A Library to Simulate Processes in the Factory Hall

Julia Gundermann Torsten Blochwitz

ESI Germany GmbH, Dresden, Germany {julia.gundermann, torsten.blochwitz}@esi-group.com

Abstract

The Modelica language is well suited to model systems with coupled discrete and continuous dynamics. This feature is crucial, if one wants to model the flow of items through manufacturing steps such as preparation, mounting, or transport in the shop floor. The library *ProcessSimulation* can be used to model such processes. By default, it omits the technical details of the process steps, and focuses on the flow of material items through the process steps. In addition to that, a base model to calculate the energy consumption in the different manufacturing steps is provided. It can be enriched with technical details of the components. The library can be used for the calculation of (net) energy consumption, but also for task planning.

Keywords: process simulation, energy consumption, Modelica library

1 Introduction

There are dedicated and advanced commercial tools for plant or manufacturing simulation and beyond, such as *Tecnomatix* (2023) (Siemens), *FlexSim* (2023) (FlexSim Software Products, Inc.) or *Arena Simulation Software* (2023) (Rockwell Automation). They base on discrete event simulation and cover a wide range of applications. To support modelling activities, some of these programs also integrate sophisticated 3D visualisation. There is also a Modelica library which can model discrete event simulation, since it adopts the Discrete EVents System (DEVS) formalism (Sanz, Urquia, and Dormido 2009; Sanz, Urquia, Cellier, et al. 2012; *DESLib* 2023). It offers a rich but complex functionality. The purpose of the library presented here is to evaluate the capabilities of simulating processes in the factory hall by means of the Modelica language with a simpler approach. It is shown that there are applications for which this way of modelling is sufficient. The Modelica language can cover both discrete and time-continuous processes. It is suited, if machines and transport means should not only be considered as event sequences, but can also be enriched with models of physical processes, i.e. mechanics and electrics which for example contribute to the consumption of energy. This publication introduces the library and its components and outlines two applications.

2 The Library

2.1 General

This library contains models of distinct groups to describe material flow through processes in the factory hall. These are

- storages to store items,
- machines to process items,
- transport devices to transport items,
- a tasks supervisor to model tasks and their preconditions on machines,
- an energy meter to observe energy consumption in the system.

The *material* which flows through the manufacturing line is identified by its amount as integer quantities. This is in contrast to the continuous material amount, which is used in the *Business Simulation Library* (2023). On the other hand, *material* is also not a set of single-wise identifiable items, like in DEVSlib (DEVSlib is a subpackage of *DESLib 2.0* (2023)). The Modelica language is not able to handle the generation and disappearance at simulation time. For that reason, DEVSlib implements an External Object in programming language C to handle a variable number of messages and entities. In our approach we only count the number of items. The item flow is handled by the definition of pairs with active and passive partners using specific connector types. There is always one active and one passive partner in the connection. The passive partner indicates its availability (`freeCapa`) and free/available `capacity`, the active partner triggers the `handOver` and defines the handed quantity (`handedCapa`). The *active* connector is defined as follows:

```
connector MaterialA
  "Active Material Handover"
  input  Boolean freeCapa;
  output Boolean handOver;
  input  Integer capacity;
  output Integer handedCapa;
end MaterialA;
```

The *passive* connector `MaterialP` is defined in the same way with input and outputs exchanged. Figure 1 shows the connectors. With the Boolean parameter `handOver` the events are triggered in which one or more items are handed over from one component to another. In this way,

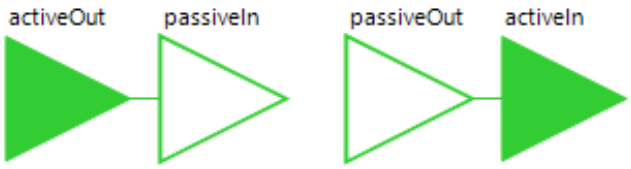

Figure 1. Connector pairings of the process simulation library. There is always one active and one passive partner in the connection.

the flow of material is as fast/efficient as locally (i.e. in a sequence of two consecutive components) possible. It should be mentioned that this modelling technique creates a lot of events during simulation. The number of variables which change as a consequence of these events is low, and their values are calculated from simple equations. Hence, the duration of a single event iteration is determined by the solver's performance and in general short, however the high number of events in total might affect simulation time.

2.2 Storages

There is a generic component named `Storage` which models the storage of items, e.g. in a storage room, but also on a wagon, or a dedicated place upfront a machine. It can be parametrized with an initial capacity and a maximal capacity. The storage is passive regarding material flow. This means it indicates its free capacity, but the process of handing over material is triggered from the connected transport device (see next section). There exists a variant of the storage named `MergeStorage`. This can be used, if a certain ratio of components of two different types (e.g. four wheels and a chassis) has to be available before processing the next step (e.g. transport to an assembling machine). Figure 2 shows the icons.

2.3 Transport Devices

There are three types of transport devices. They share the commonality that all of them actively trigger the loading and unloading of items. Items are loaded if the preceding storage has enough items to transport, and they are unloaded if the target has enough free capacity. Otherwise, the transport stops.

