Hans-Jürgen Schröttner, BSc.

Modular Production Processes

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Nikolaus Furian

Institut für Maschinenbau- und Betriebsinformatik
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Siegfried Vössner

Krottendorf, December 2020

This document was written in Overleaf compiled with pdfLaTeX2e and BIBER and is based on the LaTeX template from Karl Voit (TU Graz).

Translational aid by Dictionary.Com [Dic20] and DictLinguee [Lin20].

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used to best of my knowledge and belief.

After it has become common to look for mistakes in the past of famous personalities, especially to harm them by toring apart diploma and doctoral thesis, and it may be that I become more famous in a musically and politically way, I further declare, that all not-cited parts stem from my own intelligence. I have no intent, neither inventive nor financial, to knowingly use someones knowledge without citing.

The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date

Signature

Dedication

To mum and dad who always supported me in my educational career, despite the rocky road with all its obstacles.

Abstract

Since the beginning of the second industrial revolution through the development of the conveyor belt by Henry Ford, the basic idea of production has not changed ever since.

Due to changing requirements, nowadays one has to face several different drive technologies for one type of vehicle[1] and furthermore a number of varying equipment.

This makes it hard to plan production lines efficiently containing all necessary work stations for all types, because of the fact that most of them will be skipped and cause unused production time.

The so-called **modular production** deals with this problem and tries to find a way to finalize a vehicle with specialized equipment in the most efficient way.

This thesis describes the status quo in modular production, ideas how to solve problems occurring in production lines, and resulting practical experience planning a modular production line using Petrinets. Furthermore the practical implementation of this thesis also deals with simulation of modular structures in automotive industries.

[1] to keep the expressiveness of this thesis general, a product the production is about will be named vehicle

Acknowledgements

I want to thank *Univ.-Prof. Dipl.-Ing. Dr.techn. Siegfried Vössner* for the possibility to write a Master Thesis at the *Institut für Maschinenbau- und Betriebsinformatik*. Due to the fact that he is as fond of styrian folk music as me, we started talking at an event at the Vienna Hofburg and discussed possible upcoming master's thesis topics.

Special thanks go to my supervisors *Ass.Prof. Dipl.-Ing. Dr.techn. Nikolaus Furian* and *Dipl.-Ing. Dr.techn. Dietmar Neubacher* who stood by with advice and assistance. Furthermore they were a big help when discussing challenging parts of the thesis and when I needed help with the implementation of the simulation framework Mr. Furian invented.

I also want to thank *Prof. Franck Pommereau* of the *Université d'Évry* in France. He helped me at a technical dead end using the Python library *Snakes* for calculating and drawing Petrinets.

Contents

List of Figures

List of Tables

List of Algorithms

Part I.

Initial Situation

1. Initial Situation

1.1. Historical Overview of Production

Production has accompanied mankind from the very beginning. However, one can only speak of automated production since the industrial revolution in the 1760s. More precisely, it was *Henry Ford* over a century and a half later who invented automated assembly lines through the conveyor belt. This point in time will later be count to the period of *Industry 2.0* (figure 1.1).

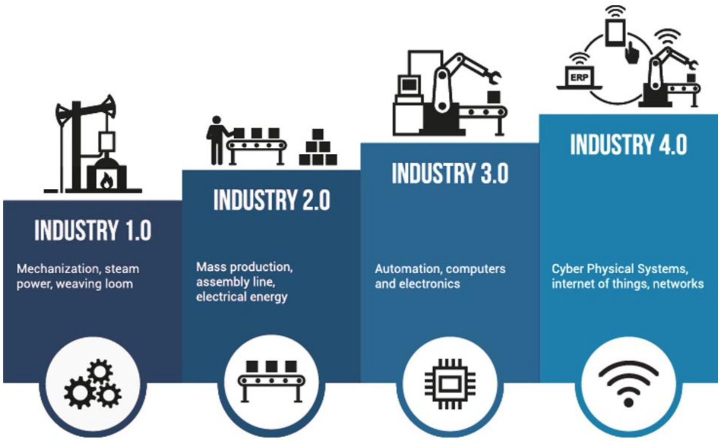


Figure 1.1.: History of Industry [Sup]

Thematically this is still far away from *modular production* but it once formed the basis for what nowadays is called *automotive industry*.

Apart from special forms of production like *custom-made*, in the major case since then every resource (no matter whether workmen, machines or tools) in a typical assembly line has precisely to do the same comprehensible job of his qualification over and over again.

This has several advantages:

- A resource (worker, machine, tool) does exactly the work its qualification is made for
- The quality of the assembled product increases

- The throughput rate increases
- The logistics are relieved as the components, materials and working resources have to be delivered to a certain work place
- The whole production line is more manageable

This represents the status quo in automotive industries. But nowadays a manufacturer has to face the challenge of a variety of variants and has to think of how to use these described advantages in a new way of production.

1.2. Case Study Magna

Magna Steyr[1] is known as an assembler company for different well-known car manufacturers. In this case study the **Mercedes G class** (in Austrian slang still known as **Puch G**) is used as an example car with a variety of challenging variants.

At present there are following different types available:

- Civil version
- Military version
- Luxury AMG improved version

One can imagine how complicated it can get to develop an assembly line for these different types.

And each type can be build up with one of these existing or future power trains:

- Diesel or Gasoline
- Hybrid
- Pure electrical
- Hydrogen

When considering also the different variants of power train of these types, a production line planning gets complex.

Now the question is, how this complexity can be solved in both a most effective and efficient way.

[1] Magna Steyr https://www.magna.com

Therefore the concrete questions of this master thesis are:

> *How can an existing assembling process be optimized through modularization?*
> *How can Petrinets help in design and execution?*
> *How can a simulation with different scenarios be build up to gain decision making information?*

The result should help to answer questions and making decisions about changing production strategies.

Part II.

Theoretical Basis

2. Theoretical Basis

The theoretical part describes the basis of partly independent topics and already explored data, which joined together represent the necessary knowledge for the research part and creating an own methodology.

2.1. Modular Production

Modular Production basically is a way to assemble parts of a product (in this certain case of a vehicle) in locally independent workplaces. Seen more specifically these parts also can be assembled in a random sequence which makes the assembling a bit of a counterpart to a typical production line.

In the research community, foremost for industry, it has gained some interest in the last decade. Even if until now it was more a theoretical subject, there are several approaches how to challenge the new requirements.

In 1997 Rogers and Bottaci [RB97] described the reason and need of a new production system with the emergence of new requirements in automotive industry. They name the variety of product variants and demands, shorter product life-cycles and increasing competition as the main reasons why a conventional production line reaches its limits.

Lara, Trujano and Garcia-Garnica [LTG05] name the emergence of new technologies and the globalization with a stronger competition as a reason for an upcoming big restructuring process which requires more modularity.

The authors of *Modularity concepts for the automotive industry: A critical review* [Pan+08] describe it in a similar way. They say analogous:

> *There will be a change from high production volume and low flexibility to products with a high variety of variants produced in small numbers.*

They further describe that the switch to modular production is already taking place in automotive industry, as parts for example like engine, transmission and axles have already been outsourced to suppliers. Also a big topic which would not be noticed is, that many manufacturers do have a platform concept which could be compared with modular processes. They mention explicitly the vw-Group as the leader of *platform concepts* and an example how modules of products can be combined with conventional conveyor belts.

At the moment nearly every big car manufacturer makes use of such a platform concept to save costs in planning, production and at least servicing.

Different Production Systems

The authors Abderrahmane, Benyoucef and Dahane [ABD11] go into more detail about production systems and name three different ones (in figure 2.1):

- FMS - Flexible Manufacturing System
- RMS - Reconfigurable Manufacturing System
- DMS - Dedicated Manufacturing System

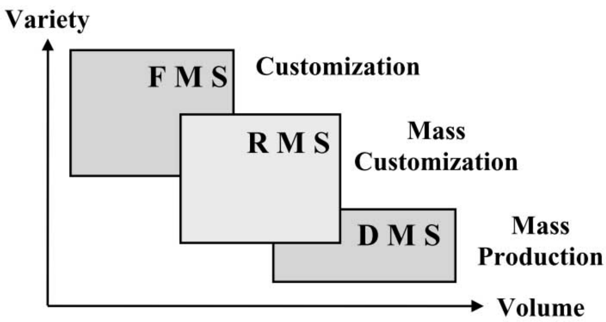


Figure 2.1.: Different production systems [ABD11]

While linear production lines can be count to DM-systems, where a certain component is produced in large numbers, the before described platform concept can already be found in the RMS sector. Reconfigurable in this case means, that the production can be adopted to the need of the market very quick.

Flexible Manufacturing Systems [ABD11] describe the kind of systems, which can be seen as real modular production lines.

Difference of Flexibility

There are several publications about how flexibility, especially in manufacturing, can be seen.

Hyun and Ahn for example [Hyu92] analogous describe three different categories: long-, middle- and short-term. In scientific language these are also known as strategic, tactical and operative. Following their statements long-term describes the flexibility of a production line to market changes, middle-term on the other hand to varying production velocity. Short-term or operative flexibility is according to them the ability to change parts of the production line itself and thereby gain a high variation of products.

Citing Granados [Gra12], flexibility in manufacturing distinguishes between routing and machine flexibility. Routing flexibility implies that assembling components can take different routes to get finished. Machine flexibility on the other hand describes the circumstance, that different machines are able to do the same working steps which results in a better capability of each work place and finally a better load balancing and production throughput.

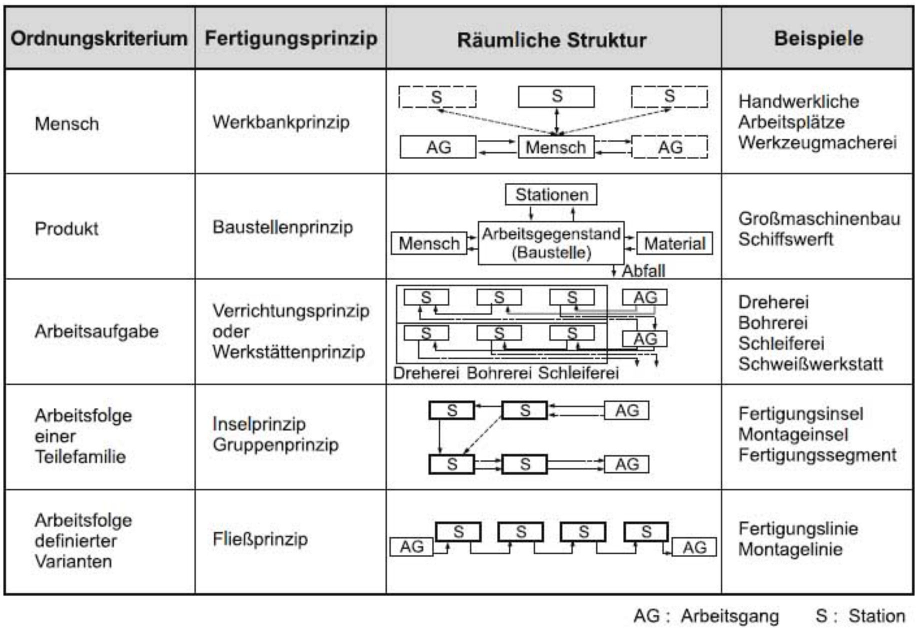
In *Manufacturing Flexibility* Swamidass [Swa00] says clearly, that one has to distinguish between flexibility of a single machine and flexibility of a whole plant. He also lists in more detail, that *plant level manufacturing flexibility* consists of hard technologies, soft technologies, design and manufacturing infrastructure.

Summing up, all these authors describe the flexibility in manufacturing as a measurement of reacting on changing requirements, either on environmental (market, customer, product) or on production level (machines).

Comparison of Manufacturing Principles

Lödding describes in his book *Verfahren der Fertigungssteuerung* [Löd16] several manufacturing principles. In figure 2.2 the three of the five named principles (*Werkbankprinzip*, *Werkstättenprinzip*, *Inselprinzip*) can be count to the family of modular production lines. *Fließprinzip* describes the typical conveyor belt and *Baustellenprinzip* more a manufacturing of low-number-products like specialized or handmade cars.

In case of this thesis, the *Inselprinzip* can be seen as what will be called *modular production* in the rest of the thesis.



Ordnungskriterium	Fertigungsprinzip	Räumliche Struktur	Beispiele
Mensch	Werkbankprinzip	S, S, S, AG, Mensch, AG	Handwerkliche Arbeitsplätze Werkzeugmacherei
Produkt	Baustellenprinzip	Stationen, Mensch, Arbeitsgegenstand (Baustelle), Material, Abfall	Großmaschinenbau Schiffswerft
Arbeitsaufgabe	Verrichtungsprinzip oder Werkstättenprinzip	S, S, S, AG, S, S, S, AG, Dreherei Bohrerei Schleiferei	Dreherei Bohrerei Schleiferei Schweißwerkstatt
Arbeitsfolge einer Teilefamilie	Inselprinzip Gruppenprinzip	S, S, AG, S, S, AG	Fertigungsinsel Montageinsel Fertigungssegment
Arbeitsfolge definierter Varianten	Fließprinzip	AG, S, S, S, S, AG	Fertigungslinie Montagelinie

AG : Arbeitsgang S : Station

Figure 2.2.: Different manufacturing principles of Lödding [Löd16]

2.1.1. Modular Production discussed

Modular production can be seen as a possibility of how components are put together to form a vehicle. Instead of a typical conveyor belt, at modular production so-called modular workstations represent the assembly places (figure 2.3).

Every workstation has its specialized skills to assemble a certain **modular** part, independent of other modular workstations in time and effort.

Compared to other modular manufacturing principles, like *Werkbankprinzip* and *Werkstättenprinzip* [Löd16], here the workstations do have varying skills and thereby represent a certain flexibility for routing items through the production line.

One idea is also a combination of *Fließprinzip* and *Inselprinzip*. In detail this means to assemble the vehicle in a conveyor belt as long as there are overlapping working steps and move then the assemblies to separated places.

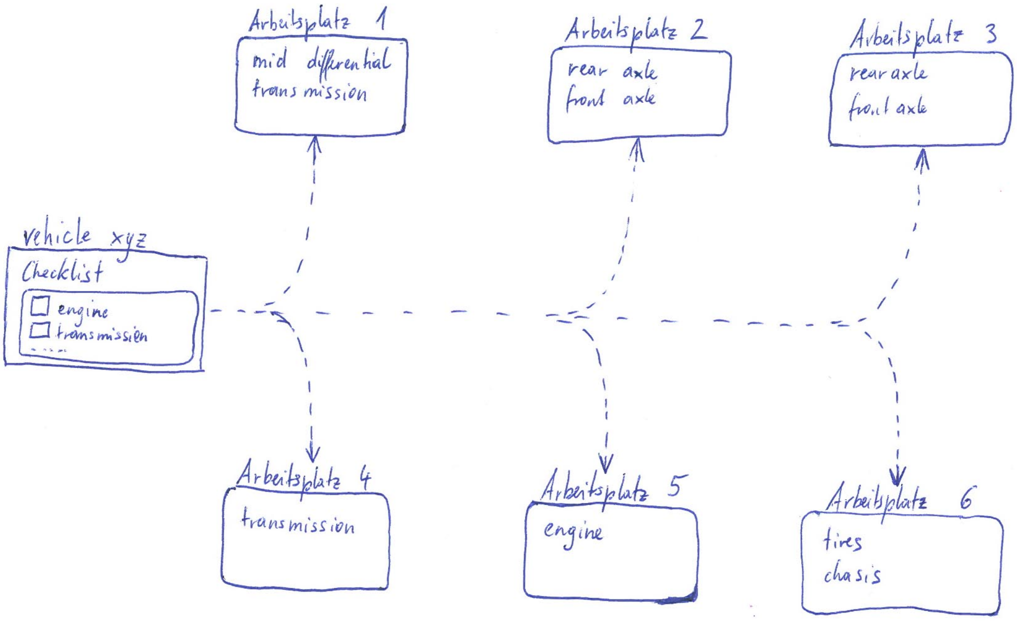


Figure 2.3.: Draft of a modular production line

2.1.2. Reasons for Modular Production Lines

As mentioned before, the basic idea of assembly lines with conveyor belts hasn't changed significantly over the last centuries. On the other hand assemblies manufactured on this lines changed though.

Automotive industry nowadays is facing different kinds of drive train from a usual internal combustion engine over hybrid solutions to electrical or hydrogen powered engines. But also the variety of variants more and more needs a higher focus.

For an assembly line it is no problem to fit up different types of cars with different variants and equipment while the body of the vehicle is still the same.

It becomes challenging in one of these example cases:

- The drive train or important parts differ
- The needed working steps don't overlap with working steps of other assemblies
- The separated working stations hold the assemblies for a different span of time.

As normal operated conveyor belts have to skip a working step in these named cases, this can cause unnecessary expenditure of time and costs.

Developing a combination of conventional and modular production lines like in figure 2.4 can help avoiding these challenging problems.

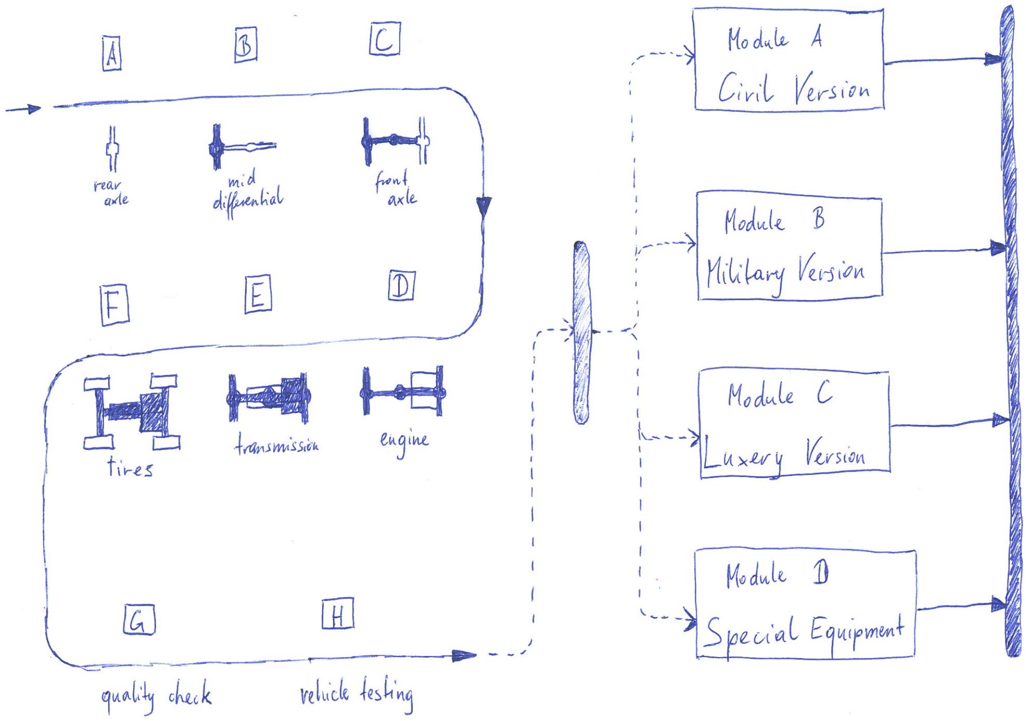


Figure 2.4.: Draft of a possible combination of modular production and conveyor belt

Nevertheless, with modular production comes also a number of challenges:

- Planning
- Transport
- Handling workstations
- Routing of components
- Logistic of assembling parts, material and workers
- Scheduling

2.1.3. Problems and Questions inventing Modular Production

The reason why modular production lines are not focused much more till now is, because the conventional conveyor belt is efficient in most cases. Especially when the variety of types is very low.

A change of the production strategy needs a high effort and also restructuring in the whole company. This starts with human resource management and ends up with handling the supply chain.

The risk of failing can be very high in case of poor planning, which could cause a horrendous impact for the producing company.

Simulations can help making decisions at which point of the assembly process a switch from conveyor belt to modular production makes sense or what benefit can be expected.

Summing up, one has to think about the risk, the overall impact and the benefit changing the production line.

2.2. Petrinets

The so-called Petrinets were founded by Carl Adam Petri in the 1960th and are widely used for different applications of modelling. Foremost it is a modelling language for concurrent systems [CGL16] which allow a step-wise representation of processes in a graphical way. Their execution semantics can be described by an exact mathematical definition

A Petrinet consists of *places (states)*, *transitions* and *arcs (edges)* connecting them. A transition (rectangle) can have one or more places (circular) as input and and also one or more as output linked. Edges between two transitions are as impossible as between two places. The edges do have weights which are decisive as to whether a transition switches or not.

Simply explained, if there are enough elements waiting in a place, namely at least the amount the transition is expecting, the transition is able to switch. Due to parallelism and enabling multiple transitions at the same time, the order can never be known in which the transition switches. So Petrinets are non-deterministic [PR08].

Petrinets can help in a simplified way to represent the process behind modular production lines. Subsequent the results of visualizing Petrinets with the framework of Franck Pommereau [Pom15] can be found in the next chapter.

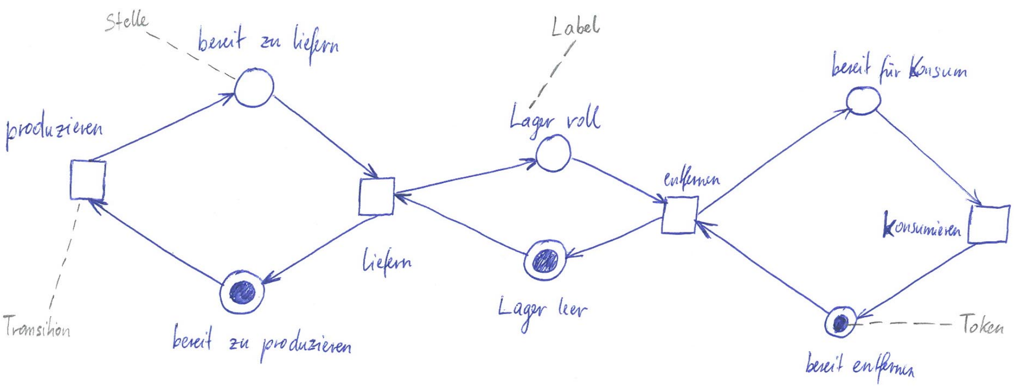


Figure 2.5.: Consumer-Producer-System with a Petrinet

Figure 2.5 represents an example of the producer-consumer-system [EET20]. A transition *produce* with a place *ready to produce* as input is waiting to switch. As soon as the place *ready to produce* contains at least one token (filled circular in the place circular), the transition switches. At the transition the working process is done and the token moves on to the next place *ready to deliver* where it is again waiting for the next transition. In a place there can be zero or a multiple number of tokens. The direction of the edges describe the potential way of the tokens.

