
Warnings	BadSignature	The signature is not ('env', 'factory')
	BadYield	Source does not yield a timeout event

`sink`

The sink is an optional parameter of type string. This order is assigned the function from the *production functions* file with the corresponding name. If workpieces of this order represent assembly workpieces concerning another process, then the default sink will never be active. If this is not the case, then it removes all workpieces from the final store without a time delay.

	Value	Explanation
Optional	yes	
Default value	infinite source	If item is not part of an assembly process
	no source	If item is part of an assembly process
Exceptions	UndefinedFunction	Function is not defined in the passed file
	InvalidFunction	Function is not a generator function
	InvalidSignature	Function does not have exactly two arguments
	InvalidYield	Yielded object is not of type int or Timeout
	InfiniteLoop	Source contains an infinite loop
Warnings	BadSignature	The signature is not ('env', 'factory')
	BadYield	Source does not yield a timeout event

`station`

The station attribute is an optional attribute of type Array. This array contains strings that represent the names of stations in the order in which items of this order visit them. The default value is an empty array, which means that the source places new workpieces directly into the final store (reflecting, for example, the retrieval of external assembly workpieces).

	Value	Explanation
Optional	yes	
Default value	[]	
Exceptions	UndefinedObject	No station is defined with this name

Note: The program does not throw exceptions related to the array's length because the size of this array is considered a reference for the length of the other arrays.

`function`

The function attribute is an optional attribute of type array. It contains strings that correspond to the names of functions defined in the *process functions* file. The index position determines the connection of process functions to stations.

	Value	Explanation
Optional	yes	
Default value	[]	
Exceptions	UndefinedFunction	No function with this name is defined
	InvalidSignature	Function does not have four arguments
	MissingParameter	Number of functions does not match the number of stations
Warnings	BadSignature	At least one argument has a bad name
	BadYield	Function does not yield a simpy.Timeout object

`demand`

The demand parameter is an optional parameter of type array. The index position of the entries connects them to the stations from the station's list. If a station performs an assembly or a pure machining process in a given process step, then it determines the structure of the entries of the array. In machining at the station with index position i, the i-th element of the demand array is an integer that determines the demand of this station. Another array of integers at the corresponding index position in an assembly, which determines the number of individual assembly pieces. The *component* attribute specifies which workpieces are used in an assembly. The default value is a list with only 1s and the length of the station list. Thus, the default case represents a pure line production.

	Value	Explanation
Optional	yes	
Default value	[1, 1, .., 1]	Only possible if there is no assembly
Exceptions	MissingParameter	Number of elements does not match the number of stations
	InvalidType	If the list contains different objects than int or list of int
	InvalidValue	Integer element in the list is not greater than zero

Note: No exceptions are thrown if the inner structure does not fit around the attribute component because demand serves as a reference to avoid redundant error messages.

`component`

The component attribute is an optional attribute of type array. The inner structure of this array corresponds to that of the demand *demand attribute*. In the case of pure processing, there is an empty array at the corresponding index position. In assembly, the inner array contains strings that correspond to the names of orders and specify what type the assembly workpieces should be. The default value is an array with only empty arrays; thus, as with the attribute demand, a pure line production is represented.

	Value	Explanation
Optional	yes	
Default value	[[], [], .., []]	Only possible if there is no assembly
Exceptions	MissingParameter	Number of elements does not match the number of stations or the length of the assembly process list does not match the length of the assembly demand list
	UndefinedObject	No item is defined with this name
	InvalidValue	Structure does not correspond to the demand structure
	InvalidType	List contains object with a type other than 'list'

Station

The station attribute is an attribute of the top-level production process object and is of the type JSON Array. This array contains JSON objects that define a station that combines all of the information about a particular station. The attributes of a station are differentiated into predefined and user-defined attributes. Every attribute whose name is not predefined is considered a user-defined attribute. This section describes all predefined attributes in detail. The possible characteristics of the user-defined attributes are described in a separate *section*.

Note: Only the individual parameters are described below. In *example 01*, a concrete example of this file is given.

`name`

The name is a required parameter of type string. It is used later to identify station objects and therefore must be unique.

	Value	Explanation
Optional	no	
Default value	/	
Exceptions	MissingParameter	If name was not set
Warnings	BadType	If name is not a string

`capacity`

The capacity is an optional integer parameter. It specifies the number of machines that the corresponding station has and thus serves to map the production type of the shop floor production. The default value is one and it thus rather represents a line production process. If a station has several machines, then one of the free machines is selected randomly before machining at this station.

	Value	Explanation
Optional	yes	
Default value	1	
Exceptions	InvalidType	If capacity is not an integer
	InvalidValue	If capacity is less than one

`storage`

The storage attribute is optional and of type integer. It describes the storage capacity of the buffer storage of a station. This attribute is a unit value.

Note: Even though this parameter is optional, it should always be set if a perfect understanding of the process does not exist; otherwise, an arbitrary accumulation of numerous objects could occur in the memory, which would slow the simulation arbitrarily.

	Value	Explanation
Optional	yes	
Default value	infinite	
Exceptions	InvalidType	If storage is not an integer
	InvalidValue	If storage is less than one

`measurement`

The measurement attribute is optional and of type Boolean. If a station is a measurement or quality control station where the item attributes are not changed, then this attribute should be set to 'true'. The effect is that workpieces will not be tracked at this station, regardless of whether they are tracked at other stations.

	Value	Explanation
Optional	yes	
Default value	false	
Exceptions	InvalidType	If measurement is not a Boolean

Factory

The factory attribute is an optional attribute of the top-level production process object. Unlike the order and station attributes, it is not an array but rather a single JSON object. This object contains all global attributes that any process function, sink, source, and global function can retrieve. All of these attributes are user-definable. The rules that apply to these attributes are described in the *'Attribute values'* section. In addition, the factory object has a predefined attribute, which is described below.

`function`

The function is an optional attribute of type array. This array contains strings that correspond to the global functions from the functions file. The global variables are controlled from these functions, whose structure is described in more detail in the *'function file* section.

	Value	Explanation
Optional	yes	
Default value	[]	
Exceptions	UndefinedFunction	No function with this name is defined
	InvalidFunction	Function is not a generator function
	InvalidSignature	Function does not have exactly two argument
	InvalidYield	Function yielded an object that is not of type
Warnings	BadSignature	Parameters are not called 'env' and 'factory'

1.2.2 Production functions

The functions input file is a Python script in which the user defines all of the functions used in the process input file. The functions must be defined in the global scope and can be classified as follows

- *Process function*
- *Source and sink*
- *Global function*
- *Distribution*

Note: The following subsections describe only the structure and functionality. The use of these functions is presented in chapter 3 (*Examples*).

Process function

All process functions referenced in the orders under the 'function' attribute must be defined in the function input file. The process functions are used to represent machining or assembly operations, and each of these functions has four arguments: *env*, *item*, *machine*, and *factory*. The following paragraphs explain what these arguments are used for:

env

The argument env points to the reference of the simulation environment of the simulation kernel. This reference can be used to access the current simulation time via the attribute now to make the behavior of the process function dependent on the simulation time.

```
current_sim_time: float = env.now
```

In addition, this reference is used to set the current process to the *active without control* state. For this purpose, a *simpy.Timeout* event is yielded through *env*. The duration of the release of control is controlled by a time interval passed in the process. The machine is blocked for this time such that, for example, maintenance or processing times can be mapped.

```
# Using a random delay
delay: float = normalvariate(10, 0.2)

# Delays must be positive
yield env.timeout(abs(delay))
```

Note: Any number of timeout events can be yielded in a process function, whereas returns are only used to manage the control flow if necessary.

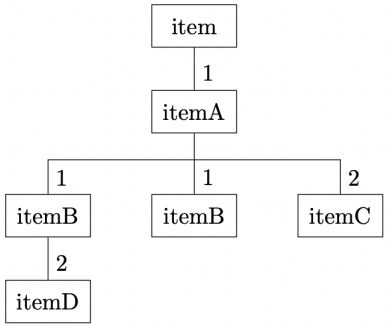
---

item

Through the argument *item*, all references to workpieces involved in the process can be accessed. The following table displays which information is available.

	Read	Write	Access
attributes	+	+	*item.attr_1* (e.g.)
id	+	-	*item.item_id*
name	+	-	*item.name*
reject	+	+	*item.reject*

The following figure illustrates what the item access structures look like when the workpieces are nested or the demand of the process is greater than one.



The item attribute always references the main workpiece of a process – itemA in this case. The figure shows that two workpieces of type itemC were assembled into itemA. Whenever the quantity is greater than one, the references are stored in lists. The access to an attribute (e.g., *attr_1*) of the first of the two itemC items looks as follows:

```
item.itemC[0].attr_1
```

If two (or more) workpieces of the same type were assembled in different assembly steps (see *itemB*), then access would be made in a special way. Starting from the second workpiece, the references are supplemented by the prefix '_' and a continuing suffix. Thus, identical workpieces from different process steps can be differentiated. Assuming the middle itemB was mounted second, access from its attributes (e.g., *attr_2*) would be as follows

```
# accessing, the first assembled itemB
item.itemB.attr_2

# accessing, the second assembled itemB
item._itemB2.attr_2
```

This structure can be nested as far as required. Thus, access to attributes (e.g., *attr_3*) of itemD is through *itemB*:

```
item.itemB.itemD.attr_3
```

machine

The *machine* argument can be used to reference the attributes of the machine on which the machining takes place. In addition, each machine of a station has its own number.