Plain Transport

This transport device loads a parametrizable number of items within a `loadTime`, transports them within a `transportTime` and unloads them at a connected target. The transport runs either up to a maximum defined number, or indefinitely as long as there are items to transport.

Conveyor

This transport device models a conveyor belt. It needs `processTime` to transport one item from start to target, and loads up to a parametrized `maxCapa` of items (all separated by `processTime/maxCapa`).

Shared transport device (e.g. automated guided vehicle AGV)

This device models the transport as defined in connected `TransportTasks`. All the information about the transport task is defined in the connected device. This includes

- the number of items to carry
- the number the device can carry at once
- the time to load the device
- the time to transport the loaded items
- a condition when the transport task is prepared to be run (e.g. to model that a transport task will only be started if there is enough free capacity in the target storage).
- a unique id

Each transport task is connected to a `SharedResource` by a specific connector pair, in which the shared resource receives each task's `id` as well as its status (prepared, finished), and sends the `currentTaskNr` to all transport tasks.

The shared resource exists in two variants - one runs the tasks "AS PLANNED", i.e. as defined by the sequence of task IDs in the shared resource. The other runs the tasks "ON DEMAND", i.e. whenever a transport task becomes status "prepared", it will be scheduled as the next transport task. If several tasks become prepared at the same time, they will be scheduled in order of increasing task id.

Figure 3 shows the icons of the transport devices. All transport devices display the number of currently transported items. In addition to that, each transport task shows

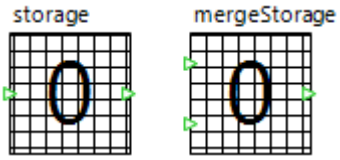

Figure 2. Storage and merge storage component. The number on the icon displays the number of (merged) items which are currently in the storage.

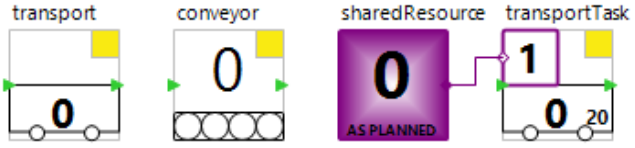

Figure 3. Transport devices of the process simulation library. From left to right: transport, conveyor, and shared resource with transport task.

DOI
10.3384/ecp204357

in its top left corner the unique task ID, which is highlighted in violet if the task is currently running. The bottom right corner shows the total number of items to transport. The icon of the shared resource shows the ID of the currently active transport task.

2.4 Machines

The library contains a plain `Machine`, which processes items. This refers to any type of process - mounting, drilling, packing, sawing, ... The plain machine runs down a `setupTime` to prepare the machine, and a `cycleTime` to process one item. It processes `nMax` items, then it stops. An item is taken and processed as soon as it is available in the connected input storage, and handed to the output storage if there is free capacity. Two output variables indicate the progress of the machine `setup`, and the `progress` of processing the current item, respectively. A `machine` is always located between two storages, one from which the items are taken, and a second one which is filled with items. Figure 4 shows two storages connected by a machine which processes items. The numbers displayed in the storages and machine vary over time and show the number of stored or processed items, respectively.

2.4.1 Machines with Tasks

Whereas this plain `Machine` runs only one process to its end, there are four more advanced machine types in the library (cf. Figure 7). They all share the following base structure: they all are vectorized versions of the plain `Machine` connected with start and target storages (configuration as in Figure 4). This structure is used to define and simulate a sequence of tasks. For each of these tasks material is taken/delivered from/to its dedicated storage. Each task has its own parameter values for number of items, setup time and cycle time.

The storages before and after the machine are empty, and have to be filled/emptied by other active processes (i.e. transport, or predecessor machine). For example, in Figure 5, transport devices have been connected. The passive connectors to the input and output storages have the size `n1`, `n2`. The values of these integer parameters are zero by default, and grow with the number of connected components (due to the Dialog annotation `connectorsizing=true`). The machine needs the same number of connections on both sides, an assert is thrown, if `n1<>n2`. The sequence of connecting the components is important. The first connected transports "belong" to the first set of task parameters, and so on.

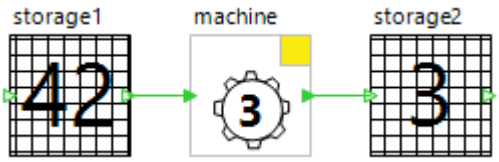

Figure 4. The Machine component, connected with an input storage and a target storage.

