Domain-Specific Modeling of Coordinate Transformations in the ASML Metrology Software

Master thesis

Sebastián Múnera-Álvarez

Supervisors:
Dr. ir. Tom Verhoeff (TU/e)
Dr. ir. Martijn van der Horst (ASML)

Eindhoven
August 31, 2015
Abstract

General-purpose programming languages are designed to be used for writing solutions in different application domains. However, software solutions for a restricted domain have to be built using generic constructs, making the code complex and often unmaintainable.

A better solution is to use domain-specific languages (DSL), which are designed to be restricted to an application domain, therefore, they are less expressive. But they improve productivity of software development and communication with domain experts.

In this thesis, it is shown how to build domain-specific languages for a particular domain in the metrology area of ASML. The domain concerns transformations between coordinate systems in the ASML machine and setpoint calculation. The focus is on the alternatives in each step of the modeling process and, in particular, which alternative gives more benefits to ASML.

It is found that internal DSLs are a suitable solution as they balance the cost-benefit relationship of maintenance and enhanced productivity. Furthermore, model-ignorant code generation can provide more efficiency and type safety. The costs involved in maintaining a parser or a code generator can be decreased by using technologies such as Eclipse Modeling Framework.
Acknowledgments

I would like to thank all those people who have supported me during the time of this project:

- Dr. ir. Tom Verhoeff for his guidance and critical comments on my ideas, which made me a better engineer and problem solver.

- Dr. ir. Martijn van der Horst for his continuous support and help throughout the whole project at ASML. He made me feel welcome in the metrology department, and guided my work towards a tool that could be aligned with ASML’s goals.

- Ed de Gast, Joost Vromen and the people from the metrology department for the valuable discussions and critical questions which in the end, made the quality of my work much higher.

- The international community in Eindhoven and my friends: Pedro, Rubén, Édgar, Rodrigo, Anuar, Rangga, Chitra, Nacho, Sago, Beto, Pawel (and the ones that I don’t mention here but still have a special place), for being an emotional support and for showing me one more time that small things are the ones that make life worthy.

- Omna for her love, company, support, understanding and for giving me the motivation to keep my feet on the ground when I needed it the most.

- My family and friends from Colombia who have always been (and surely will be) there to support me and give a sense to whatever I do.

Last but not least, I want to thank ASML for financially supporting my master with the Henk Bodt scholarship and Colfuturo for helping me with the expenses of my first year in the Netherlands.
## Contents

| List of Figures | viii |
| List of Tables | ix |

### 1 Introduction
- 1.1 Background .................................................. 1
- 1.2 ASML context ................................................. 1
  - 1.2.1 ASML definitions ..................................... 1
  - 1.2.2 Overview of ASML metrology ....................... 2
- 1.3 Proposed approach ........................................ 4
  - 1.3.1 Domain-specific languages .......................... 4
- 1.4 Objectives .................................................. 6

### 2 Problem Definition
- 2.1 Problem statement .......................................... 9
- 2.2 Considerations for a solution ............................ 13
  - 2.2.1 Frequency of variability ............................ 13
  - 2.2.2 Physical limitations of the machine ............... 13
  - 2.2.3 Mathematical limitations ........................... 14
  - 2.2.4 Types of users ....................................... 14

### 3 Case study
- 3.1 Introduction ................................................. 17
- 3.2 Model description ........................................... 17
- 3.3 Use case scenarios .......................................... 19
- 3.4 Unit testing .................................................. 20

### 4 Domain concepts and interrelations
- 4.1 Introduction .................................................. 21
- 4.2 Functional Requirements ................................... 22
- 4.3 High-level specification ................................... 22
  - 4.3.1 Population interface ................................ 23
  - 4.3.2 Operational interface ............................... 24
- 4.4 Metamodel ..................................................... 25
- 4.5 Dynamic and static components ......................... 28
- 4.6 Conclusion ................................................... 28
# Making transformation models machine processable

## 5.1 Introduction

## 5.2 Direct use of API constructors

### 5.2.1 Transformation model

### 5.2.2 Coordinate systems

### 5.2.3 Transformation functions

## 5.3 Internal DSL

## 5.4 Ad-hoc external DSL

## 5.5 Conclusion

# Code generation

## 6.1 Introduction

## 6.2 Interpretation

## 6.3 Code generation

## 6.4 Conclusion

# Conclusions and future work

## 7.1 Critical evaluation

## 7.2 Objectives revisited

## 7.3 Recommendations for ASML

### 7.3.1 Metamodel

### 7.3.2 DSLs to make models machine processable

### 7.3.3 Interpretation vs. Code generation

### 7.3.4 Staff and resources needed

## 7.4 Future work

### 7.4.1 Related to DSL

### 7.4.2 Related to metrology functions

### 7.4.3 Related to numerical computations
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Exposure process in the ASML machine</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Alignment of reticle and wafer: (a) initial situation where reticle and wafer are not aligned, (b) a possible solution.</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>MOF layered architecture.</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Architecture of a DSL</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Three-dimensional Cartesian coordinate system</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Simple example of setpoint: (a) position of the stage when the origins coincide; (b) movement of the stage by t units to align p and q.</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Schematic view of the ASML machine with an example of setpoint calculation and alignment of points on a reticle and a wafer</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Infinite setpoint solutions as the wafer stage can rotate freely along the z axis</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Metamodel of the transformation models</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Metamodel of the transformation trees</td>
<td>27</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Ranges of parameter values for the initial model ........................................... 18
3.2 Possible coordinates for TIS marks ................................................................. 19
5.1 Summary of pros and cons of model instantiation methods ............................. 33
6.1 Summary of pros and cons of interpretation vs. code generation ...................... 39
7.1 Trade-offs of DSL approaches ........................................................................... 43
7.2 Trade-offs of Interpretation vs. Code generation .............................................. 43
Chapter 1

Introduction

1.1 Background

General-purpose programming languages (GPPL) like Java or C/C++ are designed to be used for writing software solutions in different application domains. This kind of language is usually Turing complete.

The main disadvantage of GPPLs is that software solutions for a restricted domain have to be built using generic constructs, which often makes the resulting code complex, and difficult to understand and maintain.

There is another kind of language that is gaining relevance in recent years: the domain-specific languages (DSL). DSLs are restricted to one or a few application domains, and they are less expressive than GPPLs. DSLs are also more suitable for writing readable programs that solve problems in the domain that they were designed for. Known examples of DSLs include regular expressions, HTML, CSS, GraphViz, among others.

The main two reasons why DSLs are so common nowadays are [1]: (a) They improve the productivity of the programmers because they provide a high level of abstraction which allows domain concepts to be used as if they were primitive types, and (b) they improve communication between the programmers and the domain experts as the domain concepts are common.

1.2 ASML context

The situation of the ASML software is described in this section. First, the relevant ASML terms are defined in Section 1.2.1. Then, an overview of the ASML processes and the role of the metrology department is shown in Section 1.2.2.

1.2.1 ASML definitions

Reticle A representation of a chip design or pattern to be projected on a wafer. The actual size of the pattern on the reticle is usually four times bigger than the final size of the pattern on the wafer.

Wafer A thin, usually round slice of a semiconductor material from which chips are made.
Exposure  A method for defining patterns by the interaction of light or other energy source with a photoresist that is sensitive to such energy source.

Reticle stage  A supporting structure that contains the reticle and can move the reticle to a specific position.

Wafer stage  A supporting structure that contains the wafer and can move the wafer to a specific position.

Metroframe  The static frame of the ASML machine that serves as frame of reference for the other components of the machine.

Setpoint  Desired position and/or orientation of a stage that can be controlled by the software.

Exposure scan  The operation in which a pattern on a reticle is projected onto a wafer. Both reticle and wafer stages move synchronously.

TIS scan  Measurement of the position of a TIS (Transmission Image Sensor) mark on the reticle stage, which is projected below the lens with respect to a TIS sensor on the wafer stage. This involves movements of the reticle and the wafer.

Alignment  The positioning of a set of points on the reticle relative to a set of points on the wafer. This involves movements of the reticle and the wafer.

1.2.2 Overview of ASML metrology

The lithography machines built by ASML allow to project a chip design or pattern onto a wafer using a light ray. The pattern is contained in a reticle and several copies are projected in different regions of the wafer. This process is called exposure. An example of an exposure can be seen in Figure 1.1.

On a wafer, several exposures can be performed, each one adding a new layer on top of the wafer surface. Each layer requires a different reticle with a new pattern which must be positioned properly with respect to layers produced earlier. In addition, a new photoresist has to be added to the wafer for each layer, which involves taking out the wafer from the wafer stage and then putting it back in. This process changes the parameters of the exposure, and new measurements need to be performed to determine them.

The performance of the ASML machine is measured by how accurate the layers are aligned (overlay), how good the resolution of an exposed pattern on the wafer is (imaging quality) and how many wafers are produced per hour (throughput).

The metrology department’s task is to do measurements and adjustments on the components of the machine in order to ensure a good performance. An illustration of this can be seen in Figure 1.2. The goal is to align a point $A$ in the reticle with a point $B$ in the wafer. To achieve this, a solution is to move the reticle along the $y$ axis until $A$ has the same $y$ position as $B$. There can be multiple solutions since the wafer can also be moved. The job of the metrology department is to find, for instance, $d$, which indicates how the reticle has to move. Note: This simplified example illustrates a setpoint calculation which will be the focus of this work, but it will be explained in detail in subsequent chapters.

There are many different sources of error in the metrology processes: deformations in the reticle and wafer, alignment errors, among others. These errors need to be corrected and the
Figure 1.1: Exposure process in the ASML machine

Figure 1.2: Alignment of reticle and wafer: (a) initial situation where reticle and wafer are not aligned, (b) a possible solution.
mechanics of the machine are usually not enough to prevent them, therefore it is necessary to find the corrections by software. The metrology department is in charge of designing mathematical models to calculate the setpoints and to compensate for the small errors during alignments and exposures, as well as the translation of these models to software in C/C++.

