Rheinisch Westfälische Technische Hochschule Aachen
Lehrstuhl für Software Engineering

A Model-Driven Development Model for Component Structures and Part Lists

Bachelorarbeit

von

Choi, Joel

1. Prüfer: Prof. Dr. Bernhard Rumpe

2. Prüfer: Prof. Dr. Stefan Kowalewski

1. Betreuer: Dr. Judith Michael

2. Betreuer: Louis Wachtmeister

Diese Arbeit wurde vorgelegt am Lehrstuhl für Software Engineering

Aachen, den October 1, 2021

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Abstract

Bill of Materials are component lists of a product used to document information. There are different kinds of Bill-of-Materials that are used to accurately display the composition of the final product during different kinds of stages in its life cycle. In many instances, these documents are handled arbitrarily, without proper consideration of the formatting or the modeling of the data. This leads to problems during the inspection of the data, as users regard documents of different forms, which leads to an increased difficulty of understanding. Moreover, the correct exchange of the bills, without information loss, or the transmission of outdated or wrong data, is imperative during the life cycle of a product. However, in many instances, the document transfer is done manually, which is especially prone to the errors listed above.

This thesis aims to solve the two problems by introducing a unified modeling approach of the Bill-of-Materials using UML Diagrams and tables, and a tool which automates the transfer and creation of the Bill-of-Materials. The automation functions provided by the tool guarantee the correctness of the transferred data and the modeling accurately displays all information of the Bill-of-Materials in a unified manner.

This solution holds value for businesses, as the unified modeling of data provides clarity without creating further workload, which is bypassed by the tool, which automates most synchronisation processes. The time saved, by using the tool is another advantage of the solution. However, the guarantee of the correctness of the data, is what enables the companies to eliminate any potential financial deficits and delays in production.

The tool provided in this thesis therefore is a solution that provides: a guarantee of correctness, clarity, flexibility, and a gain of time and resources.

Statement of Task

When creating a car or any other product, there are challenges that arise during the process of development, in the various stages from planning through the production itself until the sale and after service. This stems from the multitude of aspects and requirements which the product needs to fulfill. In most cases the customer has some requirements for the product and its architecture in mind. Additionally, for more complex products such as cars, there is a multitude of different departments in different fields that are all involved in the process that are all responsible for some tasks in specific phases in the development. Therefore, way information about the product and the production process is stored and shared is of utmost importance. A more efficient and comprehensible form and presentation for the information results in a better communication between the client and the producer as well as between the different departments involved in the process. The representation of the information on the product and system development is most commonly displayed by Bill of Materials (BOM).

BOMs are documents that store information about the product and which parts it consists of. This document is usually stored in a tabular form, such as in an Excel file. When it comes to BOMs, there are many different types, specifically tailored towards the needs of the department currently in possession of the information. In this thesis we will work with the Engineering BOM (E-BOM) and the Manufacturing BOM (M-BOM) in particular. The E-BOM displays the functional construction of the product whilst the M-BOM focuses additionally on the manufacturing aspect. Some parts which have not been listed on the E-BOM before could be added, such as connectors like glue, bolts, and cables. Furthermore, the M-BOM could also group certain parts which have been connected to others into a subpart, which did not exist in the E-BOM before.

The differences in both lists are substantial and the synchronisation between the lists poses a challenging task. Therefore, this thesis aims to provide a possible solution to ensure a fluid interworking between these two structures. The solution we provide is a model-based process based on UML class-, object- and activity diagrams to model and derive the E-BOM from a technical system architecture and show its practical viability with an implementation. To explain the aspect of the specific implementation in more detail: the first objective is the implementation of an automated tool that creates an E-BOM from a class diagram. After the user then fills the missing attribute values in the E-BOM, again an automated process creates a new object diagram from the given information. If some information is added or changed, the tool will be able to update the diagrams or the table respectively.

As another and final feature, a solution for the before mentioned synchronisation problem of the two lists is provided. Using the derived diagrams of the E-BOM and an additional activity diagram of the Bill of process (BOP) we provide a tool that automatically derives the M-BOM using both diagrams.

Conclusively, this thesis provides a practical and automated solution for the problems of synchronisation of the E-BOM and M-BOM using a UML model-based approach, which

will be of beneficial when it comes to information storage for different domains involved in the production process.

Contents

Chapter 1

Introduction

Most companies base their monetization and therefore, their livelihood, on the sale of a product, which they plan, develop and consequently produce. As this process requires many different parties, responsible for different phases in the life cycle of the product, the communication between them is a vital. Especially for manufacturing companies of larger scale, which produce complex products, such as cars, the data management aspect is of utmost importance. This includes, the storing of data, the representation of the data, the transfer of data, and many more issues [ERZ14].

Component lists, which have all the information on parts comprising the final product, are valuable data sets, often exchanged between different parties. As many instances are involved, their requirements for the data, in terms of representation and form differs immensely. A software Engineer does not share the same opinion, on which information aspects should be included as that of an engineer or an electrician. Different departments are responsible for different tasks and require specifically formatted data comprising the information necessary for their role [Lin16]. Most companies therefore opt to use unique data sheets, which contain the data required by the instance, developing the sheets [Lin16]. Additionally, proper updating and transmission of these component lists is essential, so that each instance is well informed about the current standings in other departments.

Specifically, during product planning and production itself, the transmission of correct data is important, as here the technical system architectures, the demands posed by the customer, the product design and function, and many more properties are decided and implemented. Therefore, the focus of this thesis lies with all data transmission between relevant parties in this scope of the product life cycle.

In these phases especially, errors occur during the transmission of data. These occurrences are based on two significant weaknesses of data management in production companies. Firstly, the data sheets do not possess a unified representation. In many cases, the data is represented in different formats, depending on the needs of the user. Secondly, the transfer and maintenance of the data sheets are done manually, by the person responsible for creating the documents. With lapses in concentration and other factors, humans tend to make mistakes in tedious tasks. Particularly, the scale of component lists in car manufacturing companies is of considerable size, which means the manual editing of the documents is even more an unsuited, deprecated method. Another alternative needs to be introduced [CLL97].

Even though research has been done in this field and some solutions have been introduced [HW91, CLL97, XXH08], many companies still struggle with the implementation of these changes. A solution that is easily applicable to the already existing data management system would be better for businesses, that employ a separation of the component lists. The relevance of such a solution is further emphasized, by the growing digitalisation of data management in companies, according to the rapidly changing product demands [STT+18]. Thus, a software-based solution, is not only relevant in the current climate, but also a solution based on future trends.

The objectives posed in this thesis are the solution to the named problems, by a system based on software, and computer engineering based methods. A guarantee for the correctness of the component lists holds immense value for the companies, and is the main objective here. Clarity of data representation is another aspect which is taken into consideration. Being able to display the data in many formats in a clear manner, would increase understandability and clarity. However, the clarity and flexibility should not come at the cost of ease of usage. The user should not be subject to additional workload, but rather have the system take most of the tasks, which the user would otherwise perform.

The structure of the thesis is as follows: First, the fundamental information needed to understand the subsequent chapters is introduced. Different types of document used during the product life cycle management are elaborated. UML diagrams and related works are introduced in the same chapter. After that, the methodology behind the solution introduced in this thesis, is listed. This will be based on a running example, that will be accompanying the reader throughout. Consequently, the concrete implementation of the methodology is illustrated. During the explanation, a demonstration of the tool in the perspective of a user is shown incidentally, to give the reader a better understanding of how the tool simplifies tasks. Finally, potential future works are introduced. Some weaknesses of the current version of the tool are pointed out, with the specific information on how these weaknesses can be overcome. A conclusion follows, with a summary of the findings and the value, which they pose for businesses.

Chapter 2

Fundamentals

Fundamental concepts and expressions needed for the topics handled in this thesis are introduced. Firstly, the different ways, in which information is handled in a production process will be established. Here, different document types, such as the E-BOM, M-BOM and the BOP, are the point of interest. Next, three different types of UML diagrams, their basic structures, and components will be presented. For each diagram type introduced, a language that can model the diagram for further automation, will be displayed.

2.1 Product Lifecycle Management

In a production process of a product, information about the composition and other information specific to product needs to be stored in an efficient, and easily accessible method. Product-data-management (PDM) and the product-life-cycle-management (PLM) are the result of incorporating IT-solutions to the given problem definition [ERZ14].

PLM is the process of managing a product's life cycle to integrate people, processes, tools, and business systems, to make the needed information available at the right time and right context [ERZ14]. It thereby facilitates a better experience in collaborative creation, management, product definition, and process operation information from market concept through the design and manufacturing phase, until service and eventual product retirement [Y.12]. PDM is often wrongly used synonymously to PLM. Whilst PLM focuses on the lifecycle of a product, PDM is a system which saves all information related to a product. Both have its distinct advantages and uses. For example, selling spare parts greatly benefit from the data provided by the PDM. On the other hand, information retrieval during the production process is derived mostly from the PLM [PLV14].

The tool provided by the PLM specifically, enables different departments of a production company to work in a more efficient way and therefore in turn helps realising goals set by the company. Those goals range from reducing costs during production, increasing revenues, improved quality control, maximising values of product portfolios, and the value of the company itself. Hereby, the satisfaction of both customers and shareholders is maximised [Y.12].

Figure 2.1 displays the main lifecycles included in the production process. First the requirements for the products, posed by the customer or company itself, are specified. Soon after, concepts and the logical architecture are set, concluding the planning phase of the

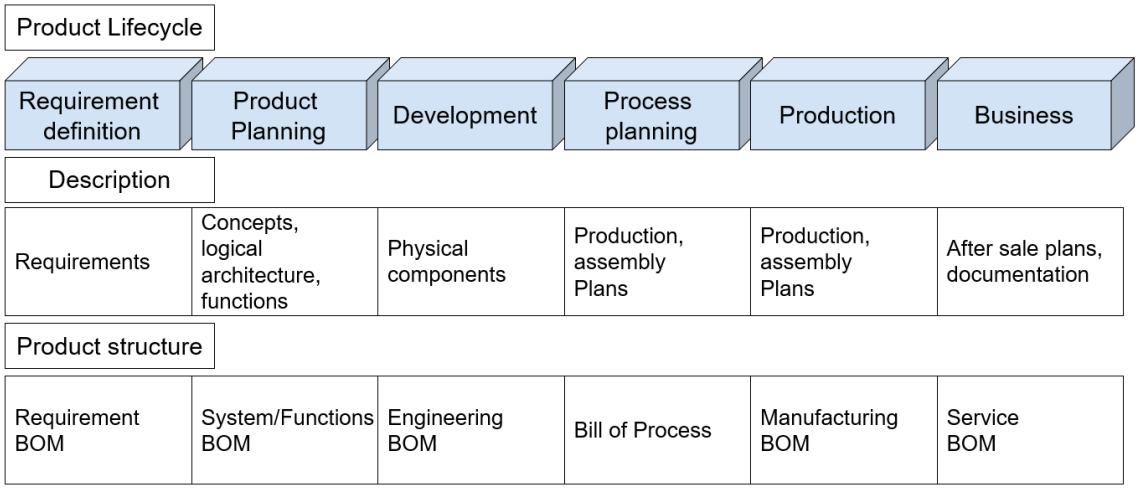


Figure 2.1: Illustration of all production phases in a lifecycle including its description and product structure, which shows the name of the data sheet storing the required information for each phase [ERZ14].

production. The next cycle is the development, in which the physical components making up the final product are revised. Production process planning and the production of the product itself contain plans for assembly and purchasing, and the models [ERZ14]. During the phases mentioned above, PLM uses different data sheets, representing information about the product structure in the current phase, all varying in form and prioritisation of information. In the figure 2.1 it is illustrated in the bottom most row.

The data sheets used for the phases are called bill-of-materials (BOM). It is a listing of all the components and the quantities that comprise the final product. Strictly thinking from a data integrity standpoint, a single BOM type would be preferable, as data management would be easier. However, in practice, one type of BOM does not usually suffice for an enterprise to have its production process be as efficient as possible [CLL97].

Assessing costs of the entire production process, pre-planning and controlling systems also, greatly benefit from a bill-of-material, under the conjecture that it provides only the necessary information for each instance. If this is not the case and all instances receive the same data sheet with the same characteristics, some will be receiving a surplus or a shortage of information. Furthermore, according to [HW91], the market shifts continuously towards the trend of being consumer focused, which results in a necessity for companies to be flexible in terms of product variation, as nowadays customer demand shifts rapidly. Again, it becomes apparent that a single type of BOM would not suffice to fulfil its purpose.

In the following, two types of BOMs, the E-BOM and the M-BOM, and the BOP are introduced.

2.1.1 Engineering Bill-of-Material

The E-BOM, also commonly referred to as the “as-designed” BOM [Lin16], contains the information relevant for the product development phase. It usually includes information on the current state of the component information including other related documents and additional clearance conditions. The design engineer assesses the parts and quantities

of components needed during the design process, and uses the E-BOM to formalise the findings. After the designer has completed the specification the E-BOM represents a the product structure, which is viewed as a series of hierarchical subsystems, which all together form the product [Lin16].

