Eindhoven University of Technology
Department of Mathematics and Computer Science

Master’s Thesis
Formalizing Material Flow Diagrams

by
R.J. Bijl

Supervisor: dr. ir. T. Verhoeff
Tutor: ir. M.F. van Amstel

Eindhoven, April 28, 2010
Preface

During the 2008 fall-semester of the Computer Science and Engineering master program, I followed the seminar of the Software Engineering and Technology (SET) group. All students had to write and present two papers on two different subjects; the first one was on Attribute Grammars, and my second subject was Model-Driven Engineering (MDE) [4]. I really liked the concepts of MDE and decided pretty soon after I finished the seminar, that I wanted to do my graduation project in that field.

In August 2009, I visited Mark van den Brand, the head of the SET group, to talk about a graduation project. He suggested a project that two PhD candidates in the SET group had wrote a proposal for. After a meeting with one of those PhD candidates, Marcel van Amstel, I decided to take on the project. The initial project was to formalize Material Flow Diagrams, a sort of building plans used at Vanderlande Industries. We decided that this would be a good starting point for my thesis, with enough directions to expand in, if the initial goal would be achieved before the end of my graduation period.

First of all, I would like to thank Marcel, who became my tutor, for his day to day guidance and my supervisor, Tom Verhoeff for the weekly meetings and insightful discussions. Furthermore, I thank Jacques Verriet for his help with the two use cases, the Software Engineering and Technology group for the daily lunches and the tips they gave me whenever I got stuck somewhere, and, of course, my family and friends, for their support during the course of my graduation.

A special thanks goes out to the Systems Engineering group of the Department of Mechanical Engineering, especially Peter Thijs, without whom I probably would have never been able to create the graphical editor.
Abstract

This thesis presents the process of formalizing Material Flow Diagrams (MFDs). MFDs describe an initial topology for Material Handling Systems (MHSs). However, due to their informal nature, there is no automated process to use the information in MFDs in following steps of the design process. Formalizing these MFDs into Formal Material Flow Diagrams (FMFDs) is a first step in introducing a Model-Driven Engineering (MDE) process, centered around FMFDs. Introducing such an MDE process can automate certain steps in the design process, making these steps more efficient and less error-prone.

To accomplish this, first a metamodel for FMFDs is created, based on an analysis of existing MFDs. This metamodel consists of an abstract framework, that can be made concrete by the use of libraries. This way it is possible to have metamodels that only take into account a sub-domain of MHSs. Based on this metamodel, a graphical editor is constructed to create FMFDs in an easy way.

Once all prerequisites to create FMFDs are provided, it is time to create model transformations and code generators to connect the FMFDs to other development aspects of MHSs. From an FMFD, both an overview of the structure of High-Level Controller software, as well as input for a warehouse simulator are generated.

Finally, some thought is given to the possible evolution of the metamodel, and the consequences that this has for all the entities that conform to the metamodel, i.e., the models, graphical editor, constraints and transformations.
# Table of Contents

1 Introduction .......................................................... 1
   1.1 The Problem Domain .............................................. 1
   1.1.1 Model-Driven Engineering .................................... 1
   1.2 The Project .......................................................... 2
   1.2.1 FALCON .......................................................... 2
   1.2.2 Material Flow Diagrams ........................................ 2
   1.2.3 Problem Statement .............................................. 2
   1.3 Organization .......................................................... 3

2 Domain Analysis ...................................................... 5
   2.1 A Material Flow Diagram .......................................... 5
   2.2 The Analysis .......................................................... 6
   2.2.1 Entities ............................................................ 6
   2.2.2 General Structure .............................................. 7
   2.3 Conclusion ............................................................ 7

3 A Metamodel for FMFDs ................................................. 9
   3.1 Metamodels in General .............................................. 9
   3.2 Considerations ........................................................ 10
   3.2.1 Level of Abstraction ........................................... 10
   3.2.2 Hierarchy .......................................................... 14
   3.2.3 Connections ........................................................ 14
   3.2.4 Constraints ........................................................ 16
   3.3 Even more abstraction: The Abstract Metamodel ................. 16
   3.3.1 The Abstract Metamodel ........................................ 16
   3.3.2 Libraries ........................................................... 16
   3.4 The Final Metamodel ................................................. 17
   3.4.1 The Abstract Metamodel ........................................ 17
   3.4.2 Libraries ........................................................... 17
   3.4.3 The Concrete Metamodel ........................................ 19
   3.5 The Eclipse Modeling Framework .................................. 20
   3.5.1 Ecore ............................................................... 20
   3.5.2 Tooling ............................................................. 20
   3.5.3 Constraints ........................................................ 21
   3.6 Related Work ........................................................... 21
# Table of Contents

4 Creating a Graphical Editor for FMFDs ........................................ 23
  4.1 How to Generate a Graphical Editor ...................................... 23
  4.2 Availability of Tooling ...................................................... 24
    4.2.1 MetaEdit+ .............................................................. 24
    4.2.2 Microsoft Visio ...................................................... 25
    4.2.3 GME ................................................................. 25
    4.2.4 GMF ................................................................. 26
    4.2.5 Some More Options .................................................. 27
    4.2.6 Conclusion ............................................................ 27
  4.3 Creating an Editor Using GMF .............................................. 27

5 Fun with Models ............................................................... 29
  5.1 Model-to-Model Transformations .......................................... 29
    5.1.1 ATL ................................................................. 30
    5.1.2 Merging the Abstract Metamodel with a Library ................. 30
    5.1.3 Computing Metrics .................................................. 33
  5.2 Code Generation ............................................................ 35
    5.2.1 Xpand ............................................................... 35
    5.2.2 Creating Class Diagrams ............................................ 35

6 Use Cases ................................................................. 37
  6.1 Generating a Framework for High-Level Controller Software ........ 37
    6.1.1 The Controller Structure .......................................... 37
    6.1.2 The Metamodels ...................................................... 38
    6.1.3 The Transformation and Generation Steps ......................... 38
  6.2 Generating Simulator Input ............................................... 39
    6.2.1 The Simulator ....................................................... 39
    6.2.2 The Approach ........................................................ 40
    6.2.3 The Metamodels ...................................................... 41
    6.2.4 Creating Models ..................................................... 42
    6.2.5 Generating the Input ................................................ 43

7 Looking Ahead: Metamodel Evolution ...................................... 45
  7.1 Metamodel Evolution in General ......................................... 45
    7.1.1 Dealing with Metamodel Evolution ................................ 45
  7.2 Evolution in the Libraries ................................................ 46
    7.2.1 The Existing Models ............................................... 46
    7.2.2 The Graphical Editor ................................................. 48
    7.2.3 The Constraints ....................................................... 48
    7.2.4 The Existing Transformations ..................................... 49
  7.3 Related Work on Model Evolution ........................................ 49
# Table of Contents

## 8 Conclusions
- 8.1 Recommendations for Further Research .............................................. 52
- 8.1.1 Metamodel Evolution ................................................................. 52
- 8.1.2 Automatic Generation of the Graphical Editor .......................... 53
- 8.1.3 Automatic Generation of a Test Suite ........................................ 53
- 8.1.4 Combining FMFDs with behavioral models ................................. 53
- 8.2 Acknowledgements ............................................................................. 54

## Bibliography .......................................................................................... 55

## A Instantiations of the Metamodel in Development .............................. 59
- A.1 Level 1 ............................................................................................... 60
- A.2 Level 2 ............................................................................................... 61
- A.3 Level 3 ............................................................................................... 62

## B List of Constraints on the Metamodel ................................................. 63

## C An Introduction to the Graphical Modeling Framework .................... 65
- C.1 The metamodel .......................................................... 65
- C.2 Let’s get started ................................................................. 65
- C.3 The Tooling Definition Model .............................................. 66
- C.4 The Graphical Definition Model .............................................. 67
  - C.4.1 The Graph element ........................................................... 68
  - C.4.2 The Edge element ........................................................... 69
  - C.4.3 The Vertex element .......................................................... 70
- C.5 The Mapping Model ........................................................................ 71
- C.6 Generating the Diagram Code ...................................................... 72
  - C.6.1 Changing and regenerating .............................................. 73
- C.7 Extensions ....................................................................................... 74
  - C.7.1 More hierarchy ................................................................. 74
  - C.7.2 Generation of Identifiers .................................................... 75
  - C.7.3 Custom Images ................................................................. 75
- C.8 Common pitfalls ............................................................................... 76
- C.9 Conclusion ....................................................................................... 76

## D ATL Code .............................................................................................. 79
- D.1 The Merge Transformation ......................................................... 79

## E Deliverables .......................................................................................... 83
- E.1 The Metamodel .............................................................................. 83
- E.2 The Graphical Editor ............................................................... 83
- E.3 The Merge Transformation .......................................................... 84
- E.4 The Other Model Transformations ............................................... 84
- E.5 The Generators ............................................................................... 84
- E.6 Use case: Generating Simulator Input .......................................... 84
Chapter 1

Introduction

In this chapter, the problem domain is described, as well as the context of the graduation project. Next, the initial research problem is stated. The chapter ends with an overview of the contents of the remaining chapters.

1.1 The Problem Domain

Throughout the years, the level of abstraction needed to construct programs efficiently has increased. These days, object-oriented programming [30] is used to create this abstraction. But this new technology also introduced new problems.

For example, these object-oriented languages require developers to pay such close attention to numerous tactical imperative programming details, that they often cannot focus on important architectural issues, such as system-wide correctness and performance [40]. And since most applications are still being written and maintained manually, key issues such as deployment and quality assurance take a lot of time and effort.

1.1.1 Model-Driven Engineering

One of the approaches to address this platform complexity, is to develop Model-Driven Engineering technologies. Model-Driven Engineering (MDE) [40] is a software engineering discipline in which models play a central role throughout the entire development process. Models are created to abstract from the specific implementation details. MDE technologies with a greater focus on architecture and corresponding automation yield higher levels of abstraction in software development. This abstraction promotes simple models with a greater focus on problem domain.

From these models, sometimes complete source code can be automatically generated, and sometimes they enable the automatic creation of frameworks, that have to be filled in manually. To this end, MDE combines Domain-Specific Languages with transformation engines and generators.

A Domain-Specific Language (DSL) is a small, usually declarative, language that offers expressive power focused on a particular problem domain [10]. In many cases, DSL programs are translated to calls to a common subroutine library and the DSL can be viewed as a means to hide the details of that library. Because of this, the modeling of problems in a DSL is easier than modeling them in generic modeling or programming languages, because the (irrelevant) details get abstracted away. So, using a DSL, someone with good knowledge of one domain would be able to create a reasonable model, without too much knowledge of modeling languages in general.

DSLs are defined through metamodels, capturing relationships between concepts in a specific domain and specifying the most important semantics and constraints belonging to those domain concepts.
Transformation engines and generators take the constructed model and create artifacts, e.g., source code. The completeness of such artifacts can range from system skeletons to complete deployable systems. Transformation engines and generator make sure that the aspects of the domain, as described by the metamodel, are properly reflected in the generated implementation. Combining these two ideas, leads to the creation of models that closely adhere to the domain in which they will be used, and the generation of artifacts that stick to the models as closely as possible. This makes the created software extremely domain specific, but, because of that, its development process is also very efficient.

1.2 The Project

1.2.1 FALCON

The project described in this thesis is part of the FALCON project [14]. FALCON stands for ‘Flexible Automated Logistics Concepts’ and is a project hosted by the Embedded Systems Institute (ESI). The main industrial partner is Vanderlande Industries, the other partners are Eindhoven University of Technology, Delft University of Technology, University of Twente, Utrecht University and Demcon.

ESI is an institute committed to extending knowledge about embedded systems. It has the explicit aim of making this knowledge publicly available. Vanderlande Industries is a Dutch company that specializes in Material Handling Systems (MHSs), and designs, constructs, installs and maintains baggage-handling, warehousing and express parcel systems.

The FALCON project is aimed at Distribution Centers (DCs), and focusses in particular “on the optimization and decomposition of global requirements concerning system performance, reliability, and cost using a model-driven approach. Starting with high-level system models, system models will be created for different design abstraction levels to analyze and guide the (de-)composition and propagation of design requirements over system components” [14]. It aims at designing the “Distribution Center of the future”, in which every process (hardware- and software-wise) is automated as much as possible.

For Eindhoven University of Technology, the Software Engineering and Technology (SET) group handles the software engineering part of the project, and focusses in particular on “model-based engineering to generate proper control software”.

1.2.2 Material Flow Diagrams

The project in this thesis focusses on Material Flow Diagrams (MFDs). Used at Vanderlande Industries, MFDs are a rough, visual representation of the structure of a MHS, commonly created in Microsoft Visio [21]. An MFD contains the elements that are part of the MHS, and the way these machines are connected to each other. Due to their informal nature, no information that can be used in the course of the design process for MHSs, can be automatically extracted from them. However, it would be a waste to only informally use the information in an MFD, to have to re-implement this information in following steps of the design process. So it would be interesting to see how MFDs could be formalized, i.e., structured in such a way that certain post-process steps could be automated. This would also fit nicely into the goals of the SET group.

1.2.3 Problem Statement

At the beginning of this project, the following problem statement was defined:

“Can we formalize Material Flow Diagrams, by constructing a metamodel that captures their syntax and semantics?”
1.3 Organization

The bigger context behind this problem statement is the introduction of a Model-Driven Engineering process, centered around MFDs. Despite the fact that an MFD is just a rough sketch of reality, it does contain all component- and hierarchal-information needed to create, for instance, high-level controller software. However, to generate these kind of artifacts from an MFD, the MFD needs to be formalized. To perform this formalization in an MDE approach, the following steps will be taken:

- **domain analysis**: what kind of concepts play a role in the domain, and how are these concepts related.
- **construction of a metamodel**: capturing the concepts, found in the domain analysis, into a metamodel, defining the syntax and semantics of the models.
- **construction of a graphical editor**: creating a tool to graphically construct models conforming to the metamodel.
- **construction of transformations/code generators**: creating the tools needed to generate artifacts from the created models.

To avoid confusion, from here on the old/existing MFDs will be called IMFDs (for Informal MFDs), whereas the new, formal MFDs (the models conforming to the MFD metamodel) will be called FMFDs.

Each of the aforementioned steps will be executed in one of the chapters to come.

1.3 Organization

The rest of this thesis is structured as follows. In Chapter 2, the domain analysis is described, which leads to the creation of the metamodel for FMFDs in Chapter 3. A graphical editor, based on the metamodel, is constructed in Chapter 4, while in Chapter 5, model transformations and code generators are developed. In Chapter 6, two use cases are presented, showing an application of the MDE techniques in practice, and Chapter 7 addresses evolution of the metamodel. Finally, a conclusion and proposal for future work is given in Chapter 8.
Chapter 2

Domain Analysis

In this chapter, an analysis of the problem domain will be performed. This analysis will be executed by looking closely into a few IMFDs provided by Vanderlande Industries. It will reveal the key entities that the domain consists of, as well as the connection between these entities. This will then be the driving force behind the development of the metamodel, shown in the next chapter.

2.1 A Material Flow Diagram

Figure 2.1 shows an example of an informal Material Flow Diagram (IMFD).

