Informatics

Interoperability of Metamodeling Frameworks

Bridging Modeling SDK for Visual Studio and Eclipse Modeling Framework

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Florian Cesal, B.Sc.
Registration Number 1058077

to the Faculty of Informatics

at the TU Wien

Advisor: Ass. Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork

Vienna, 19th January, 2023

Florian Cesal   Dominik Bork

Erklärung zur Verfassung der Arbeit

Florian Cesal, B.Sc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. Jänner 2023

Florian Cesal

Acknowledgements

Thanks are especially due to my supervisor Ass. Prof. Dipl.-Wirtsch.Inf.Univ Dr.rer.pol. Dominik Bork, who let me chose after our first meeting from several topics suitable to my experience. Through the course of numerous informative and constructive meetings, he always steered me in the right direction and informed me well regarding possible implementation approaches, thus any errors or misunderstanding could be reacted to immediately. In addition, I want to express my gratitude for receiving the opportunity to participate in a scientific conference based on the thesis' area of relevance.

I would also like to thank my mother, who fully supported me, my university and professional career right from the start, from the entrance into computer science already in the lower grades, to the preparation for my start into the bachelor's degree as well as for the financial and logistical means to enable me to study where I wanted.

To my girlfriend, who has been helping me finish my master's degree at the TU Vienna in the recent years, in good times and in bad times always lent me an ear to listen to my concerns and problems and never let me stray from the goal to finish my studies.

To my work, in which I have been employed full-time in Vienna for 5 years, which supported me with flexible working hours when the bureaucratic needs as well as exam dates including learning efforts had to be absolved.

Last but not least, I would like to thank my friends, some of whom already finished their studies earlier than I did and thus indirectly always put a healthy pressure on myself to complete mine. A lot of these friendships could be gained in the past few years, but those who have known me the longest (elementary school or high school), most of the thanks have to be attributed to, because they filled my time of study with joy, laughter and the best times of my life.

Kurzfassung

Mächtige Plattformen für modellgetriebene Softwareentwicklung existieren, jede mit ihren eigenen Stärken, Schwächen, Funktionen, Programmiersprachen und Entwicklergruppen. Um ihre individuellen Vorteile zu vereinen, wäre es von Vorteil, Interoperabilität zwischen den Platformen herzustellen, sprich Metamodelle und Modelle die in einem Framework erstellt wurden zu einem äquivalenten (Meta)modell in einem anderen Framework zu transformieren. Dadurch ist Freiheit gegeben, beliebige Metamodellplattformen zu wählen ohne dabei einen Lock-In-Effekt zu riskieren.

Zwei der wohl bekanntesten Plattformen bezüglich Metamodellierung, die darüber hinaus auch gut dokumentiert und frei verfügbar sind, sind das Eclipse Modeling Framework (EMF) und das Modeling SDK for Visual Studio (MSDKVS). Diese Arbeit gibt genauere Erklärungen wie man zwischen EMF und MSDKVS Interoperabilität erzielen kann, und das auf zwei Ebenen, nämlich der Ebene der abstrakten Syntax und der Ebene der grafischen konkreten Syntax.

Die Plattformen werden verglichen, ihre Eigenschaften dementsprechend aufeinander abgebildet und eine bidirektionale Transformationsbrücke auf ihren Metamodell- und Modellebenen implementiert. Der Ansatz wird ausführlich evaluiert mittels Transformationen von verfügbaren Metamodellen und davon generierten oder manuell erstellten Modellen. Die Evaluation bezüglich der Validität, Ausführbarkeit und der Aussagekräftigkeit der Transformationsbrücke selbst wird quantitativ und qualitativ durchgeführt.

Abstract

Powerful metamodeling frameworks for realizing Model-Driven Software Engineering (MDSE) exist, each having its own strengths, weaknesses, functionalities, programming languages and engineering communities. By combining their individual benefits, it would be preferable to establish interoperability between them, i.e. transforming metamodels and models created in one framework into equivalent (meta)models in other frameworks. Thus enabling the freedom of choosing the metamodeling platform without risking a lock-in effect.

Two of the most well known metamodeling frameworks, which are well documented and available for free are the Eclipse Modeling Framework (EMF) and the Modeling SDK for Visual Studio (MSDKVS). This thesis gives detailed explanations on how to achieve interoperability between EMF and MSDKVS on two levels, namely the abstract and graphical concrete syntax levels.

The platforms are compared, their features are mapped accordingly and a bidirectional transformation bridge on their metamodel and model layers is implemented. The approach will be extensively evaluated by transforming available metamodels and generated or manually created models thereof. The evaluation regarding the validity, executability and the expressiveness of the transformation bridge is done quantitatively and qualitatively.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

Model-Driven Software Engineering (MDSE) is made possible through the use of different metamodeling frameworks and tools which are used to define Domain-specific languages (DSLs) and the application thereof. The benefits of such languages can help designers efficiently cooperate on creating an underlying software infrastructure. The most prominent and actively used one is the Eclipse Modeling Framework (EMF). Many more such frameworks exist, some of them are available through proprietary licenses and fees and others are available as freeware. In order to be able to fully use the functionality of the metamodeling environments and the user interfaces that metamodeling frameworks provide, a need to transform metamodels created in one framework to syntactically equivalent metamodels in other frameworks has emerged.

Not only do frameworks differ in their representations, their platforms also differ in their functionality and the plugins they provide in addition to their use of different programming languages and their capabilities. The option to freely choose which programming language should be used to extend a metamodel in its functionality (validation, code generation, etc.) and the toolset different metamodel frameworks provide is also contributing to the need for nearly lossless transformations from one metamodeling environment to another. Traditionally, once developers started working with one platform, switching to different platforms later on was cumbersome and time consuming, not only because of the complexity, but also because of the scarcity of available automated support for metamodel interoperability. One well documented and freely available example as an additional metamodeling framework to the prominent EMF is the Modeling SDK for Visual Studio (MSDKVS), which has been developed and is still being maintained by Microsoft and is part of the DSL Tools Solution for Visual Studio editions. By providing interoperability among various frameworks, teams and individuals using different tools and environments can use this to strengthen collaborative development.

1.2 Aim of the work

The aim of this work is to provide a transformation framework which offers the possibility to transform metamodels and their models defined in the Modeling SDK for Visual Studio (MSDKVS) to metamodels corresponding to the syntactical elements (concrete graphical and abstract syntax) available in EMF and vice-versa, thereby achieving interoperability among these environments and their toolsets. Interoperability and its challenges, especially Model-driven interoperability (MDI), have been defined in [10]. MDI aims at providing bridges to achieve interoperability between several systems by mapping syntactic and semantic information from one system derived from its specification, representation in a specific format (e.g., XML) or a public API to another. In [28] and [29], interoperability is also mentioned within the context of being able to migrate existing systems to others. Regarding the metamodeling environment, this means the migration of models and metamodels from one tool to another while preserving structural and semantic information. Through an extensible rule framework, on which the transformation will be based, different metamodeling platform concepts can be investigated and compared in detail, such that the metamodeling framework with a stronger or weaker set of features than the other can be identified for each element.

In addition to the transformation on the metamodeling level, the result of this transformation then acts as a reference metamodel for the transformation on the layer below, the models themselves. Through the use of the result of the first transformation it will be possible to migrate models defined in one metamodeling environment to the other. One of the challenges of the transformation approach is defining the rule set, through which the source metamodel is transformed to a corresponding metamodel in the target framework. This set of rules maps source elements to target elements, where some elements are more complex in their functionality and abilities in one framework than the other, e.g., multiple versus single inheritance or geometry definitions for different shapes of entities on the model layer. Another challenge is the unavailability of a standardized definition for the meta-metamodel that metamodels defined in the MSDKVS depend upon which results in less insight on the structure of the framework itself, in contrast to EMF with its meta-metamodel definition for Ecore.

Benefits of this transformation are manifold, for example by providing interoperability between the two frameworks the current state of the art interoperability chain is extended, meaning the possibility of modeling engineers and other shareholders of a project, where metamodeling is integrated in the tool chain, are not bound to just one metamodeling framework. Metamodels and models thereof can be moved independently among this transformation chain between various metamodeling frameworks by chaining the different transformation bridges together. Through implementing the ruleset for the transformation, an implicit class diagram of elements for representing the functionalities of metamodels in MDKVS is created that can be used as a reference meta-metamodel for metamodels in this framework, tackling one of the challenges described above. Metamodel zoos, collections of metamodels that are clustered and can be searched among [4, 5, 32], that contain numerous metamodels on both sides, should act as an input for proving the validity and feasibility of such a transformation and should highlight the benefits a transformation approach provides. Through extensively analyzing both EMF and MSDKVS platforms, the proposed transformation bridge will be implemented and evaluated accordingly, considerably advancing previous attempts [7, 13] further discussed in detail in Section 4.3.

1.3 Methodological approach

Research of this work follows the design science research approach consisting of the following steps, starting from initial state of the art research, defining the problem relevance in today's environment, the reviewing of existing approaches (e.g., transformation bridges) and adapting them according to the frameworks in question and achieving a resulting metamodel on the target side to migrate models from the source modeling language to the target modeling language. Furthermore, the evaluation through defined metrics and the freely available resulting artifact for further usage contribute to the research area. Considering the design science research approach, following research questions are investigated:

- **RQ1: What is the degree of interoperability between MSDKVS and EMF in terms of user interaction and framework behavior?** When transforming (meta)models from one tool to another, differences in complexity, usability, tool support, graphical interface etc. are unavoidable. This research question deals with the qualitative analysis on how much of the original features can still be used, ideally semantically equivalent, after the transformation into the target environment has been executed, to what degree they differ and how their interaction and behavior has changed. The qualitative analysis is done through examination of both framework environments regarding their interaction possibilities with the transformed entities in comparison to their representations and functionality in the source framework.

- **RQ2: What are the quantitative differences and similarities of transformed metamodels and models in the target framework compared to the original?** Quantitative, comparative analysis of the M3 bridge of a transformation on both metamodel and model level has to be done in order to show similarities and differences in how a metamodel and its models are syntactically represented by, e.g., number of classes, abstract classes, relationships, inheritance relationships, shapes, etc. Based on the ruleset definitions, assumptions and formulas regarding these numbers can be made beforehand and are evaluated against the transformation results.

With regards to the design science approach principles, the following steps are performed in order to achieve the desired results and are described in more detail in the following chapters.

1.3.1 Review of frameworks

In-depth analysis of metamodeling frameworks has to be done in order to identify possible transformation rules based on their syntaxes, similarities and differences among semantic and syntactic representations and model generation functionalities as well as validity constraints.

1.3.2 Review of existing transformation approaches between different metamodeling frameworks

In [9, 26, 27, 29, 31] identical approaches are used for transforming modeling languages realized in a variety of different metamodeling tools (Microsoft Visio, Aris, MetaEdit+, ADOxx). Further work is available in identifying common structures among different metamodels and also different metrics in terms of usage statistics of abstract syntax elements (element classes, element relationships) and their complexity in [39]. A transformation approach for an older version of the DSL Tools for Visual Studio, which nowadays contains the Modeling SDK for Visual Studio for defining domain specific languages, to EMF has been conducted in [7, 13] but does not use all syntax elements and functionalities that both frameworks offer, e.g., concrete graphical syntax elements realized with Sirius in EMF, and the amount of abstract syntax transformation rules are kept at a minimum as well as the resulting evaluation of the feasibility and validity of the approach are missing.

1.3.3 Defining a ruleset for transforming metamodels between MSDKVS and EMF

Based on the previous steps, [29] defines a basic ruleset for transformations among the most common features of current metamodeling frameworks which is then adapted to each combination of transformations that is implemented. What is missing are the concrete syntax elements, [29] focuses on the abstract syntax of metamodeling tools. In addition to the adapted ruleset for the transformation approach between the abstract syntaxes of MSDKVS and EMF, their graphical syntax capabilities are also added to the ruleset. Common elements that exist in both representation formats are identified and compared against each other. These rulesets (abstract and concrete syntax rulesets) will be adapted and extended incrementally in subsequent chapters, when exclusive functionalities, that have to be mapped explicitly, are investigated in detail.

1.3.4 Applying the ruleset defined in step 3 to a transformation bridge based on XML representations

The transformation approach on the metamodel layer is realized as a M3 XML transformation bridge that executes the set of rules defined in step 3 sequentially to transform metamodels (their XML representations) defined in MSDKVS to EMF (XML representation files as well) and vice-versa.

1.3.5 Extensive evaluation of the transformation approach through metamodeling zoos

Through applications focusing on searches among metamodels and source code repositories, like Github and Google Code, it is possible to filter available metamodels created by the community and use them as input for the artifact to analyze and prove the validity of the resulting metamodeling syntax. A web application to crawl through previously aggregated search results is available in [34]. The transformation approach is then evaluated against certain criteria, such as completeness, complexity, and reusability.

1.3.6 Transforming of models based on resulting metamodel on target framework

The transformator creates a metamodel corresponding to the features of the target metamodeling tool. Models from the source tool are then fed into the second part of the transformation, namely the transformation of models. The resulting model is validated against the previously created metamodel and evaluated on the target framework. Again, metrics such as completeness, complexity, not only concerning the transformation approach itself but also lines of code or runtime, are investigated as well. Additional qualitative aspects such as semantic equivalence regarding the use of the different frameworks based on source and target model are analyzed and compared (see research question 1 and 2).

1.3.7 Research contributions

The contribution results of this thesis' research are manifold. First, mapping tables, that not only target abstract syntax features of both frameworks, but a novel approach in also including the graphical concrete syntax capabilities, are established. The MSDKVS' internal meta-metamodel structure is exposed in the form of a class diagram. Last but not least, the transformator, source and transformed models and metamodels are available in a publicly hosted github repository. On top of that, the transformator itself will be available as a website widget [1] for transforming (meta)model files.

1.4 Structure of the thesis

The thesis is structured as follows. Chapter 2 provides necessary definitions of the foundations for metamodeling. Chapter 3 investigates the two platforms that are used as target and source frameworks for the transformation approach, at first separately with example models and then also giving an overview on similarities and differences that should be taken into account when defining the rules the transformation is based on. The next Chapter 4 gives an overview on the domain of model-driven interoperability and existing transformation approaches based on other frameworks and related works,

[1] `https://me.big.tuwien.ac.at/`, last accessed on 04.01.2023

including an approach targeting the DSL Tools Framework for Visual Studio, which will be explicitly compared in detail to this thesis' approach. Chapter 5 defines the rulesets, on which the transformations from MSDKVS to EMF and vice-versa are built upon, where not only abstract syntax elements but also graphical concrete syntax elements used for visually representing said elements are considered.

Based on these rules, Chapter 6 explains the implementation for the M3 transformation bridge in detail. The results of the implementation are then used in Chapter 7 for a study on the feasibility of the transformation approach, taking into account a number of metamodels contained in so-called metamodeling zoos that can be found on different websites and inside publicly available code repositories. After having successfully transformed metamodels between the frameworks, models based on these metamodels should also be transformed. How modeling is realized in both frameworks will be described in depth in Chapter 8. Afterwards, Chapter 9 evaluates the full M3 bridge approach by comparing the outputs of different runs of the transformation bridge. The research questions posed above are answered accordingly.

Chapter 10 ends with a conclusion on the feasibility and completeness of the transformation and also a discussion about the limitations and possible losses in functionality that the transformation unavoidably creates.

CHAPTER 2

Foundations of Metamodeling

This chapter lays the foundation of the domain in which this thesis' topic is situated, i.e., the domain of Model-Driven Software Engineering (MDSE). The origins, ideas and descriptions of the most prominent features are given in order to understand the subsequent chapters and to get a glimpse of the main problems and obstacles the transformation approach is about to solve and overcome. A very detailed overview and further information on this topic is given in [10].

2.1 Model-Driven Software Engineering (MDSE)

Complete or partial representations of real world objects, architectures, interfaces as well as software systems can be abstracted by the use of models [15]. These models can then be shared and used as a communication base among shareholders and different teams that are collaboratively working on projects using a modeling approach. Concerning the validation and guidelines for defining said models, a hierarchy for further abstracting them exists, divided into a stack of layers. Figure 2.1 shows an example of such a hierarchy stack, consisting of four layers which have been standardized by the OMG [10, 33].

- **M0 Layer (runtime instances)** containing the application data or runtime instances.
- **M1 Layer (model layer)** describing the concrete user model based on the given metamodel (e.g., a UML model).
- **M2 Layer (metamodel layer)** defining the metamodel (or DSL).
- **M3 Layer (meta-metamodel layer)** abstracting the definition for possible metamodel structures. This level provides the functionality for interchanging of

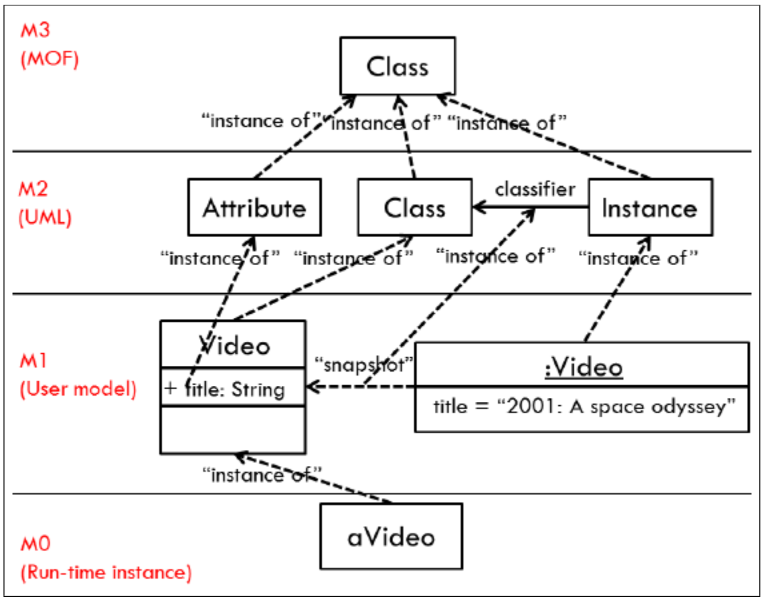


Figure 2.1: The four-layer metamodeling hierarchy [16]

tools and their metamodels. As an example, in this figure, the M3 layer consists of the Meta-Object Facility (MOF) standard for defining underlying metamodels.

Because the M3 layer can indefinitely be referenced by itself the hierarchy stops at this level and the overall system therefore is strictly enclosed. The M3 level also established the foundation for realizing interoperability of tools based on a common abstraction of their metamodels.

When speaking about modeling in the domain of developing and maintaining software, the term Model-Driven Software Engineering (MDSE) is used, where not only software developers, but also other types of stakeholders of the software artifact can be included to communicate with each other to reach the desired outcome. In such cases, the abstraction of the underlying software infrastructure in the form of one or more models can help eliminate communication barriers, foster cooperation and even generate code [10].

One outcome of a MDSE approach can be a modeling language. Several so-called meta-modeling languages or tools have emerged to simplify this task by providing developers with the necessary equipment. Most of those metamodeling languages share the same principles, which will be mentioned in Section 2.2 later in this chapter. Some prominent examples of metamodeling platforms that also provide IDEs to efficiently create the needed syntax elements include the Eclipse Modeling Framework, MetaEdit+ and the DSL Toolkit for Visual Studio. When designing metamodels in these frameworks that best suit the requirements of the defined context, each metamodel has to correspond to an underlying meta-metamodel integrated into these platforms. These metamodels are usually publicly available. The Meta Object Facility Standard (MOF) and the metamodeling language for EMF, called Ecore, both contain parts of UML class diagrams and are situated on the M3 layer of OMG's metamodeling stack.

Regarding the development of modeling languages, two different types can be distinguished based on their expected functionalities.

- **General purpose languages (GPLs)** should be considered when the domain cannot be clearly isolated and all of the available functionality of a modeling framework has to be used in order to target a broader range of problem areas and application contexts. Typical examples of GPLs can be UML (Unified Modeling Language) or programming languages like Java, Python, Go etc.
- **Domain specific languages (DSLs)** are languages specifically targeting the context of a domain that is best suited for the project's needs. Some examples for DSLs are a medical application, a language situated in the banking sector, a database application, and with regards to developing specific kinds of software products also HTML, Matlab, SQL etc. Regarding DSLs in the domain of modeling languages, the term DSML (domain specific modeling languages) can also be used [10].

Elements in modeling languages can be divided into two main syntax groups: *abstract* and *concrete* syntax. Abstract syntax elements consist of relationships, classes and their attributes. They define the domain or language on a more theoretical level through, e.g., connections between these entities, their internal structures, etc. Concrete Syntax on the other hand refers to elements, either graphical or textual, describing the abstract syntax elements with a notation easier understandable by humans. Graphical concrete syntax (GCS) elements are distinguished by their different shapes, sizes and additional graphical objects, like icons or symbols. Textual concrete syntax (TCS) elements consist of a collection of characters that describe how the element is structured. In addition to these two groups, the semantics also play an important part of giving meaning to syntax elements and guiding the designer for correctly defining the modeling language depending on the domain and its intended usage.

2.2 Features of MDSE

The following section briefly describes the important aspects of MDSE and their typical use cases with regards to modeling languages and tools.

2.2.1 Structure of modeling languages

Most commonly known modeling languages are derived from class diagrams in UML, which describe the structure of a system using classes, their attributes, their methods and relationships between them. In a UML class diagram, each attribute and method is assigned a visibility status and methods can be given multiple parameters, which can be either *in* or *out* parameters, or both.

Relationships between classes in a class diagram are determined by different available types, like association, aggregation or composition. Inheritance concepts between classes are also special types of relationships, specifically *"is-a"* relationships, where the super class can either be fully specified or abstract and the derived classes inherit features (i.e., attributes and methods) from the referenced super classes.

Relationships can have different cardinality (or multiplicity) values assigned to them, like *"zero or more"*, *"exactly one"*, *"one or more"*. Aggregation and composition relationships represent a relationship, where one class is part of another class. The difference between these two relationships are with regards to the deletion of classes, where the elimination of the compositing class deletes all classes part of the composition.

Modeling languages (specifically metamodeling languages), which are adaptations the UML class diagram design pattern, consist of the following elements:

- **Abstract syntax**, where classes, associations and attributes are defined as abstractions extracted from the software system to be mapped or generated, as well as constraint languages like OCL used for defining constraints to act as rules for validating models. Abstract syntaxes can be either defined for GPLs or DSLs, if the target domain is clearly structured and contained.
- **Concrete syntax**, which adds additional representational aspects on abstract syntax elements for distinguishing them and making them more understandable and readable based on constraints.
- **Semantics**, including generating model editors to create models based on predefined metamodels, code generators, suitable editors for operations (e.g., CRUD) on metamodel entities for creating said metamodels and transformation approaches for different types of model transformations.

The metamodeling language in the Eclipse Modeling Framework for example is called Ecore, consists of the aforementioned components of a metamodeling language and offers APIs for manipulating, graphically editing and storing models.

2.2.2 Model Transformation

Model transformations [36] between models are done by defining a mapping between metamodels that is used to transform a model corresponding to its metamodel to another model corresponding to the same or another metamodel. This mapping executes a number of operations and applies rules in order to create the resulting model.

A model transformation can either be realized as a *Model-To-Model (M2M)* transformation or a *Model-To-Text (M2T)* transformation, where the former has to conform to constraints given by metamodels and the latter includes code generation possibilities, where a metamodel on the target is not needed.

Further distinctions of model transformations are mentioned in [19, 25, 29], for example *endogenous transformations*, where the source and target metamodels reference the same meta-metamodel, and *exogenous transformations*, where the source and target meta-metamodels are not the same. Another type of a model transformation is concerned with the direction of the transformation itself, where *unidirectional* transformations can only be done in one direction and *bidirectional* transformations are done in both directions.

Model transformations have also been standardized by defining so called model transformation languages [25] like the Atlas Transformation Language (ATL) or the Epsilon Transformation Language (ETL), where each language describes model transformation by rules using metamodel elements. These rules are conducted on the model layer, transforming a source model to a target model.

Not only can model transformations be achieved between different metamodels and their resulting models defined in the same platform or framework, but it should also be possible to transform models and respectively their metamodels between different platforms in the form of model transformation bridges. Various possibilities and an overview on the most common approaches of such model transformations are given in Chapter 4. An important aspect to consider is the chaining of model transformations where a model traverses a list of intermediate states to reach the desired output (e.g., the executing program).

2.2.3 Model validation and constraint languages

For defining more complex constraints and restrictions on possible designs and use cases that metamodels and their models allow, OMG has standardized a language called *Object Constraint Language (OCL)*. OCL enriches metamodels with additional validation functionality and rules they apply on their models. Basic types, iterators over lists and mathematical quantifiers (*"for all"*, *"exists"*) are usable in OCL statements. As a consequence, relevant objects based on entities and their attributes contained in a metamodel can be queried, filtered and returned as a result for dynamic validation responses while designing models.

2.2.4 Code Generation

Code generation in the form of automating tasks and creating executable programs is also a very important part in MDSE. In the long run, it facilitates the maintenance of domain specific tasks because the underlying models comply to a metamodel designed for the requirements of the domain, meaning models instead of directly writing source code are used for developing software artifacts.

As mentioned before, regarding model transformations, code generation is a typical example for M2T transformations, where the source is a model, that is an instance of a metamodel, and the target is an executable source code or any other more sophisticated executable or interpreted file. Trivial and more complex code generation transformations can be implemented by using a set of templating engines, where either a simple template for possible classes can be created or additional functionalities like basic CRUD operations can be inferred and also generated. Advantages and different code generation techniques are further given in [10].

CHAPTER 3

Metamodeling Frameworks

The following chapter lists and explains the core aspects of both frameworks that this thesis and its transformation approach deals with, namely the Modeling SDK for Visual Studio (MSDKVS) and the EMF, and compares them according to their available documentation, similarities and differences regarding their functionality and their popularity among software engineering communities. To further demonstrate these aspects, the frameworks are compared against each other by defining the same metamodel on both frameworks, specifically family tree metamodels are created, similar in their purpose, which consist of different constraints regarding relationships between their entities (e.g., father, mother, child), inheritance features and different concrete syntax representations.

A representation of the target domain is given in Figure 3.1 as a UML class diagram which then acts as a reference for the concrete designs in each metamodeling framework.

For a sufficient overview on the various aspects provided by the frameworks described in the next sections, the UML class diagram is designed targeting the frameworks' more unique features, such as abstract classes, containment relationships, inheritance relationships and self-targeting relationships.

The domain of the class diagram describes a Family containing multiple Person entities. Each Person entity can be either a Man or a Woman. The idea behind this setup was taken from the Sirius Starter Tutorial [1]. To further include containment references, each Person lives in a Town, which is contained within a Country.

[1]`https://wiki.eclipse.org/Sirius/Tutorials/StarterTutorial`, last accessed on 04.01.2023

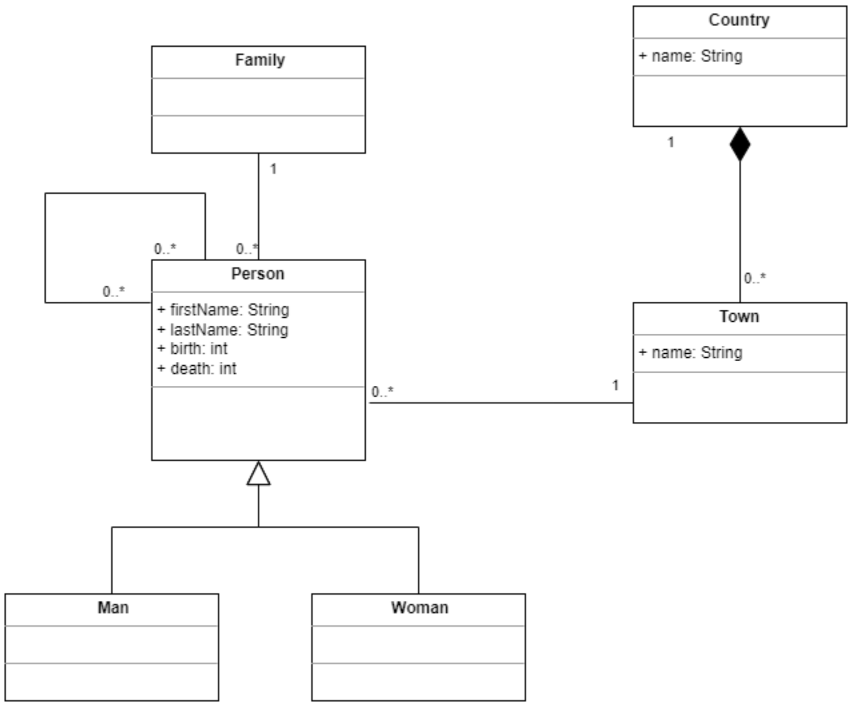


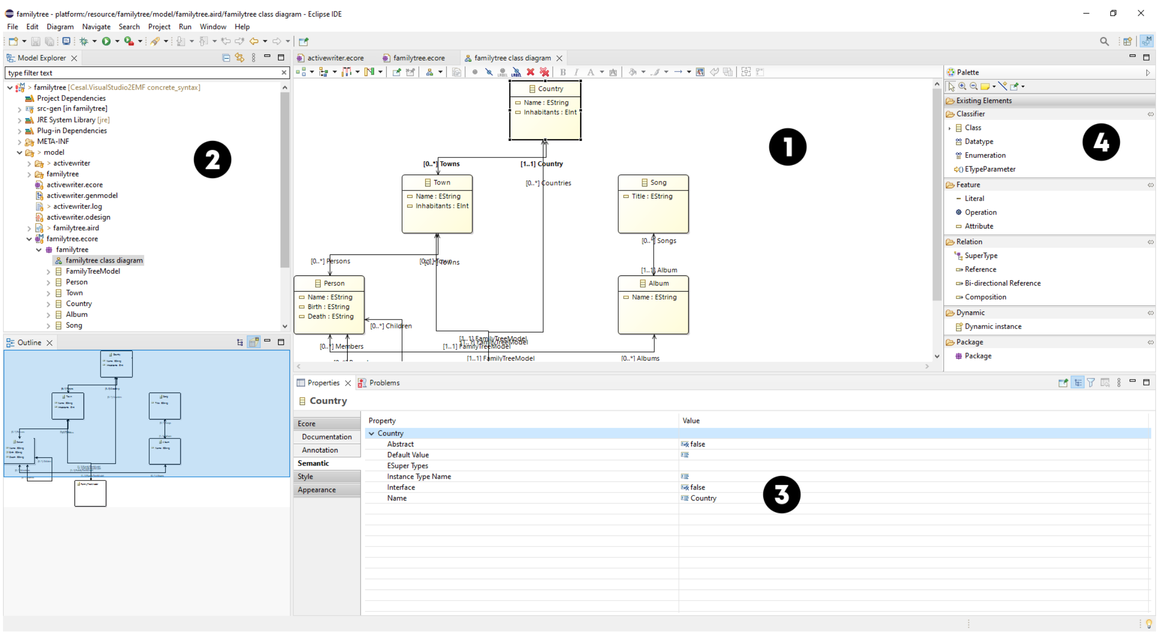
Figure 3.1: Basic Family Tree UML Class Diagram

3.1 Eclipse Modeling Framework

As mentioned before, EMF is a prominent player in the field of Model-Driven Software Engineering (MDSE). EMF provides a rich set of features for, e.g., defining metamodel, generating code, creating and validating models, transforming models, and serializing them in XMI format. Many EMF-based tools exist, and they can all be integrated into and used in modeling projects. EMF also allows runtime support for producing java classes and programmatically manipulating models through reflection. This section describes some of the core features of EMF needed for understanding the transformation approach and provides an overview on how metamodeling is realized within EMF.

3.1.1 GUI

Figure 3.2 displays the graphical user interface that is available when designing and implementing an Ecore metamodel in a modeling project in Eclipse. Ecore is the name for the explicitly available meta-metamodel inside EMF. An EMF meta-model consists of *.ecore* and *.genmodel* files for describing its purposes and functionality. In the *.ecore* file, the abstract syntax elements are defined, which will be explained in detail later in this chapter. The *.genmodel* files are used for Java code generation, i.e., interfaces of the corresponding classes, a factory to create them, concrete implementations which can be manually extended and used via an `AdapterFactory`. Each interface that is created here extends the `EObject` interface, which acts as the base of EMF classes. Additionally, *.genmodel* is used for generating edit and editor code to create model instances of the metamodel and creation wizards.


familytree - platform:/resource/familytree/model/familytree.aird/familytree class diagram - Eclipse IDE
Model Explorer
Outline
Palette
Properties
Problems
1
2
3
4

Country

Property	Value
Country	
Abstract	false
Default Value	
ESuper Types	
Instance Type Name	
Interface	false
Name	Country

Figure 3.2: Graphical user interface in Eclipse Modeling Tools Version 2022-03 (4.23.0)

- **(1) Modeling Canvas:** Here, the modeling canvas shows a class diagram initialized with the help of the Sirius default class diagram editor, which uses a standardized *.odesign* file (i.e., VSM - Viewpoint Specification Model, see Section 3.1.4 for more information) for visually creating and editing metamodels.
- **(2) Model Explorer:** The explorer for the modeling project in general, containing all files needed for displaying the model, either tree-based or visually by initializing a Sirius diagram that uses the textual metamodel tree and vice-versa.
- **(3) Properties:** In this area, different properties for the selected metamodel entity can be filled and edited with various values.
- **(4) Palette:** EMF contains a tool palette for simply dragging and dropping elements onto the canvas (1) and linking them through different types of relationships.

3.1.2 Abstract Syntax in EMF

Regarding Figure 2.1 depicting the metamodeling hierarchy, realizing metamodel support requires specifying it by instantiating concepts from the EMF meta-metamodel, namely Ecore. The expressiveness of all possible metamodels is determined by said meta-metamodel, therefore playing an essential role in the definition process. The Ecore meta-metamodel is explicitly defined in various sources, e.g., in [9, 10, 29, 31]. Figure 3.3 displays an excerpt of this meta-metamodel.

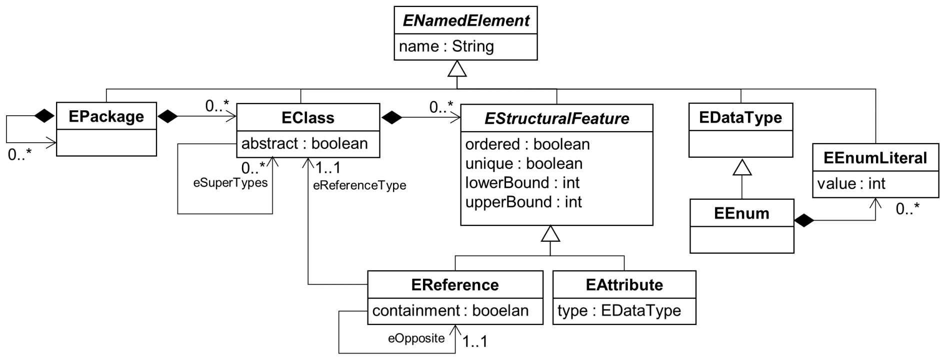


Figure 3.3: Excerpt of the Ecore meta-metamodel [9]

An Ecore metamodel is comprised of one or more *EPackages*. Each *EPackage* can contain multiple *EClasses*, whereas each *EClass* can contain multiple *EStructuralFeatures*. These features are divided into two types, namely *EAttributes* and *EReferences*. *EAttributes* resemble properties of *EClasses*, they have an *EDataType* ranging from simple datatypes (e.g., Integer, String) to more complex ones (e.g., user defined external types). *EReferences* are used for linking two *EClasses*. Inheritance relationships are realized by defining *ESuperType* relations on top of an *EClass*. EMF allows the definition of single and multiple inheritance relationships. By setting the containment flag of a *EReference* to "true", a composition relationship between source and target *EClasses* can be defined, resulting in cascading deletion behavior, where deleting the containment itself also automatically deletes its contained elements as well.

3.1.3 Model Tooling

A detailed introduction to EMF is given in `https://www.vogella.com/tutorials/EclipseEMF/article.html`, last accessed on 04.01.2023, where all aspects from project initialization, to metamodel editing (class creation, attribute definitions and relationship creation), to code generation, to model editor generation and running editor code for model creation are explained. A detailed example based on a real world scenario on how one integrates MDSE into a project with the help of EMF is also given in [10]. A quick overview on the different diagram elements available for specifying metamodels, regarding the tree based editor and the default Sirius tool palette for Ecore metamodels, is given as follows:

- **Classifier:** Containing elements for creating classes, datatypes and enumerations.
- **Feature:** Features range from class attributes to class methods and literals.
- **Relation:** All Relationships that can be defined between classes can be taken from here, like inheritance, reference, bidirectional and containment references.

- **Dynamic (only class diagram):** Creates a dynamic XMI model instance from the selected class element contained in the current model.

- **Package:** Used for grouping elements together. Packages can interchange information and element usage.

3.1.4 Concrete Syntax in EMF

The Eclipse Modeling Project Website lists three frameworks that can be embedded into an eclipse installation and used for visualizing ecore metamodels and models, namely GLSP (Graphical Language Server Platform) [2], Sirius [3] and Graphiti [4]. The integration of the Sirius framework into EMF adds functionality for defining graphical representations on top of Ecore models, their classes, relationships and the different types that Ecore offers to integrate in one's own modeling language makes it possible to design beautiful and concrete domain specific models and metamodels for better readability and usability in software project teams that integrate Model-Driven Software Engineering. For the matter of this thesis and its transformation approach, the well established Eclipse Sirius framework was selected for defining and using elements and representation options inside the transformation bridge, as it also best resembles the possibilities of graphical representations a language designer can use on the MSDKVS side, where the concrete syntax is integrated into the framework as is and cannot be exchanged. Furthermore, it offers solutions for rapid development of graphical concrete syntax on the EMF side without requiring a lot of understanding about the backend processes [37, 38].

When opening a model or a metamodel with references to a viewpoint specification model (*.odesign*), an additional file is created called an *.aird* file, which contains Sirius representation data resembling the currently displayed representation of the metamodel based on VSM metadata. This *.aird* file is dynamic in terms of its XML data representation and changes its content frequently. *.Aird* files contain the data that is needed for graphically representing the models during runtime, and are generated when a viewpoint is selected to visualize (meta)model content. Semantic data is not contained within them, and their volatility in terms of content makes it impossible to generate them programmatically.

A detailed mapping used in the transformation approach between EMF's Sirius framework and MSDKVS included diagram mapping is given in Chapter 5.

With Sirius, a Viewpoint Specification Model (VSM) containing one to several viewpoints within *.odesign* files can be used for describing not only the concrete syntax on the model layer, but also on the metamodeling layer. For the implementation of the M3B transformation, only viewpoints on the model layer are used for transforming graphical syntax elements from and to EMF. VSMs are used for managing the different concepts the Sirius framework offers, like styles, tooling and mapping to (meta)model objects.

[2]`https://www.eclipse.org/glsp/`, last accessed on 04.01.2023
[3]`https://www.eclipse.org/sirius/`, last accessed on 04.01.2023
[4]`https://www.eclipse.org/graphiti/`, last accessed on 04.01.2023

Identical to the GUI for defining metamodel elements, Sirius offers a property editor window when a corresponding element is selected, making it possible to access different domain element like *EClasses* and *ERelationships* for attributing them with customized styles and tooling capabilities. The valid elements for each description can be restricted by applying validation rules in the form of expressive query languages like OCL, AQL or Acceleo.

Regarding the structure of a VSM in Sirius, the following elements are worth mentioning:

- **Group:** The root element of any Sirius specification file. This element can be compared to the Ecore's abstract syntax *EPackage* element.
- **OwnedViewpoints:** Acts as a wrapper for multiple *OwnedRepresentations*.
- **OwnedRepresentations:** These represent the data of the metamodel and model, referencing their abstract syntax features and enriching them with a variety of appearances and behavior. Sirius supports multiple different types of representations, the most important and relevant for this thesis being the *diagram representation*. Tables, trees and matrices are the remaining ones not being targeted by this thesis' transformation approach.
- **Layers:** Every representation has to have one *defaultLayer*, and can contain multiple *additionalLayer* elements, grouping together the different styles a designer can define for specific metamodel elements.
- **NodeMappings:** As the most basic form of graphical shape mappings in Sirius, these are used for styling *EClasses* that do not contain other *EClasses*. Each mapping, depending on their target element, can define different styles usable by said abstract syntax entity.
- **ContainerMappings:** When an *EClass* is used inside a containment *EReference*, the container itself and its sub nodes can be styled by using *ContainerMappings* and *SubNodeMappings*.
- **EdgeMappings:** These types of mappings are used for graphically attributing and styling *EReferences* between two *EClasses*. Specialized properties such as, e.g., `TargetArrow` and `RoutingStyle`, are definable here.
- **ToolSections:** Every VSM can contain multiple *ToolSections*, separating the different tools into designated areas. Each *ToolSection* can contain various types of *OwnedTools*, each usable on different abstract syntax elements. What kinds of elements can be targeted by each tool can be specified by a multitude of available query languages. Each *OwnedTool* contains one ore more operations, namely *firstModelOperations* and *subModelOperations*, to define sequences of operations the tool has to execute in order to manipulate the modeling canvas when the tool is being used.