	Read	Write	Access
attributes	+	+	*machine.attr_1* (e.g.)
machine nr	+	-	*machine.nr*
name	+	-	*machine.name*

Since there are no nested structures as with the items, access is always via *machine.attr_name*.

factory

All global attributes can be reached through the *factory* reference. These can also be assigned new values from process functions.

```
factory.global_attr
```

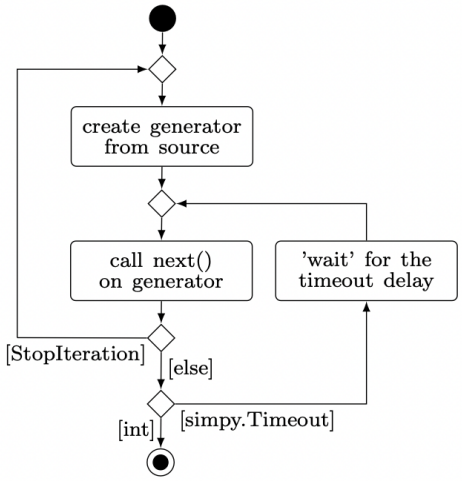
However, the behavior of the global attributes is not so controllable since, particularly with stochastic processes, how often or when a process function is called is not known. Therefore, the global attributes should only be set based on *global functions*.

Source and Sink

Each job has exactly one source and one sink. Their tasks are to create workpieces in the production process and to remove them after they have passed through the process. By matching the behavior of the source and sink, a push or pull material flow can be configured in the production system.

A source or sink is defined as a function in the global scope in the function input file and must match the values of the *sink* and *source* attributes of the orders. Such a function has exactly two arguments: *env* and *factory*. As described for the *processes*, through these arguments the user can access the current simulation time and generate timeout events, and access to global attributes is provided.

The following figure illustrates the logic of the source and sink functions. These functions can yield any number of objects of type int or *simpy.Timeout*. As soon as an *int* value is yielded, the iteration over the source or sink (generator) is aborted, and the yielded value corresponds to the number of workpieces that the source/sink generates/removes. If no int value is yielded, then the iteration stops after the last yield and starts again.



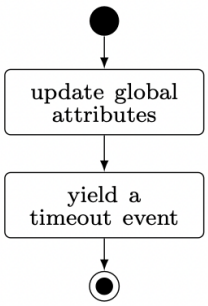
If no sink is defined, then the default sink removes all workpieces from the corresponding end storage without any time delay (if no assembly workpieces are taken from this store for another order). If workpieces are always to be ready for production, an infinite source can be defined. For this purpose, the storage of the corresponding buffer memory must be and an *int* value from the source must be yielded as the first value. Thus, the source always fills up the storage without a time offset and stops when it is full.

```
# Define an infinite source
def infinite_source(env, factory):
    yield 1
```

Example 03 gives a concrete example of the interaction of an infinite source and sink with a demand curve over time.

Global function

Global functions are specified via the *functions* attribute of the *factory* object and defined in the global scope of the function input file. The task of the global functions is to control the behavior of the global attributes. Global functions get two arguments: *env* and *factory*. Through env, as already shown with the *process functions*, *timeout* events can be generated and the simulation time can be queried, while factory is used to obtain access to the global attributes to assign new values to them depending on the time. The following figure presents the required structure of a global function schematically:



First, the global attributes are assigned updated values; arbitrarily nested structures can be used. Subsequently, at least one *timeout* event must be yielded. This is because the global functions are executed parallel to the simulation in an infinite loop; without a *timeout* event, the simulated time would not progress.

Example *02* demonstrates how to assign a time profile to global variables. Example *03* illustrates how global functions can be used to influence the behavior of the production system.

Distribution

The user-defined distributions that can be assigned to the attributes of the simulation objects are also defined in the global scope of the function input file. For content reasons, the structure of these functions is introduced together with the *attribute distributions*.

1.2.3 Attribute values

This section presents the possible distributions that can be assigned to the attributes of the simulation objects. A distribution is defined in the form of a list. The first element of the list is always a string of length 1, which serves as an identifier for the different distributions. The remaining attributes define the specific distribution parameters.

Distribution	Identifier	Parameters
User defined	/	/
Fix	f	[“f”, ν]
Binary	b	[“b”, p]
Binomial	i	[“i”, n , p]
Normal	n	[“n”, μ , σ]
Uniform	u	[“u”, a , b]
Poisson	p	[“p”, λ]
Exponential	e	[“e”, β]
Lognormal	l	[“l”, μ , σ]
Chisquare	c	[“c”, n]
Standard-t	t	[“t”, n]

User defined

If the desired distribution is not predefined, then the user can define custom distribution functions. Such a function is defined in the global scope of the function *input file*. The function's name later serves as an identifier and thus must have length 1 and not intersect with the predefined ones. Since the return value of such a function is assigned to the attributes, the return type must be int or float. However, the defined function can have any number of arguments of any type. In the distribution list, the arguments of this function are then passed in the same order.

Example:

The task is to define the following distribution function, which is not predefined, and then assign it to an attribute.

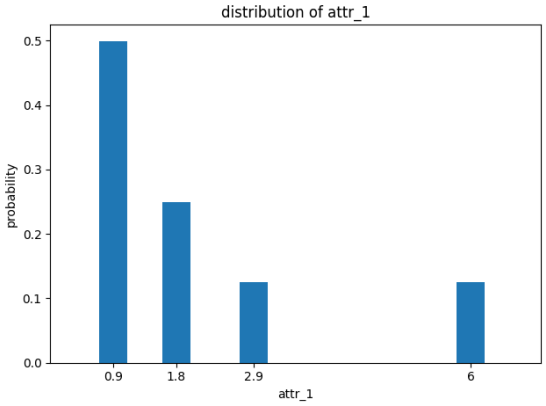
$$P(X = x) = \begin{cases} \frac{1}{2} & ; x = 0.9 \\ \frac{1}{4} & ; x = 1.8 \\ \frac{1}{8} & ; x = 2.9 \\ \frac{1}{8} & ; x = 6 \\ 0 & ; else \end{cases}$$

It is possible to define a function that uses the values shown. Alternatively, a general function is defined that can be used to map other arbitrary discrete distributions. For this purpose, x is used as a free identifier and choice from numpy.random as distribution. The attributes of the function x are two lists that contain the probabilities and discrete values.

```
from numpy.random import choice

def x(values, probabilities) -> Union[int, float]:
   return choice(a=values, p=probabilities)
```

Now, this distribution needs to be added to the attribute.

```
"attr_1": ["x",[0.9,1.8,2.9,6],[0.5,0.25,0.125,0.125]]
```



Note: User-defined distributions are not checked by the inspector.

Fixed

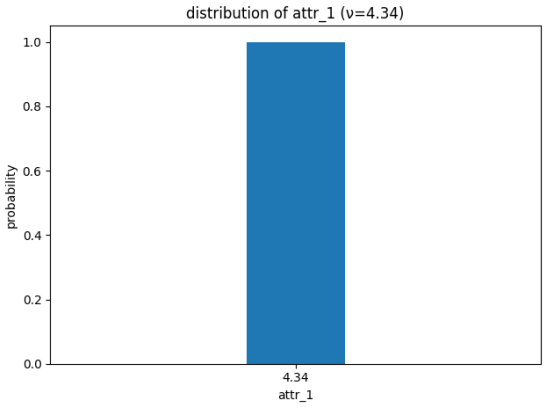
The identifier *f* indicates a fixed attribute value. The second parameter of the list specifies the fixed value ν that the attribute takes. Like all other attributes, the types *int* and *float* are possible. Distribution:

Distribution:

$$P(x = \nu) = 1\ , \quad \nu \in \mathbb{R}$$

Example:

```
"prob_of_failure": ["f",4.34]
```



Overview:

	Value	Explanation
Identifier	f	
Additional parameter	ν	Value
Exceptions	InvalidFormat	List does not have length 2
	InvalidType	ν is not of type int or float

Binary

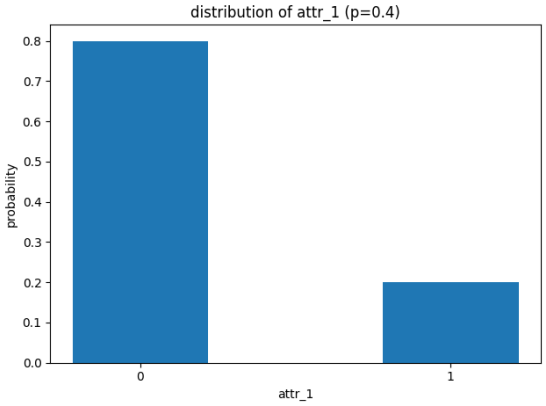
The identifier *b* indicates a binary attribute value. The second element of the list is the probability *p* that the attribute takes the value 1. *B(k,p)* indicates the probability that an attribute takes the values *k*, given probability *p*.