This is an IMFD like the ones that are currently used at Vanderlande Industries. They are commonly created using Microsoft Visio [21], and it are these IMFDs that need to be formalized. While Figure 2.1 is, of course, only one IMFD, it does capture all the concepts that occur in all the IMFDs that were regarded during the domain analysis. This is why the focus of the analysis will be on this IMFD, and in particular the bottom-left part, shown in Figure 2.2.
To understand what is depicted in Figure 2.2, the legend is shown in Figure 2.3, albeit in Dutch.

Combining the partial IMFD from Figure 2.2 and the legend, gives a good insight in what the important concepts of an IMFD are. In the following section, these concepts will be discussed.

### 2.2 The Analysis

#### 2.2.1 Entities

Looking at IMFD in Figure 2.2, there are two main elements that make up an IMFD: there are Processing Units (the rectangles, representing the units that perform actions on the goods, e.g., sorting, palletizing) and Transport Units (the arrows, representing the conveyers that transport the goods through the system).

Processing Units come in several different types. For an MFD, it is only relevant what type that exactly is. Other attributes are not required at this point of development. Some Processing Units also have Operators controlling them.

Transport Units connect Processing Units. They do have some (optional) attributes:

- buffer positions: the number of buffer positions of a Transport Unit
2.3 Conclusion

- systems capacity: the theoretical capacity of a Transport Unit (in totes per hour)
- operational capacity: the capacity of a Transport Unit in practise

2.2.2 General Structure

Figure 2.1 shows that the system is divided into levels, corresponding to physical floors in a building. Also, every floor has one or more blocks, containing the units. Furthermore, conveyers can run between blocks. These blocks do not necessarily have to be on the same floor.

2.3 Conclusion

The results of the domain analysis can be found in Table 2.1 and Table 2.2.

<table>
<thead>
<tr>
<th>Entities</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Units</td>
<td>Units that perform actions on goods</td>
<td></td>
</tr>
<tr>
<td>Transport Units</td>
<td>Conveyers that transport goods through the system</td>
<td></td>
</tr>
<tr>
<td>Operators</td>
<td>Persons that operate Processing Units</td>
<td></td>
</tr>
<tr>
<td>Levels</td>
<td>Physical floors of the building that contains the system</td>
<td></td>
</tr>
<tr>
<td>Blocks</td>
<td>Floor get divided into blocks</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: The entities that play a role in the MFD domain

<table>
<thead>
<tr>
<th>Connections</th>
<th>Entity 1</th>
<th>Entity 2</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Processing Units</td>
<td>Transport Units</td>
<td>Transport Units can physically connect to Processing Units</td>
</tr>
<tr>
<td></td>
<td>Transport Units</td>
<td>Transport Units</td>
<td>Transport Units can physically connect to other Transport Units</td>
</tr>
<tr>
<td></td>
<td>Operators</td>
<td>Processing Units</td>
<td>Operators can attend a Processing Units</td>
</tr>
<tr>
<td></td>
<td>Operators</td>
<td>Transport Units</td>
<td>Operators can attend a Transport Units</td>
</tr>
</tbody>
</table>

Table 2.2: The connections that play a role in the MFD domain

In the next chapter, the metamodel for Formal Material Flow Diagrams will be created, guided by the elements of these tables.
Chapter 3

A Metamodel for FMFDs

The second step that needs to be taken when formalizing Material Flow Diagrams, is the creation of a metamodel that describes the structure of the Material Handling Systems created by Vanderlande Industries. This is done based on the domain analysis described in the previous chapter. This metamodel provides a formal description of what an FMFD should look like, as well as the means to create such a formal representation for all the MHSs that already exist. On these formal representations (or models), various validation-checks and other operations can be performed. These will be described in the following chapters.

In the rest of the chapter, it is first shown what a metamodel exactly is. After that, the factors to consider during the design of the metamodel are described, and finally, the final version of the metamodel for FMFDs is presented.

3.1 Metamodels in General

“A metamodel makes statements about what can be expressed in the valid models of a certain modeling language” [42]. It formalizes the structure and information needed to describe the artifact that is being modeled. Figure 3.1 shows the relationship of the metamodel to the other objects in the MDE paradigm. Layer $M0$ is an object in the real world, the object that needs to be modeled (in this case, an MHS). Layer $M1$ is a model that describes this real world object. The structure of this model is described by layer $M2$, the metamodel. This metamodel is the modeling language that needs to be created. Finally, layer $M3$ describes the structure of the metamodel, as well as its own structure. It can be seen as the model of the modeling language, and is called the meta-metamodel.

![Figure 3.1: The four layer architecture, based on [17]](image-url)
What follows now are the most important aspects that need to be taken into consideration while creating the metamodel for FMFDs.

The examples of metamodels given in this chapter, are drawn according to the UML Class Diagram standard. UML, the Unified Modeling Language, is a standardized general-purpose modeling language in the field of software engineering, and is maintained by the Object Management Group [33]. For more information on UML, the reader is referred to the official UML manual [36]. For now, it suffices to state that in the following diagrams, the blocks are classes of the metamodel and the lines with diamonds on one end are containment relations. This containment is a way to create hierarchy in models. This means that the containee-class (the one that is not on the side of the diamond) is contained by the container-class, and if the container-class gets deleted, so do the containee-classes. The other lines are reference relations, either unidirectional (the ones with an arrowhead), or bidirectional (the ones without an arrowhead).

### 3.2 Considerations

#### 3.2.1 Level of Abstraction

One of the first aspects that needs to be taken into consideration when designing a metamodel, is the level of abstraction that needs to be achieved. It must be defined which properties are needed in the metamodel, and which are not. The focus is both on capturing the structure of an IMFD (by either capturing its geometry, or its topology), and trying to group certain aspects of the IMFD together. Combining these two aspects together, gives the possibilities (division into levels of abstraction) as shown in Table 3.1.

<table>
<thead>
<tr>
<th>Structure</th>
<th>Geometry</th>
<th>Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding Components</td>
<td>Level 1</td>
<td>Level 2</td>
</tr>
<tr>
<td>Grouping Components</td>
<td>Level 3.1</td>
<td>Level 3.2</td>
</tr>
<tr>
<td></td>
<td>Level 4.1</td>
<td>Level 4.2</td>
</tr>
</tbody>
</table>

Table 3.1: The different levels of abstraction

For every level, an example of a suitable metamodel is given, the pros and cons of that abstraction level are given and (a part of) an FMFD conforming to that metamodel is shown. To do so, the IMFD given in Figure 3.2 is used as a running example. This IMFD shows a part of an MHS: the arrows indicate conveyers, the two aisles at the top are Miniloads (storage units), and the two blocks are Workstations.

Figure 3.2: The IMFD of the Material Handling System that will be modeled in the next sections

3.2 Considerations  

**Level 1**  On the lowest level of abstraction almost all data gets modeled, even the position the elements have in the system (based on their position in the IMFD). The geometry of the IMFD is captured as closely as possible. The metamodel looks like Figure 3.3, assuming for now that all processing units are shaped like rectangles (this is again based on the IMFDs). It contains some notion of hierarchy, as it divides the FMFD into Levels (physical floors of a building), that contain Blocks, containing the Units. Every Unit is either a Processing Unit or a Transport Unit. A Unit can also have an Operator controlling it. These concepts are the ones found in the previous chapter.

Furthermore, there is an auxiliary datatype MFDCoord, which contains a coordinate. These coordinates are use to store the top-left and bottom-right corner of a Processing Unit, as well as the starting position of a Transport Unit. Since these coordinates are stored in the model, the position of the Units is fixed.

![Figure 3.3: The first level; including position information](image)

The advantage of this approach is that the metamodel is simple. The produced models, however, need to be checked to make sure there is no physical overlap between any of their elements (since the position and size of every element is stored in the model). Also, there is no information about what units are connected to each other (the fact that two units are positioned against each other, does not have to mean that they are connected). This is, however, information that is valuable to have in the metamodel. Connections can be used later on in the project, for instance to compute interesting metrics, or to ease the construction of model transformations (see Chapter 5).

In the end, the metamodel presented here is more a metamodel for IMFDs than it is one for MHSs. Positioning information gets copied directly from the IMFD, and assumptions are made on the shape of units, also based on the IMFD. Besides the fact that an IMFD does not contain any layout information, this also presents a problem if MHSs that do not have an IMFD need to be modeled. So all in all, the metamodel described above is not a one that is suited to model Material Handling Systems with. But for completeness sake, an instantiation of this metamodel (for the MHS described by the IMFD above) is given in Appendix A.
**Level 2** The next step is to abstract from the exact position of a unit, and instead model where the different units connect to each other. Instead of the geometry of the IMFD, the topology is captured. To do this, a graph representation is introduced, for which some extra constructs are needed:

- Every Transport Unit gets cut into sub-units (Pieces), running between Processing Units and/or Intersections, which are connections between two Transport Units.
- Every Intersection becomes an entity of its own, to be able to represent it as a vertex.

An MHS can now be represented in the following way: An MHS $T$ is represented by a graph $G = (V, E)$, where $V = \{\text{all Processing Units}\} \cup \{\text{all Intersections}\}$ and $E = \{\text{all Pieces}\}$.

The created metamodel is depicted in Figure 3.4. A Transport Unit consists of at least one Piece and zero or more Intersections. A Piece has two endpoints, called Terminals. Every Piece represents a piece of a conveyer, and runs between two endpoints (Terminals), which are either Intersections (places where two conveyers connect to each other) or Processing Units.

![Figure 3.4: The second level; including intersections and pieces](image)

The advantage of this approach is that known graph algorithms [9] can be applied to the FMFD. These algorithms have been proven to work and can, for example, check for the existence of cycles in the graph (which could indicate circular dependencies). The disadvantages are the overhead that gets created when introducing Intersection objects, and the fact that it clutters the resulting models. This is shown in Figure A.2 of Appendix A, where the top part of the IMFD from Figure 3.2 has been modeled.

While in theory, this graph representation idea provides extra functionality (recall the graph algorithms), the disadvantages stated above are too serious to continue along this path. But as stated before, information about connections is something that is useful to have in the metamodel. In Section 3.2.3 it is discussed how to model connections efficiently.

**Level 3** While studying the IMFDs provided by Vanderlande Industries, some recurring patterns emerged, e.g., that an aisle always has two connected conveyer belts; one outgoing and
one incoming (see Figure 3.5). This aisle and its two conveyer belts can be grouped into one component.

It may be useful to incorporate these components into the metamodel. This way, one could abstract from the contents of the component, and discuss properties that hold for the entire component. Such a Component can be added to the metamodel with little overhead, by adding an extra Component class, containing the Unit class. This grouping of units can be applied to both levels of abstraction mentioned in the paragraphs above, as shown in Figure 3.6.

The model resulting from this metamodel can be seen in Figure A.3 in Appendix A. Note that in this example, the model from Level 1 is extended with a component layer. Extending the Level 2 metamodel would go in a similar way.

**Level 4** Another observation is that a group of the aforementioned components often occur together. This is also shown in the example of Figure 3.5, where all components get goods from and put goods on, one and the same Transport Unit. This could be used to decrease the size of the models.

To accomplish this, two optional attributes are added to the Component class, i.e., ‘number’, a positive integer that indicates the number of consecutive components, and a link to the transportation unit all of the components utilize.

Note that this is actually a major step in abstraction. In the previous levels, there always was a certain fix on the position of a unit, be it its exact position (Level 1), or its relative position to other units (Level 2, through the connections). If components get grouped together, the specific position information of the separate components gets lost. It is known that they all exit on one and the same transportation unit, but not exactly where. This does not coincide with modeling...
the topology of the system, i.e., with modeling all connections. Since this is something that is desirable in the metamodel, this level of abstraction is one level too high.

3.2.2 Hierarchy

In Chapter 2 it is shown that a certain amount of hierarchy exists in an MHS. This is captured by the structure described in the top half of Figure 3.6. However, this approach has two shortcomings:

- It forces the creation of a ‘Level’ and a ‘Block’ layer. While this was the case in the existing IMFDs, this structure might not exist in all MHSs
- It limits the hierarchy depth to 3, since components cannot contain other components

Since hierarchy is a common phenomenon in software engineering, there is a Design Pattern that deals with it: the General Hierarchy Pattern [30]. This pattern offers “a flexible way of representing the hierarchy that naturally prevents certain objects from having subordinates. You also have to account for the fact that all the objects share common features”. This pattern is shown in Figure 3.7.

![Figure 3.7: The General Hierarchy Pattern](image)

The Pattern consists of three classes. There is the abstract superclass Node, and the two subclasses, SuperiorNode and NonSuperiorNode. The fact that Node is an abstract class is depicted by the italic font of the name. Since SuperiorNode and NonSuperiorNode are subclasses of Node, they both have all the attributes of Node, but the difference is that where NonSuperiorNode cannot have any children, the SuperiorNode can. In fact, it can have both SuperiorNodes and NonSuperiorNodes as children. This way, a tree-like hierarchy structure can be modeled.

In this case, NonSuperiorNode will be called ‘Unit’, SuperiorNode becomes ‘Container’, and Node will be called ‘Component’. Now a Component can be seen as either a Unit, or a Container, containing one or more Components. The metamodel now has all the flexibility it needs, and models with an arbitrary unit depth can be created.

However, this provides some new problems. Appropriate invariants need to be introduced, to make sure no one can model illegal FMFDs. Therefore, the following invariants were created:

- A Container can never contain itself; if the hierarchy would be done as a graph, there would be no cycles in the graph, it would be a tree.
- A Container should always contain at least one Unit, or one other Container, since otherwise the Container would be empty, and redundant.

3.2.3 Connections

The main focus of an MFD lies on the units that compose the system, and thus the possible flow of goods through the system. The way these units are connected to each other, however, play an equally important role. Whether or not units are connected might drastically change the way
the system works, i.e., whether or not one conveyer can transfer goods onto another conveyer. In Section 3.2.1, a way to model connections through a graph structure was shown, but that did not turn out to be a desirable way to do this. Instead, connections will be modeled the following way:

- If a unit can form a connection, this is modeled by letting that unit contain one or more connectors. These connectors can either be of type InConnector or of type OutConnector, and the number of connectors a certain unit can have, is set in the metamodel (also see Figure 3.8(a)).

![Diagram](image)

(a) The connectors as part of the metamodel

![Diagram](image)

(b) The connection as part of the metamodel

Figure 3.8: Connectors and connections

- A connection is not a physical entity, but rather the fact that two connectors are connected. Consider, for example, a plug and a power-outlet. The connection between the two would only exist if the plug was plugged into the outlet. In the metamodel, a Connection has references to exactly one InConnector and exactly one OutConnector, and Connections are contained by the MFD object. This last notion ensures easy access to the Connections (Figure 3.8(b)).

The choice to have a separation between InConnectors and OutConnectors is two-fold. For one, these are the only two viable alternatives there are, in the domain that is taken into consideration (the Material Handling Systems). There are no three-way-connectors, or connectors that are both InConnector and OutConnector. And even if there are, they can be modeled in a way that fits this separation. Also, by modeling the metamodel this way, it can be enforced that a connection always exists of exactly one InConnector and one OutConnector. In other words, an InConnector can only be connected to an OutConnector and vice versa. This decreases the chance that errors will be made while modeling an MHS.