2.3. Event Triggered Simulation

An important point in this thesis is to simulate and prove the methodology. The basis for this is the HCCM-framework [Fur+14] which is an enhancement of the known discrete event simulation.

HCCM stands for Hierarchical Control Conceptual Models. Basically the standard approach

- a condition is fulfilled
- entity is chosen from a queue
- activity will be triggered

would be sufficient for components waiting to be handled by a work place or a machine.

In *HCCM - A Control World View for Health Care Discrete Event Simulation* [Fur+14] the authors describe the use of a fourth approach to the already known three ones (event scheduling, activity scanning and process interaction) in discrete event simulation.

The simulation concept of the named authors replaces *conditional activities and queues by a hierarchical control tree* [Fur+14]. In their paper it is used for health care simulation at a hospital. The time-dependent priorities and skill-dependent working steps described in this example do have a certain similarity to the requirements of modular production.

The idea is to have so-called control units with activities as leaves, and that general rule sets replace the dispatching through conditions. The gained flexibility is exactly the approach which is needed in modular production lines, when queues and work places do have certain skills.

Furthermore HCCM-framework offers the possibility that components can wait in several queues of parallelized working stations at the same time. What has to be mentioned, that the queues in the simulation framework are not equal to physical stocks before working places or machines. These queues are more an approach of how to register an item at a new modular work place to get processed as soon as the machine is idle.

In figure 2.6 a simulation basis for both, a typical linear production line and a modular production line, can be seen. Comparing the two of them, one can see that there is no typical queuing for working stations. As mentioned before, the routing of items should be instant so that there is no need of physical queues between working stations. Whereas the control unit will hold queues to manage unfinished items.

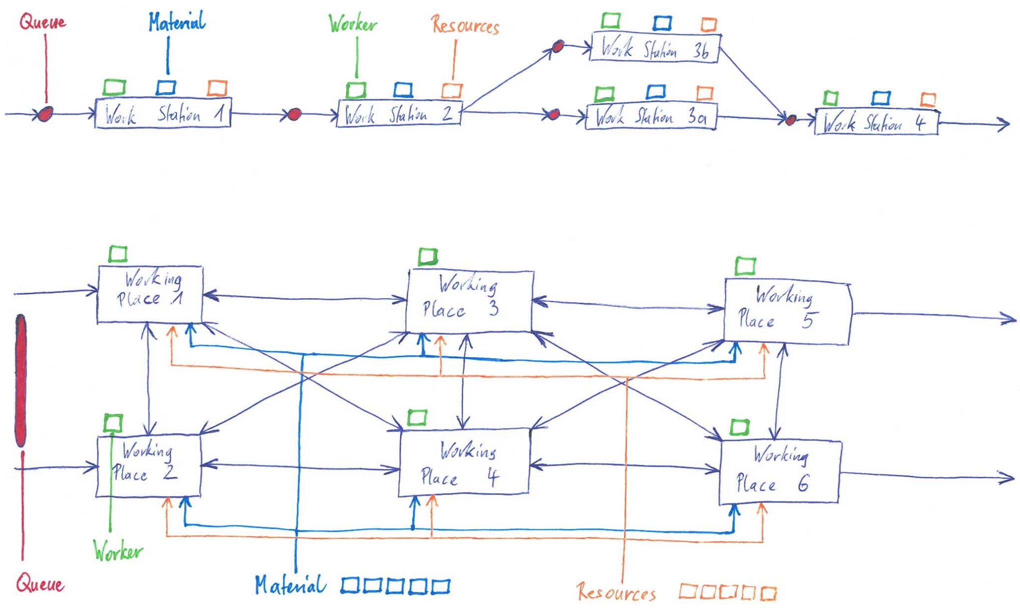


Figure 2.6.: Comparing two production systems

In addition to that in a linear production line the routing is straight forward whereas a control unit in simulation has to handle the *flexible routing* (which is done by registering items in the before mentioned queues). Therefore the control units have the knowledge about idle and working machines, waiting (registered) and processed items, and the need of material and resources for each working step in their responsible area. Reacting on these information leads to flexible routing of all kind of resources. Berndt Brehmer summarizes decisions based on previous events as *dynamic decision making*[Bre92].

Part III.

Research Concepts

3. Research Concepts

The research first deals with the methodology how the knowledge of the theory can be joined together to an approach for modular production. Furthermore it considers if Petrinets[1] can help in calculating and presenting modular production processes, and finally ends in the theoretical basis for event triggered simulation.

3.1. Methodology

The basic idea is to find a way how an *existing building instruction* could be the input of a *simulation* and is exemplary shown in figure 3.1. Therefore the steps of a building instruction are transformed into a *sequence diagram* which is the base of a *Petrinet*. Finally the simulation is fed with the data of the Petrinet.

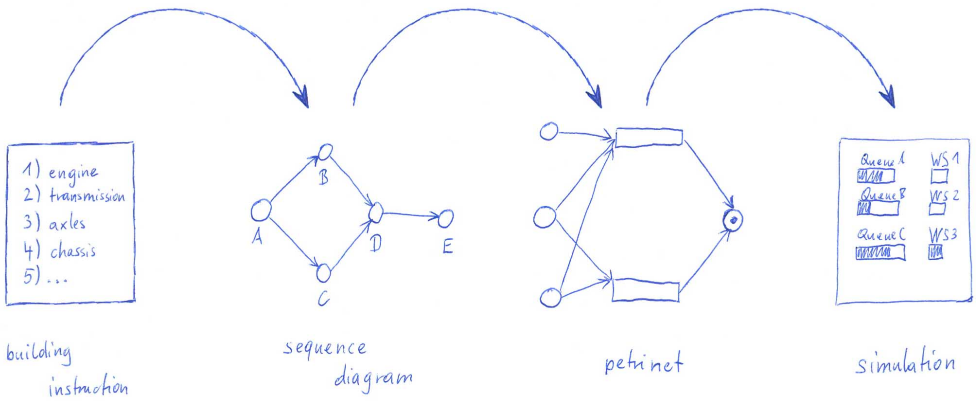


Figure 3.1.: Overview of the applied methodology

[1]models of discrete and mostly distributed systems

Task	Predecessor
A	Start
B	Start
C	Start
D	A, B, C
E	A, B, C
F	A, B, C
G	F
H	G
I	D, E, H

Table 3.1.: List of example tasks

3.1.1. Building Instruction

The building instruction is essential to describe the time schedule when, how and in which order components are assembled. Modularity in production lines as described in this thesis demands following requirements:

- Every step in the building instruction has to know its must-have-criteria, a so-called predecessor
- To make parallelization possible, at least one step has to have more than one predecessors
- Every step has the knowledge of all needed resources (tools, material, operating resources)
- Logically related steps are joined in groups (like tasks or workstations)

The next table 3.1 can be seen as an example of tasks, which in a serial way can be seen as positions at a conveyor belt. Such a task combines several steps of assembling.

3.1.2. Sequence Diagram

When having the information, which must-have-criteria has to be fulfilled to start a new task, a sequence diagram can be build up. Therefore

Task	Predecessor
A	Start
B	Start
C	Start
D	A, B, C
E	A, B, C
F	A, B, C
G	A, B, C, F
H	A, B, C, F, G
I	A, B, C, D, E, F, G, H

Table 3.2.: Calculated list of predecessors by algorithm 1

following algorithm (1) is used.

Algorithm 1: How to calculate a sequence-diagram out of working steps

```
Data: list of working steps
Result: sequence diagram of working steps
while count of steps in list >0 do
    actualStep = take the topmost step from list;
    if actualStep has predecessors then
        if the predecessor is START-flag then
            Continue;
        else
            forall predecessors of actualStep do
                Get the list of predecessors;
                Add this list to the predecessors of actualStep;
                Call this algorithm for every predecessor
                  recursively till every predecessor is a START-flag;
            end
        end
    end
end
```

The expected output is a list 3.2 with recursively gained predecessors. Compared to the simple list before, now every step knows every of its predecessors (for example **task I** consists of all previous steps).

In figure 3.2 this list is shown as graph. Here one can see a possible parallelization of steps at *A* - *B* - *C* and *D* - *E* - *F*—*G*—*H* due to the must-have-criteria of predecessors.

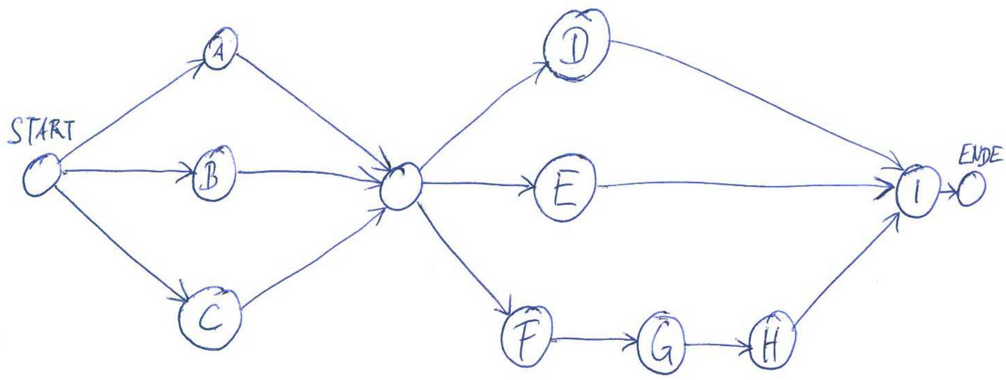


Figure 3.2.: Example of steps with predecessors

3.1.3. Petrinet Computation

As shown in figure 3.3 with a hypothetical **task B** a Petrinet is not simply a sequence diagram with a higher granularity. Through the edge weight it further consists of information a *transition* needs to switch. In the case of this thesis, the Petrinet is able to represent all our necessary information needed in a simulation. Here steps are shown as transitions and all kind of resources as states (places), whereas in a sequence diagram simply the order of steps is represented.

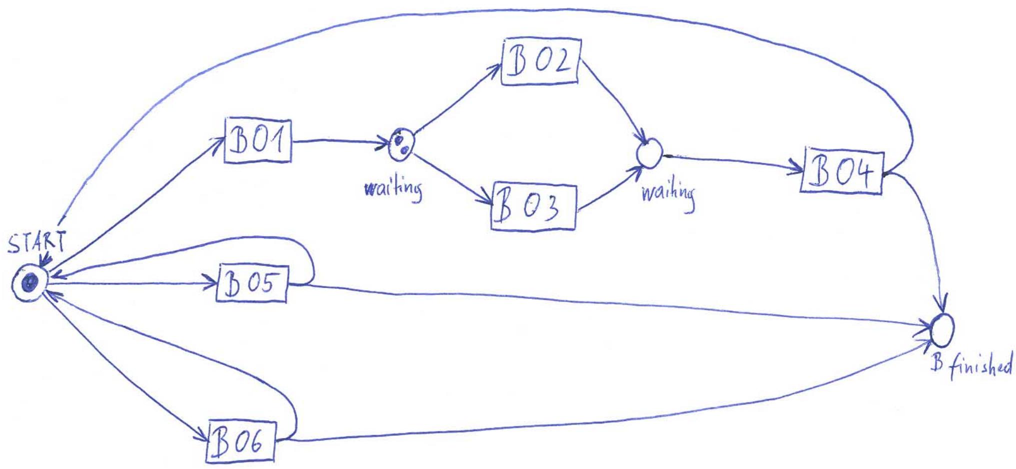


Figure 3.3.: Example Petrinet of task B

With all this information in the Petrinet, one can see at the first glance which states (resources, previous steps, and so on) are necessary to switch a transition, which in this case means to fulfill a working step.

3.1.4. Simulation Model

As a Petrinet is more a mathematical modeling language than a graphical representation, the simulation is so to say the execution of it. So drawing a Petrinet is always just a snapshot of a comprehensive sequence of activities.

The simulation extends the Petrinet with the factor of time. An information that a transition needs several states to switch causes further requirements in simulation.

An extension to theoretical models in simulation is, that there are physical workplaces which execute the working steps. To stay as dynamic as possible workplaces should have skills, whereas a skill represents a task or a set of working steps the working place can process.

Identification of Objectives

According to Robinson [Rob08], objectives can be distinguished by modeling objectives and general objectives. Last ones describe the simulation model as a tool [Fur+14].

In this case this objectives could be:

- Modeling:
  - Item gets processed in the way the sequence diagram shows
  - Every item gets processed by the number of machines equal to the number of needed skills
  - Machines/workplaces can have more than one skill
  - No large deviation between average lead time and minimum lead time
- General:
  - Run-time of simulating a week in less than one hour
  - Graphical visualization of queuing

Outputs

The output of a simulation set can be summarized in numbers and should prove or refute the modeling objectives. These measurands could be:

- Minimum lead time
- Maximum lead time
- Average lead time
- Average waiting time of an item in queues
- Distribution in percent of waiting and processing time

Input Factors

The input values consists of lists of data containing following information:

- Ordered items (complete production list of day/week)
- List of working steps for each order-item
- List of skills (all needed skills)
- List of work places (machines)
- Resources

Model Structure

The simulation model consists of following structure-items:

- Activities and events
- Items with checklists (client)
- Workplaces with skills (server)
- Queues
- Request Activities and Events Lists (RAEL)

The figure 3.4 shows the level of detail of the model.

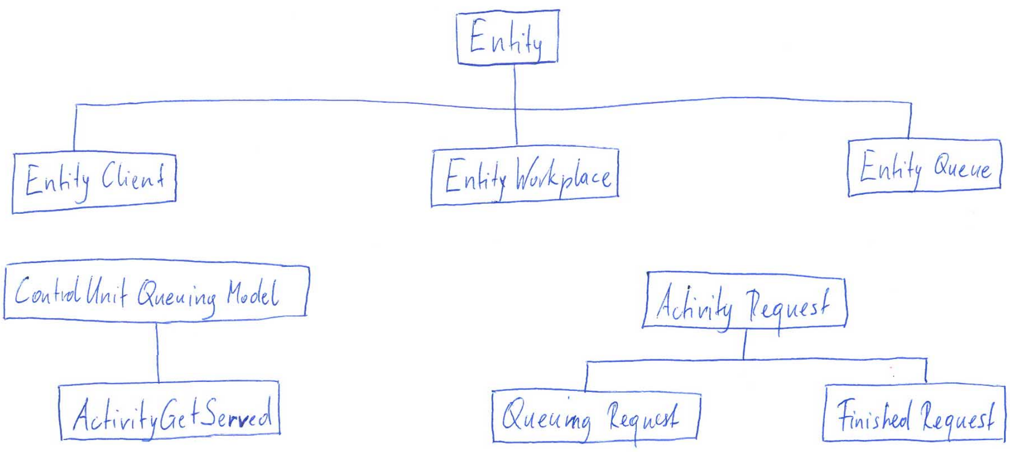


Figure 3.4.: Structure of Simulation Model

Remarkable about this structure is, that workplaces and queues are entities with skills. The *ControlUnitQueuingModel* represents the control unit which is responsible for steering processing and queuing management.

By splitting the entities to the ones shown in figure 3.4, following responsibility also can be defined:

- **EntityClient**
  A component knows which steps it has fulfilled and which are still needed
- **EntityWorkplace**
  A workplace (or machine) knows its own skills and processes components waiting for these skills
- **EntityQueue**
  A queue holds entities and handles input and output calls through FIFO (first-in-first-out) principle
- **ControlUnitQueuingModel**
  A central control unit puts queued components to idle machines and incomplete components to queues

Individual Model Behavior

In figure 3.5 one can see the process of an item in simulation. The state diagram starts with an already ordered item arriving in production line to get served.

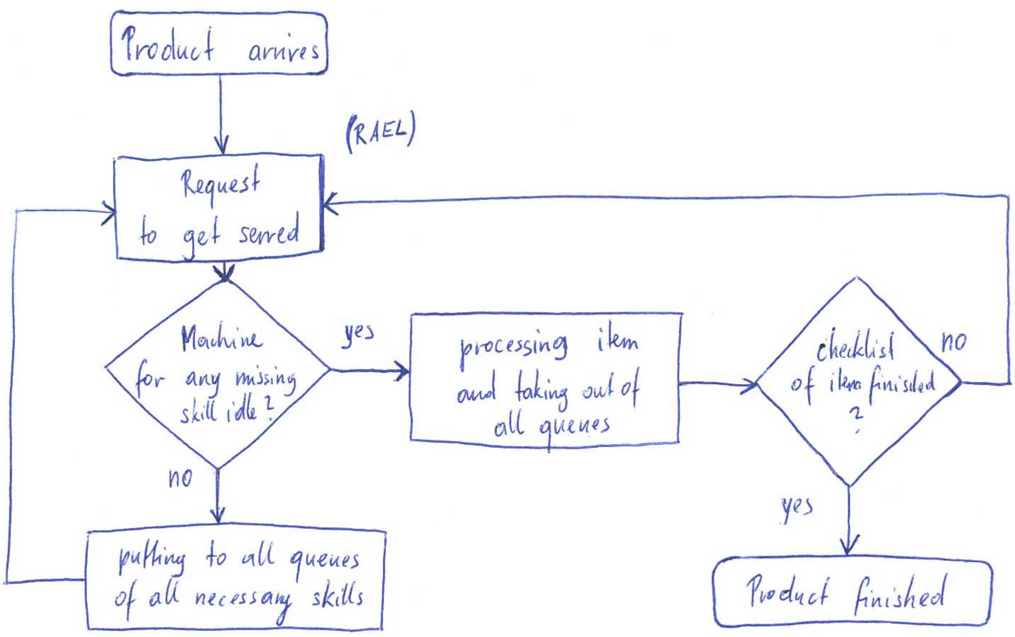


Figure 3.5.: Process of an item in simulation

As mentioned before, not every workplace but every skill has its own stock. That means, that the relation between workplace and queue is an m:n relation. The control unit will do a lookup in 1:n queues (namely for each skill) for each workplace, and every single queue can hold entities for 1:n different workplaces.

The algorithm 2 describes, how requests are put in the queues in the *ControlUnitQueuingModel*. Important part in here is, that the next needed skills of the item are identified and following the item is put to every queue of these skills.

Algorithm 2: How to put requests to the right queue

```
List getServedList = get all activities from RAEL-list which are
  waiting to get served;
foreach item in getServedList do
    List requestingSkills = get all next possible requesting skills of
      item;
    List fittingQueues = get all queues for all items of
      requestingSkills;
    foreach queue in fittingQueues do
        put item to queue;
    end
end
```

Furthermore in the *ControlUnitQueuingModel* a waiting item is assigned to the next idle workplace (machine). As the algorithm 3 shows, the intersection of skills of idle workplaces and waiting items for this skills has to be found.

Algorithm 3: How to handle clients in right queues

```
foreach idleWorkplace in workplaces do
    List skillsOfWorkplace = get all skills of idleWorkplace;
    List queuesOfSkills;
    ForEachskill of idleWorkplace  Queue queue = get queue for
      this skill out of all queues;
    if item count of queue >0 then
        add queue to queuesOfSkills;
    end
    if item count of queuesOfSkills = 0 then
        Continue;
    else
        Queue fullestQueue = get queue out of queuesOfSkills
          with largest count of holded items;
        EntityClient client = get first item of fullestQueue;
        ActivityGetServed newService = create new Activity with
          (idleWorkplace, client, skillToProcess);
        start event of newService with simulation engine;
    end
end
```

System Behavior

Basically after initialization, for each item in the order list an activity *get-served* is created after every certain time span (random). This activity is registered in the RAEL (Requested Activities and Events Lists) list.

In the custom rule set of the control unit, the logic of queuing and processing is done. Therefore every time an event is triggered, activities from the RAEL list are put to the queues of the skills they need to be processed. The decision to which queues the item is put, is based on the fact, that *all previous steps are done* and *all necessary resources are available*. For every idle work place (machine) the control unit looks up waiting items in the associated queues.

At the workplace the items are processed for a certain time. After finishing the working steps, again an activity is created for the item whether of type *get-served* or *finished*.

3.2. Research

3.2.1. Petrinet Experimental Setup

Starting with a trivial example a building instruction of a LEGO tractor is used. Usually not only in LEGO building instructions but also in automotive industry parts are assembled following a linear serial procedure which is set at the beginning. That has the consequence that no separate modules can be found in a usual building process and a process of assembling cannot be changed later.

To locate modules and figure out case dependent assembling processes, a part of the transmission of the LEGO tractor is build up in three different ways.

A simplification of any modular production and different paths of assembling can be shown by following experiment in figure 3.6.

The **transitions** are drawn as rectangles with a description whereas the **states** are drawn as circles with either label start or end or a picture describing the waiting parts.