Distribution:

$$B(k,p) = \begin{cases} p^k(1-p)^{1-k} & ;k \in \{0,1\} \\ 0 & ;else \end{cases} \quad , \quad p \in [0,1]$$

Example:

```
"attr_1": ["b",0.2]
```



Overview:

	Value	Explanation
Identifier	b	
Additional parameter	*p*	Success probability
Exceptions	InvalidFormat	List does not have length 2
	InvalidType	*p* is not of type int or float
	InvalidValue	*p* is not between 0.0 and 1.0

Binomial

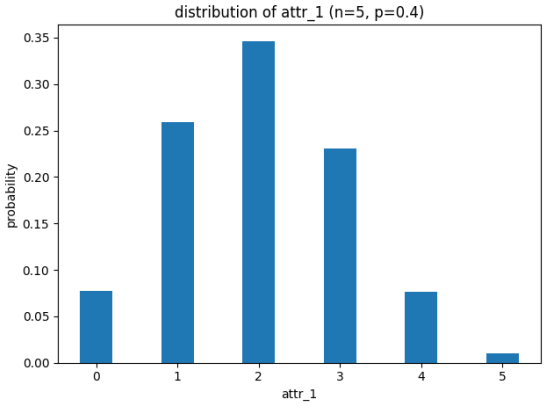
The identifier *i* indicates a binomial attribute value. The second list element is the number of trails *n*, while the third is the success probability *p*. *B(k,p)* indicates the probability that an attribute takes the values *k*, given probability *p* and the number of trails *n*.

Distribution:

$$B(k,n,p) = \begin{cases} \binom{n}{k} p^k (1-p)^{1-k} & ; k \in \{0,..,n\} \\ 0 & ; else \end{cases} \quad , \quad p \in [0,1]$$

Example:

```
"attr_1": ["i",5,0.4]
```



Overview:

	Value	Explanation
Identifier	i	
Additional parameter	n	Number of trails
Additional parameter	p	Success probability for each trail
Exceptions	InvalidFormat	List does not have length 2
	InvalidType	n is not of type int
		p is not of type int or float
	InvalidValue	n is not greater than zero
		p is not between 0.0 and 1.0

Normal

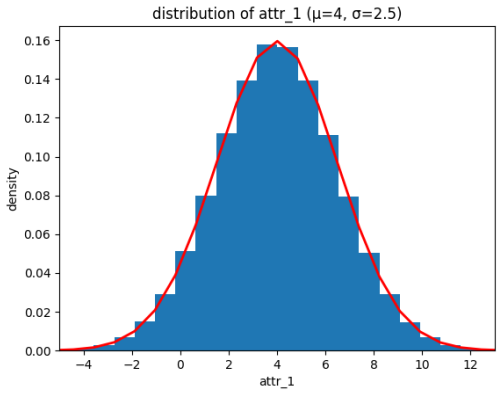
The identifier n indicates a normally distributed attribute. The second list entry corresponds to the mean μ and the third to the standard deviation σ.

Distribution:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \; ; \quad x, \mu \in \mathbb{R}, \; \sigma \geq 0$$

Example:

```
"attr_1": ["n",4,2.5]
```



Overview:

	Value	Explanation
Identifier	n	
Additional parameter	μ	Mean
	σ	Standard deviation
Exceptions	InvalidFormat	List does not have length 3
	InvalidType	μ or σ is not of type int or float
	InvalidValue	σ is smaller than zero

Uniform

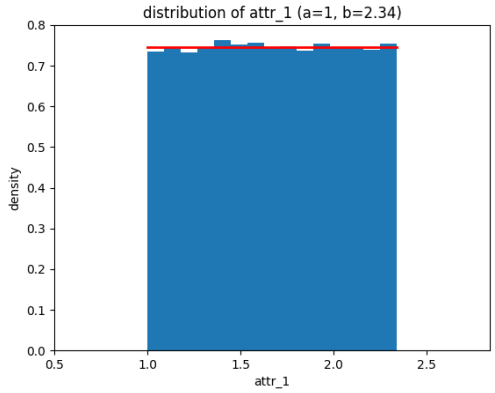
The identifier *u* indicates a uniform distributed attribute. The second list parameter *a* is the lower limit, while the third *b* sets the upper interval limit. The limits can be integers or floating-point numbers.

Distribution:

$$p(x) = \begin{cases} \frac{1}{b-a} & ; x \in [b, .., a) \\ 0 & ; else \end{cases} \quad ; \quad a, b \in \mathbb{R},\ b > a$$

Example:

```
"attr_1": ["u",1,2.34]
```



Overview:

	Value	Explanation
Identifier	u	
Additional parameter	*a*	lower bound
	a	upper bound
Exceptions	InvalidFormat	List does not have length 3
	InvalidType	*a* or *b* is not of type float or int
	InvalidValue	*a* is greater or equal *b*

Poisson

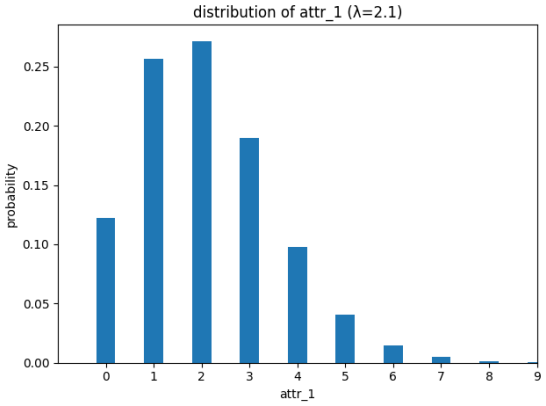
The identifier *p* indicates a Poisson-distributed attribute. The second list entry determines the rate λ, which must be type float or int and greater than or equal to zero. $P(k, \lambda)$lambda`.

Distribution:

$$P(k, \lambda) = \begin{cases} \frac{\lambda^k e^{-\lambda}}{k!} & ; k \in \mathbb{N}_{\geq 0} \\ 0 & ; else \end{cases} \quad , \quad \lambda > 0$$

Example:

```
"attr_1": ["p",2.1]
```



Overview:

	Value	Explanation
Identifier	p	
Additional parameter	λ	Rate
Exceptions	InvalidFormat	List does not have length 2
	InvalidType	λ is not of type float or int
	InvalidValue	λ is less than zero

Exponential

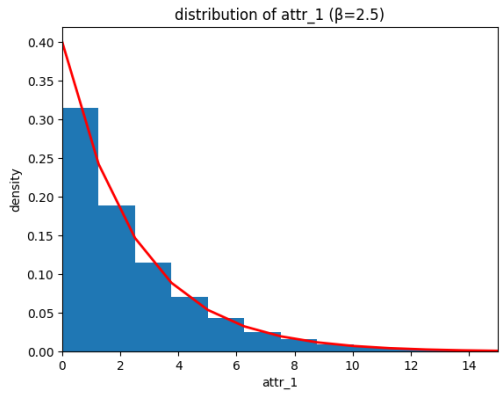
The identifier *e* indicates an exponential distributed attribute. The second list element is the scale β, which can be a positive floating-point number.

Distribution:

$$p(x) = \begin{cases} \frac{1}{\beta} e^{-\frac{x}{\beta}} & ; x \geq 0 \\ 0 & ; else \end{cases} \quad , \quad \beta \in \mathbb{R}_{>0}$$

Example:

```
"attr_1": ["e",2.5]
```



Overview:

	Value	Explanation
Identifier	e	
Additional parameter	β	Scale
Exceptions	InvalidFormat	List does not have length 2
	InvalidType	β is not of type float or int
	InvalidValue	β is less or equal to zero

Note: Often the exponential function is also defined by the rate $\lambda = \frac{1}{\beta}$, instead of the scale β. For more information see numpy.random.exponential.

Lognormal

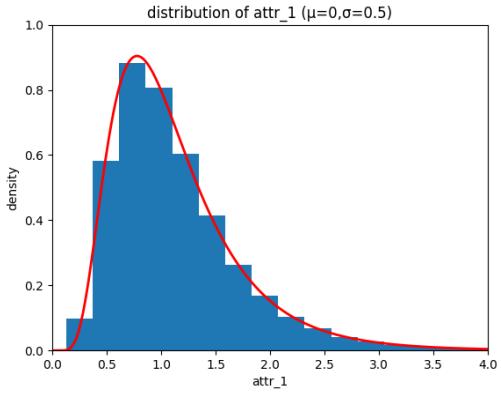
The identifier *l* indicates a lognormal distributed attribute. The second list entry corresponds to the mean μ and the third to the standard deviation σ; μ and σ must be of type int or float, while σ must also be greater than or equal to zero.