Moreover, when only one object per connection gets created, one risk of redundancy gets eliminated. If there would have references been between connectors, these references would have to be bidirectional, and would require careful maintenance. Deleting/updating the reference in one direction, should be matched in the other direction. Now that there is an object to represent the connection, no such redundancy is created.
3.2.4 Constraints

Apart from the invariants discussed in Section 3.2.2, it is useful to have a way to impose some constraints on the created models. Using constraints, it can, for example, be checked if all elements have valid identifiers. Constraints are also a way to control which connections are allowed, and which are not.

A list of all the constraints on the metamodel can be found in Appendix B.

3.3 Even more abstraction: The Abstract Metamodel

While thinking ahead about ways to maintain the metamodel (which will be the topic of Chapter 7), there was another aspect worth considering when developing a metamodel for MHSs. While the metamodel already describes a specific domain, this domain can be split into subdomains. Based on the different departments inside Vanderlande Industries there is a division into baggage-handling, warehousing, and express parcel, and these can be subdivided into smaller domains themselves. Looking at MFDs from those different subdomains, the only difference is the type of units and connectors each domain contains.

In general, there are two ways to deal with this. Either create one big metamodel that contains all entities, or create an abstract framework. This framework can be made into a concrete metamodel using libraries that contain specialization of the generic classes in the abstract framework.

While the first option is simplest, the second option ensures smaller, cleaner metamodels, specialized in smaller subdomains. Apart from this usability aspect, there is also a research argument into solving this problem this way; in the literature, there is no sign of such an approach being tried before.

This second approach introduces two new artifacts: the Abstract Metamodel and Libraries.

3.3.1 The Abstract Metamodel

The Abstract Metamodel is the framework of this new, dynamic metamodel. It contains the structures described in the previous sections (the hierarchy, the connections, the connectors) and an abstract Unit class and abstract InConnector and OutConnector classes. The Abstract Metamodel is shown in Figure 3.9. The three abstract classes need child classes, describing the available units and connectors, to be able to create FMFDs. These children, that need to be merged into the Abstract Metamodel, are defined in a Library. This way, the Abstract Metamodel gets ‘instantiated’ into a Concrete Metamodel, using the unit- and connector-classes from the Library.

3.3.2 Libraries

A Library for a certain subdomain contains all units that play a role in this subdomain, along with all the connectors, and the containment relations between the units and the connectors. The units and connectors must be merged with the Abstract Metamodel to create a concrete metamodel for the specific subdomain. Furthermore, every Library can contain its own set of constraints and datatypes, aimed at the specific domain it represents. These constraints and datatypes should also be merged with the ones that already exist in the Abstract Metamodel.

From here on, to such a merged metamodel will be referred to as a Concrete Metamodel. For a detailed description of the merge, the reader is referred to Chapter 5. For a insight in how to deal with evolution of metamodels in general, and the Libraries in particular, the reader is referred to Chapter 7.
3.4 The Final Metamodel

3.4.1 The Abstract Metamodel

Figure 3.9 shows the Abstract Metamodel.

![Abstract Metamodel Diagram](image)

It combines all the considerations discussed in the previous sections. The MFD class is the top node, containing everything. There is an abstract Component class that can contain Units and Containers, like shown in the General Hierarchy Pattern. It also contains Connections and Connectors as described in Section 3.2.3. Furthermore, every Component can have one or more Operators. The Model class is added to ease the creation of the graphical editor (see Chapter 4).

Note that although the metamodel does not include any concrete Units and Connectors, one could technically still produce models that conform to this metamodel. However, since such a (finite) model will contain at least one empty container, it will violate one of the constraints set up in Section 3.2.2.

3.4.2 Libraries

Figure 3.10 shows a Library, called Library X. It contains three Units and four Connectors, two InConnectors, and two OutConnectors. These Units and Connectors were chosen based on the IMFD from the beginning of this chapter. The attributes of the Units are chosen such that this metamodel can be used in one of the use cases of Chapter 6. Note that, while in this case all attributes of all Units are equal, this does not always have to be the case. Units can very well have different attributes, and this is why these attributes are not lifted to the abstract parent class.

Library X also contains a custom datatype MFDRole, which is an enumeration of the different roles a Unit can have in the system. This datatype is also added to be used in the use case of Chapter 6, and will be explained in more detail there.

Since the Library itself is a metamodel, it is not possible to oppose constraints on the structure of a Library through a metamodel. The meta-metamodel is also not suited for this (at least not in the way it is described in the four-layer architecture, where the meta-metamodel should also...
conform to itself). There are, however, a few naming conventions that every Library needs to fulfil. The merge transformation relies on these conventions, and will not work if the Library does not fulfil them. The naming conventions are as follows:

- The abstract Unit, Connector, InConnector and OutConnector classes are called: ‘LibUnit’, ‘LibConnector’, ‘LibInConnector’ and ‘LibOutConnector’, respectively.

- The root-class is called ‘Library.%’, where % is the name of the Library

It also would have been possible to parameterize the merge transformation is such a way that there is more freedom in the way the elements of a Library are named. However, we chose to go with the naming convention stated above.

Note that for a Library to be useful, it needs to contain at least one Unit, one InConnector and one OutConnector. Otherwise, the Concrete Metamodel would still not contain enough information to create sensible FMFDs.
3.4.3 The Concrete Metamodel

Figure 3.11 shows the Concrete Metamodel, based on the Abstract Metamodel and Library X. You can see how it combines the metamodels from Figures 3.9 and 3.10 into a full, Concrete Metamodel. Models that conform to this metamodel are FMFDs that can be used in the steps that are described in the following chapters.


3.5 **The Eclipse Modeling Framework**

The implementation of the metamodel is done in the Eclipse Modeling Framework (EMF) [19]. EMF is an Eclipse-based framework for Model-Driven Engineering. It consists of several plugins, used to, e.g., create (meta)models, do Model-to-Model transformations and perform code generation. There are a few reasons to choose EMF over other frameworks. It is open-source, has a big, active user-base, and the support forums are filled with information. Furthermore, but that is a personal reason, most of the research and development in the MDE field, performed in the SET group, is done using Eclipse and EMF. Model-to-Model transformations and code generation will be addressed in chapters to come, but first a look at how (meta)modeling using EMF works.

### 3.5.1 Ecore

EMF started out as an implementation of the MOF [34] specification, but evolved from there. EMF can be thought of as a highly efficient Java implementation of a core subset of the MOF API. However, to avoid any confusion, the MOF-like core metamodel in EMF is called Ecore [12]. All metamodels created using EMF are specified through Ecore, and the Ecore meta-metamodel itself is also specified through Ecore. Recalling the 4-Layer model shown at the beginning of this chapter, the mapping to the implementation domain would be as shown in Figure 3.12.

![Figure 3.12: The Four Layer Architecture, based on [17], with a mapping to the implementation domain](image)

The FMFDs belong to layer $M_1$, and the Abstract Metamodel, the Libraries and the Concrete Metamodel belong to layer $M_2$.

The benefit of having this one meta-metamodel, is that it allows for one formalism in which all model transformations and code generation can be created, for all Ecore-based metamodels. Chapter 5 goes deeper into these two topics.

EMF has multiple ways for users to create their own metamodels. You can import annotated Java or Rational Rose models [39], which get transformed to Ecore models by EMF. You can also created metamodels in a graphical way, through EMF itself.

### 3.5.2 Tooling

When working in a modeling framework, the framework does some things for you. Once the metamodel is created, EMF can generate all Java interface and implementation classes for all the classes in the metamodel, plus a factory and package (meta data) implementation class. This is basically everything that is necessary to create, load and edit models through Java code.

EMF also provides a basic treeview-editor to create models (semi-)graphically. A real, full-blown graphical editor is also possible, but that requires some more work. As stated before, this is the topic of the next chapter.
3.6 Related Work

3.5.3 Constraints

Finally, it is also possible to add the constraints, defined in Section 3.2.4, to the metamodel, using EMF. In EMF, constraints are written down in Check files, which have an OCL-like syntax. OCL, the Object Constraint Language is a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model [35]. The Check files are integrated into the Eclipse project through plug-ins, and it results in a ‘Validate’ button appearing in the editor (both the standard treeview editor, and the (to be developed) graphical editor). Appendix B shows the constraints and their OCL translation.

3.6 Related Work

There already has been some research in the area of metamodel composition, although the goals of that research are different from what has been described here. Ledeczi et al. [27] and Emerson et al. [13] describe an approach to connect two metamodels, describing two different domains. This way, they create a new metamodel that describes both domains.

This differs from the merging of a Library into the Abstract Metamodel, in the way that their approach is based on two existing, complete metamodels. The metamodel merging, described above, is aimed at composing a complete metamodel from two incomplete ones. Also, in the case of Ledeczi et al. and Emerson et al., the metamodels describe different domains. Still, some of the described techniques may be useful. Unfortunately, they are very much aimed towards an implementation in GME.

Lagerström et al. [26] try to merge metamodels by using Bayesian networks. In this case, the same thing holds as with the previous approaches: they merge classes of metamodels together, instead of merging two metamodels, like done in this thesis.

On a slightly different note, France et al. [16] give a method to merge two models into one, by adding merge information to the respective metamodels.
Chapter 4

Creating a Graphical Editor for FMFDs

Now that the metamodel is created, and thus a formal definition to describe Material Handling Systems exists, some tooling needs to be created to actually make models conforming to this metamodel. While EMF provides a treview-editor for free, there are a few reasons not to be satisfied with this editor. For one, it is not at all user friendly. More importantly though, to convince other parties that this new approach is better than their old one, such a treview-editor is nowhere near sufficient. To really convince people, there would need to be a (graphical) editor that they can use as easily as the products they are using at the moment.

In the case of Vanderlande Industries, this current product would be Microsoft Visio, in which you can simply drag and drop all the artifacts you need onto a canvas. To convince the people at Vanderlande Industries that the approach described in this thesis is superior to their current approach, there would at least have to be a decent graphical editor, with drag and drop functionality, to create FMFDs. Without that, no one would even consider replacing Visio with something new, and they would never embrace this new approach.

In this chapter, first a quick look at the ways to create a graphical editor is given. After this, various packages that are available for making such an editor are presented. Their pros and cons are described, and one is picked to proceed with. Finally it is shown how to create an editor, using the chosen package.

4.1 How to Generate a Graphical Editor

There are two ways to create a graphical editor, based on the Abstract Metamodel structure described in the previous chapter:

1. The Abstract Metamodel and Libraries can be merged into a Concrete Metamodel, and the editor can be created from that metamodel.

2. An editor can be created based on the Abstract Metamodel, that can draw different elements based on the Library that is used to create the Concrete Metamodel.

While the latter option may look like the most elegant solution, it is also the hardest one to accomplish. It turns out that a graphical editor cannot be generated without some level of user interaction. Also, the available tools are not really suited for implementation of the second option. For that reason, the first option is chosen.
4.2 Availability of Tooling

Before a graphical editor can be built, a choice in tooling needs to be made. There are a handful of commercial and open-source packages that deal with creating graphical editors from metamodels, and this section contains a comparison-study to find the one that is best suited for our needs.

The comparison is based on the following criteria:

- **Compatibility**: How does the package handle the created metamodel; can it be used directly, or does it need to be implemented it in a different formalism?

- **Flexibility**: What customization options does the package have, e.g., can it use custom images for different units?

- **Creation of editor**: How easy is it to create an editor through the package, can this for instance be automated?

- **Usability of editor**: How usable is the created editor? This is based on criteria like stability, platform (in)dependence and user-friendliness.

- **Validation/Verification**: (How) can the created models be validated, based on the constraints defined in Section 3.2.4?

These criteria are applied on two commercial products (MetaEdit+ and Microsoft Visio) and two open-source alternatives (GME and GMF). MetaEdit+, GME and GMF were included, because those are the products that are used the most at the time of this research. Visio was included because it is the editor currently used at Vanderlande Industries.

4.2.1 MetaEdit+

MetaEdit+ is “a completely integrated environment for building and using your own Domain-Specific Modeling (DSM) solution”[45]. It is proprietary software, owned by MetaCase, a Finnish company specialized in Domain-Specific Modeling. MetaEdit+ comes in two flavors. There is the MetaEdit+ Workbench, in which a DSL can be defined. Note that a metamodel is basically nothing more than a DSL in a graphical form.

The created metamodel can be used by the second flavor of MetaEdit+, the MetaEdit+ Modeler. This modeler generates an editor based on the metamodel provided to it, and provides the user with full modeling-tool functionality: diagramming editors, browsers, generators, etc.

Both versions have built-in repository support, allowing easy sharing of models and enabling different persons to work on the same models.

**Compatibility** MetaEdit+ is based on the GOPRR meta-metamodel (which is an abbreviation of Graph, Object, Property, Role and Relationship). Since EMF is based on the Ecore metamodel, the metamodel created in the previous chapter cannot be used directly. Research has been perform into the interchange of metamodels between MetaEdit+ and EMF [25], but this is hardly a trivial task. The best bet seems to be to re-implement the metamodel in the MetaEdit+ Workbench. Fortunately, this is not very difficult.

Created models can be transformed back to the XML representation of Ecore, using the built-in Model-to-Text transformer. However, such a transformation is not available by default; it must be constructed specifically for the metamodel that the editor is created for.

**Creation of editor** Once the metamodel is created using the Workbench, the Modeler automatically creates the editor conforming to it.
4.2 Availability of Tooling

**Customization** The Workbench allows for the customization of nodes and links (the MetaEdit+ variants of classes and references). From a layout point of view, the user is bounded to the looks and feel of the MetaEdit+ Modeler.

**Usability of editor** The editor is stable and user friendly. The user does not, however, get a stand-alone application and he will always be bounded to the looks and feel of the MetaEdit+ Modeler. While this is not necessarily a bad thing, it does mean that the user will always have to install MetaEdit+ to be able to use the editor, instead of having one executable file to run the editor with. MetaEdit+ currently supports the following platforms: Windows (2000, XP, Server 2003), Linux (i386, kernel 2.2 or later, glibc 2.1.3 or later, X11R6), Mac OS X (10.3.4 or later), HP-UX (11.x or later) and Solaris (5.6 or later).

**Validation/Verification** The Workbench has a rule editor, in which it can be specified what connections are allowed, and what constraints should hold. These rules are loaded into the Modeler together with the metamodel, and are verifiable with a press of a button.

### 4.2.2 Microsoft Visio

Microsoft Visio is a diagramming program for Microsoft Windows that uses vector graphics to create diagrams [21]. It is not an MDE framework, but is taken into consideration because it is the application currently used at Vanderlande Industries. It uses so-called Stencils to define the shapes and connections available for a certain domain.

**Compatibility** Almost none; the metamodel cannot be used, nor can one be defined in another formalism. As said before, Visio is not an MDE application, so that best the user can do, is create Stencils containing the Units and Connections from the metamodel.

One other issues is the file-formats Visio uses. The input and output files contain a mixture of text and binary data, with which a program other than Visio cannot do a whole lot.