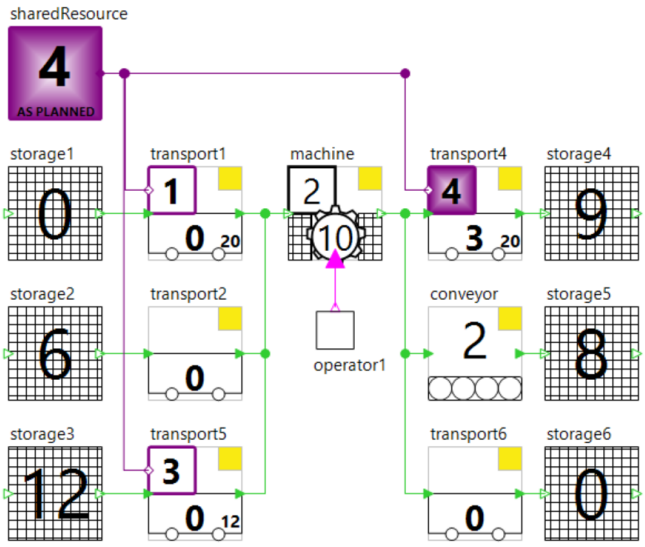

Figure 5. The Machine with local tasks. The example contains three tasks on the machine, hence three input and output processes (all transport) have been connected. The numbers on the machine icon indicate that it currently runs task 2, in which 10 items have been processed.

There are two additional features to highlight:

- The model of the machine with tasks contains an array-parameter `taskIDsC`, with which one can define positive integer IDs for all tasks (C stands for connected). In addition to that, there is a second parameter `taskIDsO` which is a reorder of the task IDs (O = ordered). This parameter allows to change the order of the tasks. Since the start or finalization of tasks can depend on transport processes, a reordering could improve total process time.

- The machine has an additional connector, to which an `Operator` must be connected. The machine runs only if the operator is available. With this additional condition shift durations or breaks can be modelled.

The machine variant with all this functionality is named `MachineWithTasks`. Further variants are extensions of this type and are explained below.

2.4.2 Machines with global tasks

If one wants to model tasks on different machines, which can depend on each other, there is a dedicated type named the `MachineWithGlobalTasks`, which is an extension of the `MachineWithTasks`. A task on this machine type will only start when all preconditioned tasks (on the same or other machines) have finished. Therefore, this machine type contains an `outer GlobalTaskList tasks`, which parametrizes each task's preconditions and controls their fulfilment. To do so, the global task list contains the number `n` and IDs of all tasks. It also contains a matrix parameter to define each task's

`preconditions[n,p]`, to denote up to `p` preceding tasks per task. Figure 6 shows the icons of the machine and the tasks component.

2.4.3 Machines with prefilled storages

There is a variant of the machine with tasks, which exists for both the machine with local tasks and global tasks. At simulation start, this variant has all start-storages in the machine filled with the capacity as defined in the parameter vector `nMax`. The maximum capacities of the target storages are set to the same values, respectively. This machine model no longer contains any (passive) material connectors. This model can be used, if only planning the processes on the machines is of interest. To reflect the auto-filled storages in the types, the type names have an "SC"-suffix (for self-contained), i.e. they are named `MachineWithTasksSC` and `MachineWithGlobalTasksSC`, respectively.

The icons of these four variants, which are all possible combinations of the features "local or global tasks" and "with connectors or prefilled storages", are displayed in Figure 7. Section 3.2 shows a small example of this functionality.

2.5 Energy Meter

An additional feature of this library is the calculation of total power and energy consumption. Most of the components in the factory hall consume or provide power. This is reflected in the library, all transport devices and the machine model contain variables and equations to calculate their power consumption. We define a global `EnergyMeter`, to easily sum them up to determine the total power and energy consumption, and use the Modelica `inner/outer` connection to collect all power terms from all components. This avoids the manual connection of all consumers or producers of power with the `EnergyMeter`, which would reduce clarity on the diagram view of the model. To facilitate the collection of power terms, the library defines an `EnergyContributor`, which is the base type of the contributing components. Between all energy contributors and the outer meter, a connection with a flow variable is created. Each energy contributor adds its power `P` to this connection automatically, the value of the energy meter's connector is the negative sum of all these contributions (since flow connections establish a sum-to-zero coupling), hence the total power consumption is determined

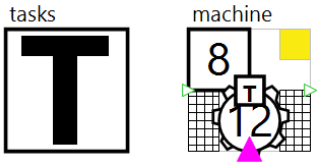

Figure 6. The Machine with global tasks, and the global task list. The machine has the same functionality as the machine with local tasks, but it needs an `outer` global task list, in which task's preconditions can be parametrized.

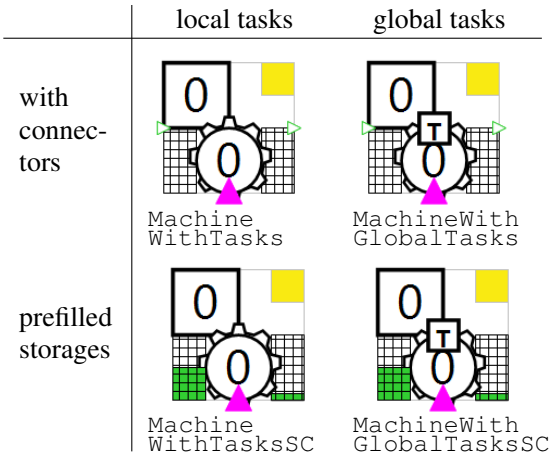

Figure 7. Variants of machines with tasks

by the negative value of the connector. Listing 1 shows part of the energy contributor's and energy meter's definition. Figure 8 shows the icon of the energy meter.