Because of small incremental changes over many years, the current implementation makes the scalability, extensibility, maintenance and debugging of the software difficult processes. Although the base mathematical models are well defined, often new challenges come up and new parameters, coordinate systems or transformations need to be included. This has proven to be a difficult task as the codebase grows over time in size and complexity. This impacts the productivity of programmers and hinders the understanding of both programmers and other stakeholders. Therefore, the advantages of DSLs, as mentioned above in section 1.1, can be valuable for the metrology software development.

1.3 Proposed approach

As mentioned above in Section 1.2.2, the metrology software can be greatly improved by a DSL. It is proposed in this document to use technologies from the field of domain-specific languages to model the situation presented in the problem statement, which will be defined precisely in Chapter 2 and evaluate different alternatives and their trade-offs. We are only concerned with setpoint calculation for reticle and wafer stages, but the approach should be extensible to cover other cases.

Using the domain knowledge that is a common language between software developers and domain experts (physicists and mathematicians), we will develop a family of DSLs that capture all the high-level concepts of the domain, and at the same time are sufficiently detailed that they contain all the ingredients needed to do setpoint calculations. The complexity inherent to the implementation details should be hidden. The trade-offs of the alternatives and design choices in every step of the DSL construction are documented, and the impact that each choice has on the development process at ASML is analyzed.

1.3.1 Domain-specific languages

In recent years, the Object Management Group (OMG) has created the Meta Object Facility (MOF) [4], which is a standard for model-driven engineering. The architecture of MOF has four layers, as can be seen in Figure 1.3.

- The M0 layer represents the physical world. In our context this represents the physical laws and behavior that drives the operation of the ASML machine.
- The M1 layer is the model. It represents concrete instances where the variables have specific values. There can be different instances through time as the variables get different values.
- The M2 layer is the metamodel. A metamodel defines structural constraints on models, and describes a set of models that conform to it. See chapter 4 for a formal definition of the metamodel that describes a transformation model as defined in Section 2.1.
- The M3 layer is the meta-metamodel. This is the language used to described metamodels. A meta-metamodel itself is a metamodel, thus it can also be used to describe the
Figure 1.3: MOF layered architecture.
common structural elements of models. A characteristic of a meta-metamodel is that it uses its own language to describe itself. This layer is mentioned here for completeness, but it will not be studied in this project.

There are several techniques to design a DSL, but all of them rely on a generic representation of the domain knowledge. This representation is the metamodel and is usually linked to a command-query interface used to manipulate and use models in memory. In Chapter 4 we describe the metamodel in detail. The DSL itself is just an interface between the user and the metamodel that allows him to create models. However, for simplicity in this document we will use the term DSL to refer to the combination of a DSL, metamodel and operational software. In Figure 1.4 the architecture of a DSL is shown. In Chapters 5 the different ways to instantiate a metamodel and their trade-offs are presented. The way to show the alternatives will be by applying them to a case study which is described in Chapter 3.

![Architecture of a DSL](image)

The operational software is a representation of a model that can be executed in a computer. Often, model-independent code reads model data which drives its behavior. Alternatively, model-specific code can be generated from a specification written in a DSL. This process is called code generation. This option is also explored in this project by providing a way to translate from a high-level model using only domain concepts to a low-level implementation of the model that could couple with other components of the ASML software. This is discussed in Chapter 6.

1.4 Objectives

The goal of this project is to give a proof-of-concept to show the feasibility of using technologies from the field of domain-specific languages, in order to improve the productivity of the metrology software development and enhance the communication with domain experts.

The proof-of-concept will be restricted to the setpoint calculation, due to time constraints (six months). Along the way, it is intended to investigate the following questions.

1. How can the metrology knowledge be structured using a domain-specific model?
2. How usable, maintainable and scalable are the different alternative domain-specific models?

3. How does the abstraction of a domain-specific model impact the efficiency of the metrology processes in terms of development time?
Chapter 2

Problem Definition

The metrology software at ASML has grown to a large and complex codebase, and it has become increasingly difficult to maintain and extend. Therefore, it is of importance to think of using domain-specific ideas that help to improve the productivity of the programmers and the communication with domain experts.

In this chapter, the specific problem to be solved is stated in Section 2.1. The limitations of the problem that need to be considered in order to give a solution are described in Section 2.2. In Section 2.2.4 a user classification is given.

2.1 Problem statement

The ASML machine has a static frame of reference, called the metroframe. The metroframe is a coordinate system that specifies any point in the machine uniquely by its coordinates. In general, a coordinate system is defined by a point called the origin and two (2D) or three (3D) independent unit vectors which can be mutually orthogonal and represent the axes. In Figure 2.1 a 3D Cartesian coordinate system is depicted. The origin is the point O and the unit vectors are i, j and k that represent the axes x, y and z respectively.

A point is uniquely identified in a coordinate system as a linear combination of the unit vectors. For instance, the point p in Figure 2.1 is represented as \( x_p \cdot i + y_p \cdot j + z_p \cdot k \); the coordinates of p are commonly written as \( (x_p, y_p, z_p) \). We will represent the coordinates of a point \( x \) with respect to a coordinate system \( c \) as \( x_c \).

In the ASML machine, in addition to the metroframe, there are other coordinate systems that use the metroframe as the frame of reference, either directly or indirectly. One coordinate system differs from another by the location of the origin in space or the relative length or direction of the unit vectors. As a consequence of this, one point in space can be identified by multiple sets of coordinates depending on which coordinate system serves as a frame of reference.

Given the coordinates of a point in a coordinate system, we can calculate the coordinates in another coordinate system of the same point, or a geometric projection of such point in another location in space. This can be achieved by means of a transformation function. When a transformation function is applied to a point, we say that the point is transformed. We will denote a transformation function that transforms from coordinate system \( a \) to coordinate system \( b \) \((a \neq b)\) as \( f_{a \rightarrow b} \). If there are three coordinate systems \( a, b, c \), such that there exists a transformation function \( f_{a \rightarrow b} \) and another \( f_{b \rightarrow c} \), it is possible to transform from \( a \) to
A coordinate system depends on parameters such as position of the origin, scaling or orientation with respect to another coordinate system. A parameter may have an associated value, but sometimes its value may be unknown. The set of parameters of all the transformation functions and their values is called a transformation state, which represents the physical state of the machine. If a transformation state has a parameter with unknown value, it means that its value still has to be determined.

In the machine, some coordinate systems are not stationary with respect to the metroframe. Such coordinate systems are associated with moving components. For example, the reticle stage or the wafer stage can move in order to align points in a reticle with points in a wafer. The description of a moving coordinate system in terms of the metroframe changes as the state of the associated component also changes. Therefore, we will parameterize the transformation function of a moving coordinate system to the metroframe by the state (location and orientation) of the component relative to the metroframe. The parameters that represent such a state are called a setpoint and they are included in the transformation state after they are determined.

We can characterize three kinds of transformations: (i) the ones that calculate the coordinates of the same point in another coordinate system; (ii) the ones that calculate the coordinates of a geometric projection of the point in another position in space, with respect to another coordinate system; and (iii) the ones that calculate the coordinates of a point in a moving component in another location after the component has moved, with respect to another coordinate system. The latter kind of transformation refers to the previously mentioned non-stationary coordinate systems.

A transformation model is a collection of coordinate systems and transformation functions between its coordinate systems.

As an example, consider Figure 2.2. A moving component called stage is associated with...
a unidimensional coordinate system $c_s$. It can be moved by an actuator on the horizontal axis with respect to the metroframe coordinate system $c_m$. The origins of $c_s$ and $c_m$ are the points $O_s$ and $O_m$, respectively. For any given state of the stage, the transformation from $c_s$ to $c_m$ involves a scaling with factor $M_x$, which represents a phenomenon like expansion by heating. The value of this factor is determined by a transformation state and is unrelated to the stage motion. In order to characterize any state of the stage in terms of the metroframe, the transformation function has to be parameterized by adding a setpoint parameter $t$ which, in this case, is the position of $O_s$ with respect to $O_m$. The parameter $t$ can be set according to a goal, in this case, to align a point $q$ in $c_s$ with a point $p$ in $c_m$, as shown in Figure 2.2b. The transformation function is thus defined as $f_{c_s \rightarrow c_m}(x; t) = M_x \cdot x + t$.

**Figure 2.2:** Simple example of setpoint: (a) position of the stage when the origins coincide; (b) movement of the stage by $t$ units to align $p$ and $q$.

A schematic view of the ASML machine is shown in Figure 2.3. A reticle $R$ is mounted on the reticle stage $RS$. Similarly, a wafer $W$ is mounted on the wafer stage $WS$. We want to project a point $p$ from $R$ onto $W$. In order to achieve this, $p$ will be positioned in the same location as point $q$ from the object plane $OP$, by calculating a setpoint translation and rotation to change the state of $RS$. As soon as $p$ is located in the correct position, it is geometrically projected onto the image plane $IP$ as point $r$ through the projection optics box $POB$. A similar process is carried out on $W$ to position a point on it at the same location as $r$, but is not shown in the figure.