Distribution:

$$p(x) = \begin{cases} \frac{1}{\sigma x \sqrt{2\pi}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}} & ; x > 0 \\ 0 & ; else \end{cases} \quad , \quad \mu \in \mathbb{R},\ \sigma \in \mathbb{R}_{>0}$$

Example:

```
"attr_1": ["l",0,0.5]
```



Overview:

	Value	Explanation
Identifier	l	
Additional parameter	μ	Mean
	σ	Standard deviation
Exceptions	InvalidFormat	List does not have length 3
	InvalidType	μ or σ is not of type float or int
	InvalidValue	σ is less than zero

Chisquare

The identifier *c* indicates a chi-square distributed attribute. The second list entry determines the degrees of freedom *n*, which must be a positive floating-point or integer.

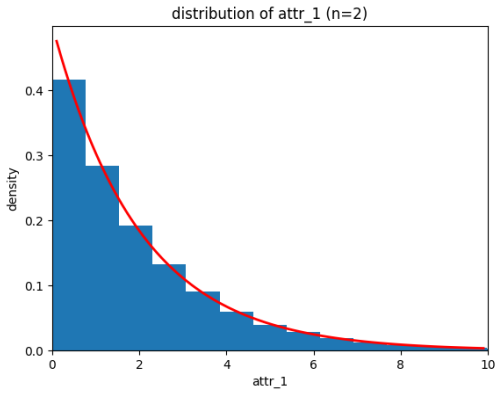
Distribution:

$$p_n(x) = \begin{cases} \frac{1}{2^{\frac{n}{2}}\Gamma(\frac{n}{2})} x^{\frac{n}{2}-1} e^{-\frac{x}{2}} & ; x > 0 \\ 0 & ; else \end{cases} \quad , \quad x \in \mathbb{R},\ n \in \mathbb{R}_{>0}$$

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt$$

Example:

```
"attr_1": ["c",2]
```



Overview:

	Value	Explanation
Identifier	c	
Additional parameter	*n*	Degrees of freedom
Exceptions	InvalidFormat	List does not have length 2
	InvalidType	*n* is not of type float or int
	InvalidValue	*n* is less than or equal to zero

Student-t

The identifier *t* indicates a student-t distributed attribute. The second list entry determines the degrees of freedom *n*, which must be a positive floating-point or integer.

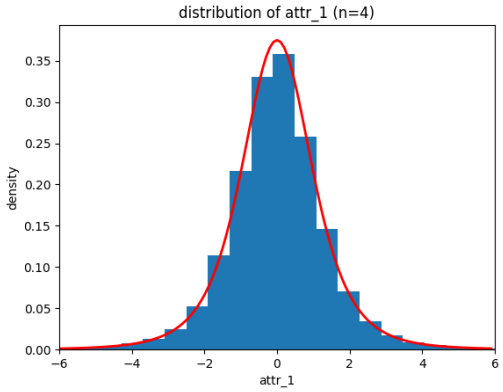
Distribution:

$$p_n(x) = \frac{\Gamma(\frac{n+1}{2})}{\Gamma(\frac{n}{2})\sqrt{n\pi}} (1 + \frac{x^2}{n})^{-\frac{n+1}{2}} \quad , \quad x \in \mathbb{R},\ n \in \mathbb{R}_{>0}$$

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt$$

Example:

```
"attr_1": ["t",4]
```



Overview:

	Value	Explanation
Identifier	t	
Additional parameter	*n*	Degrees of freedom
Exceptions	InvalidFormat	List does not have length 2
	InvalidType	*n* is not of type float or int
	InvalidValue	*n* is less than or equal to zero

1.3 Examples

In this chapter, the concrete use of the simulation program is presented through the use of examples. The aim of these examples is not to represent realistic contexts. Instead, they represent as many aspects as possible and are chronologically oriented to the later workflow. In addition, the examples are independent of each other in order to look up individual functionalities selectively. The following table serves as a guide:

Example	Focus
01	Defining a production layout Inspecting input files Visualizing input files
02	Defining machining functions Using global functions
03	Defining an infinite source Using global attributes Using a pull process principle
04	Accessing assembly workpiece attributes Rejecting items Transforming and filtering output data

Note: The examples presented hereafter are provided as executable examples in the following folder:

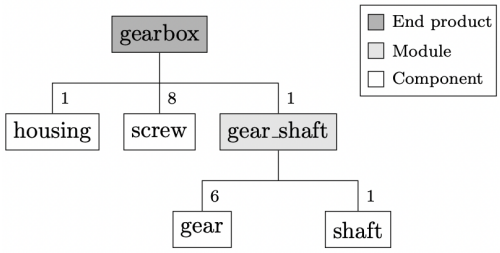
```
/ProdSim/examples/
```

1.3.1 Example 01: Gearbox

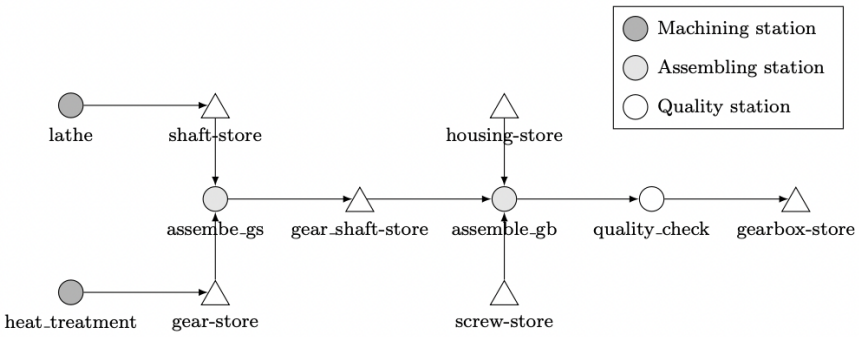
In this example, all steps are run through that should be conducted before each new simulation study. The focus is on the actual procedure and less on the process itself. Therefore, the process functions, sources, sinks, and attributes of the simulation objects are not filled with concrete content. Examples *02*, *03*, and *04* focus on the concrete modeling of process functions and sources.

Process description

Before any simulation study, the production process should first be formally described. For assembly processes, the use of a product tree is recommended to represent the product structure. The hierarchical relationship of the components with each other and the individual quantities are displayed. As shown with the *process functions*,this simplifies the later access to the workpiece attributes starting from the process functions. The following figure presents such a product tree using the example of a gearbox:



In addition, the production process should be represented in the form of a network. All product components' final stores (triangles) and all processing and assembly stations (circles) are drawn in. Then, all production processes are drawn in by directed edges between the stations. In addition, for assembly processes, the edges for the assembly workpieces from the final stores to the assembly stations are inserted.



Define orders

After describing the production process, the input files are defined. First, the orders should be specified in the JSON file. For this purpose, an order is created for each element from the product tree. Even if the elements gearbox and gear_shaft are not physical products but rather only namespaces for the union of elementary components, then these are also defined as orders. Thus, attributes can be assigned to them later.

The following procedure is recommended when defining an order:

1. Set general information (*name*, *priority*, *storage*, *source*, and *sink*)
2. Describe the process of the order (*station*, *function*, *demand*, and *component*)
3. Add custom attributes

The corresponding orders are presented as follows. The storage capacity is limited to 10 for each order to avoid unintentionally overfilling the computer memory.

```
{
  "order": [
    {
      "name": "gearbox",
      "storage": 10,
      "source": "source_1",
      "station": ["assemble_gb","quality_check"],
      "function": ["assemble_gb","quality_check"],
      "demand": [[1,8,1],2],
      "component": [["housing","screw","gear_shaft"],[]]
    },
    {
      "name": "housing",
      "source": "source_1",
      "storage": 10
    },
    {
      "name": "screw",
      "source": "source_1",
      "storage": 10
    },
    {
      "name": "gear_shaft",
      "storage": 10,
      "source": "source_1",
      "station": ["assemble_gs"],
      "function": ["assemble_gs"],
      "demand": [[6,1]],
      "component": [["gear","shaft"]]
    },
    {
      "name": "gear",
      "storage": 10,
      "source": "source_2",
      "station": ["heat_treatment"],
      "function": ["heating"],
      "demand": [8]
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "name": "shaft",
      "storage": 10,
      "source": "source_2",
      "station": ["lathe"],
      "function": ["turning"]
    }
  ]
}
```

Define stations

Next, the stations can be defined. For this purpose, a station object is created for each station in the production process. Since stations do not have as many properties as orders, the following procedure is recommended:

1. Set general information (*name*, *storage*, *capacity*, and *measurement*)
2. Add custom attributes

Here, the capacities are also limited in order not to overfill the computer memory. In addition, the station *quality_check* is a pure measuring station where no attributes are changed. Therefore, *measurement* is set to *true* for this station.

```
{
   "station": [
    {
      "name": "lathe",
      "storage": 10
    },
    {
      "name": "heat_treatment",
      "storage": 10
    },
    {
      "name": "assemble_gs",
      "storage": 10
    },
    {
      "name": "assemble_gb",
      "storage": 10
    },
    {
      "name": "quality_check",
      "storage": 10,
      "measurement": true
    }
  ]
}
```

Define factory

Finally, the global attributes and global functions must be defined. For this purpose, all attributes and global functions are assigned to the *factory* object.