Listing 1. Modelica code snippet outlining the energy meter and contributor

```
connector FlowCtr
        "Flow Connector for Meters"
 flow Real i "Flow Variable";
end FlowCtr;

model EnergyMeter "Energy Meter"
  Real P "Power";
  Real E "Energy";
 protected
  FlowCtr pc "Power Collector";
 equation
  P = - pc.i;
  der(E) = P;
end Energy;

partial model EnergyContributor
        "Energy Contributor"
  outer EnergyMeter energyMeter;
  Real P "Power
        - eqn. defined by derived type";
 protected
  FlowCtr power;
 equation
  power.i=P;
  connect(power, energyMeter.pc);
end EnergyContributor;
```

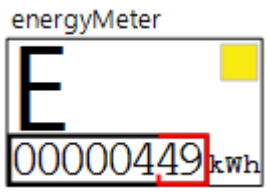

Figure 8. The Energy Meter component, displaying the total energy consumption like an electricity meter.

It shall be mentioned that the defined connector

DOI
10.3384/ecp204357

`FlowCtr` is not conform with the requirement as stated in the Modelica specification, section 9.3.1. (Modelica Association 2023): The connector is unbalanced, i.e. the numbers of flow variables is not equal to the number of variables that are neither parameter, constant, input, output, or stream. According to (Olsson et al. 2008) this prevents any model using this connector from being locally balanced. However, this request was derived for systems with multiple components of the same type. When using the `ProcessSimulation.EnergyMeter` and (extended) `EnergyContributor`s, any allowed configuration contains exactly one energy meter and zero or more energy contributors. Any such combination has a balanced number of unknowns and equations. Furthermore, the `FlowCtr`s `pc` in the meter and `power` in the contributor are declared as `protected`, which at least produces a warning in some Modelica compilers. Here it is used to prevent any component which is neither meter nor contributor from being connected to the power balance. Defining the connectors as public but omitting annotations for the Placement could have the same effect, at least in the diagram view of the model.

2.6 Comparison to existing libraries

As mentioned, there are Modelica libraries which cover applications that are also adressed by the Process Simulation library (PSL): the *DESLib* (2023) (or *DESLib 2.0* (2023)) and the *Business Simulation Library* (2023). In this section, the differences are outlined. This is not meant to be a comprehensive summary of the other libraries' functionality, merely only supposed to help figuring out for which application the Process Simulation library is advantageous or sufficient, or when to rely on existing libraries.

The Business Simulation Library (BSL) contains - beside many others - classes which are comparable to those in the PSL - e.g., `Oven` (as a variant of a machine), `Conveyor` or `MaterialStock`. Different to the PSL, it uses real-valued material flow rates instead of integer numbers of material items. This avoids events due to handovers between components. Besides, the details of class parametrization differ. To name a few differences: the BSL `Oven` prepares batches of parametrizable size, there are also parameters for the setup and process of the batch, and a loading time. This is different from the PSL `Machine`, with has a "batch-wise"(`nMax`) setup time, but element-wise process times. The BSL `Conveyor` uses the Modelica `delay` operator which creates a time lapse between the inflow and outflow of material. This delay is triggered within a `when sample(..)` statement with a model-wide sampling frequency which is by default 16/s, i.e. it creates 16 time events per seconds, independent of the flow rate. This differs from the PSL `Conveyor` which creates (2 `maxCapa/processTime`) state events per second.

The DEVSlib (sub-package of *DESLib 2.0* (2023)) implements the Parallel Discrete Event System Specification (PDEVS) formalism (Zeigler, Prähofer, and Kim 2000), which is powerful to describe (parallel) discrete event systems together with continuous state systems. To create such a system, an *Atomic DEVS* is defined. The package provides an `atomicDraft` model, which implements this Atomic DEVS, and a guide how to create own models from a duplicate of this, to model a DEVS system. Due to the underlying formalism the application range is big, among the provided examples are controllers coupled to physical systems, game of life, and a supermarket model.

In the process oriented DEVS formalism, "systems are described from the point of view of the entities that flow through them using the available resources" (Sanz, Urquia, and Dormido 2009). For assembly line models this means that it is possible to track and identify each manufactured item during its flow through the line. This differs from the approach in the Process Simulation library, where the manufactured items are counted as integer numbers. There are applications where this is sufficient, e.g. to determine the energy consumption in an assembly line.

Furthermore, DESlib provides mechanisms to introduce stochasticity into the models (by the package RandomLib). As stated in (Sanz, Urquia, and Dormido 2009), process-oriented models are usually stochastic, which is why the generation of random numbers is necessary. Models created with the Process Simulation library are fully deterministic, material in assembly lines or tasks on machines run as planned. In the existing library components, random effects could modify the conveyor belt's velocity, machine/task process times, or others, which cause delays or accelerations in different manufacturing steps, and would result in variations of total process times. Since no systematic analysis of the random effects on the system's behaviour can be provided, and the result of single simulations with random effects are hard to interpret, such effects were not implemented in the current version. However, the result of the "deterministic" simulation provides all information to determine mean process times or energy consumption.