- **User Colors Palette:** Different kinds of colors, that are usually not part of the framework itself, are explicitly added here as *entries* by their red, green and blue values, and referenced from within the viewpoints by their names.

3.2 FamilyTree Example Metamodel in EMF

Based on the UML class diagram (see Figure 3.1) and with regards to the MSDKVS DSL definition in Section 3.4 for a basic familytree language, a similar metamodel based on Ecore has been defined in EMF, shown in the following Figure 3.4.

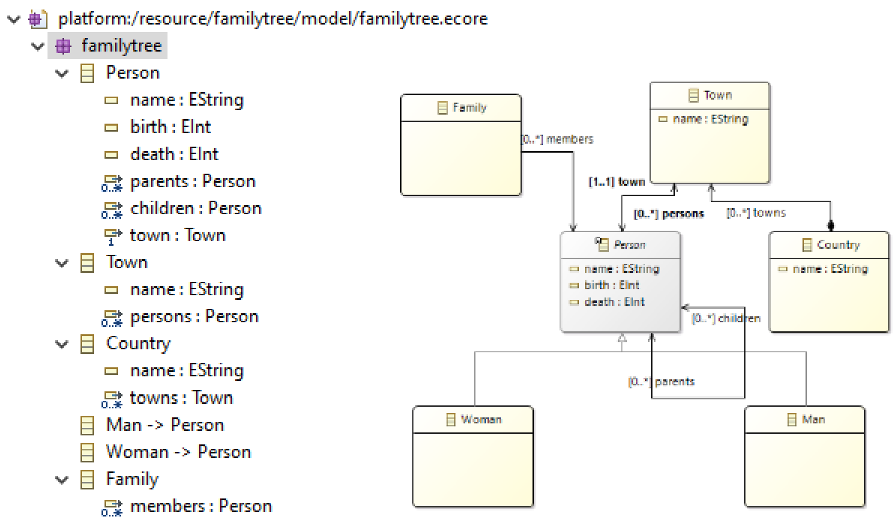


Figure 3.4: Basic Family Tree as an Ecore metamodel in EMF, left in the default Ecore tree-based editor view, right as a Sirius VSM

3.3 Modeling SDK for Visual Studio

This section describes the aspects of MSDKVS (also sometimes referred to as VMSDK, meaning Visual Studio Modeling and Visualization SDK) with regards to metamodel definition, model creation and validation, the framework's interface and different tools the framework provides. A detailed description on the framework's functionality and usage guidelines are given on the official Microsoft Online Documentation [35] or in [17].

3.3.1 Framework overview

MSDKVS supports the development of domain-specific languages by weaving together abstract and concrete syntax. MSDKVS offers a graphical user interface with an integrated editor to define classes, relationships, attributes and their types for a particular domain as well as an explorer, a property editor window and several other features such as XML serialization behavior of models, code generators using a templating engine (specifically the T4 text template engine to generate text files) and the possibility to build extensions to all these features.

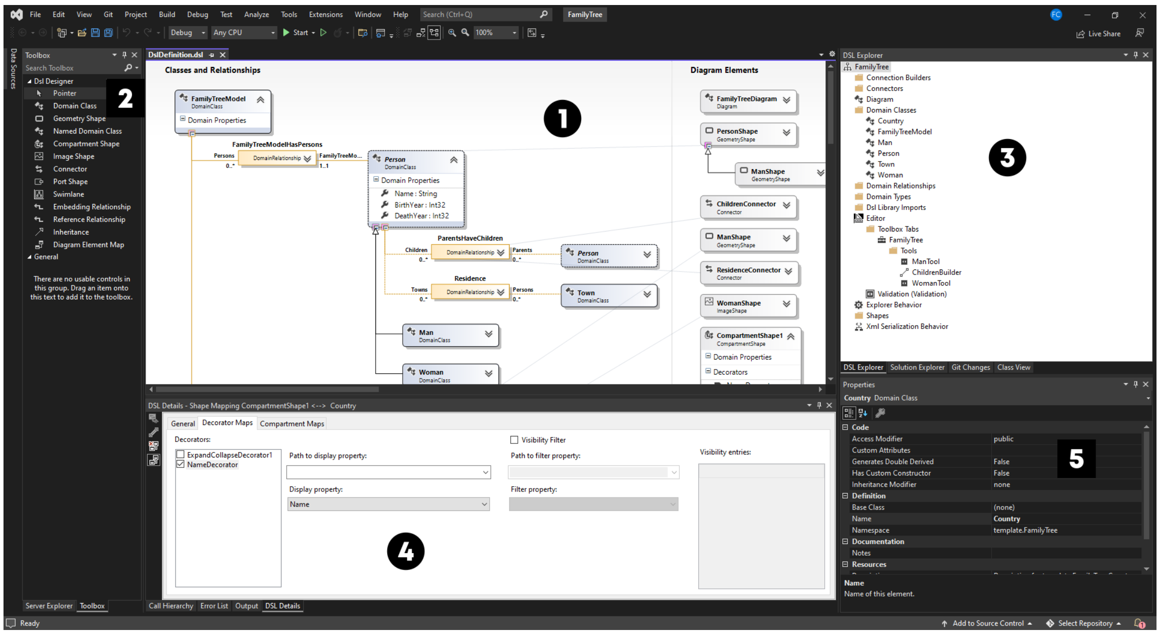
The project type in Visual Studio to create DSLs is called "Domain-Specific Language Designer" project and the modeling suite for Visual Studio can be separately installed through the Visual Studio Installer. After selecting the project, different types of project templates for creating one's own modeling language are selectable, including class diagrams, component models or task flows. Each of these templates comes with a predefined project structure to directly start working on the DSL by editing the automatically created application data. The most important file is the *DslDefinition.dsl* file within the Dsl project, containing all the syntax definitions (abstract and concrete syntax) and their validation constraints, serialization behavior, tooling capabilities and much more.

Two projects are created at startup, the Dsl project and the DslPackage project. The **Dsl** project is used for designing the language and the **DslPackage** project is used for running the separate Visual Studio instances to create models based on the designed DSLs. Distribution of published DSLs can be done through the creation of Visual Studio Extension files (*.vsix*) created when the solution is built.

3.3.2 GUI

Figure 3.5 shows a solution for DSL projects opened in the currently available 2022 Community Edition of Visual Studio. An overview with descriptions of the various numbered areas is given as follows:

- **(1) Dsl Canvas:** Elements from the Toolbox can be added and are shown on the left hand side of the canvas. The canvas is divided into two areas, whereas the larger left area contains the domain classes and domain relationships that have been created and the links between them. The smaller right side is titled "Diagram Elements" and contains shape definitions for classes, relationships and their attributes (i.e., graphical concrete syntax). These shape definitions can be mapped to elements in the "Classes and Relationships" area through "Diagram Element Mappers" to graphically enhance model elements.
- **(2) Toolbox:** The Dsl Designer section of the toolbox contains all nodes that can be dragged and dropped onto the canvas, creating either a class, relationship, links and different types of shapes.
- **(3) DSL Explorer:** This window can be seen as a graphical representation of all values contained within the XML schema representation of the DSL, where not only the nodes in the canvas are listed in their corresponding categories, but also the editor functionalities like toolbox operations, serialization behavior and custom C# types as well as *DomainEnumerations* can be added and removed.



Code	
Access Modifier	public
Custom Attributes	
Generates Double Derived	False
Has Custom Constructor	False
Inheritance Modifier	none
Definition	
Base Class	(none)
Name	**Country**
Namespace	template.FamilyTree
Documentation	
Notes	
Resources	

Figure 3.5: Graphical user interface in Visual Studio Community 2022 of MSDKVS

- **(4) DSL Details:** This window displays content when a class node, a relationship node, shape node or a mapping between these nodes has been selected. Here, links and values for decorators and delete propagation behavior can be tied together and added to the selected element.
- **(5) Properties:** This window contains all possible properties and their values for the selected element on the canvas. These entries resemble attributes in the elements' XML representation.

3.3.3 Code generation

While designing a DSL, MSDKVS adds an API to it that can be called to read model information and update these instances dynamically through generating text or source code files using T4 Text Templates. These template files can be supplemented with C# or Visual Basic code snippets that use the specific model instance as reference.

3.3.4 Model creation and validation

To create a model for the previously implemented DSL in the DSL designer GUI, a separate experimental Visual Studio instance has to be started by running the automatically created *DslPackage* project within the solution. The DSL definition contains information about what kinds of classes and relationships the model can contain as well as their appearances by defining several styles for these class elements (called shapes), for relationships (called connectors) and their domain properties (called decorators).

3.3.5 Abstract Syntax in MSDKVS

While MSDKVS does not offer a publicly available view on the meta-metamodel, the transformation itself implicitly offers the ability to generate said meta-metamodel by using the data structures for serializing the input metamodel files in the form of class diagram. The full representation of said class diagram after the transformation on the M2 layer has been implemented can be investigated in Appendix D at the end of this thesis. An abstracted excerpt of the meta-metamodel file is shown in Figure 3.6.

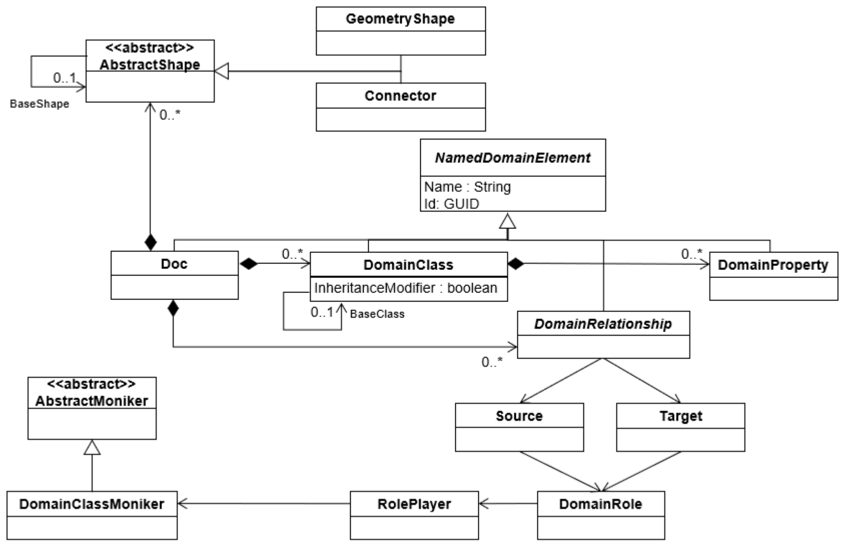


Figure 3.6: Excerpt of the MSDKVS meta-metamodel [15]

When creating a DSL in MSDKVS, one element always has to act as the root element of the dependency tree and every subsequently created *DomainClass*, if not targeted by another embedded relationship, is referenced by this root class. The root class by default has the same name as the DSL itself, but can be changed after initial creation. Regarding transformation mappings, relationships from every class to a DSL's root class are embedded relationships, meaning that they can directly be mapped to containment references in Ecore.

The possible entities that can be created on the metamodeling canvas are available in the Dsl Designer Toolbox. The following table contains all objects within the toolbox that create abstract syntax elements and are added to the serialized XML representation file of the DSL. First, a list of the elements of the *DslDefinition* project canvas obtained from within the left part of the workbench titled "Classes and Relationships" is given as follows:

- **Domain Class:** These are used to define model elements for the domain. *DomainClasses* are the most common elements for defining a DSL.

- **Domain Relationships:** Relationships in general are used for connecting model elements (i.e., instances of domain classes) through links. Relationships offer different customization features (e.g., `multiplicities`) and their source and target objects are called roles, which can have different names than the classes they represent. An important feature of MSDKVS is that domain relationships can be the target of other relationships as they are themselves usable as classes, i.e., can contain attributes.

- **Embedding Relationship:** Displayed with solid lines, embedding relationships are used for defining containment references (i.e., class ownership relations) between two elements in the language. In MSDKVS, the root element has an embedding relationship to all domain classes either directly or indirectly through other referenced classes, and this element itself acts as the metamodel diagram, where all model elements are added to on the M1 layer. Another important aspect is that, if the parent element of the embedding relationship is deleted, all children are removed as well.

- **Reference Relationship:** All relationships that are not embedding are called reference relationships and are displayed as connector shapes (described in Section 3.3.6) and the deletion of referenced elements does not cascade to other linked elements. Every relationship contains a source and a target role, resembling other DSL elements. No distinction between bi-directional references and simple references, as is done in Ecore, is made here, thus usually containing less reference entities than EMF.

- **Inheritance:** Inheritance relationships are a special kind of relationship, where one node inherits features (i.e., relationships and attributes) of the node it is related to. It can be compared to the typical object-oriented definition of inheritance, except that in the environment of metamodeling in MSDKVS, not only class and relationship nodes, but also shape nodes can be inherited from. This is an important aspect that has to be considered when transforming MSDKVS DSLs, such that, e.g., each decorator property from the shape that is inherited from is also transformed for the object the sub shape is mapped to. MSDKVS also only allows single inheritance, in contrast to EMF where an *EClass* can inherit attributes and behavior from more than one base class, which will also be elaborated in more detail in Chapter 5 when the transformation rules are described.

These canvas items are displayed in views that can be adjusted through different contextual actions, like reordering of classes or collapsing / expanding of entities. Each of these elements can be attributed by adding a number of *DomainProperties* to them, e.g., a "name" property with type String or a "created" property with type DateTime.

Regarding their XML representations, the *DslDefinition.dsl* file, when opened in a text editor, contains all canvas objects and their mappings to tool palettes, shapes (i.e., graphical concrete syntax) and serialization properties. Every object on the canvas is given a so-called *Moniker* type to be referenced in other parts of the editor. Monikers are uniquely identifying names for elements. The XML content is grouped in areas where cross referencing other elements through these moniker types is made possible. These groups include:

- **Classes**, where all domain classes and their properties are nested within.
- **Relationships**, where different relationship types with their source and end points defined as roles and referenced through domain class monikers are contained.
- **Types**, where a number of external common system types is listed by default, such as String, Double or Boolean. *DomainEnumerations* and custom types are listed here as well.

3.3.6 Concrete Syntax in MSDKVS

Every class and relationship can be decorated with corresponding shapes that are maintained on the right hand side within the editor's graphical interface. They describe the appearances of the metamodel object instances on the model layer. Compared to EMF and Sirius respectively, MSDKVS does not offer the possibility to define concrete syntax used on the M2 (metamodel) layer. How metamodel entities are represented is not adjustable and is the same for every metamodel defined on this platform.

The exact representation of the concrete syntax elements is visible when examining the projects *DslDefinition.dsl* file that is used for displaying and editing the metamodels, serialized in XML format. As has been done for the abstract syntax, the canvas elements and XML tags relevant for transforming graphical concrete syntax elements are listed as follows with descriptions on how they are represented and editable inside the GUI. A comparison and the definition of mapping rules between these elements and the corresponding syntax on the EMF side is finally given in Sections 5.2.2 and 5.3.2 in Chapter 5.

- **Geometry Shape:** Each geometry shape defines the layout and graphical representation of the linked *DomainClass*. Each shape contains several *DomainProperties*, which can define, e.g., the color of the *DomainClass* and the default size parameters. Additionally, for every *DomainProperty* the targeted class has, one or more decorators can be defined through *DecoratorMaps* that can contain numerous properties like font attributes, text orientation and different line types for the class nodes' borders like dashed or dotted.
- **Compartment Shape:** These shapes can contain multiple *DomainClasses*, displayable in different styles like lists or tables. In this thesis' example, this would be the country entity containing multiple town entities. Only *DomainClasses* having containment references can be visualized by *CompartmentShapes*.

- **Image Shape:** Instead of geometries, these shapes are represented on the model layer based on their image target, which is normally required to be a bitmap file and must be part of the resources folder of the modeling project. Only *DomainClasses* can be visualized by *ImageShapes*.
- **Connector:** *Connectors* define the layout and styles for *DomainRelationships*.
- **Port Shape:** *Ports* are shapes that are attached to other shapes and they can be used to define specific regions for connecting other diagram elements.
- **Swimlane:** With *Swimlanes*, the language designer is able to divide the diagram, either horizontally or vertically, whereas a *Swimlane* resembles a model element on the canvas.
- **Diagram Element Map:** This tool is used to map a *DomainClass* or a *Domain-Relationship* to a valid shape on the canvas.

Regarding objects in the right area of Figure 3.5, following noteworthy regions can be extracted from their XML schema content:

- **Shapes:** This area contains all shapes that have been defined on the canvas. Every shape except connectors are declared here with properties defining their appearances (in MSDKVS called decorators). Each shape can contain multiple decorators and these decorators reference *DomainProperties* that belong to the referenced class (i.e., domain class, compartment). Same as for every other object, shapes contain unique id values (*GUIDs*) for being able to cross reference them in other areas.
- **Connectors:** Contains a list of shapes for *DomainRelationships*.
- **XmlSerializationBehavior:** Contains different cross references to other objects via their moniker types and adds additional information on their serialization features as well as modeling functionalities and available context menus when interacting with the tree editor of the underlying model. How the modeling entities are represented inside the XMI serialized file is configurable here.
- **ExplorerBehavior:** Custom explorer behavior for nodes is declared in this section. This area is out of scope for this thesis' transformator as it does not directly affect the abstract and concrete graphical syntax validity.
- **ConnectionBuilders:** These are used for adding reference creators that are used for linking objects together with *Connector* shapes, targeted by tools.
- **Diagram:** This element contains all shape maps between shape nodes and syntax elements as well as design properties for the modeling canvas when running the experimental IDE instance, like color attributes for backgrounds and texts.
- **Designer:** The designer region covers the definition of tools with which the modeler is able to design the model based on its DSL.

3.4 FamilyTree Example DSL in MSDKVS

In comparison to the previous defined diagram and metamodel in UML and EMF respectively, Figure 3.7 shows the equivalently defined DSL in MSDKVS, with the same classes, references and attributes for easy comparison.

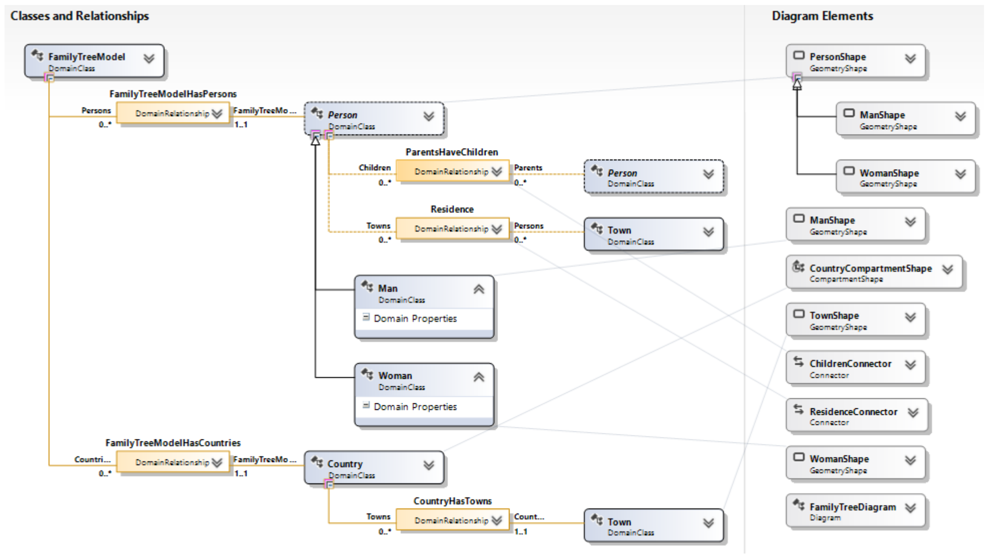


Figure 3.7: Basic Family Tree as DSL in MSDKVS

3.5 Comparative Analysis of Framework Features

For getting a good overview on what features are available within each framework, the following Tables 3.1 and 3.2 to 3.4 are used for comparing the two platforms. The tables consist of three columns, the first column is used to describe and name the identified feature, the other two columns denote for each framework respectively whether or not the feature is available in it and how it is implemented. The list of relevant elements was adapted and extended from [30] in terms of first class concrete syntax concepts extracted through detailed investigation of the two frameworks (i.e., their available documentation and hands-on experience). The concepts have been abstracted for better comparison thus creating a standardized table on core abstract and graphical syntax features in metamodeling frameworks. The features (i.e., rows of the table) are then again discussed separately while taking both frameworks into account to discover the major similarities and differences.

Criteria	**Ecore**	**MSDKVS**
ASM Concepts		
Class	EClass	DomainClass
Relationship	EReference	DomainRelationship
Attribute	EAttribute	DomainProperty
Enumerations	EEnum	DomainEnumeration
Role	✗	DomainRole
Grouping	EPackage	Language, Namespace
Classes		
Abstract Classes	✓	✓
User-defined root element	✓	✗
Relationships		
Arity	binary	binary
Composition	✓	✓
Multiplicity	✓	✓
Inverse	✓	✗
Endpoints	EClass	DomainRole
Unique Names	✓(per Class)	✓
Link to Model	✗	✓
Attributes		
Applicable to	EClass	DomainClass, DomainRelationship, Shapes
Multiplicity	single-/multi-valued	single-/multi-valued
Unique	✓	✓
Ordered	✓	✗
Default Value	✓	✓
Custom Data Type	✓	✓
Access Modifier	✓	✓
Enumerations	✓	✓
Roles		
Multiplicity	-	✓
Dependency	-	DomainRelationship
Inheritance		
Single/Multiple	multiple	single
Instantiation	single	single
Class Inheritance	✓	✓
Relationship Inheritance	✗	✓
Validation	✓	✓
Constraint Language	OCL	GPL (C#, VB)

Table 3.1: Comparison of abstract syntax features in MSDKVS and EMF with regards to their meta-metamodeling concepts

3.5.1 Abstract Syntax Features

ASM Concepts

These are the main abstract language concepts and are integrated into most metamodeling frameworks. The core features listed here are the definitions of classes, links (i.e., relationships between classes) and attributes (i.e., properties of said classes and, if available, their relationships). As we will explicitly look at MSDKVS in the course of this thesis, the concept of a role has to be also included as one of the main abstract syntax features. Typically associated with *DomainRelationships*, roles are acting as the source and target entities used as inputs for these relationships. Roles can be used to attribute additional values on top of a class entity. This means that one class can have multiple roles spread out through the language definition that act as so-called role players on both ends of a link (i.e., an intermediate object for referencing the class itself).

Grouping

Many metamodels contained within publicly available collections, especially Ecore metamodels, often use more than one package. These packages contain and group definitions of the aforementioned ASM concepts and can make them available to other packages defined in a metamodel by attributing them to referencable namespaces. MSDKVS does not enable the multiple grouping concept, as only one package (i.e., *Language* or namespace) can be defined in a language project.

Classes

The class concept is the main concept a language designer needs to define the basic structure of his or her metamodel. The definition of classes can be made either abstract, as an interface or, by default, public. The ability to manually define a root element is only available in EMF, as in MSDKVS, the root element of every language definition is always a container class created at the initialization of the project. This root element acts as the modeling canvas on the M1 layer.

Relationships

Relationships, in EMF called *EReferences*, a type of *EStructuralFeatures*, in MSDKVS called *DomainRelationships*, in addition to classes are needed to define a metamodel or DSL. Relationships can have different `Arities`, can be defined bi-directional, can be a containment of subclasses, can have multiplicities on either source or target entities etc. These source or target entities are *EClasses* in EMF and *DomainRoles* in MSDKVS, whereas *DomainRoles* reference *DomainClasses* through moniker types. Each *DomainClass* can have multiple *DomainRoles* spread across the DSL.

Attributes

Attributes are properties added to model entities. In EMF, only *EClasses* are attributable, whereas on MSDKVS, in addition to *DomainClasses*, also *DomainRelationships* and Shapes (*GeometryShapes*, *CompartmentShapes*, *Connectors* etc.) can contain multiple different Attributes. Attributes in EMF are another type of *EStructuralFeatures*, namely the `ecore:EAttribute` type. Inside MSDKVS, attributes are called *DomainProperties*. The unique possibility of attributing *DomainRelationships* makes transforming DSLs to Ecore metamodels interesting, as such relationships have to be transformed to corresponding *EClasses* acting as each target and source of additional *EReferences* in order to preserve their attributes while transforming. Attributes can either be simple types like Integer, String or Boolean or more complex types like a collection of types or user-defined custom types (i.e., *EDataTypes* in EMF and *ExternalTypes* in MSDKVS). Attributes can have default values configured to them. How these properties should be accessed after code generation can also be distinguished on both MSDKVS and EMF. Last but not least, enumeration types, special kinds of attributes, can also be defined on both platforms, *EEnumerations* in EMF and *DomainEnumerations* in MSDKVS respectively.

Roles

As roles are only realizable inside MSDKVS, special transformation rules only apply in the direction of MSDKVS2EMF. Multiplicities are applied to these *DomainRoles* and have to be transformed to *EReferences* and their *EClasses* accordingly.

Inheritance

As mentioned in Section 3.1.2, EMF allows defining inheritance structures containing one or more super classes, whereas MSDKVS only allows single inheritance, meaning one *DomainClass* can only inherit from one other *DomainClass* maximum. Multiple inheritance is a feature that not many modeling frameworks provide. The inherited features super classes pass down to their sub classes include properties and relationship structures. An important aspect of MSDKVS though is the ability to not only inherit these features among *DomainClasses*, but also among *DomainRelationships* and even their corresponding shapes as well.

Validation

Validating the well-formedness of a given metamodel is possible, available out of the box and also encouraged in both platforms. EMF allows turning auto-validating on and off, whereas MSDKVS validates metamodels as soon as the DSL is opened in the respective editor inside Visual Studio. Information regarding syntax validity is either considered as an information, a warning or an error, same as when debugging source code, whereas warnings or information do not break the program's ability to be compiled and executed.

Constraint Language

Further constraining possible design decisions made on the underlying modeling layer (the M1 layer) is available in both EMF and MSDKVS. EMF uses the *Object Constraint Language* (OCL) for describing rules applying to meta-models and their models, specifically the Eclipse OCL variant [5]. MSDKVS on the other hand relies on general programming languages (GPLs), specifically C# and Visual Basic. Through T4 text templates and C# or VB code snippets a language designer can narrow down the valid interactions a modeler has with the designed modeling language.

[5]`https://projects.eclipse.org/projects/modeling.mdt.ocl`, last accessed on 04.01.2023

Criteria	Ecore (with Sirius)	MSDKVS
GCM Concepts		
Integrated	Sirius	✓
Diagram Canvas	✓	✓
Diagram Elements		
Mapped to	EClass, ERelationship, EAttribute	DomainClass, DomainRelationship, DomainProperty
Class Shapes	NodeMapping	GeometryShape, ImageShape
Relationship Shapes	EdgeMapping	Connector
Composition Shapes	ContainerMapping	CompartmentShape
Attribute Layout	Labeling	DecoratorMap
Special Shapes	BorderedNode	Swimlane, Port
Layers	✓	✗
Class Shapes		
Geometries	✓	✓
Icons	✓	✓
Coloring	✓	✓
Layout	✓	✓
Class Geometries		
Four sided shapes	Square, Diamond	Rectangle
Curved shapes	Ellipse, Dot	Ellipse, Circle, RoundedRectangle
Special shapes	Basic shape, Note, Gauge	✗
Class Icons		
Image file format	✓	bitmap
Relationship Shapes		
Line Style	solid, dot, dash, dash-dot	solid, dot, dash, dash-dot, dash-dot-dot, custom
Routing Style	straight, manhattan, tree	rectilinear, straight
Arrowing	✓	✓
Coloring	✓	✓
Sizing	✓	✓
Line Styles		
Solid	✓	✓
Dot	✓	✓
Dash	✓	✓
Dash-Dot	✓	✓
Dash-Dot-Dot	✗	✓
Custom	✗	✓

Table 3.2: Comparison of graphical concrete syntax features in MSDKVS and EMF with regards to their meta-metamodeling concepts

Criteria	**Ecore (with Sirius)**	**MSDKVS**
Routing Styles		
Straight	✓	✓
Manhattan	✓	rectilinear
Tree	✓	✗
Composition Shapes		
Geometries	✓	✓
Icons	✓	✓
Coloring	✓	✓
Layout	✓	✓
Background Style	✓	✓
Inner shapes	✓	✗
Composition Geometries		
Gradient	✓	✓
Parallelogram	✓	✗
Image	✓	✗
Appearance		
Fill Color	✓	✓
Gradient	✓	✓
Coloring		
Named colors	✓	✓
RGB	✓	✓
Style		
Border	Line Style, Size, Color	Line Style, Size, Color
Background	✓	✓
Foreground	✓	✓
Labeling	✓	✓
Labeling		
Sizing	✓	✓
Formatting	✓	✓
Alignment	✓	✓
Offsetting	✗	✓
Positioning	✓	✓
Inheritance		
Shape Inheritance	✗	✓

Table 3.3: Comparison of graphical concrete syntax features in MSDKVS and EMF with regards to their meta-metamodeling concepts continued

Criteria	**Ecore (with Sirius)**	**MSDKVS**
Tool Palette		
Sectioning	✓	✓
Creation Tools	✓	✓
Edition Tools	✓	implicit
Deletion Tools	✓	implicit
Copy Paste Tools	✓	implicit
Creation Tools		
Class Creation Tool	Node Creation	Element Tool
Relationship Creation Tool	Edge Creation	Connection Tool
Composition Creation Tool	Container Creation	Element Tool
Filter Mechanisms		
Mapping Filter	✓	✗
Variable Filter	✓	✗

Table 3.4: Comparison of graphical concrete syntax features in MSDKVS and EMF with regards to their meta-metamodeling concepts continued

3.5.2 Graphical Concrete Syntax Features

The identified graphical concrete modeling syntax concepts were retrieved and abstracted after investigating both EMF, including its Sirius framework, and MSDKVS to be able to map and compare them to other metamodeling frameworks that also offer some kind of graphical notation. The following list based on the previous tables explain each subsection with regards to these two frameworks.

GCM Concepts

Whereas MSDKVS offers integrated graphical concrete syntax elements, functionality and validation, the Eclipse environment has different frameworks which are used to design metamodels within a GUI and provide tool support, mentioned in Section 3.1.4. In the course of the M3B and this thesis' transformation approach we consider Sirius as it is the most commonly used framework and best resembles the possibilities of viewpoint representation in comparison to the MSDKVS side. Sirius uses so called *Viewpoint Specification Projects (VSP)* containing descriptive model files ending with *.odesign* [37]. These files contain the specification of the graphical representation of a model and are comprised of layer definitions and tool sections containing toolbox operations with a structured dependency tree of further inner operation mappings, style mappings for model shapes, font layout properties, custom color definitions, and much more. Another identified and important aspect is the notion of a diagram canvas, where model entities can be dragged to and from and the metamodel (or DSL) can thus be graphically manipulated. Available not only on the M2 layer, but on the M1 layer as well, where tools defined through the graphical concrete syntax portion of the framework are used to restrict the possible operations a model designer has. Both frameworks have to set an entity object (i.e., a class entity) to be the diagram canvas' root element. In EMF, this is done when creating the model (i.e., the M1 layer), in MSDKVS this is done when designing the DSL (i.e., the M2 layer), where only one element can act as the root element and, when starting a Visual Studio instance for creating models, becomes the canvas entity. Therefore, it is necessary for this root element on MSDKVS to be able to traverse through all metamodel objects either through reference or containment relationships.

Diagram Elements

The available diagram elements are usually comprised of different shapes and modifications thereof to which abstract syntax elements like classes, relationships and also attributes can be mapped. Both frameworks offer the possibility to define specific shapes to the aforementioned abstract syntax. *NodeMappings* are used in EMF to define shapes on classes, in MSDKVS they are called *GeometryShapes* or *ImageShapes*. Relationships are shaped by *EdgeMappings* on EMF and by *Connectors* on MSDKVS. Classes that use containment references have to be mapped to either *ContainerMappings* in EMF and *CompartmentShapes* in MSDKVS, where each framework also offers the possibility to define shapes on the contained elements as well. Attributes, on either classes or even relationships when talking about metamodeling in MSDKVS, can be graphically altered by using *Labels* within style elements on EMF and through *DecoratorMaps* in MSDKVS, their targets ranging from more simple decorators like *TextDecorators* to more sophisticated ones like *IconDecorators* and *ExpandCollapseDecorators*. Aside from these standardized shapes, each framework also contains special shapes, which have to be considered when transforming the graphical concrete syntax. Sirius features *BorderedNodes*, which can be mapped to MSDKVS *Port* shapes applied to a compartment's subnodes. MSDKVS further offers *Swimlane* support for dividing the canvas into separate areas.

Class Shapes

Identified categories of class shapes that were found in both frameworks include geometrical class shapes and custom icon shapes. Each of these can be further customized by applying colors on different styling aspects like background, foreground or borders. Layout properties like width or height parameters and weight can be applied to class shapes on both platforms.

Class Geometries

Based on the available geometry shape definitions for classes following distinctions were recognized:

- **Four-sided Shapes**: Shapes that have four sides include *Squares* and *Diamonds* on EMF and *Rectangles* on MSDKVS, although square-like shapes can be defined by using the same height and width attributes.

- **Curved Shapes**: Both frameworks offer *Ellipses* as curved shapes, as well as *Dots* on EMF and *Circles* on MSDKVS. *Rounded Rectangles* are a special kind of rectangular shape where the edges are rounded based on a specific shape property.
- **Special Shapes**: As opposed to the previously mentioned special shapes, these special shapes are explicit types of class geometries. EMF allows setting the type of a *NodeMapping* to be used as, e.g., *Notes* or *Gauges*. MSDKVS does not have other class shapes than those mentioned previously.

Class Icons

Regarding shapes, metamodeling platforms also allow the submission of custom image files to adapt the look and feel of the models. EMF and MSDKVS differ in their support of various file formats. Icons can be used to, for example, add custom appearances to tool palette items or composition shapes. As mentioned before, every image that is attributed to an element on the language canvas has to be in BMP format for MSDKVS, EMF does not exclude any of the well known image formats. This characteristic has to be explicitly considered when transforming between these frameworks.

Relationship Shapes

Relationship Shapes are mapped to relationships on the metamodeling layer. The design properties set here are reflected on instances of said relationships on the modeling canvas in the runtime instances of the editor. Each relationship shape can set a line style, routing style for connecting lines between two classes. Arrowing targets the start and end layout of a reference. Each link can also be colored and sized accordingly.

Line Styles

Both metamodeling platforms offer the opportunity to set the line style of a relationship, where usually the number of used dots and dashes can be set by predefined order configurations. MSDKVS also supports the possibility to define custom sequences of this dot and dashes.

Routing Styles

Routing styles are concerned with how a link between two elements is formed, which can either be a straight line or in the form of the manhattan routing style, which is called `rectilinear` style on MSDKVS.

Composition Shapes

Composition shapes offer nearly identical styling options as class shapes, the exception being setting shapes for inner classes. MSDKVS does not offer this option whereas EMF allows defining inner shapes.

Composition Geometries

Several types of geometries or shapes for compositions have been found, the default being a gradient, which is present in both platforms. Looking at EMF and Sirius, further shapes are offered which include parallelograms (i.e., `Lozenges`) and also image shapes, which are both not available in MSDKVS.

Appearance

Appearances regarding class shapes, in particular composition shapes, include additional coloring options such as setting a gradient or a single filling color.

Coloring

As metamodeling frameworks often depend on an underlying programming language (e.g., EMF on Java, MSDKVS on C#), the available coloring options, different styles, and appearance attributes are limited by the languages' libraries. As for MSDKVS, three different types of color palettes are available (system, web, and custom). EMF offers a selection of basic system colors per default.

Style

These style section targets every possible shape of the selected platforms. Styles can target different configuration mappings, e.g., a border line style for a class or the foreground color of a composition shape. Styles from MSDKVS and EMF can be easily mapped to each other as both offer the same styling functionalities.

Labeling

Labeling includes the opportunity to style labels, which are mapped to attributes of classes and relationships, accordingly. Normally, these attributes are represented through text labels, therefore editing the font sizes, font families, positioning, text alignment among other things are common features. Setting an offset to said text labels is only available on MSDKVS although it is considered a prominent feature especially in web development so it was added to the list of recognized features.

Inheritance

MSDKVS offers the possibility to inherit shape properties among shapes. The equivalent shape mappings on EMF with their styles and attributes do not offer this functionality.

Tool Palette

Different types of tools have to be defined to work with defined metamodeling in a runtime environment on a modeling canvas. The granularity of what types of tools can be created and customized varies greatly between EMF and MSDKVS. MSDKVS only allows the definition of essential element creation tools for domain classes and domain relationships. On the other hand, EMF offers the definition of a vast amount of additional tools containing, e.g., edition tools, copy-paste tools, or reconnect edge tools. This functionality is not customizable on MSDKVS, but some are automatically used when a creation tool is created. Thus, copying and pasting or deleting elements on the modeling canvas works out of the box. Having no tool palette defined does not mean that interaction on the modeling layer is not possible. Integrated textual tree editors with context menu operations is always available to add elements to the model.

Creation Tools

Regarding the creation tools, which are integrated into both platforms, each one offers a class creation tool (*NodeCreation* in EMF / Sirius and *ElementTool* in MSDKVS) and a relationship creation tool (*EdgeCreation* in EMF / Sirius and *ConnectionTool* in MSDKVS). Creating a composition uses a separate creation tool on EMF, namely the *ContainerCreation* Tool. MSDKVS creates composition instances through its *ElementTools* as well.

Filter Mechanisms

Additional functionality on the modeling canvas includes the possibility of filtering the instances of classes and relationships based on queries. Different types of filters can be defined on EMF, MSDKVS does not allow applying such filter functionality, therefore it is not explicitly targeted in the mapping rules for the M3B, but it should be considered when additional frameworks are used for transforming graphical concrete syntax elements.

CHAPTER 4

Model-driven Interoperability and Related Work

This chapter first gives a definition of Model-driven Interoperability in the domain of metamodeling and language engineering and afterwards presents existing approaches regarding model transformations between different frameworks and platforms and concludes with related work and gives a quick overview on how transformations among various prominent frameworks have been implemented in the past.

4.1 Model-Driven Interoperability (MDI)

As mentioned in Chapter 1, interoperability deals with the exchange of information between two or more systems, and the ability to use that information in each system respectively [22]. Interoperability overcomes the barriers of encapsulating existing platforms and connects features among different systems, making the underlying tools platform-independent. As modeling languages for software development gained popularity in the early 2000s, a need for transforming the grammarware technical space (i.e., EBNF-based grammar tools) to the modelware technical space existed [40] to achieve interoperability between these sets of tools. When this interoperability was established, a plethora of metamodeling platforms followed, which in turn also raised the need for interoperability amongst them.

Interoperability can be seen as a means for migrating existing systems to newer ones, that, e.g., are better supported or are more well known among project members, or suitable in terms of backwards compatibility to downgrade to previous versions for legacy purposes. Or it can strengthen the collaboration among developers with different programming language backgrounds by providing a format for combining and working on a split code base.