The three kinds of transformation functions mentioned above can be identified in Figure 2.3. The coordinates of point $p$ in the reticle coordinate system (RCS) have to be transformed to reticle stage coordinate system (RSCS) in order to determine how the reticle stage
has to be moved. Therefore, the transformation function from RCS to RSCS is of type (i). The motion of the reticle stage is parameterized by \( t \) and \( \alpha \) which represent translation and rotation setpoints, respectively. Thus, the transformation function from RSCS to metroframe coordinate system is of type (iii). When point \( p \) is located at the same position as \( q \), it is projected onto \( IP \). The transformation function from \( OP \) coordinate system to \( IP \) coordinate system is of type (ii), since \( r \) is the geometric projection of \( q \).

\[ \]  

Formally, we are interested in the following two problems. Given are a transformation model with a set of coordinate systems \( C \) and a set of transformations \( T \):

- For a parameterized \( f_{a \rightarrow b} \in T \), let \( S \) be the set of setpoint parameters of \( f_{a \rightarrow b} \) (such as scaling, orientation or translation). Find values of the setpoint parameters in \( S \), such that \( y^b = f_{a \rightarrow b}(x^a; S) \), given \( x^a \) and \( y^b \). In other words, find a desired state of the moving component associated with \( a \), such that \( x \) coincides with \( y \). It is assumed that all the parameters unrelated to setpoints are known.

Figure 2.3: Schematic view of the ASML machine with an example of setpoint calculation and alignment of points on a reticle and a wafer
• For a transformation $f_{a \rightarrow b} \in T$, find $y^b$ given $x^a$, such that $y^b = f_{a \rightarrow b}(x^a)$. It is assumed that all the parameters are known. The point $y$ can be the same as $x$ or not, depending on the kind of transformation.

2.2 Considerations for a solution

A solution for the problem previously stated needs to take into account some constraints and limitations of the ASML machine. In order to select a computational model for the concepts defined in the problem statement, it is necessary to know how they change over time. In Section 2.2.1 it is shown how often transformation models, transformation states and parameters vary.

The components of the machine can suffer effects from external phenomena and can have mechanical limitations. In Section 2.2.2 it is explained how such physical limitations can constrain a possible solution for the problem.

Besides physical limitations there are potential mathematical limitations such as numerical issues which are intrinsic to computers, or non-deterministic behavior due to degrees of freedom in mathematical models. In Section 2.2.3 these limitations are described.

2.2.1 Frequency of variability

A computational solution for the concepts presented in Section 2.1 has to take into account how often they change over time. This is specially relevant when deciding to generate code for parts of the solution in order to improve efficiency.

• A transformation model is only modified when: new parameters need to be considered, new components are added to the machine, better mathematical models are found.

• Non-setpoint parameters change (and therefore the transformation state), for instance, when a reticle or wafer is replaced. This causes new calibrations to be performed and therefore the parameters in the transformation state get new values. We say that a transformation state changes, when at least one of its parameter values is changed.

• Setpoint parameters are also part of the transformation state, but have a different frequency of variability. The setpoint parameters are changed every time that a component needs to be moved. For example, when performing an exposure scan, different regions of the reticle have to be exposed. For each region a new setpoint is calculated.

Variability over time is relevant in Chapter 6 where it is discussed how software efficiency and type-safety can be improved by generating code.

2.2.2 Physical limitations of the machine

The ASML machine has mechanical parts, thus mathematical models of the machine have to take into account real world limitations:

• A transformation state is a mathematical representation of a state of the physical system. As such, the range of values that the parameters can get is usually restricted. Each parameter can have values within a specific range.
• The points are restricted to \( \mathbb{R}^2 \) or \( \mathbb{R}^3 \).

• The solution of a setpoint can be mathematically valid, while due to mechanical limitations, it can be not physically feasible.

### 2.2.3 Mathematical limitations

There are mathematical limitations due to numerical issues or the mathematical methods to solve setpoints:

• In Figure 2.3 there is only one solution: move reticle stage according to \( t \); and rotate reticle stage according to \( \alpha \). There are no degrees of freedom. However, in most cases, there are several solutions to solve the same problem. In Figure 2.4, the wafer stage can rotate about the \( z \) axis freely, giving an infinite number of solutions. To give only one solution, the user can specify a fixed value for rotation on the \( z \) axis. The user can also specify either the \( x \) or \( y \) component of the translation vector of the wafer stage from the initial to the final position.

• Another option is to solve for more than one point. This constrains the problem and can result in only one solution. However, it is possible that this constrains the problem so much that there is no exact solution anymore. In this case, it is desirable to find a solution which is as close as possible to the optimal. To solve this, external specialized libraries can be used.

• It is known that computers have numerical limitations due to finite representation of numbers. Due to this, some mathematical operations can increase numerical errors when evaluated in a certain way, whereas a mathematically equivalent evaluation can give a smaller error. This limitation is not addressed in this project. More information can be found in [7].

### 2.2.4 Types of users

According to the purpose and variability of usage, in general a DSL has different types of users who affect it in some way. The three types of users are the following:

**Maintainer** Creates and maintains the DSL. This type of user defines the metamodel and the implementation of the DSL functionality.

**Modeler** Creates an instance of the transformation model. In other words, this user provides model data and uses a DSL syntax to create a computable model.

**Client** Uses a specific model that the modeler has created. For example, this is the type of user that requests a setpoint calculation. This can be an external user or a software component.
Figure 2.4: Infinite setpoint solutions as the wafer stage can rotate freely along the z axis
Chapter 3

Case study

3.1 Introduction

In order to show clearly how usable, maintainable and extensible a DSL is, use case scenarios and unit testing will be defined.

The use case scenarios will be useful to illustrate the different approaches of making a model machine processable or using it, as well as the trade-offs involved.

The unit testing is useful to show how robust a DSL implementation is, by verifying that the properties defined in the high-level specification always hold.

3.2 Model description

The physical world will be initially represented in two dimensions, therefore, we will have to handle \((x, y)\) coordinates only. The set of coordinate systems initially consists of the following: RCS, RSCS, RZCS (metroframe related to reticle stage), WZCS (metroframe related to wafer stage), WCS and WSCS.

In general, a transformation function from coordinate system \(A\) to coordinate system \(B\) \((A \rightarrow B)\) is represented as:

\[
x^B = R_A \cdot M_A \cdot x^A + c_A
\]

The elements of the transformation function are as follows:

- \(x^A\): point with respect to coordinate system \(A\).
- \(R_A\): rotation matrix of the following form:

\[
R_A = \begin{pmatrix}
\cos R_z & -\sin R_z \\
\sin R_z & \cos R_z
\end{pmatrix}
\]

where \(R_z\) is a rotation parameter from a transformation state.

- \(M_A\): magnification matrix of the form:

\[
M_A = \begin{pmatrix}
M_x & 0 \\
0 & M_y
\end{pmatrix}
\]
where $M_x$ and $M_y$ are the magnification factors in $x$ and $y$. They are included in a transformation state.

- $c_A$: translation vector of the form:

$$c_A = \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

where $t_x$ and $t_y$ are the translations in $x$ and $y$. They are included in a transformation state.

The transformation functions are: RCS $\rightarrow$ RSCS, RSCS $\rightarrow$ RZCS, RZCS $\rightarrow$ WZCS, WZCS $\rightarrow$ WCS. The reticle stage setpoint is in RSCS $\rightarrow$ RZCS in the parameters of $R_{RZCS}$ and $c_{RZCS}$. The wafer stage setpoint is in WZCS $\rightarrow$ WCS in the parameters of $R_{WSCS}$ and $c_{WSCS}$.

The values that all the parameters can take are limited to a range. In Table 3.1 the range of values for each parameter is shown.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_x$ [RCS]</td>
<td>Reticle translation $x$</td>
<td>$\pm 100 \mu m$</td>
</tr>
<tr>
<td>$t_y$ [RCS]</td>
<td>Reticle translation $y$</td>
<td>$\pm 100 \mu m$</td>
</tr>
<tr>
<td>$M_x$ [RCS]</td>
<td>Reticle magnification $x$</td>
<td>$\pm 10$ ppm</td>
</tr>
<tr>
<td>$M_y$ [RCS]</td>
<td>Reticle magnification $y$</td>
<td>$\pm 10$ ppm</td>
</tr>
<tr>
<td>$R_z$ [RCS]</td>
<td>Reticle rotation $R_z$</td>
<td>$\pm 0.5$ mrad</td>
</tr>
<tr>
<td>$t_x$ [RSCS]</td>
<td>Reticle stage $x$ set-point</td>
<td>$\pm 2.5$ mm</td>
</tr>
<tr>
<td>$t_y$ [RSCS]</td>
<td>Reticle stage $y$ set-point</td>
<td>$\pm 130$ mm</td>
</tr>
<tr>
<td>$M_x$ [RSCS]</td>
<td>Reticle SPM scaling $x$</td>
<td>$\pm 10$ ppm</td>
</tr>
<tr>
<td>$M_y$ [RSCS]</td>
<td>Reticle SPM scaling $y$</td>
<td>$\pm 10$ ppm</td>
</tr>
<tr>
<td>$R_z$ [RSCS]</td>
<td>Reticle stage $R_z$ set-point</td>
<td>$\pm 1.2$ mrad</td>
</tr>
<tr>
<td>$t_x$ [RZCS]</td>
<td>Reticle to metro level translation $x$</td>
<td>$\pm 112$ $\mu m$</td>
</tr>
<tr>
<td>$t_y$ [RZCS]</td>
<td>Reticle to metro level translation $y$</td>
<td>$\pm 112$ $\mu m$</td>
</tr>
<tr>
<td>$M_x$ [RZCS]</td>
<td>Reticle to metro level scaling $x$</td>
<td>$0.25 \pm 30$ ppm</td>
</tr>
<tr>
<td>$M_y$ [RZCS]</td>
<td>Reticle to metro level scaling $y$</td>
<td>$0.25 \pm 30$ ppm</td>
</tr>
<tr>
<td>$R_z$ [RZCS]</td>
<td>Reticle load offset $R_z$</td>
<td>$\pm 0.70$ $\mu m$</td>
</tr>
<tr>
<td>$t_x$ [WZCS]</td>
<td>Wafer stage $x$ set-point</td>
<td>$\pm 200$ mm</td>
</tr>
<tr>
<td>$t_y$ [WZCS]</td>
<td>Wafer stage $y$ set-point</td>
<td>$\pm 200$ mm</td>
</tr>
<tr>
<td>$M_x$ [WZCS]</td>
<td>Wafer SPM scaling $x$</td>
<td>$\pm 10$ ppm</td>
</tr>
<tr>
<td>$M_y$ [WZCS]</td>
<td>Wafer SPM scaling $y$</td>
<td>$\pm 10$ ppm</td>
</tr>
<tr>
<td>$R_z$ [WZCS]</td>
<td>Wafer stage $R_z$ set-point</td>
<td>$\pm 200$ $\mu$rad</td>
</tr>
<tr>
<td>$t_x$ [WSCS]</td>
<td>Wafer load offset $x$</td>
<td>$\pm 12$ $\mu m$</td>
</tr>
<tr>
<td>$t_y$ [WSCS]</td>
<td>Wafer load offset $y$</td>
<td>$\pm 12$ $\mu m$</td>
</tr>
<tr>
<td>$M_x$ [WSCS]</td>
<td>Wafer magnification $x$</td>
<td>$\pm 10$ ppm</td>
</tr>
<tr>
<td>$M_y$ [WSCS]</td>
<td>Wafer magnification $y$</td>
<td>$\pm 10$ ppm</td>
</tr>
<tr>
<td>$R_z$ [WSCS]</td>
<td>Wafer load offset $R_z$</td>
<td>$\pm 120$ $\mu$rad</td>
</tr>
</tbody>
</table>