The currently available *DESLib 2.0* (2023) library version does not provide an example for an assembly line, i.e. no class models such as machines, stocks, conveyors. Hence a comparison of functionality or performance is not immediately possible.

3 Examples/Use Cases

In this section, two examples which outline the different facets of the Process Simulation library are presented.

3.1 Example 1: Energy analysis of the manufacturing of a three-wheeler

Figure 9 shows a screenshot of the (simplified) production line of a three-wheeler. As several of the components are `EnergyContributor`s, the component `inner Energy energy` is needed in the model. The contributions to the

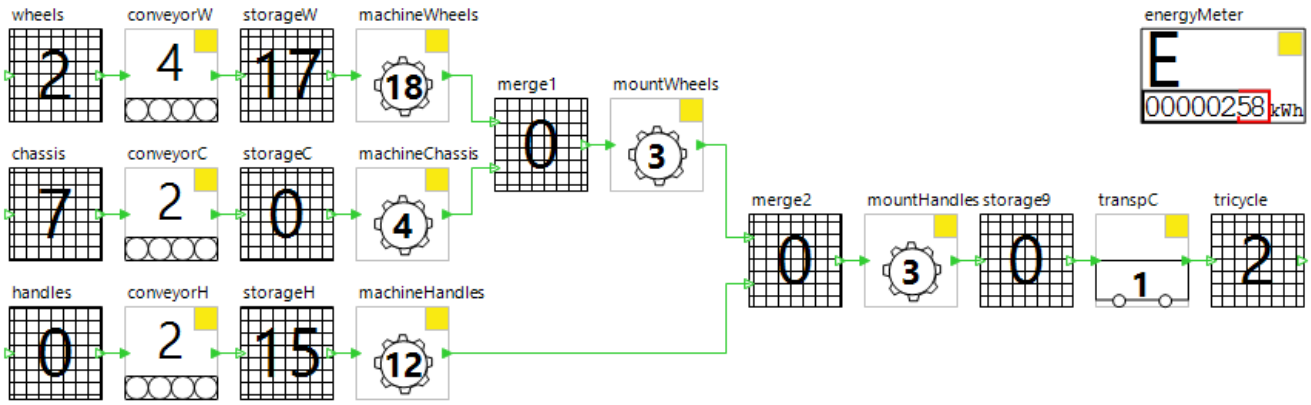

Figure 9. Simplified production line of a three-wheeler to illustrate the usage of the energy consumption. It contains pre-processing of wheels, handles and chassis, followed by two mounting machines. The transports to and from the machines are by either conveyors or plain transport (e.g. carrying). All components with a yellow box in the top right corner contribute to the overall power and energy consumption.

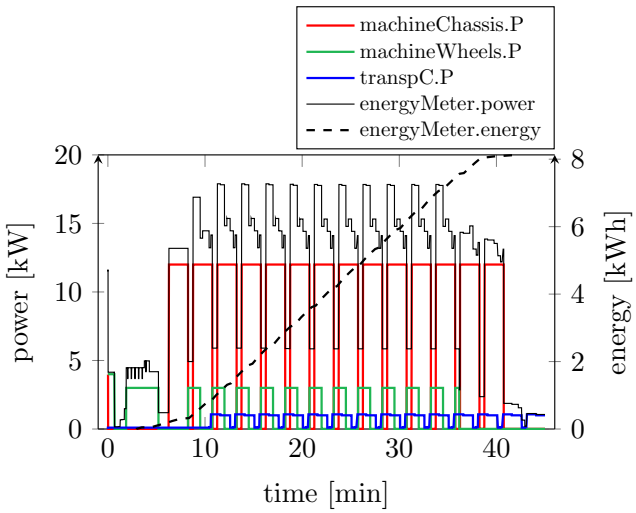

Figure 10. Power consumption of the different components of the three-wheeler production line. Shown are only the consumptions of one machine, one transport and the conveyor. Besides that the plot contains the total power and total energy consumption.

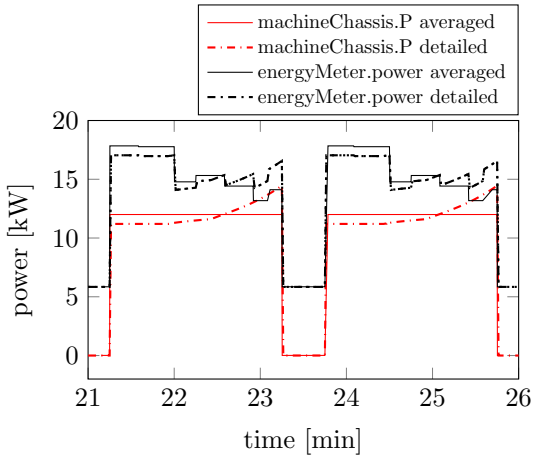

Figure 11. Power consumption of the three-wheeler manufacturing line with a detailed power modelling in the `machineChassis` component in comparison to the averaged original version. The figure shows two cycles of the machine only.

power (and energy) consumptions are parametrized within the different components. Figure 10 shows the power consumption of selected components and the entire manufacturing line. One can see that the power consumption has peaks of 17.8 kW, the main contribution stemming from the `machineChassis`. The manufacturing of 14 three-wheelers consumes an energy of 8.13 kWh in total.