As an example, two global attributes and one global function are defined as follows:

```
{
   "factory": {
      "glob_attr_1": ["f",0],
      "glob_attr_2": ["n",1,0.1],
      "function": ["glob_func_1"]
   }
}
```

Define functions

After the JSON file is set up, the Python script must be created. In this script, all previously used functions (sources, sinks, process functions, global functions, and distributions) are defined. As this focuses on the procedure, these functions are not assigned any content here. Therefore, examples *02*, *03*, and *04* should be viewed.

Inspect

After both input files are fully defined, the `inspect()` method can be called to identify errors that do not terminate the program when reading the data. Before doing so, a simulation environment must be created and the corresponding data read in.

```
from prodsim import Environment

if __name__ == '__main__':

    # Create simulation environment
    env = Environment()

    # Read in the process data
    env.read_files('.data/process.json', './data/function.py')

    # Inspect the process data
    env.inspect()
```

In the following example, two errors were deliberately introduced in the JSON file. First, the signature of the process function `turning` was changed, and the global function `global_func_1` did not yield a timeout event. After calling `inspect`, the output was as follows:

```
progress station: [====================] 100%  quality_check
progress order:   [====================] 100%  shaft
factory:          [====================] 100%  factory
WARNINGS-------------------
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "/Users/user/prodsim/inspector.py", line 522, in __inspect_order
    warnings.warn(
        prodsim.exception.BadSignature: The signature of a process function should be␣
↪(env, item, machine,
        factory), but in the function 'turning' at least one argument has a different␣
↪name.

EXCEPTIONS-----------------
Traceback (most recent call last):
  File "/Users/user/prodsim/inspector.py", line 575, in __inspect_factory
    raise prodsim.exception.InvalidFunction(
        prodsim.exception.InvalidFunction: The function 'glob_func_1' from the
        function file is not a generator function. A global function must yield at least␣
↪one timeout-event.

---------------------------
Number of Warnings:    1
Number of Exceptions:  1
---------------------------
```

Visualize

Finally, the `visualize` method can be called to check if the process was defined correctly.

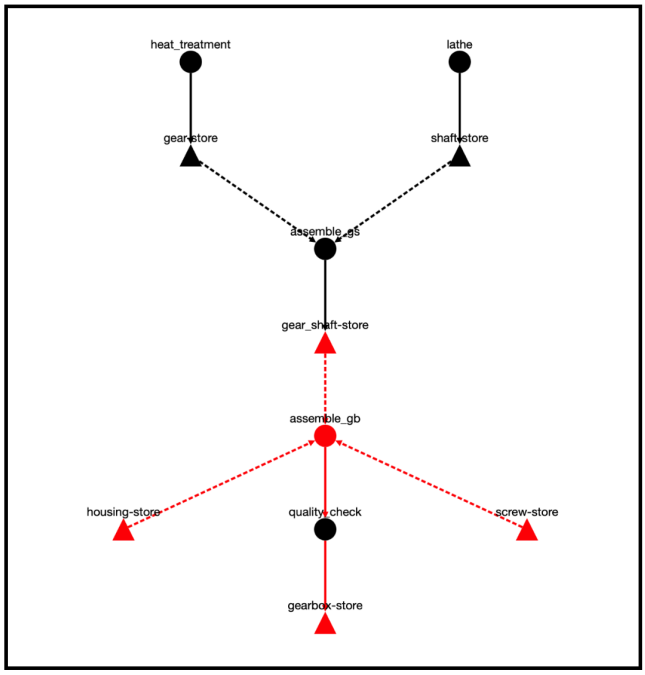
```
# Visualize the process data
env.visualize()
```

This call leads to the following output:

```
Dash is running on http://127.0.0.1:8050/

 * Serving Flask app 'ProdSim_app' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
```

By clicking on the link, a browser window opens that presents the interactive network graph.

ProdSim Visualizer



Order:

gearbox

Order Data:

properties	value
name	gearbox
priority	10
source	source_1
sink	default sink

Station Data:

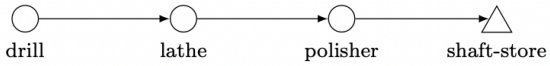
properties	value
name	assemble_gb
capacity	1
measurement	false
storage	1
function	['assemble_gb']
demand-gearbox	[1]
demand-housing	[1]
demand-screw	[8]
demand-gear_shaft	[1]

1.3.2 Example 02: Shaft

The focus of this example is the modeling of global and process functions. First, the process displayed is briefly outlined. Then, the individual functions are described in detail. Finally, the simulation output is used to validate the considerations.

Process description

Because the focus is on the functions, a simple process is deliberately used here. The process is a linear machining line, which operates on a cycle time of one minute. Since the drilling process takes 2 minutes, the station uses two machines to fulfill the cycle time. During this process, shafts are first drilled, turned, and then polished. The purpose of the simulation study is to determine the course of the surface quality over time. Shafts are not rejected during the process.



In addition, it is assumed that the process occurs in a factory with a temperature variation throughout the day, which influences the polishing process.

Process function: drilling

First, the shafts are drilled. Each machine has a probability (0.15% in this example) that the drill will break (drill_breakage). If this occurs, then the surface (surface) roughness will increase by an average of two units. In addition, the machine used for the machining process is blocked for the duration of the machining (2 minutes) by yielding a timeout event.

```
def drilling(env, item, machine, factory):

    # If the drill breaks the surface roughness increases
    if random.random() < machine.drill_breakage:
        item.surface += random.normalvariate(2, 0.1)

    # Blocking the drilling machine for machining time
    yield env.timeout(2)
```

Process function: turning

The lathe has wear that increases with each machining operation. Since the wear affects the surface quality, the lathe must be maintained whenever the wear reaches a certain level (1 in this example). This maintenance reduces the wear completely but blocks the machine for 10 minutes. The correlation (fictitious and for illustrative purposes only) between surface quality and machine wear is as follows:

$$\Delta surface = 1.5 \cdot (wear)^2 - 2$$

The wear of the machine increases by 0.006 units on average for each machining operation, so an average of 167 machining operations are possible between two rounds of maintenance.

```
def turning(env, item, machine, factory):

    # If the wear exceeds a certain limit, the machine is maintained
    if machine.wear >= 1:
        machine.wear = 0
        yield env.timeout(5)

    # The roughness achievable during machining depends on the wear of the machine
    item.surface += machine.wear**2 * 1.5

    # With each machining operation, the wear of the machine increases
    machine.wear += abs(normalvariate(0.006,0.00018))

    # Blocking the lathe for machining time
    yield env.timeout(1)
```

Process function: polishing

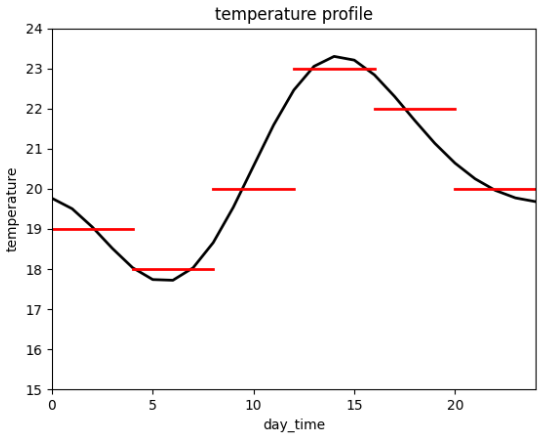
The polishing process can reduce roughness. If the temperature in the factory increases, then the polishing machine's potential to reduce the surface roughness decreases. The relationship between roughness and temperature is as follows:

$$\Delta surface = -(8 - temperature * 0.3)$$

```
def polishing(env, item, machine, factory):

    # The roughness will decrease the lower the temperature is.
    item.surface -= 8 - factory.temperature * 0.3

    yield env.timeout(1)
```

Global function: temperature

In the global function *temperature_func*, the profile of the temperature is described. In the simulated time (3 days), it is assumed that the temperature profile (black) in the following figure is given every day. The global temperature should correspond to the approximated course (red).



The temperature values are stored in a dictionary (in the global scope) and assigned to the temperature in *temperature_func*. The simulated time is checked for equality in the function, which is only allowed here because the time intervals in the timeout event are not random (otherwise a KeyError would occur).

This temperature profile is only intended to demonstrate the functionality. Of course, it is possible to define much finer profiles when corresponding data sets are available or to add certain variations to the values.

```
temp_dict = {0: 19, 240: 18, 480: 20, 720: 23, 960: 22, 1200: 20}

def temperature_func(env, factory):

    # Determinate the current daytime
    day_time = env.now % 1440

    # Set the new Temperature
```

(continues on next page)

(continued from previous page)

```
    factory.temperature = temp_dict[day_time]

    # Wait exactly 4 hours
    yield env.timeout(240)
```

Start simulation

This code shows how the simulation is started. The simulation time is 4320 since this is exactly 3 days in the unit of minutes. Since only the surface quality is of interest for the analysis, only the shafts are tracked. In addition, the column *item_id* is removed during the export of the data (For demonstration purposes only).