During the design stage the E-BOM provides enough information, but when the assembly planning and assembly stage is reached, other criteria must be considered.

2.1.2 Manufacturing Bill-of-Material

The M-BOM comprises information relevant to the production phase of the product life cycle. More specifically, it contains all essential components required, to be able to assemble the final product. In contrast to the E-BOM which contains data on the designed product, the M-BOM displays, all necessary components to assemble and deliver the final product. Therefore, the assembly sequence is taken into consideration when creating the M-BOM. This constitutes the addition of components during the assembly (e.g. tools, connectors, manuals), which were not included in prior bills [Lin16]. The structure of the M-BOM is built as a series of hierarchical assembly groups which illustrates the process of assembly of the product [CLL97].

Due to the differences in these two specifications, proper synchronisation between them for different departments participating in the production, has been proven to be a challenging task. Especially, because the contents of the M-BOM tend to shift during production without having direct impacts on the E-BOM [Lin16].

Having established the differences between the E-BOM and the M-BOM, another essential document in the production planning process, the bill-of-processes is introduced in the following section.

2.1.3 Bill-of-Process

So far, the two BOMs that were introduced stored information on which materials and components were required for production. However, this information does not fully display all information necessary for the production process. The "what" question has been answered, but the "how" question remains unanswered. The Bill-of-Process (BOP) aims to fill that gap. It contains detailed plans on the manufacturing process of the product. The machinery, plant resources, configurations, tools, and instructions needed during the production process is included in the BOP [Lit12].

2.2 UML Diagrams

Usually these documents are represented by tables, but in this thesis the approach of using specific UML diagrams to model these documents is adopted. In order to parse and use these diagrams in new applications, specific languages provided in the MontiCore framework for each diagram type are employed.

The Unified Modeling Language (UML) is a specification for the best practices, established during a wide span of time, in the use of modelling languages [MG17]. The language

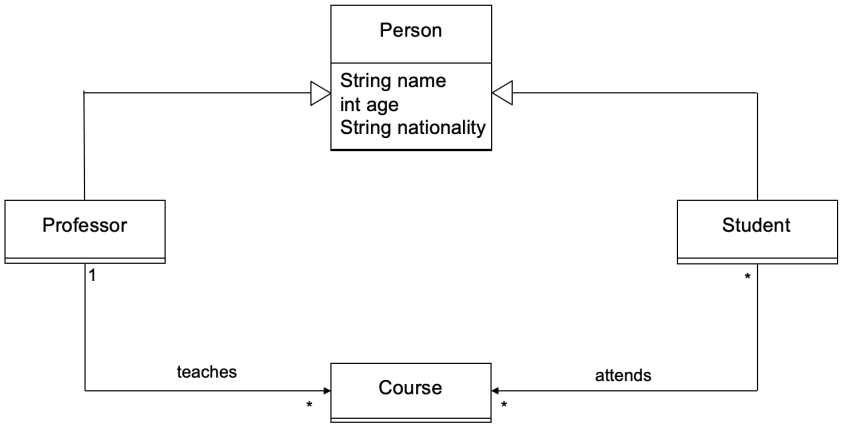


Figure 2.2: Illustration of all production phases in a lifecycle including its description and product structure, which shows the name of the data sheet storing the required information for each phase [ERZ14].

enables the presentation of software systems in various ways, within a single framework using object-oriented concepts. In UML, a model is represented graphically in the form of diagrams. It shows the reality part depicted by the model. Generally, the diagrams display the structure, or the behaviour of the system without explicitly describing its concrete implementation. Currently there are 14 different types of diagrams, which the UML offers [SSHK15]. Furthermore, [SSHK15] categorises them into two categories, first the category containing all structure diagrams, and the category with the behaviour diagrams. In this thesis 3 of them will be further elaborated upon.

2.2.1 Class Diagram

The concepts of the class diagram are derived from conceptual data modeling and object-oriented software development. The aim is to specify the data structures and object structures of a system, using the concepts. The concepts of class, generalisation, and association are the most prevalent ones. The class diagram belongs to the category of structure diagrams [SSHK15].

In the figure 2.2 for example, we see the class Person, which has both the Professor, and the Student class as subclasses, as the professor and the students both share some properties, such as the attributes age, name, or nationality. This is an example for the generalisation relationship. The class Course is associated with the Professor and the Students, because the professor teaches in the course and the student attends the course. As can be seen in the figure 2.2, associations also show the cardinalities between the two classes involved. A course is usually lead by one professor, whilst a professor can teach more than one course. Students attend various courses and multiple students attend a course.

CD4Analysis

The cd4anlaysis language is a language used to model class diagrams. It is a language provided by a MontiCore project, which works on associations, compositions, qualifiers, and more. Code generation, keeping consistency of the model intact, is possible. Its main

```
1  Classdiagramm Person {                                    CD4A
2    public class person{
3      public String name;
4      public int age;
5      public String nationality;
6    }
7    public class professor extends person{}
8
9    public class student extends person{}
10
11   public class course{}
12
13   association [1]professor -> course[*];
14
15   association [*]student ->course[*];
16 }
```

Figure 2.3: Concrete implementation of the prior shown class diagram using the cd4analysis language. The implementation can be used for further applications, by using various methods offered in cd4analysis.

usages can be listed as follows. Firstly, it can be used for code and analysis modelling. The modelling of structures of the system context, data structures of the system as well as the implementation itself are some examples to name a few. Another use is generating code, data tables, transport functions and more according to ones needs. Moving on, using the class diagrams as transitional models, as a means of mapping DSLs into an object-oriented target language, like Java or C++, is also thinkable. Lastly it is possible to use the class diagrams as results from any other generation or analysis process [www21].

The figure 2.3 shows the implementation for the class diagram in figure 2.2, in the cd4analysis language. Its concrete implementation follows the package and import statements. As displayed in the example, the semantics of the language is self-descriptive, especially for frequent users of object-oriented languages such as Java or C++. Keywords such as class diagram, class, composition, association are used to initialise the respective components.

2.2.2 Object Diagram

Another diagram included in the family of UML diagrams is the object diagram. It is a structural diagram, that displays a modelled system at a specific point in time, closely related to the class diagram [SSHK15]. The close relation with the class diagrams comes from the fact, that in this case an instantiation of the classes and its attributes is used.

The object is also in a box separated into two parts, as shown in figure 2.4. Contrary to the class diagram however, in the upper part the name of the instance of a class precedes the class name itself. The two are separated via a colon. As for the lower part, again the attributes now carry values as opposed to just the type and the name of the attribute. As mentioned before, the diagram represents a possible instance of the prior shown class

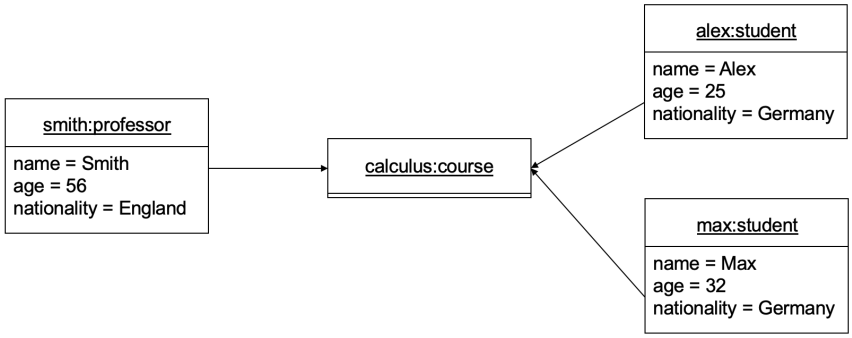


Figure 2.4: An example object diagram, which is a possible instantiation of the prior class diagram. Unlike the class diagram, concrete values for the attributes are now in place.

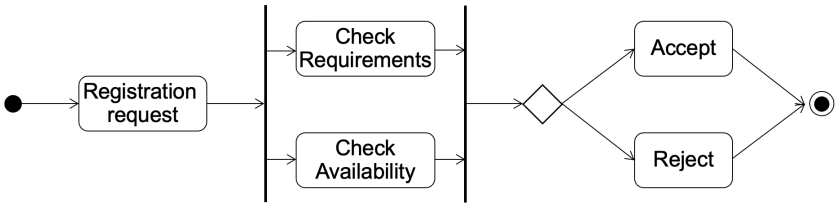


Figure 2.5: Simplified course registration process, from the perspective of a student. After registration, the student will receive an answer depending on the checks performed prior.

diagram. Here, a professor teaches the calculus course which two students from Germany take.

2.2.3 Activity Diagram

The activity diagram belongs, unlike the two above, to the behaviour diagrams. As the name suggests, behaviour diagrams enables the modeling of the behaviour of a system in detail. More specifically, behaviours reflect direct consequences of an action of at least one object, and the changes of its states over time. Both actions of a single object or an interaction between multiple objects can specify behaviour [SSHK15]. Activity diagrams are used to model processes, ranging from business processes to software processes. According to [SSHK15], there are no kinds of limitations as to what kind of processes can be modeled. Control flow mechanisms and data flow mechanisms are offered by activity diagrams, via different kinds of nodes, so that actions which make up an activity can be coordinated.

In the figure 2.5 the activity diagram for a registration process of a student for a course is shown in a simplified manner. After the registration is received, another instance checks the validity of the request. Two checks are performed simultaneously, which is modeled by the straight bars between the two activities. It is checked whether the student is eligible to participate in the course, and whether the capacity of the course is not yet full. After the checks it is decided whether the registration is successful or not. The diamond displays a choice node, where only one activity is performed, depending on different parameters, in this case the result of both tests.

```
activity Registration {                                          ADlanguage

    action request;
    action requirementscheck;
    action capacitycheck;
    action accept;
    action reject;

    fork forkNode;
    join joinNode;
    decision decision;

    transition initial -> request;
    transition request -> forkNode;
    transition forkNode ->  requirementscheck;
    transition forkNode -> capacitycheck;
    transition requirementscheck -> joinNode;
    transition capacitycheck -> joinNode;
    transition joinNode -> decision;
    transition decision -> accept;
    transition decision -> reject;
    transition accept -> final;
    transition reject -> final;

}
```

Figure 2.6: The activity diagram shown before implemented using the adlanguage. Again, using features of the adlanguage, other tasks can be performed on the data.

ADLanguage

The adlanguage is like the cd4analysis language developed in a MontiCore project and it aims to provide an implementation for activity diagrams. In figure 2.6, the activity diagram from before is implemented. Using the keyword action, the actions in the diagram are initialised. Specific control flow nodes are labeled with other keywords, such as fork, join, and decision. The transition keyword is used to implement the transitions between the actions and nodes.

2.3 Related Works

Aside from the findings in this thesis, there has already been research on the topic of transforming the E-BOM into the M-BOM using the BOP. Works from other authors related to the topic is introduced in the following.

First, [Lin16] explains PLM and the role of different BOMs and touches upon the historic development of data management in production. The importance of BOMs in a production process is established. The aim of [Lin16] is to educate the reader of various current topics such as PLM, and its phases.

In [XXH08], the problems of E-BOM and M-BOM synchronisation is acknowledged, and a new method to transform E-BOM to M-BOM based on BOP automatically is introduced. Concrete definitions of the E-BOM, M-BOM, and the BOP as quadruples are given. Next, seven transformation rules are introduced that introduced to solve three problems in the transformation process. The product structure transformation, BOP modification and

product lead-time recount. Based on these transformation rules a transformation algorithm is introduced. The algorithm is also tested on a practical case and they conclude by, stating that their algorithm enables rapid, automatic transformation of E-BOM to M-BOM using BOP, making production plans more effective and actual [XXH08].

Moving on, in [XXH08] other techniques presented by others, to solve the issue of M-BOM generation are illustrated, and their solutions presented: In [OyJ02] a three-level structure is used, to support integration of information flow between material requirements planning (MRP) and applications of PDM. The structure includes model development, analysis, system implementation, and an approach applying the concepts of semantic similarities to find common objects used in different business applications [XXH08]. Another method presented in [XLHM02] solves the BOM transformation using feature identification, which keeps the information in the different BOMs integral, accurate and consistent. The introduction to a new kind of BOM, the Process Planning BOM (PP-BOM), and the definition of various terms, such as middle, virtual, and successive parts, in addition to their mapping functions both constitute a new mapping model of BOM transformation and BOM transformation rules, in accordance with the part classification [XXH08].

Instead of introducing a solution for M-BOM generation, some try to solve the problem by introducing a unified BOM, that can serve as a singular BOM for all activities during production. Tozawa addresses in [TY09] the issues concerning the usage of multiple BOMs. It introduces an integration of multiple BOMs, so that it can cover communication needs in a global environment. The integrated BOM provides different views for different needs of departments. Toyota is named as the model company where BOM is used as a communication tool. In [ZCXZ07], another unified BOM solution is introduced. Here the BOM derivation is based on STEP/XML standard. STEP methodology is used to bypass the data exchange issue of heterogeneous system, whilst XML provides the necessary functions to deal with distributed system in web environment. Via a case study, the reliability of the implementation methods and the feasibility for utilisation on the platforms, are shown. Hegge [HW91] introduces a generic BOM, where the BOM structure for all variants of a product family is specified only once. This leads to a highly transparent BOM structure and enables the avoidance of redundancy [HW91].