One possibility to examine the power consumption in more detail is to extend the elementary components with more comprehensive models of the underlying processes. In the example considered here, the main energy contributor of Figure 9 is `machineChassis`. To enhance the calculation of the power consumption for this component, a Functional Mockup Unit (FMU) which models a detailed manufacturing process with non-constant power consumption, was imported to an extended copy of the `Machine` class. The original `machineChassis` component is replaced by a component of the new class.

Figure 11 shows the variation of power consumption of the modified machine in contrast to the previously constant value. This affects the peak power consumption, it reduces to 17.1 kW, but has no influence on the total energy consumption of the production line.

3.2 Example 2: Planning of tasks on machines and shared transport devices

This second example is created sequentially to illustrate the different machine variants, used together with other library types.

DOI
10.3384/ecp204357

a) We start with the simple scenario of a single machine, on which three tasks have to be run in sequence. We do not care for the transport to and from the machine, we simply assume the material is there. We want to know how long it takes to run these tasks. The component to model this is the `MachineWithTasksSC`. Table 1 shows the details of the tasks. Figure 12 shows the diagram view of this

Task ID	nr. of items	cycle time*	setup time*	machine	precon-ditions
1	5	45	20	1	-
2	3	15	20	1	-
3	10	12	20	1	1
4	50	2	10	2	2
5	35	3	10	2	-
					(*in minutes)

Table 1. Details of the tasks in Example 2.

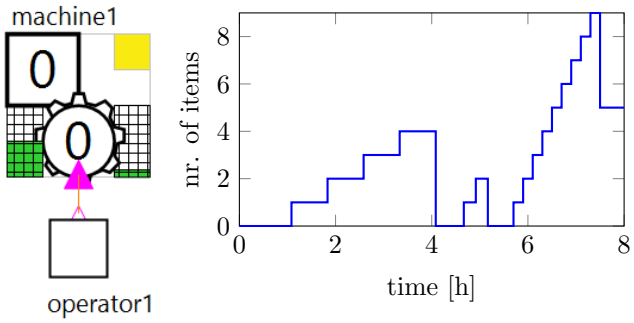

Figure 12. Example 2a: One machine with a sequence of three tasks, and a result window showing the processed items per task.

example, and a result plot with the number of processed items. It should be mentioned that as soon as the last cycle of a task is completed the machine switches to the next task, hence the variable displaying the processed items per task peaks at (`nMax`-1) items in each task (in the example 4, 2, 9). From the graph one can read that the three tasks are finished after 7.5 hours.

b) Now assume there are more tasks on a second machine, and there are some preconditions. Task 3 can only be run after task 1, task 4 only after task 2. Again, refer to Table 1 for the task details. To model this, we have to use the type `MachineWithGlobalTasksSC`, and the `inner GlobalTaskList tasks`, the latter parametrizes and controls the preconditions to the tasks. Figure 13 shows the model.

In this example the orders of the tasks on the machines influence the total process time. With this configuration one could test various task orders to figure out which one is the fastest. Technically this is done by modifying the value of the parameter `taskIDsO` on each machine to define the order of the tasks. With this small number of tasks and precondition one can find optimal solutions by choosing sensitive orderings. It makes sense to run task 5 on

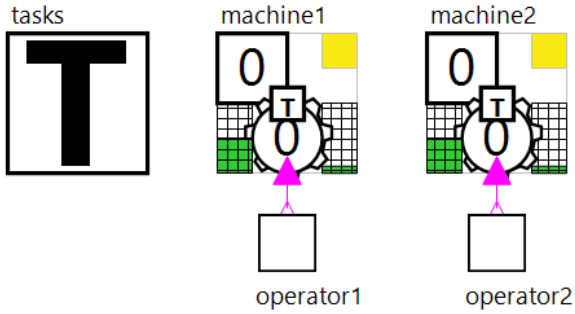

Figure 13. Example 2b: Two machines with five tasks in total.

machine 2 first, since task 4 has to wait for the finalization of task 2. Furthermore, task 3 should be executed as last one on machine 1, since tasks 1 and 2 are preconditions to other tasks. With a small number of simulations, one finds that the following task orders result in the same and fastest total simulation time, which is 7.5 hours.