Regarding the interoperability of models, which is the main topic of this thesis, it means the exchange and possible migration of models between several meta-modeling tools, where not only models but their metamodels are exchanged as well. The desired goal is to have as little information loss as possible with regards to the functionality of their core aspects and design concepts. A major problem in this regard is the diverse functionality and internal representations that these tools operate on [30]. [10] defines the aim of model-driven interoperability (MDI) as "defining bridges to achieve interoperability ... by applying model-driven techniques". Concepts (see Section 2.2.1) of different metamodels are matched and based on the results, a transformation from one environment to the other is executed step by step. Another difficulty poses the fact that some metamodeling environments do not offer a description on the meta-metamodel the metamodels conform to, thus no concrete knowledge about the possibilities on designing metamodels can be concluded out of the box. In this regard only the publicly available documentation on how the tool can be used is helpful. As mentioned in Chapter 3, MSDKVS is such a tool that does not offer a publicly available meta-metamodel. Therefore, the first step of the transformation approach proposed and implemented in the course of this thesis implicitly adds a meta-metamodel for MSDKVS inferred by the class hierarchy created for serializing the input files from DSLs created in MSDKVS. The full class diagram is displayed in Appendix D.

4.2 Existing transformation approaches

In the past there have been a lot of approaches concerning model-driven interoperability among different metamodeling languages (i.e., metamodeling frameworks) [9, 13, 26, 27, 28, 30, 31]. They typically consist of one or more model transformation bridges that are used for exchanging data between them. But there have been other proposals for generic or concrete transformations, which will be listed in this section. Generally speaking, there exist two main types of transformations that can be distinguished, transformations among metamodels or models contained in the same framework (endogenous transformations) and transformations of metamodels contained within different frameworks (exogenous transformations) [10, 23]. The latter is the one that we are concerned about in this thesis, where the meta-metamodels our source and target metamodels conform to are not the same, i.e., their origin frameworks differ. These are then so-called *horizontal exogenous transformations*, as the source and target of the transformation are situated on the same abstraction level but adhere to different meta-metamodels. Some of these works mentioned above transform only metamodels, i.e., do not consider the models and concrete syntax interoperability.

4.2.1 ATL - ATLAS Transformation Language

Regarding transformations on the M1 layer (i.e., the models themselves), ATL [1] is a prominent player for defining transformation languages to transform models based on

[1]`https://www.eclipse.org/atl/`, last accessed on 04.01.2023

similar properties in EMF. For this, ATL contains a set of defined rules and helper methods to achieve the transition from model A to model B, where each rule describes how a source part should generate the target entity and vice versa. ATL is a QVT-like language [6], where QVT (Query/View/Transformation) comprises a standardized set of model transformations defined by the OMG (Object Management Group). Besides ATL as a model transformation language, [23] also examines QVT-O and Xtend transformation languages in the course of a controlled experiment. These are categorized as M2M (model-to-model) transformations, whereas on the other hand, M2T (model-to-text) transformations are mainly used inside code generators and not directly part of transformation languages. [10] additionally mentions TGG (Triple Graph Grammars) and ETL (Epsilon Transformation Language) as members of the family of transformation languages.

4.2.2 Bridging Transformations

The most commonly and widely adapted possibility of transforming metamodels between two different frameworks is done via transformation bridges. A bridge is implemented via mapping each feature of the source framework in terms of meta-metamodel functionality to the equivalent feature, if existent, on the target framework. Because most frameworks are built upon standardized and well structured mechanisms on how a language (i.e., a metamodel or DSL) can be developed, many features, mainly concerning the definition of abstract syntax elements, are practically identical with slight adjustments depending on the actual framework. Regarding the usage of concrete syntax elements, either graphical or textual, the differences between the frameworks are more varying.

4.3 Related works

Related works regarding different transformation approaches have been collected and in the course of preparing the transformation for this thesis examined extensively. This section gives an overview on the different types of metamodel interoperability in terms of realization and their targeted frameworks.

In terms of transformation bridges, the approaches have been formalized as M3-Level-Based Bridges (M3Bs) in, e.g., [30] and implemented respectively, for example for the framework pairs Aris and EMF in [27], MetaEdit+ and EMF in [26], Microsoft Visio and EMF in [31] and all of them combined in [29] with references to additional previous work. The most recent work done in achieving interoperability between two different metamodeling platforms, namely between ADOxx [2] and EMF, has been done in [9]. Furthermore, a GME-to-EMF metamodel framework transformation bridge is available in [14]. M3Bs require frameworks to have a three level hierarchy (cf. 2.1, excluding the M0 layer for the runtime instances). Following Figure 4.1 depicts a pattern for M3Bs, where

[2]`https://www.adoxx.org/live/home`, last accessed on 04.01.2023

layers M2 and M1, metamodels and their models, are transformed based on mapping rules made on the M3 layer, hence the name M3B.

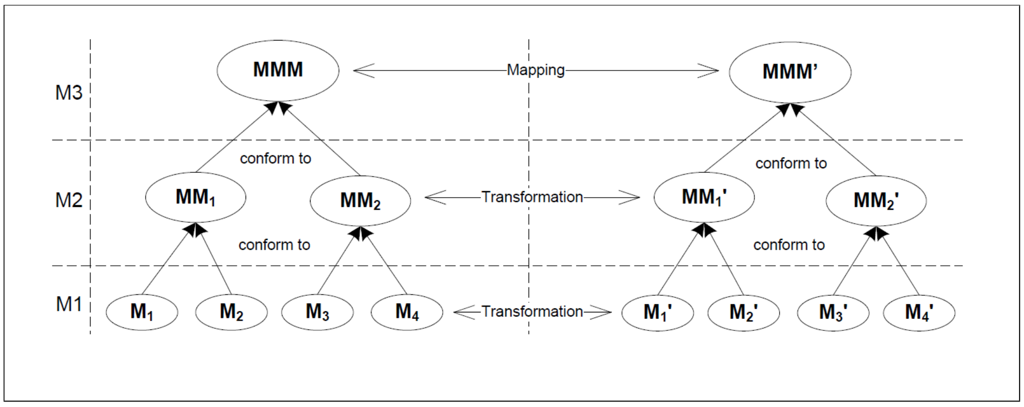


Figure 4.1: Pattern of M3-Level-Based Bridges [31]

Another transformation bridge as mentioned in Chapter 1 for the framework pair Microsoft DSL Tools (where the current MSDKVS is also a part of) and EMF has been conducted in [7, 13]. The transformation approach from these sources has been defined between 2005 and 2010, making them targeting an older release of the current version of the DSL Tools Framework for Visual Studio. Furthermore, the abstract syntax has not been completely included in the transformation approach after having investigated the publicly available source code. Additional differences regarding today's version of MSDKVS as opposed to the transformation approach in [13] are, e.g., the serialized file formats (.dsldm compared to today's .dsl mentioned in [7]), the visualization of a meta-metamodel containing the *ValueProperty* entity compared to today's *DomainProperty*, and the representation of attributes for classes and relationships. Concrete syntax elements where also not transformed accordingly, and no explicit evaluation and feasibility study was given to show the practicability and validity of the proposed approach and its implementation. One example transformation was explained in [13], the PetriNet metamodel, where the question remains if the validation of said transformed metamodel and models was successful in the target platform. In Chapter 7 this gap is addressed properly by providing an exhaustive quantitative and qualitative evaluation of the transformation bridge. [7, 13] and [14] transform metamodels and their models through a chain of ATL transformations to reach the desired outcome. An intermediate transformation of this chain is done with targeting a KM3 (Kernel MetaMetaModel) metamodel, i.e., a DSL for describing metamodels [24] as an intermediate representation of arbitrary metamodels. As a transformation already existed between KM3 and Ecore, the MS/DSL metamodels needed to be only transformed to this pivot KM3 metamodel. Thus, no direct transformation between EMF and MS/DSL tools existed, which introduces potential information loss as KM3 can be considered a generic platform-agnostic DSL to represent the 'common denominator' of several metamodels. In the course of this thesis, the previous approach was examined with preserved .dsldm files of the Atlantic-Zoo Github[3] and it was observed that the execution of the XML2DSL step always resulted in empty files. This is caused by the evolution of the MSDKVS platform (mentioned above) and the discontinuation of some of the used components in the previous approach.

Regarding tool interoperability, where one of the involved frameworks is not a meta-modeling language, like Microsoft's Excel or other applications, [6] uses the approach of defining a pivot metamodel to combine abstraction of features that are common among the source systems. The specific models are therefore not directly transformed to the target tool environment, instead this pivot metamodel serves as an intermediary entity. Another interoperability approach trying to bridge different conceptual data modeling tools (CDML) like ER and ORM2 is given in [11], where a common metamodel, namely the KF Metamodel [21], with rules for transforming from and to these conceptual modelling frameworks is used and implemented in a web based tool called *crowd 2.0* [12]. The intended usage is targeted at achieving semantic interoperability.

This thesis gives a detailed description and comparative analysis on the syntactical elements available in MSDKVS and EMF according to their XML representation and their graphical views inside the platforms' editors. Concerning the concrete syntax elements, no explicit mapping has been done previously, and the mapping between the abstract syntax elements was refined and extended to suit the frameworks' capabilities of their current versions, as well as more semantically appropriate functionalities like toolboxes and abilities of the integrated editors for manipulating models.

[3]https://github.com/atlanmod/atlantic-zoo/tree/main/AtlanticDSLTools

CHAPTER 5

Definition of Transformation Rulesets

This chapter lists and gives a description on all the ruleset that have been defined and how they should be implemented in the first step of the transformator. Every rule is investigated bidirectionally, meaning both EMF to MSDKVS and MSDKVS to EMF variants are explained, the identified mapping of source entities to the corresponding target entities are given, how much additional elements these mappings eventually generate on the target platform are discussed (e.g., additional classes or references as a result of workarounds to stay semantically equivalent on both frameworks) and also if and how much information is possibly lost.

5.1 Mapping of M3 Layer Concepts

Based on the tables regarding the comparative analysis in Section 3.5, Tables 5.1 and 5.2 list the identified rules necessary for transforming EMF and MSDKVS metamodels and groups them accordingly.

5.2 EMF to MSDKVS

5.2.1 Abstract Syntax

AS.R0: *EPackage* $\mapsto$ *Language* As there can be multiple *EPackages* defined in one Ecore metamodel, they and their contents have to be flattened, resulting in only one *EPackage* being transformed to a corresponding DSL's *Language* definition.

AS.R1: *EClass* $\mapsto$ *DomainClass* Three types of *EClassifiers* can be distinguished regarding their `type` attributes, namely *EClasses*, *EEnums* and *EDataTypes*. When

ID	Rule Name	EMF	MSDKVS
AS.R0	Group Mapping	EPackage	Language
AS.R1	Class Mapping	EClass	DomainClass
AS.R2	Relationship Mapping	EReference	DomainRelationship
AS.R3	Attribute Mapping	EAttribute	DomainProperty
AS.R4	Role Mapping	EClass	DomainRole
AS.R5	Data Type Mapping	System Types, Custom Types	System Types, Custom Types
AS.R6	Enumeration Mapping	EEnum	DomainEnumeration
AS.R7	Inheritance	Multiple	Single

Table 5.1: List of mapping rules for abstract syntax concepts

ID	Rule Name	EMF	MSDKVS
GCS.R0	Canvas Mapping	Viewpoint	Diagram
GCS.R1	Class Shape Mapping	NodeMapping	GeometryShape
GCS.R2	Icon Mapping	NodeMapping	ImageShape
GCS.R3	Relationship Shape Mapping	EdgeMapping	Connector
GCS.R4	Composition Shape Mapping	ContainerMapping	CompartmentShape
GCS.R5	Attribute Layout Mapping	Label	DecoratorMap
GCS.R6	Special Shape Mapping	BorderedNode	Swimlane, Port
GCS.R7	Color Mapping	Named Colors, RGB	Named Colors, RGB
GCS.R8	Shape Inheritance	✗	✓
GCS.R9	Tool Palette Mapping	ToolSections	ToolboxTab

Table 5.2: List of mapping rules for graphical concrete syntax concepts

transforming an *EClass*, a target *DomainClass* is created, having the same `Name`. As it is necessary for each element in MSDKVS to have a unique `Id` as an identifier, a GUID is generated upon creation. `Abstract` attributes are transformed to `InheritanceModifier` values. If the current *EClass* is the identified root class of the metamodel, it must not be abstract in MSDKVS. Inheritance feature are transformed as well, transforming possible multiple inheritance structures to multiple single inheritance structures, which will be described in detail later in Section 6.1.3, where the unique features are listed. `XmlClassData` is added to the serialized XML data as well in order to use MSDKVS' code generation and model editor capabilities.

AS.R2: $EReference \mapsto$ *DomainRelationship* Each *EClass* can contain two types of *EStructuralFeatures*, either *EReferences* or *EAttributes*. *EReferences* are transformed to *DomainRelationships*, generating also a unique identifying GUID for the required `Id` attribute. *EReferences* can either be flagged as containment references, simple relationships or a bidirectional references between two *EClasses*. Source and target entities of *EReferences* are transformed to *DomainRoles*, linking each previously transformed

DomainClass via `Monikers` using the unique names of the classes. Regarding the requirements of MSDKVS and the resulting naming strategy employed in this rule refer to Section 6.1.3. Multiplicities of a *EReference* are transformed accordingly.

AS.R3: *EAttribute* $\mapsto$ *DomainProperty* *EAttributes* are transformed to *DomainProperties*. In EMF, only *EClasses* can contain attributes. An attribute can either be defined by a simple value, e.g., `EStrings` or `EInt`, an enumeration type or a custom type, like an external Java class, resulting in different transformation outcomes such as additional *DomainClasses* or even template code files contained within the DSLs namespace. Default values are transformed trivially, as they are definable on both platforms.

AS.R4: *EClass* $\mapsto$ *DomainRole* *DomainRoles* are created for every *EReference*. As the relationships are mapped after all classes have already been transformed, the resulting *DomainClasses* are already available to be used for looking up suitable *DomainRoles*. This rule is executed in combination to rule **AS.R2**.

AS.R5: *DataType* $\mapsto$ *DataType* Different types have been mentioned previously, each transformed in its own way. As simple types are almost identical in their possible range of values, they are trivial. Classes with type *EDataType* are transformed to *ExternalTypes* having the same namespace as the resulting DSL, thus achieving identical functionality. In order to be able to correctly validate such an *ExternalType* attribute, a C# class with its name and namespace has to be referenced from within the DSL project. To achieve this, a basic *.cs* file with the name of the attribute and an empty class definition within the correct namespace is generated, one for each mapped *EDataType*. These files have to be put in the root folder of the target DSL project. More complex types (e.g. external java classes) are transformed to *DomainClasses* connected by a *DomainRelationship* using the type as a target *DomainRole*. Multiplicities of the newly generated relationship are transformed accordingly, having a 1:n relationship when the attribute value is a collection, and a 1:1 otherwise.

AS.R6: *EEnum* $\mapsto$ *DomainEnumeration* An *EEnum* is a special type of an *EClassifier*, containing multiple *ELiterals* for defining custom values that can be attributed to a class. The target *DomainEnumeration* also contains a list of *EnumerationLiterals*, thus making it possible to easily map each literal element. The only difference with regards to this rule in MSDKVS is that the value of a *EnumerationLiteral* must not contain a list of specific characters, including white spaces, commas and backslashes. Therefore, a renaming strategy has to be employed. Default literal values are transformed accordingly.

AS.R7: *Inheritance* As EMF supports multiple inheritance structures, concrete patterns have to be implemented to transform these features into semantically equivalent ones on the target side, explained in detail in Section 6.1.3.

5.2.2 Concrete Syntax

GCS.R0: *Viewpoint* $\mapsto$ *Diagram* *OwnedViewpoints* can be seen as the *EPackages* of a Sirius Viewpoint Specification Model file. One or multiple *OwnedViewpoints* can be defined inside a VSM's `Group` tag, and because of their containment trait they do not need to be flattened like multiple *EPackages*. Instead they are iterated while being transformed into MSDKVS' equivalent graphical concrete syntax elements. Each viewpoint can contain multiple *OwnedRepresentations*, and each representation contains a *DefaultLayer* with optional *AdditionalLayers* (see Section 3.1.4). The relevant concrete graphical syntax elements are contained within these layers. The resulting *Diagram* consists of mapping tags to all the different shapes created in the following steps, referenced by their `Monikers`.

Shapes

Regarding shapes in MSDKVS, it is important to note that also the default values for each attribute have to be considered while transforming. This is especially important since these default values often mean that the attributes do not have to be defined at all, thus the M2 transformator has to know which values are the default ones. These have been received through extensive testing and reading the available API documentations online. An example for such values is the `BorderSizeComputationExpression` of a shape mapping in EMF which maps to the `OutlineThickness` attribute of a shape in MSDKVS. In EMF, this value defaults to **1**, whereas in MSDKVS the default value is **0.03125**. In many cases, EMF provides simple default number values which are just multiplied on the target side. `DomainPaths` are used in MSDKVS for correctly mapping a shape entity to a domain entity. These domain paths are defined as XPath-like syntaxes and are built as follows: <RelationshipName.PropertyName/!Role>. To identify the target shape on the modeling canvas, a containment reference in addition to the source's property name and role has to be given.

GCS.R1: *NodeMapping* $\mapsto$ *GeometryShape* *NodeMappings* are transformed to *GeometryShapes*, visualizing *DomainClasses* through different geometries, like `Rectangles` or `Circles`. The comparison tables in Section 3.5 are divided into the various identified shape types after having investigated both frameworks. A *DomainClass* is then referenced accordingly by mapping the `ShapeMoniker` inside the *Diagram*.

GCS.R2: *NodeMapping* $\mapsto$ *ImageShape* *NodeMappings* that are attributed with custom images identified by their URL are transformed to *ImageShapes*. MSDKVS only allows the usage of bitmap files as image references, therefore the images have to be transformed as well to fit this requirement.

GCS.R3: *EdgeMapping* $\mapsto$ *Connector* Graphically visualizing *EReferences* is done via defining *EdgeMappings* inside the viewpoint specification model. These are mapped to *Connectors* in MSDKVS, offering many different styling options like `RoutingStyles`, `ArrowStyles` and `DashStyles` etc. Important to note is the positioning of the label on relationships. Sirius supports `BeginLabel`, `CenterLable` and `EndLabel` styles. In MSDKVS,

GCS.R4: *ContainerMapping* $\mapsto$ *CompartmentShape* Containment references, i.e., having a container with sub elements, are visualized in EMF by specifying *Contain-*

erMappings. These mappings can contain multiple *SubNodeMappings.* A mapping to MSDKVS' equivalent *CompartmentShapes* can be achieved while also considering the mappings for the contained nodes. MSDKVS requires *CompartmentShapes* to have a `CompartmentMap` defined, referencing a contained domain element via the containment reference name and a displaying attribute within the containment, supplied via the map's `PropertyDisplayed` attributed. If no *SubNodeMapping* has been defined, the resulting shape cannot be valid.

GCS.R5: $Label \mapsto DecoratorMap$ Each mapping in EMF can have various styles, depending on the target element, and each style has multiple labels, targeting different attributes and customizable parts of the elements. In MSDKVS, each shape has a list of *Decorators*, and each *Decorator* is mapped to a *DomainProperty* of the shaping entity, mapped via *DecoratorMaps* within the corresponding *ShapeMap.* Therefore, a mapping between these concepts is easily established as they are quite similar in theory.

GCS.R6: $SpecialShapeMapping$ For special shapes that are unique to each platform special workarounds and default shapes have to be declared. Regarding Sirius' special shapes, *BorderedNodes* inside containment references or even normal references, where a *NodeMapping* is the containing shape, can be mapped to *Port* shapes in MSDKVS. If additional shapes are found, they are transformed to available target shapes best resembling them.

GCS.R7: $ColorMapping$ Sirius in EMF lets users define colors through a drop down table of predefined `System Colors`, `User Fixed Colors` in separate user colors palettes, `Computed Colors` by dynamically computing RGB components, or as `Interpolated Colors`, that dynamically change the coloring of a referenced object through the definition of so-called `Color steps` [3]. Through these steps, a color can be changed through associating values via computation expressions. For this thesis' transformation approach only user fixed colors and system colors are taken into account. Most named system colors can be directly translated to MSDKVS' color scheme, although some exceptions have to be considered, e.g., Sirius' **dark_red** color can map to MSDKVS' **DarkRed** color by removing the underscore. If a corresponding color has not been found, background and border colors default to black, whereas label and filling colors default to white in MSDKVS.

GCS.R8: $ShapeInheritance$ As shape inheritance is a feature unique to MSDKVS, this transformation is trivial as it is not available in EMF.

GCS.R9: $ToolSections \mapsto ToolboxTab$ Sectioning of available tools to be able to interact with the modeling canvas in the platforms' runtime environments is possible in both frameworks. EMF offers the possibility to define a plethora of different tools, e.g., creation and deletion tools for relationships and classes, edit tools and more sophisticated tools that use queries to get the applicable elements or tools that propagate changes to other elements as well. MSDKVS only offers a subset of these possible interactions, restricting the transformator to only transform valid types of tools when transforming from EMF to MSDKVS. Each tool can have an icon associated with it in order to design custom toolbars. As with icon shapes, these images have to be converted to bitmap files in order to be usable in MSDKVS. In addition to converting the file to bitmap, another requirement in MSDKVS is for these icons to be of size 16x16 pixel. Thus, the image has to be resized accordingly.

5.3 MSDKVS to EMF

5.3.1 Abstract Syntax

AS.R0: *Language* $\mapsto$ *EPackage* As there are no multiple languages definable in one *DslDefinition.dsl* file, this step is trivial and creates only one *EPackage* element on the target side as well.

AS.R1: *DomainClass* $\mapsto$ *EClass* The most basic form of mapping when comparing MSDKVS and EMF because the class concept is de facto identical, except for the specifications mentioned later on regarding inheritance and *DomainRoles* used for targeting in *DomainRelationships*. `InheritanceModifiers` on MSDKVS (abstract, sealed, public) are transformed accordingly, whereas *public* is the default value, *abstract* directly translates to the abstract attribute of an *EClass* (i.e., *abstract="true"*). Descriptions supplied in MSDKVS are mapped to documentation tags on the transformed *EClass*.

AS.R2: *DomainRelationship* $\mapsto$ *EReference* *DomainRelationships* act as links between multiple types of domain model entities. A domain relationship always consists of source and target roles, represented by *DomainClasses* and a specific role they are given regarding the relationship definition. A *DomainRelationship* in MSDKVS can be mapped to an *EReference*, which is an *EStructuralFeature*, distinguishable by the `ecore:EReference` type. The name for the *EReference* is mapped from the source domain role from the domain relationship in MSDKVS, and the type is mapped to the target domain role, which contains a domain class moniker for retrieving the target domain class. The subsequent rules for this rule are used to transform containment references, multiplicities and bi-directional reference accordingly.

AS.R3: *DomainProperty* $\mapsto$ *EAttribute* For each *DomainProperty* either defined in a *DomainClass* or a *DomainRelationship*, a resulting *EStructuralFeature* with type `ecore:EAttribute` is generated. The resulting `EType` is mapped through a separate TypeMapper, which takes as input the name of the domain property type of MSDKVS, where either basic types like `Boolean`, `Int32` or `String` are mapped or more complex types like other domain classes, domain relationships and even external classes. If a `DefaultValue` has been set, it is transformed directly to the corresponding attribute in Ecore, which can be done for either simple types or complex types like *DomainEnumerations* as well.

AS.R4: *DomainRole* $\mapsto$ *EClass* A *DomainClass* can be referenced by multiple *DomainRoles* acting as target or source role player of a *DomainRelationship*. Every *DomainRole* always references, through monikers, the actual domain class and can be retrived through cross references in the XML data portion of the DSL. These are, e.g., used for mapping source and target (i.e., `EOpposite` attributes) of *EReferences*. Each *DomainRole* contains multiplicity values for the connected *DomainRelationship*.

AS.R5: *DataType* $\mapsto$ *DataType* As mentioned before, a `TypeMapper` is implemented that contains bidirectional mappings of the system type variants contained in both frameworks. MSDKVS differentiates between signed and unsigned integers, ranging from 16 to 64 bytes (i.e., short, int and long). `System/DateTime` is mapped to `EDate`, `System/Byte` is mapped to `EByte`, `System/Double` is mapped to `EDouble`, `System/String` is mapped to `EString`, `System/Boolean` is mapped to `EBoolean` and `System/Char` is mapped to `EChar`. More types are usable for specific use cases by default when creating a DSL in MSDKVS. These types are `NullableUsage` and `FieldCase`, which are mapped to `EBoolean` because of their possible truth values, as well as `System/Single` and `System/Guid`, which are typically in a string format and therefore mapped to the `EString` data type. When a system defined type is referenced, like a class that is either contained within a standard library of C# or a user defined class in source code, for code generation purposes and correct representation functions on the target EMF side their class names are used for creating additional *EClasses*, that are then referenced instead through additional *EReferences*, which results in an external code independent metamodel. If a transformation from this resulting metamodel back to MSDKVS is executed, these new classes would also result in additional *DomainClasses* as well, thus having eliminated the code specific definitions previously referenced through external classes.

AS.R6: *DomainEnumeration* $\mapsto$ *EEnum* A *DomainEnumeration* contains multiple *EnumerationLiterals*, each with its own `Name` and `Value` attributes. These components are transformed accordingly to *EEnums* and their *ELiterals*.

AS.R7: *Inheritance* The concept of Inheritance is trivial when transforming from MSDKVS to EMF, as both frameworks support the notion of having their classes inherit references and attributes from other classes. These inheritance references are signaled by the classes' property *BaseClass* in MSDKVS and *ESuperTypes* in EMF. Regarding *DomainRelationships* though, MSDKVS also allows Inheritance as well, making the transformation of inheritance structures based on relationships cumbersome and require special attention, resulting in additional structures on EMF as a *DomainRelationship* has to be interpreted as a class.

5.3.2 Concrete Syntax

GCS.R0: *Diagram* $\mapsto$ *Viewpoint* Regarding the concrete graphical syntax layout, the biggest difference between MSDKVS and EMF is that EMF needs additional Sirius components in order to visually represent the domain model entities on a canvas, whereas MSDKVS has an integrated graphical component to achieve this. Sirius, in reference to the Ecore metamodel definitions, has the possibility to define multiple areas of definitions that can intercommunicate and exchange information too. In MSDKVS, only one representation (i.e., viewpoint in EMF / Sirius) is available, which means, that only one mapping per domain entity to a shape definition is allowed. Therefore, when transforming the GCS of MSDKVS, only one viewpoint is generated on the Sirius side. A diagram can be defined with a variety of colors, either a filling background color or a text color that can be overridden by each shape modeled on the canvas.

Diagram Elements

GCS.R1: $GeometryShape \mapsto NodeMapping$ Geometry shapes reference *DomainClasses* only. The corresponding transformed graphical entity in Sirius is called a *NodeMapping*. Regarding appearances (e.g., geometries), styles (e.g., border style, font style) and layout (e.g., positioning) of the entity itself and also its domain properties, for which appearances can be stylized through decorator maps, rule **DIAG.R5** gives more insight on this manner.

GCS.R2: $ImageShape \mapsto NodeMapping$ As *ImageShapes* and *GeometryShapes* target *DomainClasses*, and there exist no explicit type of mapping associated with image shapes on EMF, the resulting object is also a *NodeMapping* entity. In EMF, these two types are distinguished by their type attribute. When a *NodeMapping* has the `WorkspaceImageDescription` style tag, it signals the interpreter that the referenced class is displayed as an icon on the canvas, thus transforming *ImageShapes* to mappings with this specific style.

GCS.R3: $Connector \mapsto EdgeMapping$ *Connectors*, as opposed to other shapes with their *ShapeMaps* containing further mapping information needed to establish a mapping to the concrete canvas objects, have their own *ConnectorMaps*. These maps connect a *Connector* and the targeted *DomainRelationship* via their moniker types. The shape object itself and its list of maps are used for transforming into an equivalent *EdgeMappings*, defining several styles including `RoutingStyles` and `ArrowStyles`. If a connector is targetting a domain relationship that is transformed to a class due to its attributability or inheritance mechanism, the connector has to be translated to a *NodeMapping* instead.

GCS.R4: $CompartmentShape \mapsto ContainerMapping$ *CompartmentShapes*, in addition to their mapping information in the form of a *CompartmentShapeMap*, which links the *DomainClass* and its containment reference to this shape, gives insight on the container and its sub element shapes, creating a *ContainerMapping* with multiple *SubNodeMappings* accordingly. Decorators are transformed to the mapped shape elements' styles as well.

GCS.R5: $DecoratorMap \mapsto Label$ Each attribute, which is mapped via a *DecoratorMap* to a corresponding decorator within a shape, is transformed to a *Label* with different style attributes to achieve visual similarity.

GCS.R6: $SpecialShapeMapping$ As for special shapes, that were not included in the previous steps, MSDKVS further offers the usage of *Port* and *Swimlane* shapes. As there is no equivalent shape for swimlanes in EMF, they are by default transformed to *ContainerMappings* as they can contain subelements. *Ports* are transformed to *BorderedNodes*, contained either within *NodeMappings* or *ContainerMappings*.

GCS.R7: *ColorMapping* MSDKVS uses three types of color definitions: Custom, Web and System. System colors have names that correspond to the objects in the Windows OS, like Scrollbar or WindowBackground. Web colors consist of colors that are identified by their uniquely standardized web names, which are used in web development. Custom color palettes can also be used, which are the same as in e.g., a Paint program, where the user can define them through a color picker. These are usable on all objects, where colors can be applied (e.g., diagram background color, line colors, text colors, border colors, etc.). A ColorMapper class acts as a middleware that takes the color used in MSDKVS as input and looks up the system color, to get the RGB values of a selected named color (i.e., web or system). These RGB values are then injected separately on the target EMF metamodel. A viewpoint specification model in Sirius can be supplemented with an additional section of user defined color palettes, defined by their red, green and blue values respectively and given a name by the designer. This functionality is used when transforming the MSDKVS colors to EMF. An example on how this looks inside the Sirius editor can be seen in Figure 6.8.

GCS.R8: *ShapeInheritance* Shape inheritance is a unique feature integrated into MSDKVS, where one shape (has to have the same type as the super shape, e.g., a GeometryShape can only inherit from another GeometryShape) inherits and has the ability to override different style attributes for mapped domain properties of the referenced entities from its super shape. This can be used for, e.g., distinguishing model entities based on their inheritance structure, which also implicates that shape inheritance, when used inside a DSL, goes hand in hand with class or relationship inheritances. Regarding the transformation approach and the mapping rule, all attributes and layout properties from the super shapes are first transformed and then the specific, possibly overridden and additional properties in the sub shapes are transformed to the target framework.

GCS.R9: *ToolboxTab* $\mapsto$ *ToolSections* A *Designer* section in MSDKVS can contain multiple *ToolboxTabs*, dividing the available tools into various sections for visually grouping tools together. Different types of tools exist, each targeting a different meta-metamodel concept, like *ElementTools* for *DomainClasses* and *ConnectionTools* for *DomainRelationships*. As such tools are only used for creating elements on the diagram canvas inside the modeling run-time, the resulting transformation thus only need to create the semantically equivalent *OwnedTools* types like `ContainerCreationDescription`, `EdgeCreationDescription` and `NodeCreationDescription`. Implicitly, to be able to delete the created elements as well, appropriate `DeleteElementDescription` tools are generated to establish interoperability.

CHAPTER 6

M2 Layer Transformation

6.1 Overview

The following section gives an overview on how the first part of the M3B transformation approach, the metamodel transformation, is structured in terms of the frameworks in question and their unique features. The mapping on the M3 layer, the meta-metamodel layer, and the implementation of the M2 layer transformation, where metamodels from one framework are transformed into metamodels corresponding to the meta-metamodel on the target framework by implementing the aforementioned transformation rulesets, are discussed.

Figure 6.1 shows the M3B sketch of all layers involved in the transformation for both frameworks, MSDKVS and EMF. On the left hand side, the MSDKVS column consists of the implicitly defined meta-metamodel, which the underlying DSLs conform to in terms of valid metamodeling entities and their relationships. The metamodels are serialized inside a XML based *.dsl* file. These files are used as input and output of the transformation, whereas when transforming from left to right, i.e., from MSDKVS to EMF, only one of these files is needed for transforming on the M2 layer of the metamodeling stack. The transformator, as can be seen in the figure, is divided into M2 and M1 transformation explicitly. The M2 Transformator is written in C# and deserializes the incoming files into corresponding data structures that afterwards can be used for iteration. Abstract and concrete syntax elements represented as XML tags inside these input files are examined and the mapping rules, based on the M3 concepts of the meta-metamodels of both frameworks mentioned in Chapter 5, are applied sequentially to transform the source metamodel into an equivalent metamodel on the target side.

The right hand side consists of files and definitions for the EMF framework. Here, the meta-metamodel is publicly available. An Ecore metamodel has its graphical concrete and abstract syntax contained within separate files. This applies also to the M1 level, the

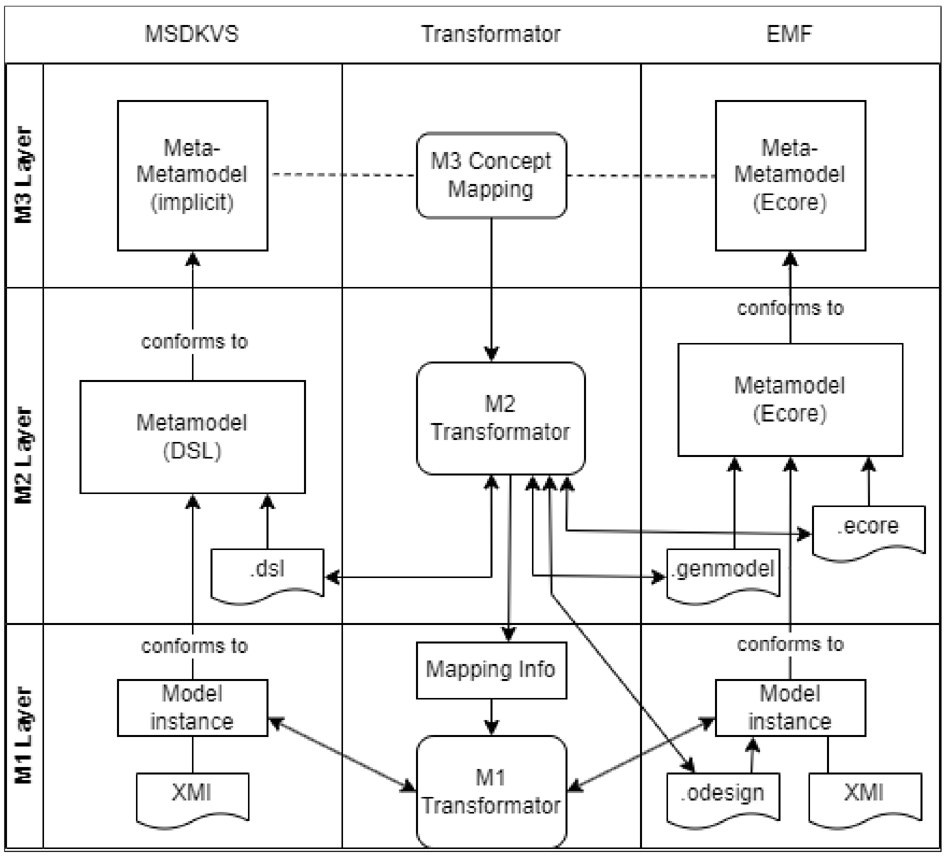


Figure 6.1: Sketch of the M3B transformation approach between MSDKVS and EMF

model layer. MSDKVS offers graphical manipulation of models out of the box, in EMF, the Sirius framework is used in order to design models and metamodels on a graphical canvas. To create a runtime instance of a metamodel, a *.genmodel* file can be used for generating the necessary projects and model code. In addition to that, an *.odesign* file is then used to visualize the models and their entities in the runtime instances of Eclipse. As a result, when a metamodel is transformed from MSDKVS to EMF, these three files are generated by default. When transforming from EMF to MSDKVS, only one file is needed in total, and the *.genmodel* and the *.odesign* files can be submitted optionally as additional input in order to transform the graphical notation of the metamodel as well.

6.1.1 M3 Layer Mapping

For being able to map each structural element that can be used inside a metamodeling framework, the meta-metamodel they conform to has to be identified. As opposed to the environment of MSDKVS, where the meta-metamodel on which metamodels are based is given only implicitly in the available documentation online [35] and in [17], an explicit definition of the Ecore meta-metamodel for metamodels in EMF is given in various sources, e.g., in [10, 29, 31], to name a few. Figure 6.2 shows an excerpt of this meta-metamodel.

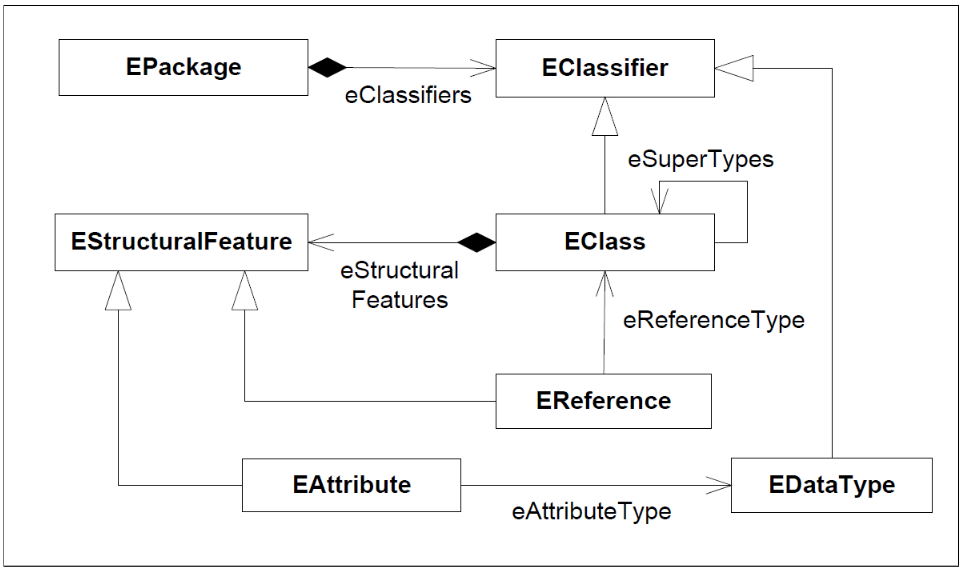


Figure 6.2: Part of the Ecore meta-metamodel with its main concepts, from [31]

As mentioned before, the meta-metamodel for MSDKVS is created by defining a class hierarchy in the XML to XML metamodel transformator written in C# and afterwards extracting this class diagram to get a visual representation of the underlying structure the metamodel concepts conform to. After having implemented and included all possible values, having examined available MSDKVS projects on different source code repositories, the meta-metamodel depicted as a class diagram in Appendix D could be extracted.

6.1.2 M2 Layer Transformation

The M2 layer transformation is implemented in C# as a .NET console application, which integrates the XML schema of both frameworks' serialization format for metamodels as a data structure, which implicitly produces a detailed and complete representation of the MSDKVS meta-metamodel. As an input, the files needed for the transformation are loaded and serialized into this data structure. Depending on the file endings, folder or zipped file contents the transformator decides what direction of transformation and rulesets should be traversed, thus enabling the resulting program to execute both directions of the transformator.

Based on the rulesets for abstract and concrete syntax elements defined in Chapter 5, the transformator executes several code snippets conforming to those rules in a sequential manner, so that the investigation of the set of functions is of a clear structure and the output of the transformator is ordered accordingly such that elements needed for, e.g., transforming reference relationships are available on the transformed data to be used as a target.

Additionally, some main criteria have to be fulfilled in order to properly evaluate the outcome and compare the result to the original metmamodel, i.e., the files have to be correctly opened and editable in the target framework, the validation of the transformed metamodel has to be free of errors that keep the editor code from being built, the metamodels have to be correctly displayed as class diagrams or in form of a tree structure, the generator code should successfully generate an executable modeling instance, and the editing of models should be similar between source and target framework regarding their tool palettes, contextual actions available on the model entities, appearances of the model nodes, deletion propagation of containment references and the ability to edit inherited properties in children entities, just to name a few. These requirements are discussed in Chapter 7 in detail.

In the following sections, challenges faced during the realization of the M2 Transformator are discussed and detailed steps, if available, on how these obstacles were solved are provided, grouped by the transformation directions.