Chapter 3

Method

After having covered the fundamentals necessary for full comprehension, the methodology behind the different implementations of tools, which were developed to provide help in the product development, process planning, and production phases are introduced and further illustrated by introducing a running example.

3.1 Running example

Throughout this chapter a running example accompanies the introductions of each feature presented. It intends to improve the understanding, and the display of the practical usage of the tools introduced.

The examples are based on a hypothetical business involved in the automotive industry, going through the steps of product development, in which the E-BOM is first formalised and adjusted, the process planning using a BOP, and the production phase, where the M-BOM is generated. All the documents generated during the process are based on the same running example, which is the simplified production of a car. The examples do not accurately reflect the data needed for an actual car, nevertheless it serves as a purpose to, firstly see how data on the product is transmitted throughout the various stages of the PLM, and secondly give an overview on the basic structure of each document type in a semi-realistic, yet consistent setting. The running example here consistently considers three attributes for the components, because it aims to give a consistent view on how the representations are transformed and created to increase clarity. Consistent attributes lead to a better understanding of the processes, however it is important to consider, that one of the main advantage of the use of class diagrams in contrast to tables, is the ability to assign separate attributes to each class. This is not possible in tables, as attributes are usually grouped in the header row, and due to the nature of a table, are the same for every component. This concept is explained further further on.

One of the problems of data management is the lack of a unified representation of the data. In this thesis a unified representation of BOMs and BOPs are introduced. The dual representation of a BOM and a BOP via a tabular representation and a class, object, or activity diagram is the modeling decision made in this thesis. Still, it does not create any further workload as automated synchronisation and generation steps are introduced consequently.

3.2 Engineering Bill of Materials

The E-BOM provides information on which components and its quantities are deemed necessary by the product designer. Here the example company is in the process of producing a new car and the design engineer finalised the E-BOM as displayed in the table 3.1. As visible in the table, the car consists of several parts with further information, such as weight, an Id, which is a unique identifier for each part, and the weight of the part. This list is not an accurate representation of what parts and attributes necessary in real instances, but rather for clarity sake, a simplified reduced version. In practice a multitude of other attributes relevant to the production and a higher number of different components are realistic. This underlines the fact, that a manual or partly-automated synchronisation of the lists is, due to its scale, time consuming and error prone.

In most E-BOMs a hierarchy between the parts exists, where main and subcomponents are differentiated. The hierarchical division of parts is based on the engineering domain. The M-BOM is introduced later and contrary to the E-BOM it has its focus on the product assembly. During the product assembly phase however, parts are added or modified, which means the M-BOM includes components not listed in the E-BOM.

Part	ID	weight	quantity
Axle	122	14,0 kg	2
Frame	123	125,7 kg	1
Engine	131	158,0 kg	1
Wheel	132	14,5 kg	4
Hood	111	35,0 kg	1
Bumper	121	50,0 kg	2
Exhaust	134	19,5 kg	1
Door	133	41,0 kg	4

Table 3.1: An example E-BOM represented in a table. Displays 4 different attributes of the components needed. Basis for all future modeling of the data in different formats.

The E-BOM itself does contain valuable information for the production process and should be easily accessible, understandable, and maintainable by all, who require access to the information. Therefore, we introduce the modeling of the E-BOM via a class diagram and an automated process of synchronisation.

3.2.1 Tabular representation of E-BOM

In practical use, the E-BOM is mostly stored in the form of tables. In table 3.1 an example of such an E-BOM table is displayed. The top most row displays the different attributes of the parts, and the left most column provides information on the part names. the concrete values are then inserted in each respective row. The example is, for clarities sake, small in size and does not contain many attributes. In reality, especially during the production of products in the automotive industry, these tables are considerably larger with more attributes and an additional hierarchy for the components, illustrated by tabular spaces in the part column. For example, weights are often split into the maximum allowed weight, the expected weight, and the actual weight. The engine consists of other subparts, such as the cylinder head, oil filters, water pumps and many more.

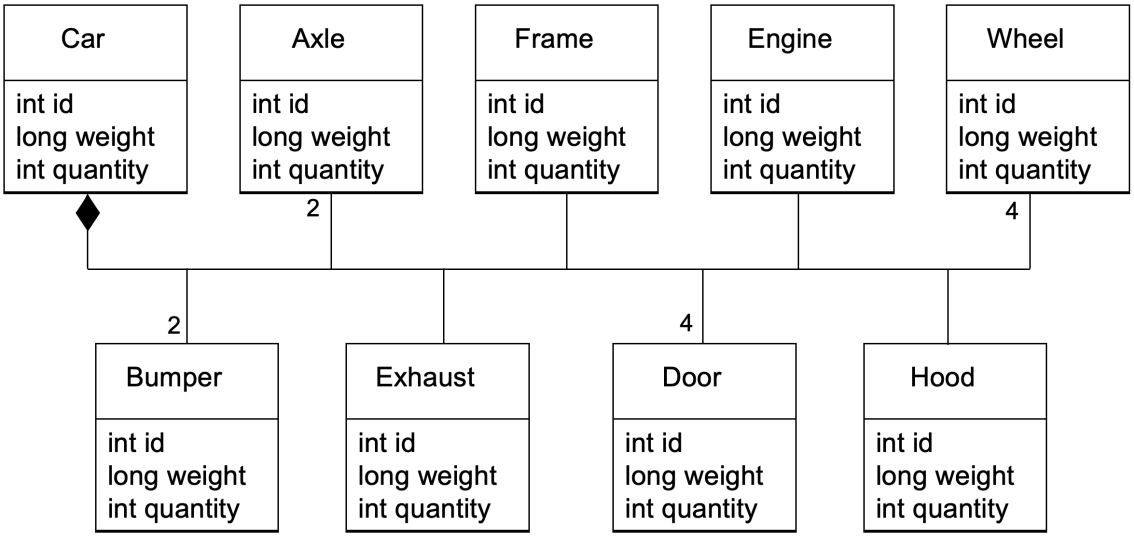


Figure 3.1: The E-BOM modeled in a class diagram.

The complexity of these lists justifies looking for applications and tools to manage the data sets efficiently. The first aspect to take into consideration is the software used to model the information. Choosing the appropriate application is based on the criteria of being able to efficiently and easily generate the table. From the perspective of the user, the usage should be intuitive and not labour intensive.

The usage of Excel to display the data, does meet the requirements of being easily accessible and understandable for its users and holds unique advantages. That is why in future parts E-BOMs are primarily saved in Excel spreadsheets. Further explanation on this choice is elaborated upon in the following chapter.

Next, the methodology of an implementation of the data set in form of a class diagram is introduced.

3.2.2 Representation of E-BOM by a Class Diagram

To make the E-BOM easily understandable, a class diagram representation is applied as a way of modeling the data. This representation is especially potent, as the object oriented nature of the BOM is perfectly suited for a modeling through a class diagram. The main class is the car itself and parts comprising the car are modeled by classes connected to the car with associations or compositions. Further information on the parts are stored as attributes of classes.

Figure 3.1 shows the class diagram of the E-BOM introduced earlier. The diagram does not only give a graphical representation, adding clarity to the E-BOM, but also can be implemented using the cd4analysis language. An implementation of the E-BOM opens up the possibility of using the data for different applications, one of which is the automated generation of the diagrams which is introduced later. Interestingly, the class diagram can illustrate the quantities of each part needed in two distinct ways: by the attribute quantity and by the cardinality of the associations between the classes. Likewise, other attributes could be displayed by other components of a class diagram, instead of just adding attributes to the classes. For example, the product quality of each part, ranging

```
1  classdiagram Car {                                    cd4analysis
2    public class Car {
3      public int ID;
4      public int weight;
5      public int quantity;
6    }
7
8    public class ColoredBody {
9      public int ID;
10     public int weight;
11     public int quantity;
12   }
13
14   public class Paint {
15     public int ID;
16     public int weight;
17     public int quantity;
18   }
19 …
```

Figure 3.2: Snippet of implementation of class diagram using the cd4analysis language. Automated synchronisation between table and class diagram is supported.

from bad to excellent could be represented by an enum. The cd4analysis language however, as of now, does not cover cardinalities other than one, more than one, or an arbitrary amount. Thus, specific numbers which are necessary in this case, cannot be implemented, meaning a representation of the quantity as an attribute is currently the better alternative. Even so, if in the future the language does support more detailed cardinalities, the usage of cardinalities to represent quantities would increase clarity.

A snippet of the implementation of the example class diagram is displayed in figure 3.2. With the implementation and the tabular representation, a tool that provides automated synchronisation between the two representations, is implemented and introduced in the following. It is worth mentioning, that the implementation of the class diagram could be useful for future works, as it provides a basis for further applications.

As shown, the class diagram representation makes sense, due to the complete and accurate representation of the data in a component list, the simplicity of transforming the list to the diagram, and the distinct advantages it provides, which are e.g. the addition of clarity and maintainability to the list.

In the following an approach that synchronises changes made in either representation is shown.

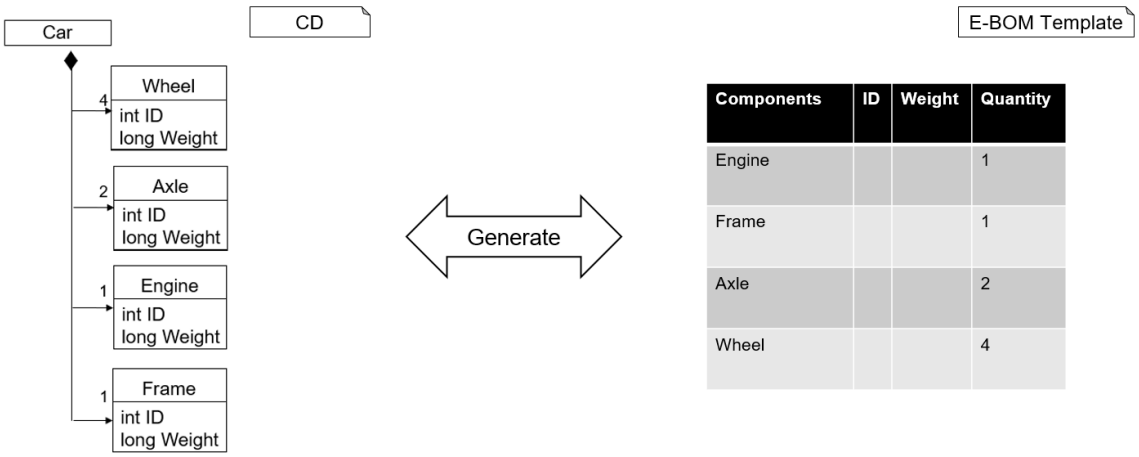


Figure 3.3: Example of synchronisation between a class diagram and its tabular representation. Either can be used to generate the other.

3.2.3 Automated Synchronisation between Class Diagram and Tablular Representation

By providing the possibility of modeling the list in tabular form and using a class diagram, making changes in either model, does not automatically update the other. Hence using both models would be inefficient, especially for lists of larger scales, as changes would be harder to track. Although, keeping both representations is preferential, as different users could want either representation, according to their needs. The original tabular representation would be preferred by someone who intends just to find specific information of components from the list and is not familiar with class diagrams. Conversely, a person who wishes to get a general overview of the composition would prefer the class diagram representation.

Considering the advantage from having both representations, a solution to the problem of synchronisation would sustain the viability of keeping both views. Considering this, an automated approach for synchronisation is introduced in the following.

The synchronisation can be performed, if there is an appropriate approach to read data from the class diagram and the tabular representation. Again, the choice of the software used to model the table is important, as it would be preferable to parse the data provided, without much hassle. Conversely, the automatic generation of a class diagram and a table is needed as a subsequent step for the synchronisation. By having functions for both reading and generating, the synchronisation can be implemented by reading the existing representation and storing the data, and consequently creating the other using the stored information.

Though the transformation of the data from one medium to the other must be tackled as well. In the case that the tabular representation is used to create the class diagram, the part names need to be read and created as classes. The information provided on the parts are then transferred to the diagram as attributes of the class. Therefore, an interface is needed to the appropriately handle the transfer and generation of the data.

Generation of Table from a Class Diagram

The generation of the table from an existing class diagram is facilitated using the parser provided in the cd4analysis language. The parser enables, the reading of data provided in the class diagram. The information from the class diagram that is necessary, namely the class and attribute names can therefore be read and stored.

By storing the class name and the attributes, the part names and the information on the parts are stored. During the generation, the class names are stored in a column and each additional information on the class is stored in a separate column in the same row. The associations of the classes are used to display hierarchy. Hierarchy is modeled by having shifting the subcomponent one column to the right. This results in all parts belonging to the same hierarchical level to be in the same column.

The information can be stored in a list containing entries for each part of the E-BOM. The entry includes the part name and the attribute names. The hierarchy is saved by having the parent part and subparts stored in the entry as well. The entry itself can be implemented by a custom data structure, which includes all the information needed.

As the information and the concrete methods to generate the table is now available, using an appropriate tool with the available data, the generation is made possible.

