Software and Systems Modeling (2024) 23:865–894
https://doi.org/10.1007/s10270-024-01169-x

SPECIAL SECTION PAPER

Establishing interoperability between EMF and MSDKVS: an M3-level-bridge to transform metamodels and models

Florian Cesal[1] · **Dominik Bork**[1]

Received: 15 March 2023 / Revised: 20 February 2024 / Accepted: 18 March 2024 / Published online: 30 April 2024

Abstract
Many powerful metamodeling platforms enabling model-driven software engineering (MDSE) exist, each with its strengths, weaknesses, functionalities, programming language(s), and developer community. Platform interoperability would enable users to exploit their mutual benefits. Such interoperability would allow the transformation of metamodels and models created in one platform into equivalent metamodels and models in other platforms. Language engineers could then freely choose the metamodeling platform without risking a lock-in effect. Two well-documented and publicly available metamodeling platforms are the eclipse modeling framework (EMF) and the modeling SDK for visual studio (MSDKVS). In this paper, we propose an M3-level-bridge (M3B) that establishes interoperability between EMF and MSDKVS on the abstract syntax level and on the graphical concrete syntax level. To establish such interoperability we (i) compare the two platforms, (ii) present a conceptual mapping between them, and (iii) implement a bidirectional transformation bridge including both the metamodel and model layer. We evaluate our approach by transforming a collection of publicly available metamodels and automatically generated or manually created models thereof. The transformation outcomes are then used to quantitatively and qualitatively evaluate the transformation's validity, executability, and expressiveness.

Keywords MSDKVS · EMF · Metamodeling · Model transformation · MDSE · Sirius · Graphical concrete syntax · Abstract syntax · M3B · DSL

1 Introduction

The definition and use of modeling languages offer many benefits in how software teams and language designers can efficiently cooperate on creating a model-based representation of the system under study. Metamodeling platforms offer means to easily define customized languages and many additional functionalities such as code generation, automatic validation, and graphically representing models. These platforms are widely used in enterprise modeling and model-driven software engineering (MDSE). However, once modelers work with one platform, switching to a different one is cumbersome, complex, and costly, especially because automated support for metamodeling platform interoperability is scarce.

This paper looks at the two well-established and actively used metamodeling platforms eclipse modeling framework (EMF) [37] and modeling SDK for visual studio (MSDKVS) [28, 36]. We propose a transformation bridge between EMF and MSDKVS related to bridges reported in [6, 25, 26, 30]. Our bridging enables language designers to switch between platforms by transforming already defined metamodels in one platform into syntactically and semantically equivalent metamodels in the target platform. Syntactic equivalence refers to a mapping of source features to similar features in the target platform, whereas semantic equivalence refers to the equivalence of a feature's meaning within the current metamodeling domain, e.g., the translation of multiple inheritances into semantically equivalent single inheritances [24, 28]. Notably, in an ideal setting, a lossless, fully identical transformation would be aimed for. Given the many differences in the conceptual, technical, and feature levels of existing metamodeling platforms (including those

Communicated by Kurt Sandkuhl, Balbir Barn, Tony Clark, and Souvik Barat.

✉ Dominik Bork
dominik.bork@tuwien.ac.at

Florian Cesal
flo.ces.fc@gmail.com

[1] Business Informatics Group, TU Wien, Favoritenstrasse 9-11, 1040 Vienna, Austria

between EMF and MSDKVS), such identical mappings are not feasible. Instead, with this paper, we aim for an equivalence relationship between concepts and features of the two metamodeling platforms in question and discuss, where the limitations of such an approach are. Aside from these limitations, we show that syntactically and semantically equivalent transformations between metamodels and models of EMF and MSDKVS are possible and, therefore, enable a bridge for users who aim to switch from one of the two platforms to the other.

The EMF-MSDKVS bridge enables (i) migration and reusability of existing metamodels across platforms, (ii) decoupling the developers of the underlying programming languages[1] these platforms are built upon, and (iii) making use of specific platform capabilities employed elsewhere [27], e.g., plug-ins only available for EMF; thereby ultimately enabling (iv) metamodeling platform-spanning toolchains, i.e., a chain of tools developing with different metamodeling platforms, conjointly realizing a complex model-driven software engineering pipeline. Capabilities, such as code generation, which are more sophisticated in, e.g., EMF, can also be a motivational factor for transforming metamodels from MSDKVS to their EMF equivalent. In the ecosystem of metamodel repositories, many projects exist that have been created with one metamodeling platform. Establishing a toolchain that allows the transformed metamodel to be interpreted by other platforms thus mitigates a platform lock-in.

Generally, transformation bridges are based on mappings between the meta-metamodels of both platforms. These mappings are created by analyzing the similarities and identifying the differences between these platforms located at the M3 layer of the standardized metamodeling stack [10]. Previous approaches implement transformation bridges targeting the platforms' abstract syntax elements (e.g., classes and relationships), mostly ignoring the platform's functionalities to graphically represent and manipulate the created models as this is where the heterogeneity between different metamodeling platforms is very rich and custom solutions, often even technologies, for each platform are in place.

This paper first analyses the EMF and MSDKVS platforms and then proposes, implements, and evaluates a transformation bridge. We build upon and improve our previous work reported in [15] primarily by (i) a model layer transformation, i.e., our bridge now enables also the exchange of models between EMF and MSDKVS; and (ii) a comprehensive evaluation of both the metamodel and the model transformation on a syntactic and semantic level. The evaluation is concerned with testing whether the transformators produce valid outcomes, i.e., that the produced models and metamodels can be imported, and that the produced metamodels can be used to start runtime instances in the target metamodeling platform.

In the remainder of this paper, Sect. 2 explains the area in which the implemented approach is situated and establishes the necessary foundations. Section 3 then discusses related works. A comprehensive analysis of the EMF and MSDKVS platforms' concepts is presented in Sect. 4, resulting in mapping rulesets the transformator has to implement, which are listed in Sect. 5. Section 6 explains both parts of the transformation bridge, namely the M2 transformator located on the metamodel layer, and the M1 transformator located on the underlying model layer. Sections 7 and 8 give insight into the evaluation process and its results. Section 9 concludes this paper with some closing remarks and an outlook on future work.

2 Metamodeling foundations

Complete or partial representations of real-world objects, architectures, or software systems can be realized through the use of models. These models can then be shared and enable communication among stakeholders [10]. Concerning the validation and guidelines for defining models, an abstraction hierarchy exists, divided into a stack of layers. An example of such a hierarchical stack, consisting of four layers, has been standardized by the object management group (OMG) [10, 31]:

M0 Layer (runtime instances) containing the application data or runtime instances; M1 Layer (model layer) describing the concrete model created by a user that is conforming to a given metamodel (e.g. a UML model); M2 Layer (metamodel layer) defining the metamodel (e.g. a UML metamodel); M3 Layer (meta-metamodel layer) abstracting the definition for possible metamodels. In the OMG metamodeling hierarchy, the M3 layer is defined by the MetaObject Facility (MOF [21]) standard.

The M3 level also establishes the foundation for realizing interoperability between metamodeling platforms based on a common abstraction of their metamodels. Modeling languages consist of the following elements, which should be taken into consideration when implementing a transformation bridge: Abstract Syntax defines classes, their attributes, and associations required to represent the relevant parts of the modeled system and constraints for restricting the set of valid models. Abstract syntaxes are most often specified via metamodels [8]. Concrete Syntax defines the visual representations for the abstract syntax elements (e.g., graphical and/or textual) [7]. An introduction to the two types of concrete syntaxes is given in [10].

[1] Note that throughout this paper we will use the term platform as a representative of all tools, frameworks, and platforms that allow metamodeling.

Metamodeling platforms offer the ability to define metamodels on the M2 layer. Some offer only the ability to define the abstract syntax of a metamodel, others also allow the definition of concrete syntaxes. As they all reference a meta-metamodel to cover all the possibilities of defining a metamodel, one can argue that the meta-metamodel can define itself on an even more abstract level, making every metamodeling platform originate from a most basic type of "meta-metametamodel", consisting only of elements and links between those elements. Attributes, as they exist in EMF and MSDKVS metamodels, can then be declared as elements as well, and a link between a class element and an attribute element can be regarded as a member variable of a class on the layer underneath.

As elements and links form the basis of our abstract notation, the same can be done for the graphical syntax. Elements can be displayed as four-sided shapes, links can be displayed as lines. If two elements are in relation to each other, they are connected by these lines. As mentioned in [10], a graphical concrete syntax (GCS) has to combine different kinds of elements, e.g., graphical symbols (meaning figures, lines, but also labels for displaying an element's information), compositional rules defining nesting and combination of graphical symbols, and a mapping between these symbols to the abstract syntax of a metamodel.

Metamodeling platforms with integrated graphical user interfaces display a modeling canvas for the positioning of elements in a two-dimensional space by assigning them x and y coordinates. The group of placed model elements are arranged as a graph and referred to as a diagram. Additional properties, such as coloring and font features, to name a few, can be edited in an additional property window. [38]

Once the language engineering part introduced previously is finished, tool developers can shift their focus on realizing features to process the models that modelers can create when using the engineered language. As stressed in [10], the two core concepts of model-based software engineering are *models* and *transformations*. Model transformation is an established research field in the modeling community [5, 33]. Basically, different categories of model transformation approaches can be differentiated based on the format of the transformation's source and target: *model-to-model transformations*, i.e., transformations where source and target are models and *model-to-text* where the source is a model and the target format is text, e.g., scenarios where source code is generated from models. When considering the model-to-model transformations, we can further differentiate transformations where the source and the target model conform to the same metamodel (*endogenous transformations*) from *exogenous transformations* where source and target models conform to different metamodels (i.e., modeling languages). In the remainder of this paper, we will focus on exogenous model-to-model transformations between metamodels and models created with EMF and MSDKVS.

Two prominent exemplars of metamodeling platforms with which the authors of this paper have worked with are introduced in the following and investigated regarding their implementation of the M3 layer and their concrete syntax.

2.1 EMF

The eclipse modeling framework (EMF) is an open source metamodeling platform that provides a rich set of features for, e.g., defining metamodels, creating and validating models, transforming models, and serializing models into XMI format. EMF allows runtime support to generate Java classes and programmatically manipulate the models through reflection. This section describes the core features of EMF and how EMF supports metamodeling [37].

Abstract Syntax in EMF. To realize metamodel support in EMF one needs to specify the metamodel by instantiating concepts from the EMF meta-metamodel, called *Ecore*, i.e., an implementation in Java of a simplified version of OMG's standardized MOF meta-metamodel, called *eMOF* (i.e., the essential MOF [21]) model. This meta-metamodel (see Fig. 1 for an excerpt) thus plays an essential role as it determines the expressiveness of all possible metamodels. An explicit definition of the Ecore meta-metamodel is given in various sources, e.g., in [6, 10, 30].

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. A composition relationship between source and target *EClasses* can be defined by setting the containment flag of an *EReference* to "true."

Concrete Syntax in EMF. The Eclipse website lists three frameworks that can be used for visualizing Ecore metamodels and models: Graphical Language Server Platform [9, 34],[2] Sirius,[3] and Graphiti.[4] For the matter of this paper, we only consider Sirius as it is the most commonly used framework and best resembles the possibilities of graphical viewpoint representations compared to MSDKVS.

[2] https://eclipse.dev/glsp/.

[3] https://eclipse.dev/sirius/.

[4] https://eclipse.dev/graphiti/.