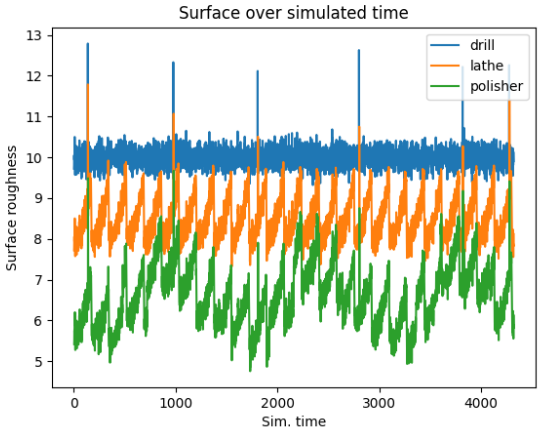
```
from prodsim import Environment

if __name__ == '__main__':

   # Create simulation environment
   env = Environemnt()

   # Read in the process files
   env.real_files('./data/process.json', './data/function.py')

   # Start the simulation
   env.simulate(sim_time=4320, track_components=['shaft'], progress_bar=True)

   # Export the simulation data
   env.data_to_csv(path_to_wd='./output/', remove_column=['item_id'], keep_
↪original=False)
```

Simulation output

The diagram below depicts the surface roughness that the shafts exhibit over the simulated time after processing at each station.

The following aspects can be identified:

1. The six outliers visible in the three plots are caused by broken drills;
2. The zigzag shape that starts at the turning process step is caused by wear, which increases until maintenance before abruptly decreasing;
3. The effect of temperature appears in the wave-like course (green). There are three cycles since exactly 3 days were simulated.



The interruptions in production due to maintenance work at the lathe cannot be recognized. The reason for this is the line thickness of the plots. The raw output data reveals the points in time at which the process is not active. This time difference does not correspond exactly to the 10 minutes since the buffer stores are first filled before the process succumbs.

shaft.csv

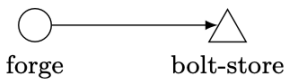
surface	station_id	time
10.222558	-1	1
9.8581171	-1	2
10.222558	0	3
⋮	⋮	⋮
10.267733	-1	174
9.9085436	-1	175
8.203002	1	181
5.9030023	2	182
⋮	⋮	⋮
8.104413	1	4319
9.9859371	-1	4319

1.3.3 Example 03: Bolt

The purpose of this example is twofold: first, it sets up a pull-controlled material flow in the production system using the interaction between the source and the sink of an order, and second, it shows that global attributes can control the material flow in the production system.

Process description

To understand the interaction of source and sink more easily, a simple process was chosen. A forge station has five forges, each of which produces six bolts per minute. These forges can be independently activated without start-up times. The finished bolt storage can hold up to 5000 bolts.



This process runs 24 hours a day, and demand fluctuates throughout the day at unknown levels. The goal is to activate the machines to ensure that demand can be met and productivity is adjusted to demand.

Source

Since the production process is controlled from the sink, it is necessary to ensure that enough input material is always available. An infinite source achieves this.

```
def infinite_source(env, factory):
    yield 1
```

An infinite source, where new input material is placed without delay, does not yield a *timeout* event. To enable a simulation with an infinite source, two conditions must be fulfilled:

1. The capacity of the buffer storage that is to be filled must be limited; and
2. The buffer storage capacity must be at least the same as the demand of the process concerning the first process step.

Note: Stores that are filled by an infinite source should not be filled by additional finite sources since the infinite sources dominate them.

Global function

There are three global attributes:

1. *number_bolts*: The number of bolts in the final storage
2. *active_machines*: The number of currently active machines
3. *max_active_machines*: The maximum allowed number of currently active machines

Since the demand (fictitious) is unknown and the production capacity is to be dynamically controlled, the number of bolts in the final storage is used as a control variable.

$$max\_active\_machine = \begin{cases} 5 & ; number\_bolts \in [0, 1000) \\ 4 & ; number\_bolts \in [1000, 2000) \\ 3 & ; number\_bolts \in [2000, 3000) \\ 2 & ; number\_bolts \in [3000, 4000) \\ 1 & ; number\_bolts \in [4000, 5000) \\ 0 & ; number\_bolts = 5000 \end{cases}$$

The idea is that when the demand increases, the number of bolts in the final storage decreases. Thus, the lower the number of bolts, the higher the number of active machines must be, such that the production capacity adjusts itself with a slight time delay to the subsequent demand without having to know the demand. To make this work, the maximum average demand must be smaller than the maximal production capacity of 30 (6 * 5).

```
control_logic = {1000: 5, 2000: 4, 3000: 3, 4000: 2, 5000: 1}

def global_control(env, factory):

    # Set max_active_machines_based on number_bolts
    for quantity in control_logic.keys():
```

(continues on next page)

(continued from previous page)

```
        if factory.number_bolts < quantity:
            factory.max_active_machines = control_logic[quantity]
            break
        factory.max_active_machines = 0

    # Update every time step (minute)
    yield env.timeout(1)
```

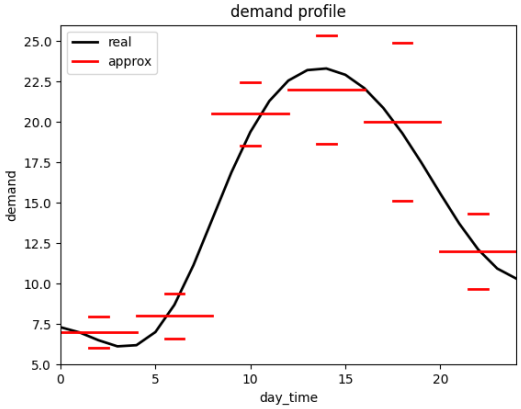
Process function: forging

As the focus is on the material flow, no attributes of the bolts are considered in this process function. Before the forging starts, whether the maximum number of active machines has been reached is checked. Since the cycle time is 1 minute, this check is repeated every minute. If this check is passed, then the number of active machines is increased, and the machine is blocked for the forging time. After the forging has finished, the global variable for storage filling is updated, and the number of active machines is updated again.

```
def forging(env, item, machine, factory):

    # Check if production capacity is reached.
    while True:
        if factory.active_machines < factory.max_active_machines:
            break
        yield env.timeout(1)

    # Update currently active machines
    factory.active_machines += 1

    # Block forge for forging time
    yield env.timeout(1)

    # Update store quantity
    factory.number_bolts += 6

    # Update currently active machines
    factory.active_machines -= 1
```

Sink

It is assumed that the demand follows the given course (black) daily and undergoes certain variations. An approximation is made by six partial intervals, which demonstrate a certain scatter (the 95% interval is indicated).



In addition, a large demand occurs for approximately 250 bolts approximately every 4 hours, which is also subject to variation. The following function presents the realization of such a source behavior. In addition, the current inventory in the final storage of the bolts is updated.

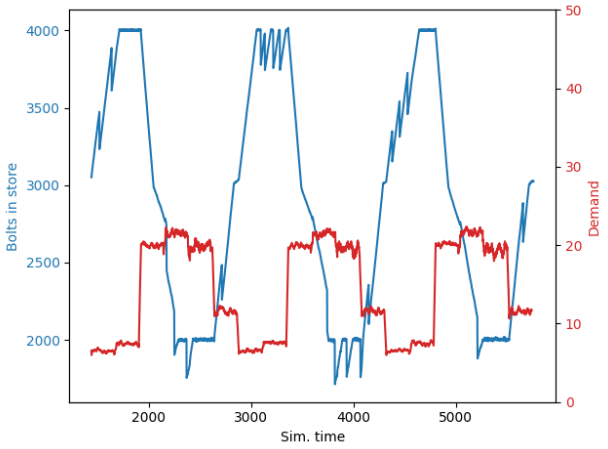
```
# Defines the demand distribution over time
time_dict = {1: [0, 4], 2: [4, 8], 3: [8, 12], 4: [12, 16], 5: [16, 20], 6: [20, 24]}
demand_dict = {1: [7, 0.5], 2: [8, 0.7], 3: [20.5, 1], 4: [22, 1.7], 5: [20, 2.5], 6:␣
↪[12, 1.2]}

def bolt_sink(env, factory):

    demand = 0
    day_time = env.now % 1440

    # Determine the standard demand
    for index, time_interval in time_dict.items():
        if time_interval[0] < day_time/60 < time_interval[1]:
            dis = demand_dict[index]
            demand += int(normalvariate(dis[0], dis[1]))
            break

    # Determining the additional demand
    if random() < 0.004:
        demand += int(abs(normalvariate(250, 20)))

    yield env.timeout(1)

    # Update number of bolts
    factory.number_bolts -= demand

    # Just for output plotting purpose
    factory.current_demand = demand

    yield demand
```

Simulation output

The following figure depicts the course of the number of bolts in the final store as well as the demand. The additional demands have been removed from the plot, and a moving average has been used for the demand. Due to oscillation processes at the beginning, the simulated days 2–4 are shown.

The following aspects can be identified:

1. At midday, the demand is approximately 20, so three to four forges must be active to meet the demand. Therefore, the average inventory at midday is 2000 (see *global_control*). At night, the demand is approximately eight, so only one to two forges are required.
2. If there is an additional demand in the steady-state (e.g., at Sim. time = 2400), then the inventory level decreases abruptly. This increases the number of active machines such that the required stock is built up again.