The three variants with their described transitions are:

- Assembly variant A
  - A1 - assembly of **spur gear** and **axle**
  - A2 - attaching **perforated sheet** and **small gearwheel**

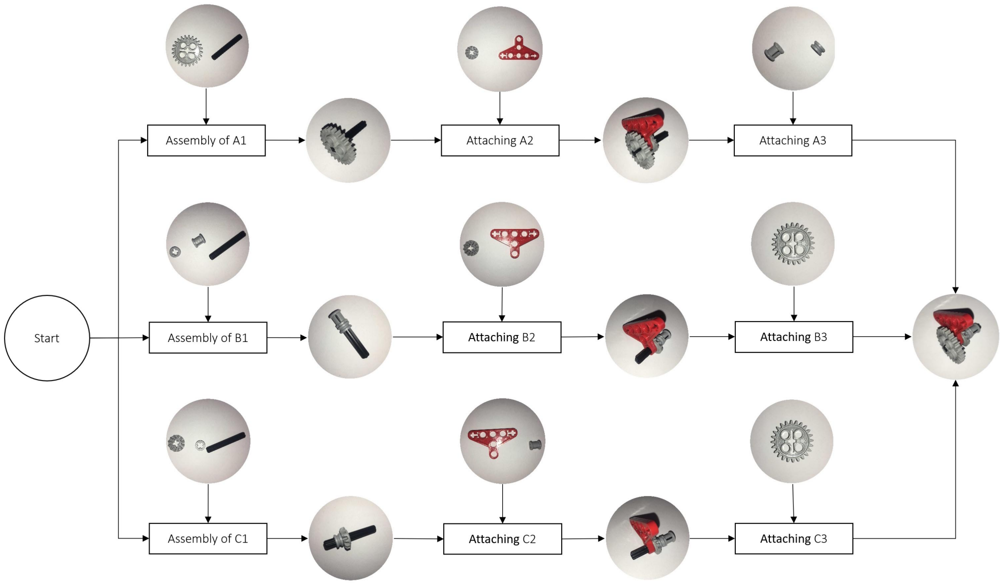


Figure 3.6.: Different variants of assembling a Lego transmission

  - A3 - attaching **big stopper** and **small stopper**

- Assembly variant B
  - B1 - assembly of **axle**, **big stopper** and **small stopper**
  - B2 - attaching **perforated sheet** and **small gearwheel**
  - B3 - attaching **spur gear**

- Assembly variant C
  - C1 - assembly of **axle**, **small stopper** and **small gearwheel**
  - C2 - attaching **perforated sheet** and **big stopper**
  - C3 - attaching **spur gear** and **axle**

The result of this experiment shows a parallel building instruction with possible ways to part into modules. Furthermore some conclusions can be read out of analyzing these paths and lead to the following figures which describe advantages of using modular production instead of a conventional one.

Following some examples of dynamic best paths at run-time will be shown:

- Critical component
- Just-in-time supplier
- New routing at run-time

In figure 3.7 the **big gearwheel** is a critical component which can be assembled as available. If it is in stock, the next transmission can be assembled in variant A. If not, variant B or C can start assembling other components till the critical component is in stock again.

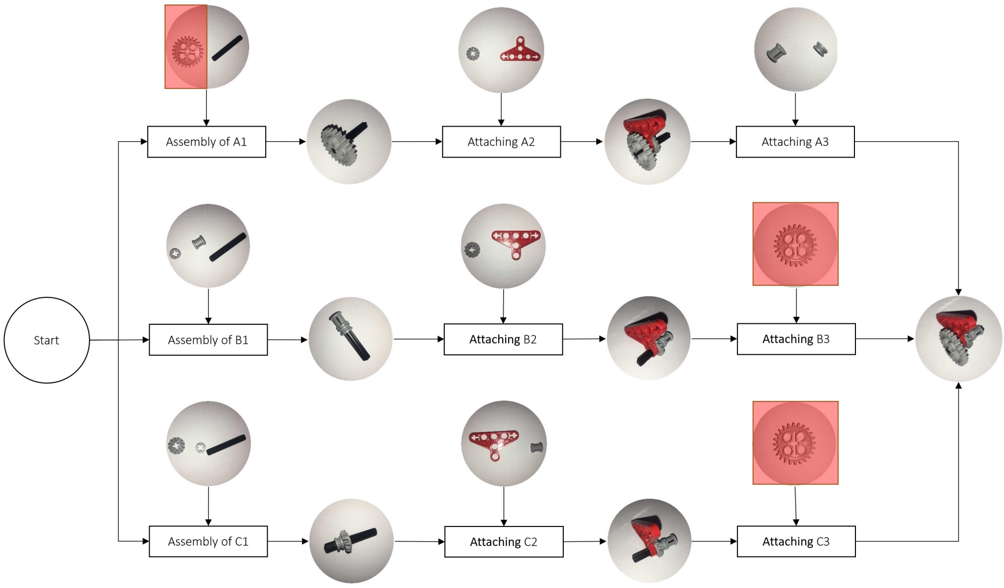


Figure 3.7.: Best path critical component

In figure 3.8 the components **axle**, **big stopper** and **small stopper** are delivered from the same supplier. To improve the stock workload and optimize just-in-time delivery variant B can be focused at assembling.

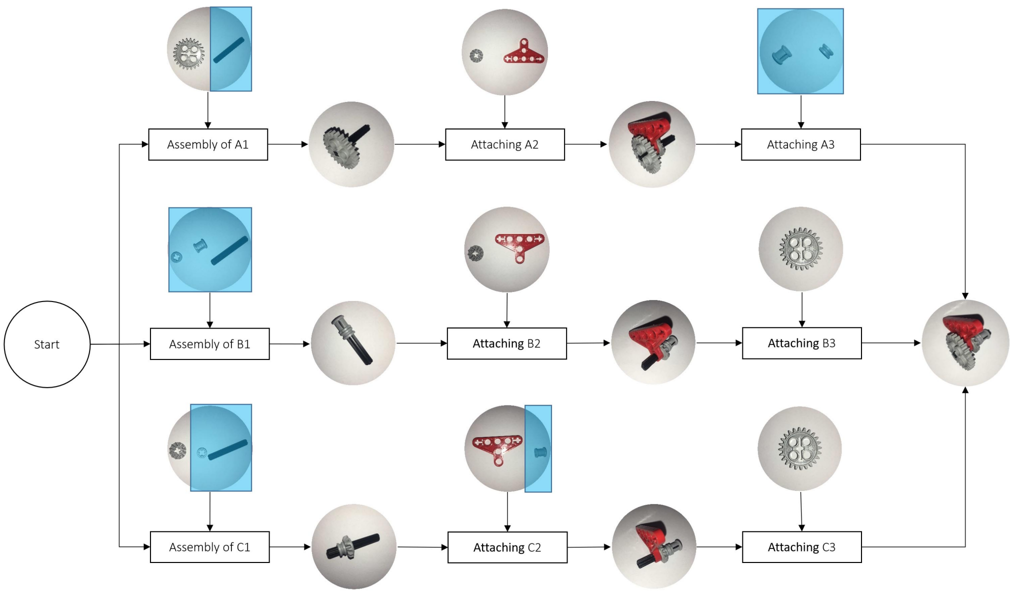


Figure 3.8.: Best path just-in-time supplier

In figure 3.9 a case is shown where missing resources, tool failures or an employee loss cause problems in finishing the component. A dynamic routing at run-time would suggest to go for variant B or variant C to not stop the whole production.

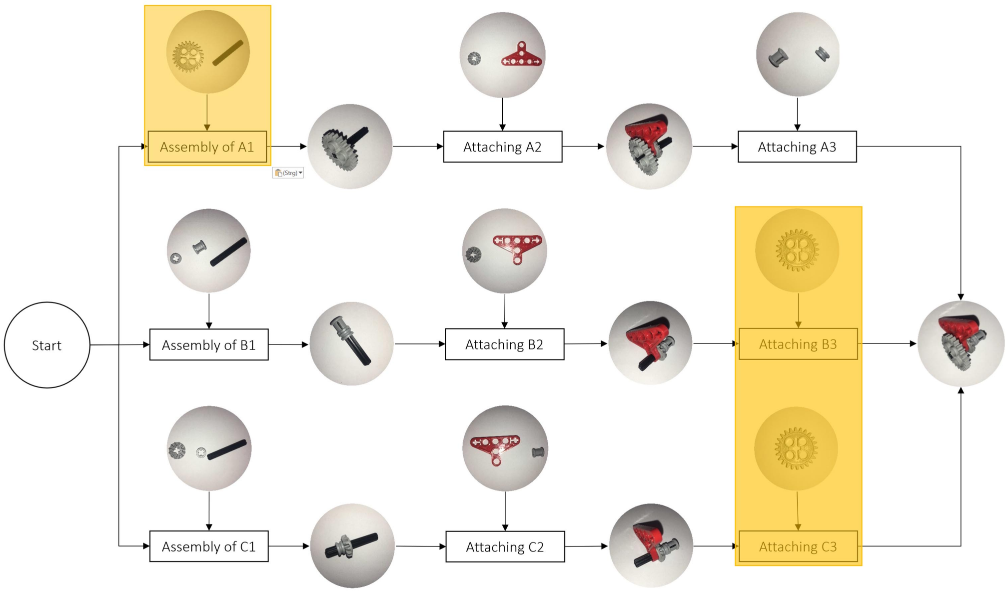


Figure 3.9.: Best path with new routing

One can imagine that more agility to react to a supplier by modular production leads to more overhead in managing the production logistic itself.

An advantage of modularizing the production process could also be the detection of similar or equal working steps. In figure 3.10 the green areas represent such a case. In modular production lines this will lead into a consolidation of the working steps **Attaching C** and **Attaching E**.

The blue area shows a possibility of outsourcing some working steps to different production location or even to a supplier. Such a decision could be caused at run-time by workload balance of modular working stations or by cost and time reasons.

The described decision between in-house production and outsourcing is shown in a detailed way in figure 3.11. For example a production manager could do a best-path-calculation due to following reasons:

- Is there a critical component which could be outsourced?
- Could this critical component cause a blocking state in the production line?

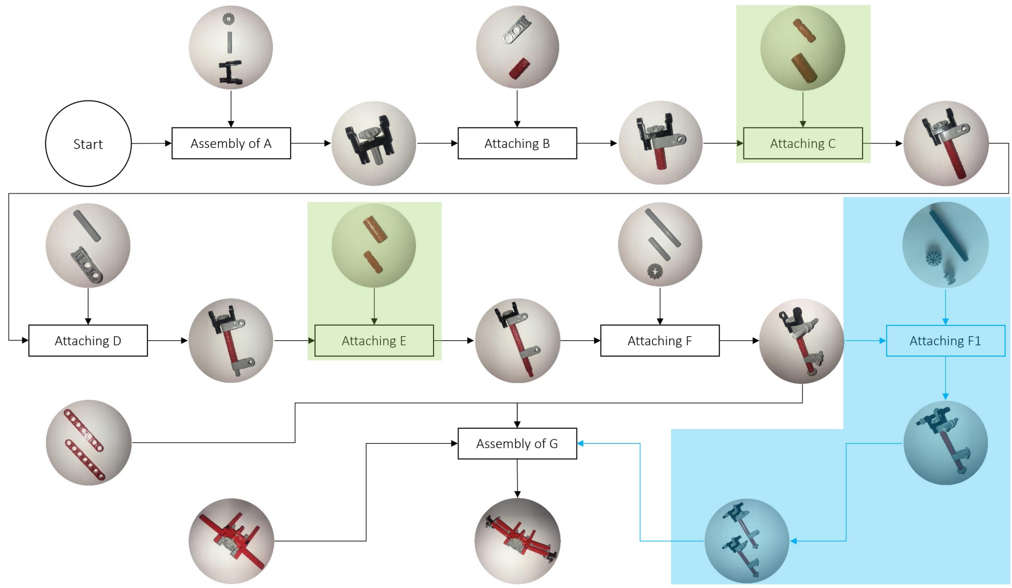


Figure 3.10.: Assembly of a differential gear

- How about varietal purity? May an outsource be more efficient due to special needs of workers qualification, material or tools/machines?

Summing up, a Petrinet could help to localize critical components (states/places as circular pictures) and can be a decision support for outsourcing or rearranging the assembly of these components.

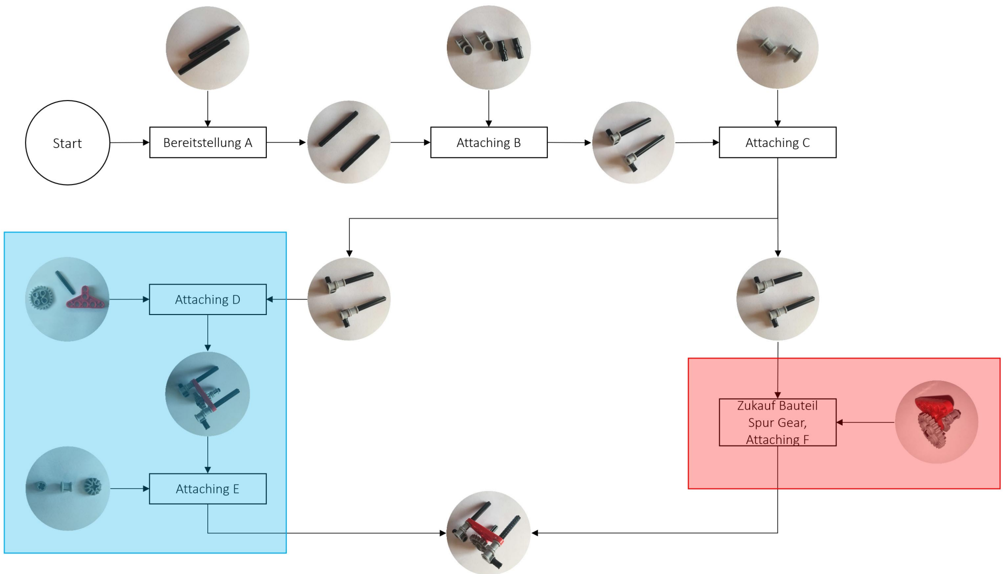


Figure 3.11.: Assembly directly on axles

3.2.2. Petrinet Real Data of Magna Steyr.

The described data of the Lego tractor is a base for handling the original data of Magna Steyr. This data of Magna consists of different inputs:

- Working steps
- Workstations
- Workers
- Material
- Working tools
- Operating resources

The goal is now to use the prior knowledge to handle this data. Subsequent the Petrinet should be the basis for a simulation of the modular production line. Before refactoring the building instruction the components for building a Petrinet have to be explored.

In this case states and transitions can be separated as follows (also described in German due to original data in EXCEL sheet):

- States
  - Activity status (Vorgangsstatus)
    predecessor, finishing a transition, component
  - Working tool (Werkzeug)
  - Material and prefabricated part (Material/Bauteil)
  - Operating resources and supplies (Betriebsmittel/Betriebsstoffe)
  - Worker and qualification (Mitarbeiter/Mitarbeiterqualifikation)
- Transitions
  - Process of one or more working steps
  - Include mandatory criteria (states that have been passed through at this point of time)

Identifying the columns in the EXCEL sheet as states and transitions leads us to the first Petrinet shown in figure 3.12.

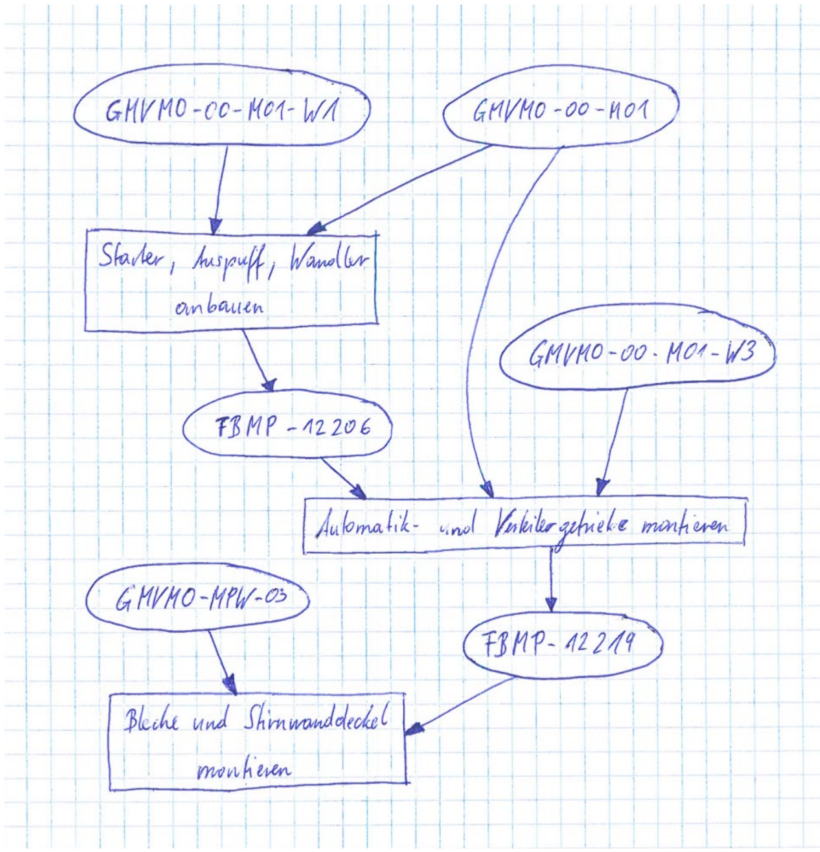


Figure 3.12.: First draft of original Magna data

3.2.3. Basics of Data Acquisition and Visualization Tool

To get a bigger picture and a more detailed view of the original data at the same time, the Petrinet is drawn by a self-written Python-script (listed in appendix C.1.1) which handles the load of data automated.

For that a Python-script has been developed which uses the packages Pandas[2] and Snakes [Pom] to handle the data and draw the Petrinet.

The output of the script (figure 3.13) gives a first impression how a Petrinet could look like when calculated and drawn automatically. It also already shows that some *states* are necessary for more than one *transition*.

[2]Python Pandas `https://pandas.pydata.org`

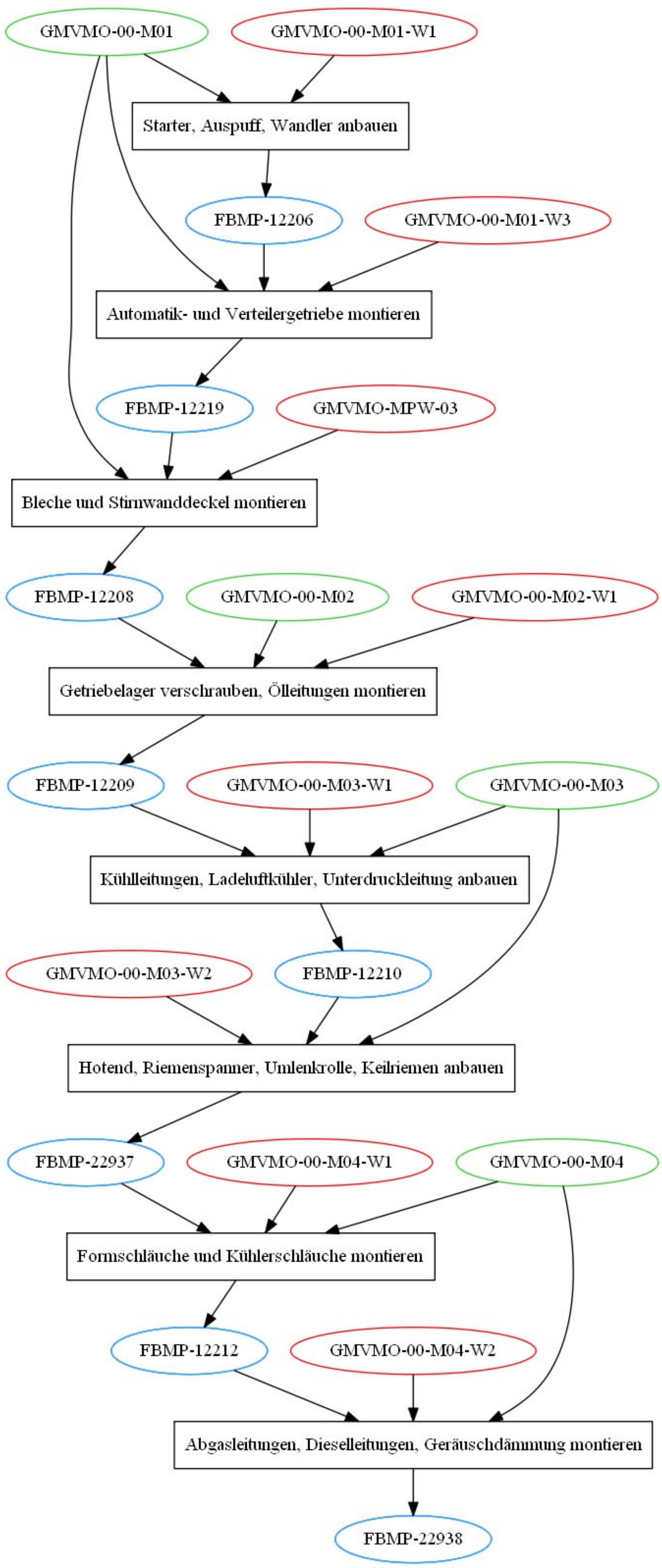


Figure 3.13.: Petrinet of Magna data drawn by Python script

3.3. Challenging Problems

As the tendency of the draft shows and the EXCEL sheet[3] may let expect, a building instruction without possible previous working steps leads into a straight-forward Petrinet without meaning.

Looking at the Petrinet-draft, output of the Python-script and the EXCEL data, there are several problems to work with this dataset:

- Anonymized data
  The data is anonymized and full of aliases.
- Illegible data
  One can not read and follow the meaning of the Petrinet as workers, material and components have nearly the same naming and cannot be distinguished from one another at a glance.
- Serially running events
  All the working steps follow a serial process. It is not clear if the previous working step is really necessary or if another way of assembly is possible.
- Assembly due to history
  The assembly process is already fully planned without a possibility to get information of a necessary previous task for modularization.

[3] confidential data of Magna Steyr, not publishable

3.4. Concepts

As described in 3.3 the data is not suitable to work with. To get an idea of how modular production can be planned, simulated and finally evaluated, a pseudo dataset has to be created.

Requirements for such a dataset can be listed as following: (whereby the italic written parts are not available yet, but the structure of the original data is still used):

- Working step
- Worker
- Material
- Working tools
- Operating resources
- *Workstation (physical place)*
- *Skills of a workstation*
- *Previous working steps (must have criteria)*

As the italic written parts are not available yet in the original Magna dataset, this information must be generated at development time before the building instructions are created.

For simulating a modular production line the data basis **previous working steps** is essential. This determines a **must-have-criteria** for each working step and makes modularity possible.

The pseudo data EXCEL sheet has also to fulfill following requirements:

- Usability
  - Building instruction of a real world object
  - Easy and imaginable parts so that a holistic view is clear
  - Clear and readable description
  - Parallelizable working steps
  - Expandable at run-time to make differences visible in simulation
- Special criteria
  - Different Types of vehicles to represent original data
  - Little differences in working steps should represent one module
  - Automatism for n:m relations
  - Working steps should contain at least one quality check
  - At least one working step which does not appear in other types

  - An order list has to be implemented with different types to represent the *Drehscheibe* at Magna's data and to influence the simulation later.

To create easy understandable, imaginable and readable data, an engine assembly instruction of an oldtimer **Steyr 290** tractor is used.