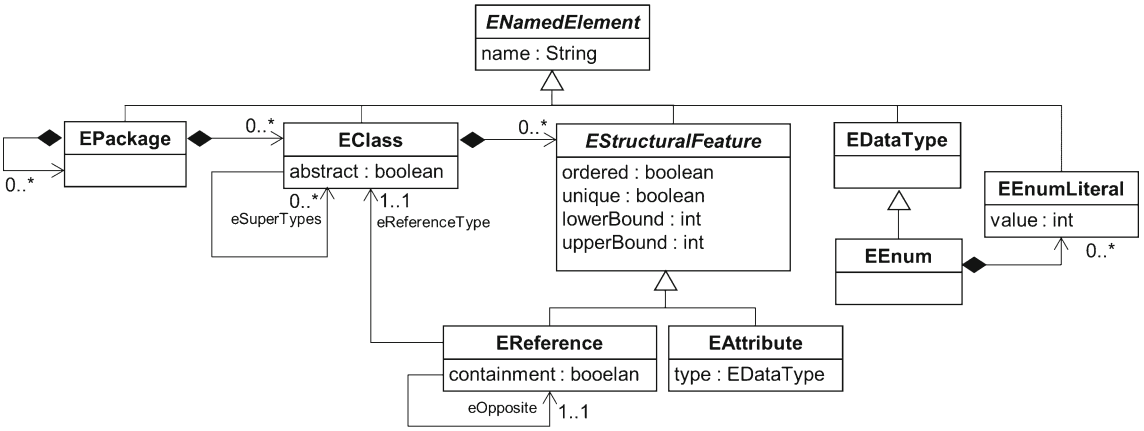


Fig. 1 Excerpt of the Ecore meta-metamodel [6]

Sirius uses *Viewpoint Specification Projects (VSP)* containing descriptive model files ending with `.odesign` [39]. 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.

Regarding the structure of a *Viewpoint Specification Model (VSM)* in Sirius, the following elements are worth mentioning: Beginning with the root element of the specification file, *Group*, which can be compared to Ecore's *EPackage* element. A group can contain one or more *OwnedViewpoints*, which in turn contain one or more *OwnedRepresentations*, describing representations for the metamodel's abstract syntax elements. Different types of representations are available, with the most suitable and relevant for the upcoming transformation bridge being the *diagram representation*. The other types, i.e., table, tree, and matrix representations, are not targeted by the M3B discussed in this paper.

Every representation has a *DefaultLayer*, with optional additional layers used for grouping and also for hiding specific elements contained within these layers. Each layer contains graphical mappings to abstract syntax elements, most notably the *NodeMappings*, used for styling standalone *EClasses*. Each mapping can define different styles for the target element (e.g., size, shape, color). Classes, that are used for containment references can be graphically attributed via *ContainerMappings*. Inner or attached classes can be styled by *SubNodeMappings*. *EdgeMappings* are used for visualizing *EReferences*, e.g., the style of arrows (start, end, and the line routing style).

Zero or multiple *ToolSections* can be defined, grouping tools for usage on the modeling canvas. These sections contain different types of *OwnedTools* that can be used for, e.g., creating and deleting elements, connecting two elements, or copying a group of elements. A tool can have one or more sequentially executed operations to manipulate the modeling canvas. Finally, the *ColorPalette* can be attributed with explicitly defined colors, usable from within the aforementioned mappings.

2.2 MSDKVS

MSDKVS supports the development of domain-specific languages by weaving abstract syntax and graphical concrete syntax (see [16, 36] for a detailed introduction). MSDKVS offers a graphical user interface with an integrated editor to define metamodels (i.e., classes, relationships, and their properties), a tree explorer, a property editor window, and several additional features such as XML serialization of metamodels and models, code generators using a templating engine, and the possibility to build extensions to these features. The currently available MSDKVS NuGet Package[5] was released in 2023 and still has an active community of users with around daily 30 downloads.

Abstract Syntax in MSDKVS. As MSDKVS does not publicly offer a representation of its meta-metamodel, the transformation approach explained in Sect. 6 implicitly offers the ability to reconstruct a MSDKVS meta-metamodel corresponding to the data of the serialized metamodel files. Figure 2 shows the core excerpt of the reconstructed MSDKVS meta-metamodel represented as a UML class diagram, containing the most apparent and most used elements and their abstract super classes, that have been identified while working on the transformation and defining its mapping rules. The full representation is provided online.[6]

When creating a DSL in MSDKVS, one element always has to act as the root element of the metamodel and every sub-

[5] https://www.nuget.org/packages/Microsoft.VisualStudio.Modeling.Sdk.

[6] Online supplementary material: https://drive.google.com/file/d/1-Uzz61MJAW5NPetEuPH-LcUtFgaG_iyy/view?usp=sharing.

sequently created *DomainClass*, if not targeted by another embedded relationship, is referenced by this root class. The root class, by default, initially has the same name as the DSL itself. *Doc* is the diagram document representing the DSL, containing shapes, relationships, classes, serialization behavior, and the mapping of shapes to abstract elements. The possible entities that can be created on the metamodeling canvas are available in the DSL Designer Toolbox. These elements include *DomainClasses* and different types of *DomainRelationships*, like embedding relationships (i.e., containers) and reference relationships. Every relationship is binary and directed and links members of a source class (i.e., *Source*) to members of a target class (i.e., *Target*). Such members include the classes themselves as well as the classes inheriting from them. Both Source and Target reference their abstract elements through *DomainRoles*, specified by *RolePlayers* that finally contain the *Moniker* type of the referenced element. Each element can further be attributed with various *DomainProperties*. External types, e.g., system types such as *String*, *Boolean*, or *DateTime* can also be referenced by a *DomainProperty*. These elements compose MSDKVS's abstract syntax. Each element inherits from the base class *NamedDomainElement*, containing a unique *Name* and an *Id*.

Regarding their XML serialization, the `DslDefinition.dsl` file, when opened in a text editor, contains all objects added on the DSL canvas and their mapping references to tool palettes, shapes (i.e., graphical concrete syntax), and other serialization properties needed for code generation. Every added abstract or concrete element is given a *Moniker* description type to be able to be referenced in different parts of the DSL. Monikers are uniquely identifying names for elements.

Concrete Syntax in MSDKVS. Every class, relationship, and attribute can be visually enhanced with different shapes and decorators that are maintained within the editor's graphical interface adjoined to the abstract syntax definitions. Through mappings between the concrete and abstract syntax definitions, the language designer can customize the appearances and interaction possibilities like toolbox entries or graphically editing attributes in the Visual Studio runtime instances.

The possible shape elements are also listed inside the DSL Designer Toolbox. As these elements are tightly included as core features of the platform, as well as serialized in the same file as the abstract syntax, some of the graphical syntax elements are included in Fig. 2. To summarize all available shapes and their common attributes, an *AbstractShape* has been introduced. Similarly, an *AbstractMoniker* represents all available moniker types.

Graphical concrete syntax elements include *GeometryShapes*, defining the visual notation of the mapped *DomainClasses*. Each *DomainProperty* of a class or a relationship, e.g., font attributes and line types, can be visually defined through *DecoratorMaps*. *CompartmentShapes* target classes that can contain other classes, either as lists, ports (i.e., attached shapes depicting either an ingoing or outgoing interface), or image shapes. *ImageShapes* are freely configurable shapes that reference an image file contained within the project's resources folder. The appearance of relationships, e.g., their line thickness and style can be defined by *Connectors*. A separate *PortShape* object is available for graphically defining contained classes within *DomainClasses*. Unique graphical syntax elements in MSDKVS include the *Swimlane* element, used for dividing an existing diagram to create visually sophisticated DSLs.

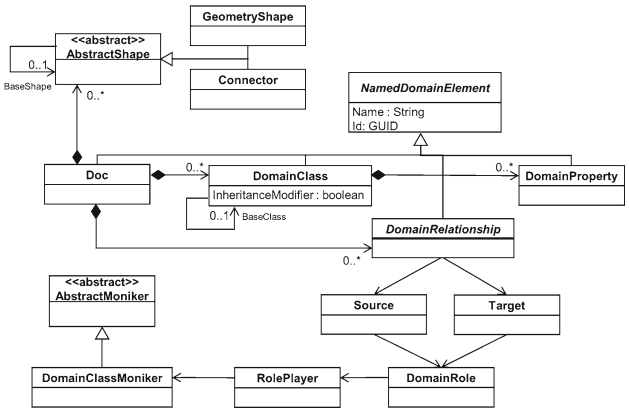


Fig. 2 Excerpt of the reconstructed MSDKVS meta-metamodel [15]

3 Related work

This section first offers an overview of existing works on metamodeling platform interoperability. It then takes a detailed look at related ambitions toward bridging EMF and MSDKVS and compares these works to this paper's approach.

3.1 Transformation bridges

Interoperability deals with the exchange of information between two or more systems, and the ability to use that information in each system respectively [20, 27]. As modeling languages for software development gained popularity in the early 2000s, a need for establishing interoperability by transforming the *grammarware* technical space (i.e., EBNF-based grammar tools) into the *modelware* technical space existed [40]. Once this interoperability was established, many metamodeling platforms followed, which in turn also raised the need for their interoperability.

Several bridges between different metamodeling platforms and modeling tools have been proposed, including

EMF and ARIS [26], EMF and MetaEdit+ [25], EMF and Visio [30], and EMF and Generic Modeling Environment [14]. Recently, a transformation bridge between ADOxx and EMF has been proposed in [6]. These transformation bridges typically consist of one or several model transformations that are used for exchanging metamodels and models between the two platforms. These are so-called *horizontal exogenous transformations* [10, 22, 28], as the source and target of the transformation are situated on the same abstraction level but adhere to different meta-metamodels. Most of these works transform metamodels, i.e., do not consider interoperability at the model level which further requires a transformation between concrete syntaxes. Table 14 lists these existing transformation bridges (i.e., M3-level-based bridges or M3Bs) between different metamodeling platforms. Consistently to these related works, we also use bidirectional (or multiple unidirectional) transformation bridges.

Many of the investigated transformation bridges are available online, either within public code repositories for downloading the libraries themselves or as widgets on a web page. Nearly all of them, except for, e.g., the Aris2EMF Bridge, have been written in Java to directly integrate EMF functionalities and transformation capabilities, like ATL, into the bridge. Links to the implementations, if available, are provided and an indicator of whether or not the available transformator is still executable in today's environments (see Table 14).

As a matter of fact, each system that can be abstracted to a 3-level architecture can be used in a transformation bridge, thus also enabling the transformation of non-metamodeling languages to achieve tool interoperability [3, 28]. Consequently, Table 14 also lists transformation approaches of tools that have an underlying structure that can be abstracted to a metamodeling architecture, e.g. Excel. Bézivin et al. [3] define a pivot metamodel to combine the abstraction of features that are common among the selected systems. The bridges in question are listed in Table 14 as Excel2SoftwareQualityControl, SoftwareQualityControl2Mantis, and SoftwareQualityControl2Bugzilla respectively, where the *SoftwareQualityControl* metamodel acts as the pivot metamodel. The specific models are, therefore, not directly transformed to the target tool environment. Instead, a pivot metamodel serves as an intermediary entity. Another interoperability approach trying to bridge different conceptual data modeling tools like ER and ORM2 is given in [11], where a common metamodel, namely the *KF Metamodel* [19], with rules for transforming from and to these conceptual modeling platforms is used and implemented in a web-based tool called *crowd 2.0* [12].

3.2 EMF and microsoft DSL tools

Research on bridging EMF and Microsoft DSL Tools has been proposed in the past [4, 13]. 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 [4]), the visualization of a meta-metamodel containing the *ValueProperty* entity compared to today's *DomainProperty*, and the representation of attributes for classes and relationships.

The previous approaches execute a chain of ATLAS transformation language transformations to generate the transformed metamodel using the *KM3 (Kernel MetaMetaModel)*, a DSL for describing metamodels [23] as an intermediate representation of arbitrary metamodels. As a transformation already existed between KM3 and Ecore, the MSDKVS metamodels needed to be only transformed to this pivot KM3 metamodel. Thus, no direct transformation between EMF and MSDKVS 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. We examined the previous approach with preserved `.dsldm` files of the Atlantic-Zoo Github[7] and learned 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.