Creation of a Class Diagram from a Table

In case a tabular representation exists, and a class diagram must be generated from the table, the transformation needs to be reverted. The extraction of the data in the table heavily depends on the choice of the software used to display the table. If a software with the option to easily read and write data from and onto the table is chosen, the transformation would be simpler. In the next chapter, the library and tool used to solve the extraction is shown. Basically, with the right tools, the content of the columns that contain the part names are stored and used to set the name of the class and the columns containing further attributes are used later to set the attributes of the class. The class itself is then created. The cd4analysis language also provides builders and pretty printers, with which a class diagram can be created, given that the right information is available. The pretty printers then print the contents of the class diagram in the file.

Figure 3.3 displays an example of a class diagram and the tabular representation of the E-BOM, where either can generate the other. The figure 3.3 does however label the table as just a template E-BOM. As evident in the figure, the table is still missing the concrete values for the components, which means that it does not yet model a complete E-BOM.

Automated Synchronisation between Object Diagram and Table

The table containing data on the class diagram contain the attributes of each classes. If these attributes are given a concrete value, by entering the value in the corresponding position in the table, a representation of an object diagram is created. Basically now, each attribute contains a specific value. This representation is the final form of BOMs, because the concrete values are the point of interest for the users, not just the attributes.

After the values for all attributes are entered, the table can be used to generate an object diagram. For the object diagrams, the ODLangugae is utilised. Same as before, reading

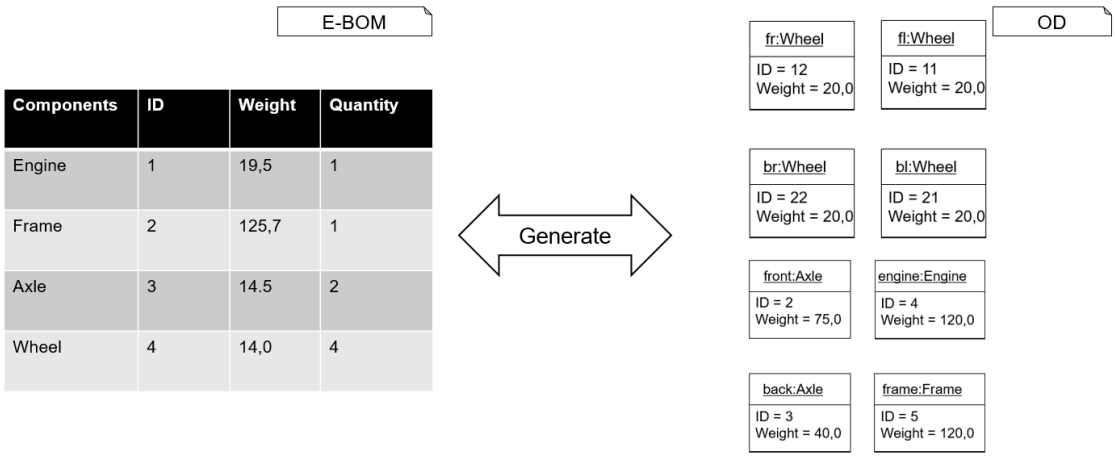


Figure 3.4: Example of synchronisation between an object diagram and its tabular representation. Either can be used to generate the other. The table can also be used to create a class diagram.

data from the table, storing it, and then using it to create the object diagram via the builders provided in the ODLanguage, are steps that can be done to generate the diagram representation. The other way around works in the same manner.

Figure 3.4 shows the possibility of generating the object diagram and the table from its adversary. Additionally, the table of the object diagram can be used to also create a class diagram as all relevant information is also displayed in the object diagram table. This is especially useful, if after the creation of the object diagram, the user the decides to add other attributes. In this case, the object diagram is fully up to date, however the class diagram would be missing attributes. Keeping the class diagram updated is important for clarity, as the object diagram illustrates the concrete instances of the components and the class diagram provides a more general view on the product composition. Inheritance displayed in the class diagram is also not as evident in the object diagram. Not being able to properly see the attribute inheritances would be a shame, as another advantage of the class diagram representation is the separation of attributes for each component. The superclass can contain the attributes relevant for all subclasses, but the attributes unique to each subclass can be displayed in the subclass alone. In the typical tabular design, this would not be possible, as the attributes are all in the same row, which means they apply to all components.

In summary, by just implementing a single representation, it is possible to automatically generate all the other representations. This means, that even though a class diagram, an object diagram, and the tabular representation of each diagram is used, no additional workload is needed and flexibility during development is introduced, as each user is allowed to use the representation of their preference. Figure 3.5 summarizes all functionalities. As of now, the generation of the class diagram from the object diagram is missing, but this is covered by the fact, that the object diagram table can create the class diagram. Here it is important to note, that during the creation of a class diagram via the E-BOM of the object diagram, a potential for information loss exists. Specific values for the attributes are in the object diagram, but not included in the class diagram. Therefore, it is important to make sure that the information on the attribute values is preserved. This can be done by having two representations of the E-BOM, one for the creation of a class diagram and another for

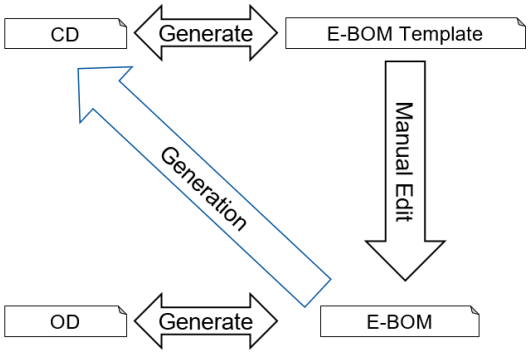


Figure 3.5: Summary of all possible synchronisations between the representations.

the creation of the object diagram. Having two versions of the BOM, which means having two tabular representations, also helps in keeping holding information on the different distribution of attributes in each class. In the object diagram tabular representation so far, the attributes are located in the header row, which inhibits the depiction of different attributes for each class. If in a separate table, the attributes names however are in the same row as the components instead of the attribute values, a varied attribution of the classes is possible. More on the concrete implementation on this is shown in later parts.

3.3 Bill of Process

The BOP includes all information regarding the production and the assembly of parts. Again, a way of modeling the BOP effectively is considered, as working with the plain, not unified representation of the data limits the potential to add automated processes and improvements in clarity. To accurately portray the data of the BOP, two criteria need to be fulfilled: First, all information regarding the workstations need to be accurately shown, and the order of the stations need to be correctly displayed. Secondly, the materials used during the activity and the resulting components from the workstation need to be visible. Additionally, more information can be displayed in the BOP, such as the machines and tools used in the station, the supervisor, the time taken in the station, and much more.

Station	Inputs	Output	Next Station
Stamping	Metal Sheet	Frame	Welding
Welding	Frame, Hood, Bumper, Door, Exhaust	Car Body	Painting
Painting	Paint, Car Body	Colored Body	Assembly
Assembly	Colored Body, Axle, Wheel, Engine	Car	Testing
Testing	Car	Car	
Inspection	Car	Car	

Table 3.2: An example BOP containing information on the workstations. Here it is a tabular representation, still a concrete approach to model the data needs to be decided

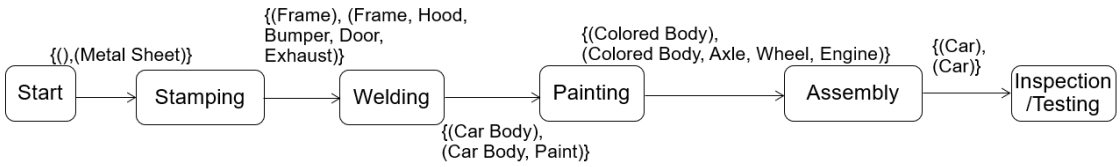


Figure 3.6: The BOP modeled as a graph. The edges are overcrowded with information as the graph provides limited possibilities to display data

The BOP seen in figure 3.2 is an exemplary bill that could extend the E-BOM from our running example. The workspaces and the information about each workstation are introduced. The next station column represents the order of the stations. The BOP 3.2 does not accurately display a realistic BOP, as much more information about the stations are usually stored. But the tool of this thesis provides expandability, according to the needs of the user, which means that the consideration of a smaller data set does not diminish the functionality and potential of the solution presented later on.

Moving forward, two alternate models for the BOP are assessed. First, the representation of the data as a graph and secondly as an activity diagram. Based on the aforementioned criteria both representations will be discussed on how well it represents the data introduced in the example BOP, and BOPs in general.

3.3.1 Representation of BOP as Graph

A possible solution is to use graphs to model the data based on the criteria. In this approach edges are used to describe the material input and output for each node. The nodes display the workstations of the production line. The edges contain a set of two elements containing the output of the prior station and the input for the next station. The complete representation of the BOP introduced earlier can be observed in figure 3.6. The graph models the stations and the material flow in the correct order.

One of the advantages of this approach is how accurately and simply it represents a very specific production process. It would be possible for example to employ optimisation processes using the graphical representation to make the processes more efficient. The simplicity and its well fittedness for optimisation can be mostly attributed to the rigid nature of the graph. It accurately represents a single production process with set stations and a rigid material flow. This in turn also means, that doing slight alterations to the stations or the information results the need to create a completely new graph from scratch.

An additional lack of flexibility is for example shown in the inability of showing parallel activities. Testing and inspection are activities in the BOP, which were intended to be carried out simultaneously, the nature of the syntax of graphs forbids it from using more than one edge. Moreover, if materials are altered for some stations, the graph is rendered inaccurate, meaning it would be necessary to redesign the whole model. Information on the workstations is also limited as node and edge descriptions are usually kept brief, meaning they do not provide sufficient space for an adequate display of the information on the workstations. If the BOP would for example include an additional column, such as tools used in the station, including the information in the graph would only be possible by adding it to the set of the edges or changing the naming of the nodes for example. Regardless of

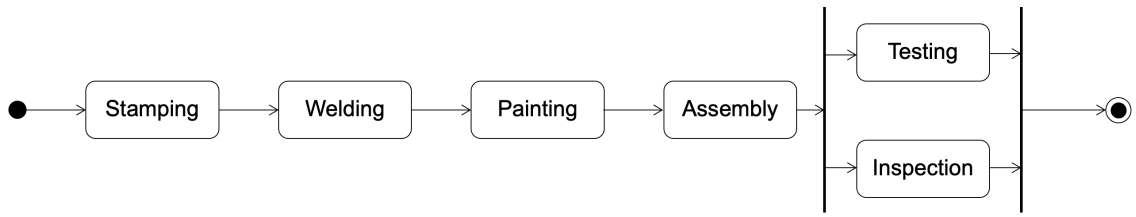


Figure 3.7: The production process of the BOP modeled in an activity diagram. It does not represent the complete factory, as material flow and additional information is missing

the solution chosen, the choice of graphical representation significantly limits the flexibility and the volume of the information of a BOP.

Having considered the outweighing downsides of this approach, the next part introduces the modeling of the BOP using activity diagrams.

3.3.2 Representation of BOP as Activity Diagram

Having previously employed class diagrams to accurately describe the E-BOM, here the application of activity diagram is introduced as the most effective way of modeling BOPs. The activity diagram suits the data presented in a BOP, as each workstation can be regarded as an activity of an activity diagram, where materials are either combined or changed. The components and different kinds of transitions between the stations offered in activity diagrams are ideal to model the production line of a factory. In figure 3.7, a hypothetical BOP for the running example and the activity diagram modeling is shown. The stations are meant to model a subset of work stations in a real car manufacturing factory. The representation of the BOP as an activity diagram provides the significant advantage of being able to implement the diagram using the adlanguage. The language provides parsers and builders, which can be used similarly to the E-BOM, to synchronise the list. Contrary to the graph representation, the activity diagram also offers more possible transitions between the nodes and in overall, a better modeling of realistic assembly lines.

Whilst the activity diagram does account for the first criteria of modeling the production line adequately, it does not satisfy the second, as concrete material flow is not yet evident from the diagram. Nevertheless, this can be used to solve a problem, which was present in the previous graph representation. By having the activity diagram displaying the general factory street and its work stations, and adding another component responsible for storing the material flow and detailed information on the stations, it is not as rigid as the previous concept. The order and tasks of workstations in a production process does not tend to change as much as the components and information in each station. The separation of the factory modeled by the activity diagram and another fitting extension, which displays the additional information of the workstations creates a proficient model for the BOP, which increases visibility and better adaptability. This does however depend on the design of the second component. In the following two sensible design choices are introduced and discussed.

Tagging Language as Extension to the Activity Diagram

Tagging languages for DSLs introduced in [GLRR15a] are one possible way of adding information to the activity diagram separately. It is used to keep a DSL clean, readable, and reusable in different contexts. By using a DSL-specific tagging language and a schema language, this approach manages to add information, whilst keeping the artifacts separated. The reason both languages are used, is because using only a generic- or DSL- specific tagging language, would lead to significant disadvantages, such as promiscuous tag models or high initial overhead [GLRR15a]. The usage of both DSL-specific tagging language and a schema language enables a specific fit to the DSL, artifact separation and reuse with different tag decorations, systematic derivation, which considerably reduces work needed to implement the tag languages, and the fact, that the language follows a defined type schema [GLRR15a]. This is an excellent solution to the problem mentioned above, as it is able to add information without changing the activity diagram itself. Additional information for the stations, such as tools used, maximum capacity, and many more, can hereby be added after having defined a tagging language and a schema language. The added information to the activity diagram is separate and comprehensible.