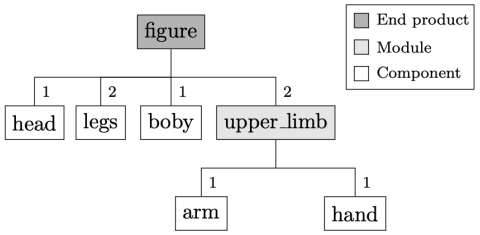
Note: Of course, this mechanism does not represent an efficiency control. The point of this example is rather the use of global quantities to limit machine activity. For example, the currently available electricity can also serve as a limit for the machines.

1.3.4 Example 04: Toy figure

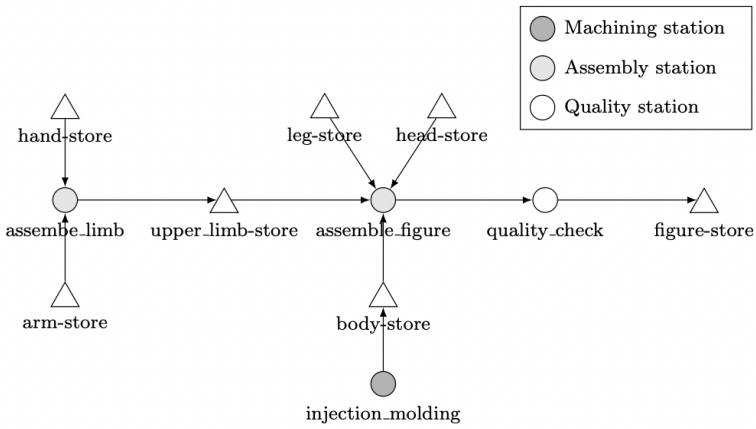
In this example, the characteristics of accessing assembly workpiece attributes are demonstrated. In addition, the usage of the workpiece attributes *reject* and *item_id* is described. Finally, the output structure is presented along with how the output can be transformed into the required format.

Process description

The production of plastic toy figures serves as an example process. The following product tree describes the components, their quantities, and the assembly relationships.



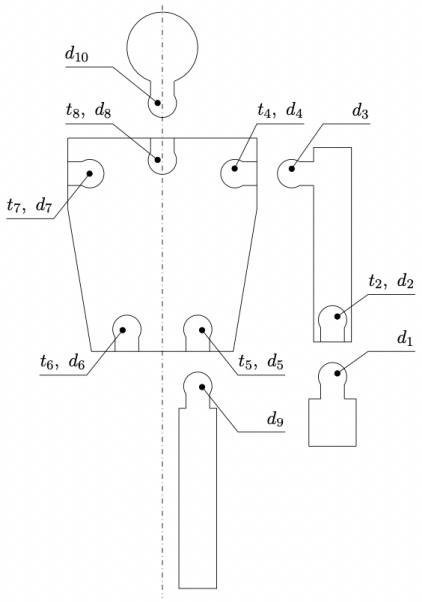
The components *arm*, *hand*, *leg* and *head* are produced externally and do not have separate machining steps within the process itself. The body component is injection-molded within the process. First, the components *arm* and *hand* are assembled into the module *upper_limb*. In the following assembly process, all components are assembled in a figure. Finally, quality is checked and incorrect figures are rejected.



The components are connected using ball-and-socket joints. Each joint has a diameter. Based on the difference in diameters, the tension that occurs in the joint is determined. For example:

$$t_4 = t_4(d_4, d_3)$$

The diameters of the joints are the attributes of the respective components, whereas the resulting tensions are determined during assembly and are therefore attributes of the modules or the final product. Since the figure is symmetrical, all *arms*, *hands*, and *legs* have the same attributes with individual characteristics.



Assemble function

This subsection describes the assembly function that is called at the station *assemble_figure*. The argument item of the function references the workpiece in whose process path the station from which the process function was called is located (figure). This attribute can be used to access all assembled workpieces that are assembled before or at the station under consideration. According to the following relationship, the tension is calculated and stored in attributes t4 to t8 of the *figure* workpiece for each ball joint:

$$t_i(d_i, d_j) = (d_j - d_i - 2)^3 + 20$$

As an example, tension t4 is used to describe the access of the required diameters; t4 depends on d3 of the right *arm* and on d4 of the body. Since item refers to figure, the module upper_limb must be accessed first. Since there are two upper_limbs, one of the two must be selected. By definition, it is declared that the first element corresponds to the right *upper_limb*. Since d3 is an attribute of the arm, the *upper_limb* must be used to access the *arm* and then d3. This results in the following:

```
d3_1 = item.upper_limb[0].arm.d3
```

The structure is similar for the diameter d4. First, *item* (or figure) must be used to refer to *body*. Since d4 is an attribute of body, d4 can be accessed as follows:

```
d4 = item.body.d4
```

The two stresses t2 have already been determined during the assembly of the component *upper_limb*.

```
def assemble_figure(env, item, machine, factory):

    # Get the diameters of the assembled items
    d3_1 = item.upper_limb[0].arm.d3
    d3_2 = item.upper_limb[1].arm.d3
    d9_1 = item.leg[0].d9
```

(continues on next page)

(continued from previous page)

```
    d9_2 = item.leg[1].d9
    d10 = item.head.d10

    def get_t(d1, d2):
        return (d2 - d1 - 2)**3 + 20

    # Calculate the tension
    item.t4 = get_t(item.body.d4, d3_1)
    item.t5 = get_t(item.body.d5, d9_1)
    item.t6 = get_t(item.body.d6, d9_2)
    item.t7 = get_t(item.body.d7, d3_2)
    item.t8 = get_t(item.body.d8, d10)

    # Block the machine for the assembly time
    yield env.timeout(1)
```

Quality check

During quality control, all figures that do not fulfill the quality requirements are rejected. The criterion used here is the tension, which must lie within a specified interval to be able to rotate the corresponding components against each other. For each tension, a check is performed to ensure that it lies within the specified interval. If not, then the *reject* attribute is set to *True*. Consequently, this item (including all assembled items) is removed from the process and is not added to the following store.

In addition, the id of the figures is stored in the global attribute *rejected_id* to identify them more easily. In the *following*, a method for identifying rejected items without global attributes is described.

```
def quality_check(env, item, machine, factory):

    # Limits for the tension
    t_min = 17.0
    t_max = 23.0

    def is_reject(t):
        if t <= t_min or t >= t_max:
            item.reject = True
            factory.rejected_id = item.item_id
            return True
        return False

    # Reject items and update profiling attributes
    if is_reject(item.t4):
        machine.r4 += 1
    if is_reject(item.t5):
        machine.r5 += 1
    if is_reject(item.t6):
        machine.r6 += 1
    if is_reject(item.t7):
        machine.r7 += 1
    if is_reject(item.t8):
```

(continues on next page)

(continued from previous page)

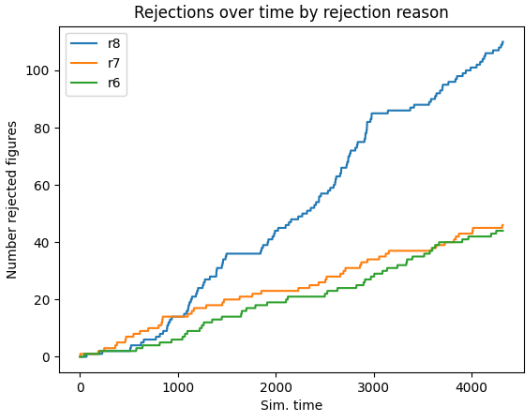
```
            machine.r8 += 1
        if is_reject(item.upper_limb[0].t2):
            machine.r2_1 += 1
        if is_reject(item.upper_limb[1].t2):
            machine.r2_2 += 1

        # Block quality machine
        yield env.timeout(1)
```

The diameters of the joints are distributed as follows:

$$d_i \sim N(40, 0.4)\ , i \in \{1, 3, 9, 10\}$$

$$d_i \sim N(42, 0.4)\ , i \in \{4, .., 7\}$$

For *d8*, a normal distribution is also assumed, but the mean diameter continues to increase due to wear during the injection. After 1500 injection processes, the mold is replaced so that the diameter starts again at 40. The following figure visualizes the behavior on the basis of the number of rejects corresponding to the rejection reasons *r6*, *r7*, and *r8*.



Note: If no attributes are changed at a station (e.g., *quality_check*), then setting the attribute *measurement* to true is recommended because the workpiece attributes will not be tracked at this station. This reduces the data usage.

Merge output data

As a standard, the simulation data for each simulation object (order, station, factory) is saved in its own file. The following text describes how these files can be merged in order to collect all information (d1,..,d10,t2,t4,..,t8) concerning a single figure for all figures in a time series. Data merging according to the underlying assembly structure is performed via the columns *item_id* and *comp*. Each order output whose workpieces represent assembly workpieces of at least one other order contains the column *comp*, which contains the *item_id* of the item for which the assembly item is assembled.