6.1.3 EMF2MSDKVS

Nested EPackage Flattening. We recognized different styles of EPackage definitions in publicly available EMF metamodels (see Table 3.1 in the Grouping row). Ecore metamodels can either have one or multiple EPackages defined, while EPackages may also have ESubPackages. Therefore, as MSDKVS usually only has one equivalent language definition, these EPackage contents are flattened and merged into one global EPackage before executing the transformation. Naming conventions and avoiding name clashes are transformed accordingly.

Entity Name Clashes. Detecting and resolving name clashes are essential when realizing metamodeling platform interoperability [9]. Different naming strategies to avoid possible name clashes, e.g., across multiple ESubPackages, are executed. For domain relationships, the MSDKVS names are changed as follows:
<sourceEClass.name>_<EReference.name>_<targetEClass.name>.
Name clashes on domain classes (if there are any), are resolved by mapping the EPackages' nsPrefix attribute to the DomainClass' Namespace attribute.

Multiple Inheritance. As EMF, in contrast to MSDKVS, supports multiple inheritance, a transformation of multiple inheritance structures into equivalent single inheritance structures is necessary. We adapted the *Expansion Strategy* pattern proposed by Crespo et al. [18] to translate the complex structures of multiple ESuperTypes in EMF into equivalent single BaseClass references in MSDKVS without information loss (see Fig. 6.3). Important to note is that also EReferences that target a super class have to be duplicated to the newly created domain classes as Domain Relationships in MSDKVS. In addition to the abstract syntax duplicates,

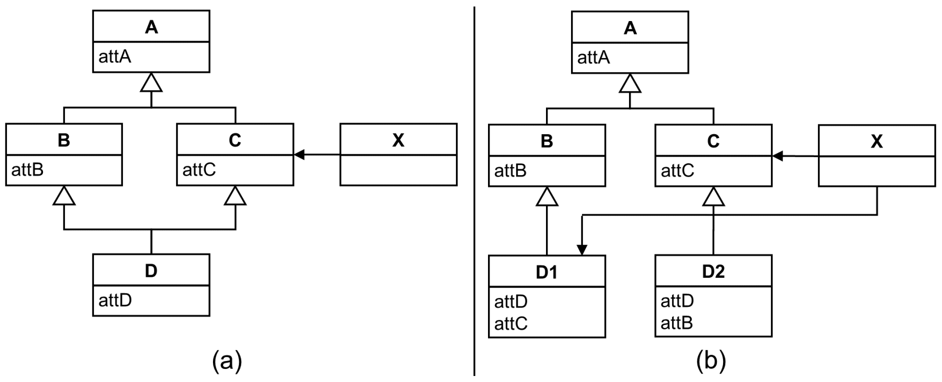


Figure 6.3: Adapted *Expansion Strategy* [18]: (a) multiple inheritance in EMF; (b) transformed single inheritance in MSDKVS

this affects the transformation of all types of graphical concrete syntax mappings from Sirius as well, especially the Creation Tools inside the modeling editor if such a duplicated class has been used as the tools' target. Concerning shapes, only the shape maps have to be duplicated and then renamed in most cases, as the decorators for styling the referenced entity's attributes are most likely to be contained inside the duplicated classes as well. If that is not the case, their shapes have to be cloned as well and used inside the tool mappings.

Root Element Pattern Matching. MSDKVS metamodels require a root element that is mapped to the diagram shape. This diagram shape provides the modeling canvas in the runtime instances of a domain model. As per API requirement, this selected root element has to be the source domain role of domain relationships marked as containment relationships, where the targets are all domain classes that are neither part of an existing containment relationship (e.g., children of compartments) nor should target any base classes they would inherit from. When transforming an Ecore metamodel into MSDKVS, existing EClasses are matched against these criteria. If such an EClass can be found, this EClass is transformed to be the root diagram element in MSDKVS. If no EClass is suitable, then an additional default domain class is generated that acts as the diagram's root element. This is especially required in order to achieve the transformation functionality on the M1 layer, as in MSDKVS only one class can be the model's root class, whereas creating a model in EMF lets the modeler choose which class should act as the model's root.

Icon Mapping. Sirius supports the definition of icon styles on different node mappings, referencing workspace images in various image file formats. For MSDKVS, a requirement to attribute a model entity with icons is that the images have to be in Bitmap format. Therefore, library calls to convert these files to the required format on the target platform are employed in the M2 transformation. Tooling icons have the additional requirement of having 16 pixels as width and height each.

Reserved Keywords. As MSDKVS operates on C#, the type of reserved keywords that are not allowed, e.g., used as the name of a class or attribute differs from its counterpart. During the evaluation step later described in Chapter 9, some Ecore metamodels contained attribute, reference or class names that proved invalid when the transformed DSL was opened inside the MSDKVS environment. Thus, the employed naming strategy prepends an underscore to a name when such a reserved keyword is being found.

***DomainEnumeration* Literal Validity.** When transforming *EEnum* entities with their literal values, special attention has to be given to the fact that MSDKVS does not allow a variety of special characters like comma, backslash or white spaces in their *EnumerationLiterals*, as was mentioned previously in Section 5.2.1. Similar to the name clash problem with regards to *DomainRelationships*, a naming strategy has to be defined when such special characters occur during the M2 transformation step. Each invalid character is transformed to an underscore, one of the only special characters not reserved by the language's API. A mapping file generated on top of the M2 transformation contains mapping information for each literal value for correct lookup in the M1 transformation step.

GUID and ID Handling. Each element, be it either graphical concrete or abstract syntax, needs an Id property, which contains a generated GUID that uniquely identifies said element. When transforming from EMF to MSDVKS, these GUIDs have to be explicity generated when the elements are created. Mappings inside the *DslDefinition.dsl* file cross reference these IDs and thus providing the linking functionality in the Visual Studio IDE and the correct code generation capabilities. Therefore, when transforming, a lookup of already created elements when the XML tree is filled is done with the help of these Id attributes. As a result, *EAttributes* in EMF which are named **Id** cannot be transformed trivially, because in MSDKVS this reserved field is a necessary property of every element. Thus, **Id** names are always transformed lowercase and the applied naming conventions can be looked up in the generated mapping information file.

Duplicate *DomainRole* Names. When transforming *EReferences* to *DomainRelationships*, no two *DomainRoles*, either used as Source or Target, can have the same `PropertyName` when their other end of the relationship is the same. Example: `ClassA` has relationships to `ClassB` and `ClassC`. If the transformation results in, e.g., having a *DomainRelationship* from ClassA to ClassB with its target role `PropertyName` being "Target" and the same applies for relationship ClassA to ClassC, then naming conventions have to be executed on the second relationship target role. The naming strategy employed updates a counter variable for how often the same target or source base names has been used, applies it to the newly create role and increments it. In the example above this results in relationship ClassA to ClassC having its target domain role renamed to `Target_1`.

Duplicate *DomainRelationship* Names. Same as for *DomainRoles*, no two *DomainRelationships* must have the same name. The similar renaming conventions apply, counting the number of already recognized equal names and adding the updated count to the newly created relationship.

6.1.4 MSDKVS2EMF

Relationship Roles. Domain Relationships in MSDKVS differ from their required representation on the target EMF side in so far, that the source and the target entities of said relationships are referencing the corresponding domain classes through monikers. Source and target domain roles can have different names attributed to them compared to their actual classes used for creating the domain relationship. This construct has to be considered when transforming from EMF to MSDKVS too, as for every EReference at least one role has to be created in MSDKVS. Domain classes are then referenced through moniker types by their unique names. When transforming from MSDKVS to EMF, the transformator has to look up the source and the target domain classes and transform these DomainClasses into the EReferences' eTypes and eOpposites accordingly.

Attributable Relationships. In MSDKVS not only classes but also relationships can have attributes. As has already been mentioned in [13], the behavior can be implemented similarly here, meaning that domain relationships with attributes attached to them are mapped to classes that are referenced from both transformed domain classes, leading to additional EClass and EReference entities on the target EMF side. Multiplicities are transformed accordingly to maintain the original behavior. An example for the implemented behavior and its results are given in Figure 6.12. Furthermore, if a Connector is attached to this Domain Relationship, it has to be transformed to a NodeMapping with two EdgeMappings in Sirius as well. When tooling is involved, its ConnectionTool has to be transformed to a NodeCreation tool with two EdgeCreation tools in order to be able to create these special structures in EMF. Although this does function for simple references between two classes, containment references have to be handled differently, as they are represented via CompartmentShapes, whereas the containment relationship is used to show the inner structure of the containment. Thus, if an embedding domain relationship contains attributes and has to be transformed to an EReference in EMF, the relationship's attributes are attributed to the targeted, inner EClasses.

Shape Inheritance. MSDKVS allows inheritance on the graphical representation of classes and relationships. Therefore, the M2 Transformator has to check possible inherited shape classes and transform them accordingly. Especially, when a base shape targets the BaseClass of a DomainClass, and the inherited shapes target the inherited classes of said BaseClass, it has to be verified if either the base or the sub shape is target of a tool, as in EMF with Sirius this would result in two shapes for the same entity on the canvas.

Implicit Modeling Tool Capabilities. MSDKVS supports only the definition of element creation tools on domain classes and domain relationships, while other tooling capabilities that can be explicitly defined in Sirius are inherently available on MSDKVS' modeling canvas. To achieve an equivalent experience, the number of tools on EMF thus must be typically higher because the M2 Transformator has to generate these additional tools for every Node or Edge Creation Tool defined in MSDKVS.

Color Naming. MSDKVS supports a variety of of colors for graphical properties (e.g., FillColor, TextColor, and BackgroundColor). Sirius only supports a small subset of these named colors, e.g., standardized system colors like white, black, and green. To be able to transform said colors from MSDKVS into equivalent colors in EMF, the M2 Transformator looks up the composing red, green, and blue color values for MSDKVS' named colors and transforms them to Custom User Palettes used by Sirius which can be named by the designer.

6.2 M2 Transformation Examples

In this section, the previously defined mapping and transformation implementation on layers M3 and M2 of the metamodeling stack are used on the example metamodels defined in Chapter 3.

Besides the analysis in Chapter 7 for the feasibility and usability of this approach through exhaustive evaluation, this section aims at guiding through the execution of the M2 transformator by using these metamodels as a template and for better understanding the approach in general. After the familytree example, additional use cases are listed to further explain some of the frameworks' more unique features mentioned above.

6.2.1 MSDKVS2EMF

The *DslDefinition.dsl* file containing the syntax defined in MSDKVS is set as input for the M2 transformator. First, Figure 6.4 shows a basic DSL created in MSDKVS depicting a familytree hierarchy with an additional residence relationship to the town class, whose elements are contained within countries. Figure 6.5 shows the output files that can be used in EMF in addition to the generated log files with information on how the transformation was executed and what syntax elements have been created.

The .ecore metamodel file and the .genmodel file are then copied to a modeling project in EMF that corresponds to the name of the generated output, i.e., familytree. To check if the files were transformed correctly they are simply opened inside the editor. Both files display a tree editor. By right-clicking the FamilyTree package inside the *.ecore* view, a new representation based on sirius class diagrams can be generated, as well as validating the metamodel's structure becomes possible. When opening the *.genmodel* file, the package FamilyTree and its properties regarding code generation in EMF are displayed, By right-clicking this root element or the package, the available generators are listed inside the context menu and can be executed sequentially, or, if available, "Generate

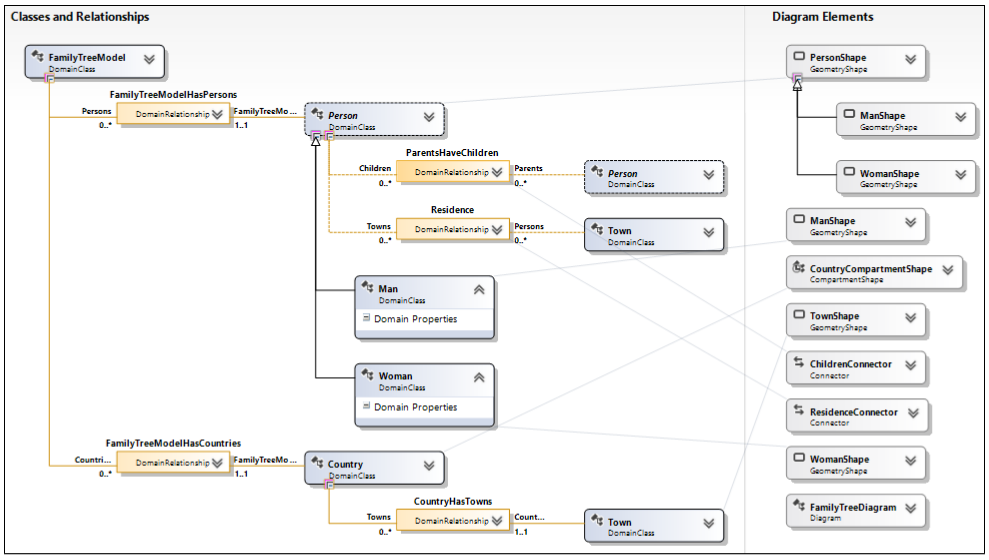
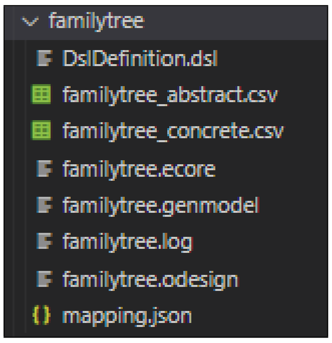


Figure 6.4: Basic familytree DSL designed in MSDKVS used as input for the M2 transformator



familytree
DslDefinition.dsl
familytree_abstract.csv
familytree_concrete.csv
familytree.ecore
familytree.genmodel
familytree.log
familytree.odesign
mapping.json

Figure 6.5: Output files for M2 Transformation MSDKVS2EMF for familytree example DSL

All" executes these steps in one go. If the transformation has created no error-prone code thus far the validation of the metamodel and the generation of code are deemed valid.

Looking at the transformed *.ecore* file (see Figure 6.7), all nodes have been transformed correctly as well as the inheritance relationships for the Person class, multiplicities, containment references, abstract classes and bidirectional references. To start the runtime instance of the modeling editor in Eclipse, the generated familytree.editor project has to be executed and selected to run as an eclipse application. An additional Eclipse process is started where a modeling project can be created to define a model based on the transformed metamodel with its file endings and its VSMs (if available).

Now, in order to be able to use the generated *.odesign* file, a VSP (Viewpoint Specification

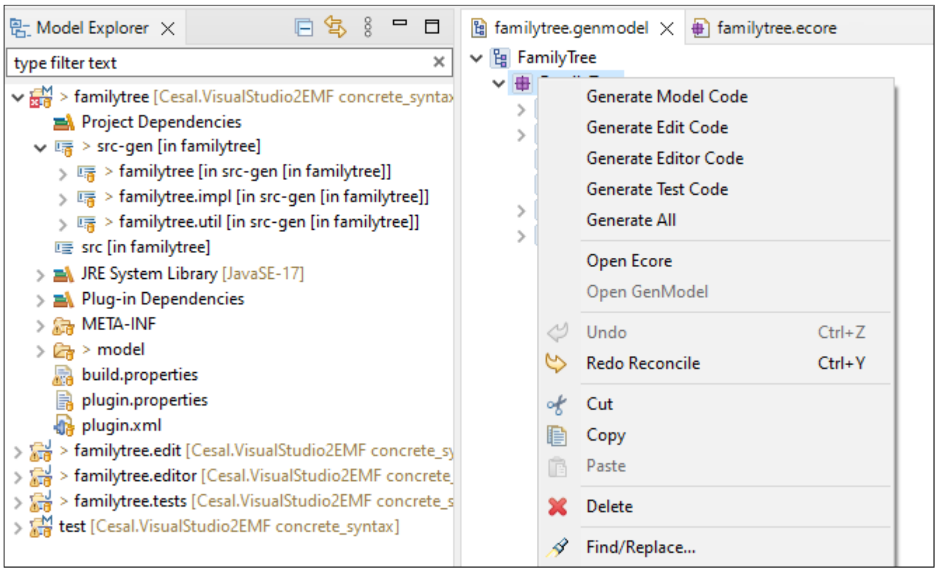

Model Explorer
type filter text
> familytree [Cesal.VisualStudio2EMF concrete_synta
Project Dependencies
> src-gen [in familytree]
> familytree [in src-gen [in familytree]]
> familytree.impl [in src-gen [in familytree]]
> familytree.util [in src-gen [in familytree]]
src [in familytree]
JRE System Library [JavaSE-17]
Plug-in Dependencies
META-INF
> model
build.properties
plugin.properties
plugin.xml
> familytree.edit [Cesal.VisualStudio2EMF concrete_s
> familytree.editor [Cesal.VisualStudio2EMF concrete
> familytree.tests [Cesal.VisualStudio2EMF concrete_s
test [Cesal.VisualStudio2EMF concrete_syntax]
familytree.genmodel
familytree.ecore
FamilyTree
Generate Model Code
Generate Edit Code
Generate Editor Code
Generate Test Code
Generate All
Open Ecore
Open GenModel
Undo Ctrl+Z
Redo Reconcile Ctrl+Y
Cut
Copy
Paste
Delete
Find/Replace...

Figure 6.6: Generated .genmodel structure with context menu and resulting generator projects in EMF

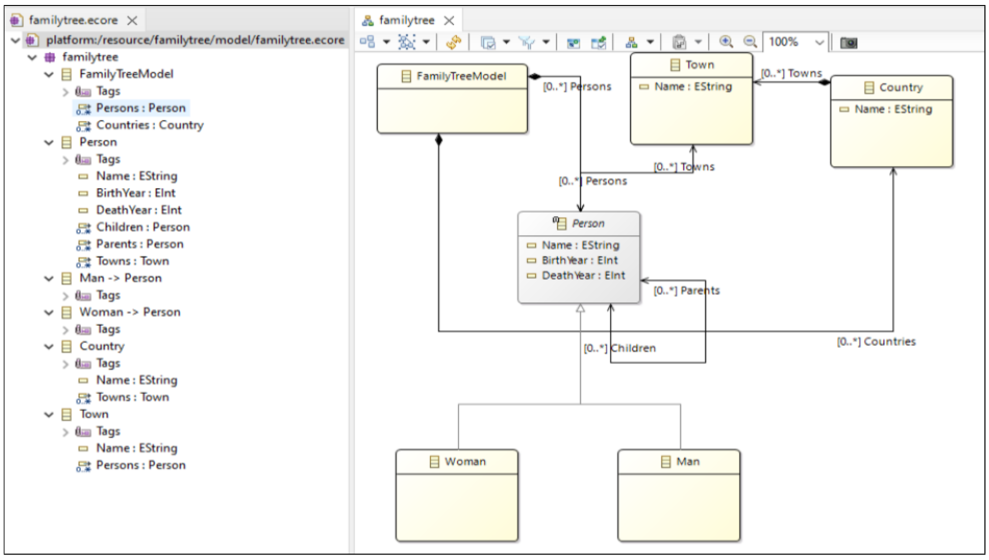
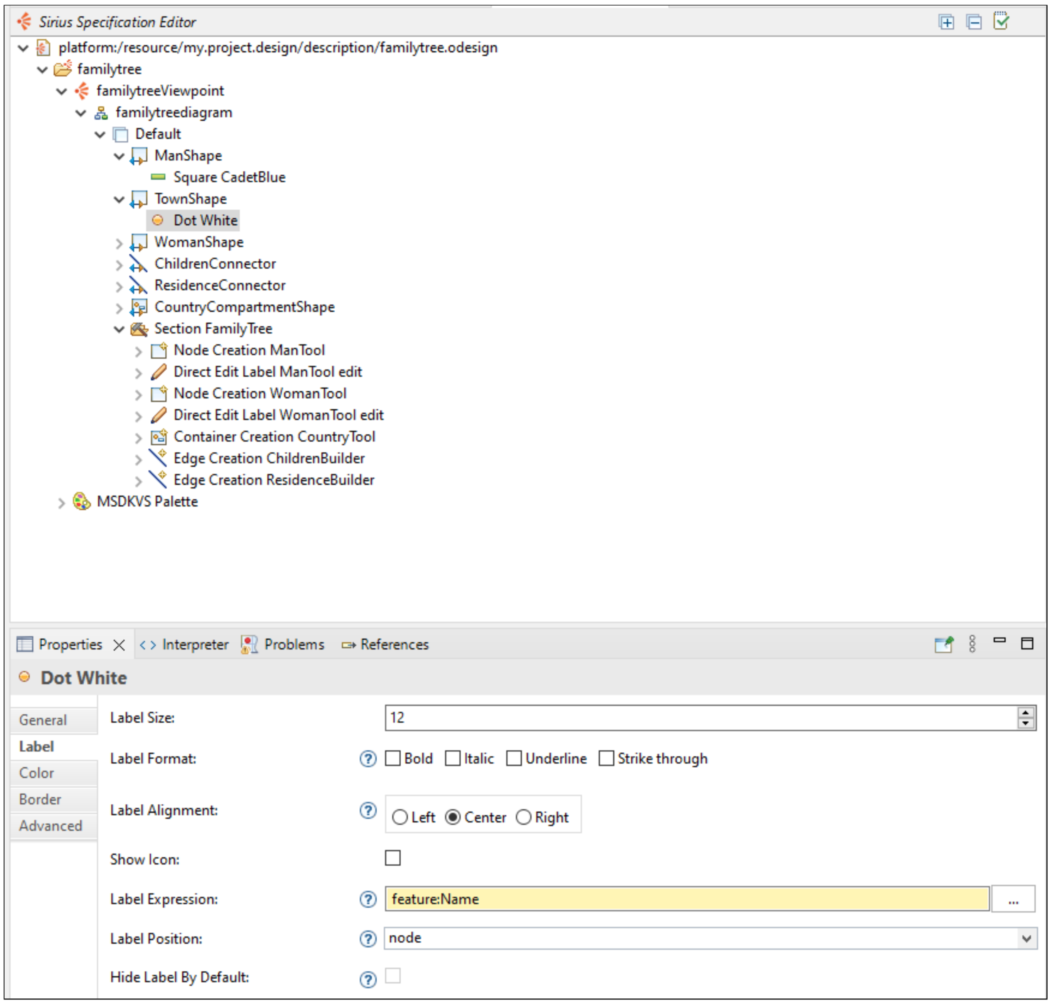


Figure 6.7: Transformed Ecore metamodel

Project) has to be created where this file is then copied into and referenced from the created model file to adapt the transformed graphical appearances and layout (i.e., transformed concrete syntax). The tree structure of the *.odesign* file is shown in Figure 6.8, having some elements expaned to show the results.

To get a glimpse of what is happening inside the code and how it is structured when



Sirius Specification Editor

```
platform:/resource/my.project.design/description/familytree.odesign
  familytree
    familytreeViewpoint
      familytreediagram
        Default
          ManShape
            Square CadetBlue
          TownShape
            Dot White
          WomanShape
          ChildrenConnector
          ResidenceConnector
          CountryCompartmentShape
          Section FamilyTree
            Node Creation ManTool
            Direct Edit Label ManTool edit
            Node Creation WomanTool
            Direct Edit Label WomanTool edit
            Container Creation CountryTool
            Edge Creation ChildrenBuilder
            Edge Creation ResidenceBuilder
    MSDKVS Palette
```

Properties | Interpreter | Problems | References

Dot White

General | **Label** | Color | Border | Advanced

Label Size:	12
Label Format:	☐ Bold ☐ Italic ☐ Underline ☐ Strike through
Label Alignment:	○ Left ● Center ○ Right
Show Icon:	☐
Label Expression:	feature:Name
Label Position:	node
Hide Label By Default:	☐

Figure 6.8: Generated Sirius VSM containing the concrete syntax on M1 layer

consulting the different code snippets that emerged from the rule mapping requirements, following listing 1 shows the transformation code for the basic transformation of a *DomainClass* to a target *EClass*.

For getting a quick look on the involved model editors in both frameworks, Figures 6.9 and 6.11 depict run-time editor instances of the source and target metamodels of the familytree transformation example.

The model editor, as seen in Figure 6.9, situated on the M1 layer of the metamodeling stack, which is opened in a separate Visual Studio runtime instance, is similar in its layout to the M2 representation for MSDKVS. On the left hand side the toolbox contains tools defined in the Toolbox Tab region on M2, the modeling canvas in the middle is used for adding, removing and editing models available through creation tools from the model toolbox, the property editor window when an entity has been selected on the bottom

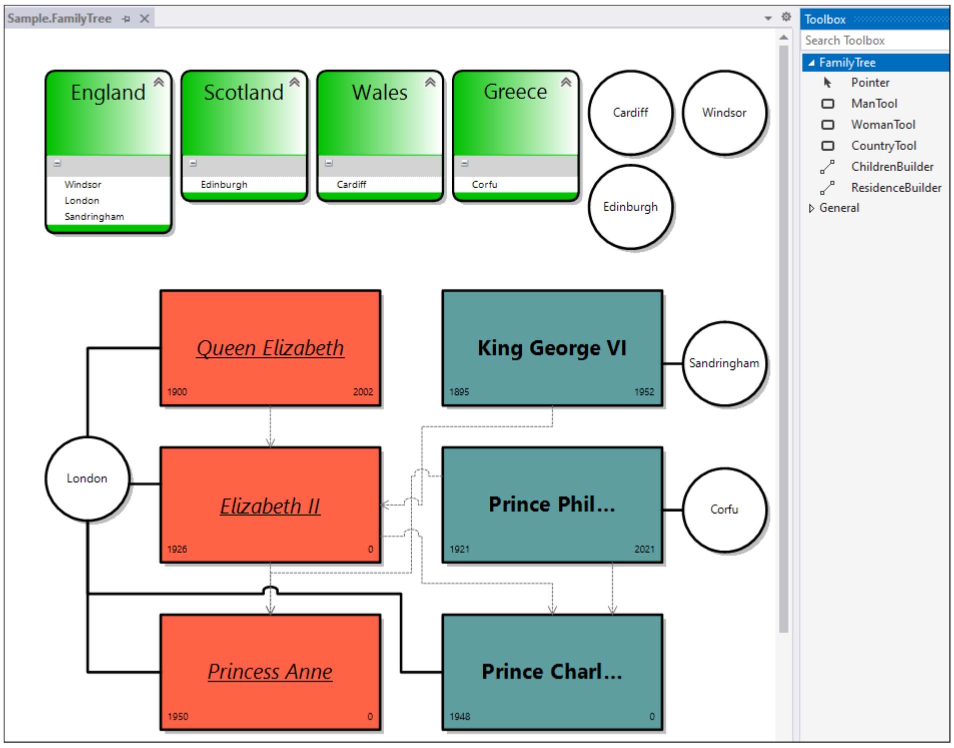
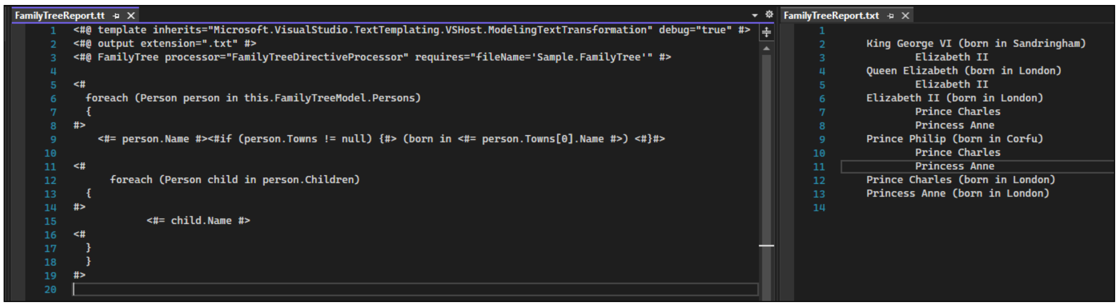


Figure 6.9: Snapshot of a model based on the previously defined metamodel in MSDKVS, modeled after the partial family tree of the royal House of Windsor [15]

right shows the available and configurable properties.

Regarding the modeling canvas, the Man and Woman Entities, which inherit their domain properties and possible domain relationships from the Person Entity, are now modelled with their specific shapes. The compartment shapes, which are used for displaying Country entities and their inner lists of Town entities, are listed as green, gradient rounded rectangle shapes in the top right of the canvas area. In each of these compartments, a list of referenced towns is displayed. For adding reference relationships between Town entities and other model entities, these towns are also represented through separate shape objects when being added to the list. These Town entities are displayed as white circles, with a black straight border line and their name centered, with a bigger font size than the Man and Woman entities. Each Person entity references their town of birth / residence relationships through connectors to those separate town shapes. The connectors can be created by using the `ResidenceBuilder` tool in the FamilyTree toolbox and by selecting one entity type of Person and Town each. Besides these residence relationships, the children relationships are also available and visible in Figure 6.9. To distinguish these reference links, each connector was attributed with a different line style on the M2 layer. As mentioned before, MSDKVS comes with additional functionality in form of T4 templates used for text or code generation. An example of a T4 template that uses our model and the resulting text file are given in Figure 6.10


FamilyTreeReport.tt

```
<#@ template inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation" debug="true" #>
<#@ output extension=".txt" #>
<#@ FamilyTree processor="FamilyTreeDirectiveProcessor" requires="fileName='Sample.FamilyTree'" #>

<#
  foreach (Person person in this.FamilyTreeModel.Persons)
  {
#>
    <#= person.Name #><#if (person.Towns != null) {#> (born in <#= person.Towns[0].Name #>) <#}#>

<#
      foreach (Person child in person.Children)
  {
#>
            <#= child.Name #>
<#
  }
  }
#>
```

FamilyTreeReport.txt

```

    King George VI (born in Sandringham)
            Elizabeth II
    Queen Elizabeth (born in London)
            Elizabeth II
    Elizabeth II (born in London)
            Prince Charles
            Princess Anne
    Prince Philip (born in Corfu)
            Prince Charles
            Princess Anne
    Prince Charles (born in London)
    Princess Anne (born in London)
```

Figure 6.10: T4 template and text file result for displaying Person entities, their birth places and their children indented

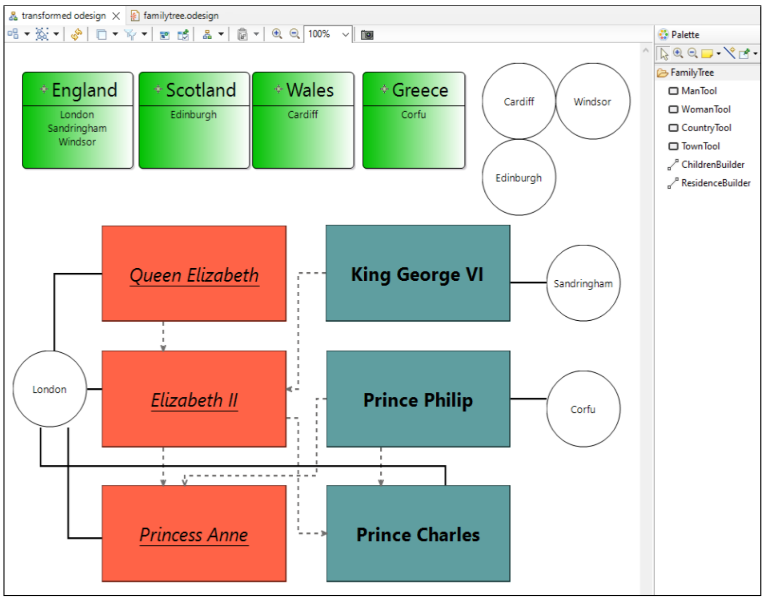


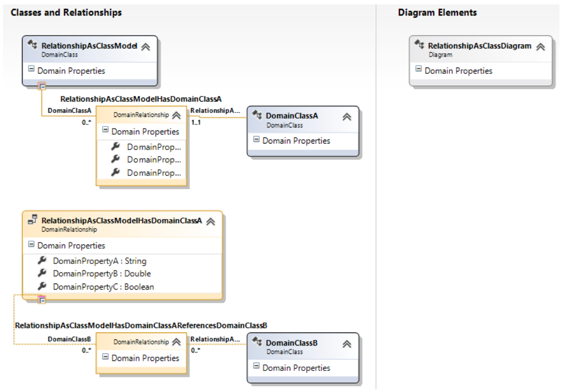
Figure 6.11: Manually created model in EMF with transformation results from the MSDKVS familytree DSL

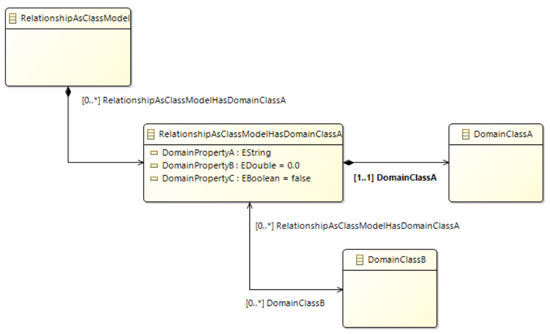
Special Use case #1 - Shape Inheritance

The transformation demonstrated previously serves as an example for this first explicitly presented use case, where MSDKVS' shape inheritance is being applied. `ManShape` and `WomanShape` inherit their shape properties from `PersonShape` 6.4, overwriting its *DecoratorMaps* in the process, thus resulting in different text label styles for the *DomainProperty* `name` 6.9 and 6.11.

Special Usecase #2 - DomainRelationship as DomainClass and with DomainProperties

Figure 6.12 shows source DSL and target Ecore metamodel of a transformation targeting a specific example where a *DomainRelationship* is defined as a *DomainClass*, i.e., it can be used as a *Source* or *Target* role inside another domain relationship. Several different *DomainProperties* have been added to this relationship.




RelationshipAsClassModel
[0..*] RelationshipAsClassModelHasDomainClassA
RelationshipAsClassModelHasDomainClassA
DomainPropertyA : EString
DomainPropertyB : EDouble = 0.0
DomainPropertyC : EBoolean = false
[1..1] DomainClassA
DomainClassA
[0..*] RelationshipAsClassModelHasDomainClassA
DomainClassB
[0..*] DomainClassB

(a) MSDVKS DSL with *DomainRelationship* used as a class and with several *DomainProperties*

(b) Transformed Ecore Metamodel

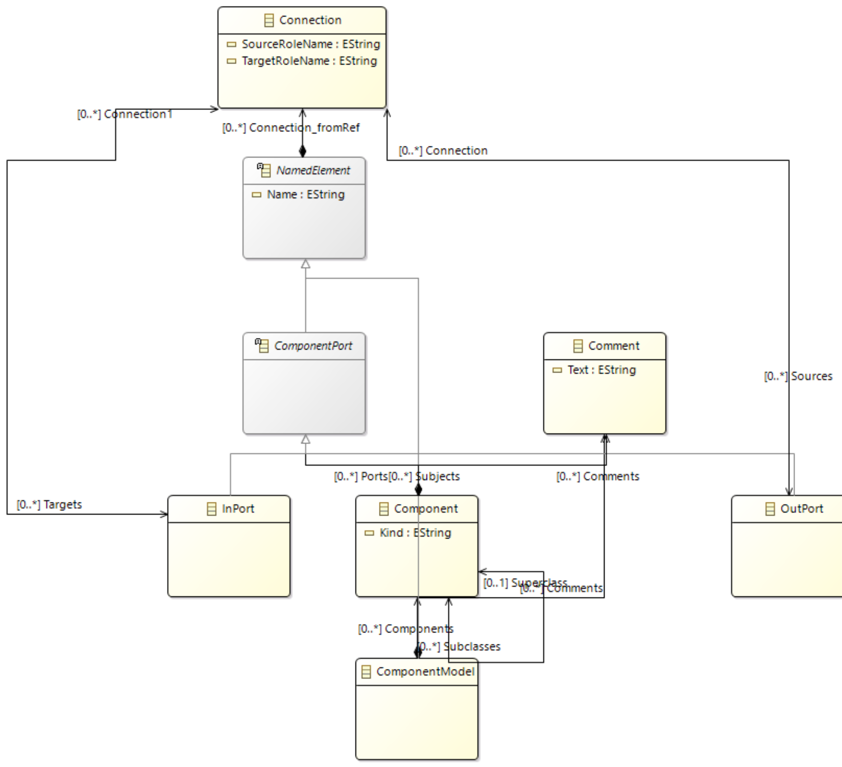
Figure 6.12: Usecase #2 - DomainRelationship used as a DomainClass with DomainProperties in a DSL (left) vs. transformed result in EMF (right)

Special Usecase #3 - Port to BorderedNodeMapping Transformation

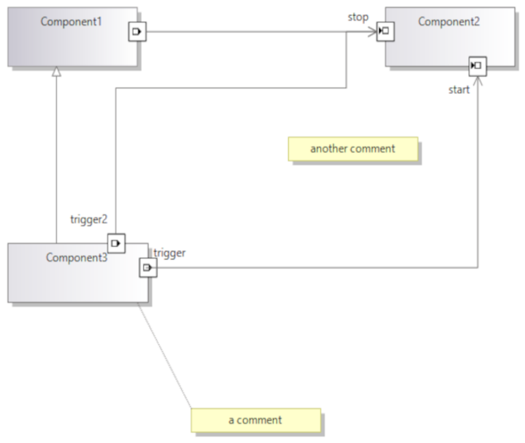
One of Visual Studio's default templates when creating a DSL project uses port shapes that can be attached to other shapes in order to decorate and create more complex shape structures and define specific visual points (i.e., ports) of a shape that can be the source or target of other relationships. The following Figure 6.13 shows how visually equivalent such Port shapes can look like when transformed to Sirius' equivalent BorderedNodeMapping, contained either within a NodeMapping or a ContainerMapping object. This usecase also demonstrates the combination of a domain relationship with attributes, which is transformed to an EClass and therefore a separate NodeMapping, and port shapes, thus resulting in an additional default shape between the connecting link of an incoming and outgoing port entity. The subfigure **a** depicts the transformed Ecore metamodel result of the port example template from Visual Studio.

6.2.2 EMF2MSDKVS

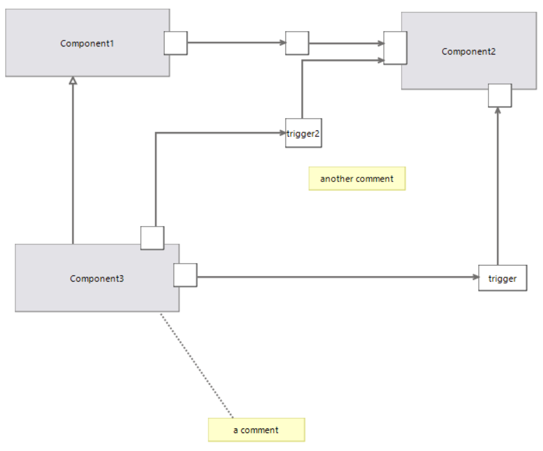
When transforming from EMF to MSDKVS, the *.ecore* metamodel file is required to execute the transformation, the *.genmodel* and *.odesign* files are optional. If provided, shapes and toolbar behavior among other concrete syntax elements will be transformed accordingly.



(a) Transformed Ecore metamodel from Port DSL template in MSDKVS



(b) MSDKVS model using ports



(c) Equivalent Ecore model using BorderedNodeMappings

Figure 6.13: Usecase #3 - Model using Port shapes in MSDKVS (left) vs. transformed result in EMF with BorderedNodeMappings (right)

Special Usecase #4 - Multiple Inheritance

Figure 6.14 and 6.15 show example transformations on the M2 level where in the former, an Ecore metamodel containing a multiple inheritance structure has been defined, the

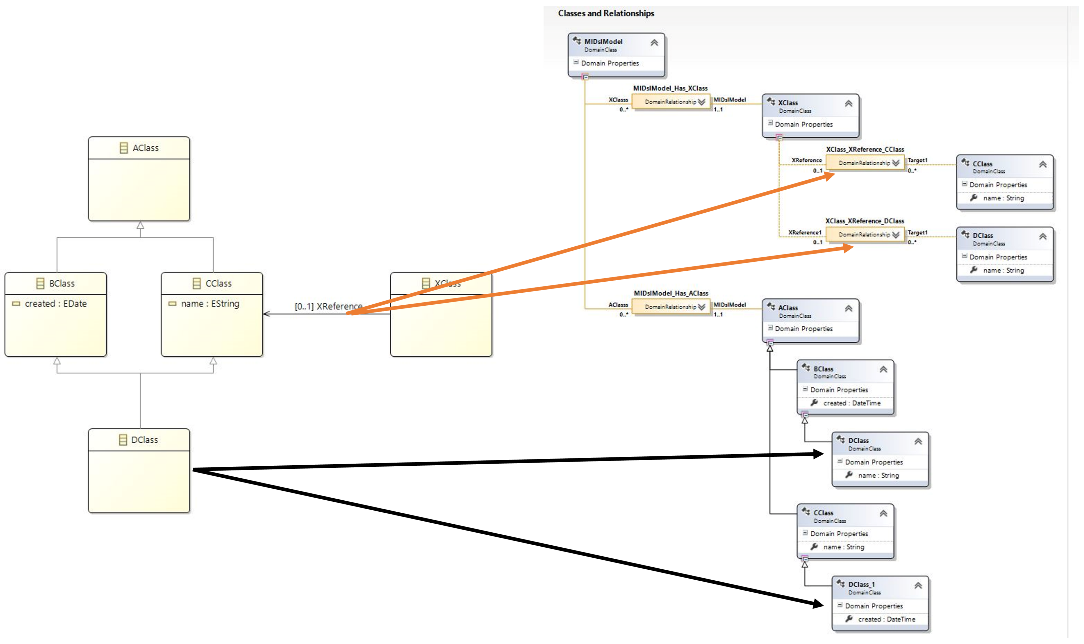


Figure 6.14: Usecase #4.1 - Multiple Inheritance in Ecore metamodel (left) vs. transformed result in MSDKVS (right)

latter containing multiple `superType` references inside one EClass. These metamodels are then being transformed to corresponding DSLs in MSDKVS, having applied the *Expansion Strategy* pattern mentioned in Section 6.1.3 respectively.