Paying attention to all these requirements, the pseudo data consists of following columns:

- Production Steps
  - Brand
  - Type
  - Working Step ID
  - Working Step Description
  - Working Step Kind (work or check)
  - Working Station (Skills)
  - Material
  - Operating Resources
  - Working Tools
  - Previous Working Steps
- Working Places
  - Physical Working Place
  - Working Station (Skills)
- Working Stations
  - Station
  - Description
- Order List
  - Order ID
  - Type

Dealing with the designed draft data some special insights can be determined. First of all some points like must-have or previous working steps have to be considered at planning time. Secondly additional components like material, operating resources and working tools have to be unified to make sense in the Petrinet. Next the modules should be chosen meaningful for production flexibility. Finally if the data fits, a Petrinet can help simulating a modular production to show bottlenecks and improvable sections.

The whole list of working steps in the pseudo data can be found in the appendix at B.1 and the working stations at B.2.

3.4.1. Stock per Skill

There are several approaches how to handle waiting items and idle work places:

- Every work place has its own stock set up like a queue where items wait to get handled.
- All items wait in one queue and idle work places do their lookup there.
- Every skill has its own stock.

The first one is basically a good approach, but not adequate if a work place has more than one skill. The problem here is, if a machine at a work place fails, items are waiting in the stock and cannot be handled by other machines. In this case a higher intelligence has to check all stocks of all work places the whole time which leads to an additional effort.

The second one is only properly for a small number of work places. If the count of them grows, blocking states and race conditions can occur when they try to fetch an item from the stock.

The third approach is the most suitable for this challenging problem. In several experiments at development time it turned out, that the impact of one faulty work place on the whole production line is very low, if every skill has its own stock. The stock-per-skill approach is shown in detail in figure 3.14.

Items do have an internal checklist consisting of skills they still need to get processed. *Item 1* still needs the skills *A*, *B* and *C*. So it is added to **all three stocks of needed skills**.

If a work places switches his state to *idle*, he does a lookup for a new item in the *stocks of his skills - A and C*. When fetching for example *Item 1* it is **removed of stocks** so that it could not be fetched a second time by a different work place.

An example for one item running through this process can be found in figure 3.15. The different colored states (*cyan, magenta, yellow*) show the approach, that an item is put to *every needed* skill.

The advantages of this stock-per-skill approach are:

- **Machine failure**
  A machine failure is not a problem at all, because just one item gets stuck. As there is no stock for the work place itself, no redistribution is necessary. And also if once a work place is closed completely, no stock has to be dispersed additionally or removed from the control-unit-logic.

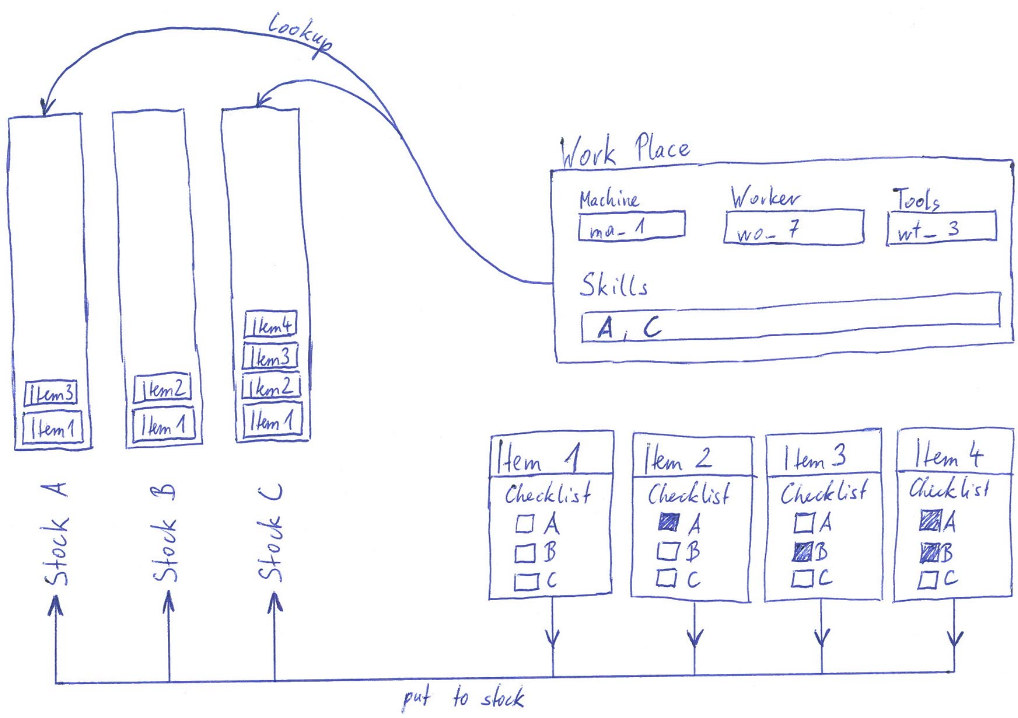


Figure 3.14.: Overview of the stock-per-skill approach

- **Load balancing**
  If there are many work places with the same skill, the work places fetch the items of the largest stock (with their corresponding skills). So for same skills it could not happen that one work place is idle and the other one has a full stock due to delays.
- **Overall view**
  A production manager can see at glance if there is a need to add or remove work places for a certain skill.
- **Management**
  No additional management overhead to check items in stocks is necessary if a work places switches to idle.
- **Scalable**
  This approach is scalable as the number of stocks grow with the number of skills and not with the number of work places.

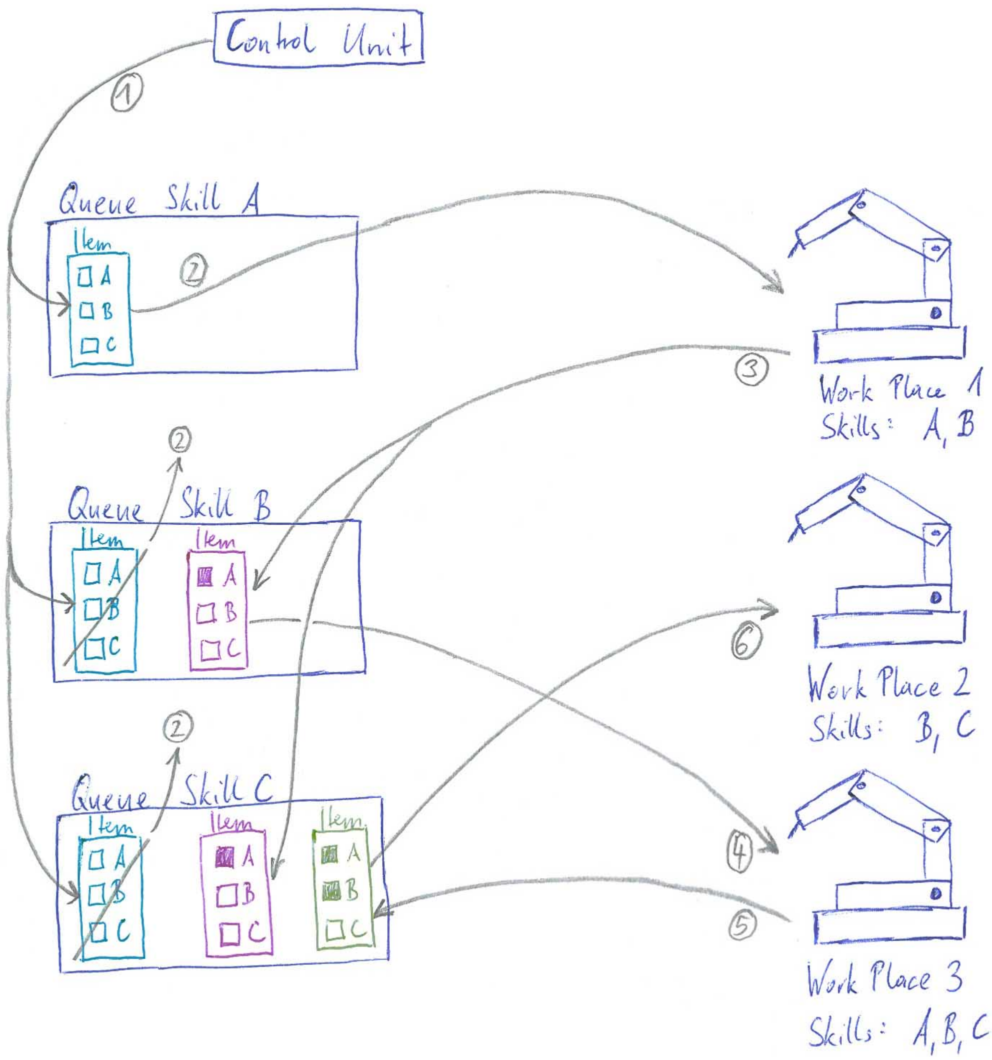


Figure 3.15.: Stock-per-skill example with one item

Part IV.

Development of the Simulation Environment

4. Development of the Simulation Environment

Besides the practical work of researching and generating ideas, this chapter deals with the implementation of findings out of theory and resulting experiments.

The main intention is to create a software which is able to fulfill following points:

- Dynamic data acquisition of existing building instructions
- Visualization of the gained information as a Petrinet
- Simulation of a complete production plan for a specific set of modules, production steps and work stations

The purpose is on the one hand to visualize a modularized building process and on the other hand simulate it to gain more information and find a best path solution.

4.1. Structure Creation

First consideration is how to create a structure which will meet all the requirements for all meta information about already existing information and logical conclusions. The challenge hereby is to keep the managing overhead very low. Otherwise the through modular production gained benefits will be dissolved at the same time by the additional effort.

Before explaining the solution approach, it is necessary to explain the most important elements (discussed in draft A.3):

- **Production Step** - *Arbeitsschritt*
  A production step describes the smallest working unit when assembling components and consists of an ID and a description. It belongs to a single work station, knows for which type of vehicle it is mandatory, which previous production steps have to be fulfilled before starting and which materials, operating resources and working tools are needed.

- **Work Station** - *Arbeitsstation*
  A work station describes both, a set of production steps to finish an assembling section and a skill which a physical work space or machine is able to handle.
- **Physical Work Space** - *Physischer Arbeitsplatz*
  The physical work space can be seen as a modular working place. It has certain skills (work stations) which it can handle. So one can say the combination of a modular working place with a certain worker and a certain machine represents a qualification to fulfill certain production steps. This physical work space is exchangeable and regardless of location.
- **Type** - *Bautyp*
  The type of vehicle is required to filter the necessary production steps. In the order list the ordered item is just described as a type of vehicle.
- **Previous Production Steps** - *Vorausgehende Arbeitsschritte*
  This is the most important property to create a modularizable structure. It describes the necessary production steps up to this point in time before starting with the actual step. Depending on this also a parallel structure can be possible.
- **Order List** - *Auftragsliste*
  The order list is simply the list of production (for customers, market, stock). It consists of the order number or a date and the ordered type of vehicle.
- **Checklist** - *Arbeitsschrittkontrollliste*
  The checklist is a virtual created object for each and every ordered item in the order list. It consists of work stations to be done and can not be found in the data list. It is a kind of ToDo-list representing the completion status of the vehicle. Depending on the checked items a vehicle is registered at the queues of the missing production steps.
- **Additional Meta Data**
  A production step also consists of certain materials, operating resources and working tools.

The UML class diagram[1] in figure 4.1 represents only the necessary classes in an overview for handling the data input described before.

Due to this structure, a dynamic changeable data input is guaranteed.

[1]drawn with http://app.creately.com

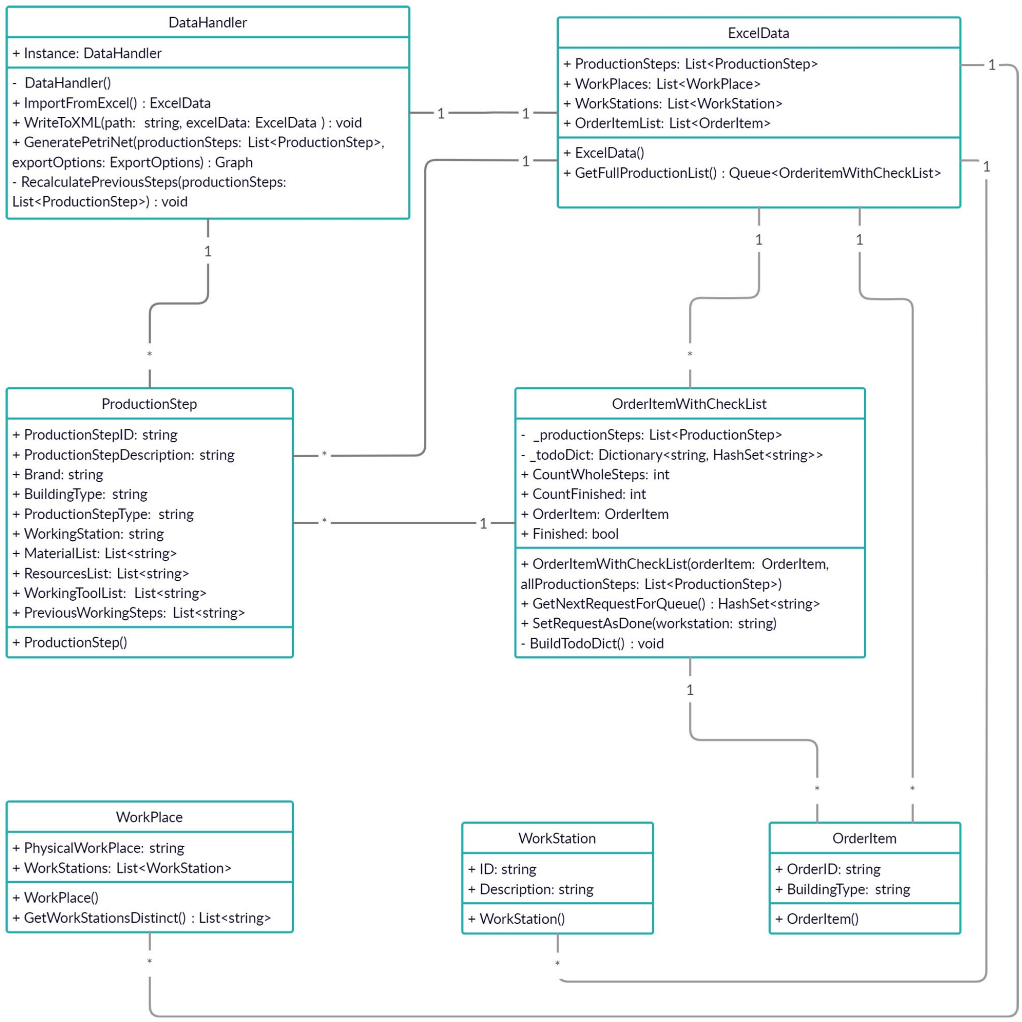


Figure 4.1.: UML class diagram for handling the data input

4.2. Data Acquisition and Preparation

The main entry point of the data acquisition part is represented by the **DataHandler**. This class is a singleton to ensure only one central responsibility for data input from EXCEL on the one hand and data output as XML on the other hand. Furthermore the calculation of the previous steps and the generation of the Petrinet graph is done here.

As described before the data is represented by an EXCEL list. To gain all the data from the worksheets, an automated serializer [Micb] is used. To write all the data objects in a reusable style the XML serializer of .NET is used.

The source code how the import and export are executed can be found in the code section at C.2.1 whereas the examples of the serializeable classes for EXCEL and XML can be found at C.2.2. This is a short overview of how automated import and export can be done with serialization attributes as prefix of every class-property. The XML output roughly looks like the shortened file which can be found at C.3.1.

The result of import is an **ExcelData** object which consists of lists of **ProductionSteps**, **WorkPlaces**, **WorkStations** and **OrderItems**. The ExcelData itself can return a full production list represented by a Queue. Every element of this Queue is a **OrderItemWithCheckList** containing all ProductionSteps for this ordered item. The hierarchical structure of all ProductionSteps of a certain type can be seen in figure 4.2.

Besides import and export the DataHandler is also responsible for calculating previous steps of every production step and finally generate a graph out of these information. In the code snippet at C.2.3 one can see how the for the visualization needed graph is generated. It consists of nodes like ProductionStep, Material, Resource and WorkingTool and the linking between them represented by edges.

The challenge hereby is to get a correct graph for every kind of building type, including the correct equipment and correctly display the relations between states and transitions.

By having this graph a building instruction is modular, as every ProductionStep has knowledge about its must-have predecessors.

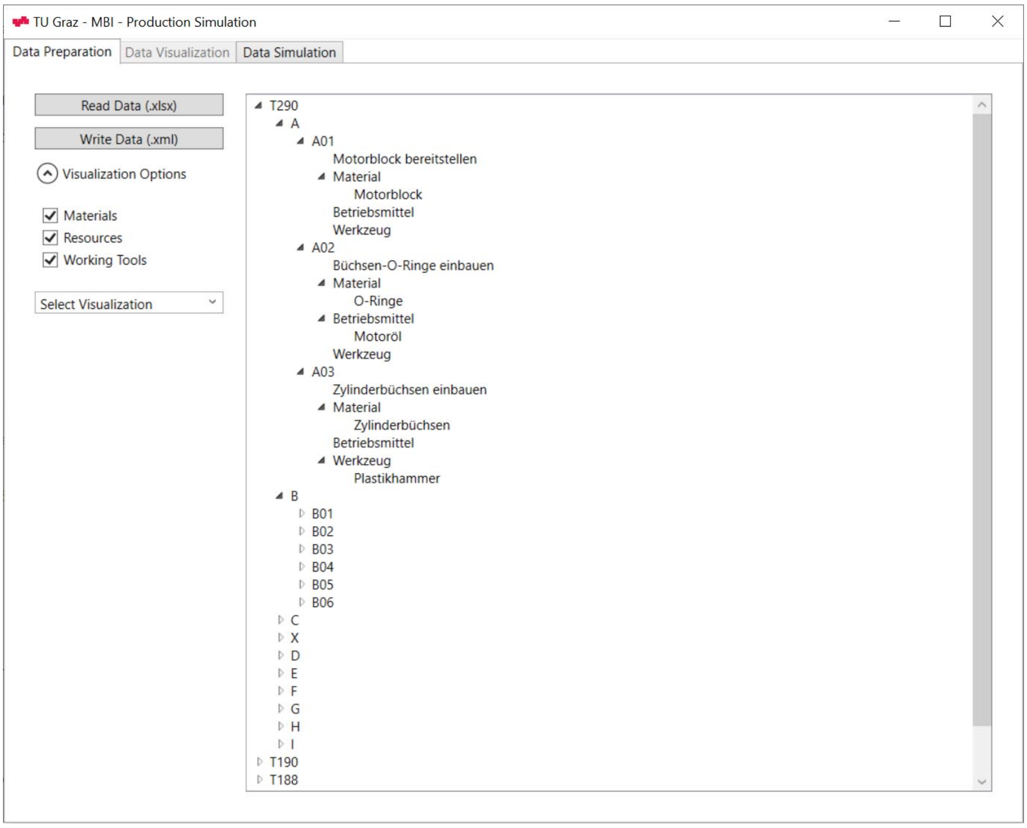


Figure 4.2.: Screenshot of the data preparation part of the software

4.3. Visualization

The visualization part is a way to present the calculated graph in a readable way to the user for each building type. As the hierarchical way (before described tree-view or XML-output) is simply confusing if the dynamic data is changed, the as Petrinet drawn graph gives an overview of modules, the needs of every production step (resources, material, working tools) and the connections between each other.

Furthermore the reason for the Petrinet graph is to put this information into the simulation later.

The figure 4.3 shows a Petrinet with standard options. The Microsoft MSAGL Graph Viewer [Mica] is used for drawing and offers several useful functions like moving states and transitions with automatic redrawing, zooming, saving and so on.

The circular objects consisting of a character and a number represent the **states** in the Petrinet which describe a finished transition. The **transition** itself is represented by a rectangular surrounded description of the production step.

The transition also can require different input-states like

- Materials (green states),
- Working Tools (blue states) and
- Operating Resources (orange states).

The diamond states describe checking points which represent a special production step every component has to pass at this point.

The most important point at visualizing the graph is that one can change the settings so that for example just production states and transitions are drawn, a so-called basic Petrinet. As the code snippet at C.2.4 shows, the visualization part has only to call the graph generation in the DataHandler. The visualization itself is done by the Microsoft Graph-Viewer [Mica].

By selecting or deselecting **Materials**, **Operating Resources** or **Working Tools** one can get a detailed view from each point of view. The figure 4.4 illustrates two examples with just states of firstly materials and secondly working tools.