This paper gives a detailed comparative analysis (see Sect. 4) of the abstract and concrete syntax elements available in the latest versions of the EMF and MSDKVS platforms that far exceed previous works. Moreover, we present the first direct transformation bridge that also transforms the graphical concrete syntax. One example transformation was explained in [13], the PetriNet metamodel, where the question remains if the validation of the transformed metamodel and models was successful in the target platform. The sources of this approach are still available in a repository, but as depicted in Table 14 and explained here, the transformation code is not working with the current versions of the platforms in question. In the paper at hand, we address these gaps by providing an exhaustive quantitative and qualitative evaluation of the transformation bridge (see Sect. 7).

4 Comparative analysis of EMF and MSDKVS

This section analyzes EMF and MSDKVS regarding their abstract and concrete syntax. The relevant elements were adapted and extended from [6, 29] in terms of concrete syntax

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

concepts extracted through a detailed investigation of both platforms. A full list of the identified, analyzed, and mapped abstract and concrete syntax elements is provided in Appendices A and B and also available together with additional material in the online supplementary material[6]. In the following, we concentrate the analysis on the core differences between EMF and MSDKVS as these differences establish the challenges of designing a direct transformation bridge (see Sect. 6).

4.1 Abstract syntax features

EMF allows the definition of classes that inherit properties and possible relationship structures from multiple classes (i.e., *multiple inheritance*) whereas MSDKVS only allows entities to inherit from one referenced base object (i.e., *single inheritance*). MSDKVS, in contrast to EMF, allows *inheritance between relationships*, meaning source and target roles of the super relationship are also available in the sub relationship. Furthermore, domain relationships in MSDKVS can also act as domain classes that can then be connected to different domain relationships as a source or a target role. One minor but challenging difference relates to the possibility of relationships between elements to have the *same name* in EMF, which leads to name clashes on the MSDKVS side where relationship names are required to be *unique*. On MSDKVS, domain classes are not directly referenced when creating a relationship. Instead, they are indirectly referenced through monikers, and a domain relationship is comprised of source and target *domain roles* also referencing these monikers. In [28], relationships of EMF are defined as *reference-relations* and of MSDKVS as *binary object-relations*. We agree on the EMF part, although we find the definition of *role-relations* more suitable for the binary relationships in MSDKVS, as they indeed explicitly create additional model elements, namely *DomainRoles*, for each source and target role of a domain relationship, which then reference the "real" metamodel element via their moniker types. This is the reason why we list the concept of a *Role* in Appendix A and attribute them to MSDKVS and not to EMF. When implementing a transformation between MSDKVS and EMF, correctly resolving these indirect dependencies to achieve syntactical and semantical equivalence, i.e. translating multi-inheritance and the different ways of treating relationships into a semantically equivalent single inheritance and relationship treatments with the same experienced behavior by modelers, are some of the challenges addressed in this paper.

4.2 Graphical concrete syntax features

MSDKVS offers the possibility to inherit properties among shapes (i.e., *shape inheritance*), while such an inheritance is not supported in EMF. Besides the support for widely used basic shapes like rectangles, circles, and icons, each platform offers *special shape types* that cannot be directly mapped to an equivalent one in the opposite platform. As metamodeling platforms often depend on an underlying programming language (e.g. EMF on Java, MSDKVS on C#), the available *coloring options*, *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. Metamodeling platforms also allow the use of custom *image files* to adapt the appearance of model elements. EMF and MSDKVS differ in their support of various file formats. Icons can be used to, e.g., add custom appearances to tool palette items or composition shapes. Different types of *tools* have to be defined to create models in a runtime environment. However, the granularity of what types of tools can be created and customized varies greatly. MSDKVS only allows the definition of essential *element creation tools* for domain classes and domain relationships. In contrast, 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 available when a creation tool is defined. Thus, copying, pasting, or deleting modeling canvas elements works out of the box on MSDKVS whereas tool developers using EMF need to implement such functionality.

5 Transformation rulesets

Based on the comparative analysis in the previous section and the detailed assessment documented in the online supplementary material[6], different rulesets composing the intended transformation bridge have to be defined. In the following, lists of mapping rules for the abstract syntax (Table 1 and the concrete syntax Table 2) are given for each transformation direction. Two selected specific rules for each, abstract and concrete syntax, shall explain in detail how the rulesets generally have been defined in terms of supporting the implementation afterward and highlight the platform-specific features on a code level. A list with explanations for every rule is available in the online supplementary material.[6]

5.1 Abstract syntax

AS.R1: ClassMapping When transforming from EMF to MSDKVS, three types of *EClassifiers* can be distinguished regarding their `type` attributes, namely *EClasses*, *EEnums*, and *EDataTypes*. When transforming an *EClass*, a target *DomainClass* is created using the same `Name`. As it is necessary for each element in MSDKVS to have

Table 1 List of mapping rules for abstract syntax concepts

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 2 List of mapping rules for graphical concrete 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

a unique `Id` as an identifier, a random GUID is generated upon creation. `Abstract` attributes are transformed to `InheritanceModifier` values. If the EMF class is abstract, the modifier receives the value "1," as it marks the resulting *DomainClass* as abstract [36]. If the current *EClass* is the identified root class of the metamodel, it must not be abstract in MSDKVS. `XmlClassData` is added to the serialized XML data as well to use MSDKVS' code generation and model editor capabilities, containing the resulting generated moniker types of the added classes to conform to the required structure on the target side.

In the direction of MSDKVS to EMF, the class concept is de facto identical, except for the specifications like inheritance and *DomainRoles* used for targeting in *DomainRelationships*, as they have to be included in an *EClass* as *EStructuralFeatures*, more concretely, *EReferences*. These transformations are executed in Rule AS.R2. `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: RelationshipMapping Each *EClass* can contain two types of *EStructuralFeatures*, either *EReferences* or *EAttributes*. *EReferences* are transformed to *DomainRelationships*, generating a unique identifying GUID for the required `Id` attribute. *EReferences* can either be flagged as containment references, simple, or bidirectional references between two *EClasses*.

Figure 3 lists these reference types, their availability in both platforms, and their mapping to the transformed platform types. EMF offers uni-, bi-directional, and composition relationships, whereas MSDKVS only offers bi-directional and composition relationships. This is due to the fact that, in MSDKVS, a *DomainRelationship* always consists of source and target roles, represented by *DomainClasses* and a specific domain role they are given regarding the relationship definition, shown as separate entities derived from the *DomainClasses*, referenced via monikers, each starting with `DR_` in the given figure.

Beginning from EMF, source and target entities of *EReferences* are transformed to *DomainRoles*, linking each previously transformed *DomainClass* via `Monikers` using the unique names of the classes. Multiplicities of a *EReference* are transformed accordingly.

Regarding the other direction, in MSDKVS, a *DomainRelationship* always consists of source and target roles, representing *DomainClasses*. A *DomainRelationship* in MSDKVS can be mapped to an *EReference* in EMF. The name for

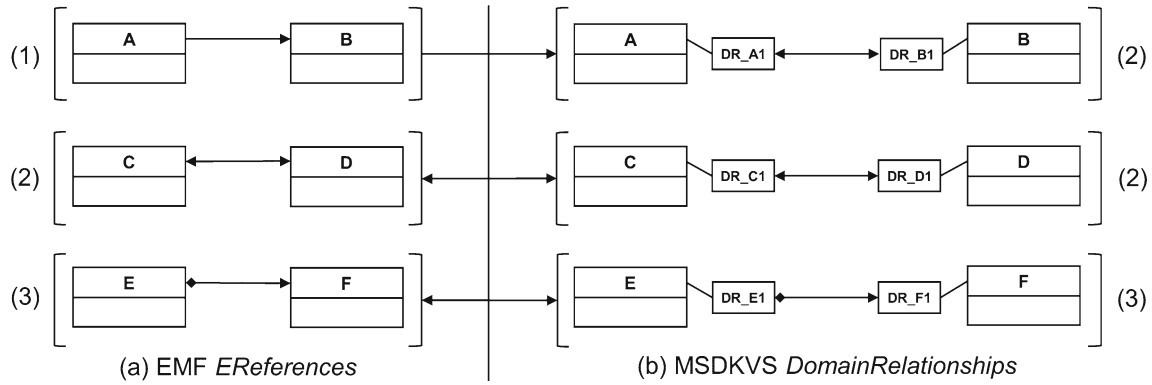


Fig. 3 Mapping of relationship types between EMF and MSDKVS. Unidirectional relationships (1), bi-directional relationships (2), and composition relationships (3)

the *EReference* is mapped from the source domain role from the relationship in MSDKVS, and the type from the reference is retrieved from the target domain role, relating to the target domain class. As a consequence, only bi-directional references are retrieved when transforming from MSDKVS to EMF. As stated in the documentation of EMF, a single, one-way reference can always be defined as a bidirectional, two-way reference, thus making it possible to directly transform each reference type between these two platforms.

AS.R7: Inheritance As EMF supports multiple inheritance structures, concrete patterns have to be applied to break down these structures into multiple, single inheritance structures on the target side. The concept of Inheritance between class structures is trivial when transforming from MSDKVS to EMF, as both platforms support the notion of having their classes inherit references and attributes from other classes. These inheritance references are signaled by the classes' properties *BaseClass* in MSDKVS and *ESuperTypes* in EMF. In MSDKVS, these base class references are done via the *DomainClassMoniker* types. MSDKVS additionally allows Inheritance between *DomainRelationships*, making the transformation of inheritance structures based on relationships challenging to maintain structural integrity, thus resulting in additional elements as a *DomainRelationship* first has to be transformed to a class and then two additional references have to be created to keep the structure of the metamodel alike.

5.2 Graphical concrete syntax

Notably, also the default values for each attribute in MSDKVS have to be considered in the transformation 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 of the available API documentation. An example of 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 one, 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: ClassShapeMapping NodeMappings in EMF are transformed into *GeometryShapes*, visualizing *DomainClasses* through different geometries, like `Rectangles` or `Circles`. The comparison tables in Appendix A are grouped into the various identified shape types after having investigated both metamodeling platforms. A *DomainClass* is then referenced accordingly by mapping the `ShapeMoniker` inside the *Diagram*.

Geometry shapes in MSDKVS reference *DomainClasses* only. The corresponding transformed graphical entity in Sirius is called a *NodeMapping*. The customization of the appearances (e.g., geometries), styles (e.g., border style, font style), and layout (e.g., positioning) of the entity itself and its domain properties are considered in rule GCS.R5. An example of a *NodeMapping* resulting from a four-sided *GeometryShape* has the `style:SquareDescription` attribute.

GCS.R7: ColorMapping Sirius in EMF lets users define colors through drop-down tables 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` [2].

Through these steps, a color can be changed by associating values via computation expressions. For our 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.

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 the 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, calculated 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.

6 Transformation bridge

Figure 4 sketches all three layers involved in realizing interoperability between EMF and MSDKVS. On the left, the MSDKVS column consists of the implicitly defined metametamodel on the M3 layer (see Fig. 2), with its user-defined metamodel on the M2 layer. The metamodels are serialized in XML format as `.dsl` files. These files are used as input and output of the transformation, depending on which platform is the source and the target of the transformation. The transformator itself is divided into transforming metamodels (M2 transformator) and models (M1 transformator). The M2 transformator is written in C# and de-serializes the incoming files into data structures that can be manipulated and worked with on the code level. 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 both platforms, are applied sequentially to transform the source metamodel into an equivalent metamodel of the target platform. Section 6.1 discusses some of the special cases for each direction that have to be considered during this step.