In Figure 6.14, the *DClass* entity uses multiple inheritance references to both *BClass* and *CClass*. The *XClass* references the *CClass*. This usecase is inspired by the Adapted Expansion Strategy example shown in Figure 6.3. On the right hand side, we see the transformed result in MSDKVS, whereas the *DClass* was duplicated, resulting in *DClass* with *BClass* as the base class, and *DClass_1* inheriting from *CClass* (black arrows). Thus, the *XReference* relationship is also duplicated (orange arrows) to connect *CClass* and *DClass* respectively, as the *DClass* loses its inherited features of *CClass*.

Figure 6.15 shows an example of multiple inheritance as well, this time containing a class having three supertype relationships and the transformed result. The *Fan* class inherits features from the *FlowElement*, *Powered* and *Named* classes. This results in two additional classes on MSDKVS side, namely *Fan_1* and *Fan_2* on top of the transformed *Fan* class (black arrows). The references from *System* to *FlowElement* are also duplicated for the additionally created subclasses (orange arrows).

Special Usecase #5 - EDataType and EEnum Transformation

As the special types of *EClasses*, namely *EDataTypes* and *EEnums*, were not included in the previous transformation example of the familytree ecore metamodel, this usecase shown in Figure 6.16 demonstrates the transformation approach regarding these special

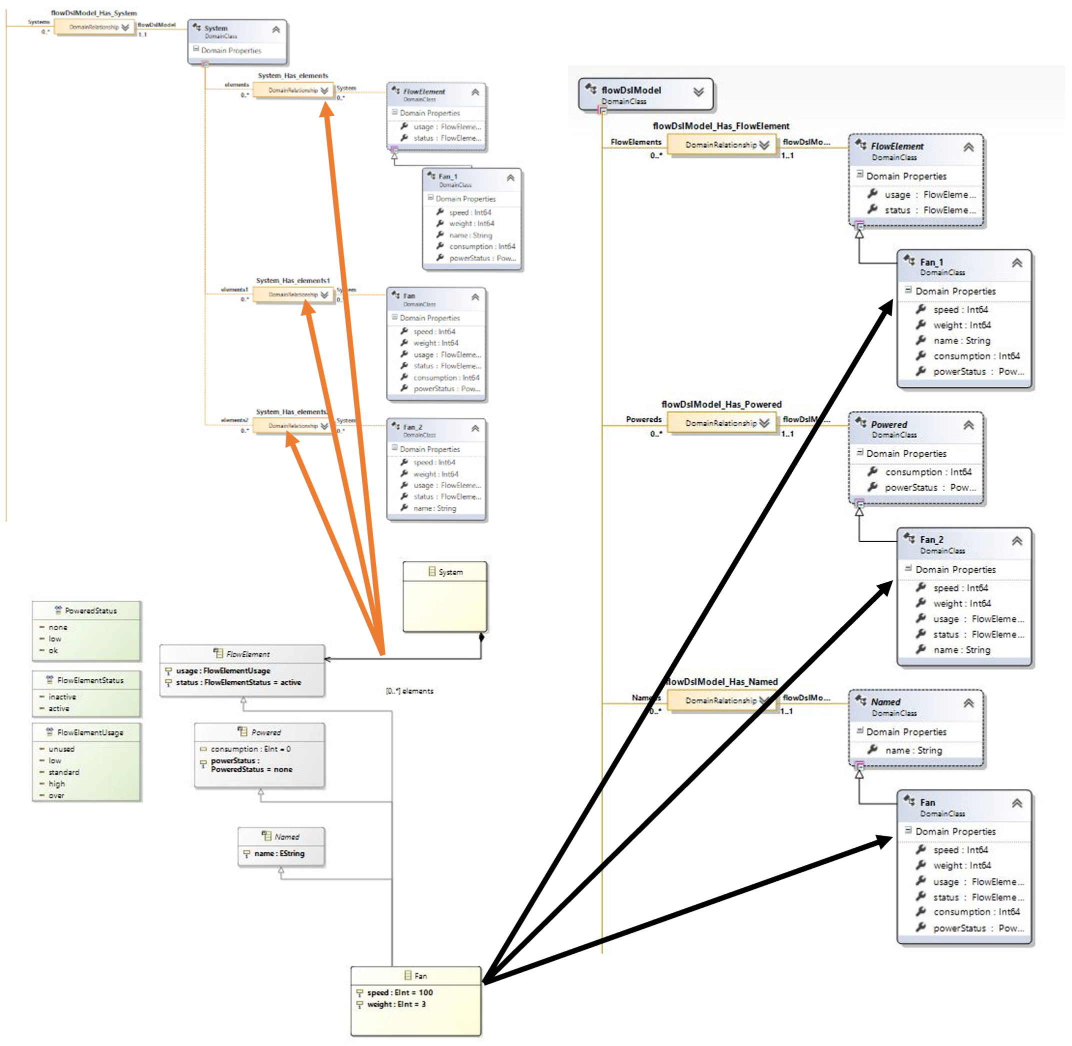


Figure 6.15: Usecase #4.2 - Multiple Inheritance with multiple `superTypes` in Ecore metamodel (bottom left) vs. transformed result in MSDKVS (top left and right)

class types in EMF. The custom datatypes result in additional, template C# classes which must be included in the MSDKVS' project folder.

6.3 Metamodel collections

To be able to answer the research questions given in Chapter 1, and to validate the transformation approach with recent technologies, a collection of diverse metamodels has to be acquired first. Many metamodels are contained within so-called metamodel zoos that are available either inside a zipped container or downloadable separately. For EMF, there

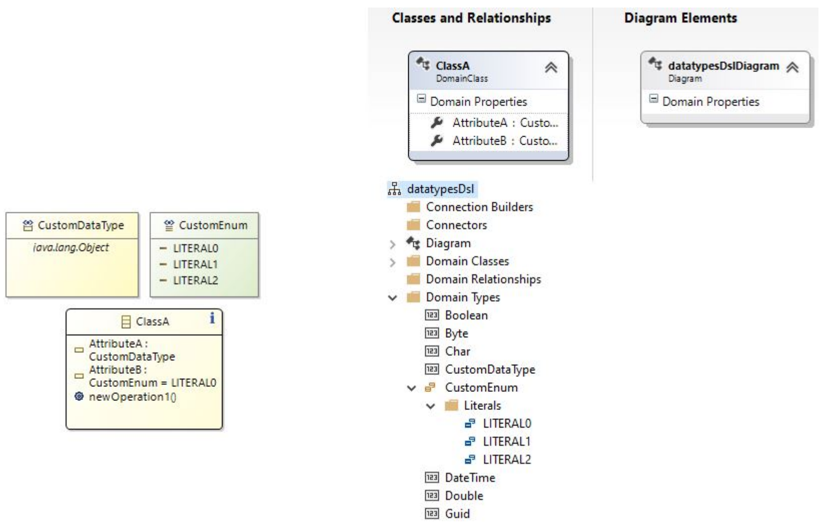


Figure 6.16: Usecase #5 - Ecore metamodel with EEnum and EDataType definition (left) vs. transformed result with DSL Explorer in MSDKVS (right)

exist well established zoos such as the AtlanMod (now known as NaoMod[1]) zoos[2], called "Atlantic Zoos", where a huge amount of metamodels modelled in different frameworks, like Ecore or UML, and for a variety of domains, exist. These metamodels range from Ecore-based language definitions for different well known software applications (e.g., LaTeX, GraphViz, Mantis) to programming language syntaxes (e.g., C, Java, HTML, Pascal) and other different domain specific metamodels like airport traffic or movie databases. A selection of these metamodels is done with regards to their sizes, complexity and their overall structures, whereas more sophisticated metamodels in terms of number of classes and relationships are in addition explicitly selected and where features like inheritance and containment references have been used more frequently to model the underlying domain. On top of these zoos, [34] also offers a search engine available as a web application where Ecore models can be searched among by using different filter criteria such as desired metamodel syntax or user-defined keywords. These diverse metamodels are then fed into the M2 metamodel transformator part of the M3B and the output is then evaluated against certain criteria defined in the subsequent Chapter 9.

Metamodel zoos for MSDKVS do not explicitly exist. Therefore, the aggregation of metamodels has to be done differently. Several source code repositories, like Github and Google Code, are filtered accordingly to retrieve the *DslDefinition.dsl* files for MSDKVS. As the framework itself is not as popular as EMF, less metamodels are found using said approach. Overall, after filtering out duplicates and removing empty or almost empty metamodels, a collection of 40+ metamodels has been collected.

[1]`https://naomod.github.io/`, last accessed on 04.01.2023
[2]`https://github.com/atlanmod/atlantic-zoo`, last accessed on 04.01.2023

CHAPTER 7

Evaluation of M2 Transformation

This chapter describes the results from executing the first part of the transformation bridge, namely the metamodel transformation, which takes as input the necessary files for designing a metamodel from the source framework and transforms them to the corresponding output files for the target framework. The transformation, as mentioned, is implemented in both directions, which means that either MSDKVS metamodels (files ending with *.dsl) and EMF metamodels (files ending with *.ecore) can be transformed. When transforming metamodels from EMF to MSDKVS, it suffices to specify only the abstract syntax information of the Ecore metamodel file, the genmodel and odesign files are optional. On top of the resulting output files, a mapping information file, named *mapping.json*, is created containing necessary information for the following model transformation, such as applied naming strategies and specially applied transformation patterns for backtracking and entity lookup.

The results are analyzed and described based on quantitative and qualitative metrics to answer parts of the research questions proposed in Chapter 1. First, a quantitative analysis in Section 7.1 is performed based on input and transformed metamodel metrics, where aspects like class count, relationship count, datatype conformity, shapes, tool similarity etc. are compared. Depending on the results, possible findings may give insight about how the metamodel is changed based on certain criteria in the source metamodel in the course of the transformation. Afterwards, the qualitative analysis in Section 7.2 lists the results of the transformation's success rate on the set of selected input metamodels and their validity check inside the target platforms. Finally, the transformation's semantics are investigated in Section 7.3 regarding a subset of source and target metamodels in terms of platform interactivity, behavior and, if available, their graphical representations.

The analysis is conducted as follows: First, a set of metamodels is collected from the previously mentioned, publicly available pools of metamodels contained either within source code repositories or maintained on several websites. An input collection of 44 metamodels for the MSDKVS and 75 for the EMF side, containing not only Ecore metamodel files, but also Sirius representations, is used to survey the transformation bridge's overall performance and feasibility. The list of selected metamodels with their source of origin is given in this thesis' appendix section D, some of them related to [41], where different grammar-based sources were fetched and their notations, usually available in (E)BNF, transformed to equivalent Ecore models and other frameworks as well. To help with the analysis, additional methods were implemented to create detailed textual output in addition to the resulting metamodels to list the steps the transformator has executed and the entities it has created on the target side in comparison to their corresponding representations in the input framework. After traversing through every metamodel and transforming them to the target framework, the source input and generated output files are compared to give a statistical overview on different selected criteria regarding their entity counts, number of inherited classes and shapes, etc., which not only describes the abstract concepts but also , if present, the concrete graphical syntax notations abstracted and generalized in Chapter 3. To suit the evaluation approach, statistics about syntax element distribution were also gathered and listed in the form of a *.csv* file for better readability and the ability to import it into an Excel sheet. Questions regarding the validity and executability of the transformed metamodels are then answered accordingly using the resulting data. These answers will then be used later on to fully discuss and revisit the initial research questions asked in Chapter 1.

7.1 Quantitative Analysis

This section dives deeper into the analytical aspects of the transformation approach on the M2 layer, the metamodel level. First, quantitative aspects are listed and compared. Analytical results and concrete findings for determining the outcome beforehand are discussed according to the source and target metric values. The used source metamodels were selected to contain a set of different domains and unique features regarding their platform, e.g., *DomainRelationships* with attributes in MSDKVS or multiple inheritance relationships in EMF. Tables 7.1 and 7.3 list statistical information about the source metamodels used in the following transformation runs. Tables 7.2 and 7.4 on the other hand show the statistical information about the resulting transformed metamodels of the target framework.

The selected metrics, especially the abstract syntax related metrics, have been chosen in accordance to their relevance and based on the analysis done in [8]. In addition, relevant metrics regarding the concrete graphical syntax elements have also been identified based on their occurrence rate among the source metamodels.

7.1.1 Abstract Syntax

Regarding the source metamodel characteristics given in Table 7.1, it can be concluded that the selection of retrieved DSLs for MSDKVS used as input for the M2 transformation is smaller on average regarding their abstract and also their graphical concrete syntax elements (see Table 7.3) compared to the ones selected from the available metamodel repositories for the Ecore environment. Many of the DSLs describe small domains with more understandable and simpler class and relationship hierarchies. Excluding the enumeration and attribute values, one can see that the used subset of available metamodels in Ecore has much higher maximum values for, e.g., defined classes and relationships, as opposed to MSDKVS. This comes from the fact that some Ecore metamodels were selected for transformation purposes that have multiple *EPackages* and *ESubPackages* defined, in addition to also referencing additional Ecore metamodels through relationships and inheritance structures.

The investigated and evaluated syntax elements where mainly adapted from the core M3 concepts identified in Section 3.5, thus distinguishing between classes, relationships and their attributes and enumeration values. Explicit attention was also given to more unique and complex features such as abstraction and inheritance relationships, to be able to apply the transformation rulesets defined in the previous chapters and to give an insight on how these platform-unique features are dealt with inside the transformator. The values are retrieved while the transformation code is executed, as mentioned before, and collected inside additional *.csv* files, one for abstract and one for concrete syntax statistics, which are then imported and aggregated inside Excel for evaluation purposes.

	EMF				**MSDKVS**			
	Min	**Med**	**Max**	**Avg**	**Min**	**Med**	**Max**	**Avg**
Grouping	1	2	50	2.83	1	1	1	1
Classes	1	12	300	33.56	2	8.5	39	10.55
Abstract Classes	0	2	83	5.77	0	1	9	1.61
Inherited Classes	0	7	335	31.79	0	3.5	34	5.20
Multiple Inheritances	0	0	134	4.75	- [1]	-	-	-
Relationships	0	11	437	36.23	0	9	40	10.86
Inherited Relationships	-[2]	-	-	-	0	0	4	0.30
Relationships as Class	-[3]	-	-	-	0	0	15	1.18
Attributes	0	10	98	21.32	1	17.5	170	29.11
Enumerations	0	0	18	1.36	0	1	21	2.43
DataTypes	0	0	60	0.92	0	0	9	1.36

[1] MSDKVS does not support multiple inheritance structures
[2] EMF does not support inheritance among relationships
[2] EMF does not support attributing relationships and using them as classes

Table 7.1: Source metamodel abstract syntax metrics used in experimental transformation runs

Based on these results, following observations about abstract syntax transformations can be made, grouped and discussed accordingly for each transformation direction.

	MSDKVS				**EMF**			
	Min	**Med**	**Max**	**Avg**	**Min**	**Med**	**Max**	**Avg**
Grouping	1	1	1	1	1	1	1	1
Classes	1	13	346	38.92	2	10	47	11.57
Abstract Classes	0	1	109	6.59	0	1	9	1.61
Inherited Classes	0	7	335	31.79	0	3.5	34	5.20
Multiple Inheritances	-	-	-	-	0	0	0	0
Relationships	0	12	3750	130.83	0	13	79	18.20
Inherited Relationships	0	0	0	0	-	-	-	-
Relationships as Class	0	0	0	0	-	-	-	-
Attributes	0	11	236	28.28	1	17.5	170	30.27
Enumerations	0	0	18	1.36	0	1	21	2.43
DataTypes	0	0	60	0.92	0	0	9	1.36

Table 7.2: Transformed metamodel abstract syntax metrics used in experimental transformation runs

EMF to MSDKVS

Grouping. Only one root layer is available in MSDKVS, transforming to exactly one *EPackage* element as well.

Number of classes. When regarding the number of classes from the Ecore metamodels and the transformed MSDKVS DSLs, one can infer, as multiple inheritance structures lead to duplicated classes and relationships, the average and maximum values are higher on the target side. When consulting the number of multiple inheritances used in the source metamodels, on average around 6 multiple inheritance structures are present, although more than half of the source metamodels do not use this feature, signaled by the median value being 0. This feature in addition to the one being the generation of root elements if no suitable source class can be found in the Ecore metamodel, add to the fact that the med, max and avg values are generally higher in terms of classes in MSDKVS.

Number of abstract classes. One of MSDKVS' requirements is that the root class, i.e., the diagram element, must not be abstract. When selecting a suitable root element from the Ecore metamodel and the found element is flagged as abstract, its transformed domain class looses said property. This is a possible explanation for the median value being lower than its source counterpart. Resolving multiple inheritances to multiple single inheritances, where abstract classes are possibly duplicated, results in the other values, max and avg, being higher than their source values.

Number of inherited classes. The number of inherited classes should generally not change, as multiple inheritances with multiple classes being referenced inside the subclasses' `ESuperType` attributes are transformed into duplicated classes in MSDKVS with each having one `BaseClass` reference.

Number of multiple inheritances. As MSDKVS does not support multiple inheritance structures, and only single inheritance structures are possible, the resulting values are not defined.The resolution of such structures are affecting the other abstract syntax values though, like additional relationships and (abstract) classes.

Number of relationships. Typically, when leaving out multiple inheritance structures, relationship counts in MSDKVS should be lower as bi-directional references from EMF, which account for two EReferences, transform to one Domain Relationship. As few of the used Ecore metamodels contain multiple EPackages and are therefore quite large regarding their class and relationship counts, it is not unusual for them to contain also a large number of multiple inheritance structures, leading to copied Domain Classes and, as a result, also duplicated Domain Relationships. The median value before and after the transformation differ only slightly, whereas the average and maximum value of relationships transformed from larger metamodels with a lot of multiple inheritance structures lead to disproportionate sums.

Number of inherited relationships. This feature is not present in EMF.

Number of relationships as class. This feature is not present in EMF.

Number of attributes. The number of attributes on average is nearly double the amount in the source platform, stemming from the fact that multiple inheritances also amount for more attributes as the attributes of super classes have to be copied to duplicated, detached sub classes.

Number of enumerations. As enumerations are both defined separately (own *EClass* type in EMF and separate section in serialized XMI format in MSDKVS), the transformation is a 1:1 mapping between both platforms, generated identical values.

Number of datatypes. Same as for enumerations, custom and external datatypes can be mapped directly to their identical feature on the other platform.

MSDKVS to EMF

Grouping. In MSDKVS, only one root language element is present, wrapping all abstract syntax element into one layer thus resulting in only one EPackage on the target side.

Number of classes. As some of the DSLs used for the transformation define domain relationships as classes, the resulting Ecore metamodels contained EClasses transformed from domain classes and domain relationships, leading to higher counts of EReferences in EMF as well.

Number of abstract classes. As abstracted domain classes can be directly mapped to EClasses marked as abstract, this value is the same for every DSL.

Number of inherited classes. Same as abstract domain classes, inherited domain classes are also directly mapped to EClasses and their superTypes attributes referencing only one super class.

Number of multiple inheritances. Multiple inheritances are not available in MSDKVS, thus transforming to zero multiple inheritance structures in EMF.

Number of relationships. The number of relationships is higher as domain relationships that were used as classes, contained attributes or were used inside domain relationship inheritances transformed to EClasses having one EReference to the source class and one EReference to the target class of the original relationship. Domain relationships, that also transform to bi-directional references in EMF, result in two EReferences as well. If a domain relationship had domain properties, but was used as a containment reference, these properties are added to the targeted transformed EClass.

Number of inherited relationships. Domain relationships, that contain the attribute *BaseRelationship*, are rarely present inside DSLs. Four out of all the used DSLs for this evaluation use this feature. Normally, the relationship that is being inherited from contains attributes, thus being transformed to an EClass entity as well, resulting in higher EClass and EReference counts in EMF.

Number of relationships as class. Domain relationships that either contain attributes or are marked as classes and thus being able to be targeted by other domain relationships are transformed to EClasses, as EMF does not offer this feature, leading to higher numbers in terms of class and reference counts.

Number of attributes. In contrast to the transformation direction EMF to MSDKVS, here, no multiple inheritance structures are available thus no attributes have to be duplicated to cloned subclasses. When domain relationships contain attributes themselves, they are transformed to classes, therefore normally resulting in the same number of attributes, except when these relationships are defined as containment relationships, then the attributes are added to the referenced EClass, if no attribute with the same name and type is available. If only the name exists in the target EClass, then the added attribute is renamed accordingly. This behavior results in fewer EAttribute counts on average.

Number of enumerations. As enumerations are both defined separately (own *EClass* type in EMF and a separate section in MSDKVS), the transformation is a 1:1 mapping between both platforms, thus generating an equal number of enumeration types.

Number of datatypes. Same as for enumerations, custom and external datatypes can be mapped directly to their identical part in the other platform.

7.1.2 Graphical Concrete Syntax

	EMF & Sirius [1]				MSDKVS			
	Min	Med	Max	Avg	Min	Med	Max	Avg
Class Shapes	0	0	12	0.60	0	2	16	3.48
Inherited Class Shapes	-	-	-	-	0	0	10	0.43
Icon Shapes	0	0	18	0.77	0	0	11	0.39
Relationship Shapes	0	0	39	1.73	0	2	10	3.30
Containment Shapes	0	0	27	0.79	0	0	5	1.23
Tools	0	0	398	9.81	0	6	25	7.84

[1] 22 out of 75 Ecore metamodels had a Sirius VSM

Table 7.3: Source metamodel concrete syntax metrics used in experimental transformation runs

	MSDKVS				EMF & Sirius			
	Min	Med	Max	Avg	Min	Med	Max	Avg
Class Shapes	0	0	12	0.60	0	3	23	4.27
Inherited Class Shapes	0	0	0	0	-	-	-	-
Icon Shapes	0	0	18	0.77	0	0	11	0.39
Relationship Shapes	0	0	39	1.73	0	2	17	3.98
Containment Shapes	0	0	27	0.79	0	0	5	1.20
Tools	0	0	14	1.39	0	10	41	13.82

Table 7.4: Transformed metamodel concrete syntax metrics used in experimental transformation runs

Regarding Table 7.3, the minimum values are all zero, which comes from the fact that all Ecore metamodels were looked at in the course of the quantitative analysis and, as mentioned before, only 22 of them did include a Viewpoint Specification Model in Sirius.

EMF to MSDKVS

The following comparisons deal with the analysis on the graphical concrete syntax layer, where multiple inheritance structures, as opposed to the abstract syntax, are not specifically reflected in the transformed values, as only additional ShapeMaps have to be defined when a class is duplicated in order to result in multiple single inheritance relationships. As mentioned before, min and med values are zeroed out as only a small subset of the selected metamodels in the evaluation also contained a Sirius VSM.

As MSDKVS offers graphically enhancing the abstract syntax element out of the box and the tools for doing so are integrated into the editor's toolbox, the barriers of defining a visual representation are easier to overcome as no additional framework is needed, thus leading to more metamodels containing shape definitions than there are VSMs defined for Ecore metamodels.

The most affected features are of course the values for tools, as Sirius offers a vast variety of additional functionalities on top of the basic requirements for creating, editing and deleting abstract syntax elements.

Number of class shapes. The number of class shapes is identical when comparing source and target metamodel concrete syntax metrics. As mentioned before, if a VSM targets an EClass that uses multiple inheritance and thus has to be duplicated in MSDKVS, only additional ShapeMaps from the initial transformed shape have to be duplicated, and no additional shape has to be generated.

Number of inherited class shapes. As shape inheritance is not available in Sirius, the resulting values are all zero.

Number of icon shapes. Same as class shapes, icon shapes are transformed equally, resulting in all cases in a 1:1 mapping.

Number of relationship shapes. As all EReference types, except containment references, are transformed to domain relationships, their EdgeMappings are transformed to Connector shapes as well, resulting also in a 1:1 mapping.

Number of containment shapes. ContainerMappings are equal to MSDKVS' Compartment shapes, thus resulting also in a 1:1 mapping between the frameworks.

Number of tools. Regarding the number of tools transformed to the target framework, many of them are lost as their functionality is typically not supported by MSDKVS' simple graphical tooling framework. Only the required creation tools for relationships and classes can be integrated within MSDKVS, resulting in lower tool values compared to Sirius, where much more sophisticated tooling can be done in the form of query languages like Acceleo, OCL or operation trees.

MSDKVS to EMF

Regarding the shape distribution after transforming from MSDKVS DSLs to EMF equivalent metamodels, the differences concern mostly platform unique features like using DomainRelationships as classes and the implicitly available tooling capabilities of the framework, as references and classes can be deleted and edited out-of-the-box.

Number of class shapes. The different class shapes and their geometries are usually transformed to equal representations on the Sirius side, but, as some of the DSLs contain DomainRelationships that are used as classes, some Connectors have to be transformed to NodeMappings, with 2 additional EdgeMappings, resulting in higher numbers in both class shapes and relationship shapes in EMF. If such a domain relationship is targeted by a connector, the number of resulting EdgeMappings is first reduced by 1, then the NodeMappings are incremented by 1 and the additional EReferences result in the target EdgeMappings being incremented by 2.

Number of inherited class shapes. As inherited class shapes are not available in Sirius, the inherited decorator maps of these shapes are used to overwrite the super shape's decorator maps, resulting in equal representations on both sides for all domain elements.

Number of icon shapes. Same as for class shapes, the transformation of IconShapes results in a 1:1 mapping to NodeMappings with style type `WorkspaceImageDescription`. As the resulting NodeMappings retrieved from dissolving attributed relationships and their connectors, which cannot have an icon attributed to them, to normal class shapes, this behavior does not affect the icon shape mapping.

Number of relationship shapes. Connectors are transformed to equivalent EdgeMappings, and generated EdgeMappings from DomainRelationships being transformed to EClasses result in higher relationship shapes in EMF.

Number of containment shapes. CompartmentShapes are transformed to their similar ContainerMappings, resulting in a 1:1 mapping.

Number of tools. MSDKVS only offers the possiblity to define creation tools for model entities like domain classes and domain relationships. When transforming, corresponding creation tools are created in Sirius. Additionally, deletion functionality, which is automatically supported in MSDKVS modeling canvas, has to be generated on top of these transformed creation tools to offer semantically equivalent interaction within both frameworks, thus leading to higher med, max and avg tooling values in EMF. DomainRelationships used as classes that are targeted by ConnectionTools have to be also considered, resulting in additional Sirius tool descriptions of type `NodeCreationDescription`, one per eliminated ConnectionTool, as well as two new ConnectionTools referencing the generated relationships.

7.2 Qualitative Analysis

To analyse the qualitative aspects of the transformed metamodels, the transformed results are opened in the target framework editor. For this, a project is created beforehand that takes the name of the transformed metamodel (i.e., the name of the source metamodel) as the project name and namespace. Both of the frameworks offer automatic validation, meaning that when the project files are opened, their internal structure is looked at and checked against a standardized schema. Validation errors, if existent, are listed accordingly. If a metamodel cannot be opened, it is deemed as faulty and the transformation on the M2 layer has to be adapted. As MSDKVS diagram editor on the M2 layer offers the ability to define concrete syntax elements upon the abstract syntax entities, the validation step checks both areas for errors. On EMF, the optional .odesign files containing the definitions for graphical concrete syntax mapping have to be validated separately.

If the validation step finds no error, the frameworks' functionalities upon creating models are tested (i.e., starting of run time instances, model canvas editing through toolbox

Table 7.5: Transformation success rates

Direction	**Abstract Syntax**			**Concrete Syntax**		
	Cases	**Errors**	**Rate**	**Cases**	**Errors**	**Rate**
MSDKVS → EMF	44	0	100%	44	0	100%
EMF → MSDKVS	75	0	100%	22[1]	2 [2]	90.91%

[1] Only 22 collected metamodels were bundled with custom .odesign files
[2] VSMs contained references to multiple Ecore metamodels

tools). Table 7.5 lists the amount of metamodels used for the transformation experiments on the M2 layer and the number of cases which returned an error and the overall success rate of both abstract and concrete syntax transformation validation results.

Success Rate MSDKVS → EMF For the evaluation, the transformed EMF files, including the *.ecore*, *.genmodel* and .odesign files, were copied to a suitable Ecore Modeling Project, having the same name as these files, thus overwriting the default files in the process. Every copied file was then opened. If the content of the file contained an error, the IDE would display these errors accordingly. If no errors were present, the file contents could be investigated, edited and validated in detail. In terms of transforming DSLs to EMF-equivalent metamodels, no errors were present while opening the resulting files, meaning the generation of editor and model code can be successfully executed and the Sirius VSM can be used for graphically enhancing the underlying models.

Success Rate EMF → MSDKVS The transformed *.dsl* file is imported into an empty Domain-Specific Language Designer project inside Visual Studio in order to examine the M2 transformation's validity in the direction of EMF2MSDKVS. Regarding the abstract syntax validation, no error messages were produced when the generated file was opened inside the editor. Although the validation process takes both abstract and concrete syntax specifications into account, based on the error message description, one can infer which area of syntax the error is targeting. Regarding the concrete syntax validation, two DSLs were deemed invalid, resulting from the fact that the Sirius shapes were targeting multiple Ecore metamodels, thus also referencing abstract syntax elements that could not be transformed.

7.3 Semantic Analysis

As a final part of the evaluation process for the M2 transformator, a selection based on the source and their transformed target metamodels used for each direction are collected. These metamodels are analyzed more in-depth regarding their behavior while creating and operating with models inside their generated runtime environments. The goal is to investigate the metamodel functionalities and their correctness to further strengthen the interoperability aspect of this thesis' transformation approach.

Following aspects, comprising mainly of the mapping of meta-metamodel concepts from one framework to the other, were chosen to analyse and investigate semantical equivalence between the two platforms:

- **Class Mapping:** Correct mapping of class entities has to be evaluated, their names, number of attributes, inheritance relationships and, if applied, possible renaming strategies on the target framework are looked at.
- **Relationship Mapping:** Multiplicity mapping, source and target class mappings and the special types of relationships like containment and bi-directional references have to be evaluated. This includes, e.g., a correct cascading of delete behavior in containment references.
- **Attribute and Enumeration Mapping:** Attribute mapping analysis includes datatype conformity, identical default values and enumeration literals (taking into account the naming conventions listed in Section 6.1.3)
- **Class Shape Mapping:** The mappings between abstract syntax element and graphical shape are to be investigated and compared between source and target framework. Especially in terms of correct coloring and styling of attributes as well as mapped geometries.
- **Relationship Shape Mapping:** Same as class shape mapping, relationship shape mappings have to be evaluated and compared. Routing styles, source and target styling, line styles and much more attributes must be considered.
- **Tool Mapping:** Sectioning of tools, tool icons and their interaction with the modeling canvas has to be compared and evaluated, especially for creating relationships that the same valid source and target entities have to be selected.

Regarding the evaluation process, subsets of ten metamodels for each direction used in the quantitative and qualitative analysis are selected for the investigation of these defined aspects. The metamodels are chosen by setting up scattered plot diagrams, each for two selected metamodel characteristics standing in correlation to one another based on representative metrics calculations discussed in [20]. These metamodels then also act as the referencing metamodels for the M1 transformation evaluation done in Chapter 9, as they have already been deemed as semantical correct metamodels.

The steps taken in order to evaluate the semantic properties of the selected metamodels, are the following:

1. Transforming source to target metamodel

2. Copy the resulting files into the target environment
3. Confirm the validity of both abstract and concrete syntax
4. Generate executable code based on the metamodel
5. Run an experimental instance of the platform
6. Create empty model files based on the transformed metamodel
7. Initialize the graphical modeling canvas, if available
8. Interact with the modeling canvas and the tree editor, i.e., create classes and relationships, edit attributes, delete elements
9. Investigate the resulting shapes and compare their properties to the source definiton

7.3.1 EMF to MSDKVS

Due to the evaluation process being a time consuming and manual task, not every metamodel transformation conducted in the previous steps could be investigated in detail. By having selected a subset using a variety of different aspects spread among all the available metamodels, this evaluation thus gives a fairly adequate overview on how semantic equivalence regarding model behavior and interaction is achieved.

Figure 7.1 shows the approach used for selecting the set of metamodels that have to be investigated further for semantic evaluation. Each diagram, having 2 distinct but connected metamodel metrics as its x and y axes, gives further insight on how distributed the values among the collected source metamodels are. The yellow dots resemble the x and y coordinates, i.e., metric values, for the metamodels that have been chosen for semantic evaluation, which will be listed afterwards in Table 7.6. The blue dots visualize the remaining metamodels used inside the feasibility approach in the previous section.

When looking at Table 7.6, four of the ten selected metamodels had graphical representations available, namely **poosl**, **behaviortree**, **sensorProject** and **simplePDL**. Their shape mappings to abstract syntax elements and their tooling sections were transformed and usable in MSDKVS correctly.

The **ATL** and **KDM** metamodel definitions could also be transformed eventually, after some minor fixes had to be made in order for MSDKVS' code serialization to work, as two `XmlRelationshipData` constructs for the same *DomainClass* or its base classes must not have the same `RoleElementName` attribute, or else a validation error would be thrown.

The more cumbersome metamodels deemed to be **UML2** and **c_sharp**, as they contain multiple inheritance structures with many super type definitions and a number of `SubEPackages`. This made evaluating the correct naming convention for domain roles and domain relationships difficult at first, since also opening the resulting DSLs in Visual

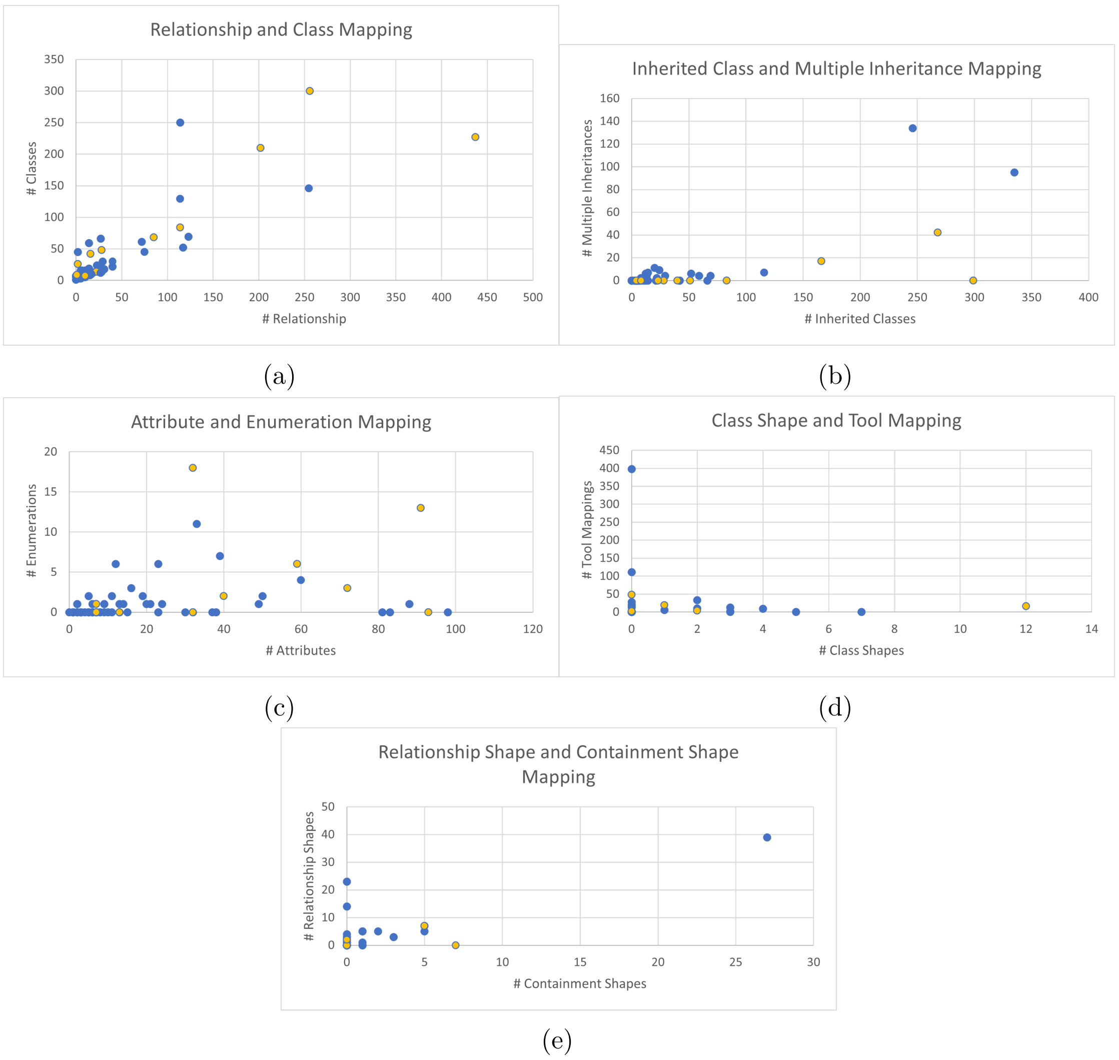


Figure 7.1: Scatter charts of metric distribution and highlighted metamodels used for manual evaluation for EMF.

Studio resulted in performance issues due to their sizes. The problems during validation of the transformed **c_sharp** DSL were eventually eliminated, as they only resulted from duplicated source domain role names. The **UML2** DSL on the other hand could not be solved completely as of yet, as **1)** the amount of relationships being rendered on the metamodeling canvas made the interaction and finding the source of the problem impossible and **2)** the amount of faulty domain relationships with errors due to the applied nested renaming strategies as the Ecore metamodel contained multiple eSuperType relationships with these super types also having multiple eSuperType relationships was still error-prone during model evaluation, thus resulting in the **UML2** metamodel not being targeted in the model transformation evaluation. As this metamodel far exceeds the average representation value of multiple inheritance structures being `4.75` contained within the source metamodels used for the feasibility study of the M2 transformation, namely 42 in combination with 437 relationships (i.e., the highest value of all used source metamodels), further investigation has to be done for these large metamodels in order to achieve better interoperability.

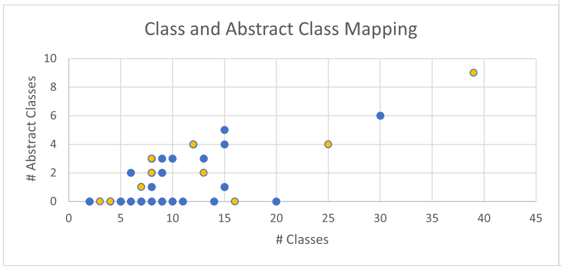
MM	**Validity**				
	Classes	**Relationships**	**Attr./Enums**	**Shapes**	**Tools**
Ant	✓	✓	✓	-	-
ATL	✓	✓	✓	-	-
behaviortree	✓	✓	✓	✓	✓
c_sharp	✓	✓	✓	-	-
HAL	✓	✓	✓	-	-
KDM	✓	✓	✓	-	-
poosl	✓	✓	✓	✓	✓
sensorProject	✓	✓	✓	✓	✓
simplePDL	✓	✓	✓	✓	✓
UML2	✓	✗	✓	-	-

Table 7.6: Semantic evaluation results for transformation direction EMF to MSDKVS.