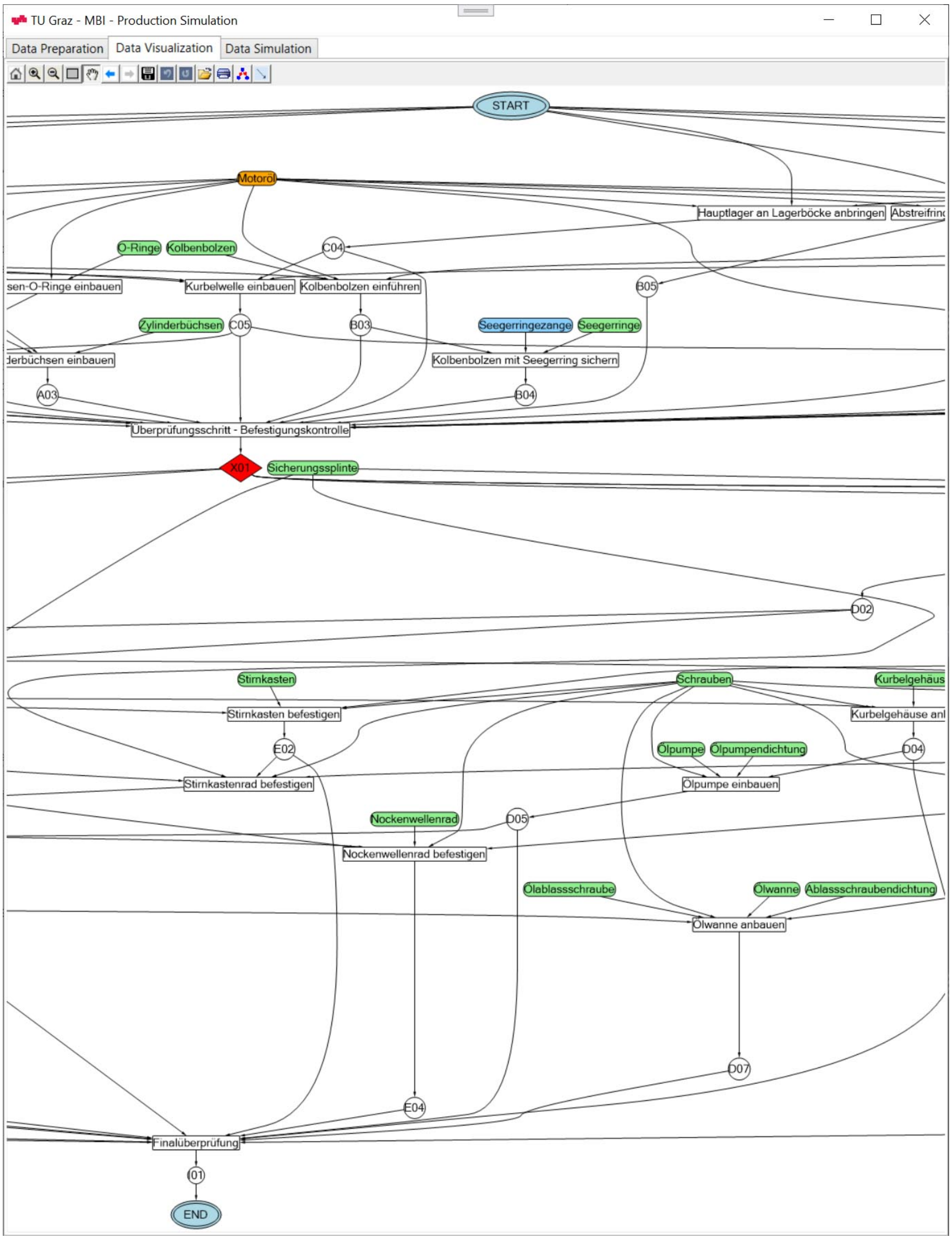


Figure 4.3.: Zoomed screenshot of the data visualization part of the software

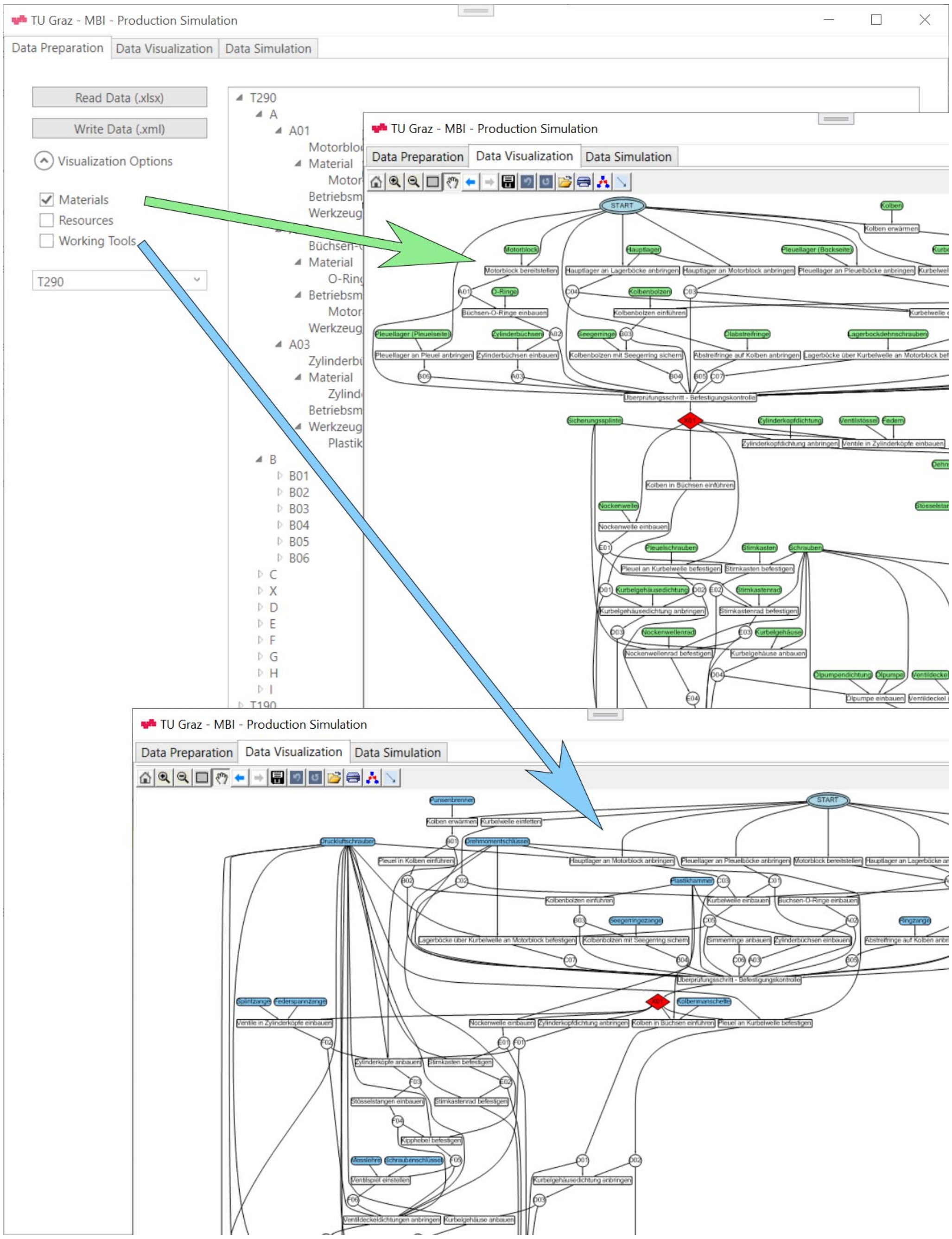


Figure 4.4.: Petrinets at different kind of visualization options

4.4. Simulation

4.4.1. Expectations

To put the modular production line, which is shown by the Petrinet, to the test with real variables an event triggered simulation is used. Therefore the HCCM-framework which is described in the theory part [Fur14] is integrated in the developed software (code-usage and extension of Nikolaus Furian's Health-Care-Simulation).

There are several questions which should be answered by the expected results for someone who may use the simulation:

- Can any *bottlenecks* be found by the simulation?
- What are the effects of changing the *number of working stations*?
- What is the impact if the *skills of the working stations* are changed?
- When having different building types which each have a different number of production steps - what are the effects if the *order list changes*?
- What are the effects of changed requirements of materials, working tools and production resources? What about delivery difficulties?
- Can machine or *working station failures* be compensated at run-time?
- What does the *state of a queue* show? Is there a direct relation between full queues and a too small number of physical working places with the needed skills?
- Can the dynamic data input (EXCEL-list) help to react in run-time to change production steps, modules, work places or working stations/skills?

If the simulation is build up as dynamic and realistic as possible to deliver answers on these questions, possible scenarios can be drawn up and the simulation can be used in reality.

4.4.2. Basic Definitions

Therefore as in the visualization part with Petrinets some requirements have to be designed. They overlap in some points but do have some more software-development-specific expectations.

- **Definition of work places**
  A work place is the physical representation where a worker assembles certain materials with special working tools at a certain time to components.

A work place can represent several working stations (skills), but only a specific one at a given time.

- **Definition of working stations**
  A working station is a description of how to do a certain job with certain skills in a spatially undefined place.
  A working station consists of a set of production steps.
  A working station (set of skills) can be offered at several work places.
- **Definition of production steps**
  A production step is the smallest possible work performed on a component.
- **Definition of order lists**
  An order list is a listing of construction types ordered by the customer or market.
- **Work places** do have queues (waiting bays) where components are waiting to be processed.
- **A work place** does a look-up in all queues of his skills if there is any waiting component and picks up the one with the oldest timestamp. The fetching works after a FIFO (first in first out) principle.

The draft A.5 shows an overview how basically the simulation process should work. The order list is a sequence planning list which building types (here Steyr T188, T190 and T290) are requested by the customer. For each building type there is a specific building instruction which production steps have to be fulfilled. With this information the simulation is fed.

4.4.3. Implementation

As seen in A.4 the work stations A, B and C can be done in a parallel way. How parallelization and queues work together in the simulation can be seen in the next figure 4.5.

This figure also shows the basic design of queues and work places and furthermore the finishing states of order items.

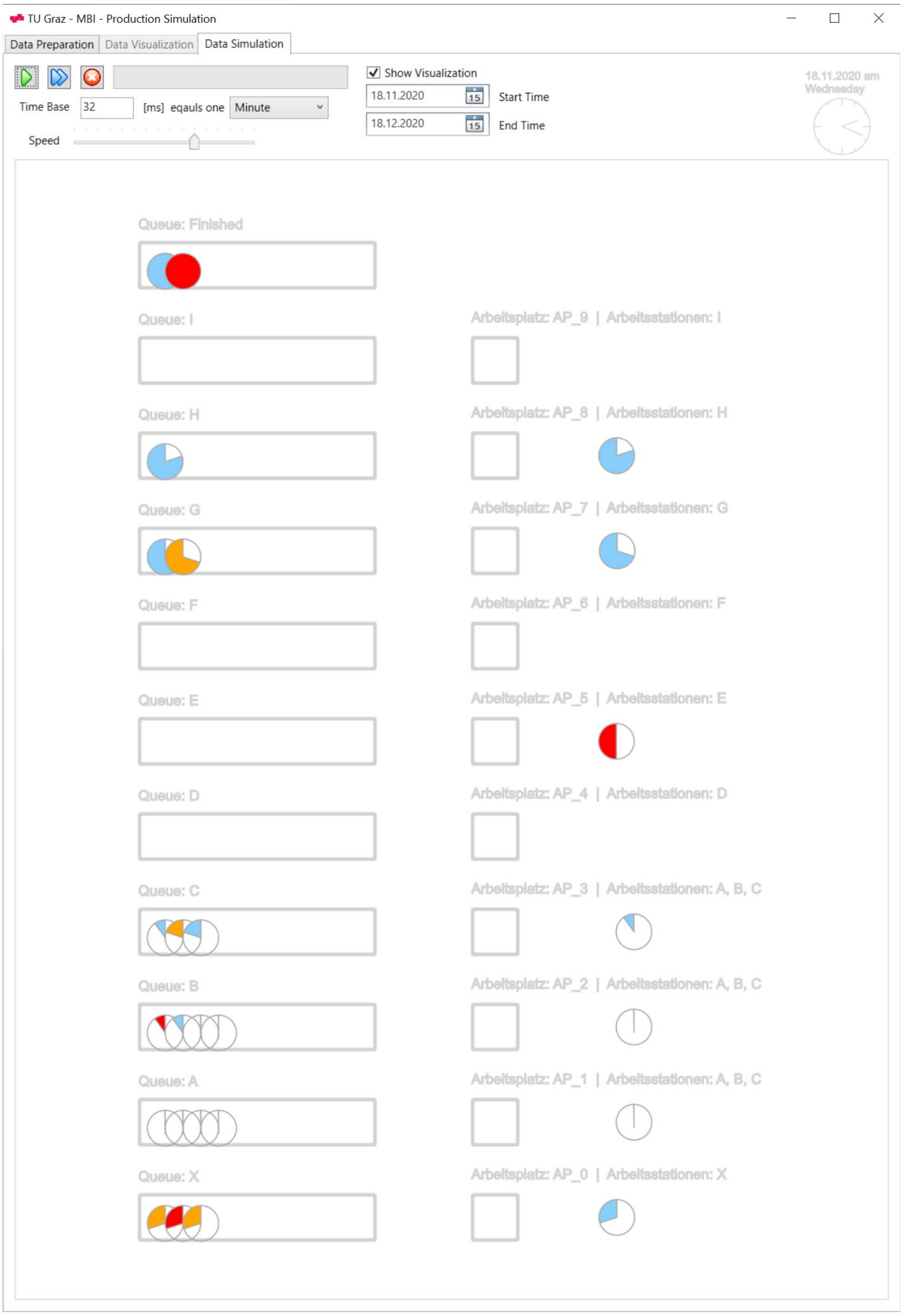


Figure 4.5.: Overview of the simulation part of the developed software

Skill	Description	Processing Time [sec]
X	Quality Check	300
A	Engine Block	2800
B	Piston	1000
C	Crankshaftk	2400

Table 4.1.: Test setting for test cases

Verification of Simulation with Test Setting

To show that the implementation of the simulation works correctly, a verification is necessary. The difference[Eas] between verification and validation is, that at verification the correctness of functional requirements of the simulation tool is proven and at validation the output accuracy is tested. As in this case no real data is available, the implementation of the model is verified with the created pseudo data.

Therefore a set of 3 different cases is tested. Conditions for these cases are:

- Transport latency has to be zero
- Input data (especially order data) remains the same at all test cases
- Processing time of each Workstation remains the same at all test cases
- A, B and C can be done parallel, X is a quality check which has A, B and C as predecessor
- Number of ordered items set to **100**
- New order item every **2800 seconds**
- Minimum processing time **6500 second**
- Processing time for skills like in 4.1
- Test case distribution like in 4.2

Simulating the given test cases with the predefined test setting leads to the results in table 4.3.

Interpretation of Results

When looking at table of results 4.3, the functionality of the simulation model and consequently the simulation itself is proven.

In case of **Alpha** the skills are evenly distributed and so the *average queue length* is in a normal spectrum. Also the *average processing time* does not deviate much from the *minimum production time*.

Test Case	Work Place	Skills
Alpha	AP_1	X
	AP_2	A, B, C
	AP_3	A, B, C
	AP_4	A, B, C
Bravo	AP_1	X
	AP_2	A, B, C
	AP_3	B, C
	AP_4	B, C
Charlie	AP_1	X
	AP_2	B, C
	AP_3	B, C
	AP_4	B, C

Table 4.2.: Test cases to proof simulation functionality

Test Case	Minimum Production Time [sec]	Minimum Processing Time [sec]	Maximum Processing Time [sec]	Average Processing Time [sec]	Skill	Average Queue Length
Alpha	6.500	8.500	16.100	9.900	A	1
					B	2
					C	2
					X	0
Bravo	6.500	14.900	23.900	17.700	A	5
					B	1
					C	1
					X	0
Charlie	6.500	∞	∞	∞	A	100
					B	2
					C	2
					X	0

Table 4.3.: Test results of previous test setting

In case of **Bravo** the skill *A*, which needs the most production effort, is represented just in one working place. This leads to a higher *average queue length* of skill *A* and a much higher *average processing time*.

Case **Charlie** cannot finish any item due to no skill *A* in any of the working places. Finally every item is waiting in queue of skill *A*. Remarkable is, that every item finishes skill *B* and *C* because of the special *stock-per-skill technique* and the fact, that through this technique every item is waiting in all queues of needed skills.

4.4.4. GUI Explanation

The GUI of the simulation is an important part to visually show how the simulation works.

Taking the figure 4.6 as example, one can see the physical work place **AP_7** which has the skills to do **working station G**. Furthermore there is a tractor of type **T190 in light blue** completed to two-thirds at present in the work place and gets assembled. In the **queue for working station G** are two further vehicles waiting to get served.

Figure 4.6.: A work place with one skill and its corresponding queue

In comparison to that in figure 4.7 the work place **AP_3** with the skills for working stations **A**, **B** and **C** fetches his next item from one of the three corresponding queues. And as the three work stations A, B and C are parallelized, one can see that the items in queue C and B contain items with different completetion states (can be seen on the circular pieces).

The figure 4.8 shows an overview of queues which are overfilled. This can for example result from a to high input rate of new ordered items to the production line and can be localized in the GUI at a glance.

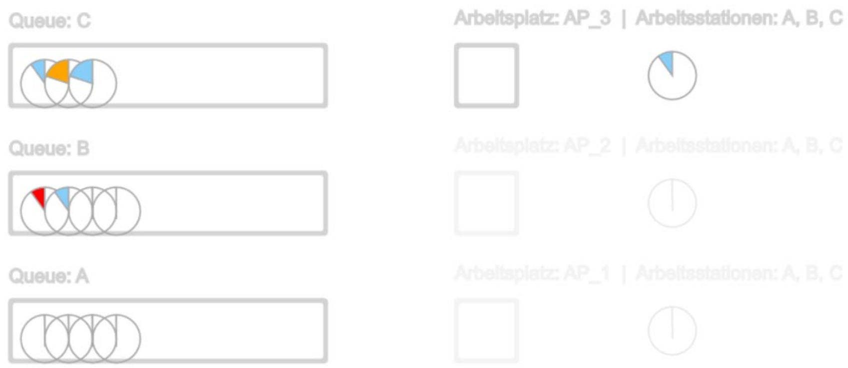


Figure 4.7.: A work place with three different skills and the corresponding queues

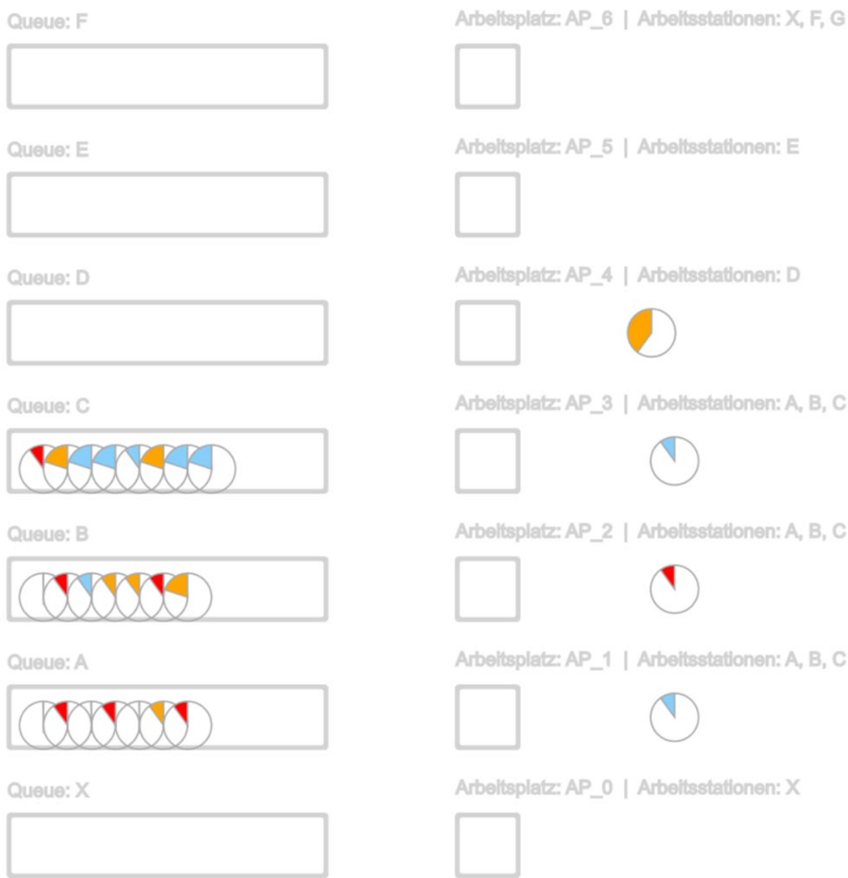


Figure 4.8.: Overview of overfilled queues and work places

4.4.5. Code Explanation

In the appendix the most important code snippets of the simulation part are attached at C.2.5. As the name indicates, the whole simulation is build up on events. So for every state change an event is thrown.

Initially activities like **ActivityGetServed** build the basis who call state-change-events like **Start** and **End**. With their inherited classes of **requests (QueingRequest, FinishedRequest)**, which are specialized **activities**, they are handled in the **ControlUnit**. The ControlUnit is also responsible for the different **machines (physical work places)** and the **stocks (queues of working stations)**.

The difference to the original framework of HCCM health care is, that the different entities have skills (EntityClient, EntityQueue, EntityWorkPlace).

So a normal procedure can be described as following:

- Initially the **ControlUnit** creates the **pyhsical work places**. Afterwards every work place gets it set of **skills/working stations**.
- A **request** can be thrown centrally from ControlUnit or from the **order item (EntityClient)** itself.
- A order item (EntityClient) which still has unfinished production steps in its checklist, will be added to all queues of the skills which can be done next (graph of previous needed steps).
- Everytime a machine (work place) is idle, the ControlUnit creates a new **ActivityGetServed** with the certain machine and a corresponding order item who still needs one of the machine skills in its checklist.
- As there is a stock-per-skill-implementation, the ControlUnit has to check all corresponding queues of the machine's skill set and take the longest waiting order item (EntityClient).
- After finishing the production step, the machine throws a finished event, the step is set to done in internal checklist of the order item (EntityClient) and it will be enqueued to the queues of still needed skills.

4.5. Conclusion

After uniting the parts of theory and ideas to this practical approach, in summary, several findings can be noted.

Firstly there is a lot of scientific information concerning modular production, several papers also for the case in automotive industry. But none of them provides an approach fitting the case that a production line is used for specialized varieties of vehicles. There are approaches from modular platforms of the vehicle itself to outsourced modules. But having modular work stations inside a manufacturers area which can be accessed in a random way seems to be an unexplored topic.

Secondly Petrinets indeed can help representing and calculating graphs for this challenge. Although in practical case and for simulation there is a need of much more additions to handle such a production line. This is where the theory reaches its limits.

Thirdly in research part there were found some ideas like stock-per-skill and checklists of items (vehicles) which indeed could be helpful approaches for real-life systems.

Fourthly the simulation is able to show in different cases and setups the bottlenecks and possible occurring problems in a production line. Due to the ability to react quickly on the dynamic data input, the simulation setup can be changed quickly. Furthermore the HCCM-framework is supportive to run the simulation with real-time data (like needed time span) in either slow motion or fast forward.

Finally the described possibilities and inventions still means an overhead for a production manager. He has to handle all the requirements of work stations and the logistic behind all resources. Nevertheless, the developed software can be a comfortable help in simulating and planning production line and - with a few simple changes - also a tool and mechanism to keep control.

Part V.

Appendix

Appendix A.