The following cutout of the csv file *arm.csv* indicates that the arm with the *item_id* 81 is mounted to an *upper_limb* item with the *item_id* 86. Likewise, *arm* 84 is assembled to *upper_limb* 87.

arm.csv

d2	d3	item_id	comp	station_id	time
39.869286	41.634014	5	nan	-1	1
⋮	⋮	⋮	⋮	⋮	⋮
40.347916	42.141098	95	nan	-1	9
39.856869	42.322212	81	86	0	9
40.032082	42.003731	70	75	1	9
40.015373	42.099506	73	76	1	9
40.532032	41.659058	84	87	0	9
40.01704	42.378185	103	nan	-1	10
⋮	⋮	⋮	⋮	⋮	⋮
39.311573	42.156609	47506	nan	-1	4319

In the file *hand.csv*, there are the two arms (item_ids: 82 and 85), which are mounted to the *upper_limbs* with the item_ids 86 and 87.

hand.csv

d1	item_id	comp	station_id	time
42.363113	6	nan	-1	1
⋮	⋮	⋮	⋮	⋮
41.320095	96	nan	-1	9
41.671734	82	86	0	9
41.903492	71	75	1	9
41.751251	74	76	1	9
42.235916	85	87	0	9
41.939342	104	nan	-1	10
⋮	⋮	⋮	⋮	⋮
41.505733	47507	nan	-1	4319

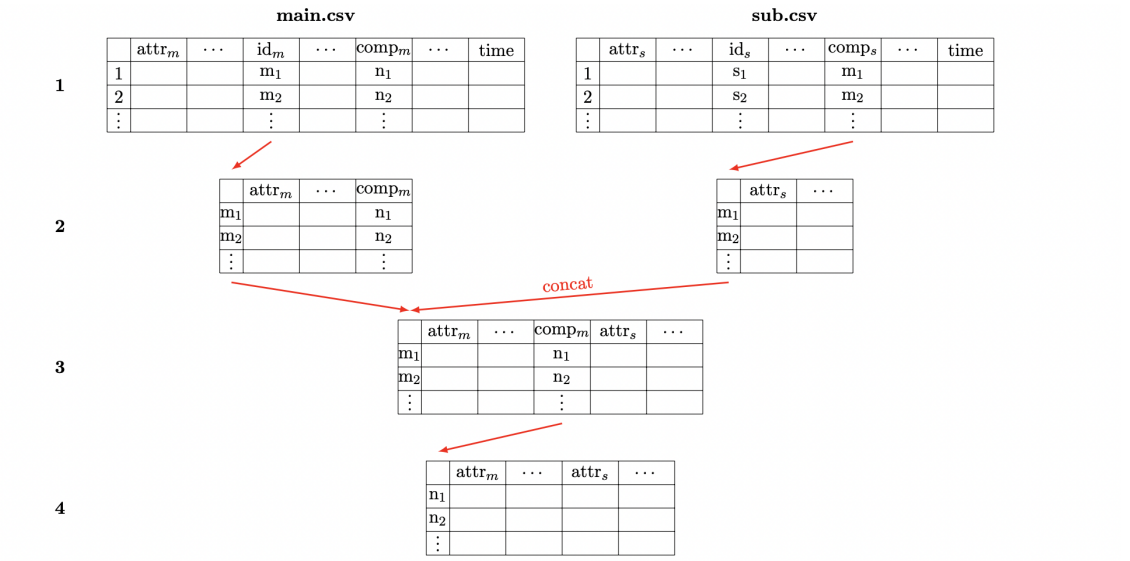
Finally, in the file *upper_limb.csv*, the two workpieces with item_ids 86 and 87 can be found. They are mounted on a *figure* with the *item_id* 77.

upper_limb.csv

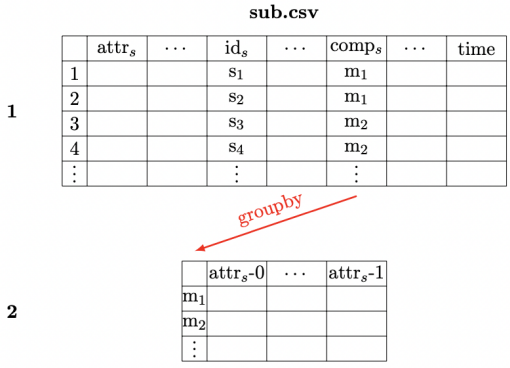
item_id	t2	comp	station_id	time
4	0	nan	-1	1
⋮	⋮	⋮	⋮	⋮
98	0	nan	-1	9
87	19.974035	nan	0	9
86	19.993654	77	1	10
87	19.974035	77	1	10
108	0	nan	-1	10
⋮	⋮	⋮	⋮	⋮
47508	0	nan	-1	4319

Based on the textually described context, the assembly structures can be automatically tracked. The following figure provides an overview of the required steps.

1. First, the csv files must be filtered so that only the rows containing items at the last station (or at the station where the current assembly structure is to be traced) are left
2. All columns that are not required are removed. Only the attribute columns are kept in files representing *sub* components, while the *comp* column is chosen as the index. In files belonging to *main* items, the *item_id* is used as the index, and all columns except the *comp* and attribute columns are deleted.
3. The *main* file is connected to the *sub* file (any number of sub files can be used) via the index (the concat method from the pandas library is recommended).
4. If the *main* item is assembled further, the *comp* column must subsequently be selected as the index to connect the new file again.



A particular detail must be taken into account. If the demand is greater than one, then the *comp* column contains the *item_id* of the *main* item multiple times (e.g., *upper_limb.csv - item_id*: 77). The following figure demonstrates how this case is handled. The file is split off (e.g., with the groupby method from the pandas library) using the *comp* column. Thus, the attributes are numbered to be able to differentiate them later. This ensures that the index set *comp* is unique and can be used to merge the files.



The following code block shows how to switch a csv file to state *1* from the first figure. The *get_df* method already considers the case of demands greater than one. Thus, the partial data sets are returned in a list.

```
def get_df(name: str, num_main_args: int, sub: bool = True, amount: int = None):

    index_col, labels = 'item_id', ['station_id']
    if sub:
        index_col, labels = 'comp', ['station_id', 'item_id']

    # set 'index_col' as row index, and remove the column 'time' for all assemble objects␣
↪by usecols (+3)
    iter_csv = pd.read_csv(path + name + '.csv', usecols=[i for i in range(
        num_main_args + 3)], iterator=True, chunksize=10_000, index_col=index_col)

    # build DataFrame and remove the columns 'labels'
    temp_df = pd.concat([chunk[chunk['station_id'] == station_id] for chunk in iter_
↪csv]).drop(labels=labels, axis=1)
```

(continues on next page)

(continued from previous page)

```
    # if there are multiple objects split the dataframe an return them as a list
    if amount is None:
        return temp_df
    return [temp_df.groupby('comp').nth(i).add_suffix('-%s' % i) for i in range(amount)]
```

The files created in this manner must be nested by hand according to the assembly structure. The following code block presents steps 2 and 3 for the final assembly step of a figure. Since the *figure* is the final assembly layer, the *comp* column does not exist in this file and cannot be set as the index. The used DataFrame *upper_limb* is previously generated according to the same logic.

```
figure = get_df("figure", 5, sub=False)
head = get_df("head", 1)
body = get_df("body", 5)
legs = get_df("leg", 1, amount=2)

figure = pd.concat([figure, head, legs[0], legs[1], upper_limb[0], upper_limb[1], body],␣
↪axis=1)
del head, legs, upper_limb, body
```

The following file depicts the results of this transformation. The row marked in yellow corresponds to the *figure* with *item_id* 77. When the values of this column are compared with the elementary csv files shown at the beginning, the values are observed to have been combined correctly. The file created in this way contains all 21 attributes of a figure per row.

merged_figures.csv

d2-0	d2-1	d1-0	d1-1	t2-0	t2-1	t8	d8	...	d10
39.869286	40.197178	42.363113	41.790482	20.120426	19.932732	18.410622	40.575005	...	41.407993
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮
39.584332	40.347916	41.810524	41.320095	20.011572	18.914202	17.642654	40.256783	...	40.925896
39.856869	40.532032	41.671734	42.235916	19.993654	19.974035	20.016668	39.691139	...	41.946579
40.01704	39.653412	41.939342	41.962372	19.999531	20.029491	19.669096	40.580677	...	41.889004
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮
39.825287	39.825287	42.476002	42.441421	20.00679	20.150728	19.846375	40.659237	...	42.194614

Identify rejected items

Finally, how rejected workpieces can be identified is described. In the quality_check function, the *item_id* of rejected items is stored globally. In the last step of the concatenation process described above, this global index set can be used to filter the items whose *item_id* appears in this set. Similarly, if the difference set is formed instead of the intersection set, nonrejected items can be obtained.

Alternatively, global attributes can be avoided if further process steps follow after the station at which workpieces are declared to be rejects. First, the item_ids of all workpieces created by a source (*station_id* = -1) are summarized in a set. Analogously, an index set can be created that contains all item_ids of items that have passed a specific station. By forming the difference set, one receives all item_ids of workpieces that represent rejects. With this set, as described above, the rejected workpieces can be identified.

PYTHON MODULE INDEX

e