Table 3.1: Ranges of parameter values for the initial model

The model will be used to calculate the reticle and wafer setpoints. The following three cases will be considered:
• TIS points: Given are three points in RCS, RZCS and WCS. The possible x and y values for these points are given in Table 3.2. Since the reticle stage has limited movement in x, the x coordinate of a RZCS point is the same as the x coordinate of a RCS point.

• Simple exposure points: Given the TIS points described previously, add four points to each mark in the reticle and wafer. The new points are placed at the same distance as the original point, 700 µm for reticle and 175 µm for wafer. For example, if a TIS mark in the reticle has coordinates (0, 0), the new points will have coordinates (−700, 0), (700, 0), (0, −700) and (0, 700). All coordinate values are in µm units.

• Real exposure points: Given are sets of points for RCS, RZCS and WCS, which are taken from real exposures.

<table>
<thead>
<tr>
<th>Coordinate system</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCS</td>
<td>−48 mm</td>
<td>−69.5 mm</td>
</tr>
<tr>
<td>RCS</td>
<td>0 mm</td>
<td>−69.5 mm</td>
</tr>
<tr>
<td>RCS</td>
<td>+48 mm</td>
<td>−69.5 mm</td>
</tr>
<tr>
<td>RCS</td>
<td>−48 mm</td>
<td>+69.5 mm</td>
</tr>
<tr>
<td>RCS</td>
<td>0 mm</td>
<td>+69.5 mm</td>
</tr>
<tr>
<td>RCS</td>
<td>+48 mm</td>
<td>+69.5 mm</td>
</tr>
<tr>
<td>RZCS</td>
<td>−48 mm</td>
<td>0 mm</td>
</tr>
<tr>
<td>RZCS</td>
<td>0 mm</td>
<td>0 mm</td>
</tr>
<tr>
<td>RZCS</td>
<td>+48 mm</td>
<td>0 mm</td>
</tr>
<tr>
<td>WCS</td>
<td>−12 mm</td>
<td>0 mm</td>
</tr>
<tr>
<td>WCS</td>
<td>+12 mm</td>
<td>0 mm</td>
</tr>
</tbody>
</table>

Table 3.2: Possible coordinates for TIS marks

3.3 Use case scenarios

• Change transformation function WCS → WCS to: \( x^{WCS} = M_{WCS}^{-1} \cdot R_{WCS}^{-1} \cdot (x^{WCS} - c_{WCS}) \). All the operators remain the same, the change goes in the shape of the transformation function. This shows how more complex transformation functions can be defined.

• Add a new coordinate system RPCS. After this change it must be possible to calculate setpoints given points in RPCS. This change reflects the fact that in the ASML machine there are actually multiple plates (among which the reticle is just one) on which scans can be done. A new transformation function must be added: RPCS → RSCS, which has the same shape as RCS → RSCS with the same parameters, except that the y translation is not centered on 0 but on 127.5 mm, −112.5 mm or −138.5 mm. This shows that a new coordinate system can be integrated in the model and can handle the complexity added by a new transformation function (and therefore a new path) to RSCS.

• Add a height map function \( f : \mathbb{R}^2 \rightarrow \mathbb{R}^2 \) (from an external library) in the transformation from RCS → RSCS. The external library takes a point in RCS and gives a translation vector back. The resulting transformation function turns to: \( x^{RSCS} = R_{RCS} \cdot M_{RCS} \cdot \).
$x^{RCS} + c_{RCS} + f(x^{RCS})$. Note that this external function is not really a height map; it is a distortion map that gives an additional translation, depending on the position on the reticle. This change shows how external functions can be treated as a normal (non-linear) operator in a transformation function.

- Change transformation function $RSCS \rightarrow RZCS$ to: $x^{RZCS} = R^{RSCS} \cdot R_{RSCS} \cdot (x^{RSCS} - M_{RSCS} \cdot c_{RSCS} + c_{RSCS}E)$. This change involves, besides a change in the shape of the transformation function, new parameters (reticle stage zeroing errors). The translation parameters of $c_{RSCS}E$ are in the range $\pm 12 \mu m$ and the rotation parameter of $R_{RSCS}$ is in $\pm 70 \mu rad$. This modification illustrates that it is possible to add new operators with new parameters to a transformation function.

- Add the $OCS$ and $ICS$ coordinate systems: $RZCS \rightarrow OCS \rightarrow ICS \rightarrow WZCS$. Add a distortion function (from an external library) in the $OCS \rightarrow ICS$ transformation. This works the same as a height map. This shows that it is possible to add an external function (distortion) in a similar way as a new operator.

- Add the $z$ dimension. All transformations should be able to handle the $z$ dimension. The magnification matrices and translation vectors will get an additional parameter for the $z$ dimension. Also, there will be two additional rotation parameters. The two new ones are $R_x$ and $R_y$. The default rotation order in the transformations will be: $R = rot(R_z) \cdot rot(R_x) \cdot rot(R_y)$.

### 3.4 Unit testing

After creating and modifying a transformation model it is important to ensure that the correct behavior is preserved. One way to achieve this is to do unit testing. If after each modification all tests pass, then we are confident that the model still behaves well.

The following are the test cases that need to be true all the time:

- Given a set of points $x_i^{RCS}$, $y_i^{RZCS}$, $z_i^{WCS}$, for $i = 1, 2, \ldots, m$, $m > 0$; and a transformation state $TS$ with all non-setpoint parameters known. Calculate the reticle setpoint using $x_i^{RSCS}$, $y_i^{RZCS}$ and $TS$ as input. Calculate the wafer setpoint using $y_i^{WZCS}$, $z_i^{WSCS}$ and $TS$ as input. Update $TS$ with the new setpoint parameters. Check that $|x_i^{RZCS} - y_i^{RZCS}| < \varepsilon$ and $|x_i^{WCS} - z_i^{WCS}| < \varepsilon$, for some $\varepsilon \approx 0$.

- Given a point $x_i^{RCS}$ and a transformation state $TS$ with all parameters known, transform $x_i^{RCS}$ to $x_i^{WCS}$. Take $x_i^{WCS}$ and transform it to $x_i^{RCS'}$. Check that $|x_i^{RCS} - x_i^{RCS'}| < \varepsilon$, for some $\varepsilon \approx 0$.

- The values of the setpoint parameters are always within the ranges in Table 3.1

The values of the points are according to Table 3.2 and the values of the transformation state parameters are calculated randomly, using values within the ranges shown in Table 3.1.
Chapter 4

Domain concepts and interrelations

4.1 Introduction

Regardless of the type of DSL to be constructed, the first and most important step of the DSL construction process involves representing the domain knowledge in a formal way. This can be accomplished by defining what we call the metamodel (see Section 1.3.1), which describes the domain concepts and their interrelations. In this project, the metamodel is a generic description of any transformation model.

The development of the metamodel should be as independent of the other components of a DSL as possible. This allows to decouple a particular DSL grammar or a code generator from the metamodel logic.

In some cases, the metamodel takes the form of an API or a library in an object-oriented language. In this chapter, we will use the terms API and metamodel interchangeably, but it is important to note that an API is a specific implementation of a metamodel. Other ways to represent it include the Ecore metamodel from the Eclipse Modeling Framework (EMF) \[2\]. The concepts are represented as abstract data types and the interrelations are represented by associations between them. In fact, it is the first approach that we take in this document.

Good API design techniques are important in this stage, such as a clear separation between the interfaces and the classes that implement them \[3\]. This allows a declarative approach where it is more important what the functionality and operations are, rather than how they are implemented. The particular implementation of the abstract data types is treated as a black box and should not be reflected in the API specification (information hiding principle).

Among the advantages that using an API as metamodel offers, we can list the following:

- The API specification can be used as channel of communication between programmers and domain experts. Usually, an API specification only occupies a few pages, which can be easily read and refactored.

- The implementation details are not relevant in the specification of the interfaces, which allows to focus on the high-level functionality.

- The separation of implementation and interface allows to develop and test different implementations in an independent way.

In this chapter, the specific requirements that the metamodel should satisfy are listed in Section 4.2. The high-level specification based on the requirements is presented in Sec-
A possible implementation of the metamodel is shown in Section 4.4. The importance of identifying the static and dynamic parts of the metamodel is discussed in Section 4.5. And finally, some conclusions on the process of creating a metamodel are presented in Section 4.6.