Drafts

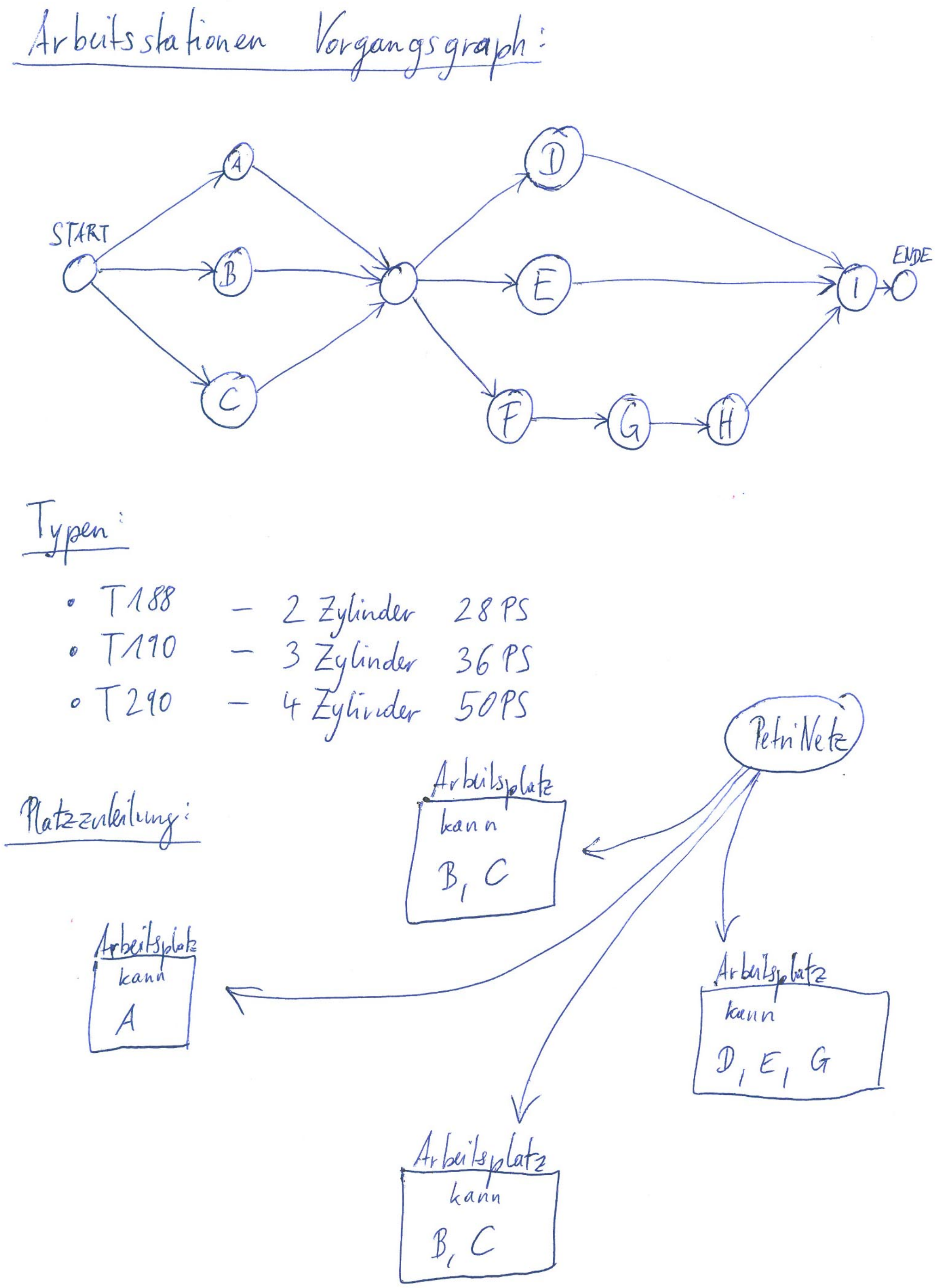


Figure A.1.: Pseudo Data Draft Overview

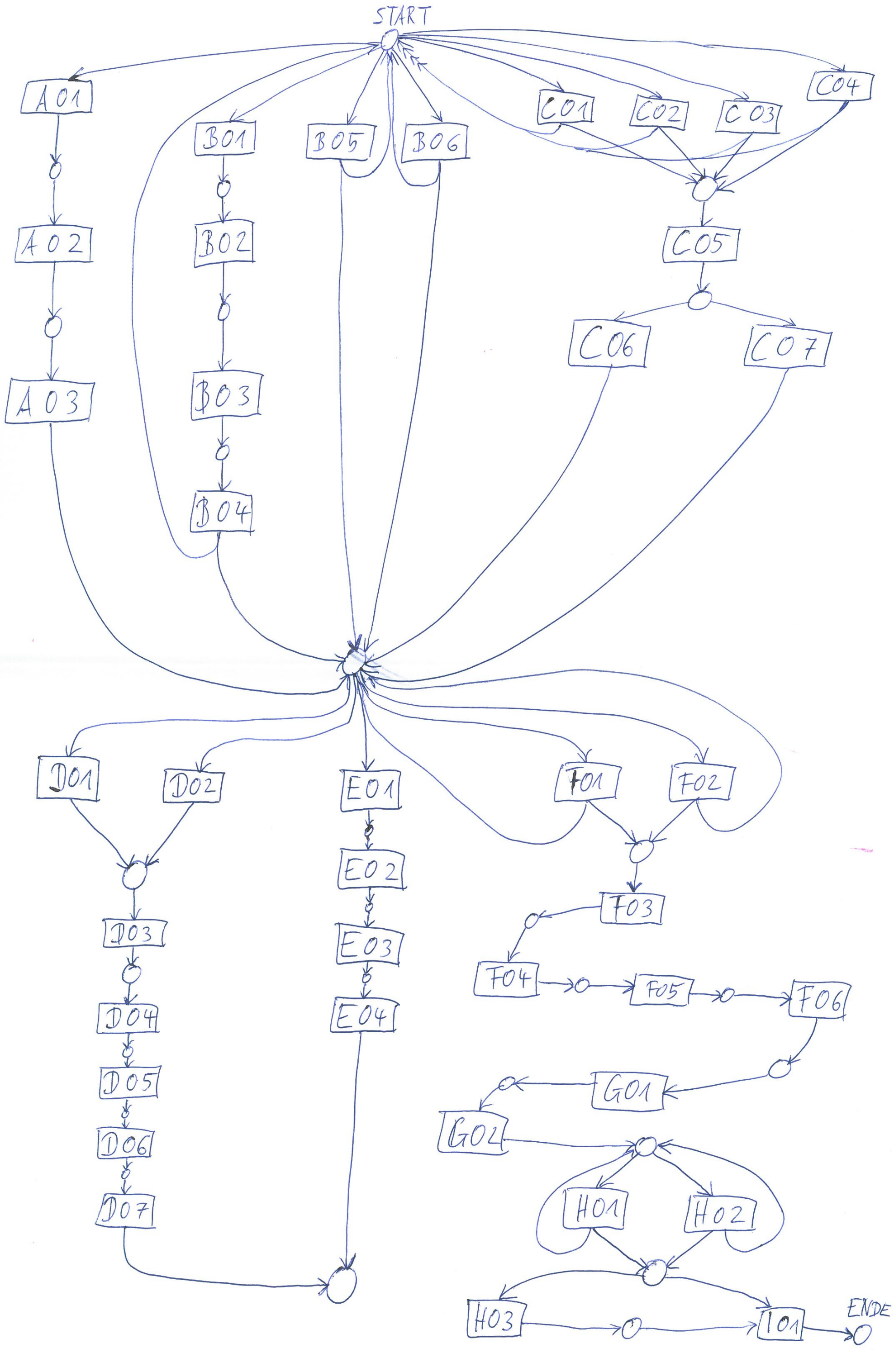


Figure A.2.: Pseudo Data Draft Detail

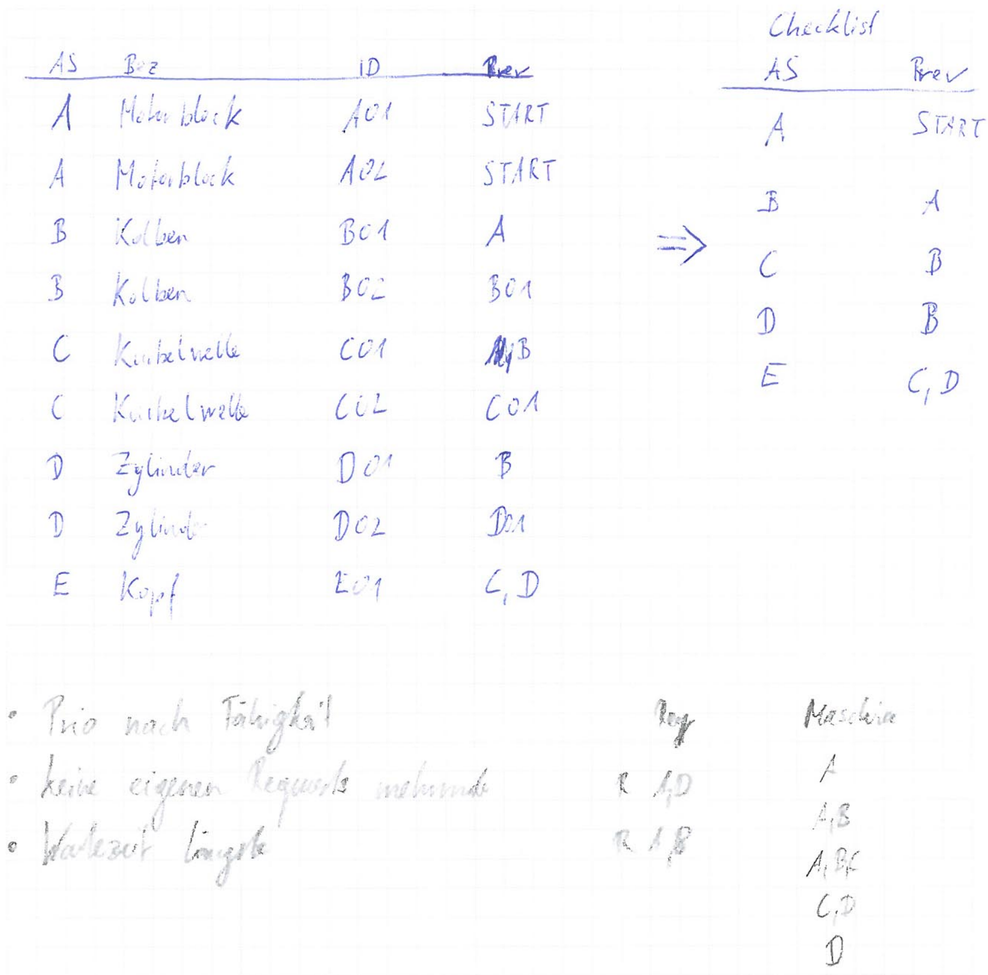


Figure A.3.: Simulation Draft of Checklists

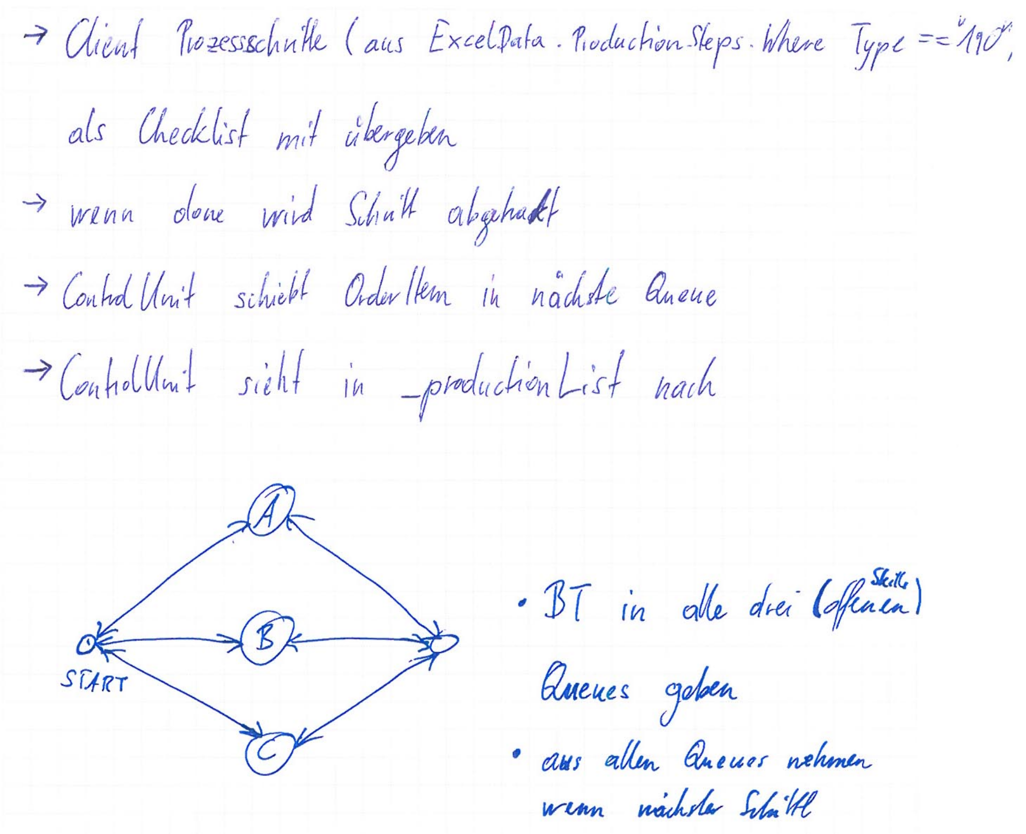


Figure A.4.: Simulation Draft of Control Unit

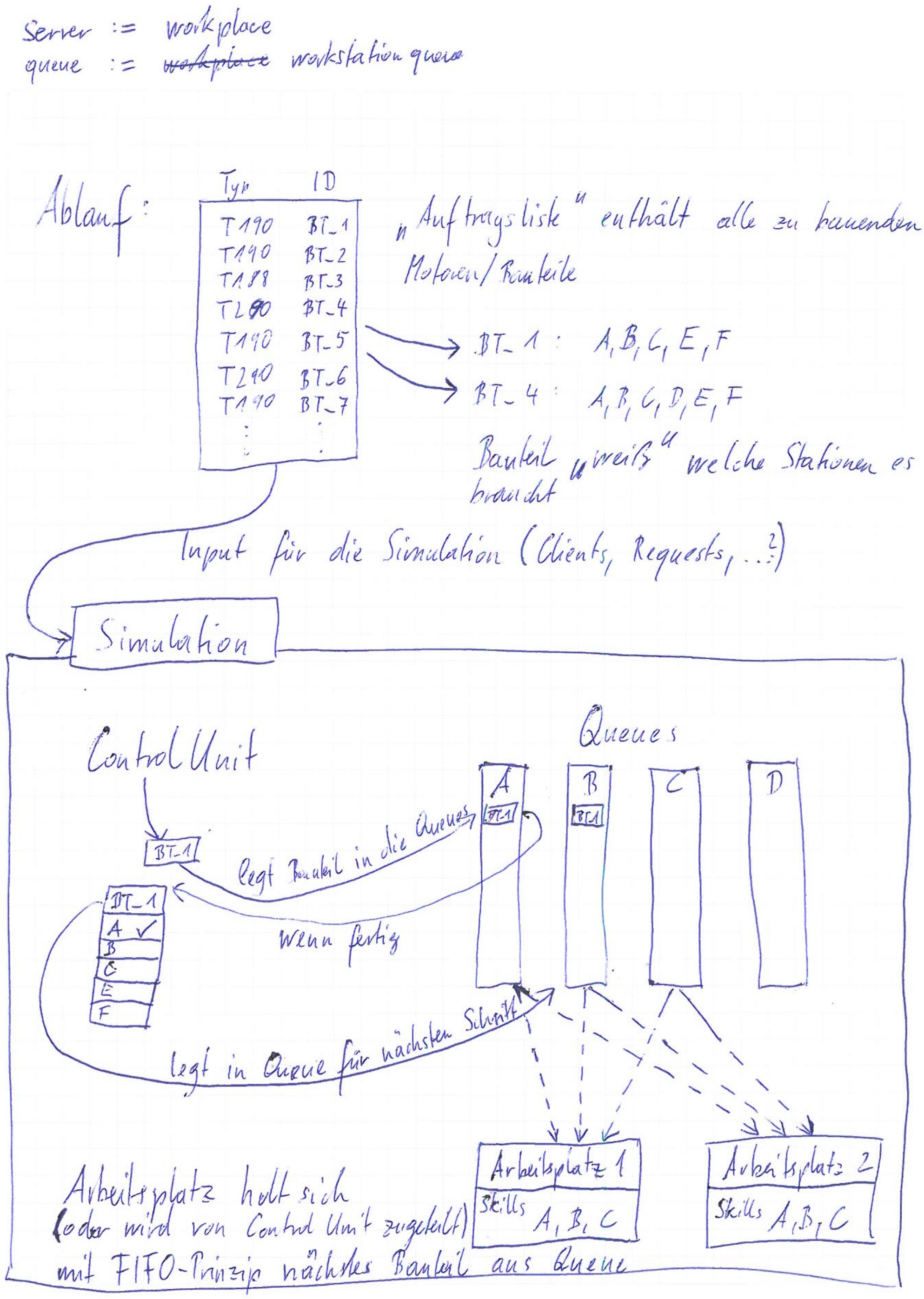


Figure A.5.: Simulation Draft of Machines and Queues

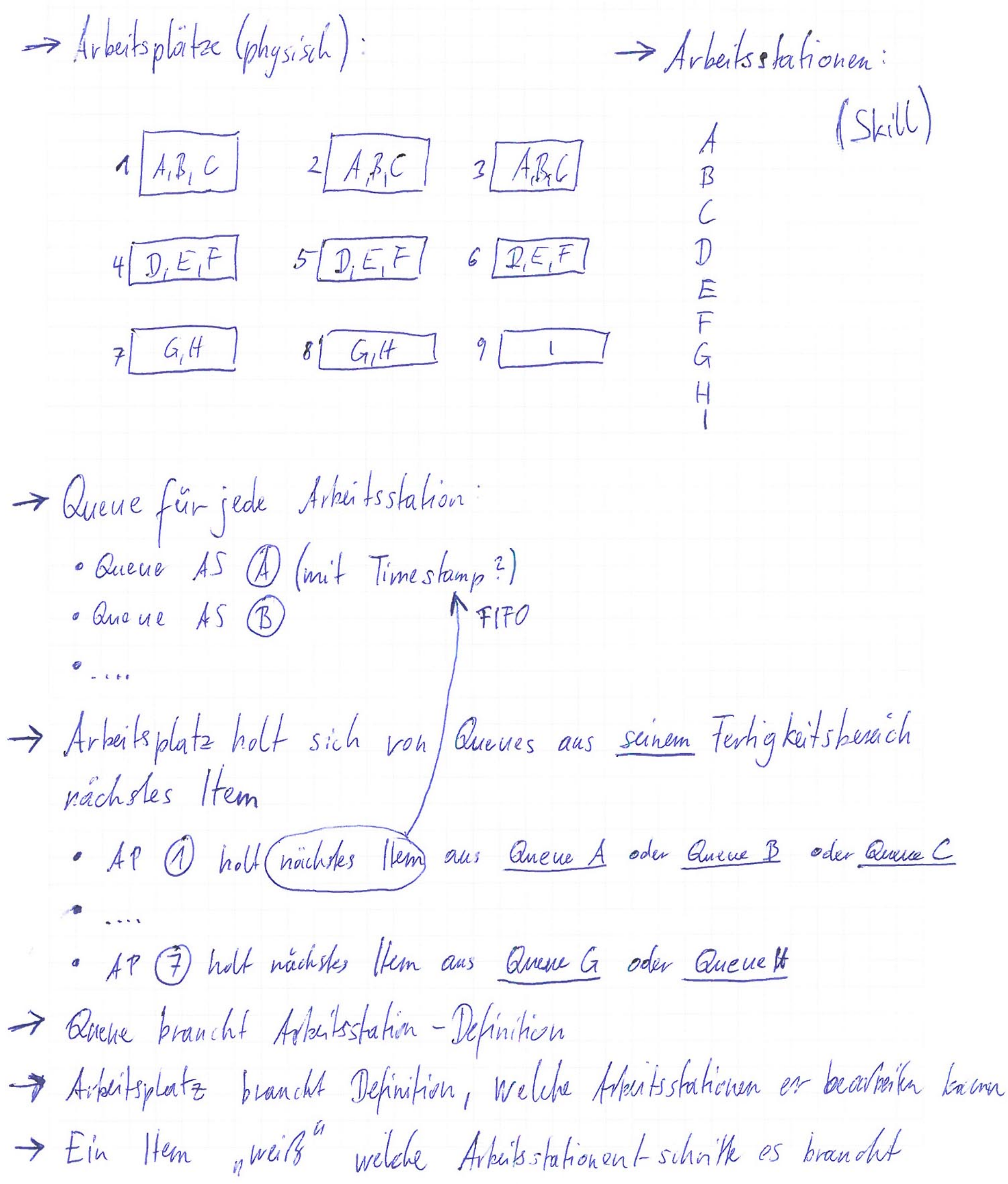


Figure A.6.: Simulation Draft of Workstations and Workplaces

Appendix B.