	Machine 1	Machine 2	Total time
Variant 1	{1, 2, 3}	{5, 4}	7 h 30 min
Variant 2	{2, 1, 3}	{5, 4}	7 h 30 min

c) As a third step we want to consider not only the processes on the machines, but also transport processes to and from them. Figure 14 shows the model. The machine types have been changed to `MachineWithGlobalTasks`, and have passive material connectors. The transports to the machines are all provided by an AGV, five transport tasks have been created. The IDs of the transport tasks (displayed in the top left corner of the `transportTask` components) are equal to the machine tasks which follow - this is merely an (intended) coincidence. Transport task 1 takes 11 minutes per component, whereas the others are all faster (3.5 minutes). It should be determined which transport task sequence and machine task sequence leads to the fastest finalization of all machine tasks. To answer this question, we analyse reasonable transport task orders together with the two fastest machine task variants. The following considerations help excluding some of the 120 variants: Firstly, the preconditions of machine tasks should be the same for the transport tasks (this transfer of numbers works here since we chose the same numbering, and only have transport tasks *to* machines). Secondly, the order of machine tasks should be kept in the order of transport tasks (i.e., transport task 1 before 2 before 3, and 5 before 4, if to test with machine tasks variant 1). Taking these restrictions into account there are 7 transport task orders to test for variant 1, and 9 for variant two.

From simulation results one can read that the transport task order {1,5,2,3,4} together with machine task variant 2 leads to the same total time to finalize all machine tasks, which is 7.5 hours. See Figure 15. The first task (task 1) on machine 1 is not delayed by transport task 1, since the transport of the first item has finished before the machine setup. Machine task 5 is delayed but is not precondition

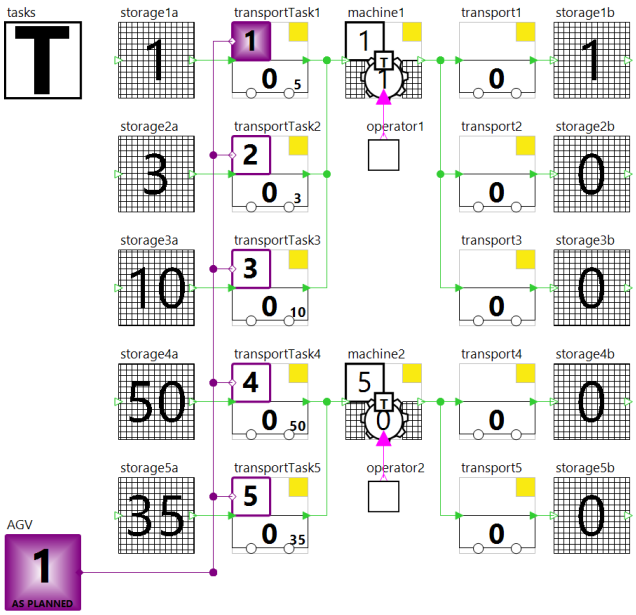

Figure 14. Example 2c: Two machines with five tasks, and transport processes to and from the machine. The transport to the machine is provided by an AGV.

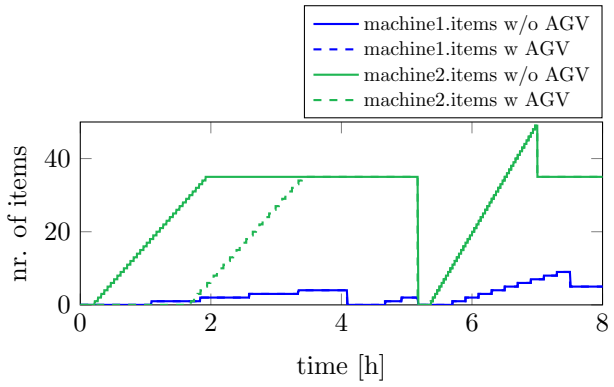

Figure 15. Example 2 b/c: Processed items on machines 1 and 2 in the variants of the model without transport and with transport (AGV) included.

to any other task. By the time machine task 5 has finished on machine 2, the necessary machine task 2 on machine 1 has finished such that machine task 4 can start right away. In total, the AGV transport does not affect the machine task finalization for this combination, and all tasks can be finished within an eight hour shift.

With this example the different variants of the machines were illustrated, in combination with the usage of the shared resource as a transport device. A combination with other transport means, like in subsection 3.1 is also possible. Of course, for more complex scenarios finding the optimal sequence of machine and/or transport tasks becomes challenging. In such cases, a dedicated optimizer should be used, it is beyond the scope of this library. The simulations are very fast, and a reordering is only a reparametrization of the model. Model creation, modification, simulation, and request of results is all possible via a scriptable interface, which eases the connection to other tools like optimizers.

One realization of such a task plan simulation in the configuration as described in example 2b was developed in the DIMOFAC project[1], where the challenge was to optimize tasks on several machines. The information about tasks and machines is provided in asset administration shells (AASs). A dedicated optimization tool for task planning (Kousi et al. 2019; Evangelou et al. 2021) reads this information, determines candidates of task orders and generates simulation requests for all of them. Based on these requests, a Modelica model is created (in *SimulationX* (2023)), which reads the task and machine specific information from their AASs. After the simulation, metrics like net machine utilization or process time are returned to the optimization tool, which evaluates this information to create new simulation requests and finally identify the optimal task schedule.