4.2 Functional Requirements

After discussion with the domain experts at the metrology department, the metamodel should have as minimal functional requirements:

REQ1 It must allow the definition of a transformation between two coordinate systems as a mathematical expression. The transformation should contain the following elements:

- Named transformation matrices which are composed of named parameters without specific values.
- One or more placeholders for the point to be transformed. For example, in the expression \( \mathbf{R} \cdot \mathbf{M} \cdot \_ + \mathbf{c} \), the symbol \( \_ \) takes the value of the input point to be transformed.

REQ2 It must allow to calculate setpoints from a set of transformations, where at least one contains setpoint parameters. The calculated setpoint when included in the corresponding transformations must allow to align one point from the reticle with one point from the wafer.

REQ3 It should provide an interface to input parameter values in order to transform points and calculate setpoints. The set of pairs (parameter name, parameter value) is the transformation state.

REQ4 It should provide an interface to input a point with respect to a coordinate system, which can be transformed to another coordinate system or can help to calculate a setpoint.

REQ5 It could provide support for debugging by tracing back in a transformation sequence. This requirement is left as future work (See Section 7.4).

REQ6 It could help to describe how parameter changes affect the measurements. This is currently done using derivative matrices which show how a parameter changes with respect to another. This requirement is left as future work (See Section 7.4).

REQ7 It could allow to calculate setpoints for an exposure scan, which is described as a trajectory. This requirement is left as future work (See Section 7.4).

4.3 High-level specification

The domain concepts and the requirements can be put together in an API, where all the functionality that the DSL should provide can be specified in a clear and concise way. In order to maximize the flexibility of the metamodel, the interfaces should not give any clue
about specific implementation details. This also makes easy the communication with other components that use the interfaces.

The high-level specification provides a neat and easy way to discuss the requirements and functionality with the domain experts. When a change needs to be done in the metamodel, it can be quickly done by refactoring the API specification.

The implementation associated to the interfaces usually takes place once the high-level specification is clearly defined and all the stakeholders involved have agreed on the functionality.

We can divide the interfaces in two types $\square$: population interface which is used to provide model-specific data in order to create a model; and the operational interface which can be used by clients once a model is created.

4.3.1 Population interface

The population interface is the set of constructors of the types which correspond to concepts from the domain. As mentioned above in Section 1.3.1 a metamodel is the description of a set of models and a model is an instance of the metamodel. The population interface is used to create a model.

The abstract data types associated to the concepts of the domain along with their attributes and invariants are shown next:

**CoordinateSystem** Represents a cartesian reference system in the ASML machine for points in a component. The points can be in $\mathbb{R}^2$ or $\mathbb{R}^3$.

**Attributes**: name: String.

**Invariants**: The name of a coordinate system is unique in a transformation model.

**MatrixAbstract** A matrix, in the linear-algebraic sense, which represents a linear operator in a transformation function, or the coordinates of a point in vector form. A matrix can be added, subtracted, multiplied with another matrix, and can be inverted and negated as well.

**Attributes**: values: 2D array of floating-point values.

**Invariants**: The number of rows and columns of a matrix are both positive integers.

**Point** Represents an element of the space containing a vector of values which represent the coordinates with respect to a coordinate system.

**Attributes**: cs: CoordinateSystem, coordinates: MatrixAbstract.

**Invariants**: A point is always associated with a coordinate system.

**TransformationState** A set of parameters represented by pairs of the form $(name, value)$. These parameters are used by the transformation function in order to transform, or calculate a setpoint. Along with the value, a valid range has to be defined.

**Attributes**: parameters: Set of pairs $(name : String, value : floating-point)$.

**Invariants**: The values of all the parameters in a transformation state are within a predefined range.
**TransformationFunction**  A linear or non-linear function which, given a point with respect to a source coordinate system as input and a transformation state, transforms the point to another target coordinate system.

**Attributes:** Set of parameter names. Its shape is implementation dependent.

**Invariants:** All the parameter names in a transformation function must also be in the transformation state.

**TransformationModel** Represents the metamodel of the transformation models. An instance of a transformation model consists of a set of coordinate systems, and a set of transformations. Each transformation is associated to two coordinate systems. A transformation model can be thought of as a graph, where the coordinate systems are the nodes and the transformation functions are the edges. An edge has a source and a target and can be interpreted as transforming points from source to target coordinate systems. It can be possible to transform from target to source, by applying the inverse transformation if it is defined. An edge can be described by two directed edges, forward and backward. However, for simplicity it is described as two-way undirected.

**Attributes:** A set of coordinate systems and a set of transformation functions.

**Invariants:**

- Every transformation function must be associated to two different coordinate systems.
- There must not be cycles in any path of transformation functions.
- There must be exactly one undirected path connecting any two coordinate systems through transformation functions.

**SetpointSolver** Represents a setpoint solver. This interface is a way to abstract from the different ways to calculate a setpoint in a transformation model.

### 4.3.2 Operational interface

The operational interface contains all the operations that a client can use, when a model is already created. The signatures of the operations according to the use cases are as follows:

- Transform a point input, which is defined in a coordinate system, with respect to another coordinate system target, given a transformation state state.

  \[
  \text{transform(input: Point, target: CoordinateSystem, state: TransformationState): Point}
  \]

**Preconditions:**

- There must be one path \( p \) from the coordinate system associated to input to target.
- All the parameters associated to the transformation functions in \( p \) must have a known value in state.

**Postconditions:** The output point has the coordinates of the input point after being transformed. The output point is associated with target.
• Calculate the setpoint of a transformation function given a list of points in the source coordinate system and the same number of points in the target coordinate system. A transformation state is also given.

\[
\text{updateTransformationStateWithSetpoint(source: Point[], target: Point[], state: TransformationState): TransformationState}
\]

**Preconditions:**

- `source` and `target` contain at least one point each, and both contain the same number of points.
- All points in `source` must be associated to the same coordinate system. Similarly with `target`.
- There must be a transformation function between the coordinate system associated with `source` and the coordinate system associated with `target`.
- The transformation function must have setpoint parameters.
- All the non-setpoint parameters in the transformation function must have a known value in `state`.

**Postconditions:** The output transformation state contains all the parameters as `state` with the same values, but the setpoint parameters of the transformation function get updated.

### 4.4 Metamodel

The high-level specification represents the functionality in terms of the use cases and domain concepts. However, to be able to represent this knowledge in a computer, data structures and algorithms need to be used to implement such specification. In this section, the design and implementation of the concepts are described.

Each interface has a class associated where the operations are implemented. Each operation needs to satisfy the contracts defined in the first step of the specification, and therefore extensive testing is necessary to guarantee this. The test cases are described later in this document.

The architecture of the metamodel is shown in Figure 4.1. The transformation model is implemented as a directed graph, where the nodes are coordinate systems, and the edges are transformation functions. In order to transform a point, the transformation functions are explored in a breadth-first order in order to find a shortest path and then do the calculation. This ensures that the minimum number of transformations are applied.

A transformation function is implemented as an expression tree. The metamodel of the transformation functions is shown in Figure 4.2.

An internal node in a transformation function tree has one child in case of a negation or an inversion, or two children in case of a binary operation. The possible binary operations are addition, substraction and multiplication. Each leaf has a set of parameter names from the associated transformation state. The leaf types can be: translation, rotation, magnification or input. An input type represents the place where an input point to be transformed is placed inside the transformation function.
Figure 4.1: Metamodel of the transformation models
Figure 4.2: Metamodel of the transformation trees
4.5 Dynamic and static components

In the previous chapters, it has been stated that domain-specific modeling helps to increase the productivity of the developers and the communication with domain experts. It is more natural to think in terms of the domain concepts. The abstract data types that represent the concepts and the operations can be defined in a metamodel, but it needs to be transformed to operational software.

The operational software is the way a particular model is executed in the computer. In this document two alternatives will be shown: using an in-memory model-independent representation that interprets model-specific data at runtime (interpreter); and generating model-specific code (code generation). The latter option implies making decisions about which parts of the model are static or dynamic.

As shown in Section 2.2, different parts of the software have different frequency of variability. According to this, one can decide what can be generated statically as code:

- Transformation model: the structure of the transformation graph, i.e. coordinate systems and transformation functions, has a low variability. Therefore, it can be statically generated.

- Transformation state: the parameters of the transformation state change values whenever a new calibration is carried out. Statically generating code for parameter values, would imply that for each calibration a new code generation has to be executed. This is a costly process, therefore, it is reasonable to leave the transformation state as dynamic.

- Setpoints: the setpoints are calculated more frequently than the other parameters in a transformation state. Therefore, it is left as dynamic.

The discrimination between static and dynamic components is only relevant when doing code generation. If the operational software is represented as an interpreter, all the components run in memory.

4.6 Conclusion

In this chapter, it was emphasized that the most important component of a domain-specific modeling effort is the metamodel. The DSL and operational software are the other components which depend on the metamodel, but are less critical.

There are different ways to represent a metamodel. In this chapter, it was shown how it can be represented as an API. The API consists of abstract data types representing the domain concepts. The use cases are operations in these types. Good API design techniques such as information hiding, good naming, simplicity, avoid explicit implementation constraints or clear and concise contracts are crucial.

The communication between software developers and domain experts is enhanced by the domain-specific abstraction. The API interfaces can be used as a tool to discuss the requirements and functionality.
Chapter 5

Making transformation models machine processable

5.1 Introduction

The mathematical description of transformation models was described precisely in Chapter 2. In this chapter, we describe the different alternatives to bring the mathematical definition of transformation models into the computational world, as models which conform to the metamodel described in Chapter 4. Once the metamodel and its semantics are defined, it can be used to create models. As it was shown in Chapter 4, the metamodel interfaces have constructors and operations that can be called from outside. With this in mind, a DSL is just a thin layer on top of the metamodel, whose purpose is to create a model using such constructors, in a more readable way. Since a DSL only calls the constructors, its implementation is less costly than the metamodel itself.