**Creation of editor** To create an editor, a Stencil containing all units and connections need to be created. This Stencil must then be loaded into Visio. The creation of the Stencils can be done in Visio itself.

**Customization** The user is stuck to the Visio interface, but as far as the shapes of the units and connections go, there is a lot of variation possible.

**Usability of editor** Creating drawings is no problem, especially for the people at Vanderlande Industries, who have been using Visio for quite some time now. Visio is a Windows-only application.

**Validation/Verification** All allowed connections are embedded in the stencils, but there is no real way to oppose constraints to the drawings that are created using Visio.

### 4.2.3 GME

The Graphical Modeling Environment (GME) is a configurable toolkit for creating domain-specific modeling and program synthesis environments [28]. It is an open-source MDE framework created by ISIS, the Institute for Software Integrated Systems. ISIS is a research organization of the School of Engineering at Vanderbilt University, from Nashville, Tennessee, USA.
GME works in the same fashion as MetaEdit+; using the GMeta component, the metamodel, including constraints and rules, can be defined. Through the GModel component, an editor is generated. It also allows for distributed access.

**Compatibility** The meta-metamodel used in GME is *MetaGME*. In this case as well, research has been done into the conversion from MetaGME into Ecore [5]. This process seems to be a bit more feasible than in the MetaEdit+ case, but again, it is probably best to re-implement the metamodel.

**Creation of editor** As with MetaEdit+, once the metamodel is defined, the GModel component automatically generates an editor.

**Customization** While the user is stuck to the layout and functionality of the GModel component, custom figures can be imported, to represent out Units and Connections.

**Usability of editor** The editor is stable, and works pretty straightforward. It does have some difficulties with hierarchies, forcing the user to ‘zoom’ into every level, without offering the possibilities to see more then two levels of hierarchy at the same time. Furthermore, GME is a Windows-only application.

**Validation/Verification** This is again built-in, based on the constraints defined in the metamodel

### 4.2.4 GMF

The Eclipse Graphical Modeling Framework (GMF) [19] provides a generative component and runtime infrastructure for developing graphical editors based on EMF. If it is possible to create a stable, functional editor through GMF, it would be the most desirable solution. Since GMF is part of EMF, it would work seamlessly with the already created metamodel.

**Compatibility** GMF is an Eclipse plug-in, and uses the same (meta-)metamodels as EMF. So nothing new would have to be defined, and all created models would conform to the metamodel that has already been created.

**Creation of editor** This is notoriously the hard part of GMF. The editor has to be built from scratch, and then all components have to be linked to their respective classes in the metamodel. This is a hard task to get started with, but fortunately, there is some knowledge on the TU/e on doing so. The Systems Engineering Group from the Department of Mechanical Engineering, has developed SCIDE, a full-blown GMF editor for two of their simulation languages [41]. Using their expertise, it will be possible to construct a working, stable GMF editor, without too much difficulties.

**Customization** Having to built an editor from scratch has one upside, and that is that it gives the user a tremendous amount of freedom when it comes to e.g., choosing what the units look like. He is still stuck to the Eclipse interface, but that is only in the outer layer of the editor.

**Usability of editor** Since GMF is an Eclipse plug-in, the editor can be run as a standalone program, only containing the plug-ins it depends on. It can also be exported as an RCP application [31], allowing it to be run on most operating systems. Based on the experience with SCIDE, a stable, user friendly editor is certainly possible through GMF.
Validation/Verification  Validation is done by the same files that are used to perform validation in EMF. These files get inserted into the editor automatically, and validation can then be done with the click of a button.

4.2.5  Some More Options

There are of course several other options that were not taken into consideration, for various reasons. These alternatives include GEMS [49] (the Generic Eclipse Modeling System, a skin over GMF, which hides too much detail), GEF [19] (the Graphical Editing Framework, a predecessor of GMF, which is still used, but inferior to GMF) and Microsoft DSL Tools [8] (which cannot deliver the sort of editor that is required in this case).

4.2.6  Conclusion

In the end, every criteria has been given a rating from the set \{++, +, +/−, −, −−\}, where ++ is very good, and −− is very poor. All results are collected and accumulated in Table 4.1.

<table>
<thead>
<tr>
<th></th>
<th>MetaEdit+</th>
<th>Microsoft Visio</th>
<th>GME</th>
<th>GMF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compatibility</td>
<td>+/−</td>
<td>−−</td>
<td>+/−</td>
<td>++</td>
</tr>
<tr>
<td>Flexibility</td>
<td>+</td>
<td>−</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>Creation of editor</td>
<td>++</td>
<td>+</td>
<td>++</td>
<td>−</td>
</tr>
<tr>
<td>Usability of editor</td>
<td>+</td>
<td>++</td>
<td>+/−</td>
<td>+</td>
</tr>
<tr>
<td>Validation/Verification</td>
<td>++</td>
<td>−−</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>Overall</td>
<td>+</td>
<td>−</td>
<td>+/−</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 4.1: The results of the comparison of graphical editing frameworks

Based on these ratings, Visio can be removed from the list of possible candidates. This is no surprise, since it is indeed not an MDE framework. GME can be seen as a trimmed down, albeit free, version of MetaEdit+: they contain roughly the same elements, but MetaEdit+ does it all just a little bit better. For this reason, GME is removed from the list as well. The choice now is between MetaEdit+ and GMF. The fact that GMF connects closely to the metamodel that has already been implemented in EMF, and a stable, working editor can be build with it, led to the conclusion that this is going to be the package to proceed with. It appears to be the best option.

4.3  Creating an Editor Using GMF

The next step is to actually build the editor, using GMF. This is a somewhat difficult process, and there is not a lot of solid documentation available. Therefore, a small tutorial has been written, to help others in creating such an editor. This tutorial contains the steps that are needed to created and maintain a graphical editor, based on a small metamodel. It can be found in Appendix C. For now, it suffices to say that a graphical editor, for the metamodel created in the previous chapter, has been successfully created. It has basic functionality to create and edit FMFDs, and a screenshot is shown in Figure 4.1.

The white rectangles are conveyers, the ellipses inside them are the connectors. An arrow between two connectors indicates that those two connectors are connected. The grey rectangles are Miniloards, and the white rectangles with bold edges represent Workstations.
Creating an FMFD using the graphical editor results in two files. One the one hand there is the FMFD model, containing all the information about the Units and Connectors in the MHS. This model conforms to the Concrete Metamodel the editor is based on. On the other hand, there is a diagram file; an XML-file that contains the positions of all model elements in the editor.

Figure 4.1: The graphical editor
Chapter 5

Fun with Models

At this point, a metamodel for FMFDs is created. Also, models that conform to this metamodel can be created. The creation of the models is done through the graphical editor that also has been created. But it took a lot of effort to get here, and until now, there are not really any advantages (apart from validity- and consistency-checking) of this Model-Driven Engineering approach. That changes with the introduction of Model-to-Model transformations and code generators. In this chapter, these two different techniques will be explained. In the following sections, their characteristics will be discussed, and a few examples will be given. These examples will be based on the metamodel for FMFDs, defined in the previous chapters.

5.1 Model-to-Model Transformations

First there are the Model-to-Model (M2M) transformations. Say there are two different domains (described by two different metamodels), in which the same data needs to be represented. In the case of Vanderlande Industries, there is for example an FMFD, and a model to run simulations on. Both models conform to different metamodels, but are based on the same real-life artifact; the Material Handling System. If the FMFD already exists, the second model could, of course, be created from scratch. However, since one of the three great virtues of a good programmer is ‘laziness’ [47], there might be a better way to obtain this second model. Maybe there is a way to derive the second model from the first one, transform the first model, as it may. This is what a model transformation is: “A conversion of a given source model in a specific formalism, into a given target model, possibly in a different formalism”. Figure 5.1 shows how model transformations fit into the model/metamodel/meta-metamodel structure from Chapter 3.

![Figure 5.1: Model Transformations (based on [24])](image-url)
The model transformation $M_t$ (conforming to metamodel $MM_t$) transforms model $M_a$ (conforming to metamodel $MM_a$) to model $M_b$ (conforming to metamodel $MM_b$). All the metamodels conform to the same meta-metamodel $MMM$. This meta-metamodel in its turn, conforms to itself. Note that $MM_a$ and $MM_b$ do not have to be different, they could be the same metamodel.

A model transformation can be a powerful tool in developing models that (closely) resemble already existing models. It saves you the time and effort it takes to create multiple new models, since now you can transform one you already have. But model transformations come at a cost.

Model transformations can be seen as pieces of software; they have the same difficulties and life cycle as traditional software. They have to be used by several developers, have to be changed according to changing requirements and should preferably be reused [2]. And the biggest problem of all is that there is usually not a specific formal definition of the transformation. Such a definition is not easy to obtain; defining a complete mapping from all possible source models to target models is not a trivial job. A few attempts to formalize an approach to create such a specification have been made, e.g., by Siikarla et al. [43], but this discussion is outside the scope of this thesis.

### 5.1.1 ATL

The transformations shown in this chapter are created using the Atlas Transformation Language (ATL). ATL is an Eclipse plug-in, and comes with the ATL Integrated Environment (IDE), which provides a number of standard development tools (syntax highlighting, debugger, etc.) that aim to ease development of ATL transformations [24]. ATL is being maintained by a French company called Obeo. This improves the quality and the support of ATL, and is one of the reasons to choose ATL to create the transformations with. There are some alternatives to ATL in EMF, e.g., Procedural QVT and Declarative QVT. QVT (Query/View/Transformation) is a standard for model transformation defined by the Object Management Group. There are, however, some reasons to choose ATL over QVT. One is that ATL is the formalism that is mostly used in the Eclipse community, the other is that we have experience with ATL within our faculty.

Next, two examples of M2M-transformation in ATL will be given. The first one is the merge of the Abstract Metamodel and a Library, as described in Chapter 3. The second one is a small transformation to compute some metrics. This second example shows how you can do some more difficult calculations using auxiliary functions.

### 5.1.2 Merging the Abstract Metamodel with a Library

The first model transformation, is the merge transformation that plays a crucial part of the idea of using different Libraries for different subdomains. The starting point is two metamodels, both conforming to the Ecore meta-metamodel. This model transformation is somewhat unconventional, in the sense that it operates on metamodels, rather than on models. In theory, however, this should not really matter; the model/metamodel relationship is similar to the metamodel/meta-metamodel relationship.

So what does the transformation looks like? The process can be divided into three steps:

1. Copy the entire Abstract Metamodel into the new Concrete Metamodel.
2. From the Library, copy the units, connectors and the references between them into the new Concrete Metamodel.
3. In the Concrete Metamodel, link the units and connectors from the Library to the abstract Unit and Connector classes from the Abstract Metamodel.

**Step 1** is a trivial transformation. All that is needed to be done, is to copy every Ecore element from the Abstract Metamodel to the Concrete Metamodel. An example of such a copy rule is given in Listing 5.1.
5.1 Model-to-Model Transformations

```
rule copyEClass(
  from a : MM!EClass
to p : MM2!EClass{
  "abstract" <- a."abstract",
  name <- a.name,
  eAnnotations <- a.eAnnotations,
  eGenericSuperTypes <- a.eGenericSuperTypes,
  eOperations <- a.eOperations,
  eStructuralFeatures <- a.eStructuralFeatures,
  eSuperTypes <- a.eSuperTypes,
  eTypeParameters <- a.eTypeParameters,
  instanceClassName <- a.instanceClassName,
  instanceTypeName <- a.instanceTypeName,
  interface <- a.interface
}
Listing 5.1: Copying all Ecore EClasses from the Abstract Metamodel (MM) to the Concrete Metamodel (MM1)
```

The reason that the \texttt{eGenericSuperTypes <- a.eGenericSuperTypes} is commented out, has to do with a weird ‘bug’ in ATL. Copying both the \texttt{eGenericSuperTypes} and the \texttt{eSuperTypes} attributes, results in two inheritance links per parent/child relationship. This is a problem that has not been solved yet. However, this work-around seems to be working, without loss of information.

\textbf{Step 2} is almost the same as Step 1, with the exception that the elements that need to be copied, need to be filtered out from the rest. This is done by introducing some helper functions. In the case of the Units, two are needed: one to find the Units (which are the EClasses that have the class called ‘LibUnit’ as parent), and one to find the abstract Unit class from the Abstract Metamodel. These helpers are shown in Listing 5.2.

```
helper context MM1!EClass def : isUnit() : Boolean =
  self.eSuperTypes->exists(i | i.name = 'LibUnit')

helper def : getAbstractUnit() : MM!EClass =
  let allEClass : OrderedSet(MM!EClass) =
    MM!EClass.allInstances()->asOrderedSet() in
  allEClass->
  select(i | (i.name = 'Unit') and
  (i.eSuperTypes->
  exists(i | i.name='Component'))
).last()
Listing 5.2: Two helper functions
```

The first helper checks whether an EClass has the ‘LibUnit’ EClass as its parent. This second helper collects all EClasses from metamodel MM (the Abstract Metamodel) and then filters out the one that has the name ‘Unit’, and has a parent called ‘Component’. This returns a list consisting of one element, on which the \texttt{last()} function will be applied, to extract the desired element (note that the \texttt{first()} function could also be used, since the list only contains one element).

The copy rule now looks like Listing 5.3.
rule copyEClassLibUnit{
  from
  a : MM1!EClass (a.isUnit())
  to
  p : MM2!EClass{
    "abstract" <- a."abstract",
    name <- a.name,
    eAnnotations <- a.eAnnotations,
    --eGenericSuperTypes <- a.eGenericSuperTypes,
    eOperations <- a.eOperations,
    eStructuralFeatures <- a.eStructuralFeatures,
    eSuperTypes <- thisModule.resolveTemp(thisModule.getAbstractUnit(), 'p'),
    eTypeParameters <- a.eTypeParameters,
    instanceClassName <- a.instanceClassName,
    instanceTypeName <- a.instanceTypeName,
    interface <- a.interface
  }
}

Listing 5.3: The copy rule for Library Units

The first helper is used in Line 3. The second one is used in Line 12, in a resolveTemp construction. This construction does the following: it uses the element in its first argument (in this case the abstract unit from the Abstract Metamodel, found by the helper), and takes the element that will be produced from this element, based on the second argument (in this case, the abstract unit that will be part of the Concrete Metamodel). Note that this produced element does not need to exist at the moment the resolveTemp line is executed. ATL will make sure that all elements will have been created and linked correctly, once the transformation has terminated.