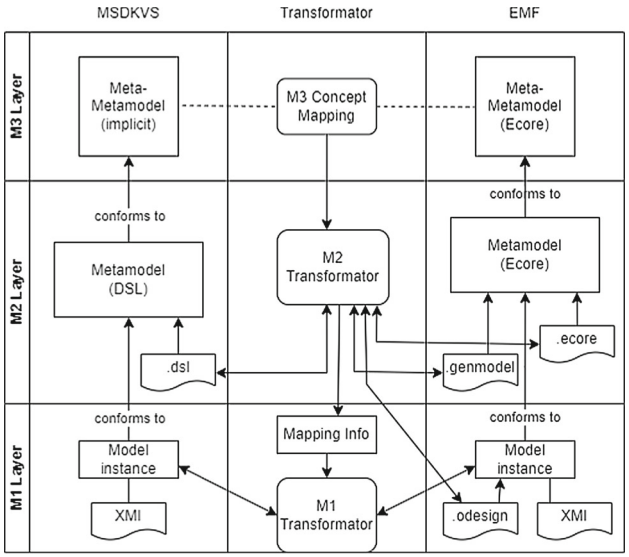


Fig. 4 Transformation bridge between EMF and MSDKVS

An additional outcome of the M2 transformation is a mapping information file, containing a serialized JSON object of all the applied strategies (e.g., renaming strategies based on duplicate relationship names or resulting from solving multiple to single inheritance structures). This file is then used as input in the M1 transformator to map the model elements accordingly. The M1 transformator is written in Java and uses the ecore reflection API to serialize and de-serialize source and target (meta)models more thoroughly (see Sect. 6.2 for details). It requires the Ecore metamodel file as input in both directions, the model file created in the source platform conforming to the previously transformed metamodel, and the aforementioned mapping file.

6.1 M2 transformation

The identified elements in each platform and their counterparts in the other platform resulted in comprehensive rulesets (cf. Sect. 5). The greatest challenges faced and detailed steps on how these were solved during the traversal and transformation of elements in the M2 Transformator are discussed in the following.

6.1.1 EMF2MSDKVS

Nested EPackage Flattening. We recognized different styles of EPackage definitions in publicly available EMF metamodels (see Table 3 in the row entitled "Grouping"). 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 trans-

Table 3 Metamodel abstract syntax metrics

	Source								Target							
	EMF				MSDKVS				MSDKVS				EMF			
	Min	Med	Max	Avg	Min	Med	Max	Avg	Min	Med	Max	Avg	Min	Med	Max	Avg
Grouping	1	2	50	2.83	1	1	1	1	1	1	1	1	1	1	1	1
Classes	1	12	300	33.56	2	8.5	39	10.55	1	13	346	38.92	2	10	47	11.57
Abstract classes	0	2	83	5.77	0	1	9	1.61	0	1	109	6.59	0	1	9	1.61
Inherited classes	0	7	335	31.79	0	3.5	34	5.20	0	7	335	31.79	0	3.5	34	5.20
Multiple inheritances	0	0	134	4.75	–[1]	–	–	–	–	–	–	–	0	0	0	0
Relationships	0	11	437	36.23	0	9	40	10.86	0	12	3750	130.83	0	13	79	18.20
Inherited relationships	–[2]	–	–	–	0	0	4	0.30	0	0	0	0	–	–	–	–
Relationships as class	–[3]	–	–	–	0	0	15	1.18	0	0	0	0	–	–	–	–
Attributes	0	10	98	21.32	1	17.5	170	29.11	0	11	236	28.28	1	17.5	170	30.27
Enumerations	0	0	18	1.36	0	1	21	2.43	0	0	18	1.36	0	1	21	2.43
DataTypes	0	0	60	0.92	0	0	9	1.36	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
[3] EMF does not support attributing relationships and using them as classes

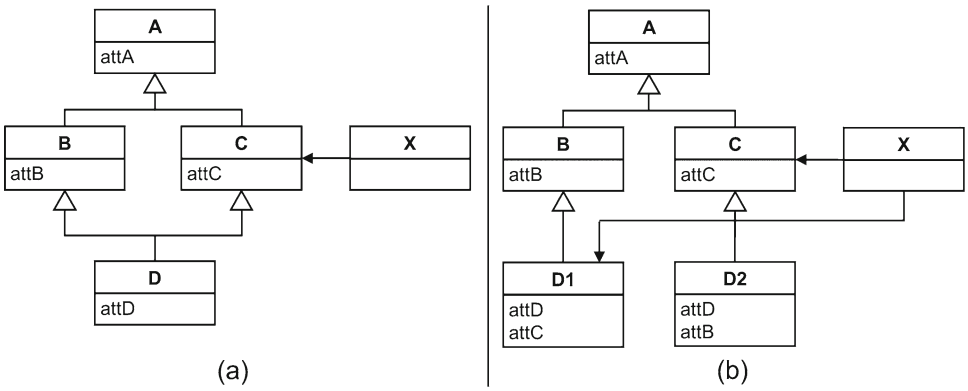


Fig. 5 Adapted *Expansion Strategy* [17]: **a** multiple inheritance in EMF; **b** transformed single inheritance in MSDKVS

formation. Naming conventions for avoiding possible name clashes are transformed accordingly.

Entity Name Clashes. Detecting and resolving name clashes are essential when realizing metamodeling platform interoperability [6]. 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 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. [17] to translate the complex structures of multiple ESuperTypes in EMF into equivalent single BaseClass references in MSDKVS without information loss (see Fig. 5). 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, this affects the transformation of all types of graphical concrete syntax mappings from Sirius as well, which results in more Shape classes on MSDKVS side and Creation Tools inside the modeling editor.

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 to MSDKVS, existing EClasses are matched against these criteria. If such an EClass can be found, this EClass is transformed and acts as the MSDKVS root diagram element. If no EClass

is suitable, then an additional default domain class is generated that acts as the diagram's root element. *Icon mapping.* Sirius supports the definition of icon styles on different node mappings by referencing workspace images in various file formats. For MSDKVS, a requirement to attribute a model entity with icons is that the images have to be in the Bitmap format. Therefore, library calls to convert these files to the required format on the target platform are employed in the M2 transformation.

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 (see Sect. 7), some Ecore metamodels contained attributes, references, 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.* For 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*. Each invalid character is transformed into 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 graphical concrete or abstract syntax element needs an Id property, which contains a generated GUID that uniquely identifies the element. When transforming from EMF to MSDVKS, these GUIDs have to be explicitly generated when the elements are created. Mappings inside the `DslDefinition.dsl` file cross-reference these IDs 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*, two different *DomainRoles*, either used as source or target, must not have the same `PropertyName` when their other end of the relationship is the same. For instance: `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 employed naming strategy updates a counter variable for how often the same target or source base names have been used, applies it to the newly created role, and increments it. In the example above, this results in the relationship ClassA to ClassC having its target domain role renamed to `Target_1`.

Duplicate DomainRelationship names. Like for *DomainRoles*, two *DomainRelationships* must not have the same name. Similar renaming conventions apply, i.e., counting the number of already recognized equal names and adding the current count to the newly created relationship.

6.1.2 MSDKVS2EMF

Relationship roles. DomainRelationships in MSDKVS differ from their required representation on the target EMF side in so far, that the source and the target entities of these 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 already mentioned in [13], this behavior can be implemented similarly, 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.

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.

Implicit modeling tool capabilities. MSDKVS supports only the explicit definition of element creation tools on domain classes and domain relationships, while some tooling capabilities that can be explicitly defined in Sirius are inherently available on MSDKVS' modeling canvas, e.g., copy and paste. To achieve an equivalent experience, the number of tools on EMF is thus typically higher because the M2 trans-

formator generates these additional tools for every Node or Edge Creation Tool defined in MSDKVS.

Color naming. MSDKVS supports a variety 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 the 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 M1 transformation

This section provides explanations of the final tasks of the transformation process, namely the transformation of models. An analysis of how models are represented in each platform is given, and their serialization formats are compared.

6.2.1 Serialization of models

Each platform offers separate executable runtime environments for defining and editing models. Each model is graphically represented by its metamodel's available graphical concrete syntax. If no graphical concrete syntax has been defined, the tree editor of each modeling instance can be used for creating classes and relationships instead. EMF and MSDKVS both serialize the model files in the XML metadata interchange (XMI) format, thus easing the realization of the M1 transformator to some extent.

6.2.2 Transformation approach

Typically, one cannot find concrete examples of models based on defined metamodels contained within model/metamodel zoos. Most or nearly all of the retrieved repositories contain only the metamodel definitions (either within `.ecore` files for EMF or `.dsl` files for MSDKVS) and no concrete models conforming to these definitions are available.

To examine the transformator's validity, a random model generator [1] is being used to generate models as input on the EMF side. Regarding the transformation of MSDKVS models to EMF models, the MSDKVS source models have been created manually to best represent the underlying domain and its features.

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.

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* and *DomainRelationship* element obtains a unique identifier (i.e., GUID) to be referenced from inside other elements. As mentioned in the M2 transformator, these GUIDs 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 Family Tree example shown later in Fig. 7a. 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.

6.2.3 EMF to MSDKVS

The files containing the source metamodel and its model are used in addition to the generated mapping file, providing critical mapping information for each abstract syntax element. Through iterating each *EClass*, and then every *EStructuralFeature*, i.e., *EAttributes* or *EReferences*, contained within these class elements, all elements are transformed by applying corresponding transformation rules. The main challenge in loading the model file into the M1 transformator was that, after investigating the models either randomly generated via the library mentioned in Sect. 8.1 or manually created inside EMF, their root XML tags could differ from each other. As MSDKVS has to have a class defined as the diagram's root class on the M2 layer, 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 as their representation depends on their relationship type.

Containment references are contained within the composite class. Regarding the serialization of containment references into XML format, EMF lists the contained elements inside the containing class elements' tag, whereas in MSDKVS, additional XML tags for the relationship itself, e.g., the relationship name, have to be generated and inserted during the transformation. As mentioned before, simple and bi-directional references use a numbering mechanism to target different 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 correct index selected.

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 transformation. Thus, the `mapping.json` file has to be consulted to find the correct inheritance structure to use for transforming the model entity. Each class mapping contains a superclass mapping, referencing the targeted transformed superclass, filtered by the source's used superclass.

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

The transformation implementation follows a sequential execution as follows. First, the transformation direction is determined based on the first input parameter of the program. Afterward, the submitted input files are de-serialized into corresponding structures (i.e., source Ecore metamodels and models as well as the mapping information). Then, the root element of the Ecore model is extracted and transformed into the equivalently mapped target root element. Afterward, the remaining classes and their attributes as well as their relationships are transformed, whereas relationships targeting not yet transformed class elements are temporarily queued to be transformed once the required classes have been transformed. Lastly, the resulting element tree based on the acquired mapped target elements, where each element contains a list of key-value pairs resembling an XML-like structure, is serialized into a model file, properly readable in the runtime instance of MSDKVS.

6.2.4 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 source model files into code, as no equivalent API is available in Java for interpreting them properly. Thus, a basic XML reader library is used for de-serializing the generic XMI structure of the model into code. Similar to the serialization structure of the transformation result for the direction of EMF to MSDKVS, the models are de-serialized to a tree-based data template containing several key-value pairs resembling their various tags and attributes referencing the metamodel attribute definitions. The following list contains 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 class already transformed are maintained to get their exact index inside the XML file for reference transformations that are not containment references.

Attribute mapping. Attribute elements are added to the transformed class elements as attributes. Important to note here is that the serialization of attributes can be different for each attribute, depending on 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 the 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 de-serialized 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 into an EMF model entity's attribute.

Relationship mapping. Contained classes are wrapped inside their composite class, with additional tags denoting their domain relationship names. Other types 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. Domain relationships, that were transformed into EClasses, are flagged specifically, as the M1 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 (an explanation and evaluation is given in Sect. 8).

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 solving multiple inheritances to multiple single inheritances from EMF to MSDKVS.

The following code sections are executed sequentially to realize the MSDKVS to EMF M1 transformation. First, the Ecore metamodel file, the model file for MSDKVS, and the mapping file generated in the M2 transformation are being loaded into memory. Using the Ecore API, an empty *Resource* is created based on the metamodel content. The resulting *EPackage* element is retrieved, set as the target root element, and registered as a dynamic package in the EPackage Registry Instance. Then, the MSDKVS model is de-serialized and put into a tree of Elements, resulting from the existing XML tags inside the model file. A tag's attributes are resembled by key-value pairs contained within the resulting elements. These XML tag elements are then iterated and mapped, with the help of the mapping information, to corresponding Ecore elements.