Nevertheless, in this thesis, this approach will not be used, as the diagrams are usually updated by engineers, whom usually do not have extensive computer science-based knowledge. To learn how to apply this approach, the user would have to learn the details on this approach. One could argue, that an automated system can be developed, that automatically generates the necessary components and provides a user interface, where one can simply add information without having to know the intricacies. However, if there are some changes which the generation process cannot cover, the tool would not be usable by the users. Even though the tagging method from [GLRR15a] is very potent, another more sensible method, the usage of tables is introduced in the next part.

Tabular Representation as Extension to the Activity Diagram

Table management is a task commonly referred to as a basic skill when it comes to working in a production company. As already pointed out previously, tabular representations of data provide a well-fitting feature for data management in a production process. Since the previous automated process was based on tabular representations, it is sensible to streamline the applications for easier usage, by using the same format. The data which the tagging language was able to add, can be also easily displayed in a table. Previously the class and object diagrams extended the tables, to increase clarity, which in this case is done by the activity diagram.

Table 3.3 shows the table, that adds the information to the activity diagram. In the first column the name of the activities is saved. Additional information on each station can then be added in other columns. The table 3.3 is very similar to the BOP itself, where only the ordering information of the workstations are omitted. One could therefore argue, that the table itself is sufficient to model the data. Although, a table by itself is limited and does not provide an efficient way of modeling complex production processes with multiple forks, parallel activities and joins. To display this in the table, it would have to be done forcefully with a very complicated approach. The usage of an activity diagram provides a simpler alternative. Additionally, the visual representation that the activity diagram provides gives a better overview on the current production process.

Station	Inputs	Output
Stamping	Metal Sheet	Frame
Welding	Frame, Hood, Bumper, Door, Exhaust	Car Body
Painting	Paint, Car Body	Colored Body
Assembly	Colored Body, Axle, Wheel, Engine	Car
Testing	Car	Car
Inspection	Car	Car

Table 3.3: A table as extension to the activity diagram, which adds further information on the stations. The table and the activity diagram together form the model of the BOP

The tool could provide another functionality in which the names of the stations can be read from the activity diagram and automatically inserted into the table. However, as the additional information is not present in the activity diagram, the whole table cannot be generated. The other way holds true as well, as the separation of information in both representations have been reasonably established previously. Therefore, it is unavoidable, that the representations need to be created separately. This also means, that the production planner needs to be able to implement an activity diagram. Still, the syntax of the ADlanguage is not complex and should not pose too much of a challenge. Adjustments during the planning are then consequently done in each representation. Synchronisation is only necessary if a new station is added. In most cases however, the work stations are mostly determined in the beginning of the production planning, especially for companies that build already existing products, such as car manufacturers.

Now that the BOP and the E-BOM are fully modeled, the generation of the M-BOM from the two preceding data sheets is possible.

3.4 Manufacturing Bill of Materials

A significant difficulty many businesses that are involved in production processes face, is the synchronisation of different BOMs used in different phases of the product life cycle. Especially the synchronisation between the E-BOM and the M-BOM is an area which would greatly benefit from an automated approach, as manual synchronisation is significantly more prone to errors. This is because many people from different domains work on the lists and most of the time they do not take the need for synchronisation into consideration. The correctness and completeness of the M-BOM is especially important, as it displays the information which is crucial for the transition from a plan of the product to the concrete, physical, finished product itself. It contains information on where each component is bought from, where and with which means the components are processed and much more. It is therefore pivotal for the completeness and the quality of the end product.

Contrary to the E-BOM, it also contains more components, as connecting parts such as cables and glue are added during the production and not during the planning of the design. Additionally, some parts need to be processed. For example, the frame of the car needs to

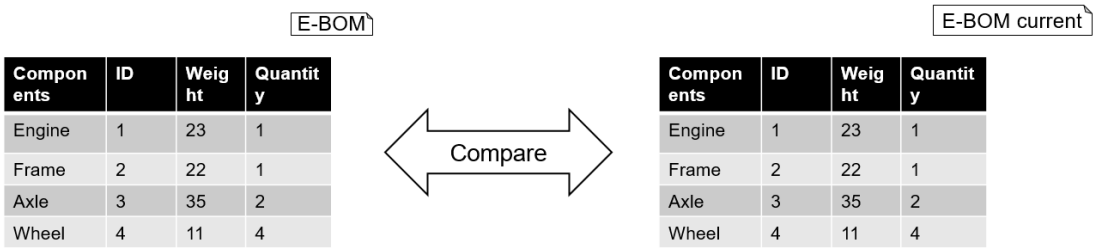


Figure 3.8: Comparison of the E-BOM file in the shared directory and the current E-BOM, to check whether an outdated version is used.

be painted. In this case the frame, the paint, and the painted frame are all included in the component list even though in reality, just the painted frame is visible. The hierarchy and the grouping of the components is also based on the assembly, which all in all means, that both lists contain fundamental differences. Therefore, an automated generation of the M-BOM from the existing E-BOM and the BOP models is introduced, which guarantees the correctness of the list and therefore solves the afore mentioned problems.

3.4.1 Generation of M-BOM

The generation of the M-BOM is arguably the most important feature, as the M-BOM holds the information relevant for the final product. The automatic generation without user input would guarantee, that no errors exist in the M-BOM. The generation process is divided into four steps:

1. Checking E-BOM Version

Prior to the generation of the M-BOM an intermediary test is performed, which checks whether the E-BOM used during the generation of the M-BOM is the most recent version. This ensures, that before the M-BOM is generated, it is known that an outdated version of the E-BOM could be referenced. In this case the M-BOM itself would also be outdated, as some components might be missing, or a surplus could exist. Therefore, by performing the test, the user can be sure, that the current E-BOM is used. Consequently, this would mean that if the generation is properly designed and implemented, the M-BOM is guaranteed to be error free.

Every time, a table for a class diagram or the object diagram is created during the creation of the E-BOM, the table is saved in two locations. A shared folder, which the M-BOM developer and the E-BOM developer have access to, and a separate folder, which only the E-BOM developer uses. If the user always generates the lists using the automated process, it is guaranteed, that the current version of the E-BOM exists in the shared directory. But if the product designer has unsaved changes, the E-BOM will be outdated.

Figure 3.8 displays the comparison done with both lists. If a difference is spotted, the user is notified, and the current version needs to be fetched. One could argue, that the list could be a single file in the shared directory, so that it is ensured, that the E-BOM is always up to date. Nonetheless, in most cases the current official version of the E-BOM

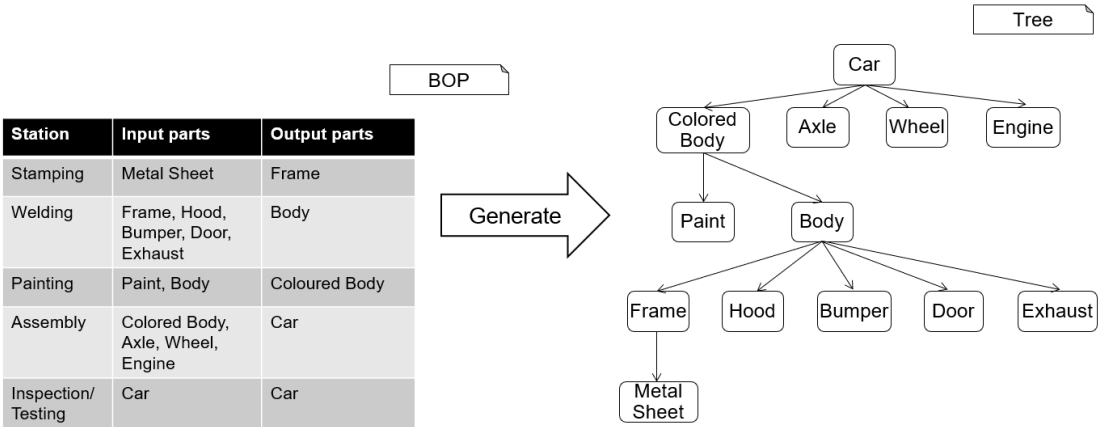


Figure 3.9: The tree resulting from the information of the BOP. It displays the hierarchy of the parts

is not always the one with the latest changes. The product designer could have separate lists depending on different criteria. Changes could be incomplete and only possible to be applied to the list after a certain amount of time. Separating the official current list and the one in development is therefore important.

2. Tree Creation from the BOP

After the test is passed successfully, the main steps for generating the M-BOM is performed. For the generation, all the parts involved in the BOP are considered as parts in the M-BOM. This means, the inputs and outputs of all stations. This assumption can be made, as in some way the parts that end up in the final product must be added somewhere along the line. However, since the BOP is modeled as a factory, some parts are listed more than once. As seen in the 3.3, the testing station for example does not alter any parts, it just tests whether the functionalities of the car work as intended. Moreover, the output of a station is also often used as part of the input of the next station, as the factory street is linear and adds parts in stations to existing parts. Therefore, a simple listing of all input and parts would not suffice, due to duplicate entries. The inputs and output are inserted as a comma separated list, which can then be split into single parts using regular expressions, when they are read. Separate checks can be done to ensure that no duplicate parts are included in the new part list.

Another aspect that needs to be addressed, is the existence of hierarchy in the BOP. The input of a station are basically the subparts of the output. In the table the hierarchy is typically represented by shifting columns to the right. Even so, to be able to create the correct shifting, it is favourable to know the hierarchy prior to the creation of the table. Therefore, a tree is constructed first, which displays the hierarchy and is also easier to use later. An algorithm that stores the parts into a tree structure to represent hierarchy is introduced. The BOP needs to be inverted first, as the output of the last station is the value of the root of the tree. From there the inputs of the station, given the station, performs changes to the parts, are then appended as children of the current node, and so forth. Figure 3.9 illustrates the hierarchy of the BOP in form of the tree. It is created by the algorithm and is consequently used to read the parts with its hierarchy.

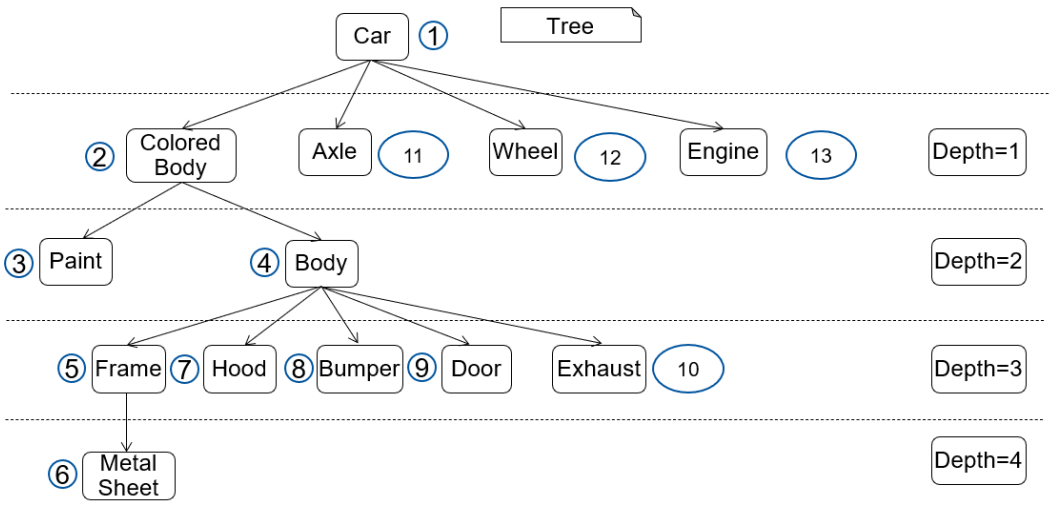


Figure 3.10: The in-order traversal of the tree. The depth of each node denotes the hierarchy level in the M-BOM

3. Traversal of the Tree

The hierarchy of each part is displayed by the depth of each node in the tree. This makes it a lot easier to insert the parts into the M-BOM table. Traversing the tree in order and returning the values of each visited nodes that are added in that order to the list, ensures, that all parts of the same hierarchy level are bundled together in the resulting list.

Figure 3.10 shows the order in which the nodes are traversed, when using the in-order traversal. This order is the same as the order in which the parts are added to the table. More concisely, the number of each node also displays the number of the row in which the part is added. The depth of the node conversely illustrates the column of the table in which the part is added. With these two parameters set, the position of each part in the table is precisely located, and the table can be created.

4. Creation of the Table

The table is created with each part in its correct position, which were found in step three. Though, as visible in figure 3.11, the attribute names up top, and the known values from the E-BOM need to be inserted, to avoid redundant workload. The attribute names can be read from the E-BOM file and added to the M-BOM. By finding the row in the M-BOM table of the parts that already exist in the E-BOM, the concrete values can be read from the E-BOM and then added to the M-BOM.