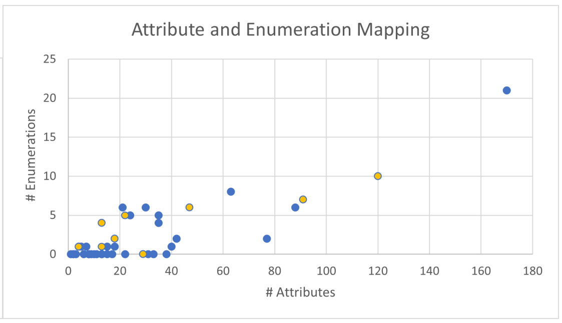
7.3.2 MSDKVS to EMF

With regards to the sizes of the selected metamodels in EMF, DSLs in MSDKVS usually have a much smaller amount of domain elements contained within, making it easier to test the transformation appropriately. Figure 7.2 displays five scatter plot diagrams for the source DSLs used for the evaluation of the M2 transformation, each containing different metric values on their respective x and y axes. Identical to the diagrams for the transformation direction EMF to MSDKVS, the yellow dots symbolize the selected DSLs. Table 7.7 displays the evaluation results. Every language definition, except **fwk_dsl**, contained at least one attributed DomainRelationship, resulting in an EClass with additional EReferences from and to it. In terms of their counts, **cqrsdsl** contained 15 class-like relationships and could not be rendered by Sirius in general as the references were all embedding relationships and for each contained class a separate model instance had to be created in order to cross reference them, thus resulting in invalid shape and layout analysis but valid abstract syntax. Furthermore, this DSL uses containment references for source domain classes that are mapped to GeometryShapes, which, in the transformator's current state, is not possible to visualize correctly after being transformed to EMF and Sirius, as no sub nodes can be added to the resulting NodeMappings, and containment references cannot be visualized by EdgeMappings. A possible solution for future work would be the generation of a Sirius representation per such containment structure in order to visualize them appropriately from separate model files.

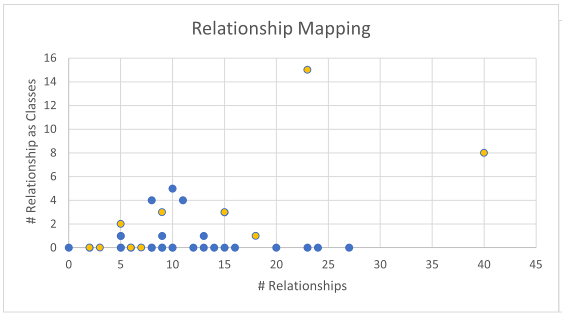
Question #1: Validity of transformed metamodels As mentioned before, the transformation validity not only concerns the abstract syntax, it also targets the constraints



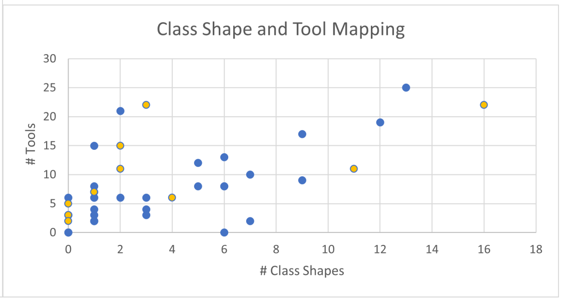
(a)



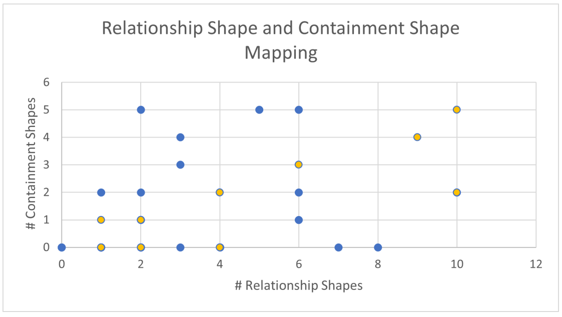
(b)



(c)



(d)



(e)

Figure 7.2: Scatter charts of metric distribution and highlighted DSLs used for manual evaluation for MSDKVS.

on the concrete graphical syntax. MSDKVS and EMF with Sirius offer out-of-the-box validation functionalities, per default automatically executed when the metamodel files inside the editors are opened. A list of possible validation errors and warnings is given on both platforms if something has not been serialized correctly.

Table 7.5 gives an insight on the success rates of the transformation runs regarding these validation constraints for both directions of the transformator. It can be derived that no erroneous cases regarding the abstract syntax transformation was found, resulting in a 100% success rate for both platforms, thus proving that M2 transformator is able to transform every identified and mapped abstract syntax element from one platform to the other without breaking.

DSL	**Validity**				
	Classes	**Relationships**	**Attr./Enums**	**Shapes**	**Tools**
agilemodeler	✓	✓	✓	✓	✓
candle	✓	✓	✓	✓	✓
cqrsdsl	✓	✓	✓	✗	✗
fwk_dsl	✓	✓	✓	✓	✓
generatorlanguage	✓	✓	✓	✓	✓
hostdesigner	✓	✓	✓	✓	✓
mbrdcmdmi	✓	✓	✓	✓	✓
mobiledsl	✓	✓	✓	✓	✓
nhmodelinglanguage	✓	✓	✓	✓	✓
spllanguage	✓	✓	✓	✓	✓

Table 7.7: Semantic evaluation results for transformation direction MSDKVS to EMF.

Regarding the validity of the concrete graphical syntax, no error was identified while transforming from MSDKVS to EMF. Having transformed the Ecore metamodels to MSDKVS, the additional Sirius validation considered the 22 metamodels which contained a graphical concrete syntax specification. In the runs that followed, two out of 22 were faulty, stemming from the fact that multiple Ecore metamodels were referenced from within the .odesign file, therefore creating shapes for not available entities.

As the M2 Transformator is implemented as a bridge between two XML serialization formats, most of the initial problems that occurred originated from deserializing the input metamodel files to a corresponding data structure used to traverse through each syntax element. XML namespace errors were among the most common types of exceptions because of limitations the default C# library provides when deserializing Ecore metamodel files, as they contain numerous classes using the same typed attributes (e.g., firstModelOperations and subModelOperations in tool sections). These errors could be eliminated eventually by changing the content of the metamodel files before deserializing to suit the libraries capabilities (i.e., implementing a pre-parsing string manipulation on the input XML metamodel files).

Question #2: Executability of transformed metamodels The executability assumes a valid metamodel in order to run the code generators for creating and executing modeling runtime instances in the frameworks. When the code generation throws errors or the modeling canvas cannot be initialized, the transformed metamodel files are deemed faulty with regards to their executability and semantic evaluation. All transformed metamodels, that threw no errors during their validation phase, could be used to generate model code on the target platforms and thus initialize runtime editor instances for further modeling purposes.

7.4 Difficulties during implementation of M2 transformation

A few difficulties arose during the implementation of the metamodel transformation in C#, especially in combination with using the XML serialization approach on the Sirius VSM files, as some of the used attributes, especially type attributes, could be set to different namespaces. Examples for this behavior are the `firstModelOperations` and `subModelOperations` for tooling. In order to define different types for a namespace of an attribute, they first have to be defined as subclasses of the XML element itself via the `XmlType` code attribute referencing the correct namespace. These types are then used as annotation in the main class, using the `XmlInclude` attribute in order to deserialize the incoming XML contents appropriately. The problem with this approach is that a type can only be used inside one XML serialization class at a time. When looking again at the firstModelOperations and subModelOperations for example, they both use the `description` namespace types `SetValue`, `CreateInstance` and `ChangeContext`, to name a few. To tackle these obstacles, the *.odesign* file contents are manipulated before being loaded into a corresponding data structure in order to eliminate such duplicate type attributes and errors during deserialization.

Difficulties in testing and opening the resulting files inside the IDEs were also discovered when some files had not been properly closed before overwriting the metamodel definition files in the editor, leading to error messages that, if closed and reopened, were not present anymore. The dynamic diagram file for graphically displaying the content sometimes had to be deleted when a different metamodel was opened inside Visual Studio, in order to regenerate the correct diagram display.

Another problem arose with regards to the transformed DSL sizes in MSDKVS, as some of the bigger Ecore metamodels that were tested resulted in about 50.000 - 100.000 lines of XML, which took a considerable amount of time to load and then could not be displayed properly inside Visual Studio, as some classes and relationships were overlapping one another. In order to "clean up" these overlapping entities, they had to be reordered manually.

As EMF and Sirius are defined in separate files, the separation of concerns lead to a better overview on implementing the M2 layer, whereas on the other hand, the *DslDefinition.dsl* files in MSDKVS containing the DSL's abstract and concrete syntax could be filled with irrelevant information for the task at hand, especially in terms of moniker usage and cross referencing entities within several different areas, making finding bugs and fixing them a tedious task.

For the validity check, the experimental run time instances of Visual Studio sometimes did not load the Toolbox tabs correctly, displaying either only a subset of tools or no tools at all. The MSDKVS API documentation lists possible solutions to this known problem, resetting the experimental instances before rebuilding the solution as an example. This did lower the amount of such faulty behavior, but not entirely.

7.5 Limitations

A few limitations adhere to the transformation approach based on the M2 layer. The M2 transformator is specific to the EMF and MSDKVS metamodels. Each additional platform requires a mapping according to the identified rule sets and each rule requires an implementation in C#. For serializing new metamodel representations from different frameworks, the XML schema has to be analyzed and corresponding data structures for (de-)serialization have to be added. Existing transformations can afterwards be reused, thus creating M3 level bridges spanning multiple metamodels. Another limitation is the realization of the transformator on the current versions of the platforms. Core updates to these platforms and their metamodel serialization therefore require similar updates of the transformator as mentioned before.

Regarding the involved metamodel entities and unique features of each platform, some limitations in the tranformation's current state are present as well. For instance, query languages like OCL, Acceleo and AQL, which are part of the Eclipse family, offer filter mechanisms for manipulating models. These queries can differ in their complexity, making more sophisticated queries hard to transform as they can be seen as languages themselves with their own syntax and grammar. The same can be said for supplemented code snippets (Java services or C# code) which are used for restricting the set of operations on underlying models, customizing different behavior resulting from tool usage, extending generated classes or dynamically changing shapes based on control patterns or queries. The possibility to filter visible model elements is also a feature that some GUIs offer. Sirius contains two kinds of filter mechanisms, namely *Mapping Filters* and *Variable Filters*, which use customizable conditions for filtering. These kinds of filters, written using the previously mentioned query languages, are also not part of the transformation implementation.

Many of the advanced tooling operations supported by Sirius (e.g., filtering using query languages, flow control operations like *Begin*, *For* and *If*, Dialog, and Model Representations operations) cannot be mapped directly to available operations in MSDKVS. As mentioned in Section 3.5, only element creation tools are supported, which target specific metamodel entities. These features are correctly mapped to and from Sirius' model operations *Change Context*, *Create Instance*, and *Set*.

CHAPTER 8

M1 Layer Transformation

Based on the previous steps, this chapter provides detailed explanations of the final tasks of the transformation process, namely the transformation of models themselves based on the results from the metamodel transformation. Before that, an analysis on how models are represented in each platform is given and their serialization formats are compared accordingly.

8.1 Serialization of Models

Each platform executes separate runtime environments for creating models conforming to metamodels. A second instance of the IDE is opened and contains references to the previously (auto)generated modeling files from valid metamodels in order to use them properly. Each model is graphically represented by their metamodel's available graphical concrete syntax, be it either directly inside the framework or in EMF with its Sirius component. If no graphical concrete syntax has been defined, the tree editor of each modeling instance can be used for creating classes and relationships instead by using the available context menu commands.

Each model file has its own extension, conforming to its metamodel name, namespace or file extension attribute, if present. Regarding the serialization of these model files inside MSDKVS and EMF, both use the XML Metadata Interchange (XMI) format, thus making the M1 transformation easier to implement as both are based on the same technology as opposed to platforms that use different serialization approaches, like ADOxx [9]. Furthermore, the XMI serialization format is in general more understandable for humans, making comparing both platforms' representations more comprehensible and implementing the following model transformator an easier task to follow. The Listings in Appendix B juxtapose the contents of each model file based on the familytree example mentioned in Section 6.2.

8.2 M1 Layer Transformation

Typically, one cannot find concrete examples of models based on defined metamodels contained within zoos. Most or nearly all of the retrieved repositories contain only the metamodel definitions (either within *.ecore* files for EMF or *.dsl* files for the MSDKVS) and no concrete models conforming to these definitions are available. To examine the feasibility of this part of the transformation, a random model generator [2] is being used to generate models as input on the EMF side. For transforming MSDKVS models to EMF models, they have been created manually to best represent the underlying domain and its features. Refer to Section 9.3 for possible limitations regarding these employed model generation libraries and techniques as well as the future work section where model generation in MSDKVS is mentioned as a proposal for future work.

The M1 transformator is written in Java, as the Ecore Java API [1], in particular the *Reflection API* for Ecore modelling, can be seamlessly integrated as such into a java project for importing the model and metamodel files based on Ecore to be iterated over, serialized and deserialized as well. This section gives detailed information on how model transformation has been achieved by using said API and explain the transformation approach in both directions using again the familytree examples based on the UML class diagram established in Chapter 3. In the next Chapter 9, the implementation is tested based on concrete examples retrieved from selected metamodel transformations executed in particular in Chapter 7, the results are analyzed and possible improvements based on discovered limitations in terms of usage will be listed.

When comparing EMF and MSDKVS models directly, one can infer that each (simple) attribute from a source class maps to an equivalent attribute on a target class. The transformation of relationships acts as the main difficulty here, because in EMF, references look like attributes (when speaking about XML syntax), whereas in MSDKVS they are expanded by default, creating sub elements of the *DomainClass* element. Furthermore, when looking at the example model files, it can be concluded that containment references (e.g., `Countries` with their lists of `Towns`) are handled differently than simple reference types.

Regarding the concept of inheritance, EMF adds an `xsi:type` attribute containing the subclass type to the superclass entity, whereas in MSDKVS, the name of the subtype is used directly as a tag.

In MSDKVS models, each *DomainClass* element and *DomainRelationship* element obtains a unique identifier (i.e., GUID) to be referenced from inside other elements. These GUIDs, like in the M2 transformator, have to be generated manually when transforming from EMF to MSDKVS on the M1 layer. Special consideration must be given to these GUIDs, as different attributes can be flagged as the identifying attribute, like the `name` attribute of a `Person` entity in the familytree example. These class elements are then referenced by using the generated GUID from the diagram element (i.e., the root element) and the value of this unique attribute value. EMF also uses a referencing technique regarding relationships by giving each class a number based on their position in the model's list of type-equivalent XML elements, which, in addition to their class name, can be used for interpreting simple and bi-directional references.

Diving into MSDKVS, different serialization behavior techniques can be applied to every domain entity (class, property or relationship). When a model is saved, two files are created, similar the the DSL functionality, namely the main file ending with the custom `FileExtension` attribute, and the diagram file used for storing the entity positions and other dynamic appearances, like reference line lengths or colors.

8.2.1 EMF to MSDKVS

Using again said JSON mapping file retrieved from the M2 transformation step, containing mapping transformations for each abstract syntax element regarding their eventual renaming in the target framework and platform unique flags, the model transformation from Ecore-based models to DSL-based models in MSDVKS is executed.

The files containing the source metamodel and its model are used in addition to the generated mapping file. An iterator steps over each *EClass* and then every *EStructuralFeature*, i.e., *EAttributes* and *EReferences*, contained within these class elements. Transformation rules apply and take into consideration the mapping file, whereas each and every abstract syntax element is mapped to one or more target framework elements.

The difficulty hereby in loading the model file into the M1 transformator was, that after investigating the models either randomly generated via the library mentioned in 9.1 or manually created inside EMF, their root XML tags could differ. As MSDKVS has to have a class defined as the diagram's root class on the M2 layer, where all other classes and references can be stepped over, the generated model file also has to have that same class as the root of its content. If that is not the case, the first element corresponding to the identified or generated root element, extracted from the generated mapping.json file in the previous transformation step, is used.

The following list contains information on how the M1 transformation tackles the mapping of the abstract syntax elements from the platform's metamodel layers using the created mapping.json file from the M2 step:

Class mapping

When a class entity is transformed, the target name is extracted from the generated mapping file and with it, an XML tag in the resulting modeling representation is created.

Attribute mapping

Attributes from one class entity can be mapped using the naming conventions documented in the mapping.json file to a target attribute contained within the previously transformed target class entity. Attributes are used the same way in both platforms, as the M2 transformator does not attribute the resulting DomainProperty XML data with the ElementReference indicator.

Relationship mapping

Relationship mappings are the most challenging mappings when transforming from EMF to MSDKVS, as they can be represented in different ways depending on their types.

Containment references are contained within the compositing class. When regarding their XML representations in EMF, such compositing classes are displayed with their inner classes as, if correctly formatted, indented sub elements. In MSDKVS, additional XML tags have to be generated to achieve the same structural functionality when transforming a containment reference. As mentioned before, **simple and bi-directional references** use a numbering mechanism to target other class elements inside the model. When transforming such relationships, the list of available elements filtered by the target classes has to be collected and the right index selected in order to correctly transform the relationship.

Inheritance mapping

Inheritance mapping is cumbersome when multiple inheritance relationships are involved, as they are distributed among several single inheritance relationships and copied classes in MSDKVS as a result of the M2 transformator. Thus, the mapping.json file has to be consulted when transforming on the M1 layer in order to find the correct inheritance structure to use for transforming the model entity. Each class mapping contains a superclass mapping, referencing the transformed superclass.

Additional information

Information such as namespace declarations inside the models' root tags or the root class mapping have to be handled accordingly. The mapping file contains additional information on how the transformed metamodel and its file extensions where named after having executed the M2 transformations in order to add said namespaces into the XMI.

Regarding the transformation code itself, following code sections are executed sequentially in order to retrieve the transformed result, the MSDKVS model file:

1. **Setting the transformation direction:** For determining the direction from which platform the model has to be transformed the first argument when executing the transformator from the command line has to be `EMF2MSDKVS`.
2. **Loading the file contents:** The submitted files must contain the *Source Ecore metamodel*, the *Source Ecore model* based on the metamodel and the *Mapping file* retrieved from the previously executed M2 transformation. The mapping file is deserialized into a data structure identical to the one created during the M2 transformation. The Ecore metamodel is loaded into an EMF `Resource` to create the `EPackage` and its `EFFactory` instances used for reading the model file contents. To dynamically deserialize the model contents, the `XMIResourceFactory` is used for reading said model.

3. **Reference the model's root element:** The root element of the Ecore model is saved into a static variable to be stepped through and retrieve its contained *EClass* objects with their *EReferences* and *EAttributes* accordingly.

4. **Transformation of the root element:** The first element to transform is the root element of the model file. In order to be able to deserialize every model file based on any Ecore metamodel, a `List` of `DynamicEObjectImpl` objects is used for keeping track of already transformed class entities used later on for resolving the remaining relationship links. The root element has to further contain the namespace declarations and the `dslVersion` attribute to be a valid MSDKVS model.

5. **Transformation of remaining classes:** Before the relationships are solved, the class entities are transformed and every relationship is put into a map variable for looking up related relationships later on. Classes are transformed by looking up the appropriate `ClassMapping` information created in the M2 step in order to get the name of the resulting DomainClass if their source names have been adapted by any renaming strategies resulting from containing reserved keywords in their original names or duplication of elements resulting from multiple inheritance structures. EStructuralReferences of type `EAttribute` are transformed next, being themselves either a simple value attribute or an enumeration literal, where a lookup on the class mapping's `AttributeMappings` is done in order to retrieve the target domain property name or enumeration name with the correct literal. Every subsequent transformed class is added to the target model's root element as an `Element` class with key-value pairs resembling these attributes.

6. **Transformation of relationships:** Each class element contains a list of `Dynamic EObjectImpl` elements, resolved after every class entity has been transformed, thus adding the correct relationship structures. Until then, it temporarily contains all references to other dynamic `EObjects`, uniquely identified by their dynamic values set inside the Ecore Reflection API through deserialization. Inheritance relationships are retrieved by resolving the `SuperClassMappings` inside the appropriate `ClassMapping` recursively in order to get the correct base class, having possibly been renamed as a result of multiple inheritances in the source metamodel.

7. **Serializing the MSDKVS model file:** The created `Element tree` based on the previous steps is traversed and saved with the help of a `StringBuilder` in order to generate the MSDKVS-readable model file. Each class and reference element is represented by an `Element` object, each attribute is represented via a class element's `ValuePair` object, having the `Key` parameter define the name of the attribute of the resulting *DomainClass*, and the `Value` parameter define the value of said attribute. Nested *Element* objects signalize the usage of different kinds of relationships, i.e., inheritance, containment or reference relationships.

8.2.2 MSDKVS to EMF

When transforming models from MSDKVS to EMF, the transformed Ecore metamodel in combination with the Reflection API is used to create the target model entities. The procedure differs in terms of loading the model files into code, MSDKVS' models are available in XMI format but Java does not know how to interpret them. Thus, a basic XML reader library is used for deserializing the generic XMI structure of the model into code. Identical to the serialization structure of the transformation result for the direction of EMF to MSDKVS, the models are deserialized to a tree based data template containing several key-value pairs resembling their various tags and attributes referencing the metamodel attribute definitions. The resulting EMF model is then saved as an XMI serialized file with the help of EMF's persistence framework.

The following list contains detailed information on how the M1 transformation tackles the mapping of the abstract syntax elements from the metamodel layer using the mapping.json file from the M2 step:

Class mapping

Class mappings are transformed to corresponding class name tags in Ecore. Lists for each classes already transformed are maintained to get their exact index inside the XML file for reference transformations that are not containment references.

Attribute mapping

Same as for transformation direction "EMF to MSDKVS", attribute elements are added to the transformed class elements as attributes as well. Important to note here is that the serialization of attributes can be different for each attribute, depending if their `Reference` flag has been set to, e.g., "Element". Thus, the mapping file has to be consulted for each attribute to retrieve the mapping information containing the value of said flag. Normally, the flag defaults to "null", meaning the attribute is used as a normal attribute. When the flag's value is "Element", the attribute has been deserialized to a separate sub element with one ValuePair object, where the key is the name of the attribute and the value equals the attribute's value, easily being translatable to an EMF model entity's attribute.

Relationship mapping

Contained classes are wrapped inside their compositing class, with additional tags denoting their domain relationship names. Other type of relationships, i.e., bi-directional references with source and target domain roles, are serialized differently. Similar to the M2 layer, target classes of relationships are referenced through monikers, their name assembled as follows: <class_name>Moniker. If such a class has an attribute other than their GUID attribute set as their naming attribute (i.e., `IsMonikerKey`), the value of this naming attribute, which must be unique, is used for further referencing the correct target classes. Examples can be seen in Figure 8.4a, where `children` relationships contain `manMoniker` and `womanMoniker` types referencing the corresponding class entities by their naming attribute `name` in addition to the root element `familyTreeModel's` unique GUID. Domain relationships, that were transformed to EClasses, are flagged specifically, as the model transformator has to do a class lookup and add two additional references to solve the "Relationship as a Class" functionality correctly. Depending on the `UseFullForm` and `OmitElement` attributes for the XML serialization behavior defined on the M2 layer, the relationships have to be transformed differently, analyzed and detailed later in subsequent sections.

Inheritance mapping

As the transformed Ecore metamodel cannot contain any multiple inheritance structures, this makes it easier to find the target element as opposed to their transformed single inheritances from EMF to MSDKVS.

Additional information

As most of the features are directly translated to Ecore features, except having domain relationships used as classes with attributes or inside a domain relationship inheritance, additional information needed to correctly transform are kept manageable and at a minimum.

Regarding the transformation code itself, following code sections are executed sequentially in order to retrieve the transformed result, i.e., the serialized Ecore model:

1. **Setting the transformation direction:** For determining the direction from which platform the model has to be transformed the first argument of the program has to be `MSDKVS2EMF`.
2. **Loading the file contents:** The transformation's generated Ecore metamodel file, the MSDKVS XMI source model file and the mapping file are loaded into corresponding structures. An empty Ecore `Resource` is created based on the metamodel content and the resulting `EPackage` element retrieved and pinned for finding the correct EClass elements later on after being registered as a daynamic package in the `EPackage Registry Instance`. The mapping file, based on the JSON serialization format, is deserialized with the help of the *FasterXML Jackson* [1] library into a corresponding data structure, identical to the one being generated inside the C# code portion, i.e., the M2 transformator.
3. **Deserializing the MDKVS model:** The MSDKVS model file, saved as an XMI resource, is deserialized into a tree of `Elements`, retrieved from the XML tags and containing key-value pairs resembling the tag's attributes, if present. These

[1] `https://github.com/FasterXML/jackson`, last accessed on 04.01.2023

tag elements are then traversed and used for looking up the target Ecore entity, be it either an EReference or an EClass, via the mapping info generated in the M2 step of the transformator, if a suitable candidate has been found.

4. **Transformation of the root element:** The root element has to be the first node in the Element tree serialized from the model's XMI contents. This element, as well as each subsequent element, is matched against the mapping information, the root element being a `ClassMapping`. If a mapping corresponding to the current tag's name has been found by comparing the mapping's `From` value, this `AbstractMapping` is then parsed and used for creating the corresponding `EObject` by looking up its `To` value inside the generated `EPackage` container. If an EClass has been retrieved, it is transformed to a unique EObject by using the `EFactoryInstance's` *create* method with this class.

5. **Transformation of nodes:** As our source model data is only comprised of elements resembling the model's XML tags and their attributes, the traversal of the content differs greatly from the other transformation direction, as no element can be automatically distinguished out of the box. As mentioned before, every Element object is checked against the mapping data, comparing its `From` and `To` parameters. As MSDKVS offers different identifying strategies based on the `IsMonikerKey` flag of the domain properties of a domain class, a separate table is maintained, containing the created EObjects and their moniker key values. This is done for every node of the model file. Each element is flagged if it has been traversed already as to avoid duplicate EObject creation. Each class element's key-value pairs are iterated in order to transform them to corresponding EAttribute values. The correct data types and enumeration literals are looked up accordingly.

6. **Transformation of relationships:** First, the mapping information of the reference itself is retrieved based on the tag's name, which was saved as the `From` value of the reference mapping. Based on the different flags that indicate if elements were skipped or not inside the model's XML file thus resulting in fewer or more Element tags, the Element tree is further traversed in order to get the correct mapping source. A list of references that could not be solved immediately, as the referencing classes may not have been transformed yet, is maintained and, after further classes have been added as EObjects, checked for references that can now be properly added as both parts are available. Based on the reference mapping, an EReference object is created with source and target EClasses retrieved by the ClassMappings of the "Source" and "Target" parameters of said mapping information.

7. **Saving the Ecore model file:** The resulting Ecore model is saved based on the identified file ending inside the mapping information and the root EObject is added to an EMF `Resource` object's contents and afterwards saved accordingly to an output file.

8.3 Examples

The following examples, one for each direction, based on the familytree metamodels created in Chapter 3 and described in the metamodel transformation sample in Section 6.2, should give an insight on how the models change when put through the M1 transformator, how the mapping file is used and how the special serialization characteristics of each platform are considered accordingly. The example contains the most prominent abstract syntax capabilities, inheritance relationships, attributes, containment relationships and bi-directional relationships. On top of that it also shows how references to classes that are themselves part of a containment reference are handled.

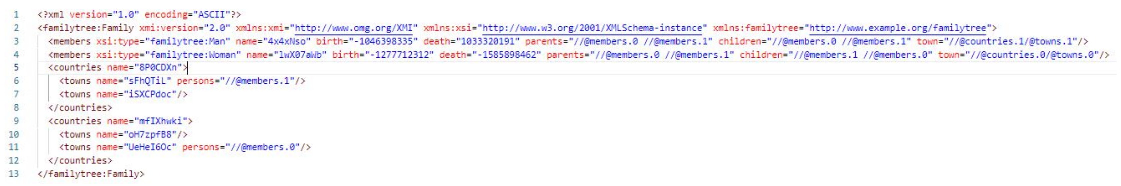
8.3.1 EMF to MSDKVS

The Ecore metamodel used for this direction's transformation is the one created based on the UML class diagram mentioned in Section 3.2. The Ecore metamodel had to be adapted, adding an additional containment reference from family to countries, and changing the type of relationship of `members` to composition, in order to have a root element containing both country and person entities for cross referencing on the M1 layer. This root element is then transformed accordingly to a root element in MSDKVS, thus enabling the correct model transformation setup. Regarding MSDKVS' serialization of models, the transformation on the M2 layer results in the following default values for the XML serialization, thus setting the `OmitElement` attribute and `UseFullForm` attribute for domain relationships to "false" and the `Representation` value to an empty string (i.e., defaulting to "Attribute") instead of "Element" for domain properties to best resemble the representation of models on the EMF side. As the API documentation states, the `OmitElement` should always be set to "false" in case there is more than one relationship between source and target class present, thus being on the saver side.

For demonstration purposes, the source model for the Ecore metamodel is generated using the model generator used in the next chapter for evaluation. Figure 8.1 shows the generated model file in EMF and its transformed equivalent for MSDKVS. Figure 8.2 shows an excerpt of the resulting mapping file from the M2 transformator, depicting the **Person** class info for the M1 transformator.

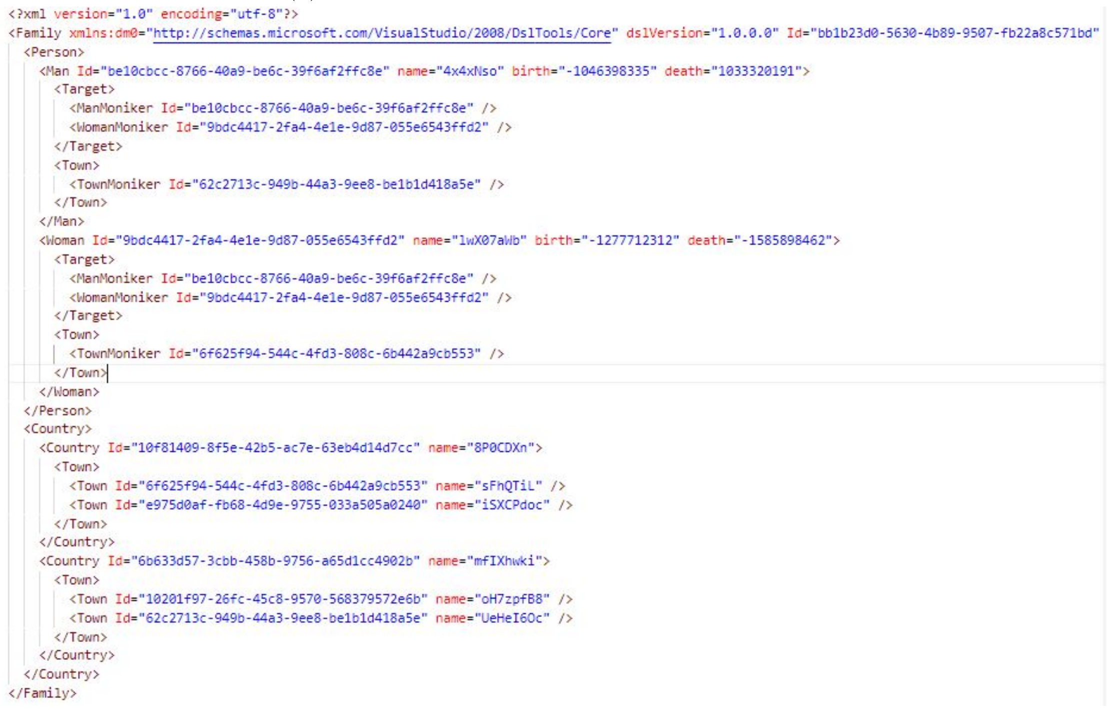
8.3.2 MSDKVS to EMF

The familytree DSL and its model used for the transformation are taken from Section 6.2.1. XMI structures of both EMF and MSDKVS models are shown in Figure 8.4. Depending on the applied serialization techniques mentioned in Section 8.2, the handling of attributes and references can differ. As mentioned before, omitting elements and using the full form defaults their respective attributes to "false" when transforming from EMF to MSDKVS, but these values have to be considered when transforming in the other direction. Therefore, each transformed domain element on the M2 layer saves this additional information inside their relationship or attribute mapping in the generated mapping.json file that is used as input for the M1 transformation, signaling what kind of XML serialization



```
<?xml version="1.0" encoding="ASCII"?>
<familytree:Family xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:familytree="http://www.example.org/familytree">
  <members xsi:type="familytree:Man" name="4x4xNso" birth="-1046398335" death="1033320191" parents="//@members.0 //@members.1" children="//@members.0 //@members.1" town="//@countries.1/@towns.1"/>
  <members xsi:type="familytree:Woman" name="1wX07aWb" birth="-1277712312" death="-1585898462" parents="//@members.0 //@members.1" children="//@members.1 //@members.0" town="//@countries.0/@towns.0"/>
  <countries name="8P0CDXn">
    <towns name="sFhQTiL" persons="//@members.1"/>
    <towns name="iSXCPdoc"/>
  </countries>
  <countries name="mfIXhwki">
    <towns name="oH7zpfB8"/>
    <towns name="UeHeI6Oc" persons="//@members.0"/>
  </countries>
</familytree:Family>
```

(a) Adapted source familytree model in EMF



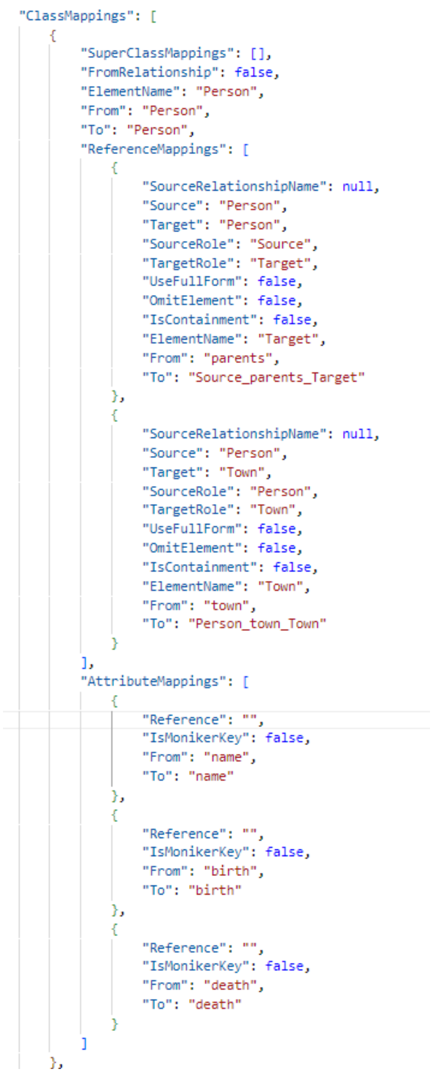
```
<?xml version="1.0" encoding="utf-8"?>
<Family xmlns:dm0="http://schemas.microsoft.com/VisualStudio/2008/DslTools/Core" dslVersion="1.0.0.0" Id="bb1b23d0-5630-4b89-9507-fb22a8c571bd"
  <Person>
    <Man Id="be10cbcc-8766-40a9-be6c-39f6af2ffc8e" name="4x4xNso" birth="-1046398335" death="1033320191">
      <Target>
        <ManMoniker Id="be10cbcc-8766-40a9-be6c-39f6af2ffc8e" />
        <WomanMoniker Id="9bdc4417-2fa4-4e1e-9d87-055e6543ffd2" />
      </Target>
      <Town>
        <TownMoniker Id="62c2713c-949b-44a3-9ee8-be1b1d418a5e" />
      </Town>
    </Man>
    <Woman Id="9bdc4417-2fa4-4e1e-9d87-055e6543ffd2" name="1wX07aWb" birth="-1277712312" death="-1585898462">
      <Target>
        <ManMoniker Id="be10cbcc-8766-40a9-be6c-39f6af2ffc8e" />
        <WomanMoniker Id="9bdc4417-2fa4-4e1e-9d87-055e6543ffd2" />
      </Target>
      <Town>
        <TownMoniker Id="6f625f94-544c-4fd3-808c-6b442a9cb553" />
      </Town>
    </Woman>
  </Person>
  <Country>
    <Country Id="10f81409-8f5e-42b5-ac7e-63eb4d14d7cc" name="8P0CDXn">
      <Town>
        <Town Id="6f625f94-544c-4fd3-808c-6b442a9cb553" name="sFhQTiL" />
        <Town Id="e975d0af-fb68-4d9e-9755-033a505a0240" name="iSXCPdoc" />
      </Town>
    </Country>
    <Country Id="6b633d57-3cbb-458b-9756-a65d1cc4902b" name="mfIXhwki">
      <Town>
        <Town Id="10201f97-26fc-45c8-9570-568379572e6b" name="oH7zpfB8" />
        <Town Id="62c2713c-949b-44a3-9ee8-be1b1d418a5e" name="UeHeI6Oc" />
      </Town>
    </Country>
  </Country>
</Family>
```

(b) Transformed familytree model in MSDKVS

Figure 8.1: Model Transformation Example MSDKVS to EMF, MSDKVS model (top) and transformed result in EMF (bottom)

the domain element uses and adapting the deserialization of the model file into data structures used to traverse each element, skipping not needed containment reference structures or attribute tags.