In this chapter, various alternatives for instantiating the metamodel and their trade-offs are discussed. In Section 5.2 it is shown how the constructors of the metamodel can be used directly. In this case, the DSL is a subset of the GPPL in which the metamodel API is implemented. In Section 5.3 an internal DSL is shown, where the constructor calls are embellished to make them more fluent, i.e. more similar to natural language. In Section 5.4 an external DSL is shown, where a custom grammar and parser are defined.

5.2 Direct use of API constructors

The structure of the metamodel as shown in Figures 4.1 and 4.2 has all the ingredients to build a model. In this section it is shown how the model defined in Chapter 3 is created using only the constructors. The syntax used in this chapter corresponds to Java, however the process should be similar in other object-oriented languages.

5.2.1 Transformation model

The transformation model is created first as an object, and through helper methods, the other ingredients are added later. The code to create a transformation model is as follows:

```java
TransformationModel model =
    GraphTransformationModel.newModel(new BinsearchSetpointSolver());
```
The transformation model is implemented as a graph, and takes as a parameter an object that implements the setpoint solver. It can also be set at runtime via a setter. The choice of leaving the setpoint solver as a parameter (or setter) allows to use different implementations.

5.2.2 Coordinate systems

The coordinate systems are instantiated as follows:

```java
CoordinateSystem rcs = new CoordinateSystem("RCS");
CoordinateSystem rpcs = new CoordinateSystem("RPCS");
CoordinateSystem rscs = new CoordinateSystem("RSCS");
CoordinateSystem rzcs = new CoordinateSystem("RZCS");
CoordinateSystem wzcs = new CoordinateSystem("WZCS");
CoordinateSystem wscs = new CoordinateSystem("WSCS");
CoordinateSystem wcs = new CoordinateSystem("WCS");
```

A coordinate system could be represented just as a string. The choice of defining a new `CoordinateSystem` type makes the type consistency easier, as a particular coordinate system needs to be instantiated before using it. Furthermore, it lets open the possibility to add new features to a coordinate system.

5.2.3 Transformation functions

The metamodel of Figure 4.2 describes the structure of transformation functions. The representation is a tree, where the internal nodes represent unary (negation or inverse) and binary operators (plus, minus, times). The leaves of the tree contain operators, which have associated a number of parameters, according to their type. The types are translation, magnification, rotation or input.

The transformation functions of the example in Chapter 3 are of the form: \( y = R \cdot M \cdot x + c \). Several instances, one for each transformation, are created, but only the first one is shown, i.e.

```java
RotationOperator R = new RotationOperator("Rz_RCS_RSCS",
                                          "R_RCS_RSCS",
                                          setpoint);
MagnificationOperator M = new MagnificationOperator("Mx_RCS_RSCS",
                                                      "My_RCS_RSCS",
                                                      "M_RCS_RSCS",
                                                      false);
TranslationOperator c = new TranslationOperator("tx_RCS_RSCS",
                                               "ty_RCS_RSCS",
                                               "c_RCS_RSCS",
                                               setpoint);
InputOperator x = new InputOperator();
```

The names of the operators are R_RCS_RSCS, M_RCS_RSCS, c_RCS_RSCS. They are used as a key to refer to the object they represent in the tree. The names of the transformation state parameters are Rz_RCS_RSCS, Mx_RCS_RSCS, My_RCS_RSCS, tx_RCS_RSCS, ty_RCS_RSCS. The values of these parameters need to be in a transformation state that uses this specific transformation function. The last parameter in the constructors determines whether that
operator is a setpoint or not. The value setpoint is true for transformations from RSCS to RZCS and from WZCS to WSCS.

The second step is to create the transformation function itself. The code is as follows:

```java
TransformationFunction forward = new TimesNode(R, M);
forward = new TimesNode(forward, x);
forward = new PlusNode(forward, c);
```

This is only one way to construct the same expression, but there can be more than one. The TransformationComposite constructor builds a tree, attaching the first parameter as the left subtree, the second parameter as the right subtree, and assigning the binary operator of the third parameter.

A slightly more complicated example involves generating the inverse transformation $x = M^{-1} \cdot R^{-1} \cdot (y - c)$:

```java
TransformationFunction inverse = new InversionNode(M);
TransformationFunction invR = new InversionNode(R);

inverse = new TimesNode(inverse, invR);

InputOperator y = new InputOperator();
TransformationFunction yMinusc = new MinusNode(y, c);

inverse = new TimesNode(inverse, yMinusc);
```

The shape of the transformation functions has an impact on the length and complexity of the code. For clarity, it was chosen to show one call to a constructor per line, but they can be nested. The construction, when done in several steps can be risky in cases where concurrency is involved, as it is not an atomic operation.

The following code adds the coordinate systems and transformation functions just created to the model:

```java
model.setTransformation(rcs, rscs, forward);
model.setTransformation(rscs, rcs, inverse);
```

The same code can be repeated for the other transformation functions, but it will not be shown for brevity.

### 5.3 Internal DSL

Building a transformation function using constructors is a tedious task. A better way to do it would be by writing an expression that resembles the original mathematical expression. This can be accomplished by modifying the language that hosts the DSL, in this case Java, so that the syntax is more fluent.

The internal DSL pattern called expression builder helps in this regard. The idea is to use a class which internally contains a temporary TransformationFunction instance. This instance is updated every time a new subexpression is added. When all the elements of the expression are in place, the final transformation function is built. The advantages of this approach are: makes easier the construction of a transformation function; and ensures atomicity by building the transformation function only in the end.

Using the same operators defined in the previous section, the code to generate the same transformation functions is as follows:
The syntax is intuitive and reflects closely the transformation function in the mathematical way. The elements of the expression are explained:

- \texttt{var(a)}: Initializes a new tree taking as a parameter an operator \texttt{a}.
- \texttt{a.times(b)}: Equivalent to \texttt{new TransformationComposite(a, b, TIMES)}.
- \texttt{a.plus(b)}: Equivalent to \texttt{new TransformationComposite(a, b, PLUS)}.
- \texttt{a.minus(b)}: Equivalent to \texttt{new TransformationComposite(a, b, MINUS)}.
- \texttt{a.inv()}: Equivalent to \texttt{new InvertedTransformation(a)}.
- \texttt{a.neg()}: Equivalent to \texttt{new NegatedTransformation(a)}.

The advantage of using an internal DSL is that all the features of the host language are available. The API constructors can be mixed with the embellished syntax of the internal DSL.

On the other hand, when a model is changed the software in the host language needs to be recompiled, since the model definition is hardcoded there.

### 5.4 Ad-hoc external DSL

The previous approaches use the GPPL in which the metamodel API is implemented in order to create a model. In this section, a new language with a custom syntax and parser to describe transformation functions is shown. This is called an \textit{ad-hoc external DSL}. The \textit{ad-hoc} part comes from the fact that a parser is handcrafted, as opposed to the case when external tools are used to define grammars (e.g. ANTLR [5] or EMF [2]). This is called plainly an \textit{external DSL}. The creation of a new language adds more complexity and costs, since a new grammar has to be defined and maintained, as well as the translation from the abstract syntax tree to an instance of the metamodel. This approach, however, is a flexible way to define a language that fits the user needs, and reduces the noise associated to the host GPPL.

The steps to implement an external DSL are: grammar definition, parsing and instantiation of metamodel. There are tools that allow to create a grammar in EBNF form and translate the abstract syntax tree to objects, such as ANTLR [5] and Xtext [6]. In this section an ad-hoc parser is described as only the transformation functions are processed. The DSL expressions are encoded as strings, which are processed by a parser written in the GPPL. For more complex DSLs, one of the tools previously mentioned is recommended.

The example transformation functions can be written in the external DSL like this (operators is a list that contains the leaves of the tree):

\begin{verbatim}
TransformationFunction forward = parse("R*M*_+c", operators);
TransformationFunction inverse = parse("M'*R'*(-c)", operators);
\end{verbatim}
The external DSL expressions are input as strings to a parser function. The expressions are written almost in the same way as they would be written on paper. The operator names should be the same as they were defined in their constructors, but it is shown here in a simpler way, e.g., $R$ instead of $R_{\text{RCS}}R_{\text{RCS}}$. The character $'$ indicates the inverse, and $\_\_$ represents a placeholder for the input point. The associativity and precedence rules for the operators work in the same way as in standard arithmetic, and determine the structure of the transformation function tree.

Adding new functionality or new elements to an external DSL implies changing the grammar, parser and the code to instantiate a model. However, when these costs are surpassed by the benefits, the external DSL turns out to be a simple and flexible way to create a model.

### 5.5 Conclusion

In this chapter, several ways to create a transformation model were described. The pros and cons of all the approaches are summarized in Table 5.1.

<table>
<thead>
<tr>
<th>Method</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| Direct use of API constructors | * No extra costs associated to parsing.  
|                     | * Usage of host language features.        | * Often yields unreadable code.           |
|                     | * The construction of an object is not always an atomic operation. |                                           |
| Internal DSL        | * Yields readable code.                   | * Extra cost to build and maintain the expression builder. |
|                     | * The construction of an object is an atomic operation. | * Model changes imply code recompilation. |
| Ad-hoc external DSL | * Can be adjusted to user needs.         | * Implementing and maintaining a parser can be expensive. |
|                     | * There is no noise associated to a host language. |                                           |
|                     | * Even more readable than an internal DSL. |                                           |

Table 5.1: Summary of pros and cons of model instantiation methods

The direct use of API constructors is enough when the construction is not complex, e.g., the coordinate system constructor. However, in some cases the resulting code is difficult to read, write and maintain, as it was the case of the transformation functions. In these cases, alternatives like internal or external DSLs can improve usability and extensibility. This brings complications associated to new elements like expression builders or parsers, which need to be implemented and maintained. Furthermore, as new features have to be included, more parts are touched.
In conclusion, when the benefits of using an internal or external DSL (usability and extensibility) surpass the costs associated to its implementation (maintainability), it is a good way to improve productivity and communication between programmers and domain experts.
Chapter 6

Code generation

6.1 Introduction

At this point, we have discussed how a user can give model-specific information to make a model machine-processable. We have two options at this point. On the one hand, model-independent code runs in memory and reads model data provided by a user through a DSL. The behavior is dependent on the model data, and when given different model data, the model-independent code changes its state, representing a different transformation model. The different ways to provide such model data were discussed in Chapter 5. When all this happens at runtime, this approach is called interpretation.