Figure 3.11 is the result of a completed, automatic generation of an M-BOM table. The user is then able to complete the table by inserting the values for the attributes manually.

Now that all of the parts are in the new list with its respective hierarchy, it should be compatible with the automation tasks implemented for the E-BOMs, especially the automated creation of an object diagram from the table representation. Nevertheless due to the added columns for the hierarchy a small adjustment is needed, so that the system recognises at which column, the attributes of the parts begin. This can be done by just

Depth = 1	Depth = 2	Depth = 3	Depth =4	ID	weight	Quantity
Colored Body						
	Paint					
	Body					
		Frame				
			Metal Sheet	26	70,0 kg	1
		Hood		21	35,0 kg	1
		Bumper		22	50,0 kg	1
		Door		24	41,0 kg	4
		Exhaust		25	19,5 kg	1
	Axle			12	14,0 kg	2
	Wheel			11	14,5 kg	4
	Engine			23	19,5 kg	1

Figure 3.11: The M-BOM table, generated after the 4 steps. It can be used to perform the same automated tasks as the E-BOM could

checking for each row where the first non-empty cell is located in, which is where the part name is. The next non empty cell is the first attribute entry and after that all other attributes are in the adjacent cell. Therefore, continuing to save the attributes until the next empty cell is reached, ensures that all attributes are recognised. By then storing the column numbers of each part, the hierarchy can be preserved and used to create the correct associations between the objects and classes, to display the hierarchy structure in the diagrams. As seen in figure 3.5 from the object diagram table, all other representations can be generated.

Conclusively the tool has the following functionalities:

1. Generation of an BOM in form of an Excel table from an existing Class diagram, or the other way around.

2. Create an Object diagram from the BOM by writing values for the attributes into the Excel file, and vice versa, create an BOM list from an object diagram.

3. Generation of a hierarchically structured M-BOM file from an E-BOM and a BOP, which can be used to perform function one.

Now that the methodology of the implementation is covered, the next chapter addresses the concrete implementation and a concrete demonstration of the tool from the perspective of users of the toolset.

Chapter 4

Implementation and Demonstration

Having covered the methodology of the automation tool in the previous chapter, the concrete implementation and a full demonstration of the tools is presented here. The demonstration is purely from the perspective of a user and intends to illustrate the clarity and simplicity of the usage of the tool. In this chapter, the explicit software choices for the representations and the code accompanying the automation is presented.

No profound previous programming experience is needed to use the tool. Due to the automation, problems and mistakes arising from the before manual synchronisation process are eliminated.

4.1 Preparation

Before the program can be used, the user needs to get the program to run on the current device. This means, that the current version of the project needs to be available. As of now, the prototype project is located on GitLab which needs to be cloned or downloaded by the user. This could pose a challenge to someone who has not used git before, yet a detailed, step-by-step guide would drop the difficulty. Next, the project needs to be opened in an editor and the command `build.gradle` needs to be run, to build the project and initialize all the needed dependencies. After that the Main class needs to be run and then the user can perform the needed tasks. Figure 4.1 shows the console output of the current tool. It gives the user the option to choose which function to utilize. There are currently

```
> Task :Main.main()
Press 1: Class diagram -> Excel table
Press 2: Excel table -> Class diagram
Press 3: Excel table -> Object diagram
Press 4: Object diagram -> Excel table
Press 5: Activity Diagram -> Excel table
Press 6: Generate MBOM
Press 7: Finished
```

Figure 4.1: The console output of the tool. Pressing the corresponding numbers triggers the functions.

six functions in the tool, all of which intend to improve data management in the product planning and the production process.

The preparation steps in the current prototype version are not optimal. Although, the tool currently does what it is primarily designed to do, which is the automation of various stages in the process of the automated generation of the M-BOM. Still, in the future, it would be preferable to simplify the preparation steps, by for example creating a graphical user interface or another simpler solution instead of the installation process, which loads the current version of the project and the dependencies automatically.

Each of the functions and its implementation is presented and discussed in the following.

4.2 Using Excel for Tabular Representations

At first, in the product design phase, the design engineer develops an E-BOM, which is subject to a lot of changes during the design, because better solutions or better, cheaper parts can be found, whilst trying to optimise the list. Costs, weight and quality are main aspects the designer considers creating the most qualitative, cost-efficient, and optimally weighing car.

For the tabular representation of the E-BOM, Excel is a popular choice amongst many, as it offers all basic functions regarding table management. Alternatively, other data management software for larger data sets, for example SQL, could be used to store and retrieve the information. Whilst the datasets for part lists in the automotive industry are not small, they are not big enough to justify using other data management software, especially because most of the people accessing the data are not from the data science domain. They do not possess the proper knowledge of how to retrieve and store data in these programs. Additionally, accessing the information with SQL queries is mostly used for cleaning, filtering, sorting, and joining data. It is usually not used and optimised to create databases by itself.

Most users come from the engineering domain, which means that they do not have experience with data management systems. However, most do have experience with the usage of Excel as a worksheet editor. Also, most enterprise computers have Excel installed, which means no additional software needs to be bought and installed. Excel also has the advantage of having the Java library Apache POI, which can be used to retrieve and print data from and on the sheet. These functions can be used for the automated synchronisation. The features in terms of table management, which Excel offers, is also useful for its users. Therefore, it is the most appropriate software with which the tabular representations are represented. Hence, for all the tables used in the tool, Excel is utilized. For the operations of retrieving data and writing data into the tables, the Apache POI library is employed.

4.3 Class Diagram to Excel Table (1)

If the product designer has previous experience with UML diagrams and the implementation of class diagrams, the class diagram representation of the BOM can be done by the designer. Then the user can generate the corresponding Excel tables, by pressing one as in put, after launching the main method. The user is then asked to insert the paths of

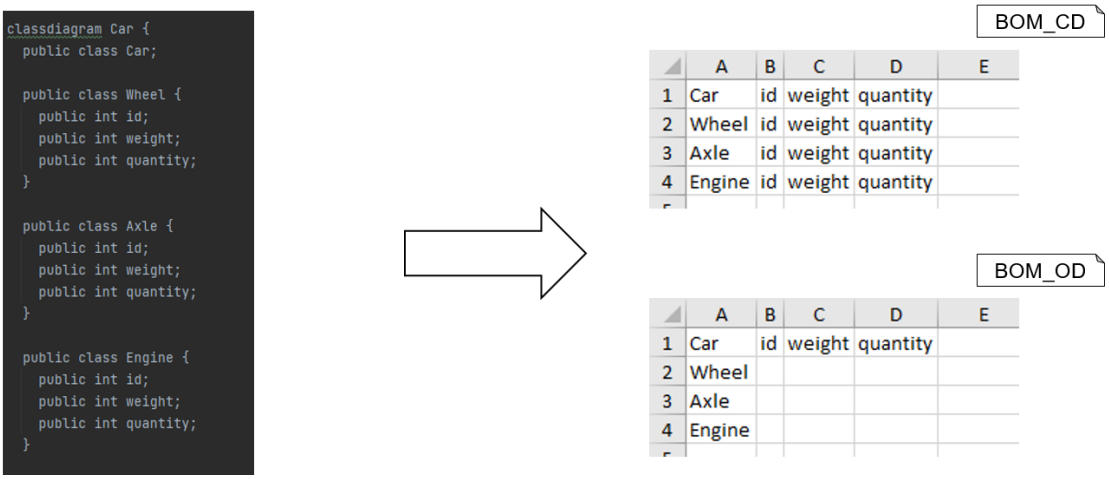


Figure 4.2: Creation of two worksheets from the class diagram. `BOM_CD` is used for class diagram generation and `BOM_OD` for object diagram generation

the Excel file and the cd file, which need to already exist. They are not automatically created by the tool, but the creation of an empty class diagram and Excel file is not challenging. The BOM is then generated as an Excel file and it is guaranteed, that the list is complete and correct, as long as the original class diagram representation of the BOM is correct. A more explicit explanation on the generation of an Excel table from a class diagram is provided in the following. For the generation of the table, the two java classes: `BOMGenerator` and `DiagramExtractor` are utilised.

The class `BOMGenerator` is responsible for the generation of all tabular representations of a BOM, by writing the information into an Excel worksheet in the corresponding positions. The `DiagramExtractor` class on the other hand, uses the method `extractCD()` to extract data from a class diagram and returning it as a list of so called BOM entries. The custom class `BOMEntry` contains the name and attributes of a part and is used extensively throughout the tool to store information on the components for further use. Currently the attributes id, weight, and quantity are hard coded into the class `BOMEntry`, however in the future the attributes could be stored in two separate ordered lists with the names and the datatypes as elements of the list, to ensure a dynamic design of the component attributes. An ordered list of `BOMEntry` objects represents the java implementation of a complete BOM, because a BOM basically is a list of entries for each component with its respective attribute values. The method `extractCD` works as follows: First a parser for the class diagram, provided by the cd4analaysis language, is initialised. The parser can read the name of the classes and get an attribute list containing all of the attributes that a class possesses. By iterating over all classes and retrieving each class name and its attribute names, the `BOMEntry` objects of each class can be constructed. Each `BOMEntry` is appended to a list of entries in an iteration of the loop. Here the attribute values for the entries are the names of the attributes itself and not the concrete values of the attributes. The reason being, that the class diagram does not contain any information on the values of the attributes.

Now that the list is filled with `BOMEntry` objects, the `generateCDBOM` method from the class `BOMGenerator` receives the list of BOM entries as input. Additionally, it receives the path of the Excel file in which the table is to be generated. The `generateCDBOM` method, as its name suggests, is the method, that creates the Excel worksheet with the

E-BOM template. First, using Apache POI, a workbook and a worksheet is created. An algorithm then generates the table in the created worksheet, by iterating over each `BOMEntry` in the list received as input. The first worksheet is called `BOM_CD` and is used for the synchronisation between the class diagram and the E-BOM template. Moreover, another Worksheet, `BOM_OD` is created, which is used, to synchronise an object diagram and the list after the general outlay of the parts and the properties are decided. The significant difference between the two sheets, lies within the position of the attribute names in the table. As illustrated in figure 4.2, the attribute names in the `BOM_OD` are in the upper most row. The rows with the component names are empty, as the user manually enters the values for the attributes in those cells. The `BOM_CD` however, has the names of the attributes in each row of the component. This separation is important, because otherwise it would not be possible to generate classes with different attributes. But the separation of attributes for each class is one of the main advantages the class diagram representation holds over a list representation.

The `BOM_CD` is created by the method `createCDSheet`, which iterates over each `BOMEntry` and adds the data to the correct position in the Excel table. The `BOM_OD` however, is created by the `createODSheet` method. Here the list of `BOMEntrys` is first converted into a list, according to the needs of an object diagram via the method `convertToOD`. In the first entry of a `BOMEntry` list for an object diagram, the name of the diagram and the attribute names are saved. In the next entries, the name of the part and the values for the attributes are stored. The `createCDSheet` method also calls the `createExcelFile` method, which generates the sheets not only in the given path, but also in the shared directory. This file is used for the comparison for the M-BOM generation, to check whether the current file is present.

Therefore, both `BOM_CD` and `BOM_OD` are generated by the `generateCDBOM` method. Consequently, the file can be used to perform further tasks of the tool.

4.4 Excel Table to Class Diagram (2)

If the designer does not have any previous experience with UML diagrams and their implementation, a table in an Excel file with the data of the parts can be created. The function of the tool, which generates the class diagram implementation from the table, is used for the transformation. It is launched by feeding 2 as an input and consequently giving the paths to the class diagram file and the Excel file. The table needs to be conform to the `BOM_CD` sheet in figure 4.2. This means, the name of the worksheet, needs to be the same, and the position of each part and attribute name, need to be according to the specifications. For the creation of the class diagram from the tabular representation, the methods `extractCDBOM` in the class `BOMExtractor` and `createCD` in in the class `DiagramCreator` perform the main tasks.

First, the method `extractBOM` receives the path to the excel file from which the data is to be extracted, as a parameter, and looks for the worksheet `BOM_CD`. By using methods from the Apache POI library, which return specific elements from a row, the position of the part name and the attribute names in the Excel file are located. The name of the component is always the first element in the row, which is why the position is determined by the method `getFirstCellNum`. Consequently, the last entry is the last attribute of all attributes and is retrieved by the method `getLastCellNum`. By then iterating backwards over the

number of attributes, all positions for the data is identified and can then be used to create the `BOMEntry` list. Again, the list of BOM Entries with the corresponding data from the Excel file is returned.