Lists

Marke	Bautyp	Arbeitsschritt-ID	Arbeitsschritt-Beschreibung	Arbeitsschritt-Typ	Arbeitsstation	Material	Betriebsmittel	Werkzeug	Vorausgehende Arbeitsschritte
Steyr	Alle	START	Start - Initialisierung	None					
Steyr	Alle	A01	Motorblock bereitstellen	Work	A	Motorblock			START
Steyr	Alle	A02	Büchsen-O-Ringe einbauen	Work	A	O-Ringe	Motoröl		A01
Steyr	Alle	A03	Zylinderbüchsen einbauen	Work	A	Zylinderbüchsen		Plastikhammer	A02
Steyr	Alle	B01	Kolben erwärmen	Work	B	Kolben		Punsenbrenner	START
Steyr	Alle	B02	Pleuel in Kolben einführen	Work	B	Pleuel			B01
Steyr	Alle	B03	Kolbenbolzen einführen	Work	B	Kolbenbolzen	Motoröl	Plastikhammer	B02
Steyr	Alle	B04	Kolbenbolzen mit Seegerring sichern	Work	B	Seegerringe		Seegerringezange	B03
Steyr	Alle	B05	Abstreifringe auf Kolben anbringen	Work	B	Ölabstreifringe	Motoröl	Ringzange	START
Steyr	Alle	B06	Pleuellager an Pleuel anbringen	Work	B	Pleuellager (Pleuelseite)	Motoröl		START
Steyr	Alle	C01	Pleuellager an Pleuelböcke anbringen	Work	C	Pleuellager (Bockseite)	Motoröl		START
Steyr	Alle	C02	Kurbelwelle einfetten	Work	C	Kurbelwelle	Graphitschmierfett		START
Steyr	Alle	C03	Hauptlager an Motorblock anbringen	Work	C	Hauptlager	Motoröl	Druckluftschrauber, Drehmomentschlüssel	START
Steyr	Alle	C04	Hauptlager an Lagerböcke anbringen	Work	C	Hauptlager	Motoröl		START
Steyr	Alle	C05	Kurbelwelle einbauen	Work	C				C01, C02, C03, C04
Steyr	Alle	C06	Simmerringe anbauen	Work	C	Simmerringe	Graphitschmierfett		C05
Steyr	Alle	C07	Lagerböcke über Kurbelwelle an Motorblock befestigen	Work	C	Lagerbockdehnschrauben		Druckluftschrauber, Drehmomentschlüssel	C05
Steyr	Alle	X01	Überprüfungsschritt - Befestigungskontrolle	Check	X				A, B, C
Steyr	Alle	D01	Kolben in Büchsen einführen	Work	D		Motoröl	Kolbenmanschette, Plastikhammer	X01
Steyr	Alle	D02	Pleuel an Kurbelwelle befestigen	Work	D	Pleuelschrauben		Druckluftschrauber, Drehmomentschlüssel	X01
Steyr	Alle	D03	Kurbelgehäusedichtung anbringen	Work	D	Kurbelgehäusedichtung	Graphitschmierfett		D01, D02
Steyr	Alle	D04	Kurbelgehäuse anbauen	Work	D	Kurbelgehäuse, Schrauben		Druckluftschrauber	D03
Steyr	Alle	D05	Ölpumpe einbauen	Work	D	Ölpumpe, Ölpumpendichtung, Schrauben			D04
Steyr	Alle	D06	Ölwannendichtung einbringen	Work	D	Ölwannendichtung	Graphitschmierfett		D05
Steyr	Alle	D07	Ölwanne anbauen	Work	D	Ölwanne, Ölablassschraube,		Druckluftschrauber	D06
Steyr	Alle	E01	Nockenwelle einbauen	Work	E	Nockenwelle		Plastikhammer	X01
Steyr	Alle	E02	Stirnkasten befestigen	Work	E	Stirnkasten, Schrauben		Druckluftschrauber	E01
Steyr	Alle	E03	Stirnkastenrad befestigen	Work	E	Stirnkastenrad, Schrauben, Sicherungssplinte		Druckluftschrauber	E02
Steyr	Alle	E04	Nockenwellenrad befestigen	Work	E	Nockenwellenrad, Schrauben, Sicherungssplinte		Druckluftschrauber	E03
Steyr	Alle	F01	Zylinderkopfdichtung anbringen	Work	F	Zylinderkopfdichtung			X01
Steyr	Alle	F02	Ventile in Zylinderköpfe einbauen	Work	F	Ventilstössel, Federn, Sicherungssplinte		Federspannzange, Splintzange	X01
Steyr	T290	F03	Zylinderköpfe anbauen	Work	F	Zylinderköpfe, Dehnschrauben		Druckluftschrauber, Drehmomentschlüssel	F01, F02
Steyr	T190	F03	Zylinderkopf anbauen	Work	F	Zylinderkopf, Dehnschrauben		Druckluftschrauber, Drehmomentschlüssel	F01, F02
Steyr	T188	F03	Zylinderkopf anbauen	Work	F	Zylinderkopf, Dehnschrauben		Druckluftschrauber, Drehmomentschlüssel	F01, F02
Steyr	Alle	F04	Stösselstangen einbauen	Work	F	Stösselstangen, Stösselschalen	Motoröl		F03
Steyr	Alle	F05	Kipphebel befestigen	Work	F	Kipphebel, Schrauben	Motoröl	Druckluftschrauber, Drehmomentschlüssel	F04
Steyr	Alle	F06	Ventilspiel einstellen	Work	F			Messlehre, Schraubenschlüssel	F05
Steyr	T290	G01	Ventildeckeldichtungen anbringen	Work	G	Ventildeckeldichtungen			F
Steyr	T190	G01	Ventildeckeldichtung anbringen	Work	G	Ventildeckeldichtung			F
Steyr	T188	G01	Ventildeckeldichtung anbringen	Work	G	Ventildeckeldichtung			F
Steyr	T290	G02	Ventildeckel (beide) anbauen	Work	G	Ventildeckel, Schrauben		Druckluftschrauber	G01
Steyr	T190	G02	Ventildeckel anbauen	Work	G	Ventildeckel, Schrauben		Druckluftschrauber	G01
Steyr	T188	G02	Ventildeckel anbauen	Work	G	Ventildeckel, Schrauben		Druckluftschrauber	G01
Steyr	Alle	H01	Abgaskrümmer anbauen	Work	H	Abgaskrümmer, Abgaskrümmerdichtungen		Druckluftschrauber	G
Steyr	Alle	H02	Ansaugkrümmer anbauen	Work	H	Ansaugkrümmer, Ansaugkrümmerdichtungen		Druckluftschrauber	G
Steyr	T290	H03	Turboladereinbau zwischen Abgas- und Ansaugkrümmer	Work	H	Turbolader, Turboladerdichtungen		Druckluftschrauber	H01, H02
Steyr	T290	I01	Finalüberprüfung	Work	I				D, E, H03
Steyr	T190	I01	Finalüberprüfung	Work	I				D, E, H01, H02
Steyr	T188	I01	Finalüberprüfung	Work	I				D, E, H01, H02
Steyr	Alle	END	Ende - Finalisierung	None					I

Figure B.1.: Pseudo Data Draft Overview

Arbeitsplatz (physisch)	Arbeitsstationen (organisatorisch) Fähigkeiten des Arbeitsplatzes
AP_0	X
AP_1	A, B, C
AP_2	A, B, C
AP_3	A, B
AP_4	D
AP_5	C, E
AP_6	F
AP_7	G
AP_8	H
AP_9	I

Arbeitsstation	Bezeichnung
X	Überprüfung
A	Motorblock
B	Kolben
C	Kurbelwelle
D	Kurbelgehäuse
E	Nockenwelle
F	Zylinderköpfe
G	Ventildeckel
H	Krümmer
I	Finalisierung

Figure B.2.: Pseudo Data Draft Detail

Appendix C.

Code Snippets

C.1. Python Code

C.1.1. Petrinet Exporting

```
import pandas as pd
from pandas import ExcelWriter
from pandas import ExcelFile
from collections import defaultdict

import os
import subprocess
import re
import string

import snakes.plugins
snakes.plugins.load('gv', 'snakes.nets', 'nets')
from nets import *

# function for data input
def readExcel():
    # reading the Excel worksheet
    Steyr = pd.read_excel('Input\\DataSteyr.xlsx',
    sheet_name='Steyr')

    # filtering the type
    # T190 and T290 will be filtered away, T188 remains
    Steyr = Steyr[Steyr.Bautyp != 'T190']
    Steyr = Steyr[Steyr.Bautyp != 'T290']

    # creating a list of working step IDs
    global arbeitsschritteIDs
    arbeitsschritteIDs = Steyr['Arbeitsschritt-ID'].unique()
    idsDict = defaultdict(list)

    # creating a dictionary of IDs
    for element in arbeitsschritteIDs:
        key = ''.join([ele for ele in element if not ele.
    isdigit()])

        if key in idsDict.keys():
            idsDict.setdefault(key, []).append(element)
        else:
            idsDict.update({key : [element]})

    # creating a dictionary of working step IDs and the
    corresponding description
    global arbeitsschritteDict
    arbeitsschritteDict = dict(zip(Steyr['Arbeitsschritt-ID'
    ], Steyr['Arbeitsschritte']))

    # debug-output of the working steps dictionary
```

```
    print(arbeitsschritteDict)

    # creating a dictionary with working steps and their corresponding previous workingsteps (must-have's)
    global abfolgeDict
    global abfolgeDictComplete
    abfolgeDict = dict(zip(Steyr['Arbeitsschritt-ID'], Steyr['Vorausgehende Arbeitsschritte']))
    abfolgeDictComplete = defaultdict(list)

    # the abfolgeDictComplete dictionary consists of the direct previous working steps
    # and also of all of the required steps
    for key in abfolgeDict.keys():
        element = abfolgeDict[key]
        values = str(element).split(",")
        values = [item.strip() for item in values]

        valuescomplete = list()
        for val in values:
            if val.strip() in idsDict.keys():
                ids = idsDict[val]
                for id in ids:
                    valuescomplete.append(id)
            else:
                valuescomplete.append(val)

        abfolgeDictComplete[key] = valuescomplete

    # debug-output of the complete previous steps dictionary
    print(abfolgeDictComplete)



# function for creating the Petrinet
def createPetrinet():
    petrinet = PetriNet("PetriNet for Magna")
    print ("Starting ...")

    # adding so-called places/stati to the Petrinet
    # here the working step IDs are places/stati
    for status in arbeitsschritteDict.keys():
        petrinet.add_place(Place(status))

    print("Added all stati ...")

    # adding all transitions to the Petrinet
    # here the working step descriptions represent the transitions
    for transition in arbeitsschritteDict.values():
        petrinet.add_transition(Transition(transition))

```

```
    print("Added all transitions ...")

    # adding all inputs between places and transitions
    for status in abfolgeDictComplete.keys():
        transition = arbeitsschritteDict[status]
        petrinet.add_input(status, transition, Variable("x"))

    print("Added all inputs ...")

    # connecting all transitions and places in the petrinet due to the information of previous working steps
    for status in arbeitsschritteDict.keys():
        transition = arbeitsschritteDict[status]
        for vorgaenger in abfolgeDictComplete[status]:
            if vorgaenger != 'nan':
                vorgaenger_transition = arbeitsschritteDict[vorgaenger]
                petrinet.add_output(status, vorgaenger_transition, Variable("y"))

    print("Added all outputs ...")


    # removing old output files if exist
    if os.path.exists("Output\\Steyr.png"):
        os.remove("Output\\Steyr.png")

    # drawing the Petrinet into a new output file
    try:
        petrinet.draw("Output\\Steyr.png", place_attr=draw_place, trans_attr=draw_transition, arc_attr=draw_arc)
        print("Finished drawing.")
    except Exception as ex:
        print("Error: " + str(ex))

    try:
        subprocess.call("\"Python 3.7 Conda\\Library\\bin\\graphviz\\dot\" -T png \".\\Output\\Steyr.png.dot\" -o \".\\Output\\Steyr.png\"")
    except Exception as ex:
        print("Error2: " + str(ex))



# help-function for drawing a place/status
def draw_place (place, attr):
    # setting the place/status name
    attr['label'] = place.name

    # setting the attribute-color
```

```
    if place.name in str(arbeitsschritteDict.keys()):
        attr['color'] = '#00FF00' #green color

# help-function for drawing a transition
def draw_transition (trans, attr):
    if str(trans.guard) == 'True':
        attr['label'] = trans.name
    else:
        attr['label'] = '%s\n%s' % (trans.name, trans.guard)

# help-function for drawing an arc (removing description)
def draw_arc(arc, attr):
    attr['label'] = ""


# main-function calls the data input and the visualization
def main():
    readExcel()
    createPetrinet()

if __name__ == "__main__":
    main()
```

C.2. C# Code

C.2.1. Import and Export Execution

```
public class DataHandler
{
    /// <summary>
    /// Imports the data from Excel worksheet and returns an ExcelData object.
    /// </summary>
    public ExcelData ImportFromExcel(string path)
    {
        try
        {
            // creating a new NPoi-Excel-mapper and reading the needed colums from worksheet
            Mapper mapper = new Mapper(path);
            List<ProductionStep> productionSteps = mapper
                .Take<ProductionStep>("Produktionsschritte")
                .Select(x => x.Value)
                .ToList();
            List<WorkPlace> workPlaces = mapper
                .Take<WorkPlace>("Arbeitsplaetze")
                .Select(x => x.Value)
                .ToList();
            List<WorkStation> workStations = mapper
                .Take<WorkStation>("Arbeitsstationen")
                .Select(x => x.Value)
                .ToList();
            List<OrderItem> orderItems = mapper
                .Take<OrderItem>("Auftragsliste")
                .Select(x => x.Value).
                ToList();

            return new ExcelData(productionSteps, workPlaces, workStations, orderItems);
        }
        catch (IOException io)
        {
            Debug.Assert(false, "ImportFromExcel - Can't access file because it's opened or protected.");
        }
        catch (Exception ex)
        {
            Debug.Assert(false, "ImportFromExcel");
        }

        return new ExcelData(new List<ProductionStep>(), new List<WorkPlace>(), new List<WorkStation>(), new
```

```
List<OrderItem>());
    }

    /// <summary>
    /// Writes the given ExcelData object to an xml-file (
object structure).
    /// </summary>
    public void WriteToXML(string path, ExcelData
excelData)
    {
        XmlTextWriter writer = null;
        try
        {
            writer = new XmlTextWriter(path, Encoding.UTF8
);
            writer.Formatting = Formatting.Indented;
            writer.Indentation = 4;

            // calculating previous production steps
            RecalculatePreviousSteps(excelData.
ProductionSteps);

            // serializing the ExcelData object to an xml-
object and writing it to file
            XmlSerializer serializer = new XmlSerializer(
typeof(ExcelData));
            serializer.Serialize(writer, excelData);
        }
        catch (Exception ex)
        {
            Debug.Assert(false, "WriteToXML");
        }
        finally
        {
            writer.Close();
        }
    }

    /// <summary>
    /// Recalculates all previous working steps of each
production step in the given list.
    /// </summary>
    private void RecalculatePreviousSteps(List<
ProductionStep> productionSteps)
    {
        // creating a list of all possible (distinct)
working steps
        List<string> allsteps = ProductionStep.
GetWorkingStepsDistinct();
        allsteps.Remove("END");

        foreach (ProductionStep item in productionSteps)
```

```
        {
            var groupsteps = item.PreviousWorkingSteps.
    Except(allsteps);
            var replacedgroupsteps = allsteps.Where(step
    => groupsteps.Any(groupstep => step.StartsWith(
    groupstep)));

            item.PreviousWorkingSteps = item.
    PreviousWorkingSteps
                .Except(groupsteps)
                .Union(replacedgroupsteps)
                .ToList();
        }
    }
}
```

C.2.2. Excel and XML Serialization Classes

```

public class ProductionStep
{
    /// <summary>
    /// Represents the Status in a PetriNet
    /// </summary>
    private string _productionStepID;
    [Column("Arbeitsschritt-ID")]
    [XmlAttribute("ID")]
    public string ProductionStepID
    {
        get
        {
            return _productionStepID;
        }
        set
        {
            _productionStepID = value;
            AllWorkingSteps.Add(value);
        }
    }

    [Column("Marke")]
    [XmlAttribute("Brand")]
    public string Brand { get; set; } = "";

    [Column("Bautyp")]
    [XmlAttribute("BuildingType")]
    public string BuildingType { get; set; } = "";

    /// <summary>
    /// Represents the Transition in a PetriNet
    /// </summary>
```

```
    [Column("Arbeitsschritt-Beschreibung")]
    [XmlElement("ProductionStepDescription")]
    public string ProductionStepDescription { get; set; } = "";

    [Column("Arbeitsschritt-Typ")]
    [XmlAttribute("ProductionStepType")]
    public ProductionStepType ProductionStepType { get; set; } = ProductionStepType.Work;

    [Column("Arbeitsstation")]
    [XmlElement("WorkingStation")]
    public string WorkingStation { get; set; } = "";


    /// <summary>
    /// Represents the corrected MaterialList
    /// </summary>
    private string _materials = "";
    [Column("Material")]
    [XmlIgnore]
    public string Materials
    {
        get
        {
            return _materials;
        }
        set
        {
            _materials = value;

            if (MaterialList?.Count > 0)
            {
                AllMaterials.AddRange(MaterialList);
            }
        }
    }

    [XmlArray("MaterialList")]
    [XmlArrayItem("Material")]
    public List<string> MaterialList
    {
        get
        {
            return _materials?
                .Split(new[] { ',', ';' }, StringSplitOptions.RemoveEmptyEntries)?
                .ToList()
                .Select(x => x = x.Trim())
                .ToList() ?? new List<string>();
        }
    }
```

```
    /// <summary>
    /// Represents the corrected ResourcesList
    /// </summary>
    private string _resources = "";
    [Column("Betriebsmittel")]
    [XmlIgnore]
    public string Resources
    {
        get
        {
            return _resources;
        }
        set
        {
            _resources = value;

            if (ResourcesList?.Count > 0)
            {
                AllResources.AddRange(ResourcesList);
            }
        }
    }
    [XmlArray("ResourcesList")]
    [XmlArrayItem("Resource")]
    public List<string> ResourcesList
    {
        get
        {
            return _resources?
                .Split(new[] { ',', ';' },
   StringSplitOptions.RemoveEmptyEntries)?
                .ToList()
                .Select(x => x = x.Trim())
                .ToList() ?? new List<string>();
        }
    }

    /// <summary>
    /// Represents the corrected WorkingTools
    /// </summary>
    private string _workingtools = "";
    [Column("Werkzeug")]
    [XmlIgnore]
    public string WorkingTools
    {
        get
        {
            return _workingtools;
        }
        set
        {
```

```
            _workingtools = value;

            if (WorkingToolsList?.Count > 0)
            {
                AllWorkingTools.AddRange(WorkingToolsList)
;
            }
        }
    }

    [XmlArray("WorkingToolsList")]
    [XmlArrayItem("WorkingTool")]
    public List<string> WorkingToolsList
    {
        get
        {
            return _workingtools?
               .Split(new[] { ',', ';' },
StringSplitOptions.RemoveEmptyEntries)?
               .ToList()
               .Select(x => x = x.Trim())
               .ToList() ?? new List<string>();
        }
    }

    /// <summary>
    /// Represents the corrected PreviousWorkingSteps
    /// </summary>
    private string _previousSteps;
    [Column("Vorausgehende Arbeitsschritte")]
    [XmlIgnore]
    public string PreviousSteps
    {
        get
        {
            return _previousSteps;
        }
        set
        {
            if (string.IsNullOrEmpty(value))
            {
                _previousSteps = "";
            }
            else
            {
                _previousSteps = value;
                PreviousWorkingSteps = (_previousSteps ??
"")
                    .Split(',')
                    .ToList()
                    .Select(x => x = x.Trim())
                    .ToList();
```

```
                }
            }
        }

        [XmlArray("PreviousWorkingSteps")]
        [XmlArrayItem("PreviousWorkingStep")]
        public List<string> PreviousWorkingSteps { get; set; } = new List<string>();

        /// <summary>
        /// Represents all existing WorkingSteps
        /// </summary>
        [XmlIgnore]
        private static List<string> AllWorkingSteps = new List<string>();
        public static List<string> GetWorkingStepsDistinct()
        {
            return AllWorkingSteps
                .Distinct()
                .ToList();
        }

        /// <summary>
        /// Represents all existing Materials
        /// </summary>
        [XmlIgnore]
        private static List<string> AllMaterials = new List<string>();
        public static List<string> GetMaterialsDistinct()
        {
            return AllMaterials
                .Distinct()
                .ToList();
        }

        /// <summary>
        /// Represents all existing Resources
        /// </summary>
        [XmlIgnore]
        private static List<string> AllResources = new List<string>();
        public static List<string> GetResourcesDistinct()
        {
            return AllResources
                .Distinct()
                .ToList();
        }

        /// <summary>
        /// Represents all existing WorkingTools
        /// </summary>
        [XmlIgnore]
```

```
    private static List<string> AllWorkingTools = new List
    <string>();
    public static List<string> GetWorkingToolsDistinct()
    {
        return AllWorkingTools
            .Distinct()
            .ToList();
    }

    /// <summary>
    /// States whether the ProductionStep is done or not
    /// </summary>
    [XmlIgnore]
    public bool StepDone { get; set; } = false;
}


public class WorkPlace
{
    [Column("Arbeitsplatz (physisch)")]
    [XmlElement("PhysicalWorkPlace")]
    public string PhysicalWorkPlace { get; set; }

    /// <summary>
    /// Represents the physical work space (modular work
    place)
    /// </summary>
    private string _workStation;
    [Column("Arbeitsstationen (organisatorisch),
    Faehigkeiten des Arbeitsplatzes")]
    [XmlIgnore]
    public string Skills
    {
        get
        {
            return _workStation;
        }
        set
        {
            if (string.IsNullOrEmpty(value))
            {
                _workStation = "";
            }
            else
            {
                _workStation = value;
                WorkStations = (_workStation ?? "")
                    .Split(',')
                    .ToList()
                    .Select(x => x = x.Trim())
                    .ToList();

```

```
                if (WorkStations?.Count > 0)
                {
                    AllWorkStations.AddRange(WorkStations);
                }
            }
        }
    }

    /// <summary>
    /// Represents a list of handleable WorkStations (skills) at this WorkPlace
    /// </summary>
    [XmlArray("WorkStations")]
    [XmlArrayItem("WorkStation")]
    public List<string> WorkStations { get; set; } = new List<string>();


    [XmlIgnore]
    private static List<string> AllWorkStations = new List<string>();
    public static List<string> GetWorkStationsDistinct()
    {
        return AllWorkStations
            .Distinct()
            .ToList();
    }
}

public class WorkStation
{
    [Column("Arbeitsstation")]
    [XmlAttribute("ID")]
    public string ID { get; set; }

    [Column("Bezeichnung")]
    [XmlAttribute("Description")]
    public string Description { get; set; }
}

public class OrderItem
{
    [Column("Auftragsnummer")]
    [XmlAttribute("ID")]
    public string OrderID { get; set; }

    [Column("Bautyp")]
    [XmlAttribute("BuildingType")]
    public string BuildingType { get; set; }
}
```