Also, the EClasses and EDataTypes from the Library need to become contained in the EPackage of the new Concrete Metamodel. This code is shown in Listing 5.4.

rule copyEPackage{
  from
  a : MM1!EPackage
  to
  p : MM2!EPackage{
    eClassifiers <- a.eClassifiers,
    eClassifiers <- thisModule.getLibClasses()->collect(e | thisModule.resolveTemp(e.debug('Adding Library Class'), 'p')),
    eClassifiers <- thisModule.getLibDataTypes()->collect(e | thisModule.resolveTemp(e.debug('Adding DataType'), 'p')),
    name <- 'FMFD_' + thisModule.getLibraryName(),
    nsURI <- 'http://FMFD_' + thisModule.getLibraryName() + '/1.0',
    nsPrefix <- 'FMFD_' + thisModule.getLibraryName(),
    eAnnotations <- a.eAnnotations,
    eFactoryInstance <- a.eFactoryInstance,
    eSubpackages <- a.eSubpackages
  }
}

Listing 5.4: The copy rule for Library References

In this rule, a new EPackage for the Concrete Metamodel gets created (based on the Abstract Metamodel). The EClasses and EDataTypes from the Library get collected and their resolved objects get linked to this new EPackage.
Finally, Step 3 also requires some creativity. To correctly copy the references from the Library, the rule from Listing 5.5 is needed.

```
rule copyEReferenceLib{
  from
  a : MM1!EReference (a.isMergableRef())
  to
  p : MM2!EReference{
    changeable <- a.changeable,
    containment <- a.containment,
    defaultValueLiteral <- a.defaultValueLiteral,
    "derived" <- a."derived",
    eAnnotations <- a.eAnnotations,
    --eGenericType <- a.eGenericType,
    eKeys <- a.eKeys,
    eOpposite <- a.eOpposite,
    --eType <- a.eType,
    lowerBound <- a.lowerBound,
    name <- a.name,
    ordered <- a.ordered,
    resolveProxies <- a.resolveProxies,
    transient <- a.transient,
    "unique" <- a."unique",
    unsettable <- a.unsettable,
    upperBound <- a.upperBound,
    volatile <- a.volatile
  }
  do {
    p.refSetValue('eType', thisModule.resolveTemp(a.eType,'p'));
    p.refSetValue('eGenericType', thisModule.resolveTemp(a.eGenericType, 'p'));
  }
}
```

Listing 5.5: The copy rule for Library References

The correct values for the eType and eGenericType of an EReference need to be calculated through resolveTemp constructions, in the same way shown above. However, for some reason this does not work directly, and these values need to be set in an imperative section. This anomaly is discussed in Appendix D.1.

### 5.1.3 Computing Metrics

The second example shows the use of helper functions even better. It shows how to calculate some metrics about a given FMFD, and store these metrics in a metrics model. Calculating metrics on a model can, for example, help to determine the quality and complexity of that model [2]. For this transformation, only simple metrics are considered, i.e., metrics that return an integer value.

A metrics model is nothing more than a collections of classes, each containing a metric and its value. The metamodel is shown in Figure 5.2.

![Figure 5.2: The metamodel for the metrics model](image_url)
The transformation to create such a metrics model from an FMFD consists of exactly one matched rule, and a handful of helpers. Listing 5.6 shows the rule, which takes the MFD element from the input model and transforms it into three Metric elements of the target model, each of them containing its own metric. The 'MName' attribute of every Metric element gets a value directly, but the 'MValue' attribute needs to be calculated through a helper.

```
rule fillMetrics{
  from a : MM!MFD to
  q : MM1!Metric(
    MName <- 'NumberOfUnits',
    MValue <- a.getNumberOfUnits()
  ),
  s : MM1!Metric(
    MName <- 'NumberOfContainers',
    MValue <- a.getNumberOfContainers()
  ),
  r : MM1!Metric(
    MName <- 'MaxNestingDepth',
    MValue <- a.getMaxDepth()
  ),
  p : MM1!MetricSuite(
    MId <- 'M1',
    MFDId <- a.MId,
    contains <- q,
    contains <- r,
    contains <- s
  )
}
```

Listing 5.6: The first part of the transformation to compute metrics

One of the metrics is the MaxNestingDepth metric. It calculates the maximum number of Containers that a Unit is contained by. It does so by using the helpers shown in Listing 5.7.

```
helper context MM!MFD def: getMaxDepth() : Integer =
  self.getNestingDepth(0).last()

helper context MM!MFD def: getNestingDepth(i:Integer) : OrderedSet{Integer} =
  self.component->iterate(contain ; depth : OrderedSet{Integer} =
    OrderedSet[ ] | |
    if contain.oclIsTypeOf(MM!Container) then
      depth.union(contain.getNestingDepth(i+1))
    else
      depth.append(i+1)
    endif
  )
```

Listing 5.7: The helpers to calculate the maximum nesting depth

The getNestingDepth helper is a recursive helper that returns a sorted list containing the depth of every Unit in the model. This helper comes in two flavors, one working on the MFD element, and one on Container elements (although only the first one is shown in Listing 5.7).
getMaxDepth helper takes the last (and the biggest) element of this list, which is the maximum nesting depth.

5.2 Code Generation

The second MDE technique is code generation. Code generation is the art of turning a code template into ‘real’ code, based on the properties of a model. Some argue that code generation is also a form of model transformation, since source code can also be viewed as an abstraction (and thus a model) of reality. This code also conform to a metamodel (the syntax of that code). For this reason, code generation is also referred to as Model-to-Text transformations (M2T), and is also classified as a model transformation. So most things stated about Model-to-Model transformations in the previous sections (apart from the fact that the two models must have the same meta-metamodel), also holds for M2T-transformations. But since this discussion has no effect on the actual mechanics of the technique, this is all there is to be said about it. The remainder of this thesis will use the term ‘code generation’.

5.2.1 Xpand

In the Eclipse Modeling Framework, there are several code generation engines, all with their own pros and cons [19]. From this collection, Xpand is chosen, because there is some knowledge about it in the SET group. Xpand is a statically-typed template language and includes an editor which provides features like syntax coloring and error highlighting. It creates and fills files based on templates; one template for every metamodel class.

Suppose that it is interesting to generate a stripped down class diagram of an FMFD, e.g., for demonstration purposes. What follows is a small example of a template used to generate a .dot file, describing such a class diagram. dot is a language to draw directed graphs. It reads attributed graph text files and writes drawings, either as graph files or in a graphics format such as GIF, PNG, SVG, PDF, or PostScript [18].

5.2.2 Creating Class Diagrams

As said before, a complete Xpand template contains a template for every class of the metamodel that will have code generated from it. Listing 5.8 shows the templates for the MFD class.

When you look at an Xpand file for the first time, you immediately notice the French quotation marks, or guillemets (‘≪’ and ‘≫’). Every Xpand command is surrounded by these guillemets. The rest of the text will become output. In Listing 5.8 both the metamodel and an extension file, containing auxiliary functions, are loaded. After that, the main function is defined, linked to the MFD element.

The template creates a .dot file, which is named after the MName attribute of the MFD element. This file is then filled with dot code. For a description of the dot syntax, the reader is referred to the dot guide [18]. Remember that everything between the guillemets will be evaluated by Xpand, and the rest will be written to the file. So the first part (lines 11 through 20) of the template fills the dot file. In lines 21 - 22, all the Container and Unit children of the MFD are collected, using the auxiliary functions from the extension file. Xpand iterates over these children, evoking the correct template for each of them. After that, the connections between the MFD class and its children are added. Finally, all the connection arrows are added to the code, by iterating over all the connections in the model (line 29).
Note that the layout of the resulting .dot file is very poor. This is a limitation of Xpand, since it does not have a pretty printer or a code formatter. This can be solved in two ways; either format the Xpand file in a way that the generated file is formatted correctly (which most likely ruins the readability of the Xpand code), or run the generated code through an external code formatter. In almost all cases, the latter option is preferred over the first one.
Chapter 6

Use Cases

All of the previously presented techniques come into play in two use cases that were performed during the course of the project. The first one exists of a M2M transformation in ATL, and a code generation step through XPand. The second use case uses the Java implementation generated by EMF, to perform both the M2M Transformation, and the code generation programmatically. The location of all the files created for these use cases is given in Appendix E.

6.1 Generating a Framework for High-Level Controller Software

First, a framework for High-Level Controller (HLC) Software will be generated from the structure of an FMFD. However, since a formal description of such a framework is still under development, only an overview image of the structure of the HLC software will be generated. In this section, the structure of such a framework will be described, as well as the steps taken to generate the overview.

6.1.1 The Controller Structure

The idea of the HLC software is to introduce an agent structure [48] to handle various planning aspects, such as delivery and replenishment. This use case is centered around the system shown in Figure 6.1.

Figure 6.1: A Picking Area System
This is a system for case picking. It consists of a General Backup Aisle, containing spare goods for the system, and one of more Picking Areas (PA). These PAs have their own Backup Aisle and one or more Zones, used to pick goods from the PA Backup Aisle.

Every element of the system has its own planning agents, to handle delivery, replenishment, storage and relocation. In this use case, only the first two types of planning agents (or Planners) are considered. These Planners are connected to each other as shown in Figure 6.2:

![Figure 6.2: The connection of the Planners](image)

Through the connections, messages regarding delivery and replenishment will be send. The exact content of these messages is not relevant at this point.

### 6.1.2 The Metamodels

On the one hand, there is the metamodel for FMFDs, which has been thoroughly discussed in the previous chapters. On the other hand, there is the metamodel for HLC models. This metamodel is shown in Figure 6.3.

![Figure 6.3: The HLC metamodel](image)

It is a hierarchial structure as described in Section 3.2.2, where every TreeElement has four planners. The delivery and replenishment planners are connected as shown above.

### 6.1.3 The Transformation and Generation Steps

The structure of a HLC model is based on the hierarchial structure of an FMFD. This means that some more thought has to be put into the design on an FMFD; every Container will become a TreeElement, and will get four Planners.
Once the HLC model has been created, artifacts can be derived from it. As stated before, the ultimate goal is to generate high-level controller software based on the FMFDs. But since the formal description of such software is not yet present, we will settle for an image that shows the structure of such software (since the structure is already defined). This is done by generating dot-code (recall the example in Section 5.2.2), representing an overview image. This code generator is a straightforward traversal of the model. The overview image, resulting from system described above, is a hierarchical system containing one PA. The image is shown in Figure 6.4.

![Figure 6.4: An overview image depicting the HLC structure of the PA example](image)

The ovals depict the different Planners, the arrows are the connections between the Planners. Note the somewhat random layout of the image. This is caused by the internal algorithms of GraphViz [44], the program the interprets dot files, and creates images from them.

6.2 Generating Simulator Input

In a second, a bit more elaborate use case, input for a simulator will be generated, based on an FMFD. This simulator calculates performance measures for the system modeled in the FMFD. In this section, first the simulator will be introduced. After that, the steps required to generate the input will be shown.

6.2.1 The Simulator

The ability to perform a performance analysis of a Material Handling System very quickly, is a big pro in the design process. Having to wait a long time for such an analysis to finish, every time a small portion of the setup is changed, can become very tiresome. It also limits the number of configurations that can be tested in a certain amount of time. That is why Jacques Verriet of ESI created a Lightweight Simulator (LWS), which can very quickly simulate a given system. It enables the designers to perform dynamic performance analysis in the early stages of the design process. For this simulator, an MHS gets translated into a collection of segments, that together form the system that needs to be analyzed. Over these segments, totes (order bins) travel through the system, to be filled with goods at certain designated points.

An input file for the LWS looks like Listing 6.1. It is a header line, followed by a list of segments. Segments are uni-directional conveyers, i.e., starting at \((X_1, Y_1, Z_1)\) and ending at \((X_2, Y_2, Z_2)\). Segments \(A\) and \(B\) are connected if the head of \(A\) coincides with the tail of \(B\), or vice versa.
Chapter 6: Use Cases

The LWS is discretized, meaning that every segment consists of a fixed number of \textit{positions} (sections on which a tote can be stored). The assumption is that it takes exactly one time-unit to move a tote from one position to another (connected) position.

The segments get a specific role, based on the \textit{role} parameter. The different roles a segment can have are:

- (N)one: a normal Conveyer.
- (I)nput: a Conveyer that acts as an input to the entire system.
- (O)utput: a Conveyer that acts as an output of the entire system.
- (P)icking: a Workstation.
- (C)onsolidation: a collection point for completed totes.
- (S)equencing: an auxiliary segment, used for scheduling.

Workstations have an \textit{activityTime} and \textit{activityTimeVariability}, indicating the distribution of the time their operation takes. These operations are modeled as time-delays in the routes. The meaning of the other parameters is not relevant for this use case.

6.2.2 The Approach

The idea to generate input for the LWS from an FMFD provides a problem. Recall that an FMFD is an abstract description of the structure of an MHS, and does not contain the positional information needed in a layout. However, as this scenario is a proof of concept, reality can be bend a bit, and the sizes and positions of units in the FMFD will be interpreted as real the sizes and positions of the units.

The data needed to generate input for the LWS, is contained in the two files created by the graphical editor. On the one hand there is the FMFD, containing all the units in the system, along with the values of their parameters; on the other hand there is the diagram file. This diagram file contains all the positional information, but presents another problem: ATL cannot be used to generate a new model based on this diagram file. Since there is no metamodel for this diagram file (and it would be very hard to define one), transformation rules cannot be defined on it.

Fortunately, there is another way. Recall that EMF can generate all Java interface and implementation classes for all the classes in the metamodel, plus a factory and package (meta data) implementation class; basically everything that is necessary to create, load and edit models through Java code. So an FMFD and the corresponding diagram file could be loaded into a Java program, processed, and combined into a new Layout model. This Layout model will contain all positional data that is needed to generate input for the LWS.
6.2 Generating Simulator Input

6.2.3 The Metamodules

This use case is based on FMFDs conforming to the metamodel described in Chapter 3, i.e., FMFDs containing three different units: Miniload, Workstation, and Conveyor.

To facilitate the input generation, a metamodel for layout data needs to be developed. This is a small metamodel and looks like Figure 6.5.

![Figure 6.5: The Layout metamodel](image)

It is a parent class SystemLayout, containing a LayoutUnit for every Unit in the FMFD. This LayoutUnit contains the start- and end-coordinates of the Unit, along the number of positions; everything that is not contained in the FMFD, but is needed to draw a layout.

At this stage, it might be useful to combine the FMFD- and the Layout-metamodules, since they both contain (disjunct) data about the MHS that is modeled. However, merging the two models is not really an option, since they both describe a different domain (an abstract topology of a MHS, and concrete positioning information). For this reason, a new ‘layer’ on top of the two metamodules will be created.

The idea is that there will be one SuperUnit for every element of the modeled MHS. This SuperUnit is linked to its corresponding Unit from the FMFD metamodule, and its corresponding LayoutUnit from the Layout metamodule. This can be seen in Figure 6.6(a). Its structure is similar to the Layout metamodule. Because the links from this metamodule to the other two metamodules cannot be shown graphically, metamodule is also displayed as treeview, in Figure 6.6(b).

![Figure 6.6: The SuperModel](image)

Note that while every SuperUnit has exactly one corresponding Unit, it may have zero or more corresponding LayoutUnits. The reason for this is the way the Layout needs to be structured in order to generate input for the LWS. The next sections will go deeper into this, describing the three steps needed to produce the LWS input:

- Creating the SuperModel-model.
- Creating the Layout-model.
- Generating the output.
6.2.4 Creating Models

First the SuperModel is created. This is done based on the FMFD, and the process is simple. You iterate over the contents of the FMFD, and for every Unit (i.e., every class that has Unit as its supertype), a SuperUnit is created. This SuperUnit is linked to the Unit from the FMFD.