After obtaining all elements inside the source model file, the root element is first transformed. As each XML tag is saved as an element inside the resulting tree, the traversal of these elements differs greatly in comparison to the other direction, as the correct type of each element has to be distinguished properly before transforming it. The mapping information containing the names of the source and the target elements helps in finding the correct target element. A separate table is maintained, containing the created EObjects and their moniker key values for every node of the model file. If an element has been traversed, it is flagged to omit duplicate processing. All key-value pairs are then transformed into EAttribute values with the correct EDataType.

The transformation of relationships takes the different serialization types into account, possibly skips elements if the *UseFullForm* method has been used, and the correct referencing of transformed class entities by looking at the moniker key values. Finally, the resulting Ecore data is saved inside a model file using the Ecore API.

6.3 MSDKVS2EMF transformation example

For showcasing the transformation bridge, we will, in the following, refer to a small example of a family tree metamodel that we created in MSDKVS and subsequently transformed, using our transformation bridge, into a valid EMF metamodel. The example, adapted from the tutorial of Sirius,[8] comes also with a graphical concrete syntax specification on MSDKVS' side, which enables to exemplify the feasibility of the transformation bridge. The goal of this example case is thus to illustrate the feasibility of realizing syntactic and semantic equivalence between the two platforms involving the abstract and the graphical concrete syntax. Full images are provided online.[6]

Figure 6a shows the source metamodel inside the Visual Studio IDE. In this example, a basic family tree metamodel with graphical concrete syntax descriptions has been created that contains a compartment relationship between *Country* and *Town* and an inheritance structure between the *Person* domain class as the base class of *Man* and *Woman*. Figure 7a shows a manually created model of an excerpt of the family tree of the British House of Windsor based on the previously defined metamodel in MSDKVS. The result of executing the M2 transformator on the source MSDKVS metamodel is shown in Fig. 6b, showing the resulting Ecore metamodel both graphically and in a tree structure. The resulting EMF model and its entities depicted in Fig. 7b is created using our realized M1 transformator. For better comparability, the EMF model elements have been positioned accordingly on the canvas.

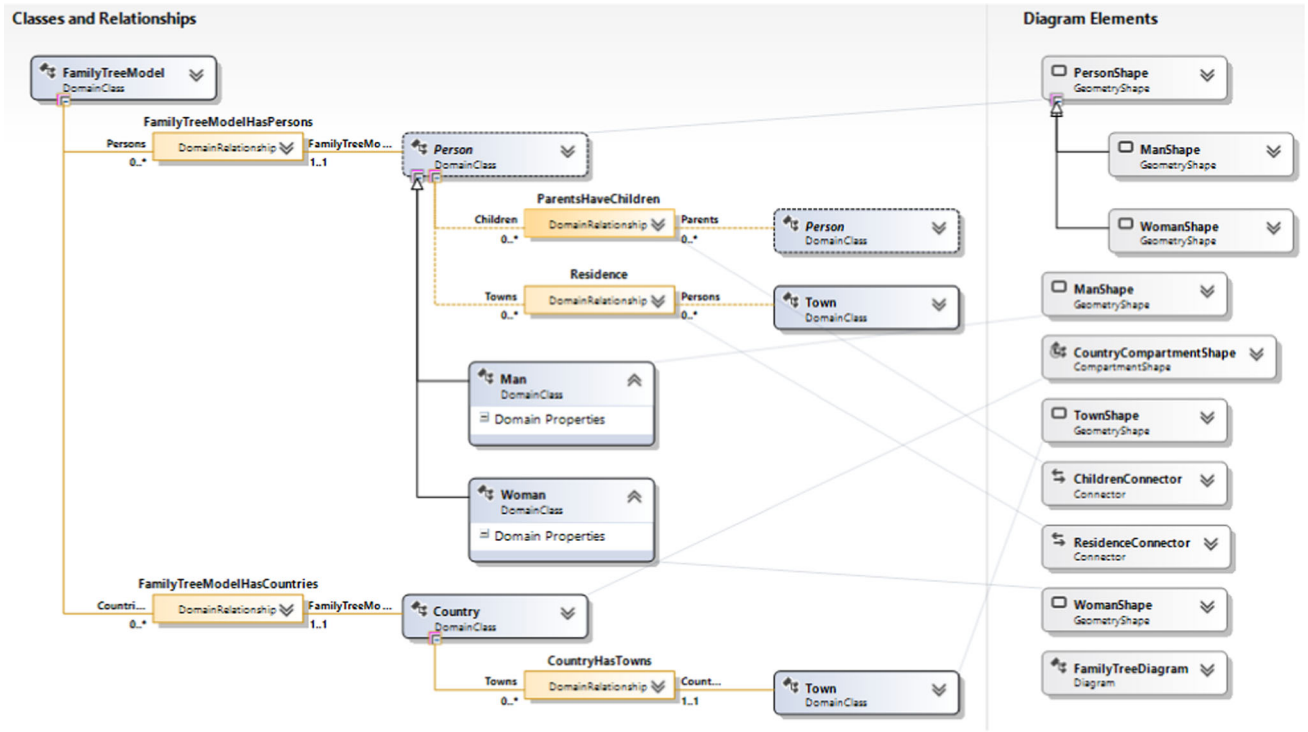
7 M2 evaluation

This section reports on the results of experimenting with the M2 transformator. The transformation is implemented as two uni-directional transformations which means that either MSDKVS metamodels (`*.dsl` files) or EMF metamodels (`*.ecore` files) with optional `.genmodel` and `.odesign` files for graphical concrete syntax mappings and code editor generation settings serve as input.

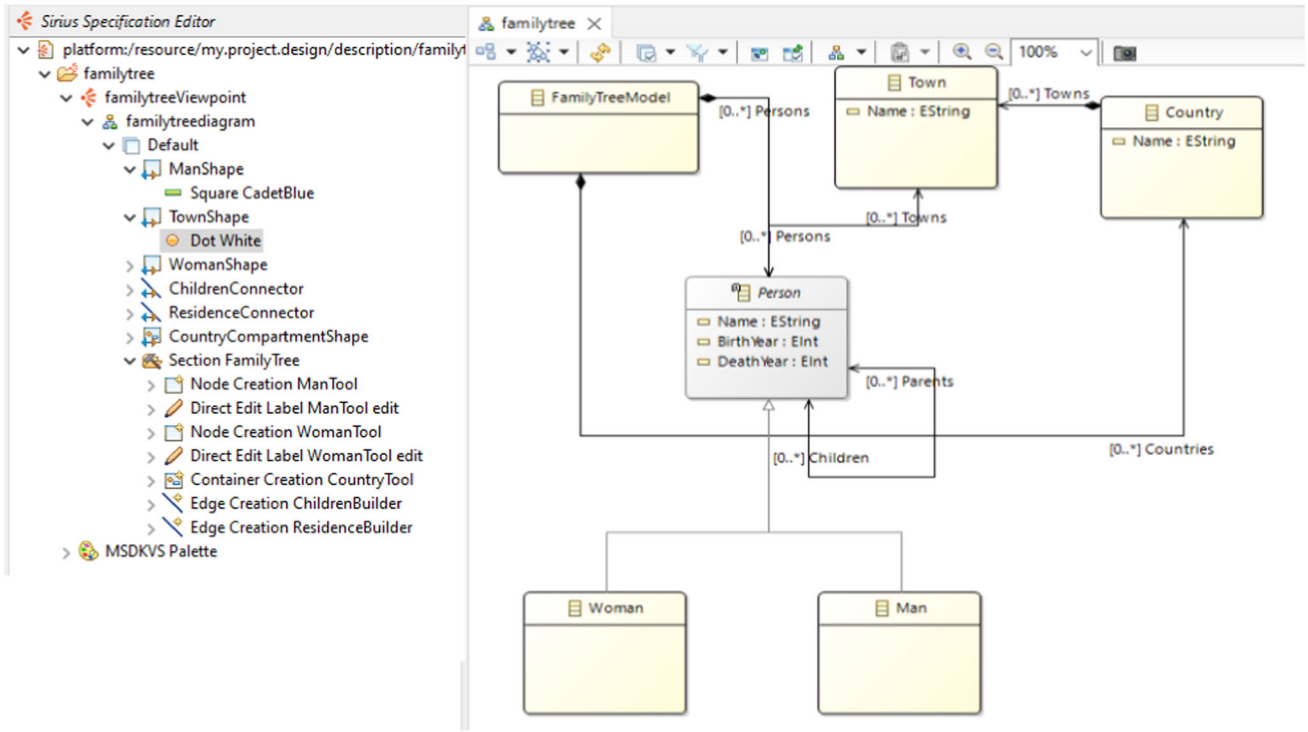
We searched and selected a representative set of metamodels of both platforms from publicly available collections and also through dedicated metamodel search engines [32]. A collection of 44 metamodels from MSDKVS and 75 randomly selected metamodels from the AtlanMod Atlantic Zoo[9] with additional 22 metamodels referencing and containing Sirius VSMs and 18 metamodels (some of which overlapping with VSM available metamodels) containing EMF specific features like multiple inheritance or nested

[8] https://wiki.eclipse.org/Sirius/Tutorials/StarterTutorial.

[9] https://github.com/atlanmod/atlantic-zoo.

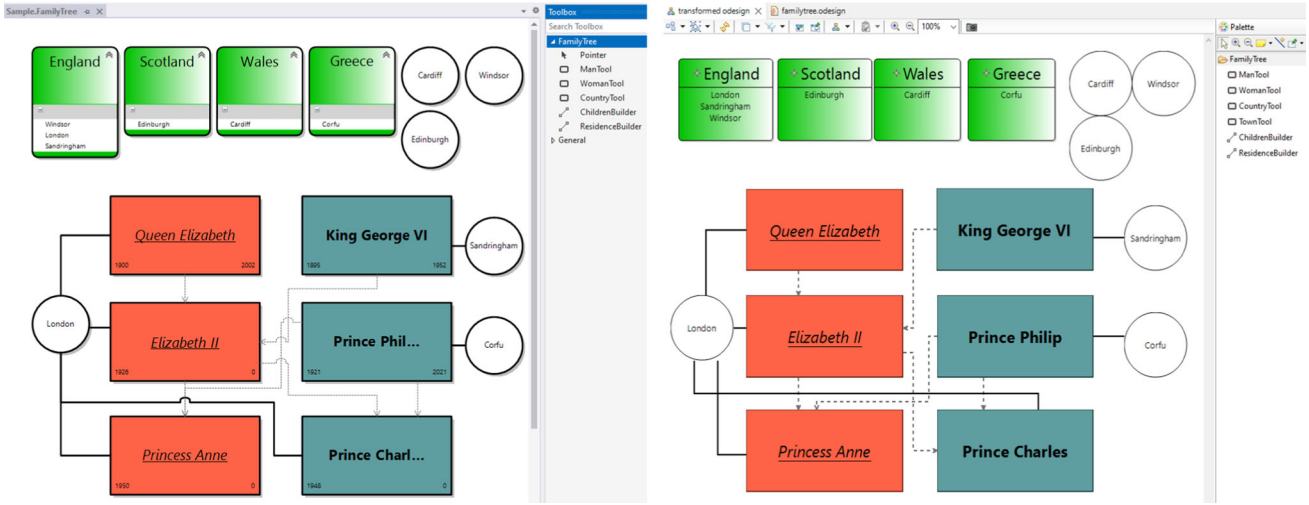


(a) Family Tree metamodel in MSDKVS



(b) Transformed Family Tree VSM (left) and metamodel (right) in EMF

Fig. 6 Family Tree metamodel in MSDKVS and transformed into EMF



(a) A created Family Tree model in MSDKVS (b) Transformed Family Tree model in EMF

Fig. 7 Family Tree model in MSDKVS and transformed into EMF

Table 4 Metamodel concrete syntax metrics

	Source								Target							
	EMF				MSDKVS				MSDKVS				EMF			
	Min	Med	Max	Avg	Min	Med	Max	Avg	Min	Med	Max	Avg	Min	Med	Max	Avg
Class shapes	0	0	12	0.60	0	2	16	3.48	0	0	12	0.60	0	3	23	4.27
Inherited class shapes	–	–	–	–	0	0	10	0.43	0	0	0	0	–	–	–	–
Icon shapes	0	0	18	0.77	0	0	11	0.39	0	0	18	0.77	0	0	11	0.39
Relationship shapes	0	0	39	1.73	0	2	10	3.30	0	0	39	1.73	0	2	17	3.98
Containment shapes	0	0	27	0.79	0	0	5	1.23	0	0	27	0.79	0	0	5	1.20
Tools	0	0	398	9.81	0	6	25	7.84	0	0	14	1.39	0	10	41	13.82

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

EPackages was composed. Thus, the metamodels have been selected mostly at random; all manual additions were motivated by the goal to have a representative set of metamodels (i.e., a set that differs in size, contains metamodeling concepts, and is equipped with a graphical concrete syntax specification).