C.2.3. Graph Generation

```
public class DataHandler
{
    /// <summary>
    /// Creates and returns the graph containing the Petrinet which will be drawn in the visualisation part.
    /// This method needs therefore a list of production steps (working steps for a specific type)
    /// and export options wich contain the styling for visualization.
    /// </summary>
    public Graph GeneratePetriNet(List<ProductionStep> productionSteps, ExportOptions exportOptions = null)
    {
        // recalculating all previous working steps of each given production step
        RecalculatePreviousSteps(productionSteps);
        // creating a new graph for the Petrinet
        Graph petrinet = new Graph("PetriNet");

        if (exportOptions == null)
        {
            exportOptions = new ExportOptions();
        }

        // gaining all transitions with following stati
        foreach (ProductionStep step in productionSteps)
        {
            Node transition = new Node(step.ProductionStepDescription);
            transition.Restyle(NodeType.Transition);

            // decision of node type for correct styling
            Node status = new Node(step.ProductionStepID);
            switch (step.ProductionStepType)
            {
                case ProductionStepType.None:
                    status.Restyle(NodeType.StartStop);
                    break;
                case ProductionStepType.Work:
                    status.Restyle(NodeType.Status);
                    break;
                case ProductionStepType.Check:
                    status.Restyle(NodeType.StatusCheck);
                    break;
            }

            petrinet.AddNode(status);

            if (step.ProductionStepType != ProductionStepType.None)
```

```
            {
                petrinet.AddNode(transition);
                petrinet.AddEdge(transition.Id, status.Id)
;
            }
        }

        // gaining all transitions with previous stati
        foreach (ProductionStep step in productionSteps)
        {
            foreach (string previousStep in step.
PreviousWorkingSteps)
            {
                if (!petrinet.Edges.Any(x => (x.Source ==
previousStep && x.Target == step.
ProductionStepDescription) || (x.Source == step.
ProductionStepDescription && x.Target == previousStep))
 &&
                    step.ProductionStepType !=
ProductionStepType.None)
                {
                    petrinet.AddEdge(previousStep, step.
ProductionStepDescription);
                }
                else if (step.ProductionStepType ==
ProductionStepType.None)
                {
                    petrinet.AddEdge(previousStep, step.
ProductionStepID);
                }
            }

            // decisions if Material/Resource/WorkingTool
are checked and should be added to the graph
            if (exportOptions.Material)
            {
                foreach (string materialname in step.
MaterialList)
                {
                    Node material = new Node(materialname)
;
                    material.Restyle(NodeType.
StatusMaterial);

                    petrinet.AddNode(material);
                    petrinet.AddEdge(materialname, step.
ProductionStepDescription);
                }
            }
            if (exportOptions.Resource)
            {
                foreach (string resourcename in step.
```

```
ResourcesList)
                {
                    Node resource = new Node(resourcename);
                    resource.Restyle(NodeType.StatusResource);

                    petrinet.AddNode(resource);
                    petrinet.AddEdge(resourcename, step.ProductionStepDescription);
                }
            }
            if (exportOptions.WorkingTool)
            {
                foreach (string workingtoolname in step.WorkingToolsList)
                {
                    Node workingtool = new Node(workingtoolname);
                    workingtool.Restyle(NodeType.StatusWorkingTool);

                    petrinet.AddNode(workingtool);
                    petrinet.AddEdge(workingtoolname, step.ProductionStepDescription);
                }
            }
        }

        return petrinet;
    }
}
```

C.2.4. Visualization

```
public partial class MainWindow : Window
{
    private void RedrawVisualization()
    {
        if (cbVisualizeData.SelectedItem == null)
        {
            MessageBox.Show("No redrawing due to no selection of visualizing data.", "No redrawign", MessageBoxButton.OK, MessageBoxImage.Information);
            return;
        }

        // getting all production steps of the filter building type (steps for "all" and steps for the selected "type")
        List<ProductionStep> filteredSteps = _excelData.ProductionSteps
            .Where(x => x.BuildingType.ToUpper() == "ALLE" || x.BuildingType == cbVisualizeData.SelectedItem.ToString())
            .ToList();

        // getting the Graph of the DataHandler
        petrinetViewer.BackColor = System.Drawing.Color.White;
        petrinetViewer.Graph = DataHandler.Instance.GeneratePetriNet(filteredSteps, _exportOptions);

        _exportOptions.HasChanged = false;
    }
}
```

C.2.5. Simulation

```
public class ActivityGetServed : Activity
{
    /// <summary>
    /// Overrides the state change at start. Server is not idle, and end event is triggered.
    /// </summary>
    override public void StateChangeStartEvent(DateTime time, ISimulationEngine simEngine)
    {
        double serviceTimeMinutes = ((SimulationModelQueuing)ParentControlUnit.ParentSimulationModel).ServiceTime;

        Server.IsIdle = false;
        simEngine.AddScheduledEvent(EndEvent, time + TimeSpan.FromMinutes(Distributions.Instance.Exponential(serviceTimeMinutes)));
    }

    /// <summary>
    /// Overrides the state change at end. Server is set idle again
    /// </summary>
    override public void StateChangeEndEvent(DateTime time, ISimulationEngine simEngine)
    {
        Server.IsIdle = true;

        Client.OrderItemWithCheckList.SetRequestAsDone(WorkingStation.ID);

        HashSet<string> nextrequests = Client.OrderItemWithCheckList.GetNextRequestsForQueue();

        if (nextrequests.Count == 0)
        {
            this.EndEvent.SequentialEvents.Add(new EventClientFinished(ParentControlUnit, Client));
            ParentControlUnit.RAELFinished.Add(new FinishedRequest("Finished", Client));
        }
        else
        {
            ParentControlUnit.AddRequest(new QueingRequest("GetServed", Client, time));
        }
    }
}
```

```

public class EventClientArrival : Event
{
    /// <summary>
    /// Overriden state change of the event. Request for service is made, next client arrival is scheduled
    /// </summary>
    /// <param name="time">Time the client arrives</param>
    /// <param name="simEngine">SimEngine responsible for simulation execution</param>
    protected override void StateChange(DateTime time, ISimulationEngine simEngine)
    {
        #region Using order list of ExcelData

        // next arrival is scheduled
        if ((ParentControlUnit as ControlUnitQueuingModel)._productionQueue.Count > 0)
        {
            EntityClient nextClient = new EntityClient((ParentControlUnit as ControlUnitQueuingModel)._productionQueue.Dequeue());
            EventClientArrival nextClientArrival = new EventClientArrival(ParentControlUnit, nextClient);

            double arrivalTimeMinutes = ((SimulationModelQueuing)ParentControlUnit.ParentSimulationModel).ArrivalTime;

            simEngine.AddScheduledEvent(nextClientArrival, time + TimeSpan.FromMinutes(Distributions.Instance.Exponential(arrivalTimeMinutes)));

            ParentControlUnit.AddRequest(new QueingRequest("GetServed", Client, time));
        }
        else
        {
            // adding the last Request
            ParentControlUnit.AddRequest(new QueingRequest("GetServed", Client, time));
        }
    }
}



public class ControlUnitQueuingModel : ControlUnit
{
    public Queue<OrderItemWithCheckList> _productionQueue = new Queue<OrderItemWithCheckList>();
    public ExcelData Exceldata;
```

```

    /// <summary>
    /// Number queues to be modeled
    /// </summary>
    public List<EntityQueue> Queues { get; set; }

    /// <summary>
    /// Number servers to be modeled
    /// </summary>
    public List<EntityWorkPlace> WorkPlaces { get; set; }

    /// <summary>
    /// Basic constructor, entities are added to model
    /// </summary>
    /// <param name="name">Name of control</param>
    /// <param name="parentControlUnit">Root control unit, null in this example</param>
    /// <param name="parentSimulationModel">Simulation model control belongs to</param>
    /// <param name="numberQueues">Number queues to be modeled</param>
    /// <param name="numberServers">Number servers to be modeled</param>
    public ControlUnitQueuingModel(string name, ControlUnit parentControlUnit, SimulationModel parentSimulationModel, ExcelData excelData) : base(name, parentControlUnit, parentSimulationModel)
    {
        Exceldata = excelData;
        Queues = new List<EntityQueue>();
        WorkPlaces = new List<EntityWorkPlace>();
        _productionQueue = excelData.GetFullProductionList();

        List<string> workplaces = WorkPlace.GetWorkStationsDistinct() ?? new List<string>();

        for (int i = 0; i < workplaces.Count; i++)
        {
            EntityQueue newQueue = new EntityQueue(workplaces[i]);
            AddEntity(newQueue);
            Queues.Add(newQueue);
        }
        EntityQueue finishedQueue = new EntityQueue("Finished");
        AddEntity(finishedQueue);
        Queues.Add(finishedQueue);

        for (int i = 0; i < excelData.WorkPlaces.Count; i++)
        {
```

```
            SkillSet skillset = new SkillSet(excelData.
WorkPlaces[i].WorkStations);
            EntityWorkPlace newWorkPlace = new
EntityWorkPlace(excelData.WorkPlaces[i].
PhysicalWorkPlace, skillset);
            AddEntity(newWorkPlace);
            WorkPlaces.Add(newWorkPlace);
        }
    }

    /// <summary>
    /// Arrival stream of clients is initialized
    /// </summary>
    /// <param name="startTime">Start time of simulation</
param>
    /// <param name="simEngine">End time of simulation</
param>
    protected override void CustomInitialize(DateTime
startTime, ISimulationEngine simEngine)
    {
        EntityClient nextClient = new EntityClient(
_productionQueue.Dequeue());
        EventClientArrival nextClientArrival = new
EventClientArrival(this, nextClient);

        double arrivalTimeMinutes = ((
SimulationModelQueuing)ParentSimulationModel).
ArrivalTime;

        simEngine.AddScheduledEvent(nextClientArrival,
startTime
            + TimeSpan.FromMinutes(Distributions.Instance.
Exponential(arrivalTimeMinutes)));
    }

    /// <summary>
    /// Custom rule set, basically incoming clients are
assinged to queues with minimum length
    /// and clients are selected from front of queues by
FIFO (so FIFO within a single queue and FIFO
    /// of queue fronts)
    /// </summary>
    /// <param name="time">Time rules are executed</param>
    /// <param name="simEngine">SimEngine responsible for
simulation execution</param>
    /// <returns></returns>
    protected override bool PerformCustomRules(DateTime
time, ISimulationEngine simEngine)
    {
        #region Handle finished clients

        if (RAELFinished.Count > 0)
```

```
        {
            EntityQueue finishedQueue = Queues
                .Where(x => x.Identifier == "Finished")
                .First();

            foreach (var item in RAELFinished.Cast<
FinishedRequest>().Select(x => x.Client).ToList())
            {
                finishedQueue.HoldedEntities.Add(item);
            }

            RAELFinished.Clear();
        }

        #endregion

        #region Put requests to right queue

        List<QueingRequest> getServedRequests = RAEL.Where
(p => p.Activity == "GetServed").Cast<QueingRequest>().
ToList();

        foreach (var servedRequest in getServedRequests)
        {
            HashSet<string> nextRequests = servedRequest.
Client.OrderItemWithCheckList.GetNextRequestsForQueue()
;
            IEnumerable<EntityQueue> fittingQueues =
Queues.Where(queue => nextRequests.Contains(queue.
Identifier));
            EntityQueue correctQueue = fittingQueues
                .Where(queue => queue.HoldedEntities.Count
 == fittingQueues.Min(x => x.HoldedEntities.Count))
                .FirstOrDefault();

            if (correctQueue != null)
            {
                correctQueue.HoldedEntities.Add(
servedRequest.Client);
                RemoveRequest(servedRequest);
            }
        }

        #endregion

        #region workplaces handle clients in right queues

        foreach (var workplace in WorkPlaces.Where(
workplace => workplace.IsIdle))
        {
            List<string> skills = workplace.SkillSet.
Skills
```

```
                    .Select(x => x.Skill)
                    .ToList();
                var bestQueues = Queues
                    .Where(queue => skills.Contains(queue.
    Identifier));
                int maxCount = bestQueues.Max(x => x.
    HoldedEntities.Count);

                if (maxCount == 0)
                {
                    continue;
                }
                else
                {
                    EntityQueue bestQueue = bestQueues
                        .Where(queue => queue.HoldedEntities.
    Count == maxCount)
                        .FirstOrDefault();

                    EntityClient client = (EntityClient)
    bestQueue.HoldedEntities.First();
                    bestQueue.HoldedEntities.RemoveAt(0);

                    WorkStation workstationToDo = Exceldata.
    WorkStations.Where(x => x.ID == bestQueue.Identifier).
    FirstOrDefault();

                    ActivityGetServed newService = new
    ActivityGetServed(this, client, workplace,
    workstationToDo);
                    newService.StartEvent.Trigger(time,
    simEngine);
                }
            }

            #endregion

            return false;
        }
}
```

C.3. XML Files

C.3.1. Hierarchical Structure of Excel Data

```xml
<?xml version="1.0" encoding="utf-8"?>
<Data xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<ProductionSteps>
  <ProductionStep ID="START" Brand="Steyr" BuildingType="Alle" ProductionStepType="None">
    <ProductionStepDescription>Start - Initialisierung</ProductionStepDescription>
    <MaterialList />
    <ResourcesList />
    <WorkingToolsList />
    <PreviousWorkingSteps />
  </ProductionStep>
  <ProductionStep ID="A01" Brand="Steyr" BuildingType="Alle" ProductionStepType="Work">
    <ProductionStepDescription>Motorblock bereitstellen</ProductionStepDescription>
    <WorkingStation>A</WorkingStation>
    <MaterialList>
      <Material>Motorblock</Material>
    </MaterialList>
    <ResourcesList />
    <WorkingToolsList />
    <PreviousWorkingSteps>
      <PreviousWorkingStep>START</PreviousWorkingStep>
    </PreviousWorkingSteps>
  </ProductionStep>
  <ProductionStep ID="A02" Brand="Steyr" BuildingType="Alle" ProductionStepType="Work">
    <ProductionStepDescription>Buechsen-O-Ringe einbauen</ProductionStepDescription>
    <WorkingStation>A</WorkingStation>
    <MaterialList>
      <Material>O-Ringe</Material>
    </MaterialList>
    <ResourcesList>
      <Resource>Motoroel</Resource>
    </ResourcesList>
    <WorkingToolsList />
```

```xml
    <PreviousWorkingSteps>
      <PreviousWorkingStep>A01</PreviousWorkingStep>
    </PreviousWorkingSteps>
  </ProductionStep>

  <ProductionStep ID="C01" Brand="Steyr" BuildingType="Alle" ProductionStepType="Work">
    <ProductionStepDescription>Pleuellager an Pleuelboecke anbringen</ProductionStepDescription>
    <WorkingStation>C</WorkingStation>
    <MaterialList>
      <Material>Pleuellager (Bockseite)</Material>
    </MaterialList>
    <ResourcesList>
      <Resource>Motoroel</Resource>
    </ResourcesList>
    <WorkingToolsList />
    <PreviousWorkingSteps>
      <PreviousWorkingStep>START</PreviousWorkingStep>
    </PreviousWorkingSteps>
  </ProductionStep>

  <ProductionStep ID="X01" Brand="Steyr" BuildingType="Alle" ProductionStepType="Check">
    <ProductionStepDescription>ueberpruefungsschritt - Befestigungskontrolle</ProductionStepDescription>
    <WorkingStation>X</WorkingStation>
    <MaterialList />
    <ResourcesList />
    <WorkingToolsList />
    <PreviousWorkingSteps>
      <PreviousWorkingStep>A01</PreviousWorkingStep>
      <PreviousWorkingStep>A02</PreviousWorkingStep>
      <PreviousWorkingStep>A03</PreviousWorkingStep>
      <PreviousWorkingStep>B01</PreviousWorkingStep>
      <PreviousWorkingStep>B02</PreviousWorkingStep>
      <PreviousWorkingStep>B03</PreviousWorkingStep>
      <PreviousWorkingStep>B04</PreviousWorkingStep>
      <PreviousWorkingStep>B05</PreviousWorkingStep>
      <PreviousWorkingStep>B06</PreviousWorkingStep>
      <PreviousWorkingStep>C01</PreviousWorkingStep>
      <PreviousWorkingStep>C02</PreviousWorkingStep>
      <PreviousWorkingStep>C03</PreviousWorkingStep>
```

```
      <PreviousWorkingStep>C04</PreviousWorkingStep>
      <PreviousWorkingStep>C05</PreviousWorkingStep>
      <PreviousWorkingStep>C06</PreviousWorkingStep>
      <PreviousWorkingStep>C07</PreviousWorkingStep>
    </PreviousWorkingSteps>
  </ProductionStep>

</ProductionSteps>
<WorkPlaces>
  <WorkPlace>
    <PhysicalWorkPlace>AP_2</PhysicalWorkPlace>
    <WorkStations>
      <WorkStation>A</WorkStation>
      <WorkStation>B</WorkStation>
      <WorkStation>C</WorkStation>
    </WorkStations>
  </WorkPlace>

  <WorkPlace>
    <PhysicalWorkPlace>AP_4</PhysicalWorkPlace>
    <WorkStations>
      <WorkStation>D</WorkStation>
    </WorkStations>
  </WorkPlace>
</WorkPlaces>
<WorkStations>
  <WorkStation ID="X" Description="ueberpruefung" />
  <WorkStation ID="A" Description="Motorblock" />
  <WorkStation ID="B" Description="Kolben" />
  <WorkStation ID="C" Description="Kurbelwelle" />
  <WorkStation ID="D" Description="Kurbelgehaeuse" />
  <WorkStation ID="E" Description="Nockenwelle" />
  <WorkStation ID="F" Description="Zylinderkoepfe" />
  <WorkStation ID="G" Description="Ventildeckel" />
  <WorkStation ID="H" Description="Kruemmer" />
  <WorkStation ID="I" Description="Finalisierung" />
</WorkStations>
<OrderItems>
  <OrderItem ID="BT_0001" BuildingType="T188" />
  <OrderItem ID="BT_0002" BuildingType="T190" />
  <OrderItem ID="BT_0003" BuildingType="T290" />
  <OrderItem ID="BT_0004" BuildingType="T190" />
  <OrderItem ID="BT_0005" BuildingType="T190" />
```

```
  <OrderItem ID="BT_0006" BuildingType="T190" />
  <OrderItem ID="BT_0007" BuildingType="T290" />
  <OrderItem ID="BT_0008" BuildingType="T188" />
  <OrderItem ID="BT_0009" BuildingType="T190" />
  <OrderItem ID="BT_0010" BuildingType="T190" />
</OrderItems>
</Data>
```

Bibliography

[ABD11] Bensmaine Abderrahmane, Lyes Benyoucef, and Mohammed Dahane. "Process plan generation in reconfigurable manufacturing systems using adapted NSGA-II and AMOSA." In: Aug. 2011, pp. 863–868 (cit. on p. 10).

[Bre92] Berndt Brehmer. "Dynamic decision making: Human control of complex systems." In: *Acta Psychologica, Volume 81, Issue 3*. 1992, pp. 211–241 (cit. on p. 18).

[CGL16] Zhu En Chay, Bing Feng Goh, and Maurice Ling. "PNet: A Python Library for Petri Net Modeling and Simulation." In: 2016, pp. 24–30 (cit. on p. 15).

[Dic20] Dictionary.Com. *German-English Dictionary*. 2020. URL: http://dictionary.com/ (cit. on p. ii).

[Eas] Easterbrook. *The difference between Verification and Validation*. URL: https://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation (visited on 12/21/2020) (cit. on p. 56).

[EET20] Claudia Ermel, Karsten Ehrig, and Gabriele Taentzer. *TU Berlin*. 2020. URL: https://www.user.tu-berlin.de/lieske/tfs/projekte/petrieditor/details.htm (cit. on p. 16).

[Fur+14] Nikolaus Furian et al. "A conceptual modeling framework for discrete event simulation using hierarchical control structures." In: *In Proceedings 28th European Conference on Modeling and Simulation pp 206-2013*. 2014, pp. 206–213 (cit. on pp. 16, 17, 25).

[Fur14] Nikolaus Furian. "HCCM - A control world view for health care discrete event simulation." In: *In Proceedings 28th European Conference on Modeling and Simulation pp 206-2013*. 2014, pp. 206–213 (cit. on p. 53).

[Gra12] Victor Granados. "Modelling and optimization of flexible manufacturing systems." PhD thesis. July 2012 (cit. on p. 11).

[Hyu92] J. Hyun. "A Unifying Framework for Manufacturing Flexibility." In: 1992 (cit. on p. 11).

[Lin20] Linguee. *German-English Dictionary*. 2020. URL: https://www.linguee.de/ (cit. on p. ii).

[Löd16] Hermann Lödding. Vol. 4. Springer, 2016, p. 124 (cit. on pp. 11, 12).

[LTG05] A. Lara, G. Trujano, and A. Garcia-Garnica. "Modular production and technological up-grading in the automotive industry: a case study." In: *Int. J. Automotive Technology and Management, Vol. 5, No. 2*. 2005, pp. 199–215 (cit. on p. 9).

[Mica] Microsoft. *Microsoft MSAGL Graphic Viewer*. URL: https://github.com/microsoft/automatic-graph-layout3 (visited on 11/18/2020) (cit. on p. 50).

[Micb] Microsoft. *Microsoft NPOI Excel Mapper*. URL: https://github.com/donnytian/Npoi.Mapper (visited on 11/18/2020) (cit. on p. 48).

[Pan+08] J. Pandremenos et al. "Modularity concepts for the automotive industry: A critical review." In: *CIRP Journal of Manufacturing Science and Technology 1 (2009)*. 2008, pp. 148–152 (cit. on p. 9).

[Pom] Franck Pommereau. *Snakes*. URL: https://snakes.ibisc.univ-evry.fr (visited on 11/19/2020) (cit. on p. 35).

[Pom15] *SNAKES: a flexible high-level Petri nets library*. Springer, 2015 (cit. on p. 15).

[PR08] Carl Adam Petri and Wolfgang Reisig. "Petri net." In: 2008, p. 6477 (cit. on p. 15).

[RB97] G.G. Rogers and L. Bottaci. *Modular production systems: a new manufacturing paradigm*. 1997. URL: https://link.springer.com/article/10.1023/A:1018560922013 (visited on 11/10/2020) (cit. on p. 9).

[Rob08] Stewart Robinson. "Conceptual modelling for simulation Part I: Definition and requirements." In: *Journal of the Operational Research Society* 59 (2008), pp. 278–290 (cit. on p. 25).

[Sup] Supertek. URL: https://en.supertek.de/products-and-services/machine-and-plant-engineering/industry-4.0 (visited on 12/01/2020) (cit. on p. 3).

[Swa00] Paul M. Swamidass. "Manufacturing Flexibility." In: *Innovations in Competitive Manufacturing*. Springer, 2000, pp. 117–136 (cit. on p. 11).