Now it is time to calculate the lengths en directions of the conveyers. This is done by scanning the diagram file for lines that represent the elements of the FMFD. These lines look something like:

```
<element xmi:type="FMFDConcreteMetamodel:Conveyor"
  href="ZPS.fmfdconcretemetamodel#Cv1262610585975"/>
<layoutConstraint xmi:type="notation:Bounds"
  xmi:id="_dppl4h_GEd-pSMrjqu6aQ"
  x="10" y="558" width="246" height="21"/>
```

They hold the coordinates of the top-left position of the element, as well as the width and height, and can be linked to the corresponding Unit (and also to the SuperUnit) through the identifier. Through the width/height ratio, it can be determined whether the unit is placed horizontally or vertically. Combining this with the types of the outermost Connectors of the Unit, gives the direction of the unit. Figure 6.7 shows how this works.

![Figure 6.7: Getting the direction of a Unit, based on its two outermost Connectors](image)

Then a Java class called ‘JavaUnit’ gets created, to collect all this data. This way there is a collection of JavaUnits, that corresponds to all the Units of the FMFD. Now there are two last actions to perform, to let all these JavaUnits contain the necessary information:

- The number of positions need to be added; this is done as follows: the number of positions of a Conveyor is the length of the Conveyor divided by 10. The number of positions of a workstation is 1.

- A Miniload gets split into two Conveyers, one with an Input Role, and one with an Output Role.

Now that there is a collection of JavaUnits, you can simply iterate over it, creating a LayoutUnit for every JavaUnit and link it to the associated SuperUnit, based on the identifiers. The SuperModel now contains both positioning data and properties for every unit, and the next step can begin: generating the input file for the LWS.
6.2 Generating Simulator Input

6.2.5 Generating the Input

Generating the LWS input from the Layout model is a code generation step. As shown in Chapter 5, this can be done with Xpand. However, since the transformation step is done in a Java environment, this generation step will be done programmatically as well. This generation step is nothing more than a simple nested iteration. The general structure is as follows:

Iterate over the SuperUnits from the SuperModel-model. For every SuperUnit:

- Follow the FMFDUnit reference to the associated Unit.
- Gather the data that is in the Unit into a String.
- Follow the LayoutUnit reference to the associated (collection of) LayoutUnits.
- For every LayoutUnit in the associated collection, its contents gets concatenated with that of the Unit String, and write it to the output file.
Chapter 7

Looking Ahead: Metamodel Evolution

We end this thesis by having a look at the notion of (meta)model evolution: how can the evolution of metamodels, and the co-evolution of the related models and artifacts, be dealt with? Although this topic is interesting enough to be handled in a separate chapter, implementing a complete evolution approach is beyond the scope of this thesis. This is, however, an essential part of the MDE paradigm, and certainly something that is worth looking into in a possible follow-up project.

7.1 Metamodel Evolution in General

Something that always needs to be taken into consideration when working with metamodels that describe a changing domain, is the notion of evolution; how does the metamodel, and the models and other artifacts conforming to it, deal with changes in the problem domain. This evolution of the metamodel can have different causes. The metamodel could be altered to reflect changes in the domain, but it can also be altered to correct modeling errors. Finally, changes to the metamodel could have nothing to do with the problem domain, but rather with a change in the internal representation of the domain in the metamodel.

7.1.1 Dealing with Metamodel Evolution

The metamodel evolves alongside its problem domain. The models and derived artifacts, however, need to co-evolve in order to remain compliant with the evolving metamodel [15]. So the real problem is dealing with the models and the other derived artifacts. Minor changes to the metamodel will most likely not affect the already existing models, but what to with the changes that cause models to no longer conform to their metamodel?

There are two ways to deal with this co-evolution:

- Leave the old models and artifacts as they are, keep an old version of the metamodel and treat them as legacy artifacts.
- Create a transformation to transform the old models in such a way that they conform to the new metamodel. Of course, a copy of the old model can always be kept as a legacy model, if that would be desirable.

Both approaches have their pros and cons. The first choice is the easy way out. Apart from updating the metamodel to describe the new situation, it takes no effort. This may be acceptable
if the domain almost never changes, but if the metamodel models a rapidly evolving domain, this
approach is not recommended. It would lead to an enormous amount of slightly different meta-
models, and this way it would be hard to keep track of which models belong to what metamodel.
The second approach fixes that problem, but has issues of its own. There now is a need for model
transformations, to address the changes in the metamodel. Furthermore, these transformations
must preserve all the data that is present in the models; it may add some new data, but it should
not remove and/or alter any data that is already in the model.
When, however, the level of abstraction in the metamodel changes, and the existing data is no
longer necessary or relevant, it might be a good idea to both keep the original models as legacy
models, and co-evolve the models alongside the metamodel. This way, the data that is not needed
anymore in the new situation (but may be needed again in the future) is not lost.
But if all the data needs to remain intact, it might be better to treat the old models as legacy
models. Such a big change usually indicates a drastic change in the problem domain, and then it
might not be desirable (or even possible) to transform all old models.

7.2 Evolution in the Libraries

If the question of evolution gets extended to the domain of the Concrete Metamodel, defined in
Chapter 3, things change a little bit. Since the abstract part of the metamodel will not change
(under the fair assumption that a hierarchy, as well as the connection structure will always exist),
only the Libraries need to be taken into consideration. Since a Library has a very strict structure
(as described in Section 3.4.2), only a limited amount of evolution can take place.
The fact remains that when the Library, that the Concrete Metamodel is based on, evolves, it
causes the Concrete Metamodel to change. This, possibly, breaks conformity with everything that
is related to the Concrete Metamodel. The following artifacts in particular need to be thought
about:

- The existing models
- The graphical editor based on the metamodel
- The constraints that hold on the metamodel
- The transformations regarding the metamodel

In the remainder of this chapter, each of these artifacts will be considered. A sketch of an approach
to handle each of them will be given, based on a few methods that have already been developed.
Note that no real implementation is given.

7.2.1 The Existing Models

Once the Concrete Metamodel is updated, with respect to the changes in the Library, the models
might need to be transformed to become conforming to the Concrete Metamodel again. Roughly,
the type of changes to the metamodel, with respect to the existing models, can be divided into
two categories: changes that break the conformity of models, and changes that do not.
The second category is the easy one: nothing needs to be done there. Examples of this are
new classes, that do not have obligatory relations with already existing classes, or new, optional
attributes.
The first category is where the problem lies. A transformation is needed to get the original
models conforming to the Concrete Metamodel again. Ideally, this transformation can be derived
by analyzing the changes between the old and the new metamodel. However, sometimes these
changes are too complex to be dealt with automatically. User input is then needed to ensure a proper transformation.

Cicchetti et al. [6] propose a method to automatically perform co-evolution. They base their method on the three different cases shown above:

1. Non-breaking changes; changes that do not affect the models.
2. Breaking and resolvable changes; changes that affect the models, but can be handled automatically.
3. Breaking and unresolvable changes; changes that affect the models, and need user input to be handled correctly.

These three categories get subdivided, and a proposal to perform (semi-)automated co-evolution is given. This method is based on a changes-metamodel, a metamodel that describes the changes between the two versions of the metamodel. Since there are a few constraints to the manner in which evolution can occur, as shown in the previous section, a simplified version of the method is applicable in the case of the Concrete Metamodel. So only a few of the cases described by Cicchetti et al. can occur. There is no need to take changes in hierarchy, or interaction between sub- and superclasses into account, since such changes will never occur in a Library.

Based on these observations, a trimmed down version of the ‘Changes table’ created by Cicchetti et al. can be drawn up. This is shown in Table 7.1.

<table>
<thead>
<tr>
<th>Change type</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-breaking changes</td>
<td>Generalize attribute</td>
</tr>
<tr>
<td></td>
<td>Add class</td>
</tr>
<tr>
<td></td>
<td>Add non-obligatory attribute</td>
</tr>
<tr>
<td>Breaking and resolvable changes</td>
<td>Remove class</td>
</tr>
<tr>
<td></td>
<td>Remove attribute</td>
</tr>
<tr>
<td></td>
<td>Rename class/attribute</td>
</tr>
<tr>
<td></td>
<td>Move attribute</td>
</tr>
<tr>
<td></td>
<td>Extract/inline class</td>
</tr>
<tr>
<td>Breaking and unresolvable changes</td>
<td>Add obligatory attribute</td>
</tr>
<tr>
<td></td>
<td>Restrict attribute</td>
</tr>
</tbody>
</table>

Table 7.1: The different types of changes (based on [6])

With generalizing or restricting an attribute, it is mend that the cardinality of the attribute is changed. For example, if there is an attribute \( x \), which must have a value between 1 and 10, then generalizing it would be changing these bounds to ‘between 0 and 20’. The other way around (changing the bounds to ‘between 5 and 10’) would be restricting.

Syntactically speaking, generalizing does not need user intervention. No bounds will be crossed, if the only thing that happens is that the bounds get wider. If may, however, be the case that the changing of the bounds has a deeper, semantical meaning. In that case, while the model might still be syntactically correct, it does need some user interaction to become semantically correct again.

In general, semantical changes that do not alter the syntax of the metamodel, need user interaction to be reflected correctly into the conforming models (and other artifacts, for that matter).

Restricting an attribute needs human intervention, as does adding an obligatory attribute; user intervention is needed to set the attribute to the correct value. Apart from these cases, all the changes can be handled automatically. Note that there is no mention of ‘references’ in Table 7.1. This is done because references can be seen as an attribute of a certain class, with a type (the target class) and a cardinality. This keeps the table from getting filled with items that in practice mean the same thing.
To perform this co-evolution, a stripped down version of the work of Cicchetti et al. could be used. The complete method has been tested and implemented in ATL, and can be found on the website of Davide Di Ruscio [7].

7.2.2 The Graphical Editor

The graphical editor poses another problem, when considering evolution. GMF was chosen for its compatibility with everything already created, and the fact that the editor could be customized to suit every need. However, the biggest problem with GMF was the fact that everything needs to be defined by hand, from the looks of the icons, to the mapping of the graphical elements to the metamodel. So it figures that a change in the Library requires some effort to get the editor back in sync with it.

This effort comes in a few different flavors: there are four types of object involved (classes, references, datatypes and attributes), and one of three things can happen to them. They can either be added, removed, or changed. Please note that this division does take into account the specific evolutions that can occur in a Library. When considering all the possible types of changes to a metamodel, as sketched in Table 7.1, the list of different types of solutions would become larger. Regardless of what type of object gets altered in what way, it will not always effect the graphical editor. As we can see in Table 7.2, there is another distinction to make.

<table>
<thead>
<tr>
<th>Change Type</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not effecting the editor</td>
<td>Add/remove/change non-graphical attributes</td>
</tr>
<tr>
<td></td>
<td>Add/remove/change datatype</td>
</tr>
<tr>
<td>Effecting the editor</td>
<td>Add/remove/change class</td>
</tr>
<tr>
<td></td>
<td>Add/remove/change graphical attributes</td>
</tr>
<tr>
<td></td>
<td>Add/remove/change reference</td>
</tr>
</tbody>
</table>

Table 7.2: The different types of changes, w.r.t. the graphical editor

With ‘non-graphical’ attributes, those attributes that do not have a graphical representation in the editor are mend. There is no need to worry about the first category, since those changes do not have any influence on the editor. The second category of changes is the one where human intervention would be needed. As with the evolution of models, a altered version of the work of Cicchetti et al. could be used.

Another way to deal with the co-evolution of the graphical editor, is to change the structure of such an editor. As described in Chapter 4, the ideal situation would be an editor that has different palettes, based on the metamodel (and thus on the Library) that is currently loaded. This way, the editor would not be affected by evolution of the Libraries, since it would simply load a new palette, based on the new Library. Also, legacy models could still be edited this way, provided that the old metamodels they conform to still exist.

7.2.3 The Constraints

The constraints in the Check format (as shown in Appendix B) have a very specific form. They are called on a certain class, may traverse some references, and then check an attribute for a certain property. This makes evolution of the constraints relatively easy:

- Added classes/references/attributes can be ignored. These will not break conformity of the constraints file to the metamodel. If there is need for constraints over these new elements, they need to be added by hand.

- Deletion of any of the classes/references/attributes in a constraints would render it useless. Therefore, it should be deleted from the list.
• Changes in classes/references/attributes can be resolved by using the changes-metamodel defined by Cicchetti et al., and a slightly altered version of their approach.

7.2.4 The Existing Transformations

What is left now are the model transformations and code generators that are defined on the Concrete Metamodel. Chapter 5 showed how useful these are, and why it is a good idea to use them. This is why it would make sense to look at ways to evolve them together with the metamodels, instead of creating new transformations after every change in the metamodel they operate on.

Code Generators
The evolution of code generators seems to be relatively easy. All that is done in a generator, is iterate over certain classes of the model, following references, and handling the attributes in a predefined way. This makes evolution of a code generators basically the same as evolution of the constraints file, discussed in the previous section.

Model Transformations
The evolution of model transformations is a bit more involved. A metamodel can play different roles in the model transformation: it can belong to the source model, the target model, or both. Another subdivision (like with the models) is between obligatory and non-obligatory attributes or classes. Combining these two divisions, a table just like Table 7.1 could be created. Now, a similar method like the one mentioned in Section 7.2.1 could be implemented to handle the evolution of metamodels on the level of model transformations.

7.3 Related Work on Model Evolution

When looking at the handling of evolution within Eclipse, there is the Atlas Model Weaver (AMW) [11], part of the Eclipse Generative Modeling Technologies (GMT) project. AMW is a tool for establishing relationships, i.e., links, between models. Using these relationships, transformations between metamodels can be generated automatically. This is a technique that might be used to handle metamodel evolution, with respect to the transformation of the models conforming to the old metamodel. An example of using AMW to generate model transformations is given by Jossic et al. [23].

Another related issue, is the notion of model-versioning. There are several packages available aimed at the versioning of source code, but the versioning of models is something that has been fully developed yet. However, since models are the central aspect of Model-Driven Engineering, the availability of a good versioning system for models could be invaluable. Within the SET group, Zvezdan Protic is on working metamodel-assisted model comparison [38]. Being able to accurately compare models, is a first step into the versioning of models.
Chapter 8

Conclusions

In this thesis, we tried to give an answer to the problem statement from Section 1.2.3. This problem statement was:

“Can we formalize Material Flow Diagrams, by constructing a metamodel that captures their syntax and semantics?”

It was formulated in the bigger context of introducing Model-Driven Engineering processes centered around Material Flow Diagrams into the processes at Vanderlande Industries. In this bigger context, the creation of a metamodel for MFDs was the first step to take. This is shown in Figure 8.1, which depicts the flow of introducing MDE processes around a certain artifact.