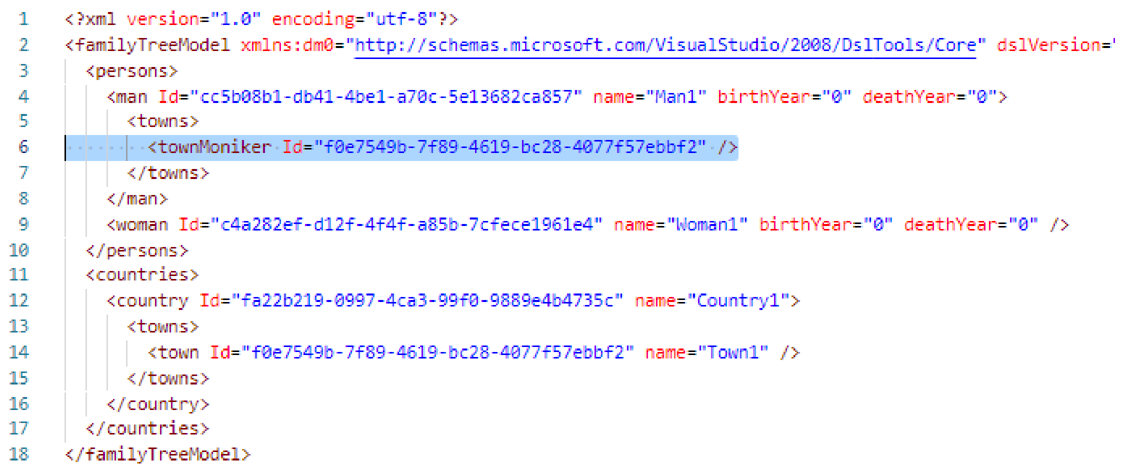
The XML serialization for DomainProperties contains a settable boolean flag named `IsMonikerKey`, defining how to reference the attributes inside the models' schema. If no domain property with said flag is set to true, then the entity's GUID is used for referencing it from other eligible entities, called *Id Moniker* cross referencing[35] . As opposed to this Id moniker, when a designated domain property is flagged, then it is called *Qualified Key Moniker* cross referencing, where the automatic validation ensures the uniqueness of this property's values. Figure 8.3 shows the differences in model serialization in terms of this behavior. The **Town** element which is also part of a containment reference inside **Country** is targeted by its unique GUID with no moniker key having been flagged. Different outcomes based on containment references, domain classes' named attributes and XML serialization of these attributes can be seen here.



```
"ClassMappings": [
    {
        "SuperClassMappings": [],
        "FromRelationship": false,
        "ElementName": "Person",
        "From": "Person",
        "To": "Person",
        "ReferenceMappings": [
            {
                "SourceRelationshipName": null,
                "Source": "Person",
                "Target": "Person",
                "SourceRole": "Source",
                "TargetRole": "Target",
                "UseFullForm": false,
                "OmitElement": false,
                "IsContainment": false,
                "ElementName": "Target",
                "From": "parents",
                "To": "Source_parents_Target"
            },
            {
                "SourceRelationshipName": null,
                "Source": "Person",
                "Target": "Town",
                "SourceRole": "Person",
                "TargetRole": "Town",
                "UseFullForm": false,
                "OmitElement": false,
                "IsContainment": false,
                "ElementName": "Town",
                "From": "town",
                "To": "Person_town_Town"
            }
        ],
        "AttributeMappings": [
            {
                "Reference": "",
                "IsMonikerKey": false,
                "From": "name",
                "To": "name"
            },
            {
                "Reference": "",
                "IsMonikerKey": false,
                "From": "birth",
                "To": "birth"
            },
            {
                "Reference": "",
                "IsMonikerKey": false,
                "From": "death",
                "To": "death"
            }
        ]
    },
```

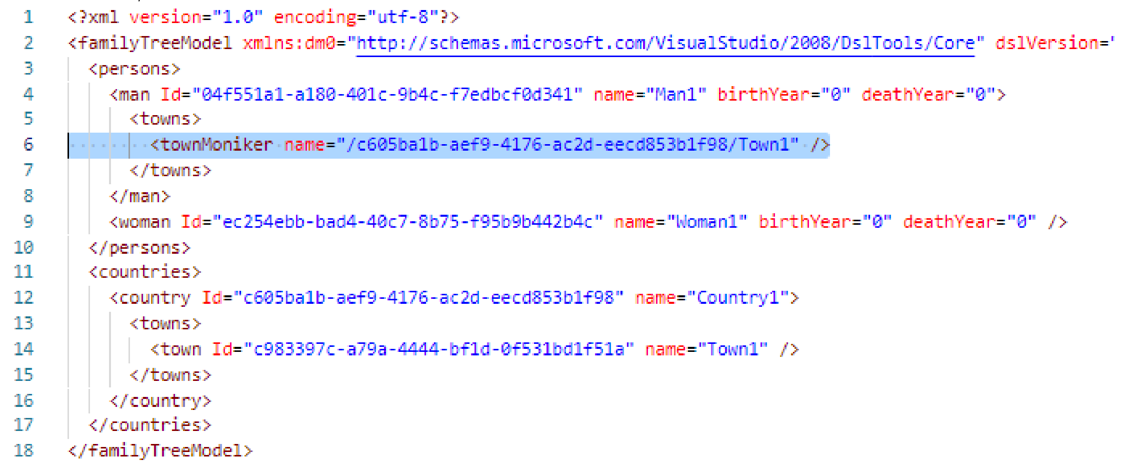
Figure 8.2: Example class mapping of class **Person** resulting from the M2 transformation

When a domain relationship has been defined as a domain class and contains attributes, the `UseFullForm` attribute is automatically set to "true" to resemble class entities, meaning the model content is also affected.



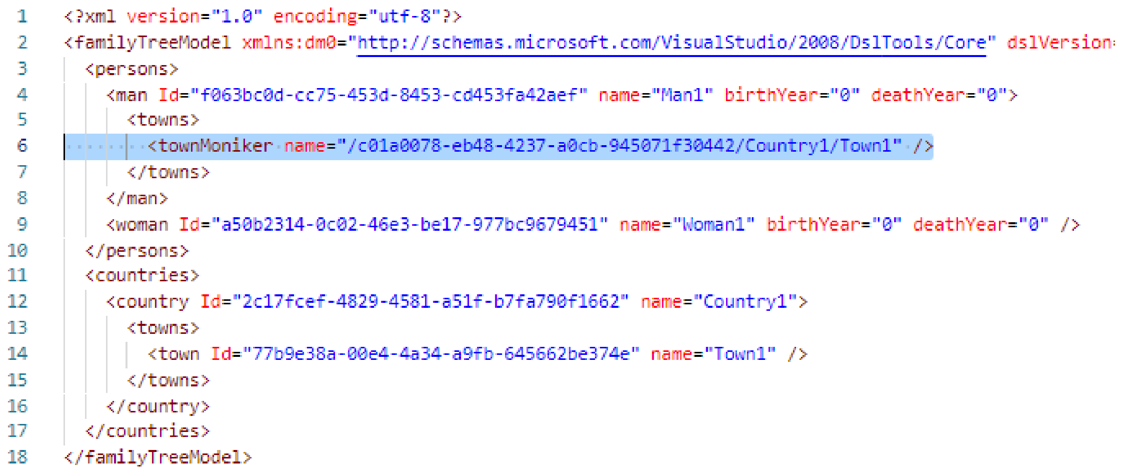
```xml
<?xml version="1.0" encoding="utf-8"?>
<familyTreeModel xmlns:dm0="http://schemas.microsoft.com/VisualStudio/2008/DslTools/Core" dslVersion='
  <persons>
    <man Id="cc5b08b1-db41-4be1-a70c-5e13682ca857" name="Man1" birthYear="0" deathYear="0">
      <towns>
        <townMoniker Id="f0e7549b-7f89-4619-bc28-4077f57ebbf2" />
      </towns>
    </man>
    <woman Id="c4a282ef-d12f-4f4f-a85b-7cfece1961e4" name="Woman1" birthYear="0" deathYear="0" />
  </persons>
  <countries>
    <country Id="fa22b219-0997-4ca3-99f0-9889e4b4735c" name="Country1">
      <towns>
        <town Id="f0e7549b-7f89-4619-bc28-4077f57ebbf2" name="Town1" />
      </towns>
    </country>
  </countries>
</familyTreeModel>
```

(a) Simple familytree model having `IsMonikerKey` for name attributes of Town and Country set to false / null



```xml
<?xml version="1.0" encoding="utf-8"?>
<familyTreeModel xmlns:dm0="http://schemas.microsoft.com/VisualStudio/2008/DslTools/Core" dslVersion='
  <persons>
    <man Id="04f551a1-a180-401c-9b4c-f7edbcf0d341" name="Man1" birthYear="0" deathYear="0">
      <towns>
        <townMoniker name="/c605ba1b-aef9-4176-ac2d-eecd853b1f98/Town1" />
      </towns>
    </man>
    <woman Id="ec254ebb-bad4-40c7-8b75-f95b9b442b4c" name="Woman1" birthYear="0" deathYear="0" />
  </persons>
  <countries>
    <country Id="c605ba1b-aef9-4176-ac2d-eecd853b1f98" name="Country1">
      <towns>
        <town Id="c983397c-a79a-4444-bf1d-0f531bd1f51a" name="Town1" />
      </towns>
    </country>
  </countries>
</familyTreeModel>
```

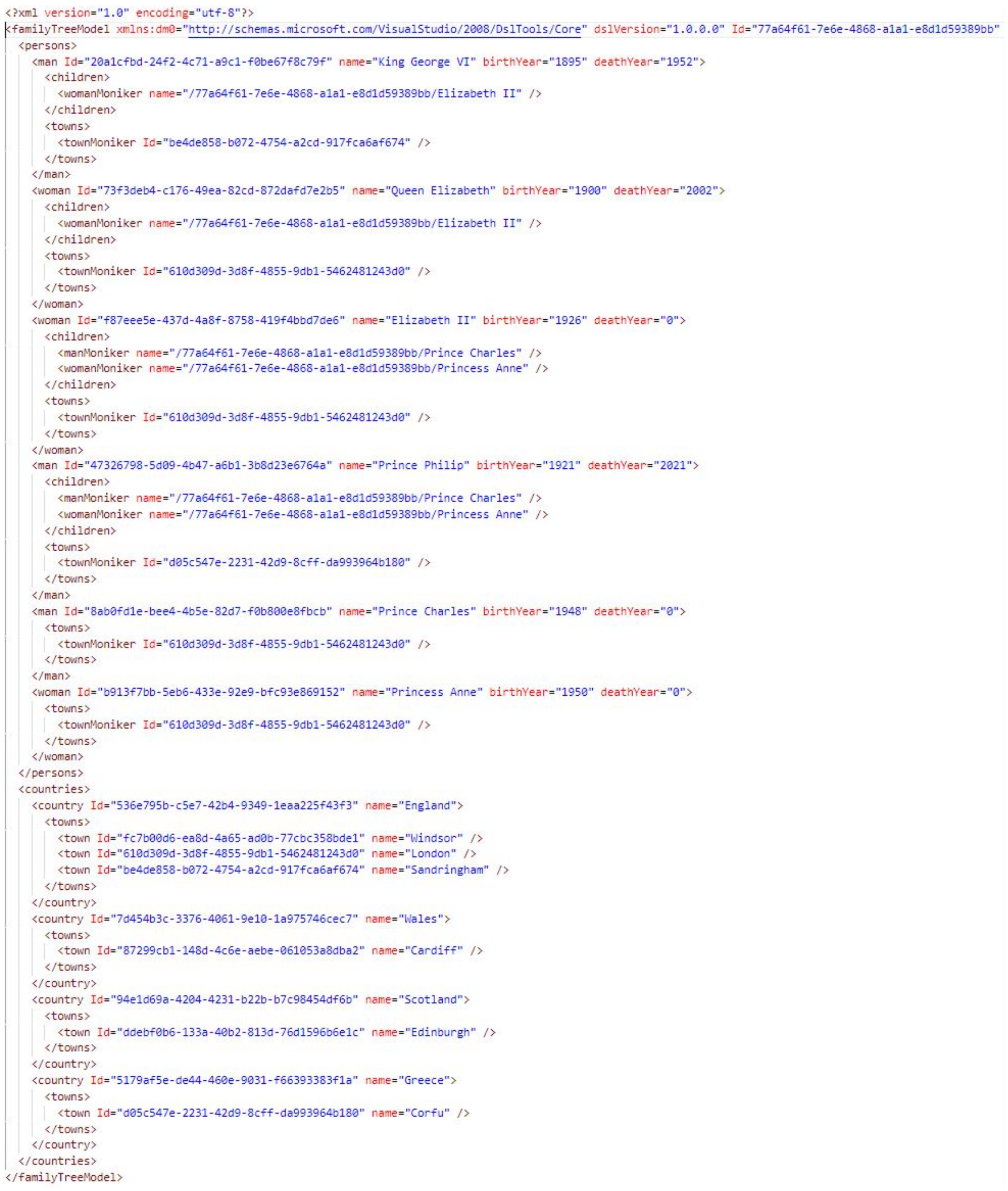
(b) Simple familytree model having `IsMonikerKey` for name attribute of Town set to true and for Country set to false



```xml
<?xml version="1.0" encoding="utf-8"?>
<familyTreeModel xmlns:dm0="http://schemas.microsoft.com/VisualStudio/2008/DslTools/Core" dslVersion=
  <persons>
    <man Id="f063bc0d-cc75-453d-8453-cd453fa42aef" name="Man1" birthYear="0" deathYear="0">
      <towns>
        <townMoniker name="/c01a0078-eb48-4237-a0cb-945071f30442/Country1/Town1" />
      </towns>
    </man>
    <woman Id="a50b2314-0c02-46e3-be17-977bc9679451" name="Woman1" birthYear="0" deathYear="0" />
  </persons>
  <countries>
    <country Id="2c17fcef-4829-4581-a51f-b7fa790f1662" name="Country1">
      <towns>
        <town Id="77b9e38a-00e4-4a34-a9fb-645662be374e" name="Town1" />
      </towns>
    </country>
  </countries>
</familyTreeModel>
```

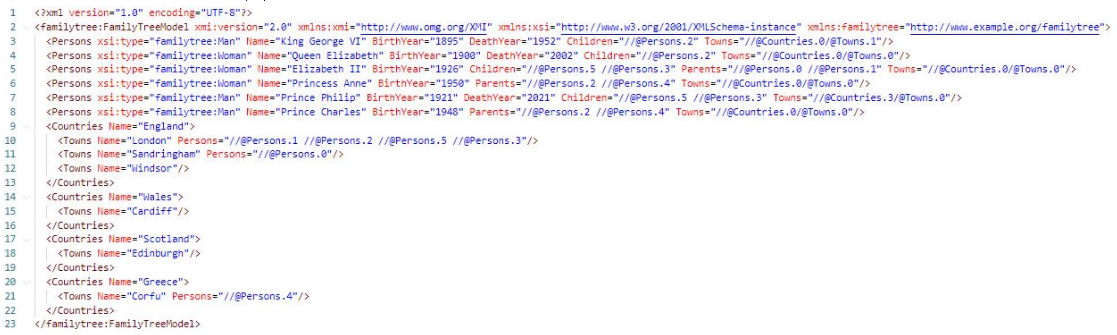
(c) Simple familytree model having `IsMonikerKey` for name attributes of Town and Country set to true

Figure 8.3: Examples for `IsMonikerKey` effects for cross referencing model entities



```xml
<?xml version="1.0" encoding="utf-8"?>
<familyTreeModel xmlns:dm0="http://schemas.microsoft.com/VisualStudio/2008/DslTools/Core" dslVersion="1.0.0.0" Id="77a64f61-7e6e-4868-a1a1-e8d1d59389bb"
  <persons>
    <man Id="20a1cfbd-24f2-4c71-a9c1-f0be67f8c79f" name="King George VI" birthYear="1895" deathYear="1952">
      <children>
        <womanMoniker name="/77a64f61-7e6e-4868-a1a1-e8d1d59389bb/Elizabeth II" />
      </children>
      <towns>
        <townMoniker Id="be4de858-b072-4754-a2cd-917fca6af674" />
      </towns>
    </man>
    <woman Id="73f3deb4-c176-49ea-82cd-872dafd7e2b5" name="Queen Elizabeth" birthYear="1900" deathYear="2002">
      <children>
        <womanMoniker name="/77a64f61-7e6e-4868-a1a1-e8d1d59389bb/Elizabeth II" />
      </children>
      <towns>
        <townMoniker Id="610d309d-3d8f-4855-9db1-5462481243d0" />
      </towns>
    </woman>
    <woman Id="f87eee5e-437d-4a8f-8758-419f4bbd7de6" name="Elizabeth II" birthYear="1926" deathYear="0">
      <children>
        <manMoniker name="/77a64f61-7e6e-4868-a1a1-e8d1d59389bb/Prince Charles" />
        <womanMoniker name="/77a64f61-7e6e-4868-a1a1-e8d1d59389bb/Princess Anne" />
      </children>
      <towns>
        <townMoniker Id="610d309d-3d8f-4855-9db1-5462481243d0" />
      </towns>
    </woman>
    <man Id="47326798-5d09-4b47-a6b1-3b8d23e6764a" name="Prince Philip" birthYear="1921" deathYear="2021">
      <children>
        <manMoniker name="/77a64f61-7e6e-4868-a1a1-e8d1d59389bb/Prince Charles" />
        <womanMoniker name="/77a64f61-7e6e-4868-a1a1-e8d1d59389bb/Princess Anne" />
      </children>
      <towns>
        <townMoniker Id="d05c547e-2231-42d9-8cff-da993964b180" />
      </towns>
    </man>
    <man Id="8ab0fd1e-bee4-4b5e-82d7-f0b800e8fbcb" name="Prince Charles" birthYear="1948" deathYear="0">
      <towns>
        <townMoniker Id="610d309d-3d8f-4855-9db1-5462481243d0" />
      </towns>
    </man>
    <woman Id="b913f7bb-5eb6-433e-92e9-bfc93e869152" name="Princess Anne" birthYear="1950" deathYear="0">
      <towns>
        <townMoniker Id="610d309d-3d8f-4855-9db1-5462481243d0" />
      </towns>
    </woman>
  </persons>
  <countries>
    <country Id="536e795b-c5e7-42b4-9349-1eaa225f43f3" name="England">
      <towns>
        <town Id="fc7b00d6-ea8d-4a65-ad0b-77cbc358bde1" name="Windsor" />
        <town Id="610d309d-3d8f-4855-9db1-5462481243d0" name="London" />
        <town Id="be4de858-b072-4754-a2cd-917fca6af674" name="Sandringham" />
      </towns>
    </country>
    <country Id="7d454b3c-3376-4061-9e10-1a975746cec7" name="Wales">
      <towns>
        <town Id="87299cb1-148d-4c6e-aebe-061053a8dba2" name="Cardiff" />
      </towns>
    </country>
    <country Id="94e1d69a-4204-4231-b22b-b7c98454df6b" name="Scotland">
      <towns>
        <town Id="ddebf0b6-133a-40b2-813d-76d1596b6e1c" name="Edinburgh" />
      </towns>
    </country>
    <country Id="5179af5e-de44-460e-9031-f66393383f1a" name="Greece">
      <towns>
        <town Id="d05c547e-2231-42d9-8cff-da993964b180" name="Corfu" />
      </towns>
    </country>
  </countries>
</familyTreeModel>
```

(a) Excerpt from source familytree model in MSDKVS



```xml
<?xml version="1.0" encoding="UTF-8"?>
<familytree:FamilyTreeModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:familytree="http://www.example.org/familytree">
  <Persons xsi:type="familytree:Man" Name="King George VI" BirthYear="1895" DeathYear="1952" Children="//@Persons.2" Towns="//@Countries.0/@Towns.1"/>
  <Persons xsi:type="familytree:Woman" Name="Queen Elizabeth" BirthYear="1900" DeathYear="2002" Children="//@Persons.2" Towns="//@Countries.0/@Towns.0"/>
  <Persons xsi:type="familytree:Woman" Name="Elizabeth II" BirthYear="1926" Children="//@Persons.5 //@Persons.3" Parents="//@Persons.0 //@Persons.1" Towns="//@Countries.0/@Towns.0"/>
  <Persons xsi:type="familytree:Woman" Name="Princess Anne" BirthYear="1950" Parents="//@Persons.2 //@Persons.4" Towns="//@Countries.0/@Towns.0"/>
  <Persons xsi:type="familytree:Man" Name="Prince Philip" BirthYear="1921" DeathYear="2021" Children="//@Persons.5 //@Persons.3" Towns="//@Countries.3/@Towns.0"/>
  <Persons xsi:type="familytree:Man" Name="Prince Charles" BirthYear="1948" Parents="//@Persons.2 //@Persons.4" Towns="//@Countries.0/@Towns.0"/>
  <Countries Name="England">
    <Towns Name="London" Persons="//@Persons.1 //@Persons.2 //@Persons.5 //@Persons.3"/>
    <Towns Name="Sandringham" Persons="//@Persons.0"/>
    <Towns Name="Windsor"/>
  </Countries>
  <Countries Name="Wales">
    <Towns Name="Cardiff"/>
  </Countries>
  <Countries Name="Scotland">
    <Towns Name="Edinburgh"/>
  </Countries>
  <Countries Name="Greece">
    <Towns Name="Corfu" Persons="//@Persons.4"/>
  </Countries>
</familytree:FamilyTreeModel>
```

(b) Transformed familytree model in EMF

Figure 8.4: Model Transformation Example MSDKVS to EMF, MSDKVS model (top) and transformed result in EMF (bottom)

CHAPTER 9

Evaluation of M3 Bridge

In addition to the feasibility study of the first part of the transformation (see Chapter 7), this chapter considers the transformation implementation, including the aforementioned transformation of models, in terms of investigating the complexity and also the degree of completion, i.e., if and how all possible definitions can be transformed.

9.1 Model Generation

For testing the transformation approach on the M1 layer, a random model generator customizable with several parameters, presented in [2], is used for generating different models based on an Ecore metamodel. A selection of previously transformed metamodels is used, with the generated *mapping.json* file, for evaluating the model transformation approach for the direction **EMF2MSDKVS**. For the direction **MSDKVS2EMF**, the underlying models have been generated manually as no generator of models is available.

For the Ecore model generator, a variety of optional parameters can be used to generate models based on one's own needs, like the average size of a model regarding its elements, an average number of reference per class element as well as attribute value lengths. For this thesis' purposes and for better readability of the generated model files, an **average size of 20** was used to make the models still readable and understandable for humans, as well as a **reference degree of 4** per object and a **average variable length of 8** for attributes.

After having tried to generate some models based on the selected source Ecore metamodels, some limitations could already be recognized. The input metamodel file must not contain more than one EPackage definition, or else the generator's internal diagnostic functionalities reports an error that additional metamodels have to be explicitly referenced. Additionally, many of the source Ecore metamodels, as they were themselves generated, did not have their EPackages' `namespace prefix`, `namespace URI` and `instance` `type name` defined, leading to errors inside the generator as well. These errors had to be fixed manually, thus being able to generate the amount of models used for the M1 evaluation. If a metamodel had additional EPackages, typically the *PrimitiveTypes* EPackage, existing references had to be resolved manually as well, resulting in one EPackage in general for proper usage. These changes made to the source metamodels were checked beforehand for validity, as to not use invalid metamodels for the purpose of evaluating the model transformation.

9.2 Transformation Results

Following Tables 9.1 and 9.2 list the selected transformed metamodels used in the M2 transformator previously discussed in Chapter 9, the number of manually created or generated models based on these results and the execution results of the M1 transformator. Each category identified in Section 8.2 is listed as a separate column for structured evaluation and the numbers depicted show how many of the generated models result in valid transformed models on the target platform. Next, each column is analyzed individually, explaining its purpose and results regarding the source and their transformed models, listing found errors or unintended behavior and possible solutions and improvements for future releases. Afterwards, the limitations of the M1 transformator based on the *Ecore Reflection API* for **EMF2MSDKVS** and *JAXB* library for serializing and deserializing the input models of MSDKVS, used for the direction **MSDKVS2EMF**, and output models are discussed. As the semantic evaluation regarding the interactivity of models based on the M2 transformation results has already been done in Chapter 7, the following results take only the validity of the transformed models into account, especially regarding the validity of the mapping strategies discussed in Section 8.2 in addition to the comparison of their graphical representation, if available, depicted in column "Layout". If no graphical concrete syntax is available, the model explorer's embedding tree is used for investigating the model contents. Reference relationships are not available inside this tree editor and are only visible if a *Connector* has been defined for these types of relationships, therefore they cannot be checked for validity inside the modeling run time in Visual Studio. As currently no model generator in MSDKVS is available, basic reference links are added inside the model's XMI representation file in order to check the validity of those.

9.2.1 EMF to MSDKVS

A subset of eight Ecore metamodels was chosen based on the previous quantitative metrics distribution and qualitative analysis. The following Table 9.1 displays the results of the validity checks for different criteria in addition to the layout of the transformed Sirius components in terms of coloring, labeling and more. Each metamodel was used to generate ten model files. These files were then transformed using the M1 transformation of the M3B.

MM	**Validity**					
	#	**Class**	**Attribute**	**Rel.**	**Inh.**	**Layout**
Ant	10	10/10	10/10	10/10	10/10	-
ATL	10	10/10	10/10	10/10	10/10	-
behaviortree	10	10/10	10/10	10/10	10/10	10/10
c_sharp	10	10/10	10/10	0/10	0/10	-
HAL	10	10/10	10/10	10/10	10/10	-
KDM	10	10/10	10/10	10/10	10/10	-
poosl	10	10/10	10/10	10/10	10/10	10/10
sensorProject	10	10/10	10/10	10/10	10/10	10/10

Table 9.1: Model transformation results for transformation direction EMF to MSDKVS.

- **Class transformation:** For every model used, the class transformation resulted in correctly mapped class entities on the target platform, having the correct naming based on the mapping information created in the M2 transformator. If a class had been duplicated and renamed by applying the strategies on the M2 transformator as a result of the expansion strategy, transforming multiple inheritance structures to single inheritance structures, the correct type could be identified accordingly.

- **Attribute transformation:** Same as for the class transformation, every class entity contained the correctly mapped attributes, correctly mapped data types, default values and enumeration literals.

- **Relationship transformation:** Regarding the validity of the relationship transformation part, only the **c_sharp** metamodel proved to be invalid, as it contains multiple inheritance structures nested within multiple inheritance structures, thus resulting in wrongly resolved references during the M1 transformation. Regarding the containment, simple and bi-directional references, the transformed MSDKVS model's proved to be valid in terms of syntax and semantic.

- **Inheritance transformation:** As mentioned before, inheritance relationships regarding nested multiple inheritances was the only recognized error-prone aspect of the M1 transformation. As nearly all of the metamodels used in the M2 transformator did not use such complex multiple inheritance trees, one can infer that these occur very rarely. Nonetheless, these errors will be looked at accordingly and, possibly with the help of enhanced mapping information from the M2 transformator or additional data structures maintained inside the M1 transformator for storing inheritance depths.

- **Layout behavior:** As for the layout, i.e., the correct and identical display of model content on the modeling canvas inside the platform's run time instances, half of the models had a Sirius VSMs available, all of them showing valid and identical model shapes based on the transformed metamodel files. If there was no graphical representation available, the inherent tree structures were used for comparing both model contents, which is reflected in the abstract syntax information listed in the other columns.

9.2.2 MSDKVS to EMF

As opposed to EMF, there is no model generator currently available in MSDKVS. In the scope of this thesis' M1 transformator evaluation, models for the selected DSLs have been created manually, best resembling the mentioned parameterized generation of models on the EMF side. Thus, as creating different models based on the transformed metamodels in MSDKVS is a time-consuming task, only 4 models per DSL have been created to show the feasibility and validity of the M1 transformator.

Table 9.2 shows the selected DSLs, the amount of valid models for different validity criteria and also the three important flags mentioned before that influence how the model entities are serialized inside the XML files. Typically, `OmitElement` has not been set in any of the looked at DSL definitions. As `UseFullForm` is per default set to true when a domain relationship is created, most DSLs use this feature. In the subset of DSLs selected for M1 evaluation, only one DSL, namely **hostdesigner**, did not set any domain relationships to use their full form representation. Regarding serializing attributes as separate XML elements instead of a domain class or domain relationship's attribute by setting the `Representation` attribute for *XmlPropertyData* to "Element", the **candle** DSL was the only language making use of this feature.

DSL	**Validity**								
	#	**Class**	**Attribute**	**Rel.**	**Inh.**	**Layout**	**OE**[1]	**UFF**[2]	**ER**[3]
agilemodeler	4	4/4	4/4	4/4	4/4	4/4	✗	✓	✗
candle	4	4/4	4/4	4/4	4/4	4/4	✗	✓	✓
fwk_dsl	4	4/4	4/4	4/4	4/4	4/4	✗	✓	✗
generatorlanguage	4	4/4	4/4	4/4	4/4	4/4	✗	✓	✗
hostdesigner	4	4/4	4/4	4/4	4/4	4/4	✗	✗	✗
mbrdcmdmi	4	4/4	4/4	4/4	4/4	4/4	✗	✓	✗
mobiledsl	4	4/4	4/4	4/4	4/4	4/4	✗	✓	✗
spllanguage	4	4/4	4/4	4/4	4/4	4/4	✗	✓	✗

[1] Indicator for containing at least one `OmitElement` *XmlRelationshipData*
[2] Indicator for containing at least one `UseFullForm` *XmlRelationshipData*
[3] Indicator for containing at least one *XmlPropertyData* data having `Representation` attribute set to "Element"

Table 9.2: Model transformation results for transformation direction MSDKVS to EMF.

- **Class transformation:** Regarding the transformation of domain class entities to EClass objects, no significant differences or error-prone behavior occurred during the M1 transformation and during the final examination.
- **Attribute transformation:** Every DSL except **candle** uses the default serialization technique for attributes, i.e., identical to an XML tag attribute. **Candle** DSL uses customized serialization data for one or more domain properties, such

that the model attributes are serialized as subsidiary XML tags for the domain class or domain relationship. The M1 transformation works for both ways and all transformed models were interpreted and displayed correctly.

- **Relationship transformation:** The MSDKVS relationship serialization is primarily influenced by the `OmitElement` and `UseFullForm` flags attached to a domain relationship's XML data. The mapping information created in the M2 step attaches each of these flag values to the resulting reference mapping, thus being able to check the valid structure in the model deserialization to an element tree used for traversing and transforming to the correct representation on the EMF side. Regarding our input DSLs, typically, elements were not flagged as omitted, as this is the default setting. The **hostdesigner** DSL was the only one, were no full form of relationships was serialized, thus skipping the XML tags named after the domain relationship name itself. The M1 transformator works either way, correctly identifying and looking up the corresponding EReference based on the Source and Target values available through the Class Mappings retrieved from the Reference Mappings.

- **Inheritance transformation:** As MSDKVS only offers single BaseClass references from domain classes and even domain relationships, no complex structures as opposed to EMF's **c__sharp** metamodel were present in the source model collection used for the M1 transformation evaluation, therefore resulting in equal single inheritance ESuperType relations on the target side, making finding the correct type of an EClass easier, with or without having multiple nested inheritance relationships.

- **Layout behavior:** As opposed to EMF, MSDKVS elminates the requirement on defining graphical concrete syntax on top of the abstract syntax elements by integrating the necessary tools into the DSL creation interface, thus resulting in nearly every DSL found containing graphical syntax elements. This results in Sirius VSMs on the target side after transformation, making it also possible for every Ecore metamodel and model to validate the correct visualization properties. Regarding the input set of manually created models on top of selected DSLs, the M1 transformation produced satisfying and graphically similar results on the target side, with some minor changes regarding especially the sizes of produced shapes when the diagram representation is first opened, as these values are dynamically created inside the *.aird* files.

- **OmitElement behavior:** As no DSL contained domain relationships flagged by the "OmitElement" attribute, this behavior could not be evaluated entirely.

- **UseFullForm behavior:** As mentioned before, by default this is set to `true`, meaning that the references inside the model classes result in more subsequent XML tags. The M1 transformator adapts its deserialization of the model file based on the instructions received from finding the correct reference mapping.

- **ElementReference behavior:** While discussing the attribute transformation behavior and mentioning MSDKVS' different styles of serializing domain properties, both styles (as `Attribute` and as `Element`) resulted in valid target models correctly interpreted by the Ecore system.

9.3 Limitations

One major limitation is the requirement of always having an element from the abstract syntax definition acting as the root element, be it either in MSDKVS, where this requirement for the M2 transformator originated (see Chapter 6), or EMF, in order to read the model file correctly. The generator [2] used for generating a number of models based on Ecore metamodels, as stated, targets "... any non-abstract EClass without a required containing EReference ..." for a potential root element. This sometimes results in XMI model files with no concrete metamodel root element as the XML root tag, making transforming them impossible based on the current approach. When a model is manually designed in a runtime instance of a metamodel in Eclipse, one has to define the root element beforehand when the model file is created. Therefore, only metamodel instances in EMF where a designated root element could be extracted that is able to contain all other syntax elements were used for model generation and evaluation of the M1 transformator.

This also leads to the M1 transformation only being able to transform one model file at a time, thus no referencing model entities in other files, especially in EMF, is supported.

The other limitation, with regards to the evaluation procedure, was the unavailability of a model generator in MSDKVS. Although, with the help of the framework's API and the T4 templating engine, it would be possible to edit model files in the experimental runtime instances of Visual Studio programmatically and thus eventually realizing a rudimentary model generator, the evaluation mentioned above relied on manually creating the source models as the creation of such a model generator has not been done yet. This can be added to the list of future work regarding improvements upon the evaluation method.

CHAPTER 10

Conclusion

In the course of this scientific thesis, interoperability between the EMF and MSDKVS platforms by conceptualizing and implementing a bidirectional transformation bridge able to transform metamodels and their models defined within both platforms was established. The evaluation was conducted by choosing a representative set of publicly available metamodels and a set of generated or manually created models thereof.

By providing mapping rules also for the graphical notations available in both metamodeling platforms, it proved to be possible to transform means of graphically representing these elements of transformed metamodels to achieve not only syntactic but also visual interoperability.

A new conceptualizing approach in identifying common graphical concrete syntax features of metamodels based on the chosen platforms and abstracting them to possibly also fit other metamodel features as well has been done and applied in the form of additional mapping rules for transforming these new syntax elements. An in-depth look at the metamodeling platforms was given in the beginning, which also describes how these frameworks tackle the requirement of graphically representing metamodels and models in their environments respectively, later on used for the comparative analysis not only on the abstract syntax but also on a graphical layer.

Based on previous work, an interoperability approach between the formerly known DSL Tools for Visual Studio and EMF has been achieved through ATL transformations that uses a pivot KM3 metamodel to be transformed to and from for reusability has been analysed. This thesis shows the differences to the M3B methodological approach mentioned here and also uses the most current versions of the platforms, reviving, adapting and extending the knowledge made in the previous approach to fit the needs of today's modeling environments. Furthermore, an exhaustive analytical feedback is given, which also differs from the previous approach as there were no concrete statistical resources available to look into how feasible and valid the former approach was regarding the executability of the transformed metamodels and models in the target platforms, especially focusing on the core differences and unique abstract syntax features that both platforms contain, such as *Multiple Inheritance* in EMF and *DomainRelationship Inheritance* in MSDKVS.

Based on the results of this research process, a new M3 level-based bridge was proposed and its concrete implementation described for each direction (either from EMF to MSDKVS or MSDKVS to EMF). The evaluation was done using a representative set of distinct metamodels found within a variety of publicly available sources, mainly code repositories. By transforming and executing these metamodels, relevant data was filtered, e.g., class and reference counts in source and target metamodel, and their behavior was tested, such as deletion of containment references or inheritance structures to show semantic equivalence when working with both platforms. Most of the metamodels used in this scenario could be transformed and evaluated with regards to their executability and validity. Specific erroneous cases, if existent, were listed and possible solutions for future adaptations of the transformation code were given.

Based on the output on the M2 layer, the resulting files where then used to transform models from one framework into the other, giving a detailed insight on how this was achieved and the obstacles that had to be considered. Afterwards, the full workflow of the M3B was considered and evaluated based on a set of models conforming to a subset of metamodels used in the M2 step of the transformation, either manually created or generated through the means of available libraries.

10.1 Research Questions

Regarding the design science methodological approach of this work, following research questions were posed in Chapter 1. The answers to the relevant questions given during the M2 transformation evaluation in Chapter 7 are integrated and adapted with additional remarks regarding the M1 transformation to give a final answer to the initial questions suiting the M3B as a whole:

RQ1: What is the degree of interoperability between MSDKVS and EMF in terms of user interaction and framework behavior? The M3B proposed and implemented in the course of this thesis takes as input any available and valid Ecore metamodel and, if available, its Sirius VSM, or MSDKVS DSL and transforms them accordingly to equivalent representations on the target framework. For every metamodel used in the experiment, a verification process was followed through by opening the transformed metamodel in the target platform, checking for errors and applying the platform specific validation techniques. Afterwards, a selection of metamodels was used to look at a variety of important criteria in terms of syntax and interaction in particular.

Models that were either generated or manually created were used to analyze the model transformation of the M3B in terms of validity and interaction on the modeling canvases inside the platforms. For the M2 layer, a diverse set of collected metamodels from various sources was used to successfully show the feasibility of the M3B approach from a practical point of view. Combining M1 and M2 transformation, it can be concluded that interoperability in today's versions of both EMF and MSDKVS can be achieved, not only on an abstract, but on the concrete graphical syntax layer as well. The transformation approach resulting from this thesis shows that syntactical and semantic equivalence can be accomplished, even considering many platform-unique features like multiple inheritance, attributable domain relationships and even explicitly definable XML serialization techniques for the M1 layer that are not existent on other platforms. Simple metamodels regarding their class and relationship entities achieved near lossless transformation. More complex structures and features also turned out to be translatable correctly, by loosing as little original structural integrity as possible. The error cases identified only resulted in having complex inheritance hierarchies present in the EMF metamodel instances, making finding the correct element to use cumbersome and error-prone in the M1 transformator.

By extending the available mapping information file resulting from the M2 transformation with additional information, the transformation approach can be adapted and further improved upon in the future, possibly leading to even better results in terms of semantic equivalence on more metamodels or future framework updates.

RQ2: What are the quantitative differences and similarities of transformed metamodels and models in the target framework compared to the original? This question was partly answered in Chapter 7, where the M2 transformator was looked at in detail, comparing the input and output values and discussing the discovered differences from source to target framework. Many of the quantitative differences occur when a platform unique feature has been employed during the metamodel design phase, e.g., multiple inheritance structures, resulting in higher metric counts of different abstract and even graphical concrete syntax elements. With regards to the model transformation, these platform unique features also play a significant role in how the models are represented and the serialized model files are interpreted by the platforms' runtime environment. In MSDKVS, the file sizes are much larger in general than in EMF and Sirius combined, as they contain multiple cross referencing functionalities between abstract and concrete graphical syntax and serialization behavior. When regarding the M1 transformation, each element from the source environment can be mapped, using the mapping file from the M2 step, to its counterpart inside the target framework. The main and possibly only major difference deduced from analyzing the modeling capabilities in both platforms is the number of XML tags needed for representing, e.g., a containment reference, an inheritance relationship or other types of nested abstract syntax formations, as they add much more lines of code to MSDKVS and make it harder for a simple XML parser to generate structures that can be iterated effectively, in addition to the dynamically changing content of a model based on serialization configurations in MSDKVS.

An important part, that fortunately both frameworks employ, although differently, is the referencing of other modeling elements inside the serialized model file. EMF uses the @ declaration, whereas MSDKVS relies on the *moniker* type references using GUIDs or manually defined key attributes. But, as both methods are used for targeting model elements based on reference entities, a mapping could be easily established and implemented.

10.2 Future work

As this thesis provides a novel approach in regard to transform graphical concrete syntaxes and only lists possible factors of graphical concrete syntax elements contained either within MSDKVS or EMF, a further possible research direction could analyze the graphical representations available among a wider range of metamodeling platforms and compare them accordingly, adapting the proposed abstraction of recognized features from this thesis.

Furthermore, the realization of roundtrip transformations based on defined mapping tables could be relevant, where a source metamodel created in one metamodeling platform is first transformed to an equivalent metamodel in another metamodeling platform and afterwards, as such bridges are typically realized as bidirectional or two unidirectional transformations, the result of the first transformation is again transformed back to the original source metamodeling platform. Depending on special use cases and different functionalities implemented on these different platforms, the original source metamodel and the roundtrip metamodel usually will not be identical.

This information loss could be avoided with the help of additional transformation information logging created during the first transformation. Using the source and target metamodel together with the adapted mapping file for this thesis' M2 transformation log, it seems feasible to transform the target metamodel back into the source one, reconstructing the state before the first transformation, thereby achieving lossless round-trip interoperability between metamodeling platforms. This would be interesting for cases, where two teams of language designers each work on different platforms, meaning changes on the same metamodel made in one platform should immediately be reflected in the other and vice versa.

As mentioned in Chapter 8, the source models for the M1 transformation from MSDKVS to EMF were created manually. Possible future work, also in terms of better evaluating this thesis' M3B with arbitrary MSDKVS models, would be the implementation of a model generator for DSLs designed for today's version of MSDKVS, ideally in the same notion as has been done on the EMF side.

In addition, further research could be made in the direction of possibly resolving the limitations mentioned in Section 7.5 for the metamodel transformation and in Section 9.3 for the M1 transformation, respectively. Especially with regards to the additional Sirius operation structures and transforming the validation constraint query languages, like OCL, AQL, Acceleo and also external services and data, this would be an interesting consideration, as well as for dynamically changing shapes indicated by a traversal of valid entity states and based on different criteria, supplied either via external code or directly in the DSL schema, if supported.

Last but not least, the bridge's code and metamodels used in the evaluation will be deployed open source for the modeling community and the transformation bridge will be available as a usable service at: `http://me.big.tuwien.ac.at/`.

List of Figures

List of Tables

Acronyms

DSL Domain-specific language. 1

EMF Eclipse Modeling Framework. 1, 9, 13

MDSE Model-Driven Software Engineering. 14

MSDKVS Modeling SDK for Visual Studio. 13, 19, 42

VSM Viewpoint Specification Model. 17

Bibliography

[1] Emf api documentation overview. `https://download.eclipse.org/modeling/emf/emf/javadoc/2.4.2/org/eclipse/emf/ecore/package-summary.html`, last accessed on 23.12.2022.

[2] Model generator jar for ecore models. `https://modeling-languages.com/a-pseudo-random-instance-generator-for-emf-models/`, last accessed on 23.12.2022.

[3] Official sirius online documentation. `https://www.eclipse.org/sirius/doc/`, last accessed on 23.12.2022.