With the evaluation, we thus aim to respond to the following research questions:

RQ1: Are the transformed metamodels valid when opened in the target platform?, and

RQ2: Are the transformed metamodels executable, i.e. can editor code be generated and runtime instances successfully started?

7.1 Experimental setup

In the following, we dive deeper into the analytical aspects of the metamodel transformation approach. First, quantitative aspects are listed and compared. Table 3 shows statistical information about the experiment's source metamodels' abstract syntax, as well as the metrics of the resulting transformed metamodels in the target platforms. Table 4 depicts the metamodels' concrete graphical syntax metrics, showing both source and target metamodel values. To analyze the qualitative aspects, the transformed metamodels are opened with the target platform. Both platforms offer automatic validation, meaning that when the project files are opened, their internal structure is validated. Validation errors, if present, are listed accordingly. As the 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 the EMF side, the transformed `.odesign` files containing the definitions for graphical concrete syntax mapping required separate, manual validation. An overview of the success rates for both directions is provided in Table 7. If the validation yielded no errors, the platforms' functionalities upon creating models were tested.

7.2 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 was defined. These metamodels were analyzed in-depth regarding their behavior while creating and operating with models inside their generated runtime environments. The goal was to investigate the metamodel functionalities to further strengthen the interoperability aspect of this paper's transformation bridge.

The following aspects, comprising mainly the mapping of meta-metamodel concepts from one platform to the other, were chosen to investigate semantical equivalence between the source and the transformed metamodels:

- *Class mapping:* Correct mapping of class entities, their names, number of attributes, inheritance relationships, and, if applied, possible renaming strategies in the target environment.
- *Relationship mapping:* Multiplicity mapping, source and target class mappings, and the special types of relationships like containment and bi-directional references. This includes, e.g., a correct cascading of delete behavior in containment references.
- *Attribute and enumeration mapping:* Datatype conformity, identical default values, and enumeration literals (taking into account the naming conventions listed in Sect. 6.1.1)
- *Class shape mapping:* Correct coloring and styling of classes and attributes as well as mapped geometries.
- *Relationship shape mapping:* Routing styles, source and target styling, line styles, and attributes must be considered.
- *Tool mapping:* Sectioning of tools, tool icons, and their interaction with the modeling canvas (e.g., creation of relationships between two valid entities, creation of classes).

Subsets of ten metamodels for each direction used in the quantitative and qualitative analysis were 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 [18]. These metamodels then also act as the referencing metamodels for the M1 transformation evaluation done in Sect. 8.

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

1. Transform 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 definition

7.2.1 EMF to MSDKVS

By having selected a subset of metamodels using a variety of different aspects spread among all the available metamodels, this evaluation aims to give an adequate overview of how semantic equivalence regarding model behavior and interaction is achieved.

Figure 8 shows the approach used for selecting the set of metamodels that have to be investigated further for semantic evaluation. Each diagram, having two distinct but connected metamodel metrics as its x and y axes, gives further insight into how distributed the values among the collected source metamodels are. The yellow dots represent the x and y coordinates, i.e., metric values, for the metamodels that have been chosen for semantic evaluation, which will be listed afterward in Table 5. The blue dots visualize the remaining metamodels used inside the feasibility approach in the previous section.

When looking at Table 5, 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 are deemed to be UML2 and c_sharp, as they contain multiple inheritance structures with many supertype definitions and a number of `SubEPackages`. This made evaluating the correct naming conventions for domain roles and domain relationships difficult at first, since also opening the resulting DSLs in Visual Studio resulted in performance issues due to their file sizes and entity counts. 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 number of relation-

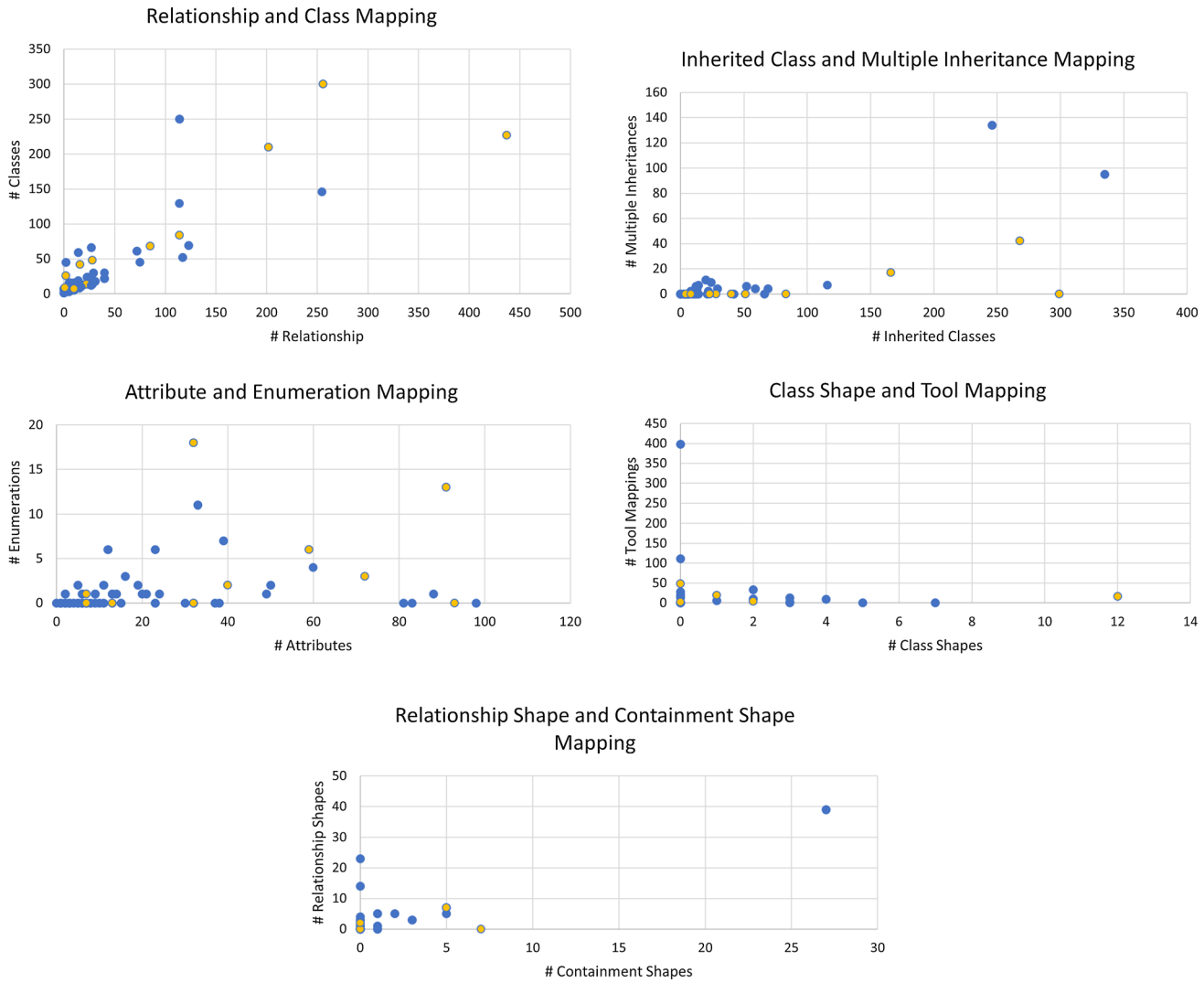


Fig. 8 Metric distribution and highlighted metamodels used for manual evaluation for EMF

Table 5 Semantic evaluation results for transformation direction EMF to MSDKVS

Metamodel	Validity				Tools
	Classes	Relationships	Attributes/Enums	Shapes	
Ant	✓	✓	✓	-	-
ATL	✓	✓	✓	-	-
behaviortree	✓	✓	✓	✓	✓
c_sharp	✓	✓	✓	-	-
HAL	✓	✓	✓	-	-
KDM	✓	✓	✓	-	-
poosl	✓	✓	✓	✓	✓
sensorProject	✓	✓	✓	✓	✓
simplePDL	✓	✓	✓	✓	✓
UML2	✓	✗	✓	-	-

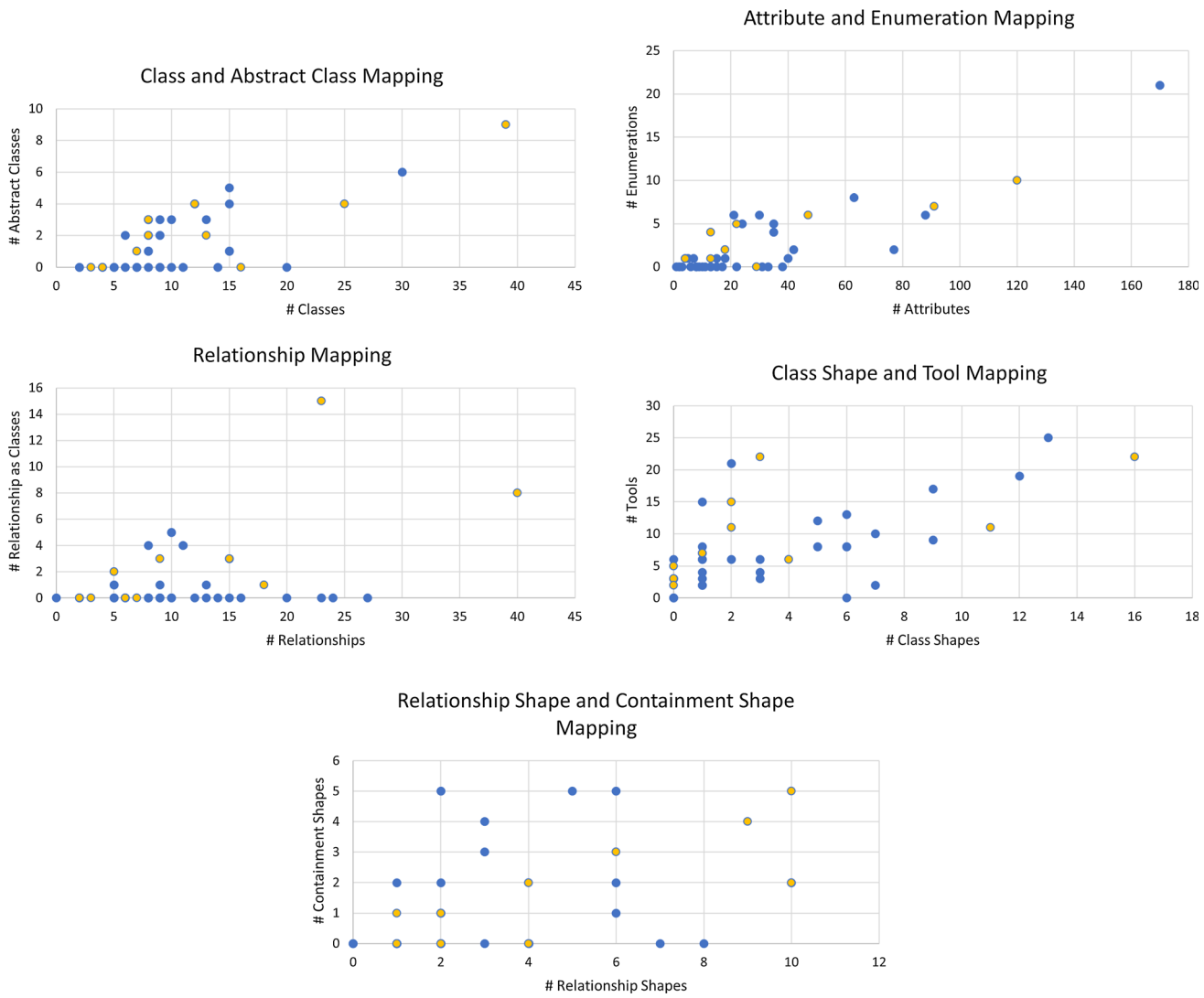


Fig. 9 Metric distribution and highlighted DSLs used for manual evaluation for MSDKVS

ships being rendered on the metamodeling canvas made the interaction and finding the source of the problem impossible, and (2) the number of faulty domain relationships with errors due to the applied nested renaming strategies as the Ecore metamodel contained multiple eSuperType relationships with these supertypes 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 generalizability of the interoperability.