Figure 8.1: The steps taken in our Model-Driven Engineering Approach

The problem statement was answered positively by indeed creating such a metamodel. After this, both a graphical editor and model transformations were created. More precisely, the following artifacts were created during the course of the project (Appendix E gives a list of the physical locations of these artifacts):

- A metamodel for FMFDs, based on
  - An Abstract Metamodel, containing general hierarchy and connection information.
  - Libraries, containing information on a specific subdomain.
  - A Concrete Metamodel, combining the Abstract Metamodel with a Library.
- A graphical editor, based on a Concrete Metamodel, to create FMFDs with. While the editor itself is a proof of concept, and is not yet as user-friendly as possible, we have shown that it is relatively easy to create and maintain such an editor.
• Model transformations; one transformation to merge the Abstract Metamodel with a Library, in order to create a Concrete Metamodel, and on transformation to compute some basic metrics of FMFDs.

• Code generators; one that generates dot code, to display a stripped down class diagram, depicting the structure of an FMFD.

We also performed two use cases. In the first one, we derived a framework for high-level controller software from the hierarchy structure of an FMFD. Using this framework, we were able to generate an image displaying the structure of the controller software.

With the second use case, we extracted a layout model from an FMFD, and used this layout model to generate input for a Lightweight Warehouse Simulator created at ESI.

To be able to perform these use cases, multiple (meta)models where created. An overview is given in Figure 8.2.

![Figure 8.2: An overview of all the (meta)models we created](image)

8.1 Recommendations for Further Research

8.1.1 Metamodel Evolution

Chapter 7 describes the evolution of metamodels in general, and of the Concrete Metamodel in particular. Due to the split between the generic structure, in the Abstract Metamodel, and the actual Units, in the Libraries, the evolution of the metamodels described in this thesis is more limited than in general. The evolution of the Concrete Metamodel is limited to the evolution of the Library it depends on, and as such, simplified versions of existing evolution algorithms can be used on the FMFDs conforming to the evolved Concrete Metamodel.

A complete implementation of such a evolution approach was beyond the scope of this project. However, since metamodel evolution, and the co-evolution of the models, and other artifacts, that conform to the metamodel, is an essential part of the MDE paradigm, this subject is worth investigating more in-depth in a follow-up project.
8.1 Recommendations for Further Research

8.1.2 Automatic Generation of the Graphical Editor

We have shown how to create a graphical editor using GMF. This is a somewhat involved process, but most of the actions that have to be performed, are standard actions based on the metamodel element that we want to display. This gives rise to the idea that it might be possible to automate at least a part of this process. As with the Abstract Metamodel, it would be possible to isolate a portion of the editor code that is the same for every editor that is based on a Concrete Metamodel, a sort of Abstract Editor. Then the only thing that needs to be generated, is the part of the editor that depends on the elements of a Library. To accommodate this automatic generation of a graphical editor, it may be needed to add some extra information to a Unit, e.g., the color of the node that will represent it in the graphical editor.

Related to this automatic generation, is the way the graphical editor reacts to evolution of the metamodel. If it is feasible to evolve the editor alongside the metamodel (for instance in the way that models can co-evolve), this would be preferable to generating a new editor every time the metamodel changes.

8.1.3 Automatic Generation of a Test Suite

Another direction that is interesting to look at, is the automatic generation of a test suite. A test suite is a collection of models, all conforming to the same metamodel. These models are designed in such a way that they cover all aspects of the metamodel. In the case of our Concrete Metamodel, such a test suite could consist of models containing exactly one Unit, models containing tens of thousands of Units, models with hierarchy, models without hierarchy, etc.

Ideally, such a test suite could be generated automatically for an arbitrary metamodel. This means that a program to do so should first analyze the metamodel, and then generate a proper test suite.

Some research has already been done into the automatic generation of models. Mougenot et al. [32] suggested the use of the Boltzmann random sampling theory to generate random models. They did, however, ignore aspects like constraints that have to hold for the generated models.

8.1.4 Combining FMFDs with behavioral models

In the SET group, a DSL for communicating system, called SLCO, or the ‘Simple Language for Communicating Objects’ [1], has been developed. Models in this DSL can be transformed into various simulation formats (POOSL [46]), and model checking formats (SPIN [20]). It is also possible to generate NQC-code [3] and dot code [18].

As a proof of concept, the ‘LEGO case’ was created. It is a LEGO Mindstorms [29] construction consisting of two conveyers, whose contents need to be merged onto a third conveyer. The goal of this LEGO case was to define the behavior of the system in an SLCO model, and than use model transformations to eventually generate NQC code. This NQC code can then be used to control the Mindstorms controllers [37].

A direction for further research is to investigate how the topology information of an FMFD can be combined with the behavioral SLCO models. Say we have, for example, the behavior models for the different parts of the LEGO-case (in the form of an SLCO model). If we could create a Library (and thus a Concrete Metamodel) to model this LEGO case, a transformation could be made to combine the two into a big SLCO-model. This model can then be simulated.

If this works, it could be generalized, e.g., by developing a transformation that creates a coupling between an FMFD and the behavior of its components.
8.2 Acknowledgements

This work has been carried out as part of the FALCON project under the responsibility of the Embedded Systems Institute with Vanderlande Industries as the carrying industrial partner. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute (BSIK03021) program.
Bibliography


Appendix A

Instantiations of the Metamodel in Development

In this appendix the instantiations of the metamodels described in Section 3.2 are given. The IMFD that the models describe is given in Figure 3.2.
Figure A.1: The MFD in the model containing positions
Figure A.2: The MFD in the model using graph representation
Figure A.3: The MFD in the model using graph representation
Appendix B

List of Constraints on the Metamodel

Listing B.1 is a list of the constraints that are defined on the metamodel, as described in Section 3.2.4. In fact, it is a listing of the FMFDConstraints.chk file, containing the constraints. The meaning of each of the constraints is described after the keyword ERROR.

1 // Checking the MFD
2 context MFD ERROR "The MFD must have valid MId and MName":
3     this.MName != null && this.MName != '' && this.MId != null && this.MId != ''
4 ;
5 // Checking the Components
6 context Component ERROR "All Components must have valid CId and CName":
7     this.CId != null && this.CId != ''
8 ;
9 context MFD ERROR "Every component must have a unique CId (no distinction in caps)"
10     let s = getAllComponents(this):
11     s.forEach(s1 | !s.exists(s2|(s1 != s2) && (s1.CId.toLowerCase() == s2.CId.toLowerCase())));
12 // Checking containers
13 context Container ERROR "Containers may not be empty";
14     !this.components.isEmpty;
15 // Checking the Connections
16 context Connection ERROR "All Connections must have valid ConId":
17     this.ConId != null && this.ConId != ''
18 ;
19 context MFD ERROR "Every Connection must have a unique ConId"
20     this.connections.forEach(c1 | !this.connections.exists(c2|(c1 != c2) && (c1.ConId.toLowerCase() == c2.ConId.toLowerCase())));
21 // Checking the connectors
22 context Connector ERROR "All inConnectors must have valid CtId":
23     this.CntId != null && this.CntId != ''
24 ;
25 context MFD ERROR "Every inConnector must have a unique CtId"
26     let c = this.connections.inConnector:
27     c.forEach(c1 | !c.exists(c2 | (c1 != c2) && (c1.CntId.toLowerCase() == c2.CntId.toLowerCase())));
context MFD ERROR "Every outConnector must have a unique CntId":
let c = this.connections.outConnector:
c.forEach(function(c1) {
  c2 = this.connections.outConnector.find(c2 => c1 !== c2 && c1.CntId.toLowerCase() === c2.CntId.toLowerCase());
  if (c2) throw new Error("Every outConnector must have a unique CntId");
});

Listing B.1: The constrains on the metamodel for FMFDs
Appendix C

An Introduction to the Graphical Modeling Framework

C.1 The metamodel

In this tutorial, we will build a graphical editor based on a simple metamodel, describing directed graphs. The metamodel, which we store as `graph.ecore` is shown in Figure C.1.

![Figure C.1: Our example metamodel, describing graphs](image)

A Graph contains zero or more Vertices and zero or more Edges. Every Edge has exactly two Vertices, and every Vertex has one or more incoming Edges, and zero or more outgoing Edges. Furthermore, every Graph, Vertex and Edge has its own identifier. The Model element is the root of our diagram we are going to draw; it has been added solely for the creation of the editor.

As said before, this is a very simple metamodel, but it contains most of the aspect that we need to deal with when creating an editor in GMF: nodes, connections, containment and attributes. Next we’ll look at how to get started.

C.2 Let’s get started

First of all, we only use the wizards that come with GMF to create the appropriate files, not to create their contents also. We have not found a good description of how the wizards work exactly, and if we build the whole thing from scratch, we have much more control over the looks and feel of our editor.

So we do things by hand. First of all, we create a new GMF project, let’s call it `my.gmf.tutorial`. It automatically creates a project with a few subfolders. In the model folder, we create our metamodel (calling it `graph.ecore`) After this, we create the Genmodel (right-click on `graph.ecore` and do ‘New -> Others -> EMF Generator Model’) (Figure C.2).
Now we can start to create a GMF editor. To do this, we need to construct three models:

- **Tooling Definition Model**: This model contains the different items that can be drawn using the editor.

- **Graphical Definition Model**: This model contains the graphical representation of the items defined in the Tooling Definition Model.

- **Mapping Model**: This model maps the items from the Graphical Definition Model and the Tooling Definition Model onto the correct classes of the metamodel.

In the next three sections, we will look deeper into each of these three models.

### C.3 The Tooling Definition Model

The Tooling Definition Model is the easiest of the three. In it we define the available tools, i.e., the items that we can draw with our editor. We right-click on our metamodel and go ‘New -> Other -> Graphical Modeling Framework -> Simple Tooling Definition Model’. Make sure the name is correct (graph.gmftool), and click ‘Next’. In the next screen, we select our metamodel (if that is not already done), and we take Model as our diagram node. We click ‘Next’, and DESELECT everything in the final screen, before we click ‘Finish’.

We now see that GMF already created a Palette for us (which holds all the tools), and a ToolGroup called ‘graph’. A ToolGroup is a container for the Creation Tools, such that you can separate different parts of the model in different ToolGroups. In our case, this is not necessary, since we only have three Creation Tools. We now create the following structure:

- **Tool Group** graph
C.4 The Graphical Definition Model

- Creation Tool Graph
- Creation Tool Vertex
- Creation Tool Edge

All we have to do now is assign images to the tools. For now, we will go for the standard images. In Section C.7 we will discuss custom images. So we right-click one of the Creation Tools, and add two new children: Small Icon Default Image and Large Icon Default Image. Do this for all three Creation Tools. Our model will now look like Figure C.3.

Figure C.3: The gmftool file

And that is all for the Tooling Definition Model. If we extend our model, all we need to do here is add a new CreationTool for every new object we can draw. As said before, you can also create new ToolGroups is you want, to group together certain Creation Tools.

C.4 The Graphical Definition Model

In the Graphical Definition Model, we define what our items (which we defined in the Tooling Model) will look like. This is probably the most involved part of creating an editor, so we will take our time for this. We will explain the most important parts, and introduce some eye-candy. We have never found a complete description of what every option means, and a lot of the time you just have to experiment with some of the options, in other to get the editor looking the way you want to.

The creation of the Graphical Definition Model is practically the same as with the Tooling Model: We right-click on our metamodel and go 'New -> Other -> Graphical Modeling Framework -> Simple Graphical Definition Model'. Make sure the name is correct (graph.gmfgraph), and click 'Next'. In the next screen, we select our metamodel (if that is not already done), and we take Model as our diagram node. We click 'Next', and DESELECT everything in the final screen, before we click 'Finish'.
GMF creates a Canvas and a FigureGallery for us. A Canvas is the top node, containing all graphical information. A FigureGallery is the same thing as a ToolGroup in the previous section; it is just a container to group related objects. You can rename it if you like.

C.4.1 The Graph element

First we create a graphical representation of the Graph element. We want a Graph to be a square, divided into two subsquares. One of them is at the top and only contains the ID of the Graph, the other one fills the rest of the square and contains the Vertices and Edges.

The main Rectangle
We create the following structure:

\[
x \text{ Figure Gallery Graph}
\]

- Figure Descriptor \textit{GraphFigure}
  - Rectangle \textit{GraphShape} ("Line Width" = ‘2’)
    * Border Layout
    * Foreground \textit{blue}
    * Insets 5 (set all values to ‘5’)
  - Rectangle \textit{GraphIDShape}
    * Border Layout Data \textit{BEGINNING} ("Alignment" = ‘BEGINNING’, “Vertical” = ‘true’)
    * Flow Layout \textit{true} ("Vertical" = ‘true’)
    * Foreground \textit{white}
    * Label \textit{GraphIDLabel} (“Default” = ‘<.. ID ..>’)
  - Rectangle \textit{GraphBodyShape}
    * Border Layout Data \textit{CENTER} ("Alignment" = ‘CENTER’ and “Vertical” = ‘true’)
    * Foreground \textit{grey}
  - Child Access \textit{getFigureGraphIDLabel}
  - Child Access \textit{getFigureGraphIDShape}
  - Child Access \textit{getFigureGraphBodyShape}

Some remarks about the things we did here:

- We can change the color of the border and the background of a Figure by adding a “Foreground Color” and a “Background Color”, resp. You can choose to either give a constant color, or to give it an RGB color.
- “BorderLayout” enables us to align the contents of a Figure properly.
- “Insets” sets some margins to everything that is displayed inside the Figure it belongs to.
- “Border Layout Data” sets the way that children of a “BorderLayout” parent should be aligned.
- “Flow Layout” makes sure the ID gets displayed correctly.
- “Child Access” allows us to reference to certain Figures from other points in this model.
Finishing

We are now almost done with defining the Graph element. We now add a Nodes node to the Canvas, call it ‘GraphNode’ and link it to the Graph figure, by selecting the ‘Figure Descriptor GraphFigure’ in the “Figure” attribute.

We also need to add a Labels Diagram Label to the Canvas, and link it to ‘GraphIDLabel’ by setting the “Accessor” attribute to ‘getFigureGraphIDLabel’ and Figure to ‘GraphFigure’. We call this Diagram Label ‘GraphID’.

Finally, we add a compartment to the Canvas. Compartments are part of Figures that can contain other Figures. In our case, ‘GraphBodyShape’ will contain Vertices and Edges. So we add a Compartment, call it ‘GraphBody’, and link it to the Figure Descriptor by selecting the correct one for the Figure property. Also, we assign the correct “Accessor” (‘getFigureGraphBodyShape’).

And that’s it for the Graph element. We will go a bit faster through the other two elements. By now, your gmfgraph file will look like Figure C.4.

C.4.2 The Edge element

Compared to the Graph element, the Edge is remarkably easy. The following structure suffices:

```xml
<Figure Gallery Graph
  <Figure Descriptor GraphFigure
  <Figure Descriptor EdgeFigure
    <Polyline Connection EdgeShape
      * Foreground black
  </Figure Descriptor EdgeFigure
  </Figure Descriptor GraphFigure
```
• Polyline Decoration *Arrowhead*
  
  - (-1,1)
  - (0,0)
  - (-1,-1)

Since we model a directed graph, we want an arrowhead at the target end of an edge. And, of course, we have to create that arrowhead by ourselves. We do this by adding a Polyline Decoration to our Figure Gallery. A Polyline Decoration consists of a sequence of coordinates through which a line is drawn. This Decoration will always be attached to the end of a Polyline, which is (0,0). To add this Decoration to our ‘EdgeShape’, we click on it, and set “TargetDecoration” to ‘Polyline Decoration Arrowhead’.