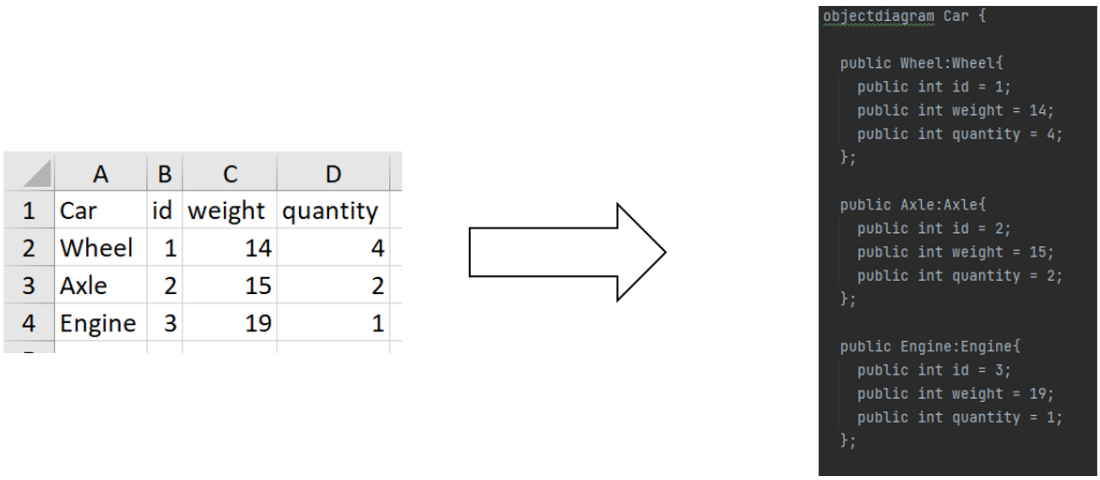
In `createcd`, the list is received in addition to the path, in which the class diagram will be generated. The cd4analysis language provides builders, with which class definitions, attribute lists, and other necessary components can be built. A pretty printer, that is also provided, then prints the contents of the class diagram, which has previously been build. In a loop that iterates over each `BOMEntry` in the list, the class names, attributes, the data types are build using the builders. Finally, the pretty printer is used to write the contents of the class diagram in the requested cd file. This operation is exactly the opposite direction of the figure 4.2, generating the class diagram implementation from the `BOM_CD` worksheet. Hence, both the table created by the user and the class diagram implementation is available. Consequently, the BOM developer can optionally use the class diagram as an overview of the product composition. The comprehension of the diagram does not require any profound knowledge of UML diagrams.

Now that the parts and its attributes are set, the engineer can then proceed to fill out the specific values for the attributes of the parts. The representation of the model using a class diagram does not offer the option to specify concrete values. The concrete instantiation of the classes results in a list of objects, which form an object diagram.

4.5 Excel Table to Object Diagram (3)

If the user decides to create an object diagram from an Excel file, the third option of the tool is triggered. Ann object diagram is created from an excel table after the user has filled in the concrete values. The values are entered to the worksheet `BOM_OD` of the E-BOM file in the blank cells of the corresponding component row.

As discussed earlier, during the generation of the Excel table, the two worksheets `BOM_CD`, `BOM_OD` are created. The concrete difference is the position of the attribute names. In `BOM_CD` the attribute names are located in each row, whilst `BOM_OD` contains the attribute names in the top most row and empty spaces below, which can be filled by the user. This results in an instantiation of the classes, from which an object diagram can be created. The separation of worksheets according to its type eases complications, which would otherwise arise. Still, it is worth pointing out, that the `BOM_OD` has difficulties in displaying cases, where different classes comprise different attributes. A single row displaying the attributes for all objects limits the modeling capabilities. Nonetheless, the other table `BOM_CD` is generated simultaneously. This worksheet does cover the different attributes of the classes. The `BOM_OD` therefore has all occurring attributes in the top most row and the objects which do not have certain attributes, do not have a value for the corresponding column. This task is performed in the `extractODBOM` method, which is almost analogue to the `extractCDBOM` method. Instead of the `BOM_CD` worksheet, here the `BOM_OD` worksheet is read. Figure 4.3 shows the worksheet from earlier with added values. Here the data from the worksheet is retrieved in the same manner as before, the only difference being the concrete data retrieved and stored in a `BOMEntry`. The first `BOMEntry` now contains the diagram name, and the attribute names. The following `BOMEntrys` read from the table contain the part name and the concrete values for the attributes.



	A	B	C	D
1	Car	id	weight	quantity
2	Wheel	1	14	4
3	Axle	2	15	2
4	Engine	3	19	1

```
objectdiagram Car {

  public Wheel:Wheel{
    public int id = 1;
    public int weight = 14;
    public int quantity = 4;
  };

  public Axle:Axle{
    public int id = 2;
    public int weight = 15;
    public int quantity = 2;
  };

  public Engine:Engine{
    public int id = 3;
    public int weight = 19;
    public int quantity = 1;
  };
```

Figure 4.3: The object diagram generation from the `BOM_OD` worksheet. Generating the table from the diagram is also possible

With the completed list of the entries, the object diagram is generated in the `createOD` method from the `DiagramCreator` class. Again, the `createOD` method has many similarities with the `createCD` method. First the builders, needed for the creation of the object diagram, are initialised. Here, the builders from the od language are utilized. Consequently, the builders are used to build each object of the diagram with each of its attributes and values. Instead of just having the attribute names, the values of the attributes need to be built as well. The names of the attributes are retrieved from the first `BOMEntry` and the following contain the values for each attribute. Soon after, all objects are added to the object diagram builder, which are consequently build. The parser `OD4DataFullPrettyPrinter` is then used to fully print the object diagram into the od file. The path of the file is specified during the selection of the option three of the tool. The resulting object diagram file can is shown in figure 4.3.

If the user makes changes to the object diagram, the corresponding Excel table can be generated with the following function.

4.6 Object Diagram to Excel Table (4)

By selecting option four, the Excel table of the object diagram is generated. The main idea is the same as in the first function, which is to parse the diagram and create a list of `BOMEntrys`, that are consequently used to create an Excel table. The class `DiagramExtractor` also contains the method `extractOD`, which is responsible for the extraction of the data from the object diagram into a list of `BOMEntrys` and returning them. This is done by first parsing the object diagram and then iterating a loop over all objects in the diagram. The first BOMEntry contains the object diagram name and the attribute names of the components. This ensures, that the format of the `BOM_OD` sheet remains intact. The following entries contain the name of the component and the values for each attribute.

This list is then fed as a parameter for the `generateODBOM` method, which is again similar to the `generateCDBOM` method. The data from the list is transferred to the `BOM_OD` sheet

and created by the `createODSheet` method. The `BOM_CD` sheet is also created, so that the class diagram can still be created from the data in the Excel file. By executing the `convertToCD` method, the list of BOM entries is converted to a list of BOM entries, which can be used to create the `BOM_CD` sheet. Again, the `createExcelFile` method generates two separate files, one for the directory maintained by the product designer, and the other in the shared directory.

These steps can be mixed and repeated until the design engineers finalises the E-BOM. Consequently, after or during the design of the E-BOM, the production process and the corresponding BOP is developed. The implementation and the usage of the functions for the generation of the BOP is presented next.

4.7 BOP Generation (5)

The adlanguage is used to implement the activity diagram. As mentioned in the methodology chapter, the activity diagram is accompanied by a tabular representation of the data on the workstations. Similarly, to the cd4analysis and the OD language, the adlanguage possesses parsers, builders, and pretty printers, which offer the potential for automated processes. Hence, it could be possible to create the activity diagram from the table and the other way around, just like in the options before. Although, as discussed earlier, the separation of the data in both representations has been done to display data in the most fitting model. Therefore, it is impossible to create either representation fully from the other. Thus, the production designer, needs to know how to create either representations, which could be a problem if the designer does not know how to use the adlanguage. Still, the language is easy to learn and use, which therefore should not be a problem.

The representations cannot be fully created by the other, is what has been stated before. This does however mean, that some overlapping parts might still be automatically generated. If the table is created first, the names of the stations are written in the first column. Those could be used to create stations in the activity diagram. The ordering would still be missing in this case. If the activity diagram exists, the parser could store the names of the stations and add them into the Excel table. This is done in the method `generateBOP` from the class `BOPGenerator`. Similarly, to the E-BOM generator, a worksheet with the sheet name BOP is created. Subsequently, a first row with the descriptions of station, inputs, and output is created. After having added the names of the activities, the tool then asks the user to manually write the inputs and output that are going to be added into the table for each station. The user is also reminded to write the parts in a comma separated list, which plays part in the extraction of the table. The generator then proceeds to write the content into the corresponding positions of the table.

The class `BOPEntry` serves the same function as the `BOMEntry`. It stores the data from a single workstation, which currently includes the list of inputs, output, and the station name. In future works, this can be dynamically modeled by employing lists of information categories for the stations. A list of BOP entries comprises the complete tabular representation of the workstation data. Changes to the BOP which include the workstation names, can then be done in either the activity diagram or table and the other is subsequently generated. All other information, such as a change in the order or the addition of further information on the workstations, are not subject to the synchronisation process, as this information is only part of either representation. The code frame for the

generation is already in the implementation. Even so, as a solution for a proper import of the adlanguage dependency has not been found, the complete functionality is not included in the prototype tool. But the addition of the tool is not difficult, as just the import of the dependency must be done and an extraction of the names of the activity diagram has to be performed.

After the development of the BOP is finished, a final activity diagram and a list with the station names and inputs and outputs for each station is present. In the next step the automated generation of the M-BOM is demonstrated based on the E-BOM and the BOP.

4.8 M-BOM Generation (6)

Now that the BOP and the E-BOM is present the user can automatically generate the M-BOM by choosing the option 6 in the tool. The function `generateMBOM` from the class `BOMGenerator` receives the path to the E-BOM Excel file, the BOP Excel file, the E-BOM from the shared repository, and the path of the file in which the M-BOM is generated.

4.8.1 Checking E-BOM Version

Before the generation, the test, which checks whether the E-BOM is up to date is performed. If there is a problem the user receiving the error message notifies the responsible parties, so that the changes in the E-BOM that were not synchronised, can be revised. This error check is necessary as otherwise an older version of the E-BOM could be used for the generation. Performing the test successfully, ensures that the list is completely up to date and valid, eliminating any problems arising from the usage of outdated lists. If the user always uses the functions of the tool to create the E-BOM, it is always up to date as all components are automatically, completely updated and stored in the shared repository. The only case in which the E-BOM is not correctly updated, is when the table is edited without using the transformation function. In this case, the test in the tool needs to be able to catch the discrepancy in the lists.

This task is performed by the method `testVersion`. The E-BOM file which is constantly changed and updated is in the folder which contains all BOM related data. Basically, the `testVersion` check compares the current E-BOM with the E-BOM in the shared repository. More specifically, it compares each `BOMEntry` from both lists, and returns a truth value depending on the uniformity of the documents. If it returns a false value, the E-BOM has been manually modified without using the automated transformations to update the data. The `testVersion` method stops the M-BOM generation as long as the E-BOM check is not validated. After the user has corrected any missing updates, the generation of the M-BOM is executed.

After the checks, the M-BOM is automatically generated.

4.8.2 BOP Extraction and Tree Construction

The list of BOP entries is retrieved in the class the `BOPExtractor`. The method `extractBOP` employs a similar algorithm to the one in the `BOMExtractor` `extractBOM`

method. It reads the data from the correct positions in the Excel file, and stores it into a `BOPEntry` object. Here, the comma separation done by the user input earlier comes into play. The string read by the extractor is then separated by a regular expression to form a list of Strings which form the list of inputs in a station. For each station a `BOPEntry` object is created and appended to a list of `BOPEntrys`. The method then returns the list, which includes all information included in the BOP.

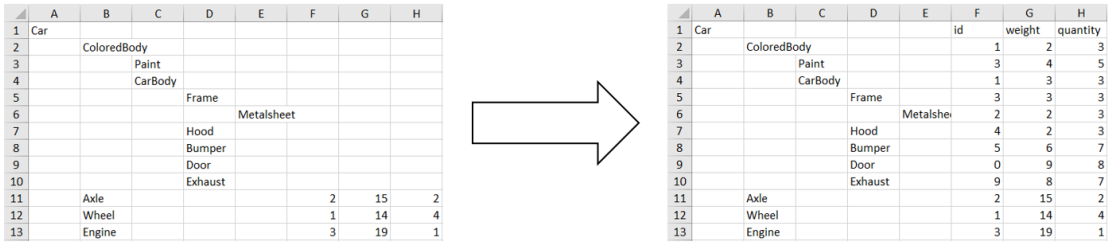
From the list of BOP entries, the additional components, and the hierarchy of the parts are derived. The M-BOM features more components than the E-BOM, as components relevant to the assembly process are added. The hierarchy of the M-BOM cannot be derived from the E-BOM, as the E-BOM bases it on the engineering domains, whilst the hierarchy for the M-BOM is based on the production process shown in the BOP. As the input in a station are subparts of the output and the next station always includes the output of the previous station, a connected hierarchy between the components can be formed. This approach assumes that each station receives the output of the previous station and uses it as a subpart for further processing. The algorithm in `generateMBOM` that forms the M-BOM, traverses the following procedure:

First, the workbook and the worksheet in which the M-BOM is to be created, is initialised. Next, using the method `createTree`, a tree is created from the `bopEntries` list. This is done prior to the writing into the excel file, to simplify the writing process. The tree is modeled by the custom `Node` class. A `Node` object can have a parent node and a child node. The list for the BOP Entries is reversed at first. This is done, because due to the nature of the BOP the final product, which represents the root of the tree is the output of the last station. The `createTree` method then creates the root node of the tree, which is the end product, and now the parts in the first BOP entry. From that node it iterates over all sub parts and creates them as children of the root node. The children are the inputs for each output. There is also a check to see whether changes to the parts happen in a station, as there are some stations such as testing stations, that do not actually facilitate any physical changes to the parts. After that, the sub classes of the children are added, and so forth until all parts are included in the tree.