7.2.2 MSDKVS to EMF

The size of MSDKVS metamodels (i.e., the number of domain classes) is usually much smaller compared to EMF, making it easier to appropriately test the transformation. Figure 9 shows five scatter plot diagrams for the source DSLs used to evaluate the M2 transformation, each containing different metric values on their respective x and y axes. The yellow dots symbolize the selected DSLs, identical to the diagrams for the transformation direction EMF to MSDKVS. Table 6 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. cqrsdsl contained 15 class-like relationships and could not be rendered by Sirius in general, as the references were all embedding relationships. For each contained class, a separate model instance had to be created

Table 6 Semantic evaluation results for transformation direction MSDKVS to EMF

DSL	Validity				
	Classes	Relationships	Attributes/Enums	Shapes	Tools
Agilemodeler	✓	✓	✓	✓	✓
Candle	✓	✓	✓	✓	✓
cqrsdsl	✓	✓	✓	✗	✗
fwk_dsl	✓	✓	✓	✓	✓
Generatorlanguage	✓	✓	✓	✓	✓
Hostdesigner	✓	✓	✓	✓	✓
Mbrdcmdmi	✓	✓	✓	✓	✓
Mobiledsl	✓	✓	✓	✓	✓
Nhmodelinglanguage	✓	✓	✓	✓	✓
Spllanguage	✓	✓	✓	✓	✓

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 EdgeMappings cannot visualize containment references. A possible solution would be generating a Sirius representation per such containment structure to visualize them appropriately from separate models.

7.3 Results

RQ1: Validity of transformed metamodels. For the transformation validity (see Table 7) we were not only concerned about the abstract syntax but also about the constraints on the concrete graphical syntax. In the currently implemented version of the transformations, we were able to address and correct all previously mentioned errors [15] thus resulting in a 100% success rate among graphical concrete syntax elements in the direction of MSDKVS to EMF.

EMF → MSDKVS is also yielding a 100% success rate (based on 75 cases) considering the abstract syntax. 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. Future work will investigate how to consider multiple referenced Ecore metamodels during the transformation.

As the M2 transformator is implemented as a bridge between two XML serialization formats, most of the initial problems that occurred originated from de-serializing the input metamodel files to data structures. XML namespace errors were among the most common types of exceptions because of limitations the default C# library provides when de-serializing 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 the de-serialization.

RQ2: Executability of transformed metamodels. The executability assumes a valid metamodel to run the code generators for creating and executing modeling runtime instances in the platforms. When the code generation throws errors or the modeling canvas cannot be initialized, the transformed metamodel files are deemed faulty regarding 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 modeling purposes. As a final step, a small subset of metamodels on each side and their transformed results were selected for manually evaluating and comparing their interaction on the modeling layer, simi-

Table 7 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

lar to how the example in Sect. 6.3 was conducted. Regarding, e.g., their tool palette functionalities, interaction on the modeling canvas, and entity styling, no significant inequalities were observed besides the ones mentioned in the previous section.

7.3.1 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 the new metamodel representations, the XML schema has to be analyzed and corresponding data structures for (de-)serialization have to be added. Existing transformations can afterward be reused, thus creating M3-level bridges spanning multiple metamodel platforms. Another limitation is the realization of the transformator on the current platform versions. Core updates to these platform metamodels, therefore, require similar updates to the transformator.

Regarding the involved metamodel entities and unique features of each platform, some limitations in the transformation's current state are present as well. Many of the advanced model operations supported by Sirius (e.g., filtering using query languages, flow control operations like *Begin*, *For* and *If*, Dialog) cannot be mapped directly to available operations in MSDKVS. As mentioned in Sect. 4, 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*.

8 M1 evaluation

In addition to the feasibility study of the first part of the transformation, this section considers the transformation implementation on the model layer to investigate the complexity and degree of completion, i.e., if and how all possible definitions can be transformed.

8.1 Model generation

For testing the transformation approach on the M1 layer, a random model generator, customizable with several parameters and presented in [1], 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 EMF → MSDKVS. For the opposite direction, the underlying models have been created manually as no model generator is available.

The Ecore model generator can be configured with a variety of optional parameters, like the average size of a model, the average number of references per class element, and attribute value lengths. For this paper's purposes and for better readability of the generated model files, an average size of 20, an average number of references of four per object, and an average variable length of eight for attributes were used to create models that are comprehensible by humans.

8.2 Transformation results

Tables 8 and 9 list the selected transformed metamodels used in the M2 transformator discussed in Sect. 7, the number of manually created or generated models based on these results, and the execution results of the M1 transformator. Each category identified in Sect. 6.2 structures the evaluation. The numbers indicate how many generated models are valid 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. Afterward, the limitations of the M1 transformator based on the *Ecore Reflection API* for EMF → MSDKVS and *JAXB* library for serializing and de-serializing the input models of MSDKVS, used for the direction MSDKVS → EMF, and output models are discussed.

As the semantic evaluation regarding the interoperability of models based on the M2 transformation results has already

Table 8 Model transformation results for transformation direction EMF to MSDKVS

Metamodel	Validity				
	Class	Attribute	Relationship	Inheritance	Layout
Ant	10/10	10/10	10/10	10/10	–
ATL	10/10	10/10	10/10	10/10	–
behaviortree	10/10	10/10	10/10	10/10	10/10
c_sharp	10/10	10/10	0/10	0/10	–
HAL	10/10	10/10	10/10	10/10	–
KDM	10/10	10/10	10/10	10/10	–
poosl	10/10	10/10	10/10	10/10	10/10
sensorProject	10/10	10/10	10/10	10/10	10/10

been done in the previous section, the following results take only the validity of the transformed models into account, especially regarding the validity of the mapping strategies discussed in Sect. 6.2 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 no model generator in MSDKVS is currently available, basic reference links are added inside the model's XMI representation file to check the validity of those.

8.2.1 EMF to MSDKVS

A subset of eight Ecore metamodels was chosen based on the previous quantitative metrics distribution and qualitative analysis. Table 8 shows 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 different model files. These files were then transformed using the M1 transformation.

- *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, 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. Regarding the containment, simple, and bi-directional references, the transformed MSDKVS models proved to be valid in terms of syntax and semantics.
- *Inheritance transformation:* As mentioned before, inheritance relationships regarding nested multiple inheritances were 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.
- *Layout behavior:* Regarding the correct and identical visualization of model content on the modeling canvas inside the platform's run time instances, half of the models had 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.

8.2.2 MSDKVS to EMF

In the scope of this paper's M1 transformator evaluation, models for the selected DSLs have been created manually, best resembling the mentioned parameterized generation of models on the EMF side. As creating different models based on the transformed metamodels is a time-consuming task,

Table 9 Model transformation results for transformation direction MSDKVS to EMF

DSL	Validity							
	Class	Attribute	Relationship	Inheritance	Layout	OE[1]	UFF[2]	ER[3]
Agilemodeler	4/4	4/4	4/4	4/4	4/4	✗	✓	✗
Candle	4/4	4/4	4/4	4/4	4/4	✗	✓	✓
fwk_dsl	4/4	4/4	4/4	4/4	4/4	✗	✓	✗
Generatorlanguage	4/4	4/4	4/4	4/4	4/4	✗	✓	✗
Hostdesigner	4/4	4/4	4/4	4/4	4/4	✗	✗	✗
mbrdcmdmi	4/4	4/4	4/4	4/4	4/4	✗	✓	✗
Mobiledsl	4/4	4/4	4/4	4/4	4/4	✗	✓	✗
Spllanguage	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"

we created four models per DSL to show the feasibility and validity of the M1 transformator.

Table 9 shows the selected DSLs, the number 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.

- *Class transformation:* All class entities to EClass objects transformations have been executed properly.
- *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 de-serialization 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, where 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 domain relationships, no complex structures were present in the source model collection. This resulted in equal single inheritance ESuperType relationships on the target side, simplifying finding the correct type of an EClass, with or without having multiple nested inheritance relationships.
- *Layout behavior:* Most MSDKVS DSLs found contained graphical syntax elements, resulting in Sirius VSMs on the EMF-side after transformation. This enabled us to also validate the correct visualization properties for those DSLs. Regarding the input set of manually created models, 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.
- *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 de-serialization of the model file based on the instructions received from finding the correct reference mapping.
- *ElementReference behavior:* Both MSDKVS styles for serializing domain properties (i.e., as `Attribute` and as `Element`) resulted in valid target models correctly interpreted by the Ecore system.

8.3 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 Sect. 6), or EMF, to read the model file correctly. The generator [1] 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 remaining 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 next limitation relates to the unavailability of an MSDKVS model generator. Although, with the help of the platform'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. This can be added to the list of future work.

9 Conclusion

In the course of this paper, interoperability between the two metamodeling platforms EMF and MSDKVS has been conceptualized, implemented, and evaluated. A bidirectional transformation bridge to transform metamodels and models between both platforms was implemented. The evaluation was conducted by using a representative set of diverse and publicly available metamodels and DSLs[6], and generated or manually created models. By providing mapping rules for the graphical notations available in both metamodeling platforms, we showed that it is possible to transform means of graphically representing elements of transformed metamodels to achieve not only syntactic but also visual interoperability.

Future research aims to further extend the interoperability by looking at the available query languages, like OCL and AQL. Another line of research targets round-trip transformations by, e.g., extending the available mapping information file resulting from the M2 transformation with additional information. Moreover, the bridge itself does offer a wide variety of opportunities, not only from a scientific point of view but also from a more practical one, e.g.:

- Developers can benefit from our concrete syntax mappings when realizing their own bridges. Our approach adds insights into recognizing, abstracting, and grouping concrete syntax structures on top of the known abstract syntax.
- With every new bridge, like the one we presented, the community gains insight into what the most popular structures and features, regarding graphical and abstract syntax, of metamodeling platforms are. This could inform a deep systematic comparison of the available (and future) platforms.
- Developers, having decided to use EMF or MSDKVS, are not locked in anymore. Instead, they can now transform their developed metamodels into initially equivalent artifacts on the respective target platform.

As new metamodel platforms are being developed regularly, seeing the importance of bridging these environments and chaining them together can be beneficial and motivational in terms of integrating possible access points early on in the new metamodeling platforms themselves, e.g., an API that can be easily integrated into existing M3B tools, a standardized exchange format, or a graphical interface for easy usage and extractability. Web-based platforms like GLSP [34] and the development of (web-based) modeling tools are receiving a huge focus in the modeling communities [35]. The paper at hand has the potential to inform the development of these new platforms with the intent to have interoperability with currently established metamodeling platforms in mind.