[4] Ö. Babur, L. Cleophas, and M. van den Brand. Hierarchical clustering of metamodels for comparative analysis and visualization. In A. Wasowski and H. Lönn, editors, *Modelling Foundations and Applications - 12th European Conference, ECMFA@STAF 2016, Vienna, Austria, July 6-7, 2016, Proceedings*, volume 9764 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2016.

[5] F. Basciani, J. D. Rocco, D. D. Ruscio, L. Iovino, and A. Pierantonio. Automated clustering of metamodel repositories. In S. Nurcan, P. Soffer, M. Bajec, and J. Eder, editors, *Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings*, volume 9694 of *Lecture Notes in Computer Science*, pages 342–358. Springer, 2016.

[6] J. Bézivin, H. Bruneliere, F. Jouault, and I. Kurtev. Model Engineering Support for Tool Interoperability. In *Workshop in Software Model Engineering*, 2005.

[7] J. Bézivin, G. Hillairet, F. Jouault, W. Piers, and I. Kurtev. Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In *Int. Workshop on Software Factories at OOPSLA*, 2005.

[8] D. Bork. Metamodel-based analysis of domain-specific conceptual modeling methods. In R. A. Buchmann, D. Karagiannis, and M. Kirikova, editors, *The Practice of Enterprise Modeling - 11th IFIP WG 8.1. Working Conference, PoEM 2018, Vienna, Austria, October 31 - November 2, 2018, Proceedings*, volume 335 of *Lecture Notes in Business Information Processing*, pages 172–187. Springer, 2018.

[9] D. Bork, K. Anagnostou, and M. Wimmer. Towards interoperable metamodeling platforms: The case of bridging adoxx and emf. In *Advanced Information Systems Engineering. 34th International Conference, CAiSE 2022*, pages 479–497. Springer, 2022.

[10] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice, Second Edition.* Synthesis Lectures on Software Engineering. Morgan & Claypool, 2017.

[11] G. Braun, P. R. Fillottrani, and C. M. Keet. A framework for interoperability between models with hybrid tools. *Journal of Intelligent Information Systems*, pages 1–26, 2022.

[12] G. A. Braun, G. Marinelli, E. R. Gavagnin, L. A. Cecchi, and P. R. Fillottrani. Web interoperability for ontology development and support with crowd 2.0. In Z. Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 4980–4983. ijcai.org, 2021.

[13] H. Brunelière, J. Cabot, C. Clasen, F. Jouault, and J. Bézivin. Towards model driven tool interoperability: Bridging eclipse and microsoft modeling tools. In *6th European Conference on Modelling Foundations and Applications ECMFA*, pages 32–47. Springer, 2010.

[14] J. Bézivin, C. Brunette, R. Chevrel, F. Jouault, and I. Kurtev. Bridging the generic modeling environment (gme) and the eclipse modeling framework. 01 2005.

[15] F. Cesal and D. Bork. Establishing interoperability between the EMF and the MSDKVS metamodeling platforms. In B. S. Barn and K. Sandkuhl, editors, *The Practice of Enterprise Modeling - 15th IFIP WG 8.1 Working Conference, PoEM 2022, London, UK, November 23-25, 2022, Proceedings*, volume 456 of *Lecture Notes in Business Information Processing*, pages 167–182. Springer, 2022.

[16] S. Cook, C. Bock, P. Rivett, T. Rutt, E. Seidewitz, B. Selic, and D. Tolbert. Unified modeling language (UML) version 2.5.1. Standard, Object Management Group (OMG), Dec. 2017.

[17] S. Cook, G. Jones, S. Kent, and A. Wills. Domain-specific development with visual studio dsl tools. 05 2007.

[18] Y. Crespo, J. Marqués, and J. Rodríguez. On the translation of multiple inheritance hierarchies into single inheritance hierarchies. pages 30–37, 01 2002.

[19] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–646, 2006.

[20] J. Di Rocco, D. Di Ruscio, L. Iovino, and A. Pierantonio. Mining metrics for understanding metamodel characteristics. In *Modeling in Software Engineering*. ACM, 2014.

[21] P. R. Fillottrani and C. M. Keet. Conceptual model interoperability: A metamodel-driven approach. In A. Bikakis, P. Fodor, and D. Roman, editors, *Rules on the Web. From Theory to Applications - 8th International Symposium, RuleML 2014, Co-located with the 21st European Conference on Artificial Intelligence, ECAI 2014, Prague, Czech Republic, August 18-20, 2014. Proceedings*, volume 8620 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2014.

[22] A. Geraci, F. Katki, L. McMonegal, B. Meyer, J. Lane, P. Wilson, J. Radatz, M. Yee, H. Porteous, and F. Springsteel. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, 1991.

[23] R. Hebig, C. Seidl, T. Berger, J. K. Pedersen, and A. Wasowski. Model transformation languages under a magnifying glass: a controlled experiment with xtend, atl, and QVT. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 445–455. ACM, 2018.

[24] F. Jouault and J. Bézivin. KM3: A DSL for metamodel specification. In R. Gorrieri and H. Wehrheim, editors, *8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer, 2006.

[25] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer. Model transformation by-example: A survey of the first wave. In A. Düsterhöft, M. Klettke, and K. Schewe, editors, *Conceptual Modelling and Its Theoretical Foundations - Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday*, volume 7260 of *Lecture Notes in Computer Science*, pages 197–215. Springer, 2012.

[26] H. Kern. The interchange of (meta)models between metaedit+ and eclipse emf using m3-level-based bridges. In *8th Workshop on Domain-Specific Modeling*, pages 14–19, 2008.

[27] H. Kern. *Modellaustausch zwischen ARIS und Eclipse EMF durch Verwendung einer M3-Level-basierten Brücke*, pages 123–137. 09 2008.

[28] H. Kern. Study of interoperability between meta-modeling tools. In M. Ganzha, L. A. Maciaszek, and M. Paprzycki, editors, *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, Poland, September 7-10, 2014*, volume 2 of *Annals of Computer Science and Information Systems*, pages 1629–1637, 2014.

[29] H. Kern. *Model interoperability between meta-modeling environments by using m3-level-based bridges*. PhD thesis, Leipzig University, Germany, 2016.

[30] H. Kern, A. Hummel, and S. Kühne. Towards a comparative analysis of meta-metamodels. In C. V. Lopes, editor, *SPLASH'11 Workshops*, pages 7–12. ACM, 2011.

[31] H. Kern and S. Kühne. Integration of microsoft visio and eclipse modeling framework using m3-level-based bridges. In *Workshop on Model-Driven Tool & Process Integration*, 2009.

[32] M. Koegel and J. Helming. Emfstore: a model repository for EMF models. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 307–308. ACM, 2010.

[33] T. Kühne. Matters of (meta-)modeling. *Softw. Syst. Model.*, 5(4):369–385, 2006.

[34] J. A. H. López and J. S. Cuadrado. MAR: a structure-based search engine for models. In E. Syriani, H. A. Sahraoui, J. de Lara, and S. Abrahão, editors, *MoDELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems, Virtual Event, Canada, 18-23 October, 2020*, pages 57–67. ACM, 2020.

[35] Microsoft. Official online documentation of the modeling sdk for visual studio, 2022. `https://docs.microsoft.com/en-us/visualstudio/modeling/modeling-sdk-for-visual-studio-domain-specific-languages`, last accessed on 23.12.2022.

[36] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, 2003.

[37] V. Viyović, M. Maksimović, and B. Perisić. Sirius: A rapid development of dsm graphical editor. In *Intelligent Engineering Systems INES 2014*, pages 233–238. IEEE, 2014.

[38] V. Vujović, M. Maksimović, and B. Perišić. Comparative analysis of dsm graphical editor frameworks: Graphiti vs. sirius. In *23nd International Electrotechnical and Computer Science Conference ERK*, pages 7–10, 2014.

[39] J. R. Williams, A. Zolotas, N. D. Matragkas, L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. What do metamodels really look like? In M. R. V. Chaudron, M. Genero, S. Abrahão, and L. Pareto, editors, *Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013), Miami, USA, October 1, 2013*, volume 1078 of *CEUR Workshop Proceedings*, pages 55–60. CEUR-WS.org, 2013.

[40] M. Wimmer and G. Kramler. Bridging grammarware and modelware. In J. Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, pages 159–168. Springer, 2005.

[41] V. Zaytsev. Grammar zoo: A corpus of experimental grammarware. *Sci. Comput. Program.*, 98:28–51, 2015.

Appendix A - Code Snippets

```
/// <summary>
/// DSL.R1
/// DomainClass -> EClass
/// DSL.R1/1 - Abstraction (InheritanceModifier = "Abstract" ->
    abstract="true")
/// DSL.R1/2 - Interface (not available in MSDKVS)
/// DSL.R1/3 - Documentation (Notes -> EAnnotation)
/// </summary>
/// <param name="domainClass"></param>
/// <returns></returns>
private static EClassifiers
    TransformDomainClass2EClassifier(DomainClass domainClass)
{
  EClassifiers eClassifier = new EClassifiers()
  {
    Name = domainClass.Name,
    Type = "ecore:EClass",
    Abstract = "false",
    EStructuralFeatures = new List<EStructuralFeatures>()
  };

  if (!string.IsNullOrEmpty(domainClass.InheritanceModifier))
  {
    if (domainClass.InheritanceModifier.Equals("Abstract"))
    {
      eClassifier.Abstract = "true";
    }
  }

  if (!string.IsNullOrEmpty(domainClass.Description))
  {
    var details = new Details()
    {
      Key = "documentation",
      Value = domainClass.Description
    };

    eClassifier.EAnnotations = new EAnnotations()
```

```
        {
          Source = "Tags"
        };
        eClassifier.EAnnotations.Details.Add(details);
      }

      if (domainClass.Properties != null &&
          domainClass.Properties.DomainProperties != null &&
          domainClass.Properties.DomainProperties.Count > 0)
      {
        foreach (DomainProperty domainProperty in
             domainClass.Properties.DomainProperties)
        {
          eClassifier.EStructuralFeatures.
          Add(TransformDomainProperty2EAttribute(domainProperty));
        }
      }

      return eClassifier;
    }
```
Listing 1: M2 Code Snippet, mapping of DomainClass to EClassifier

Appendix B - Model XML Representations

```xml
<?xml version="1.0" encoding="UTF-8"?>
<familytree:FamilyTreeModel xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:familytree="http://www.example.org/familytree">
  <Persons xsi:type="familytree:Man" Name="King George VI"
      BirthYear="1895" DeathYear="1952" Children="//@Persons.2"
      Towns="//@Countries.0/@Towns.1"/>
  <Persons xsi:type="familytree:Woman" Name="Queen Elizabeth"
      BirthYear="1900" DeathYear="2002" Children="//@Persons.2"
      Towns="//@Countries.0/@Towns.0"/>
  <Persons xsi:type="familytree:Woman" Name="Elizabeth II"
      BirthYear="1926" Children="//@Persons.3 //@Persons.5"
      Parents="//@Persons.0 //@Persons.1"
      Towns="//@Countries.0/@Towns.0"/>
  <Persons xsi:type="familytree:Woman" Name="Princess Anne"
      BirthYear="1950" Parents="//@Persons.2 //@Persons.4"
      Towns="//@Countries.0/@Towns.0"/>
  <Persons xsi:type="familytree:Man" Name="Prince Philip"
      BirthYear="1921" DeathYear="2021" Children="//@Persons.3
      //@Persons.5" Towns="//@Countries.3/@Towns.0"/>
  <Persons xsi:type="familytree:Man" Name="Prince Charles"
      BirthYear="1948" Parents="//@Persons.2 //@Persons.4"
      Towns="//@Countries.0/@Towns.0"/>
  <Countries Name="England">
    <Towns Name="London" Persons="//@Persons.1 //@Persons.2
        //@Persons.3 //@Persons.5"/>
    <Towns Name="Sandringham" Persons="//@Persons.0"/>
    <Towns Name="Windsor"/>
  </Countries>
  <Countries Name="Scotland">
    <Towns Name="Edinburgh"/>
  </Countries>
  <Countries Name="Wales">
    <Towns Name="Cardiff"/>
  </Countries>
  <Countries Name="Greece">
    <Towns Name="Corfu" Persons="//@Persons.4"/>
  </Countries>
</familytree:FamilyTreeModel>
```

Listing 2: XMI content of model file based on familytree example in EMF

```
<?xml version="1.0" encoding="utf-8"?>
<familyTreeModel
    xmlns:dm0="http://schemas.microsoft.com/VisualStudio/2008/DslTools/Core"
    dslVersion="1.0.0.0" Id="77a64f61-7e6e-4868-a1a1-e8d1d59389bb"
    xmlns="http://schemas.microsoft.com/dsltools/FamilyTree">
  <persons>
    <man Id="20a1cfbd-24f2-4c71-a9c1-f0be67f8c79f" name="King George
        VI" birthYear="1895" deathYear="1952">
      <children>
        <womanMoniker
            name="/77a64f61-7e6e-4868-a1a1-e8d1d59389bb/Elizabeth II"
            />
      </children>
      <towns>
        <residence Id="dcae3049-d3cd-4760-b3d6-61e87b0f80ef">
          <townMoniker Id="be4de858-b072-4754-a2cd-917fca6af674" />
        </residence>
      </towns>
    </man>
    <woman Id="73f3deb4-c176-49ea-82cd-872dafd7e2b5" name="Queen
        Elizabeth" birthYear="1900" deathYear="2002">
      <children>
        <womanMoniker
            name="/77a64f61-7e6e-4868-a1a1-e8d1d59389bb/Elizabeth II"
            />
      </children>
      <towns>
        <residence Id="153b8adc-da94-459d-a83a-83e67e39556f">
          <townMoniker Id="610d309d-3d8f-4855-9db1-5462481243d0" />
        </residence>
      </towns>
    </woman>
    <woman Id="f87eee5e-437d-4a8f-8758-419f4bbd7de6" name="Elizabeth
        II" birthYear="1926" deathYear="0">
      <children>
        <manMoniker
            name="/77a64f61-7e6e-4868-a1a1-e8d1d59389bb/Prince
            Charles" />
        <womanMoniker
            name="/77a64f61-7e6e-4868-a1a1-e8d1d59389bb/Princess
            Anne" />
      </children>
      <towns>
        <residence Id="a53b61ee-262a-4e28-ad6f-83ce65daf956">
          <townMoniker Id="610d309d-3d8f-4855-9db1-5462481243d0" />
        </residence>
      </towns>
    </woman>
    <man Id="47326798-5d09-4b47-a6b1-3b8d23e6764a" name="Prince
       Philip" birthYear="1921" deathYear="2021">
      <children>
        <manMoniker
            name="/77a64f61-7e6e-4868-a1a1-e8d1d59389bb/Prince
            Charles" />
        <womanMoniker
            name="/77a64f61-7e6e-4868-a1a1-e8d1d59389bb/Princess
            Anne" />
      </children>
      <towns>
        <residence Id="9f968be1-f6d0-4e23-8ff5-e0e4909821d4">
          <townMoniker Id="d05c547e-2231-42d9-8cff-da993964b180" />
        </residence>
      </towns>
    </man>
    <man Id="8ab0fd1e-bee4-4b5e-82d7-f0b800e8fbcb" name="Prince
       Charles" birthYear="1948" deathYear="0">
      <towns>
        <residence Id="62fe407f-6980-46d1-b12d-6e817d5ccb7f">
          <townMoniker Id="610d309d-3d8f-4855-9db1-5462481243d0" />
        </residence>
      </towns>
    </man>
    <woman Id="b913f7bb-5eb6-433e-92e9-bfc93e869152" name="Princess
       Anne" birthYear="1950" deathYear="0">
      <towns>
        <residence Id="87161d7a-9ea6-4239-a1c2-588f5e946526">
          <townMoniker Id="610d309d-3d8f-4855-9db1-5462481243d0" />
        </residence>
      </towns>
    </woman>
  </persons>
  <countries>
    <familyTreeModelHasCountries
       Id="517ec3b1-a2a6-4b6f-8049-62a1fdddfe50">
      <country Id="536e795b-c5e7-42b4-9349-1eaa225f43f3"
         name="England">
        <towns>
          <countryHasTowns Id="8cec4f63-7f61-4d93-abcd-355535f1007f">
            <town Id="fc7b00d6-ea8d-4a65-ad0b-77cbc358bde1"
                name="Windsor" />
        </countryHasTowns>
        <countryHasTowns Id="c70655a5-1b36-4b1b-b6e7-1226d6b7f8b1">
          <town Id="610d309d-3d8f-4855-9db1-5462481243d0"
             name="London" />
        </countryHasTowns>
        <countryHasTowns Id="774e624a-9ee4-4769-a3bd-f57bb43a3bb0">
          <town Id="be4de858-b072-4754-a2cd-917fca6af674"
             name="Sandringham" />
        </countryHasTowns>
      </towns>
    </country>
  </familyTreeModelHasCountries>
  <familyTreeModelHasCountries
      Id="4422c21e-16df-4da8-b2c3-73fdf0605809">
    <country Id="7d454b3c-3376-4061-9e10-1a975746cec7"
       name="Wales">
      <towns>
        <countryHasTowns Id="88b5ad2a-5b71-4599-87f2-b1a7655055dc">
          <town Id="87299cb1-148d-4c6e-aebe-061053a8dba2"
             name="Cardiff" />
        </countryHasTowns>
      </towns>
    </country>
  </familyTreeModelHasCountries>
  <familyTreeModelHasCountries
      Id="78264054-e1dc-45fb-9590-673f6a8dbbfb">
    <country Id="94e1d69a-4204-4231-b22b-b7c98454df6b"
       name="Scotland">
      <towns>
        <countryHasTowns Id="eb14d71c-9435-475a-9b87-87382208f25b">
          <town Id="ddebf0b6-133a-40b2-813d-76d1596b6e1c"
             name="Edinburgh" />
        </countryHasTowns>
      </towns>
    </country>
  </familyTreeModelHasCountries>
  <familyTreeModelHasCountries
      Id="c9d60115-4dee-407d-8c4a-7a959368136e">
    <country Id="5179af5e-de44-460e-9031-f66393383f1a"
       name="Greece">
      <towns>
        <countryHasTowns Id="29031a79-1a24-4a07-8294-b60fcf21f193">
          <town Id="d05c547e-2231-42d9-8cff-da993964b180"
             name="Corfu" />
        </countryHasTowns>
      </towns>
    </country>
  </familyTreeModelHasCountries>
</countries>
```

```
</familyTreeModel>
```

Listing 3: XMI content of model file based on familytree example in MSDKVS, having UseFullForm set to true and OmitElement set to false

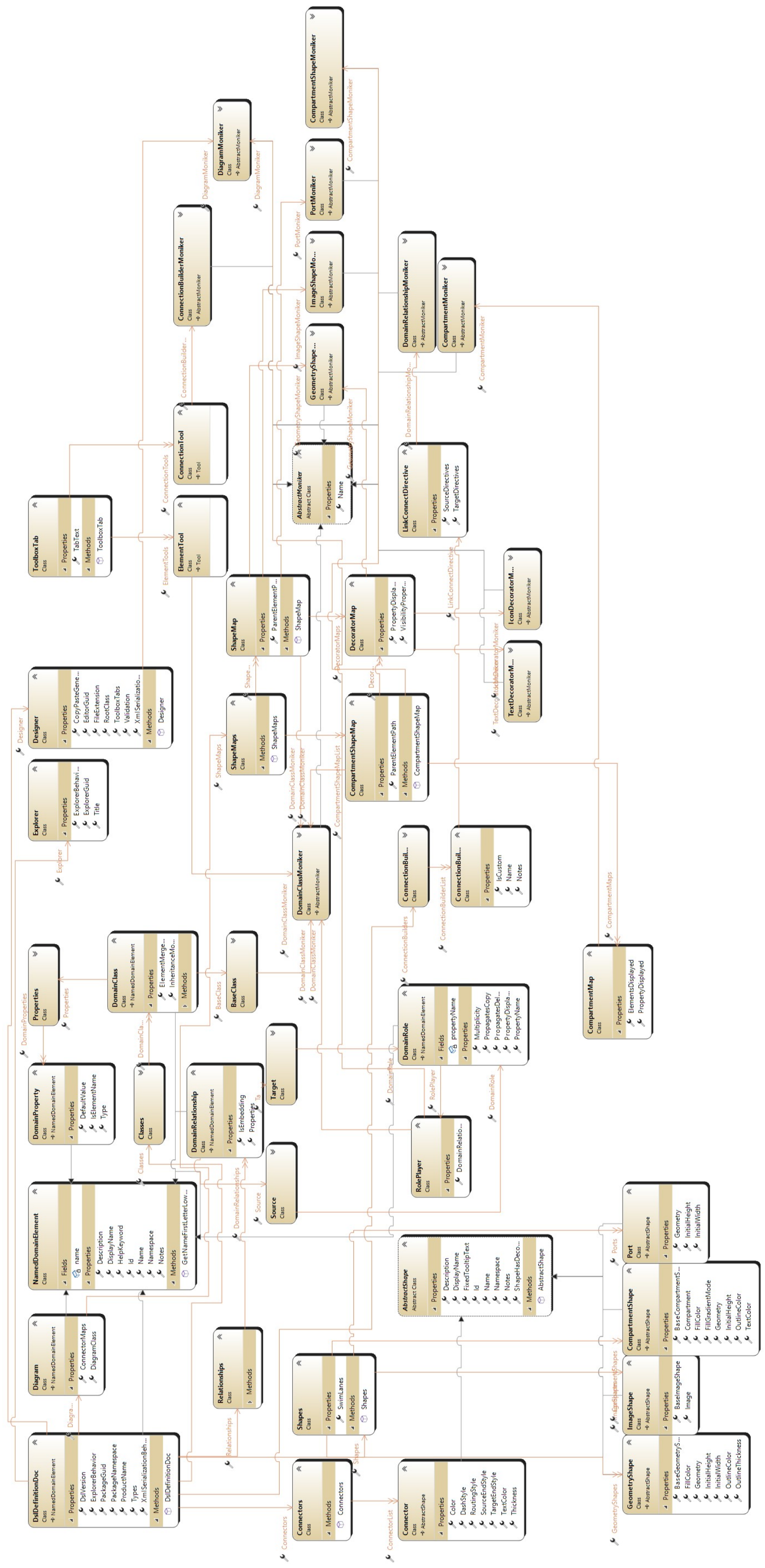
Appendix C - List of Metamodels

EMF	75 Metamodels	22 VSMs	
File	*Name*	*URL*	*Creator*
Agate.ecore	Agate	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
airports.ecore	airports	https://github.com/SPJ1998/Sirius_Aeropuerto/blob/master/com.abernard.airports/model/airports.ecore	Individual Account
Ant.ecore	Ant	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
ATL.ecore	ATL	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
ATOM.ecore	ATOM	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
BDDv2.ecore	BDDv2	https://github.com/atlanmod/coqtl-model-import/blob/main/resources/TT2BDD/BDDv2.ecore	CoqTL
behaviortree.ecore	behaviortree	https://github.com/MiRON-project/Miron-Framework/blob/master/Tools/Eclipse-Modeling-Tools/behaviortree.metamodel/metamodel/behaviortree.ecore	MiRoN
BibTeX.ecore	BibTeX	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
BPMN.ecore	BPMN	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
Bugzilla.ecore	Bugzilla	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
c_sharp.ecore	c_sharp	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
MetaModel.ecore	capstone	https://github.com/ferchouche/Capstone/blob/master/com.capstone.project/model/MetaModel.ecore	Individual Account
class_diagramm.ecore	class_diagramm	https://github.com/DevBoost/EMFText-Zoo/blob/master/BreedingStation/Misc/org.emftext.language.class_diagramm/metamodel/class_diagramm.ecore	EMFText Zoo
COBOL.ecore	COBOL	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
CPP.ecore	CPP	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
DSL.ecore	DSL	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
EclipseDay.ecore	EclipseDay	https://github.com/meeduse/Samples/tree/main/IdmDay/specs	Individual Account
egraphs.ecore	egraphs	https://github.com/eclipse/org.eclipse.emf.diffmerge.core/tree/master/tests/org.eclipse.emf.diffmerge.structures.model/model	Eclipse
esehat.ecore	esehat	https://github.com/AreegT/Research-mdse/blob/master/esehat/model/esehat.ecore	Individual Account
filesystem.ecore	filesystem	-	Online Material
flow.ecore	flow	https://github.com/ObeoNetwork/Flow-Designer/blob/master/plugins/fr.obeo.dsl.designer.sample.flow/model/flow.ecore	Obeo Network
flowChartDebugger.ecore	flowChartDebugger	https://github.com/musahmad/flowChartDebugger/blob/master/flowChartDebugger.model/model/flowChartDebugger.ecore	Individual Account
footballWorldCups.ecore	footballWorldCups	https://github.com/matkve/TDT4250_project/blob/caef547f76a400aef981af96c71d6ed95648985e/tdt4250.football.model/model/footballWorldCups.ecore	Individual Account
FSM.ecore	FSM	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
Gantt.ecore	Gantt	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
GeoTrans.ecore	GeoTrans	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
gore.ecore	gore	https://github.com/gssi/metamodel-dataset-analysis-toolchain/blob/master/metamodels/gore.ecore	Gran Sasso Science Institute
GUI.ecore	GUI	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
HAL.ecore	HAL	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
HTML.ecore	HTML	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
ical.ecore	ical	https://github.com/theoricien/ical-emf/blob/master/iCal/model/iCal.ecore	Individual Account
KDM.ecore	KDM	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
KM3.ecore	KM3	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
lamp.ecore	lamp	https://github.com/occiware/MoDMaCAO/blob/master/plugins/org.modmacao.lamp/model/lamp.ecore	OCCIware
LaTeX.ecore	LaTeX	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
Mantis.ecore	Mantis	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
Matlab.ecore	Matlab	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
mongodb.ecore	mongodb	https://github.com/occiware/MoDMaCAO/blob/master/plugins/org.modmacao.mongodb/model/mongodb.ecore	OCCIware
movies.ecore	movies	https://github.com/gdaniel/imdb-model-migrator/blob/main/imdb/model/movies.ecore	IMDB Model Migrator
MySQL.ecore	MySQL	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
office.ecore	office	https://github.com/tue-mdse/ocl-dataset/blob/master/dataset/repos/damenac/puzzle/plugins/fr.inria.diverse.puzzle.metrics/temp/office.ecore	Data Set of OCL Expressions
OWL.ecore	OWL	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
PetriNet.ecore	PetriNet	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
poosl.ecore	poosl	https://github.com/eclipse/poosl/blob/main/plugins/org.eclipse.poosl.model/model/poosl.ecore	Eclipse

Program.ecore	Program	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
PROVE.ecore	PROVE	https://github.com/TAU-SERI/PROVE/blob/master/dsm.PROVE/model/PROVE.ecore	Individual Account
Publication.ecore	Publication	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
R2ML.ecore	R2ML	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
rails.ecore	rails	https://github.com/gssi/modelAnalyzer/blob/main/src/it/gssi/cs/repository/rails.ecore	Gran Sasso Science Institute
regular_expressions.ecore	regular_expressions	https://github.com/tue-mdse/ocl-dataset/blob/master/dataset/repos/damenac/puzzle/plugins/fr.inria.diverse.puzzle.metrics/testdata/regular_expressions.ecore	Data Set of OCL Expressions
Relational.ecore	Relational	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
RSS-2.0.ecore	RSS-2.0	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
safe2.ecore	safe2	https://github.com/osate/osate2-asap/blob/main/org.osate.asap.model/model/safe2.ecore	OSATE
sandwich.ecore	sandwich	https://github.com/gssi/modelAnalyzer/blob/main/src/it/gssi/cs/repository/sandwich.ecore	Gran Sasso Science Institute
schedularDSL2.ecore	schedularDSL2	-	Online Material
secdfd.ecore	secdfd	https://github.com/SvenPeldszus/GRaViTY-SecDFD-Mapping/blob/master/implementation/org.gravity.mapping.secdfd/model/Secdfd.ecore	Individual Account
sensorProject.ecore	sensorProject	https://github.com/dimitrantz/Thesis/blob/master/corseda-ecore/project/SensorProject/model/sensorProject.ecore	Individual Account
shml.ecore	shml	https://github.com/ECNUCPS/SHML/blob/master/LanguageWorkbench/ecnu.models.shml.model/model/shml.ecore	Individual Account
MiningMart.ecore	MiningMart	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
SimpleAirlineDomain.ecore	SimpleAirlineDomain	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
simpleGraph.ecore	simpleGraph	https://github.com/gssi/metamodel-dataset-analysis-toolchain/blob/master/metamodels/simpleGraph.ecore	Gran Sasso Science Institute
simplegui.ecore	simplegui	https://github.com/jreimone/refactory/blob/master/matching/org.modelrefactoring.guery.test/metamodels/simplegui.ecore	Refactory
simplePDL.ecore	simplePDL	https://github.com/kimokipo/Enseeiht/blob/2A/GLS/IDM/B-01/livrables/Simplepdl.ecore	Individual Account
simTL4J.ecore	simTL4J	https://github.com/gssi/modelAnalyzer/blob/main/src/it/gssi/cs/repository/simTL4J.ecore	Gran Sasso Science Institute
stateMachine.ecore	statemachine	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
SysML.ecore	SysML	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
tinyos_metamodel.ecore	tinyos	https://github.com/husseinmarah/DSML4TinyOS/blob/master/tinyos_metamodel/model/tinyos_metamodel.ecore	Individual Account
Trace.ecore	Trace	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
UML2.ecore	UML2	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
WikiTable.ecore	WikiTable	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
WSDL.ecore	WSDL	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
XHTML.ecore	XHTML	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
xml.ecore	xml	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
XSchema.ecore	XSchema	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
XSLT.ecore	XSLT	https://github.com/atlanmod/atlantic-zoo/tree/main/AtlantEcore	AtlanMod
MSDKVS	**44 DSLs**		
File	*Name*		
DslDefinition (47)	activewriter	https://github.com/castleproject-deprecated/Projects/blob/master/activewriter/trunk/src/Dsl/DslDefinition.dsl	Castle Project
DslDefinition (45)	agilemodeler	https://github.com/jeswin/AgileFx/blob/master/AgileModeler/AgileModeler/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (14)	applicationcooperationviewpoint	https://github.com/gaelgael5/galileo/blob/main/src/ApplicationCooperationViewPoint/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (44)	binbindomainlanguage	https://github.com/lotosbin/binbin-dsl/blob/master/BinbinDomainLanguage/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (27)	candle	https://github.com/malain/candle/blob/master/Package/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (11)	commentreviewrate	https://github.com/PeFerreira98/ISEP.Master.EDOM.Old/blob/main/edom-20-21-team-405/part2/tool2-ms/solution/CommentReviewRate/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (2)	configurationsectiondesigner	https://github.com/hybridview/ConfigurationSectionDesigner/blob/master/src/Dsl.VS2022/DslDefinition.dsl	Individual Account
DslDefinition (22)	cqrsdsl	https://github.com/MerrionComputing/CQRSDesigner/blob/master/src/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (34)	datacontractdsl	https://github.com/chrhodes/Explore/blob/master/WssfSrc/Data%20Contract%20DSL/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (46)	desi	https://github.com/interaktiviti/visual-design-patterns/blob/master/Visual%20Design%20Patterns/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (30)	diagramdsl	-	Online Material
DslDefinition (23)	elevenentitymodel	https://github.com/nekk29/Eleven.VS.Templates.Ado/blob/master/Eleven.VS.Templates.Ado.Dsl/DslDefinition.dsl	Individual Account

DslDefinition (12)	extendedfeaturemodels	https://github.com/TotemApps/ExtendedFeatureModelsDSL/blob/main/Deprecated/ExtendedFeatureModels/Dsl/DslDefinition.dsl	TotemApps
DslDefinition (37)	fwk_dsl	https://github.com/spzenk/sfdocsamples/blob/master/SF/T4/Language2/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (24)	gadgeteerdsl	https://github.com/martinca-msft/Gadgeteer/blob/master/NETMF_44/GadgeteerCore/VisualStudio/Designer/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (20)	generatorlanguage	https://github.com/Seavus/Ultramarine/blob/master/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (15)	gorgeous	https://github.com/HugoVinhal98/Comment-review-and-rate-DSL/blob/master/part2/tool2-ms/gorgeous/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (9)	gvsjgggaslproyectoips	https://github.com/jaimej14/GVSJGGGASLProyectoIPS/blob/master/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (35)	hostdesigner	https://github.com/Phidiax/open-wssf-2015/blob/master/WSSF/Host%20Designer%20DSL/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (25)	iotdsl	https://github.com/JavierMartinDocavo/IoTDsl2/blob/master/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (8)	jdcccajdomdcmproyectoips	https://github.com/DaniilCM/Model_Driven_Development/blob/master/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (10)	jpbsdsjdfproyectoips	https://github.com/Arturox-exe/Arduino_Creator/blob/master/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (1)	lgold	-	Online Material
DslDefinition (42)	library	https://github.com/NuPattern/NuPattern/blob/master/Src/Library/Source/DslDefinition.dsl	NuPattern
DslDefinition (48)	linqtordf	https://github.com/kamelmansouri/linqtordf/blob/master/src/designer/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (18)	mbrdcmdmi	https://github.com/tebiondl/PracticasMDD/blob/main/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (26)	microsoftdataentitydesign	https://github.com/dotnet/ef6tools/blob/main/src/EFTools/EntityDesignEntityDesigner/DslDefinition.dsl	.NET Platform
DslDefinition (21)	mobiledsl	https://github.com/ArturGudiev/MobileDSL/blob/master/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (4)	moneymanagermodel	https://github.com/PeFerreira98/ISEP.Master.EDOM_2021/blob/master/part2/tool2-ms/EDOM/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (7)	mpjaamprototool	https://github.com/javieralvarezlo/MDD/blob/main/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (19)	nhmodelinglanguage	https://github.com/MihailRomanov/TechTalks_DSL_in_Net/blob/master/VsExtensions/NHModelingLanguage/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (17)	nhydrate	https://github.com/nHydrate/nHydrate/blob/master/Source/nHydrate.Dsl/DslDefinition.dsl	nHydrate
DslDefinition (39)	patternmodel	https://github.com/NuPattern/NuPattern/blob/master/Src/Runtime/Source/Runtime.Schema/DslDefinition.dsl	NuPattern
DslDefinition (33)	pdizzle	https://github.com/4deeptech/PDizzle.Dsl/blob/master/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (16)	practicaips	https://github.com/LuxeonVoid/PracticalPS-MDD/blob/main/PracticalPS/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (41)	productstatestore	https://github.com/NuPattern/NuPattern/blob/master/Src/Runtime/Source/Runtime.Store/DslDefinition.dsl	NuPattern
DslDefinition (6)	pugsmbfjmspproyectoips	https://github.com/Chesqoo/PUGSMBFJMSPProyectoIPS/blob/main/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (28)	rapidentity	https://github.com/ahmedsalako/Rapid-Entity-Framework/blob/master/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (5)	rgrllcamsproyectomdd	https://github.com/rubengr16/RGRLLCAMSProyectoMDD/blob/master/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (36)	servicecontractdsl	https://github.com/Phidiax/open-wssf-2015/blob/master/WSSF/Service%20Contract%20DSL/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (43)	smarthome	https://github.com/latacita/tentecsharp/blob/master/Alejandro/Sw/smartHomeCodeGenerator/smartHome/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (13)	spllanguage	https://github.com/TotemApps/ExtendedFeatureModelsDSL/blob/main/SplLanguage/Dsl/DslDefinition.dsl	TotemApps
DslDefinition (38)	sqlmappermapping	https://github.com/npenin/uss/blob/master/Evaluant.Uss.SqlMapper.Designer/Dsl/DslDefinition.dsl	Individual Account
DslDefinition (40)	workflowdesign	https://github.com/NuPattern/NuPattern/blob/master/Src/Authoring/Source/Authoring.WorkflowDesign/DslDefinition.dsl	NuPattern

Appendix D - Class Diagram of MSDKVS


DslDefinitionDoc
Class
NamedDomainElement
Properties
DslVersion
ExplorerBehavior
PackageGuid
PackageNamespace
ProductName
Types
XmlSerializationBeh...
Methods
DslDefinitionDoc
Diagram
Class
NamedDomainElement
Properties
ConnectorMaps
DiagramClass
NamedDomainElement
Class
Fields
name
Properties
Description
DisplayName
HelpKeyword
Id
Name
Namespace
Notes
Methods
GetNameFirstLetterLow...
DomainProperty
Class
NamedDomainElement
Properties
DefaultValue
IsElementName
Type
Properties
Class
Explorer
Class
Properties
ExplorerBehavi...
ExplorerGuid
Title
Designer
Class
Properties
CopyPasteGene...
EditorGuid
FileExtension
RootClass
ToolboxTabs
Validation
XmlSerializatio...
Methods
Designer
ToolboxTab
Class
Properties
TabText
Methods
ToolboxTab
ConnectionTool
Class
Tool
ElementTool
Class
Tool
ConnectionBuilderMoniker
Class
AbstractMoniker
DiagramMoniker
Class
AbstractMoniker
CompartmentShapeMoniker
Class
AbstractMoniker
PortMoniker
Class
AbstractMoniker
ImageShapeMo...
Class
AbstractMoniker
GeometryShape...
Class
AbstractMoniker
AbstractMoniker
Abstract Class
Properties
Name
DomainRelationshipMoniker
Class
AbstractMoniker
CompartmentMoniker
Class
AbstractMoniker
LinkConnectDirective
Class
Properties
SourceDirectives
TargetDirectives
IconDecoratorM...
Class
AbstractMoniker
TextDecoratorM...
Class
AbstractMoniker
ShapeMap
Class
Properties
ParentElementP...
Methods
ShapeMap
DecoratorMap
Class
Properties
PropertyDispla...
VisibilityProper...
ShapeMaps
Class
Methods
ShapeMaps
CompartmentShapeMap
Class
Properties
ParentElementPath
Methods
CompartmentShapeMap
DomainClassMoniker
Class
AbstractMoniker
ConnectionBuil...
Class
ConnectionBuil...
Class
Properties
IsCustom
Name
Notes
CompartmentMap
Class
Properties
ElementsDisplayed
PropertyDisplayed
DomainClass
Class
NamedDomainElement
Properties
ElementMerge...
InheritanceMo...
Methods
BaseClass
Class
Classes
Class
DomainRelationship
Class
NamedDomainElement
Properties
IsEmbedding
Properties
Target
Class
Source
Class
DomainRole
Class
NamedDomainElement
Fields
propertyName
Properties
Multiplicity
PropagatesCopy
PropagatesDel...
PropertyDispla...
PropertyName
RolePlayer
Class
Properties
DomainRelatio...
Relationships
Class
Methods
Shapes
Class
Properties
SwimLanes
Methods
Shapes
Connectors
Class
Methods
Connectors
AbstractShape
Abstract Class
Properties
Description
DisplayName
FixedTooltipText
Id
Name
Namespace
Notes
ShapeHasDeco...
Methods
AbstractShape
Connector
Class
AbstractShape
Properties
Color
DashStyle
RoutingStyle
SourceEndStyle
TargetEndStyle
TextColor
Thickness
Port
Class
AbstractShape
Properties
Geometry
InitialHeight
InitialWidth
CompartmentShape
Class
AbstractShape
Properties
BaseCompartmentS...
Compartment
FillColor
FillGradientMode
Geometry
InitialHeight
OutlineColor
TextColor
ImageShape
Class
AbstractShape
Properties
BaseImageShape
Image
GeometryShape
Class
AbstractShape
Properties
BaseGeometryS...
FillColor
Geometry
InitialHeight
InitialWidth
OutlineColor
OutlineThickness
Diagra...
Relationships
DomainRelationships
Classes
DomainCla...
DomainProperties
Properties
Explorer
Designer
ElementTools
ConnectionTools
ConnectionBuilder...
DiagramMoniker
PortMoniker
ImageShapeMoniker
GeometryShapeMoniker
CompartmentShapeMoniker
DomainRelationshipMo...
CompartmentMoniker
LinkConnectDirective
DecoratorMaps
Decor...
TextDecoratorMoniker
IconDecoratorMoniker
Shape...
ShapeMaps
CompartmentShapeMapList
DomainClassMoniker
BaseClass
ConnectionBuilders
ConnectionBuilderList
CompartmentMaps
DomainRole
RolePlayer
Source
Ta...
Shapes
Connectors
ConnectorList
Ports
GeometryShapes
ImageShapes