4 Summary and Outlook

In this publication the *ProcessSimulation* library was presented. It can be used as a low-level entry point to model material flow through a production line, to evaluate the performance of a manufacturing system in terms of energy and machine or transport task order. The library can be extended if needed, e.g. by multiple-merge storages, or machines with multiple material outputs. Besides, the consideration of random effects on the process times remains an open issue. Regarding the machine types the library can flexibly be extended such that the calculation of the energy consumption becomes more fine-grained. One could use this to identify/reduce maximum demand loads to the power station which provides the energy in a manufacturing line, which helps to save costs. The concept of an energy meter which monitors the consumption (or production) of energy can be transferred to other electricity net types (230V, 400V, high voltage), heat, water, or compressed air consumption.

In 3.2, a scenario to optimize production plans was illustrated. A separate optimization tool was used to determine the optimal schedule of tasks. For a closer integration to the simulation environment (*SimulationX*), it remains future work to develop python scripts (e.g., using dedicated libraries, such as *Python MIP (Mixed-Integer Linear Programming) Tools* (2023)) for the optimization of tasks on machines and transport devices. The necessary python interface exists in *SimulationX*.

The modelling of the material transport between components was realized by connectors which trigger the handover immediately when material and capacity is available, and all components work with pre-planned velocity. A remaining task is to extend this concept with signal interfaces to externally control the processes - handover, status of machines, transport/manufacturing velocities. This

[1] `https://dimofac.eu/` (accessed on June 1, 2023)

DOI
10.3384/ecp204357

could be used for virtual commissioning of a manufacturing line, to evaluate the reliability and robustness of control signals for the line on a simulation model of it.

Acknowledgements

This research has been supported by the European Union's Horizon 2020 research and innovation programme under the grant agreement No 870092, the project DIMOFAC (Digital and Intelligent MOdular FACtories).

References

Arena Simulation Software (2023). URL: https://www.rockwellautomation.com/en-us/products/software/arena-simulation.html (visited on 2023-05-23).

Business Simulation Library (2023). URL: https://github.com/modelica-3rdparty/BusinessSimulation (visited on 2023-05-23).

DESLib (2023). URL: http://www.euclides.dia.uned.es/DESLib/ (visited on 2023-08-15).

DESLib 2.0 (2023). URL: http://www.euclides.dia.uned.es/vsanz/files/DESLib-2.0web.zip (visited on 2023-08-15).

Evangelou, George et al. (2021). "An approach for task and action planning in Human–Robot Collaborative cells using AI". In: *Procedia CIRP* 97. 8th CIRP Conference of Assembly Technology and Systems, pp. 476–481. DOI: https://doi.org/10.1016/j.procir.2020.08.006.

FlexSim (2023). URL: https://www.flexsim.com/manufacturing-simulation/ (visited on 2023-05-23).

Kousi, Niki et al. (2019). "AI based combined scheduling and motion planning in flexible robotic assembly lines". In: *Procedia CIRP* 86. 7th CIRP Global Web Conference – Towards shifted production value stream patterns through inference of data, models, and technology (CIRPe 2019), pp. 74–79. DOI: https://doi.org/10.1016/j.procir.2020.01.041.

Modelica Association (2023). *Modelica – A Unified Object-Oriented Language for Systems Modeling. Language Specification Version 3.6*. Tech. rep. Linköping: Modelica Association. URL: http://www.modelica.org.

Olsson, Hans et al. (2008). "Balanced Models in Modelica 3.0 for Increased Model Quality". In: *Proceeedings of the 6th International Modelica Conference*. Linköping University Electronic Press. LiU Electronic Press, pp. 21–33. URL: https://elib.dlr.de/55892/.

Python MIP (Mixed-Integer Linear Programming) Tools (2023). URL: https://pypi.org/project/mip/ (visited on 2023-06-05).

Sanz, Victorino, Alfonso Urquia, François E. Cellier, et al. (2012). "Modeling of hybrid control systems using the DEVSLib Modelica library". In: *Control Engineering Practice* 20, pp. 24–34.

Sanz, Victorino, Alfonso Urquia, and S. Dormido (2009-10). "Parallel DEVS and Process-Oriented Modeling in Modelica". In: *Proceedings of the 7th International Modelica Conference*. Linköping University Electronic Press. LiU Electronic Press. DOI: 10.3384/ecp09430104.

SimulationX (2023). URL: www.SimulationX.com (visited on 2023-06-07).

Tecnomatix (2023). URL: https://plm.sw.siemens.com/en-US/tecnomatix/products/plant-simulation-software/ (visited on 2023-05-23).

Zeigler, Bernard, Herbert Prähofer, and Tag Gon Kim (2000-01). *Theory of Modeling and Simulation 2nd Edition*. San Diego, CA: Academic Press.