On the other hand, model-specific code can be generated from the model data, that when executed, will give the same result as the interpreted approach, but possibly faster. This code can then be compiled in the target platform or language, and executed. This approach is called code generation.

In this chapter, code generation and its trade-offs, with respect to interpretation, are analyzed and discussed. Different aspects need to be considered when deciding whether code generation is a good choice or not. Some of them include: which parts of a model are static or dynamic (this aspect was briefly discussed in Sections 2.2.1 and 4.5), how a model interacts with clients, what is the impact on efficiency, among others. The advantages and disadvantages of interpretation are discussed in Section 6.2 and the same for code generation is discussed in Section 6.3.

6.2 Interpretation

In Chapter 5 it was discussed in what ways a transformation model can be created from a description given by a user. Different approaches were analyzed, but all of them use a representation of the metamodel that is stored in memory, and responds to queries and commands from a client directly (interpreter).

There are cases when an interpreter is not enough and model-specific code has to be generated. One case when it is necessary to generate code is when the operational software has to be executed in an environment where it is expensive to run an interpreter in memory [1], e.g. embedded systems with low resources. There are other benefits derived from code generation like compile-time type checking and faster execution.
Having an interpreter saves the effort of building a code generator. However, in ASML the communication between software components is done through interfaces. With this in mind, using an interpreter would involve making adjustments on the interfaces in order for a model to be able to communicate with other components. Whereas, using code generation would allow for generating, automatically from a model, ASML-compliant code which satisfies the interfacing rules.

An interpreter running in memory has the following characteristics:

- It allows for more flexibility, compared to code generation, as changes on a transformation model can be done at runtime, without the need to re-compile code.

- Specific model constraints or properties have to be checked at runtime. For instance, checking that the coordinates of a point are associated with the same coordinate system as the one a transformation function expects.

- Interpretation can affect efficiency, in opposition to code generation. For instance, when a data structure needs to be traversed repeatedly in order to perform a calculation. This can be solved by using techniques like caching or lazy evaluation, but this can compromise simplicity and introduce bugs.

### 6.3 Code generation

As opposed to the case when an interpreter runs in memory, it is also possible to translate model-specific data into code. The implementation and maintainance of a code generator can be expensive, but there are different options to generate code [1]:

- **Model-aware generation**: Reproduce the model-independent code of the metamodel (generic section) along with configuration code that represents the code that a user would write in order to create a model (configuration section).

- **Model-ignorant generation**: Navigate the structure of a model-independent object in memory, that has read model data, and generate code that represents such model. The resulting code does not necessarily resemble the original metamodel code. This approach usually involves “unfolding” the data structures, i.e. generating a hard-coded version of the objects, but using simpler language constructs.

Model-aware generation is often easier to perform, in the sense that only the configuration code is new. However, the target platform has to be able to run the model-independent code in memory in order to execute the configuration code. This also implies that interpretation features can be mixed with the generated code. When the generic and the configuration sections are well separated, it is possible to make changes in the model at runtime without the need to re-generate code. This can be accomplished by interpreting the configuration code, e.g. from a plain text file.

When using an external DSL it is possible to read model data from external storage (e.g. XML file, JSON). This avoids the need of regenerating code when a model changes, since it can be read again from the external storage at runtime.

Since model-aware generation reproduces the generic code that implements the metamodel, the generated code is not necessarily more efficient than the interpreted approach.
Model-ignorant generation is often more difficult to perform since the model data structures need to be navigated and their content exposed in the code. Furthermore, in order to exploit the potential benefits of code generation, a “clever” code generator that makes more efficient code by taking advantage of specific model properties and structure, can be challenging to construct. Usually, the resulting code is simpler to read, as it only uses basic constructs like control flow or function calls, making the structure of the objects explicit. A downside of this approach is that, as changes in the model are performed, the entire code needs to be re-generated. However, in our case, a transformation model does not change too often.

Code generation can be a costly process, and it is desirable not to do it too often. However, more efficient software can be generated using a model-ignorant approach, as no explicit representation of the metamodel is needed. This is useful when the target platform has limited resources. A particular example is when a point needs to be transformed form a coordinate system to another. Having an interpreter, an instance of the transformation graph is running in memory. A breadth-first search finds a shortest path from the origin coordinate system to the target coordinate system. This process is computationally expensive if done repeatedly. On the other hand, having generated the code for that particular instance of the transformation graph, a preprocessing step can hard-code a shortest path from the source coordinate system to the target coordinate system, saving the time to search the path in the graph.

Other aspects can be relevant when deciding what kind of generation to use:

- In the context of this project, it is important to produce code that complies with the ASML standards. This can be a challenging task as the code needs to work correctly with other components.
- It is desirable to have compile-time type checking rather than runtime checking. This can be achieved by generating code for new types of the instances of the abstract data types in the metamodel. In this way, type checking is done for free by the compiler. However, this benefit comes with the cost of re-generating code every time a change in the model needs to be done.

In particular, code generation can bring the following benefits to the problem of setpoint calculation:

- Coordinate systems can be translated from objects identified by a string, to new types. For example, a coordinate system named "RCS" can be translated to a class named RCSCoordinateSystem. The advantage of this translation is that checking the identity of a coordinate system is shifted from runtime, where a comparison with a string has to be done, to compile time, where the compiler just checks the object type.
- All the objects that use a coordinate system can also benefit from its static typing. For example, the abstract data type Point has an associated coordinate system. In the implementation, several algorithms need to assert that a point has the right coordinate system associated. Otherwise, it must throw an exception. By generating code that assigns a type like Point<RCSCoordinateSystem> p, checking that p is associated to the "RCS" coordinate system is done at compile time. This helps to ensure structural correctness of the model.
• One way to generate code for a transformation model is by encoding the graph structure in method calls. For example, the method \( r = \text{withRespectTo}(\text{Point} \ p, \ \text{CoordinateSystem} \ rzcs, \ ...) \), transforms point \( p \) from the coordinate system to which it is associated, say RCS, to coordinate system RZCS. This involves traversing the graph from RCS to RZCS through RSCS (see Chapter 3 for more details). This code can be “unfolded” like this:

  - \( \text{Point}<\text{RSCSCoordinateSystem}> \ p1 = \text{rcsWithRespectTo}(\text{Point}<\text{RCSCoordinateSystem}> p, \ \text{RSCSCoordinateSystem} \ rscs) \).
  - \( \text{Point}<\text{RZCSCoordinateSystem}> \ p2 = \text{rcsWithRespectTo}(\text{Point}<\text{RSCSCoordinateSystem}> p1, \ \text{RZCSCoordinateSystem} \ rzcs) \).

  The method \( \text{rcsWithRespectTo} \) calls internally the method \( \text{rcsWithRespectTo} \). An external control method needs to be generated to trigger the first call and decide when to stop the function calls. In addition, extra checking needs to be done to not end up in an infinite loop due to a cycle in the graph. However, the result is type-safe code which is easier to follow.

• The code for the transformation functions can be generated in a similar way as the transformation graph, by using a sequence of function calls representing the traversal over the tree.

• There can be different ways to calculate setpoints. One way is by solving multiple linear equations, using one or more points on each side of the transformation function. Usually, an external library is used to solve such equations, but re-writing the transformation function into the shape that the library expects can be expensive at runtime. Thus, the generated code can have the transformation function already written in a canonical form that the library can understand.

### 6.4 Conclusion

In this chapter, the differences and trade-offs between interpretation and code generation were discussed. In Table 6.1 a summary of the pros and cons between interpretation and code generation is shown.

From the trade-offs and the specific conditions of the ASML software, the model-ignorant code generation is a suitable option to be implemented as it can provide more efficient code, and to some extent, compile-time type checking. The structure of a transformation model, which is implemented as a graph, does not change often. Therefore, the cost of regenerating code due to changes is compensated by the frequency it is realized. Given that efficiency and static type checking at compile time are relevant, the advantages of model-ignorant generation add value to the approach. It remains to be determined how the cost of implementing and maintaining a code generator can be balanced with the benefits it brings.
<table>
<thead>
<tr>
<th>Method</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpretation</td>
<td>* Provides flexibility to modify a model at runtime.</td>
<td>* Allows instance data checking only during runtime, when compile time is more desirable.</td>
</tr>
<tr>
<td></td>
<td>* Makes easy model instantiation as specific configuration data can be fed directly to a metamodel object.</td>
<td>* The metamodel has to be implemented such that it complies with the ASML standards.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* It can be inefficient when several recomputations are done.</td>
</tr>
<tr>
<td>Model-aware code generation</td>
<td>* It involves reproducing the metamodel code along with specific configuration code. Thus, the code generation is not so costly.</td>
<td>* It is not more efficient than the interpreter as the metamodel has to be replicated.</td>
</tr>
<tr>
<td></td>
<td>* It can be mixed with interpretation to modify a model at runtime.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>* It can provide ASML compliant code.</td>
<td></td>
</tr>
<tr>
<td>Model-ignorant code generation</td>
<td>* Model-ignorant generation can provide instance data type checking at compile time.</td>
<td>* The cost of implementing and maintaining a model-ignorant code generator can surpass the benefits of having it.</td>
</tr>
<tr>
<td></td>
<td>* It can yield readable code that can be easily debugged.</td>
<td>* Model modifications imply code re-generation.</td>
</tr>
<tr>
<td></td>
<td>* It can provide a more efficient model representation than its interpreter counterpart.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>* It can provide ASML compliant code.</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Summary of pros and cons of interpretation vs. code generation
Chapter 7

Conclusions and future work

In this chapter, we discuss the results from the previous chapters and show the feasibility of using technologies from the field of domain-specific languages in the AMSL metrology software.