After this, we only have to add a Connection to the Canvas. We link this Connection to the correct Figure, call it ‘Edge’ and that’s it for the Edge element. Our gmfgraph file now looks like Figure C.5.

![Figure C.5: The second part of the gmfgraph file](image)

### C.4.3 The **Vertex** element

We define a Vertex as a circle, with its ID in it:

```
  x Figure Gallery Graph
    • Polyline Decoration *Arrowhead*
    • Figure Descriptor *GraphFigure*
    • Figure Descriptor *EdgeFigure*
    • Figure Descriptor *VertexFigure*
```
C.5 The Mapping Model

Now that we have our Tooling and Graphical Definition, we need to tie them together, and make the link to the metamodel. For this, we need the final of our three models: the Mapping Model. So again, we go ‘New -> Other -> Graphical Modeling Framework -> Guide Mapping Model Creation’. Make sure the name is correct (graph.gmfgraph), and click ‘Next’. We choose our metamodel, and ‘Model’ as class, and in the next two screens, we select our Tooling and Graphical Model. In the last screen, we remove everything in the “Nodes” and “Links” lists, and we click Finish.

What we see now is the start of our Mapping Model. GMF already created a Canvas Mapping, pointing to the top nodes of the metamodel, the Tooling Model and the Graphical model. And now our work starts.
The Graph
The first thing we do is add a Top Node Reference to the Mapping. This is a reference to the first node in our metamodel that actually means something. Recall that we only added the ‘Model’ class to act as the canvas on which we draw our graphs. So we add a Top Reference Node and set its ‘Containment Feature’ to ‘Model.contains:Graph’. In this Top Reference Node, we add a Node Mapping, which will be the actual node for the Graph element. In this Node Mapping, we set “Element” to ‘Graph’, “Related Diagrams” (hidden under “Misc”) to ‘Canvas Mapping’, “Diagram Node” to ‘Node GraphNode’ and “Tool” to ‘Creation Tool Graph’. Finally, we add a Compartment Mapping to the NodeMapping, pointing to ‘Compartment GraphBody’ (which will contain the Vertices and Edges), and a Feature Label Mapping, for the ID. In this LabelMapping, we set “Features to display” to ‘Graph.GId:EString’ and “Diagram Label” (again hidden in behind “Misc”) to ‘Diagram Label GraphID’.

The Vertex
To add children to a node, you add a Child Reference to them. The idea is exactly the same as for a TopNodeReference, so you set the correct “Containment Feature” (‘Graph.vertices:Vertex’ is this case). The thing you have to do extra, is point to the Compartment we want this node to be placed in. Since we only have defined one compartment, that should not be a hard choice. Now we create a new Node Mapping, and fill it in exactly the same way we did before, setting “Element”, “Related Diagrams”, “Diagram Node” and “Tool” to their correct values. Finally, we add a Feature Label Mapping pointing to ‘Diagram Label VertexID’.

The Edge
The Edge element is a bit different from the rest, since it is not a node, but a connection. To add it, add a Link Mapping child to Mapping. The first thing we now do is set “Element” to ‘Edge’. Then we can set the “Containment Feature” (only one option), and the “Source Feature” (‘Edge.outVertex:Vertex’) and “Target Feature” (‘Edge.inVertex:Vertex’). The “Related Diagrams”, “Diagram Node” and “Tool” properties are set as before. And that is the end of the Mapping Model. It should look like Figure C.7.

C.6 Generating the Diagram Code
Now we are just two generation steps away from our editor. First, we open the Genmodel, right-click on the top node and choose ‘Generate all’. Next, we right-click on graph.gmfmap and select ‘Create generator model...’. In the screen that pops up, just click Next and Finish. All relevant models are already added (and if not, you can’t click Next anyway, and you have to select the proper model). Once you click Finish, a graph.gmfgen file will be created. There is one thing we need to manually edit in the gmfgen file. The “ListLayout” properties of Compartments is set to ‘true’ by default. This means that all items will be stuck as much to the top as possible, and you are not able to move them. This is undesirable in most cases, so we need to fix this:

- Open the gmfgen file, and expand the top node.
- Expand “Gen Editor Generator graph.diagram” and finally expand “Gen Diagram ModelEditPart”.
- Set the property “List Layout” of “Gen Compartment GraphGraphBodyEditPart” to ‘true’ (also see Figure C.8).

Now all we have to do is right-click on graph.gmfgen and choose ‘Generate Diagram Code’. We’ll see a folder, my.gmf.tutorial.diagram, getting created. If we right-click this folder,
C.6 Generating the Diagram Code

Figure C.7: The gmfmap file

and go: ‘Run as... -> Eclipse Application’, we can start using our graphical editor. Figure C.9 shows what our editor now looks like.

We have the elements from our Tooling Model on the right-hand side, and using those we can draw the objects defined in our Graphical Model. To create a new diagram, we first have to create a new Folder (we called it ‘Graphs’). In this folder we create a new Graph Model (‘New -> Other... -> Example EMF Model Creation Wizards -> Graph Model’). We now get a treeview-editor, which is of course not what we worked so hard for. But if we right-click the .graph file we just created, we can select ‘Initialize graph_diagram diagram file’. Now we have a file that works with our new editor. Any changes made to this file (except the diagram-specific changes such as location) will be updated in the original file as well.

C.6.1 Changing and regenerating

If we change anything in one of our models, and we need to update our editor, we go through the following procedure:

- If we have changed anything in the metamodel, we first have to do the following:
  - Reload the graph.genmodel
  - Delete everything in the src folder (including subpackages)
  - Delete my.gmf.tutorial.edit, my.gmf.tutorial.editor and my.gmf.tutorial.test (select ‘Delete project contents from disk’!)
- Delete my.gmf.tutorial.diagram (select ‘Delete project contents from disk’!)
- Right-click on graph.gmfmap and select ‘Create generator model...’ (we don’t have to remove the graph.gmfgen file first, which is good, since it will remind the changes we made to it)
• Right-click `graph.gmfgen` and choose ‘Generate Diagram Code’

• Right-click on the new `my.gmf.tutorial.diagram`, and go: ‘Run as... -> Eclipse Application’

C.7 Extensions

C.7.1 More hierarchy

Let’s say we want to extend the metamodel to introduce a bit more hierarchy. We do this by letting a Graph be able to contain other Graphs. Put aside whether or not this is desirable or not, we do this to show a modeling technique. The metamodel will now look like Figure C.10. A Graph now becomes a recursive definition, since each Graph can contain multiple other Graphs.

We do not have to edit our Tooling Model or Graphical Model (since we don’t introduce new entities), but we do need to edit our Mapping Model. We add a new Child Reference to the ‘Graph’ Node Mapping. We now do the same as before, set “Compartment” to ‘Compartment Mapping GraphBody’, and “Containment Feature” to ‘Graph.graphs:Graph’. However, we cannot introduce a new Node Mapping to represent the Graph, since we can now have infinite depth. So what we do is enter a reference to the ‘Graph’ Node Mapping we already have. We give the attribute “Referenced Child” the value ‘Node Mapping Graph/GraphNode’ and voila, we have our recursive definition of the graph model (Figure C.11). We can now regenerate all the code and then test our new editor.
C.7.2 Generation of Identifiers

Now imagine that we don’t really care about the name of our Edges, but we do want all of them to have a unique identifier. We could of course enter an ID manually every time we create an Edge, but this can also be automated. If we want all identifiers to be the string “Edge” concatenated with the Java `currentTimeMillis()` function [22], we do the following (also see Figure C.12):

- Add a Feature Seq Initializer to the Edge Link Mapping
- In here, create a new Feature Value Spec, and set the “Feature” property to ‘Edge.EId:EString’
- Finally, create a new Value Expression, set “Language” to ‘Literal’, and “Body” to ‘“Edge” + java.lang.System.currentTimeMillis()’

C.7.3 Custom Images

One last option to customize the look and feel of your editor is the use of custom images in the Tooling Pane of the editor. This is actually a very easy task, and it makes your editor feel a lot more personal.

To do this, all have to do is go to `my.gmf.tutorial.edit/icons/full/obj16`. In this folder there are four images, corresponding to the four elements in our Tooling Model. We can replace this images with custom images, and we’re done.

One last thing we can do to make our editor look more slick, is removing the icons inside the Vertices. This is simple enough, if you know where to look for it. We have to open our
Now all we need to do is regenerate the code, as shown in Section C.6.1. However, one important thing to notice is that we can no longer delete the entire my.gmf.tutorial.edit folder, since it now contains our custom images. Instead, we only delete the my.gmf.tutorial.edit/src folder.

Our editor now looks like Figure C.13.

C.8 Common pitfalls

• Note that names in the main window will not always update automatically. This is fixed by just dragging the object in question up or down one position, or closing and opening the file. This is a common bug in GMF, and if you are not aware of it, you might experience a lot of frustration.

• Don't try to rearrange you Creation Tools too often. If you rearrange the Creation Tool, the Mapping Model will get confused and you have to set all Creation Tool parameters again.

C.9 Conclusion

In this tutorial we have shown how to create a graphical editor through GMF, based on a simple metamodel. The metamodels was chosen in such a way that it had the most important characteristics a bigger metamodel would also have. Furthermore, we showed some extra concepts, to make the editor more useful, and prettier. We also gave some attention to a few of the most common pitfalls in GMF, based on our own experience.
C.9 Conclusion

Figure C.11: The new Mapping Model

Figure C.12: Initializing Identifiers
Figure C.13: Our editor with custom icons, and no icons inside the vertices
Appendix D

ATL Code

In this appendix we give an overview of the problems we encountered during the creation of the M2M-transformations described in Chapter 5.

D.1 The Merge Transformation

The problem with this transformation, is that it operates on metamodels, rather than on models. And while, in theory, this should not really matter (since the model/metamodel relationship is similar to the metamodel/meta-metamodel relationship), it does; especially to ATL.

First of all, ATL needs to be told that is dealing with meta-metamodels (Figure D.1(a)). Furthermore, to be able to copy Ecore datatypes from one metamodel to the other, inter-model references need to be allowed (Figure D.1(b)).

While this last setting fixes the copying of Ecore datatypes, it apparently breaks other features. It was not possible to perform the transformation in Listing D.1, since, for some reason, the a.eType and a.eGenericType seemed to be empty.

rule copyEReferenceLib{
   from a : MM1!EReference (a.isMergableRef())
   to p : MM2!EReference{
      eGenericType <- thisModule.resolveTemp(a.eType,'p'),
      eKeys <- a.eKeys,
      eOpposite <- a.eOpposite,
      eType <- a.thisModule.resolveTemp(a.eGenericType,'p'),
      ...
   }
}

Listing D.1: The copy rule for Library References

After consulting the Eclipse forums, the following solution appeared:

The problem you have is due to the "allowInterModelReferences" parameter you needed for the previous EString issue... As you want to keep the parameter to "true", there is a trick you can use. Instead of eGenericType and eType bindings, set those attributes into an imperative section, like here:
rule copyEReferenceB{
  from
    a : Ecore!EReference
  to
    p : Ecore!EReference(
      changeable <- a.changeable,
      containment <- a.containment,
      defaultValueLiteral <- a.defaultValueLiteral,
      derived <- a.derived,
      eAnnotations <- a.eAnnotations,
      eGenericType <- a.eGenericType,
      eKeys <- a.eKeys,
      eOpposite <- a.eOpposite,
      eType <- a.eType,
      lowerBound <- a.lowerBound,
      name <- a.name,
      ordered <- a.ordered,
      resolveProxies <- a.resolveProxies,
      transient <- a.transient,
      unique <- a.unique,
      unsettable <- a.unsettable,
      upperBound <- a.upperBound,
      volatile <- a.volatile
  )
  do {
    p.refSetValue('eType', thisModule.resolveTemp(a.eType,'p'));
    p.refSetValue('eGenericType',
                 thisModule.resolveTemp(a.eGenericType,'p'));
  }
}

This did the trick, although it seems odd that it only works if it is done like this.
D.1 The Merge Transformation

(a) Telling ATL that it’s dealing with meta-metamodels

(b) Allowing inter-model references

Figure D.1: The ATL settings needed to run the merge transformation
Appendix E

Deliverables

All the files created during the course of the graduation project, can be found in the FALCON repository of the SET group, in a folder called Formalizing MFDs. In this folder, I have put an export of the Eclipse Workspace, that contained all the files. This Appendix lists which files can be found at what precise location.

All files have been created using Eclipse Modeling Tools 1.2.1.2009091-0703. This version of Eclipse contains, among others:

- EMF 2.5.0.v200906151043
- GMF 1.2.0.v20090615-0700
- ATL 3.0.1.v200909150941
- Xpand 0.7.2.v200908120436

E.1 The Metamodel

As stated before, the metamodel was implemented using EMF:

- my.mfd.abstractmetamodel/metamodel/FMFDAbstractMetamodel.ecore: The Abstract Metamodel for MFDs
- my.mfd.abstractmetamodel/metamodel/LibraryX.ecore: The Library discussed in Section 3.4.2
- my.mfd.concretemetamodel/metamodel/FMFDConcreteMetamodel.ecore: The Concrete metamodel, based on the Abstract Metamodel and Library X
- my.mfd.concretemetamodel/metamodel/FMFDConstraints.chk: The constraints implemented in OCL

E.2 The Graphical Editor

The files for the GMF implementation of the editor:

- my.mfd.concretemetamodel/metamodel/FMFDConcreteMetamodel.gmftool: The Tooling Definition Model
• my.mfd.concretemetamodel/metamodel/FMFDConcreteMetamodel.gmfgraph: The Graphical Definition Model
• my.mfd.concretemetamodel/metamodel/FMFDConcreteMetamodel.gmfmap: The Mapping Definition
• my.mfd.concretemetamodel/metamodel/FMFDConcreteMetamodel.gmfgen: The Generator Model

E.3 The Merge Transformation
The my.mfd.mergeLibrary folder contains the files need to perform the merge of the Abstract Metamodel and a Library, as described in Chapter 3.

• Ecore.ecore: The Ecore meta-metamodel
• merge.atl: The merge transformation

E.4 The Other Model Transformations
The my.mfd.transformations/ folder contains the ATL transformations presented in Chapters 5 and 6. In input, output and transformations folders contain the input, output en transformation files. The runTransformations.xml file is created to perform all transformations in the transformations folder on all models in the input folder.

• my.mfd.transformations/transformations/FMFDtoHLC.atl: The transformation to transform an FMFD into a HLC model
• my.mfd.transformations/transformations/FMFDtoMetrics.atl: The transformation to derive some simple metrics from an FMFD

E.5 The Generators

• my.hlc.diagramgenerator: The files needed to generate dot-code for a stripped down class diagram of a HLC model
• my.hlc.dotgenerator: The files needed to generate dot-code for an overview image of a HLC model
• my.mfd.diagramgenerator: The files needed to generate dot-code for a stripped down class diagram of a FMFD model

E.6 Use case: Generating Simulator Input
The files to generate LWS input from an FMFD can be found in the subfolder my.mfd.concretemetamodel/src/generateInput.