Stronger toolchains and the possibility to migrate legacy systems developed in old environments that are no longer receiving support from the developers are a welcoming change that eliminates the burden for developing teams using these legacy systems to transport their code bases to new platforms. With the help of developers working on bridging frameworks, the fear of having to scratch existing projects to use newer technologies is reduced.

The bridge's code, as well as the full list of selected metamodels, are available open source, and the transformation bridge has been deployed as a freely usable 'MSDKVS - EMF Converter' service at: http://me.big.tuwien.ac.at/.

Acknowledgements The authors acknowledge TU Wien Bibliothek for financial support through its Open Access Funding Programme.

Funding Open access funding provided by TU Wien (TUW).

Appendix A Abstract syntax mapping table

See Table 10.

Table 10 Comparison of abstract syntax features in MSDKVS and EMF

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)

Appendix B Graphical concrete syntax mapping table

See Tables 11, 12 and 13.

Table 11 Comparison of graphical concrete syntax features in MSDKVS and EMF with regards to their meta-metamodeling concepts

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 12 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

Table 12 continued

Criteria	Ecore (with Sirius)	MSDKVS
Composition creation tool	Container Creation	Element Tool
Filter mechanisms		
Mapping filter	✓	✗
Variable filter	✓	✗

Table 13 Comparison of graphical concrete syntax features in MSDKVS and EMF with regards to their meta-metamodeling concepts continued

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	✗	✓

Appendix C Transformation approaches

See Table 14.

Table 14 List of approaches and implementations regarding the transformation of investigated tools and their metamodeling concepts

Tools	Language	Source available	Executable
ADOxx2EMF [6]	Java	✓[1]	✓
MetaEdit+2EMF [25]	Java (Eclipse Plug-in)	✓[2]	-[8]
Visio2EMF [30]	Java	✓[3]	-[8]
Aris2EMF [26]	ARIS-Script	✗	✗
DSLTools2EMF [4]	Java	✓[4]	✗
Excel2SoftwareQualityControl [3]	Java	✓[5]	✗
SoftwareQualityControl2Mantis [3]	Java	✓[6]	✓
SoftwareQualityControl2Bugzilla [3]	Java	✓[7]	✓
GME2EMF [14]	Java	✗	✗

[1] https://me.big.tuwien.ac.at/adoxxemf/
[2] http://www.informatik.uni-leipzig.de/~kern/metaedit.emf.bridge_1.1.0.jar
[3] http://sourceforge.net/projects/visioemfbridge/
[4] https://www.eclipse.org/atl/atlTransformations/DSL2EMF/DSLBridge.zip
[5] https://www.eclipse.org/atl/atlTransformations/MSOfficeExcel2SoftwareQualityControl/MicrosoftOfficeExcel2SoftwareQualityControl.zip
[6] https://www.eclipse.org/atl/atlTransformations/SoftwareQualityControl2MantisBT/SoftwareQualityControl2MantisBugTracker.zip
[7] https://www.eclipse.org/atl/atlTransformations/SoftwareQualityControl2Bugzilla/SoftwareQualityControl2Bugzilla.zip
[8] Could not be validated due to involvement of proprietary platforms

References

1. Model generator jar for ecore models. https://modeling-languages.com/a-pseudo-random-instance-generator-for-emf-models/. Accessed 15 Mar 2023
2. Official sirius online documentation. https://www.eclipse.org/sirius/doc/. Accessed 15 Mar 2023
3. Bézivin, J., Bruneliere, H., Jouault, F., Kurtev, I.: Model engineering support for tool interoperability. In: Workshop in Software Model Engineering (2005)
4. Bézivin, J., Hillairet, G., Jouault, F., Piers, W., Kurtev, I.: Bridging the MS/DSL tools and the eclipse modeling framework. In: Proceedings of the international workshop on software factories at OOPSLA (2005)
5. Bill, R., Fleck, M., Troya, J., Mayerhofer, T., Wimmer, M.: A local and global tour on momot. Softw. Syst. Model. **18**(2), 1017–1046 (2019). https://doi.org/10.1007/s10270-017-0644-3
6. Bork, D., Anagnostou, K., Wimmer, M.: Towards interoperable metamodeling platforms: the case of bridging adoxx and emf. In: Advanced Information Systems Engineering. 34th International Conference, CAiSE 2022. pp. 479–497. Springer (2022)
7. Bork, D., Karagiannis, D., Pittl, B.: Systematic analysis and evaluation of visual conceptual modeling language notations. In: 12th International Conference on Research Challenges in Information Science. pp. 1–11. IEEE (2018). https://doi.org/10.1109/RCIS.2018.8406652
8. Bork, D., Karagiannis, D., Pittl, B.: A survey of modeling language specification techniques. Inf. Syst. (2020). https://doi.org/10.1016/j.is.2019.101425
9. Bork, D., Langer, P., Ortmayr, T.: A vision for flexible glsp-based web modeling tools. In: Almeida, J.P.A., Kaczmarek-Heß, M., Koschmider, A., Proper, H.A. (eds.) The Practice of Enterprise Modeling: 16th IFIP Working Conference, PoEM 2023, Vienna, Austria, November 28–December 1, 2023, Proceedings. Lecture Notes in Business Information Processing, vol. 497, pp. 109–124. Springer, Berlin (2023). https://doi.org/10.1007/978-3-031-48583-1_7
10. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice, Second Edition. Synthesis Lectures on Software Engineering, Morgan & Claypool (2017)
11. Braun, G., Fillottrani, P.R., Keet, C.M.: A framework for interoperability between models with hybrid tools. J. Intell. Inf. Syst. **60**(2), 437–462 (2022)
12. Braun, G.A., Marinelli, G., Gavagnin, E.R., Cecchi, L.A., Fillottrani, P.R.: Web interoperability for ontology development and support with crowd 2.0. In: Zhou, Z. (ed.) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021. pp. 4980–4983. ijcai.org (2021). https://doi.org/10.24963/ijcai.2021/707
13. Brunelière, H., Cabot, J., Clasen, C., Jouault, F., Bézivin, J.: Towards model driven tool interoperability: bridging eclipse and microsoft modeling tools. In: 6th European Conference on Modelling Foundations and Applications ECMFA. pp. 32–47. Springer (2010)
14. Bézivin, J., Brunette, C., Chevrel, R., Jouault, F., Kurtev, I.: Bridging the generic modeling environment (gme) and the eclipse modeling framework (2005)
15. Cesal, F., Bork, D.: Establishing interoperability between the EMF and the MSDKVS metamodeling platforms. In: Barn, B.S., Sandkuhl, K. (eds.) The Practice of Enterprise Modeling: 15th IFIP WG 8.1 Working Conference, PoEM 2022, London, UK, November 23-25, 2022, Proceedings. Lecture Notes in Business Information Processing, vol. 456, pp. 167–182. Springer, Berlin (2022). https://doi.org/10.1007/978-3-031-21488-2_11
16. Cook, S., Jones, G., Kent, S., Wills, A.: Domain-specific development with visual studio dsl tools (2007)
17. Crespo, Y., Marqués, J., Rodríguez, J.: On the translation of multiple inheritance hierarchies into single inheritance hierarchies. pp. 30–37 (2002)
18. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Mining metrics for understanding metamodel characteristics. In: Modeling in software engineering. ACM (2014)
19. Fillottrani, P.R., Keet, C.M.: Conceptual model interoperability: A metamodel-driven approach. In: Bikakis, A., Fodor, P., Roman, D. (eds.) 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. Lecture Notes in Computer Science, vol. 8620, pp. 52–66. Springer, Berlin (2014). https://doi.org/10.1007/978-3-319-09870-8_4
20. Geraci, A., Katki, F., McMonegal, L., Meyer, B., Lane, J., Wilson, P., Radatz, J., Yee, M., Porteous, H., Springsteel, F.: IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries. IEEE Press (1991)
21. Group, O.M.: Omg meta object facility (mof) core specification (2019). https://www.omg.org/spec/MOF/2.5.1/PDF. Accessed 04 Sept 2023
22. Hebig, R., Seidl, C., Berger, T., Pedersen, J.K., Wasowski, A.: 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. pp. 445–455. ACM (2018)
23. Jouault, F., Bézivin, J.: KM3: a DSL for metamodel specification. In: Gorrieri, R., Wehrheim, H. (eds.) 8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, pp. 171–185. Springer, Berlin (2006)
24. Karsai, G., Sztipanovits, J., Lédeczi, Á., Bapty, T.: Model-integrated development of embedded software. Proc. IEEE **91**(1), 145–164 (2003). https://doi.org/10.1109/JPROC.2002.805824
25. Kern, H.: The interchange of (meta)models between metaedit+ and eclipse emf using m3-level-based bridges. In: 8th workshop on domain-specific modeling, pp. 14–19 (2008)
26. Kern, H.: Modellaustausch zwischen ARIS und Eclipse EMF durch Verwendung einer M3-Level-basierten Brücke, pp. 123–137 (2008)
27. Kern, H.: Study of interoperability between meta-modeling tools. In: Ganzha, M., Maciaszek, L.A., Paprzycki, M. (eds.) Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, Poland, September 7–10, 2014. Annals of Computer Science and Information Systems, vol. 2, pp. 1629–1637 (2014). https://doi.org/10.15439/2014F255
28. Kern, H.: Model Interoperability Between Meta-Modeling Environments by Using m3-Level-Based Bridges. Ph.D. thesis, Leipzig University, Germany (2016)
29. Kern, H., Hummel, A., Kühne, S.: Towards a comparative analysis of meta-metamodels. In: Lopes, C.V. (ed.) SPLASH'11 Workshops. pp. 7–12. ACM (2011)
30. Kern, H., Kühne, S.: Integration of microsoft visio and eclipse modeling framework using m3-level-based bridges. In: Workshop on Model-Driven Tool & Process Integration (2009)
31. Kühne, T.: Matters of (meta-)modeling. Softw. Syst. Model. **5**(4), 369–385 (2006)
32. López, J.A.H., Cuadrado, J.S.: MAR: a structure-based search engine for models. In: 23rd International Conference on Model Driven Engineering Languages and Systems. pp. 57–67. ACM (2020)
33. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theoret. Comput. Sci. **152**, 125–142 (2006)
34. Metin, H., Bork, D.: On developing and operating glsp-based web modeling tools: lessons learned from bigUML. In: Proceedings of the 26th International Conference on Model Driven Engineering Languages and Systems, MODELS 2023. IEEE (2023)
35. Michael, J., Bork, D., Wimmer, M., Mayr, H.C.: Quo vadis modeling? findings of a community survey, an ad-hoc bibliometric analysis, and expert interviews on data, process, and software modeling. Softw. Syst. Model. (2024). https://doi.org/10.1007/s10270-023-01128-y
36. 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. Accessed 15 Mar 2023
37. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional (2009)
38. Ternes, B., Rosenthal, K., Strecker, S.: User interface design research for modeling tools A literature study. Enterp. Model. Inf. Syst. Archit. Int. J. Concept. Model. **16**, 4:1-4:30 (2021). https://doi.org/10.18417/EMISA.16.4
39. Viyović, V., Maksimović, M., Perisić, B.: Sirius: A rapid development of dsm graphical editor. In: Intelligent Engineering Systems INES 2014. pp. 233–238. IEEE (2014)
40. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Bruel, J. (ed.) Satellite Events at the MoDELS 2005 Conference, pp. 159–168. Springer (2005)

Florian Cesal studied Software Engineering in Hagenberg (Bachelor's degree) and TU Wien (Master's degree), where he finished his thesis and his curriculum in 2023 while also working full-time in a company developing software architectures for point of sale (POS) systems in the DACH region.

Dominik Bork is an Assistant Professor of Business Systems Engineering at the Faculty of Informatics, Institute of Information Systems Engineering, Business Informatics Group at TU Wien. His research interests comprise conceptual modeling, model -driven engineering, and modeling tool development. A primary focus of ongoing research is on the mutual benefits of conceptual modeling and artificial intelligence. For more information, you can contact the author at dominik.bork@tuwien.ac.at or visit https://www.model-engineering.info/.