7.1 Critical evaluation

Currently, the ASML metrology software is complex, and it has become more difficult to maintain and extend. The idea of using a DSL was born from the necessity to reduce such complexity, by expressing the functional requirements in terms of domain concepts on a higher level of abstraction. This level of abstraction brings benefits, such as: better communication between programmers and domain experts; and higher productivity. The former results from the fact that specifying functionality, properties and constraints is easier in terms of domain concepts. The latter is a consequence of the fact that reading a piece of code in terms of high-level constructs is easier.

Creating a DSL can be a challenging task and it is crucial to choose the right technologies in order to implement it. Specifically, in ASML, the transition from the traditional way of developing software to a more domain-specific way, can be more difficult to overcome, if the chosen techniques are not in line with current engineers’ background and skills. This is because the easier it is to grasp a new technology, the more successful it will be.

Regardless of the type of DSL technology, it is crucial to design a suitable representation of the domain knowledge. This implies defining the right abstract data types that represent the domain concepts, as well as the operations on them. The de facto way of representing such domain knowledge is as a metamodel.

In a metamodel the domain concepts are represented as entities with attributes, operations and relationships with other entities. There are several ways in which a metamodel can be implemented. For example, as an Ecore model in EMF, or as a set of interfaces and classes (API).

Using technologies such as EMF requires specialized knowledge in order to express transformations from a graphical metamodel to code. Features such as code generation or an editor to create models using specific model data come at low cost, requiring specialized effort to get them working.

Implementing a metamodel as an API brings more flexibility as there is more freedom in choices related to design or use of features of a specific language. However, there is
an added cost because DSL parsing and code generation have to be maintained. In this project, the domain was restricted to a subset of the metrology functions, which made easy implementing an API from scratch in short time.

The different ways to make a transformation model machine processable have both advantages and disadvantages. Using API constructors directly is straightforward, however, as was the case for transformation function trees, it can result in complex code. Internal DSL and ad-hoc external DSL have about the same costs and benefits. An expression builder or parser can be difficult to maintain. But the resulting code is more readable, if implemented properly.

It is advantageous to have model ignorant code generation as it can provide more efficient code and compile-time type checking. However, the code generator can be challenging to implement and maintain.

7.2 Objectives revisited

From the results obtained, we will try to analyze the feasibility of the proof-of-concept from the perspective of the questions posed originally:

1. How can the metrology knowledge be structured using a domain-specific model? The answer to this question is by defining the domain concepts as abstract data types, and creating interfaces where the functionality is specified in a high-level way. The classes that implement the interfaces have to be decoupled from the high-level specification (separation of concerns).

2. How usable, maintainable and scalable are the different alternative domain-specific models? The DSL alternatives make the software more readable, and therefore, improves productivity and communication between domain experts and programmers. But there are costs in maintaining parsers or expression builders. Changes in the functionality of the software becomes easier as they are done in a higher level of abstraction. Scalability is possible if there is a good separation between high-level abstractions and a specific implementation.

3. How does the abstraction of a domain-specific model impact the efficiency of the metrology processes in terms of development time? A careful design of the metamodel is crucial. The initial design can be the most costly process. Parsers, expression builders and code generators can be also costly. However, once they are implemented and tested, the time to add new functionality, improvements and changes is reduced abruptly.

7.3 Recommendations for ASML

Given the analysis made in this document, and the different alternatives that each component of the DSL architecture, it is important to give a clear recommendation on the path to follow, on the viewpoint of the potential users and the resources of ASML.
7.3.1 Metamodel

The recommendation here is simple. Always abstract the domain by defining abstract data types that represent domain concepts and develop a metamodel that can then be used to implement different DSLs.

7.3.2 DSLs to make models machine processable

In Table 7.1, the following criteria are evaluated against each approach: readability, model maintainability, code maintainability and integration with host language. The first two are related to the modeler user, and the last two are related to the maintainer user.

<table>
<thead>
<tr>
<th>Modeler</th>
<th>Maintainer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readable</td>
<td>Models easy to maintain</td>
</tr>
<tr>
<td>Using API constructors</td>
<td>✖️</td>
</tr>
<tr>
<td>Internal DSL</td>
<td>✅</td>
</tr>
<tr>
<td>Ad-hoc external DSL</td>
<td>✅</td>
</tr>
</tbody>
</table>

Table 7.1: Trade-offs of DSL approaches

Since the goal is to improve productivity and communication by using a DSL, the readability and model maintainability are the most relevant factors. Both, internal and external DSL help to achieve this.

For a code maintainer, it is more difficult to maintain a completely new grammar and parser, as they are decoupled from the host language, as in the case of external DSLs. Internal DSLs, however, are integrated with the host language. Therefore, they are recommended as they balance the benefits for a modeler user with the costs for a maintainer user.

7.3.3 Interpretation vs. Code generation

In Table 7.2, the following criteria are evaluated against each approach of code generation: efficiency, maintainability (maintainer user) and whether it provides compile-time checking.

<table>
<thead>
<tr>
<th></th>
<th>Efficiency from model structure</th>
<th>Easy to maintain</th>
<th>Compile-time checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpretation</td>
<td>✖️</td>
<td>✅</td>
<td>✖️</td>
</tr>
<tr>
<td>Model-aware code generation</td>
<td>✖️</td>
<td>✖️</td>
<td>✖️</td>
</tr>
<tr>
<td>Model-ignorant code generation</td>
<td>✅</td>
<td>✖️</td>
<td>✅</td>
</tr>
</tbody>
</table>

Table 7.2: Trade-offs of Interpretation vs. Code generation
Both interpretation and model-aware code generation reuse the metamodel code in order to instantiate models. Therefore, no advantage is taken from the specific structure of a particular model. Model-ignorant code generation, however, allows this.

Building a code generator can be more or less costly, according to how the code is generated. A model-aware code generator replicates metamodel code, and adds configuration code. On the other hand, a model-ignorant code generator has the potential to provide a smart way to generate code in such a way that the specific model structure is exploited. This provides more efficiency.

A good bonus point of model-ignorant code generation is that it can provide compile-time checking, by translating model data to new types.

The recommendation is to use model-ignorant code generation. The drawback is the cost associated to maintaining a good code generator. But the benefits compensate for this cost.

7.3.4 Staff and resources needed

In order to carry out the recommendations given in this section, it is necessary to have software engineers who can make abstractions from the real world, and can translate that to a metamodel and well-designed APIs or libraries.

It was shown in this work that building an internal DSL for a restricted domain requires tweaking the host language (C++) in a way that it provides a fluent interface. The software engineers in charge of maintaining this, only require good knowledge of the language and a good understanding of the needs of the domain experts.

Other specific technologies like EMF require engineers who have experience building model-driven software using those technologies.

7.4 Future work

The scope of this project was to model in a domain-specific way transformation functions and setpoint calculation. However, this is just the first stepping stone towards a complete migration of the ASML software from the current state, which relies almost completely on GPPLs, to a state where DSLs improve the software quality.

In this section, a list of possible next steps are shown, addressing different aspects: related to DSL, related to metrology functions and related to numerical computations.

7.4.1 Related to DSL

EMF transformations In this document it was mentioned briefly how EMF is a useful tool for DSL modeling. It will be useful to compare the pros and cons of the approach presented here with EMF transformations.

Other alternatives to populate a model Explore other ways to define a metamodel (define types), such as: XML Schema Definition (XSD) or JSON Schema. The analogous

http://www.w3schools.com/schema/
http://json-schema.org/
alternatives for providing model data are XML and JSON. Yet other options are INI files, YAML or ATerm.  

**Generated code that complies to ASML standards** It will be interesting to build a code generator that takes advantage of the numerical properties of transformation functions and setpoint calculation.

### 7.4.2 Related to metrology functions

**Debugging** A DSL allows a programmer or stakeholder to easily understand a model behavior. However, there can be implementation bugs. Adding a way to debug a transformation sequence or setpoint calculation (logging, memory analysis, etc) will be useful for the software development process at ASML (see requirement [REQ5](#) in Section [4.2](#)).

**Effect of parameters on measurements** An important function of the metrology department is to describe how certain parameters explain the behavior of the machine. It is critical that a DSL that will be integrated with the ASML software does this. As parameters get updated based on measurements, a transformation state also changes (see requirement [REQ6](#) in Section [4.2](#)).

**Setpoint trajectory** In exposures, a sequence of setpoints need to be calculated to describe a trajectory that a reticle or a wafer have to follow. This can be described using a DSL in a similar way as single setpoints. This is a critical part that has to added to a DSL that would be integrated with the ASML software (see requirement [REQ7](#) in Section [4.2](#)).

### 7.4.3 Related to numerical computations

**Rotations using quaternions** In the metrology processes an important part of the mathematical modeling are the rotations. Currently, Euler angles are used to describe the rotations in 2D or 3D. A better approach might be to use quaternions to represent such rotations, since they are more numerically stable and may be more efficient.

**Minimize numerical errors** The transformation functions shown in this document had a specific shape, but it was not discussed what numerical problems such shape can bring when performing numerical computations. An interesting topic will be to investigate further how to define transformation functions and setpoint calculation such that the numerical errors are minimized.
Bibliography