4.8.3 Tree Traversal and Storing Values

Having sorted the hierarchy first, a simple traversal of the given tree can be used to write the parts in the correct hierarchy. The tree can also be used for further applications if needed.

In the `traverseTree` method, the tree built in the `createtree` method, which is now a Node element, is traversed. The data from the traversed nodes are written into the worksheet. The algorithm uses a stack to traverse the tree in order and in each visited node, the name of the part is written. The hierarchy is defined by the depth in the tree, and is derived here by using the method `getDepth`. This method checks the depth of a node, by going upwards in the tree checking whether parents exist, until it reaches the root node. The depth of the node then is used as the cell position in which the data is written into. This creates the structured parts list that displays the components and its subcomponents based on the production process. After the traversal is finished, the component names are positioned in the M-BOM Excel file, respective to their hierarchy.



	A	B	C	D	E	F	G	H
1	Car							
2		ColoredBody						
3			Paint					
4			CarBody					
5				Frame				
6					Metalsheet			
7				Hood				
8				Bumper				
9				Door				
10				Exhaust				
11		Axle				2	15	2
12		Wheel				1	14	4
13		Engine				3	19	1

	A	B	C	D	E	F	G	H
1	Car					id	weight	quantity
2		ColoredBody				1	2	3
3			Paint			3	4	5
4			CarBody			1	3	3
5				Frame		3	3	3
6					Metalshe	2	2	3
7				Hood		4	2	3
8				Bumper		5	6	7
9				Door		0	9	8
10				Exhaust		9	8	7
11		Axle				2	15	2
12		Wheel				1	14	4
13		Engine				3	19	1

Figure 4.4: The M-BOM table on the left is the table before the attribute names and the existing values from the E-BOM are added automatically

4.8.4 M-BOM Worksheet Creation

The `createExcelFile` method then creates the table, seen on the left of figure 4.4. Nonetheless, the M-BOM generation is not yet finished, as the attribute names and the values for the parts, which were already listed in the E-BOM, are missing.

Missing values are added by using the method `addEBOMValues`. It receives the list of BOM entries and the workbook as parameters and finds the maximum indentation of the table. This information is needed, as the column number of the values depend on the maximum depth of the hierarchy tree. Then, the `BOM_OD` sheet of the intermediary M-BOM Excel file is inspected in a loop, iterating over the first column of each row. For each component name in the E-BOM, the list of the M-BOM is traversed, looking for the exact component and its position. After a overlapping component has been found, the attribute values are read from the BOM entry and then added into the worksheet. The exact cell is derived from the row, which was found in the loop and the column, which the maximum depth of the earlier search provided. In figure 4.4, the Excel table on the right is the resulting table. Conversely, the `BOM_CD` sheet is also generated via the `createCDSheet` method. First, the `extractODBOM` method is used to get the E-BOM for the object diagram representation, which is subsequently transformed into the class diagram representation list, using `convertToCD`. This list of BOM entries is then used to create the class diagram sheet.

Now the user can proceed to fill out all empty spaces. The resulting table is a complete M-BOM list without missing components and the hierarchical structure from the BOP. The functions one to four, can again be performed on that table, because the M-BOM is basically a BOM file with both `BOM_OD` and `BOM_CD` sheets. From the users' perspective, the option six is selected and the paths for the Excel file in which the M-BOM is to be stored and for the BOP and E-BOM files are requested. After the user inputs the paths, the Excel file selected by the user contains the M-BOM.

Chapter 5

Conclusions and Future Work

The system implemented to automate the generation of the M-BOM using the BOP and E-BOM is possible and does fulfil its purpose. Having all synchronisation processes automated, potential errors are eliminated, due to the fact that in real life the part lists are much larger and contain significantly more information than the examples provided in the thesis. It removes a notable amount of workload which would otherwise arise in the synchronisation processes. Through the incorporation of the diagram implementations, the lists are visualised and provide the potential for further expansions to the program. The ability to assign different attributes to each class is another advantage over conventional part lists. The solution presented simultaneously also enables the creation of a class diagram from an object diagram specification, without the loss of information, which are the values for the attributes.

Having stated the perks of the system, it does still provide the possibility of expanding the program, in terms of optimisation, flexibility, and functionality.

First, regarding the implementation of the program there are some points that can be improved upon. Currently, the BOM synchronisation with the class diagram is fairly static, as the attributes weight and id are hardcoded. In practice however, there are many more attributes that define the part. It would be imaginable to have instead of using hardcoded attributes, to use a dynamic list of attributes which are then fed into the attributebuilder in the cd4analysis language. Other data types such as the identifier of the attributes are also not dynamic. A separate list that contains the identifiers of each attribute from the attribute list is a possible solution.

As already mentioned in the previous chapter, the interface between the BOP and the E-BOM is not optimal. The main purpose of the automated system is to eliminate errors from manual tasks. Nonetheless, the creation of BOPs from E-BOMs are not included in this version. It is not part of the implementation so far, because the BOP only partly uses the E-BOM and mostly adds new information, that requires user input. As user input is unavoidable in this phase, for now nothing is in place to connect the E-BOM and the BOP. The check, whether the E-BOM is up to date currently acts as an intermediate solution. It is imaginable to implement a function in the future that makes sure that the changes in the E-BOM are also taken into consideration in the BOP. Additionally, the generation of the activity diagram from the Excel table is currently limited. It provides only the creation of a linear activity diagram without parallel, decision nodes, and other functionalities usually offered by the activity diagram. The table representation is in this

case not an ideal solution, as the modeling of a complicated activity diagram in a table is not easily achievable. For users who cannot implement the activity diagram however the addition of an automated creation of a activity diagram with all its possible forms would be imperative.

Moving on from the implementation, expansions to the diagram languages could also improve the system significantly. Currently the cardinalities of the associations cannot be specified further than just one or more than one. In most cases in production however, the cardinalities are usually represented in more discrete numbers. For example, a car has exactly four doors. In the current implementation, this is covered by having the attribute quantity for each part. This is a possible solution, but it is not optimal, because as of now, not all functionalities of the class diagram representation are fully taken advantage of. Furthermore, the adlanguage does not currently support the import of the library in Gradle, which increases the additional work needed to make use of the library. If the dependency can be formed via gradle like the other diagram libraries, the user would only have to run the gradle.build once and would be set. As shown in the demonstration chapter, the installation process is labor heavy and not friendly towards users that are not familiar with git or Gradle. Therefore, another expansion would be to create a program with a graphical user interface that automatically gets the latest version and updates of the program and automatically initialises all components. This would go in line with the intention of having minimal user input needed to get from the start to the desired end-product.

Another aspect arising due to the program being in its early stages is, that runtime and optimisations did not have the highest priority during development. The aim in this thesis, was to get all the functionalities running correctly and at a reasonable speed. Documentation in the code and inherent code style was also kept consistently throughout the creation of the code. Even so, it is possible that faster runtimes can be achieved by adjusting the algorithm design, which would be advantageous especially with larger data sets and devices with less advanced specifications. The design of the new data structures introduced can also be improved upon in future works.

The solution provided to inhibit information loss in the creation of the class diagram from the object diagram, deserves proper analysis an review. If a future user decides to expand the tool, by using another representation to store and output data, other than Excel spreadsheets, this aspect should not be forgotten, as it provides flexibility in the development of the BOMs. A possible alternative to the dual spreadsheet solution, would be to store the data in another extension. Still, the use of said extension is unavoidable, because the class diagram cannot hold both the attributes and the attribute values. In terms of using spreadsheet as the extension, the solution provided can be regarded as one of the optimal solutions. One could consider having both representations in a single spreadsheet, but that could create unnecessary complications during the implementation.

All in all, a solution for two fundamental problems in the product planning and the production are introduced in this thesis: First, the unified modeling of component lists and production processes, based on UML diagrams and tabular representations, tackled the problem of having arbitrary representations of the data. This provides a consistent, stable model for the relevant documents, resulting in a facilitated data management system. Even though multiple representations are introduced for a single document, flexibility is preserved by the automation functions provided by the tool. The user is only obliged to create a single model and can then automatically generate all subsequent representations.

Another advantage, the modeling of BOMs through class diagrams provides, is the ability to set unique attributes for each component, which the list representation is not able to.

Secondly, the tool guarantees the correctness of the data, with the automation functions in the tool. The user creates a single file and all other files are automatically generated, which eliminates any potential for errors. The usage of the does not require any profound programming knowledge. This means that, in addition to the flexibility, uniformity, and the guarantee of correctness, the tool also offers a simplicity in its usage. Hence, the communication between instances involved in the life cycle of a product, which was previously inconsistent and error prone, is unified and guaranteed to be without error, by the modeling and the tool. Specifically, this means that the engineers do not have to hassle with synchronising the component lists manually and discussing about the representation of the data. Additionally, wrong information in the M-BOM is eliminated, meaning no surplus or deficit of components exist, which could have resulted in the delay of production and a financial burden on the company. The employers of this tool are thus guaranteed to save time, effort, and monetary properties, which are all in the highest interest of the business.

Bibliography

[CLL97] Sheng-Hung Chang, Wen-Liang Lee, and Rong-Kwei Li. Manufacturing bill-of-material planning. *Production Planning & Control*, 8(5):437–450, 1997.

[ERZ14] Martin Eigner, Daniil Roubanov, and Radoslav Zafirov, editors. *Modellbasierte Virtuelle Produktentwicklung*. Springer Vieweg, 2014.

[GLRR15a] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering tagging languages for DSLs. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 34–43, 2015.

[GLRR15b] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering tagging languages for DSLs. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 34–43, 2015.

[HW91] H.M.H. Hegge and J.C. Wortmann. Generic bill-of-material: a new product model. *International Journal of Production Economics*, 23(1):117–128, 1991.

[KAB17] Jens Kiefer, Sebastian Allegretti, and Theresa Breckle. Quality- and Lifecycle-oriented Production Engineering in Automotive Industry. *Procedia CIRP*, 62:446–451, 12 2017.

[Lin16] Udo Lindemann. *Handbuch Produktentwicklung*. Hanser, 2016.

[Lit12] Matthew Littlefield. The Evolution of MOM and PLM: Enterprise Bill of Process. https://blog.lnsresearch.com/bid/141670/the-evolution-of-mom-and-plm-enterprise-bill-of-process, 03 2012.

[MG17] Object Management Group. OMG Unified Modeling Language – Version 2.5.1. https://www.omg.org/spec/UML/2.5.1, Dec 2017.

[OyJ02] C. Ou-yang and T. A. Jiang. Developing an Integration Framework to Support the Information Flow Between PDM and MRP. *The International Journal of Advanced Manufacturing Technology*, 19(2):131–141, 2002.

[PLV14] Merja Peltokoski, Mika Lohtander, and Juha Varis. The role of Product Data Management (PDM) in engineering design and the key differences between PDM and Product Lifecycle Management (PLM). 04 2014.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*, volume 2nd Edition. Springer, 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*, volume 2nd Edition. Springer, 2012.

[SSHK15] Martina Seidl, Marion Scholz, Christian Huemer, and Gerti Kappel. *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Springer Publishing Company, Incorporated, 2015.

[STT+18] Johannes Stoldt, Thies Uwe Trapp, Stefan Toussaint, Marian Süße, Andreas Schlegel, and Matthias Putz. Planning for Digitalisation in SMEs using Tools of the Digital Factory. *Procedia CIRP*, 72:179–184, 2018. 51st CIRP Conference on Manufacturing Systems.

[TY09] Yoshio Tozawa and Mikio Yotsukura. Integration of Bills of Material towards a Communication Tool. In *2009 WRI World Congress on Computer Science and Information Engineering*, volume 5, pages 446–450, 2009.

[www10a] MontiCore website `http://monticore.org/`, june 2010.

[www10b] Software Engineering website `http://www.se-rwth.de/`, june 2010.

[www21] GitHub repository `https://git.rwth-aachen.de/monticore/cd4analysis`, june 2021.

[XLHM02] X.B., X.W. Liu, Y. Huang, and Ma. *Computer Integrated Manufacturing Systems*, 8(12):983–987, 01 2002.

[XXH08] H. Xu, X. Xu, and Ting He. Research on Transformation Engineering BOM into Manufacturing BOM Based on BOP. *Applied Mechanics and Materials*, 10-12:99–103, 01 2008.

[Y.12] Jaykumar Y. Concept and Evolution of PLM. *International Journal of Applied Information Systems*, 4:25–28, 09 2012.

[ZCXZ07] Shifan Zhu, Dongmei Cheng, Kai Xue, and Xiaohua Zhang. A unified bill of material based on step/xml. In Weiming Shen, Junzhou Luo, Zongkai Lin, Jean-Paul A. Barthès, and Qi Hao, editors, *Computer Supported Cooperative Work in Design III*, pages 267–276, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.